

UNIVERSIDAD DE OVIEDO



ESCUELA UNIVERSITARIA DE INGENIERÍA TÉCNICA EN INFORMÁTICA DE OVIEDO

PROYECTO FIN DE CARRERA

GUÍA DE CONSTRUCCIÓN DE SOFTWARE EN JAVA CON PATRONES DE DISEÑO

DIRECTOR:

JUAN MANUEL CUEVA LOVELLE

AUTOR:

FRANCISCO JAVIER MARTÍNEZ JUAN

VºBº del Director del Proyecto

GUÍA DE CONSTRUCCIÓN DE SOFTWARE EN JAVA CON PATRONES DE DISEÑO

RESUMEN

Se trata de una aplicación, que de un modo didáctico, sirva para la comprensión y utilización de patrones de diseño de forma clara y correcta en la construcción de software.

El proyecto está dividido en dos partes fundamentales:

- Una introducción teórica a los patrones de diseño.
- Ejemplos de aplicaciones utilizando patrones de diseño.

El desarrollo del proyecto consiste en la utilización de patrones de diseño en la realización de programas, con un fin didáctico, explicando detalladamente los patrones utilizados, así como la implementación de dichos ejemplos en uno de los últimos lenguajes de programación que han aparecido y que mayor futuro tienen, debido a sus numerosas ventajas, el lenguaje Java.

PALABRAS CLAVE

Guía, Patrones de diseño, Lenguaje Java, Tecnología Orientada a Objetos, Notación UML, Software, Singleton, Iterator, Strategy, Observer, Prototype, Composite, Decorator, Factory Method, State, Template Method, Command, Adapter, Mediator, Null Object, JHotDraw.

GUIDE TO SOFTWARE CONSTRUCTION USING DESIGN PATTERNS IN JAVA

ABSTRACT

This is an application with a didactic purpose: be useful to the understanding and the use of design patterns -in a right and fair way- in the building of software.

The project has two parts:

- An theoretic introduction to design patterns.
- Examples of applications using design patterns.

The development of the project consists in the use of design patterns in the making of programs with a didactic purpose, explaining in detail the patterns used as well as the implementation of that examples in one of the latest appeared and best-futured (due to its numerous advantages) programming languages: Java language.

KEY WORDS

Guide, Design patterns, Java language, Object-Oriented technology, UML notation, Software, Singleton, Iterator, Strategy, Observer, Prototype, Composite, Decorator, Factory Method, State, Template Method, Command, Adapter, Mediator, Null Object, JHotDraw.

TABLA DE CONTENIDOS

1. INTRODUCCIÓN.....	1
1.1. JUSTIFICACIÓN.....	1
1.2. INTRODUCCIÓN A LOS PATRONES.....	2
1.2.1. <i>Desarrollo histórico</i>	4
1.2.2. <i>Patrones de software</i>	5
1.2.2.1. Definiciones.....	5
1.2.2.2. Características de los patrones software.....	6
1.2.2.3. Clases de patrones software.....	7
1.3. PATRONES DE DISEÑO.....	8
1.3.1. <i>¿Qué es un patrón de diseño?</i>	9
1.3.2. <i>Descripción de patrones de diseño</i>	10
1.3.3. <i>Cualidades de un patrón de diseño</i>	12
1.3.4. <i>Clasificación de los patrones de diseño</i>	13
1.3.5. <i>El catálogo de patrones de diseño</i>	15
1.3.6. <i>Patrones de diseño, reglas y creatividad</i>	23
1.3.7. <i>Patrones y algoritmos</i>	24
1.3.8. <i>Patrones y frameworks</i>	25
1.3.9. <i>Donde usar los patrones</i>	26
1.3.10. <i>Seleccionar un patrón de diseño</i>	32
1.3.11. <i>Como usar un patrón de diseño</i>	33
1.3.12. <i>El futuro de los patrones</i>	34
1.4. ANTIPATRONES.....	34
1.5. CONCLUSIÓN.....	36
2. EXPLICACIÓN DETALLADA DE LOS PATRONES DE DISEÑO UTILIZADOS EN LOS EJEMPLOS.....	37
2.1. SINGLETON [GoF95].....	37
2.1.1. <i>Objetivo</i>	37
2.1.2. <i>Aplicabilidad</i>	37
2.1.3. <i>Solución</i>	37
2.1.4. <i>Consecuencias</i>	38
2.1.5. <i>Implementación</i>	38
2.1.6. <i>Usos en el API de Java</i>	38
2.1.7. <i>Patrones relacionados</i>	39
2.2. ITERATOR [GoF95].....	41
2.2.1. <i>Objetivo</i>	41
2.2.2. <i>Aplicabilidad</i>	41
2.2.3. <i>Solución</i>	41
2.2.4. <i>Consecuencias</i>	42
2.2.5. <i>Implementación</i>	42
2.2.6. <i>Usos en el API de Java</i>	43
2.2.7. <i>Patrones relacionados</i>	43
2.3. STRATEGY [GoF95].....	45
2.3.1. <i>Objetivo</i>	45
2.3.2. <i>Aplicabilidad</i>	45
2.3.3. <i>Solución</i>	45
2.3.4. <i>Consecuencias</i>	46
2.3.5. <i>Implementación</i>	46
2.3.6. <i>Usos en el API de Java</i>	47
2.3.7. <i>Patrones relacionados</i>	47

2.4. OBSERVER [GoF95].....	49
2.4.1. <i>Objetivo</i>	49
2.4.2. <i>Aplicabilidad</i>	49
2.4.3. <i>Solución</i>	49
2.4.4. <i>Consecuencias</i>	51
2.4.5. <i>Implementación</i>	52
2.4.6. <i>Usos en el API de Java</i>	53
2.4.7. <i>Patrones relacionados</i>	54
2.5. PROTOTYPE [GoF95].....	55
2.5.1. <i>Objetivo</i>	55
2.5.2. <i>Aplicabilidad</i>	55
2.5.3. <i>Solución</i>	56
2.5.4. <i>Consecuencias</i>	56
2.5.5. <i>Implementación</i>	57
2.5.6. <i>Usos en el API de Java</i>	58
2.5.7. <i>Patrones relacionados</i>	58
2.6. COMPOSITE [GoF95].....	59
2.6.1. <i>Objetivo</i>	59
2.6.2. <i>Aplicabilidad</i>	59
2.6.3. <i>Solución</i>	59
2.6.4. <i>Consecuencias</i>	61
2.6.5. <i>Implementación</i>	61
2.6.6. <i>Usos en el API de Java</i>	62
2.6.7. <i>Patrones relacionados</i>	62
2.7. DECORATOR [GoF95].....	65
2.7.1. <i>Objetivo</i>	65
2.7.2. <i>Aplicabilidad</i>	65
2.7.3. <i>Solución</i>	65
2.7.4. <i>Consecuencias</i>	66
2.7.5. <i>Implementación</i>	67
2.7.6. <i>Patrones relacionados</i>	67
2.8. FACTORY METHOD [GoF95].....	69
2.8.1. <i>Objetivo</i>	69
2.8.2. <i>Aplicabilidad</i>	69
2.8.3. <i>Solución</i>	69
2.8.4. <i>Consecuencias</i>	71
2.8.5. <i>Implementación</i>	71
2.8.6. <i>Usos en el API de Java</i>	72
2.8.7. <i>Patrones relacionados</i>	72
2.9. STATE [GoF95].....	75
2.9.1. <i>Objetivo</i>	75
2.9.2. <i>Aplicabilidad</i>	75
2.9.3. <i>Solución</i>	75
2.9.4. <i>Consecuencias</i>	76
2.9.5. <i>Implementación</i>	77
2.9.6. <i>Patrones relacionados</i>	77
2.10. TEMPLATE METHOD [GoF95]	79
2.10.1. <i>Objetivo</i>	79
2.10.2. <i>Aplicabilidad</i>	79
2.10.3. <i>Solución</i>	79
2.10.4. <i>Consecuencias</i>	80
2.10.5. <i>Implementación</i>	81
2.10.6. <i>Patrones relacionados</i>	81
2.11. COMMAND [GoF95]	83
2.11.1. <i>Objetivo</i>	83
2.11.2. <i>Aplicabilidad</i>	83
2.11.3. <i>Solución</i>	83
2.11.4. <i>Consecuencias</i>	84

2.11.5. Implementación.....	85
2.11.6. Usos en el API de Java.....	86
2.11.7. Patrones relacionados.....	86
2.12. ADAPTER [GoF95].....	89
2.12.1. Objetivo.....	89
2.12.2. Aplicabilidad.....	89
2.12.3. Solución.....	89
2.12.4. Consecuencias.....	90
2.12.5. Implementación.....	90
2.12.6. Usos en el API de Java.....	91
2.12.7. Patrones relacionados.....	93
2.13. MEDIATOR [GoF95].....	95
2.13.1. Objetivo.....	95
2.13.2. Aplicabilidad.....	95
2.13.3. Solución.....	95
2.13.4. Consecuencias.....	97
2.13.5. Implementación.....	98
2.13.6. Usos en el API de Java.....	99
2.13.7. Patrones relacionados.....	99
2.14. NULL OBJECT [WOOLF97].....	101
2.14.1. Objetivo.....	101
2.14.2. Aplicabilidad.....	101
2.14.3. Solución.....	101
2.14.4. Consecuencias.....	102
2.14.5. Implementación.....	102
2.14.6. Patrones relacionados.....	103
3. EJEMPLOS DE APLICACIONES UTILIZANDO PATRONES DE DISEÑO.....	105
3.1. APLET PARA ESCUCHAR AUDIO.....	107
3.1.1. ¿En qué consiste la aplicación?.....	107
3.1.2. Patrones de diseño utilizados.....	108
3.1.3. Diagrama de clases.....	108
3.1.4. Código en Java.....	109
3.1.5. Funcionamiento de la aplicación.....	112
3.2. LISTADOS DE FORMA ASCENDENTE Y DESCENDENTE DE UNA LISTA.....	115
3.2.1. ¿En qué consiste la aplicación?.....	115
3.2.2. Patrones de diseño utilizados.....	116
3.2.3. Diagrama de clases.....	116
3.2.4. Código en Java.....	117
3.2.5. Funcionamiento de la aplicación.....	120
3.3. ORDENACIÓN DE UN VECTOR DE FORMA ASCENDENTE Y DESCENDENTE.....	121
3.3.1. ¿En qué consiste la aplicación?.....	121
3.3.2. Patrones de diseño utilizados.....	122
3.3.3. Diagrama de clases.....	123
3.3.4. Código en Java.....	123
3.3.5. Funcionamiento de la aplicación.....	130
3.4. EJEMPLO VISUAL DEL PATRÓN OBSERVER.....	133
3.4.1. ¿En qué consiste la aplicación?.....	133
3.4.2. Patrones de diseño utilizados.....	134
3.4.3. Diagrama de clases.....	136
3.4.4. Código en Java.....	137
3.4.5. Funcionamiento de la aplicación.....	144
3.5. JHOTDRAW 5.1.....	145
3.5.1. ¿En qué consiste la aplicación?.....	145
3.5.2. Diagrama de clases general.....	148
3.5.3. Patrones de diseño utilizados y código en Java.....	150
3.5.3.1. Abstracciones fundamentales.....	151
3.5.3.2. Herramientas.....	203

3.5.3.3. Actualización de pantalla	216
3.5.3.4. Manipulación de figuras	238
3.5.3.5. Conexión de figuras	265
3.5.3.6. Entorno de ejecución	283
3.5.3.7. Más patrones de diseño	308
3.5.3.8. Conclusiones	337
3.5.4. <i>Funcionamiento de la aplicación</i>	338
3.5.4.1. JavaDrawApp.....	338
3.5.4.2. JavaDrawApplet	340
3.5.4.3. JavaDrawViewer.....	341
3.5.4.4. PertApp	342
3.5.4.5. PertApplet.....	345
3.5.4.6. NothingApp	345
3.5.4.7. NothingApplet.....	348
3.5.4.8. NetApp.....	349
4. CONCLUSIONES FINALES	351
5. BIBLIOGRAFÍA.....	353
ANEXO A: CÓDIGO COMPLETO EN JAVA DEL APARTADO 3.1.4	355
CLASE MANEJADORAUDIO.....	355
CLASE AUDIOPLAYER.....	357
PÁGINA WEB AUDIOPLAYER.....	360
ANEXO B: CÓDIGO COMPLETO EN JAVA DEL APARTADO 3.2.4	361
CLASE COLECCION.....	361
CLASE LISTAITERATOR	362
CLASE DIALOGABOUT	365
ANEXO C: CÓDIGO COMPLETO EN JAVA DEL APARTADO 3.3.4	369
INTERFACE ORDENACION	369
CLASE ALGBURBUJAASC	369
CLASE ALGBURBUJADDESC	370
CLASE ALGORITMOS.....	371
CLASE COLECCION.....	372
CLASE ORDENACIONSTRATEGY	374
CLASE DIALOGABOUT	378
ANEXO D: CÓDIGO COMPLETO EN JAVA DEL APARTADO 3.4.4	381
CLASE VALOROBSERVABLE.....	381
CLASE TEXTOOBSERVADOR.....	383
CLASE BARRAOBSERVADOR.....	385
CLASE PATRONOBSERVER.....	388
CLASE DIALOGABOUT	390
ANEXO E: CÓDIGO COMPLETO EN JAVA DEL JHOTDRAW 5.1	393
PAQUETE CH.IFA.DRAW.UTIL	395
<i>Interface Animatable</i>	395
<i>Interface PaletteListener</i>	396
<i>Interface Storable</i>	396
<i>Clase Clipboard</i>	397
<i>Clase ColorMap</i>	398
<i>Clase Command</i>	400
<i>Clase CommandButton</i>	401
<i>Clase CommandChoice</i>	402
<i>Clase CommandMenu</i>	403
<i>Clase Filler</i>	405

<i>Class FloatingTextField</i>	406
<i>Class Geom</i>	408
<i>Class IconKit</i>	412
<i>Class PaletteButton</i>	415
<i>Class PaletteIcon</i>	418
<i>Class PaletteLayout</i>	418
<i>Class ReverseVectorEnumerator</i>	420
<i>Class StorableInput</i>	421
<i>Class StorableOutput</i>	424
PAQUETE CH.IFA.DRAW.FRAMEWORK.....	429
<i>Interface ConnectionFigure</i>	429
<i>Interface Connector</i>	432
<i>Interface Drawing</i>	434
<i>Interface DrawingChangeListener</i>	438
<i>Interface DrawingEditor</i>	438
<i>Interface DrawingView</i>	440
<i>Interface Figure</i>	445
<i>Interface FigureChangeListener</i>	449
<i>Interface FigureEnumeration</i>	450
<i>Interface Handle</i>	451
<i>Interface Locator</i>	453
<i>Interface Painter</i>	454
<i>Interface PointConstrainer</i>	455
<i>Interface Tool</i>	456
<i>Class DrawingChangeEvent</i>	458
<i>Class FigureChangeEvent</i>	458
<i>Class FigureSelection</i>	459
<i>Error HJDError</i>	461
PAQUETE CH.IFA.DRAW.STANDARD.....	463
<i>Interface TextHolder</i>	464
<i>Class AbstractConnector</i>	465
<i>Class AbstractFigure</i>	467
<i>Class AbstractHandle</i>	474
<i>Class AbstractLocator</i>	477
<i>Class AbstractTool</i>	478
<i>Class ActionTool</i>	480
<i>Class AlignCommand</i>	481
<i>Class BoxHandleKit</i>	483
<i>Class BringToFrontCommand</i>	486
<i>Class BufferedUpdateStrategy</i>	487
<i>Class ChangeAttributeCommand</i>	488
<i>Class ChangeConnectionEndHandle</i>	489
<i>Class ChangeConnectionHandle</i>	491
<i>Class ChangeConnectionStartHandle</i>	494
<i>Class ChopBoxConnector</i>	495
<i>Class CompositeFigure</i>	496
<i>Class ConnectionHandle</i>	504
<i>Class ConnectionTool</i>	507
<i>Class CopyCommand</i>	512
<i>Class CreationTool</i>	513
<i>Class CutCommand</i>	515
<i>Class DecoratorFigure</i>	516
<i>Class DeleteCommand</i>	521
<i>Class DragTracker</i>	522
<i>Class DuplicateCommand</i>	523
<i>Class FigureChangeEventMulticaster</i>	524
<i>Class FigureEnumerator</i>	526
<i>Class FigureTransferCommand</i>	527

Class <i>GridConstrainer</i>	528
Class <i>HandleTracker</i>	529
Class <i>LocatorConnector</i>	530
Class <i>LocatorHandle</i>	532
Class <i>NullHandle</i>	533
Class <i>OffsetLocator</i>	534
Class <i>PasteCommand</i>	536
Class <i>RelativeLocator</i>	537
Class <i>ReverseFigureEnumerator</i>	539
Class <i>SelectAreaTracker</i>	540
Class <i>SelectionTool</i>	542
Class <i>SendToBackCommand</i>	544
Class <i>SimpleUpdateStrategy</i>	545
Class <i>StandardDrawing</i>	546
Class <i>StandardDrawingView</i>	549
Class <i>ToggleGridCommand</i>	562
Class <i>ToolButton</i>	563
PAQUETE CH.IFA.DRAW.FIGURES	565
Interface <i>LineDecoration</i>	565
Class <i>ArrowTip</i>	566
Class <i>AttributeFigure</i>	568
Class <i>BorderDecorator</i>	571
Class <i>BorderTool</i>	573
Class <i>ChopEllipseConnector</i>	574
Class <i>ConnectedTextTool</i>	574
Class <i>ElbowConnection</i>	576
Class <i>ElbowHandle</i>	577
Class <i>EllipseFigure</i>	580
Class <i>FigureAttributes</i>	582
Class <i>FontSizeHandle</i>	585
Class <i>GroupCommand</i>	586
Class <i>GroupFigure</i>	587
Class <i>GroupHandle</i>	588
Class <i>ImageFigure</i>	589
Class <i>InsertImageCommand</i>	592
Class <i>LineConnection</i>	593
Class <i>LineFigure</i>	598
Class <i>NumberTextFigure</i>	600
Class <i>PolyLineConnector</i>	601
Class <i>PolyLineFigure</i>	603
Class <i>PolyLineHandle</i>	608
Class <i>PolyLineLocator</i>	609
Class <i>RadiusHandle</i>	610
Class <i>RectangleFigure</i>	611
Class <i>RoundRectangleFigure</i>	613
Class <i>ScribbleTool</i>	615
Class <i>ShortestDistanceConnector</i>	617
Class <i>TextFigure</i>	620
Class <i>TextTool</i>	626
Class <i>UngroupCommand</i>	629
PAQUETE CH.IFA.DRAW.CONTRIB.....	631
Class <i>ChopPolygonConnector</i>	631
Class <i>DiamondFigure</i>	632
Class <i>PolygonFigure</i>	633
Class <i>PolygonHandle</i>	642
Class <i>PolygonScaleHandle</i>	643
Class <i>PolygonTool</i>	645
Class <i>TriangleFigure</i>	647

<i>Clase TriangleRotationHandle</i>	650
PAQUETE CH.IFA.DRAW.APPLET	653
<i>Clase DrawApplet</i>	653
PAQUETE CH.IFA.DRAW.APPLICATION.....	665
<i>Clase DrawApplication</i>	665
PAQUETE CH.IFA.DRAW.SAMPLES.JAVADRAW	681
<i>Clase AnimationDecorator</i>	681
<i>Clase Animator</i>	683
<i>Clase BouncingDrawing</i>	684
<i>Clase FollowURLTool</i>	685
<i>Clase JavaDrawApp</i>	686
<i>Clase JavaDrawApplet</i>	690
<i>Clase JavaDrawViewer</i>	692
<i>Clase MySelectionTool</i>	694
<i>Clase PatternPainter</i>	695
<i>Clase URLTool</i>	696
PAQUETE CH.IFA.DRAW.SAMPLES.PERT	699
<i>Clase PertApplet</i>	699
<i>Clase PertApplication</i>	700
<i>Clase PertDependency</i>	701
<i>Clase PertFigure</i>	702
<i>Clase PertFigureCreationTool</i>	708
PAQUETE CH.IFA.DRAW.SAMPLES.NOTHING.....	709
<i>Clase NothingApp</i>	709
<i>Clase NothingApplet</i>	710
PAQUETE CH.IFA.DRAW.SAMPLES.NET	713
<i>Clase NetApp</i>	713
<i>Clase NodeFigure</i>	714

1. INTRODUCCIÓN

1.1. JUSTIFICACIÓN

El desarrollo de software es una tarea complicada, la cual depende en gran medida de la experiencia de las personas involucradas, en particular de los desarrolladores.

1. El 80% de los aportes vienen del 20% del personal.

La comprensión del software es uno de los problemas más complicados en la tarea de mantenimiento y evolución.

El hombre durante su historia ha dominado cierta técnica pasando por un proceso:

- Se realizan las operaciones de una manera artesanal. Los *expertos* aprenden por un proceso de ensayo y error y por transmisión de otros *expertos*.
- Se crea una ciencia alrededor de la tarea.
- Se desarrollan las técnicas generalmente aceptadas en el área.
- Se logra un conocimiento común sobre cómo aplicar las técnicas. Hay un *metalenguaje* común ente los *expertos*.

La sociedad requiere sistemas más complejos y más grandes. Los recursos para desarrollarlos cada vez son más escasos. Debe existir un mecanismo de reutilización.

2. El 80% del esfuerzo esta en el 20% del código desarrollado.

Las Tecnologías Orientadas a Objetos son las más utilizadas en los últimos años para el desarrollo de aplicaciones software. Se ha comprobado como este paradigma de programación presenta muchas ventajas.

Uno de los objetivos que se buscan al utilizar esta técnica es conseguir la reutilización. Entre los beneficios que se consiguen con la reutilización están:

1. Reducción de tiempos.
2. Disminución del esfuerzo de mantenimiento.
3. Eficiencia.
4. Consistencia.
5. Fiabilidad.
6. Protección de la inversión en desarrollos.

Este mecanismo de reutilización, según el modelo de Objetos, debe afectar tanto al personal (con una formación constante a lo largo del tiempo) como a los diseños y especificaciones así como al código fuente (mediante herencia, genericidad,...).

Entre los diferentes mecanismos de reutilización están:

- **Componentes:** elemento de software suficientemente pequeño para crearse y mantenerse pero suficientemente grande para poder utilizarse.
- **Frameworks:** bibliotecas de clases preparadas para la reutilización que pueden utilizar a su vez componentes.
- **Objetos distribuidos:** paradigma que distribuye los objetos de cooperación a través de una red heterogénea y permite que los objetos interoperen como un todo unificado.
- **Patrones de diseño**

Sin embargo los mecanismos de reutilización también presentan algunos obstáculos:

1. El síndrome No Inventado Aquí: los desarrolladores de software no se suelen fiar de lo que no está supervisado por ellos mismos.
2. Acceso a los fuentes de componentes y frameworks.
3. Impedimentos legales, comerciales o estratégicos.
4. Formato de distribución de componentes (CORBA, ActiveX, ...).
5. Dilema entre reutilizar y rehacer.
6. Existe una inercia personal a los cambios.

Basándonos en la regla de reutilización (“se debe ser consumidor de reutilización antes de ser un productor de reutilización”), este proyecto trata de ser una **guía didáctica** de la utilización de **patrones de diseño** en el **desarrollo de aplicaciones**. Como veremos en los siguientes puntos los patrones de diseño son un mecanismo de reutilización utilizado en la fase de diseño. Para poder comprender la utilización de los patrones se deben tener unos conocimientos fundamentales del modelo de Objetos. Por esta causa se dan por supuestos ciertos conceptos fundamentales de las Tecnologías Orientadas a Objetos (clase, objeto, encapsulación, herencia, polimorfismo, agregación, ...) que se van a utilizar en los siguientes puntos del tutorial de patrones de diseño.

1.2. INTRODUCCIÓN A LOS PATRONES

Los patrones como elemento de la reutilización, comenzaron a utilizarse en la arquitectura con el objetivo de reutilizar diseños que se habían aplicado en otras construcciones y que se catalogaron como completos.

Christopher Alexander fue el primero en intentar crear un formato específico para patrones en la arquitectura. Alexander argumenta que los métodos comunes aplicados en la arquitectura dan lugar a productos que no satisfacen las demandas y requerimientos de los usuarios y son ineficientes a la hora de conseguir el propósito de todo diseño y esfuerzo de la ingeniería: mejorar la condición humana. Alexander describe algunos diseños eternos para tratar de

conseguir sus metas. Propone, así, un paradigma para la arquitectura basado en tres conceptos: la calidad, la puerta y el camino.

La Calidad (la Calidad Sin Nombre): la esencia de todas las cosas vivas y útiles que nos hacen sentir vivos, nos da satisfacción y mejora la condición humana.

La Puerta: el mecanismo que nos permite alcanzar la calidad. Se manifiesta como un lenguaje común de patrones. La puerta es el conducto hacia la calidad.

El Camino (El Camino Eterno): siguiendo el camino, se puede atravesar la puerta para llegar a la calidad.

De este modo el patrón trata de extraer la esencia de ese diseño (lo que se puede llamar “la calidad sin nombre”) para que pueda ser utilizada por otros arquitectos cuando se enfrentan a problemas parecidos a los que resolvió dicho diseño.

Alexander intenta resolver problemas arquitectónicos utilizando estos patrones. Para ello trata de extraer la parte común de los buenos diseños (que pueden ser dispares), con el objetivo de volver a utilizarse en otros diseños.

Christopher Alexander da la siguiente definición de patrón:

“Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo ni siquiera dos veces de la misma forma”.

Si nos fijamos en las construcciones de una determinada zona rural observaremos que todas ellas poseen apariencias parejas (tejados de pizarra con gran pendiente, etc.), pese a que los requisitos personales por fuerza han debido ser distintos. De alguna manera la esencia del diseño se ha copiado de una construcción a otra, y a esta esencia se pliegan de forma natural los diversos requisitos. Diríase aquí que existe un patrón que soluciona de forma simple y efectiva los problemas de construcción en tal zona.

Se puede utilizar una metáfora textil para explicar lo que es un patrón: es como la pieza que utiliza el sastre a la hora de confeccionar vestidos y trajes. De esta forma este patrón además de contener las especificaciones de corte y confección del producto final, representa a la vez, una parte de ese producto.

En definitiva se puede definir un patrón como *“una solución a un problema en un determinado contexto”*.

1.2.1. Desarrollo histórico

El término patrón se utiliza inicialmente en el campo de la arquitectura, por Christopher Alexander, a finales de los 70's. Este conocimiento es transportado al ámbito del desarrollo de software orientado a objetos y se aplica al diseño. De allí es extrapolado al desarrollo en general y a las demás etapas.

Desde 1964 hasta 1979 Christopher Alexander escribe varios libros acerca del planeamiento urbano y la construcción de edificios. Entre sus libros están:

- ✦ “*A pattern Language: Towns/Building/Construction*” de Christopher Alexander, Sara Ishikawa, Murria silverstein, Max Jacobson, Ingrid Fiksdahl-King y Shlomo Angel, 1977, Oxford University Press. 253 patrones, con el formato específico propuesto por Alexander, se dan cita en este texto, en el que además se propugna una integración del mejor-vivir con el medio físico circundante: gente-patrones-gente. Cuando se habla del “*libro de Alexander*” o del “*libro AIS*” (las iniciales de los primeros autores) se refieren a esta obra.
- ✦ “*The Timeless Way of Building*” en 1979.

En 1987, Ward Cunningham y Kent Beck trabajaron con Smaltalk y diseñaron interfaces de usuario. Decidieron, para ello, utilizar alguna de las ideas de Alexander para desarrollar un lenguaje pequeño de patrones para servir de guía a los programadores de Smaltalk. Así dieron lugar al libro “*Using Pattern Languages for Object-Oriented Programs*”.

Poco después, Jim Coplien comenzó a realizar un catálogo de idioms (que son un tipo de patrones) en C++ y publica su libro “*Advanced C++ Programming Styles and Idioms*” en 1991.

Desde 1990 a 1994, Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides (el grupo de los cuatro) realizaron un primer catálogo de patrones de diseño.

Posteriormente hubo diferentes discusiones a cerca de los patrones en las que participaron notables personalidades de los patrones.

Poco después de Abril de 1994 el libro “*Design Patterns: Elements of Reusable Object-Oriented Software*” (*Gang of Four*, [GoF]) es publicado.

En 1997 Brad Appleton publica “*Patterns and Software: Essential Concepts and Terminology*”.

Mark Grand publica en 1998 “*Patterns in Java (volume 1)*” que es un catálogo de patrones de diseño ilustrados con UML. En 1999 el mismo autor publica “*Patterns in Java (volume 2)*” en donde se añaden algunos patrones de diseño más y otros tipos de patrones tales como patrones de organización de código, patrones de optimización de código, etc.

1.2.2. Patrones de software

Los patrones para el desarrollo de software son uno de los últimos avances de la Tecnología Orientada a Objetos. Los patrones son una forma literaria para resolver problemas de ingeniería del software, que tienen sus raíces en los patrones de la arquitectura.

Los diseñadores y analistas de software más experimentados aplican de forma intuitiva algunos criterios que solucionan los problemas de manera elegante y efectiva. La ingeniería del software se enfrenta a problemas variados que hay que identificar para poder utilizar la misma solución (aunque matizada) con problemas similares.

Por otra parte, las metodologías Orientadas a Objetos tienen como uno de sus principios “no reinventar la rueda” para la resolución de diferentes problemas. Por lo tanto los patrones se convierten en una parte muy importante en las Tecnologías Orientadas a Objetos para poder conseguir la reutilización.

La ingeniería del software, tomando conceptos aplicados por primera vez en arquitectura, intenta construir patrones software para la resolución de problemas en dicho campo. Para conseguir esto debe existir una comunicación entre los distintos ingenieros para compartir los resultados obtenidos. Por tanto debe existir también un esquema de documentación con el objetivo de que la comunicación pueda entenderse de forma correcta. Esta comunicación no se debe reducir a la implementación, ya que únicamente fomenta el uso del “cortar y pegar”. Pueden referirse a distintos niveles de abstracción, desde un proceso de desarrollo hasta la utilización eficiente de un lenguaje de programación.

El objetivo de los patrones es crear un lenguaje común a una comunidad de desarrolladores para comunicar experiencia sobre los problemas y sus soluciones.

1.2.2.1. Definiciones

Los diferentes autores han dado diversas definiciones de lo que es un patrón software. Veamos a continuación alguna de ellas:

1. Dirk Riehle y Heinz Zullighoven:
Un patrón es la abstracción de una forma concreta que puede repetirse en contextos específicos.
2. Otras definiciones más aceptadas por la comunidad software son:
Un patrón es una información que captura la estructura esencial y la perspicacia de una familia de soluciones probadas con éxito para un problema repetitivo que surge en un cierto contexto y sistema.
Un patrón es una unidad de información nombrada, instructiva e intuitiva que captura la esencia de una familia exitosa de soluciones probadas a un problema recurrente dentro de un cierto contexto.

3. Según Richard Gabriel:

Cada patrón es una regla de tres partes, la cual expresa una relación entre un cierto contexto, un conjunto de fuerzas que ocurren repetidamente en ese contexto y una cierta configuración software que permite a estas fuerzas resolverse por si mismas.

Esta definición es similar a la dada por Alexander: *Cada patrón es una relación entre un cierto contexto, un problema y una solución. El patrón es, resumiendo, al mismo tiempo una cosa que tiene su lugar en el mundo, y la regla que nos dice cómo crear esa cosa y cuándo debemos crearla. Es al mismo tiempo una cosa y un proceso; al mismo tiempo una descripción que tiene vida y una descripción del proceso que la generó.*

1.2.2.2. Características de los patrones software

Hay que tener en cuenta que no todas las soluciones que tengan, en principio, las características de un patrón son un patrón, sino que debe probarse que es una solución a un problema que se repite. Para que se pueda considerar un patrón, éste debe pasar por unas pruebas que reciben el nombre de test de patrones. Mientras tanto esa solución recibe el nombre de proto-patrón.

Según Jim Coplien los buenos patrones deben tener las siguientes características:

- ✿ **Solucionar un problema:** los patrones capturan soluciones, no sólo principios o estrategias abstractas.
- ✿ **Ser un concepto probado:** los patrones capturan soluciones demostradas, no teorías o especulaciones.
- ✿ **La solución no es obvia:** muchas técnicas de solución de problemas tratan de hallar soluciones por medio de principios básicos. Los mejores patrones generan una solución a un problema de forma indirecta.
- ✿ **Describe participantes y relaciones entre ellos:** los patrones no sólo describen módulos sino estructuras del sistema y mecanismos más complejos.
- ✿ **El patrón tiene un componente humano significativo:** todo software proporciona a los seres humanos confort o calidad de vida (estética y utilidad).

Los patrones indican repetición, si algo no se repite, no es posible que sea un patrón. Pero la repetición no es la única característica importante. También necesitamos mostrar que un patrón se adapta para poder usarlo, y que es útil. **La repetición** es una característica cuantitativa pura, **la adaptabilidad** y **utilidad** son características cualitativas. Podemos mostrar la repetición simplemente aplicando *la regla de tres* (en al menos tres sistemas existentes); mostrar la adaptabilidad explicando *como* el patrón es exitoso; y mostrar la utilidad explicando *porque* es exitoso y beneficioso.

Así que aparte de la repetición, un patrón debe describir *como* la solución salda o resuelve sus objetivos, y *porque* está es una buena solución. Se puede decir que un patrón es donde la

teoría y la práctica se juntan para fortalecerse y complementarse, para mostrar que la estructura que describe es útil, utilizable y usada.

Un patrón debe ser *útil* porque enseña como el patrón que tenemos en nuestra mente puede ser transformado en una instancia del patrón en el mundo real, como una cosa que añade valor a nuestra vida como diseñadores. Un patrón debe ser también *utilizable* porque muestra como un patrón descrito de una forma literaria puede ser transformado en un patrón que tenemos en nuestra mente. Y un patrón debe ser *usado* porque es como los patrones que existen en el mundo real llegan a ser documentados como patrones de una forma literaria.

Esto potencia una continua repetición del ciclo desde los escritores de patrones, a los lectores de patrones, y a los usuarios de patrones: los escritores documentan los patrones de forma literaria haciéndolos utilizables para los lectores de patrones, quienes pueden recordarlos en sus mentes, lo cual los hace ser útiles para los diseñadores, quienes pueden usarlos en el mundo real, y mejorar la calidad de vida de los usuarios.

1.2.2.3. Clases de patrones software

Existen diferentes ámbitos dentro de la ingeniería del software donde se pueden aplicar los patrones:

1. **Patrones de arquitectura:** expresa una organización o esquema estructural fundamental para sistemas software. Proporciona un conjunto de subsistemas predefinidos, especifica sus responsabilidades, e incluye una guía para organizar las relaciones entre ellos.
2. **Patrones de diseño:** proporciona un esquema para refinar los subsistemas o componentes de un sistema software, o las relaciones entre ellos. Describe estructuras repetitivas de comunicar componentes que resuelven un problema de diseño en un contexto particular.
3. **Patrones de programación (Idioms patterns):** un idioma es un patrón de bajo nivel de un lenguaje de programación específico. Describe como implementar aspectos de componentes o de las relaciones entre ellos utilizando las facilidades del lenguaje de programación dado.
4. **Patrones de análisis:** describen un conjunto de prácticas que aseguran la obtención de un buen modelo de un problema y su solución.
5. **Patrones organizacionales:** describen la estructura y prácticas de las organizaciones humanas, especialmente en las que producen, usan o administran software.

La diferencia entre estas clases de patrones está en los diferentes niveles de abstracción y detalle, y del contexto particular en el cual se aplican o de la etapa en el proceso de desarrollo. Así, los patrones de arquitectura son estrategias de alto nivel que involucran a los componentes, las propiedades y mecanismos globales de un sistema. Los patrones de diseño son tácticas de media escala relacionados con la estructura y el comportamiento de entidades y sus relaciones. No influyen sobre toda la estructura del sistema, pero define micro

arquitecturas de subsistemas y componentes. Los patrones de programación son específicos de las técnicas de un lenguaje de programación que afectan a partes pequeñas del comportamiento de los componentes de un sistema. Los patrones de análisis se refieren a la etapa de análisis del ciclo de vida de construcción de software. Los patrones organizacionales describen la estructuración del personal en el desarrollo de software.

También se puede hablar de otros tipos de patrones software, como pueden ser:

1. Patrones de programación concurrente.
2. Patrones de interfaz gráfica.
3. Patrones de organización del código.
4. Patrones de optimización de código.
5. Patrones de robustez de código.
6. Patrones de la fase de prueba.

Entre las ventajas que se pueden citar de la utilización de patrones están:

1. Facilitan la comunicación interna.
2. Ahorran tiempo y experimentos inútiles.
3. Mejoran la calidad del diseño y la implementación.
4. Son como “normas de productividad”.
5. Facilitan el aprendizaje de los paquetes Java.

1.3. PATRONES DE DISEÑO

Una cosa que los diseñadores expertos no hacen es resolver cada problema desde el principio. A menudo reutilizan las soluciones que ellos han obtenido en el pasado. Cuando encuentran una buena solución, utilizan esta solución una y otra vez. Consecuentemente tu podrías encontrar patrones de clases y comunicaciones entre objetos en muchos sistemas orientados a objetos. Estos patrones solucionan problemas específicos del diseño y hacen los diseños orientados a objetos más flexibles, elegantes y por último reutilizables. Los patrones de diseño ayudan a los diseñadores a reutilizar con éxito diseños para obtener nuevos diseños. Un diseñador que conoce algunos patrones puede aplicarlos inmediatamente a problemas de diseño sin tener que descubrirlos.

El valor de la experiencia en los diseños es muy importante. Cuando tienes que hacer un diseño que crees que ya has solucionado antes pero no sabes exactamente cuando o como lo hiciste. Si pudieras recordar los detalles de los problemas anteriores y como los solucionaste, entonces podrías reutilizar la experiencia en vez de tener que volver a pensarlo.

El objetivo de los patrones de diseño es guardar la experiencia en diseños de programas orientados a objetos. Cada patrón de diseño nombra, explica y evalúa un importante diseño en los sistemas orientados a objetos. Es decir se trata de agrupar la experiencia en diseño de una forma que la gente pueda utilizarlos con efectividad. Por eso se han documentado los mas importantes patrones de diseño y presentado en catálogos.

Los patrones de diseño hacen más fácil reutilizar con éxito los diseños y arquitecturas. Expresando estas técnicas verificadas como patrones de diseño se hacen más accesibles para los diseñadores de nuevos sistemas. Los patrones de diseño te ayudan a elegir diseños alternativos que hacen un sistema reutilizable y evitan alternativas que comprometan la reutilización. Los patrones de diseño pueden incluso mejorar la documentación y mantenimiento de sistemas existentes. Es decir, los patrones de diseño ayudan a un diseñador a conseguir un diseño correcto rápidamente.

1.3.1. ¿Qué es un patrón de diseño?

Christopher Alexander dice: *“Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces, sin hacerlo ni siquiera dos veces de la misma forma”*.

Los trabajos de Alexander intentan identificar y resolver, en un marco descriptivo formal aunque no exacto, problemas esenciales en el dominio de la arquitectura. A los diseñadores de software les ha parecido trasladar muchas de las ideas de Alexander al dominio software. En software las soluciones son expresadas en términos de objetos e interfaces en lugar de paredes y puertas, pero el núcleo de ambos tipos de patrones es una solución a un problema en un contexto.

Alexander ha servido, en realidad, de catalizador de ciertas tendencias “constructivas” utilizadas en el diseño de sistemas software.

En general, un patrón tiene cuatro elementos esenciales:

1. El **nombre del patrón** se utiliza para describir un problema de diseño, su solución, y consecuencias en una o dos palabras. Nombrar un patrón incrementa inmediatamente nuestro vocabulario de diseño. Esto nos permite diseños a un alto nivel de abstracción. Tener un vocabulario de patrones nos permite hablar sobre ellos con nuestros amigos, en nuestra documentación, e incluso a nosotros mismos.
2. El **problema** describe cuando aplicar el patrón. Se explica el problema y su contexto. Esto podría describir problemas de diseño específicos tales como algoritmos como objetos. Podría describir estructuras de clases o objetos que son sintomáticas de un diseño inflexible. Algunas veces el problema incluirá una lista de condiciones que deben cumplirse para poder aplicar el patrón.
3. La **solución** describe los elementos que forma el diseño, sus relaciones, responsabilidades y colaboraciones. La solución no describe un diseño particular o implementación, porque un patrón es como una plantilla que puede ser aplicada en diferentes situaciones. En cambio, los patrones proveen una descripción

abstracta de un problema de diseño y como una disposición general de los elementos (clases y objetos en nuestro caso) lo soluciona.

4. Las **consecuencias** son los resultados de aplicar el patrón. Estas son muy importantes para la evaluación de diseños alternativos y para comprender los costes y beneficios de la aplicación del patrón.

Los patrones de diseño tienen un cierto nivel de abstracción. Los patrones de diseño no son diseños tales como la realización de listas y tablas hash que pueden ser codificadas en clases y reutilizadas. Un algoritmo puede ser un ejemplo de implementación de un patrón, pero es demasiado incompleto, específico y rígido para ser un patrón. Una regla o heurística puede participar en los efectos de un patrón, pero un patrón es mucho más. Los patrones de diseño son descripciones de las comunicaciones de objetos y clases que son personalizadas para resolver un problema general de diseño en un contexto particular.

Un patrón de diseño nombra, abstrae e identifica los aspectos clave de un diseño estructurado, común, que lo hace útil para la creación de diseños orientados a objetos reutilizables. Los patrones de diseño identifican las clases participantes y las instancias, sus papeles y colaboraciones, y la distribución de responsabilidades. Cada patrón de diseño se enfoca sobre un particular diseño orientado a objetos. Se describe cuando se aplica, las características de otros diseños y las consecuencias y ventajas de su uso.

Los patrones de diseño se pueden utilizar en cualquier lenguaje de programación orientado a objetos, adaptando los diseños generales a las características de la implementación particular.

1.3.2. Descripción de patrones de diseño

¿Cómo describimos los patrones de diseño? Las notaciones gráficas, aunque importantes y útiles, no son suficientes. Estás solo recogen el producto final del proceso de diseño como las relaciones entre clases y objetos. Para reutilizar el diseño, nosotros debemos también guardar las decisiones, alternativas y ventajas que nos llevaron a ese diseño. Los ejemplos concretos son también muy importantes, porque ellos nos ayudan a ver el funcionamiento del diseño.

Para describir los patrones de diseño se utiliza un formato consistente. Cada patrón es dividido en secciones de acuerdo con la siguiente plantilla. La plantilla nos muestra una estructura uniforme para la información, de tal forma que los patrones de diseño sean fáciles de aprender, comparar y utilizar.

1. Nombre del patrón

Esta sección consiste de un nombre del patrón y una referencia bibliografía que indica de donde procede el patrón. El nombre es significativo y corto, fácil de recordar y asociar a la información que sigue.

2. Objetivo

Esta sección contiene unas pocas frases describiendo el patrón. El objetivo aporta la esencia de la solución que es proporcionada por el patrón. El objetivo esta dirigido a programadores con experiencia que pueden reconocer el patrón como uno que ellos ya conocen, pero para el cual ellos no le han dado un nombre. Después de reconocer el patrón por su nombre y objetivo, esto podría ser suficiente para comprender el resto de la descripción del patrón.

3. Contexto

La sección de Contexto describe el problema que el patrón soluciona. Este problema suele ser introducido en términos de un ejemplo concreto. Después de presentar el problema en el ejemplo, la sección de Contexto sugiere una solución de diseño a ese problema.

4. Aplicabilidad

La sección Aplicabilidad resume las consideraciones que guían a la solución general presentada en la sección Solución. En que situaciones es aplicable el patrón.

5. Solución

La sección Solución es el núcleo del patrón. Se describe una solución general al problema que el patrón soluciona. Esta descripción puede incluir, diagramas y texto que identifique la estructura del patrón, sus participantes y sus colaboraciones para mostrar como se soluciona el problema. Debe describir tanto la estructura dinámica como el comportamiento estático.

6. Consecuencias

La sección Consecuencias explica las implicaciones, buenas y malas, del uso de la solución.

7. Implementación

La sección de Implementación describe las consideraciones importantes que se han de tener en cuenta cuando se codifica la solución. También puede contener algunas variaciones o simplificaciones de la solución.

8. Usos en el API de Java

Cuando hay un ejemplo apropiado del patrón en el núcleo del API de Java es comentado en esta sección. Los patrones que no se utilizan en el núcleo del API de Java no contienen esta sección.

9. Código del ejemplo

Esta sección contiene el código del ejemplo que enseña una muestra de la implementación para un diseño que utiliza el patrón. En la mayoría de estos casos, este será el diseño descrito en la sección de Contexto.

10. Patrones relacionados

Esta sección contiene una lista de los patrones que están relacionados con el patrón que se describe.

1.3.3. Cualidades de un patrón de diseño

En adición a los elementos mencionados anteriormente, un buen escritor de patrones expuso varias cualidades deseables. Doug Lea, en su libro “*Christopher Alexander: an Introduction for Object-Oriented Designers*” muestra una descripción detallada de estas cualidades, las cuales son resumidas a continuación:

- **Encapsulación y abstracción:** cada patrón encapsula un problema bien definido y su solución en un dominio particular. Los patrones deberían de proporcionar límites claros que ayuden a cristalizar el entorno del problema y el entorno de la solución empaquetados en un entramado distinto, con fragmentos interconectados. Los patrones también sirven como abstracciones las cuales contienen dominios conocidos y experiencia, y podrían ocurrir en distintos niveles jerárquicos de granularidad conceptual.
- **Extensión y variabilidad:** cada patrón debería ser abierto por extensión o parametrización por otros patrones, de tal forma que pueden aplicarse juntos para solucionar un gran problema. Un patrón solución debería ser también capaz de realizar un variedad infinita de implementaciones (de forma individual, y también en conjunción con otros patrones).
- **Generatividad y composición:** cada patrón, una vez aplicado, genera un contexto resultante, el cual concuerda con el contexto inicial de uno o más de uno de los patrones del catálogo. Esta subsecuencia de patrones podría luego ser aplicada progresivamente para conseguir el objetivo final de generación de un “todo” o solución completa. Los patrones son aplicados por el principio de evolución fragmentada. Pero los patrones no son simplemente de naturaleza lineal, más bien esos

patrones en un nivel particular de abstracción y granularidad podrían guiar hacia o ser compuestos con otros patrones para modificar niveles de escala.

- **Equilibrio:** cada patrón debe realizar algún tipo de balance entre sus efectos y restricciones. Esto podría ser debido a uno o más de un heurístico que son utilizados para minimizar el conflicto sin el contexto de la solución. Las invariaciones representadas en un problema subyacente solucionan el principio o filosofía para el dominio particular, y proveen una razón fundamental para cada paso o regla en el patrón.

El objetivo es este, si esta bien escrito, cada patrón describe un todo que es más grande que la suma de sus partes, debido a la acertada coreografía de sus elementos trabajando juntos para satisfacer todas sus variaciones reclamadas.

1.3.4. Clasificación de los patrones de diseño

Dado que hay muchos patrones de diseño necesitamos un modo de organizarlos. En esta sección clasificamos los patrones de diseño de tal forma que podamos referirnos a familias de patrones relacionados. La clasificación nos ayuda a saber lo que hace un patrón. Según el libro “*Patterns in Java (Volume 1)*” existen seis categorías:

PATRONES DE DISEÑO FUNDAMENTALES

Los patrones de esta categoría son los más fundamentales e importantes patrones de diseño conocidos. Estos patrones son utilizados extensivamente en otros patrones de diseño.

PATRONES DE CREACIÓN

Los patrones de creación muestran la guía de cómo crear objetos cuando sus creaciones requieren tomar decisiones. Estas decisiones normalmente serán resueltas dinámicamente decidiendo que clases instanciar o sobre que objetos un objeto delegará responsabilidades. La valía de los patrones de creación nos dice como estructurar y encapsular estas decisiones.

A menudo hay varios patrones de creación que puedes aplicar en una situación. Algunas veces se pueden combinar múltiples patrones ventajosamente. En otros casos se debe elegir entre los patrones que compiten. Por estas razones es importante conocer los seis patrones descritos en esta categoría.

PATRONES DE PARTICIÓN

En la etapa de análisis, tu examinarás el problema para identificar los actores, casos de uso, requerimientos y las relaciones que constituyen el problema. Los patrones de esta categoría proveen la guía sobre como dividir actores complejos y casos de uso en múltiples clases.

PATRONES ESTRUCTURALES

Los patrones de esta categoría describen las formas comunes en que diferentes tipos de objetos pueden ser organizados para trabajar unos con otros.

PATRONES DE COMPORTAMIENTO

Los patrones de este tipo son utilizados para organizar, manejar y combinar comportamientos.

PATRONES DE CONCURRENCIA

Los patrones de esta categoría permiten coordinar las operaciones concurrentes. Estos patrones se dirigen principalmente a dos tipos diferentes de problemas:

1. **Recursos compartidos:** Cuando las operaciones concurrentes acceden a los mismos datos o otros tipos de recursos compartidos, podría darse la posibilidad de que las operaciones interfirieran unas con otras si ellas acceden a los recursos al mismo tiempo. Para garantizar que cada operación se ejecuta correctamente, la operación debe ser protegida para acceder a los recursos compartidos en solitario. Sin embargo, si las operaciones están completamente protegidas, entonces podrían bloquearse y no ser capaces de finalizar su ejecución.

El bloqueo es una situación en la cual una operación espera por otra para realizar algo antes de que esta proceda. Porque cada operación esta esperando por la otra para hacer algo, entonces ambas esperan para siempre y nunca hacen nada.

2. **Secuencia de operaciones:** Si las operaciones son protegidas para acceder a un recurso compartido una cada vez, entonces podría ser necesario garantizar que ellas acceden a los recursos compartidos en un orden particular. Por ejemplo, un objeto nunca será borrado de una estructura de datos antes de que esté sea añadido a la estructura de datos.

Fundamentales	De creación	De Partición	Estructurales	De comportamiento	De Concurrency
Delegation (When Not to Use Inheritance)	Factory Method	Layered Initialization	Adapter	Chain of Responsibility	Single Threaded Execution
Interface	Abstract Factory	Filter	Iterator	Command	Guarded Suspension
Immutable	Builder	Composite	Bridge	Little Language	Balking
Marker Interface	Prototype		Facade	Mediator	Scheduler
Proxy	Singleton		Flyweight	Snapshot	Read/Write Lock
	Object Pool		Dynamic Linkage	Observer	Producer-Consumer
			Virtual Proxy	State	Two-Phase Termination
			Decorator	Null Object	
			Cache Management	Strategy	
				Template Method	
				Visitor	

1.3.5. El catálogo de patrones de diseño

Si aceptamos que los patrones pueden resultar útiles en el desarrollo de software, el siguiente paso es reunirlos en catálogos de forma que resulten accesibles mediante distintos criterios, pues lo que necesitamos no es tan sólo la completa descripción de cada uno de los patrones sino, esencialmente, la correspondencia entre un problema real y un patrón (o conjunto de patrones) determinado.

Lo que se pretende con un catálogo de patrones no es favorecer al diseñador experto (que quizás no necesite en absoluto de los patrones), sino más bien ayudar al diseñador inexperto a adquirir con cierta rapidez las habilidades de aquél, como también comunicar al posible cliente, si es el caso, las decisiones de diseño de forma clara y autosuficiente.

Un catálogo de patrones es un medio para comunicar la experiencia de forma efectiva, reduciendo lo que se conoce como “*curva de aprendizaje*” del diseño.

A continuación se exponen los 41 patrones de diseño que contiene el libro “*Patterns in Java (Volumen 1)*” clasificados por categorías poniendo el nombre del patrón, seguido de una referencia bibliográfica (entre corchetes) que indica de donde procede el patrón y su objetivo:

PATRONES DE DISEÑO FUNDAMENTALES

Los patrones de esta categoría son los más fundamentales e importantes patrones de diseño conocidos. Estos patrones son utilizados extensivamente en otros patrones de diseño.

Delegation: (When Not to Use Inheritance) [Grand98]

Es una forma de extender y reutilizar las funcionalidades de una clase escribiendo una clase adicional con funcionalidad añadida que utiliza instancias de la clase original que suministra las funcionalidades originales.

Interface [Grand98]

Mantiene una clase que utiliza datos y servicios suministrados por instancias de otras clases independientes teniendo acceso a estas instancias a través de un interface.

Immutable [Grand98]

Incrementa la robustez de los objetos que comparten referencias a los mismos objetos y reduce el alto acceso concurrente a un objeto. Prohíbe que ninguna información del estado de los objetos cambie después de que el objeto es construido. El patrón Immutable también evita la necesidad de sincronizar múltiples hilos de ejecución que comparten un objeto.

Marker Interface [Grand98]

Utiliza interfaces que no declaran métodos o variables que indiquen la semántica de los atributos de una clase. Esto funciona particularmente bien con utilidades de clases que deben determinar algo acerca de los objetos sin asumir que ellos son una instancia de alguna clase particular.

Proxy [GoF95]

Es un patrón muy general que ocurre en muchos otros patrones, pero nunca él mismo en su puro estado. El patrón Proxy hace llamadas a métodos de un objeto que ocurre indirectamente a través de un objeto proxy que actúa como un sustituto para otro objeto, delegando las llamadas a métodos a ese objeto. Las clases para los objetos proxy son declaradas de una forma que normalmente eliminan el conocimiento de los objetos del cliente que son tratados con un proxy. Es decir, este patrón provee un representante de acceso a otro objeto.

PATRONES DE CREACIÓN

Los patrones de creación muestran la guía de cómo crear objetos cuando sus creaciones requieren tomar decisiones. Estas decisiones normalmente serán resueltas dinámicamente decidiendo que clases instanciar o sobre que objetos un objeto delegará responsabilidades. La valía de los patrones de creación nos dice como estructurar y encapsular estas decisiones.

A menudo hay varios patrones de creación que puedes aplicar en una situación. Algunas veces se pueden combinar múltiples patrones ventajosamente. En otros casos se debe elegir entre los patrones que compiten. Por estas razones es importante conocer los seis patrones descritos en esta categoría.

Factory Method [GoF95]

Tu escribes una clase para reutilizar con tipos arbitrarios de datos. Tu organizas esta clase de tal forma que se puedan instanciar otras clases sin depender de ninguna de las clases que puedan ser instanciadas. La clase reutilizable es capaz de permanecer independiente de las otras clases instanciadas delegando la elección de que clase instanciar a otros objetos y referirse a los objetos recién creados a través de un interface común. Define una interface para crear un objeto, dejando a las subclasses decidir el tipo específico. Permite delegar la responsabilidad de instanciación a las subclasses.

Abstract Factory [GoF95]

Dado un conjunto de clases abstractas relacionadas, el patrón Abstract Factory permite el modo de crear instancias de estas clases abstractas desde el correspondiente conjunto de subclasses concretas. Proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar su clase concreta. El patrón Abstract Factory puede ser muy útil para permitir a un programa trabajar con una variedad compleja de entidades externas, tales como diferentes sistemas de ventanas con una funcionalidad similar.

Builder [GoF95]

Permite a un objeto cliente construir un objeto complejo especificando solamente su tipo y contenido. El cliente es privado de los detalles de construcción del objeto. Separa la construcción de un objeto complejo de su representación, para que el mismo proceso de construcción permita crear varias representaciones.

Prototype [GoF95]

Permite a un objeto crear objetos personalizados sin conocer su clase exacta o los detalles de cómo crearlos. Su tarea es dar objetos prototipo a un objeto que inicializa la creación de objetos. El objeto de creación e inicialización crea objetos mandando a sus objetos prototipo que hagan una copia de si mismos.

Singleton [GoF95]

Garantiza que solamente se crea una instancia de la clase y provee un punto de acceso global a él. Todos los objetos que utilizan una instancia de esa clase usan la misma instancia.

Object Pool [Grand98]

Administra la reutilización de objetos cuando un tipo de objetos es caro de crear o solamente un número limitado de objetos puede ser creado.

PATRONES DE PARTICIÓN

En la etapa de análisis, tu examinas un problema para identificar los actores, casos de uso, requerimientos y las relaciones que constituyen el problema. Los patrones de esta categoría proveen la guía sobre como dividir actores complejos y casos de uso en múltiples clases.

Layered Initialization [Grand98]

Cuando se necesitan múltiples implementaciones de una abstracción, normalmente se define una clase que encapsula la lógica común y subclases que encapsulan las diferentes lógicas especializadas. Esto no se puede hacer cuando la lógica común es utilizada para decidir que clases especializadas se crean. El patrón Layered Initialization soluciona este problema encapsulando las lógicas comunes y especializadas para crear un objeto en clases no relacionadas.

Filter [BMRSS96]

Permite a los objetos que realizan diferentes transformaciones y cálculos sobre un conjunto de datos y que tienen interfaces compatibles conectar dinámicamente en el orden en que se realizan las operaciones arbitrariamente sobre el conjunto de datos.

Composite [GoF95]

Permite construir objetos complejos mediante composición recursiva de objetos similares. El patrón Composite también permite que los objetos del árbol sean manipulados por un manejador consistente, para requerir todos los objetos hay una superclase o un interfaz común. Permite a los clientes tratar de la misma manera tanto a objetos individuales como a compuestos.

PATRONES ESTRUCTURALES

Los patrones de esta categoría describen las formas comunes en que diferentes tipos de objetos pueden ser organizados para trabajar unos con otros.

Adapter [GoF95]

Una clase Adapter implementa un interfaz que conoce a sus clientes y proporciona acceso a una instancia de una clase que no conoce a sus clientes, es decir convierte la interfaz de una clase en una interfaz que el cliente espera. Un objeto Adapter proporciona la funcionalidad prometida por un interfaz sin tener que conocer que clase es utilizada para implementar ese interfaz. Permite trabajar juntas a dos clases con interfaces incompatibles.

Iterator [GoF95]

Define un interface que declara métodos para acceder secuencialmente a los objetos de una colección. Una clase que accede a una colección solamente a través de un interface independiente de la clase que implementa el interface.

Bridge [GoF95]

Es útil cuando hay una jerarquía de abstracciones y la correspondiente jerarquía de implementaciones. Más que combinar las abstracciones e implementaciones en muchas clases distintas, el patrón Bridge implementa las abstracciones e implementaciones como clases independientes que se pueden combinar dinámicamente. Es decir, desacopla una abstracción de su implementación y les permite variar independientemente.

Facade [GoF95]

Simplifica los accesos a un conjunto de objetos relacionados proporcionando un objeto que todos los objetos de fuera del conjunto utilizan para comunicarse con el conjunto. Define una interface de más alto nivel que permite usar el sistema más fácil.

Flyweight [GoF95]

Si instancias de una clase contienen la misma información pueden ser utilizadas intercambiamente, el patrón Flyweight permite a un programa evitar el coste de múltiples instancias que contienen la misma información compartiendo una única instancia. Soporta la representación de un gran número de objetos pequeños de una manera eficiente.

Dynamic Linkage [Grand98]

Permite a un programa solicitar la carga y utilización de clases arbitrarias que implementan un interface conocido.

Virtual Proxy [Larman98]

Si un objeto es caro de instanciar y puede que no sea necesario, podría tener ventajas posponer la instanciación hasta que este claro que el objeto se necesita. El patrón Virtual Proxy oculta el hecho de que el objeto todavía no existe a sus clientes, teniendo éstos acceso al objeto indirectamente a través de un objeto proxy que implementa el mismo interface que el

objeto que puede no existir. La técnica de retrasar la instanciación de un objeto hasta que este es necesario se llama *instanciación perezosa*.

Decorator [GoF95]

Extiende la funcionalidad de un objeto dinámicamente de tal modo que es transparente a sus clientes, utilizando una instancia de una subclase de la clase original que delega las operaciones al objeto original. Provee una alternativa muy flexible para agregar funcionalidad a una clase.

Cache Management [Grand98]

Permite el acceso rápido a objetos que de lo contrario podrían llevar un gran tiempo de acceso. Esto implica tener una copia de los objetos que son costosos de construir. El objeto podría ser costoso de construir por un número de razones tales como necesitar un gran cálculo o ser sacado de una base de datos.

PATRONES DE COMPORTAMIENTO

Los patrones de este tipo son utilizados para organizar, manejar y combinar comportamientos.

Chain of Responsibility [GoF95]

Permite a un objeto enviar un comando sin conocer que objeto o objetos lo recibirán, evitando el acoplamiento entre el que envía y el que recibe una petición. Esto permite el paso del comando a un objeto de la cadena que es parte de una gran estructura. Cada objeto de la cadena podría manejar el comando, pasar el comando al siguiente objeto de la cadena, o las dos cosas.

Command [GoF95]

Encapsula una operación en un objeto, permitiendo parametrizar operaciones, de tal forma que se pueda controlar su selección y secuencia, ponerlas en la cola, deshacerlas y manipularlas.

Little Language [Grand98]

Supón que necesitas resolver muchos problemas similares y te das cuenta de que las soluciones a estos problemas pueden ser expresadas como diferentes combinaciones de un pequeño número de elementos o operaciones. El modo más simple de expresar las soluciones a estos problemas podría ser definir un pequeño lenguaje. Los tipos comunes de problemas que se pueden solucionar con un pequeño lenguaje son búsquedas de estructuras de datos comunes, creación de complejas estructuras de datos, y asignación de formato a los datos.

Mediator [GoF95]

Define un objeto que encapsula la forma en la cual interactúan otros. Utiliza un objeto para coordinar cambios de estado entre otros objetos. Colocando la lógica en un objeto para manejar cambios de estado de otros objetos, en lugar de distribuir la lógica sobre los otros objetos, dando como resultado una implementación mas cohesiva de la lógica y decrementando las relaciones entre otros objetos. Promueve el acoplamiento débil.

Snapshot [Grand98]

Captura una instantánea del estado de un objeto pudiendo ser restaurado más tarde. El objeto que inicializa la captura o restauración del estado no necesita conocer ninguna información del estado. Sólo se necesita conocer cual es el objeto cuyo estado se va a restaurar o capturar implementando un interface particular.

Observer [GoF95]

Permite a los objetos captar dinámicamente las dependencias entre objetos, de tal forma que un objeto notificará a los objetos dependientes de él cuando cambia su estado, siendo actualizados automáticamente.

State [GoF95]

Encapsula los estados de un objeto como objetos separados, cada pertenencia en una subclase separada de una clase abstracta estado. Permite a un objeto cambiar su comportamiento cuando su estado interno cambia. El objeto parecerá haber cambiado de clase.

Null Object [Woolf97]

Facilita una alternativa para utilizar null para indicar la ausencia de un objeto al que delegue una operación. Utilizando null para indicar la ausencia de cada uno de los objetos requeridos se realiza una pregunta para ver si es null antes de cada llamada a los métodos de otros objetos. En lugar de utilizar null, el patrón Null Object utiliza una referencia a un objeto que no hace nada.

Strategy [GoF95]

Encapsula algoritmos relacionados en clases que son subclases de una superclase común. Esto permite la selección de un algoritmo que varia según el objeto y también le permite la variación en el tiempo.

Template Method [GoF95]

Escribe una clase abstracta que contiene parte de la lógica necesaria para realizar su finalidad. Organiza la clase de tal forma que sus métodos concretos llaman a un método abstracto

donde la lógica buscada tendría que aparecer. Facilita la lógica buscada en métodos de subclases que sobrescriben a los métodos abstractos. Define un esqueleto de un algoritmo, delegando algunos pasos a las subclases. Permite redefinir ciertos pedazos del algoritmo sin cambiar su estructura.

Visitor [GoF95]

Un modo de implementar una operación que involucra a los objetos en una estructura compleja es facilitar la lógica en cada una de sus clases para soportar la operación. El patrón Visitor facilita un modo alternativo para implementar cada operación que evita complicar la estructura de las clases de los objetos colocando toda la lógica necesaria en una clase Visitor separada. El patrón Visitor también permite variar la lógica utilizando diferentes clases Visitor, es decir, permite definir nuevas operaciones sin cambiar las clases de los objetos sobre las cuales operan.

PATRONES DE CONCURRENCIA

Los patrones de esta categoría permiten coordinar las operaciones concurrentes. Estos patrones se dirigen principalmente a dos tipos diferentes de problemas:

1. **Recursos compartidos:** Cuando las operaciones concurrentes acceden a los mismos datos o otros tipos de recursos compartidos, podría darse la posibilidad de que las operaciones interfirieran unas con otras si ellas acceden a los recursos al mismo tiempo. Para garantizar que cada operación se ejecuta correctamente, la operación debe ser protegida para acceder a los recursos compartidos en solitario. Sin embargo, si las operaciones están completamente protegidas, entonces podrían bloquearse y no ser capaces de finalizar su ejecución.

El bloqueo es una situación en la cual una operación espera por otra para realizar algo antes de que esta proceda. Porque cada operación esta esperando por la otra para hacer algo, entonces ambas esperan para siempre y nunca hacen nada.

2. **Secuencia de operaciones:** Si las operaciones son protegidas para acceder a un recurso compartido una cada vez, entonces podría ser necesario garantizar que ellas acceden a los recursos compartidos en un orden particular. Por ejemplo, un objeto nunca será borrado de una estructura de datos antes de que esté sea añadido a la estructura de datos.

Single Threaded Execution [Grand98]

Algunos métodos acceden a datos o otros recursos compartidos de tal forma que producen resultados incorrectos si hay llamadas concurrentes a un método y ambas llamadas acceden a los datos o otros recursos compartidos al mismo tiempo. El patrón Single Threaded Execution

soluciona este problema impidiendo las llamadas concurrentes a un método que provocan ejecuciones concurrentes del método.

Guarded Suspensión [Lea97]

Suspende la ejecución de una llamada a un método hasta que se satisface una precondition.

Balking [Lea97]

Si un método de un objeto es llamado cuando el objeto no está en un estado apropiado para ejecutar ese método, tiene que retornar del método sin hacer nada.

Scheduler [Lea97]

Controla el orden en el cual los hilos son organizados para ejecutar el código del hilo único utilizando un objeto que sigue explícitamente en la sucesión de hilos que esperan. El patrón Scheduler provee un mecanismo para implementar una política de organización. Esto es independiente de cada una de las políticas de organización específicas.

Read/Write Lock [Lea97]

Permite accesos concurrentes de lectura a un objeto pero requiere el acceso exclusivo para operaciones de escritura.

Producer-Consumer

Coordina la producción y consumo asíncrono de información o objetos.

Two-Phase Termination [Grand98]

Facilita el cierre de forma ordenada de un hilo o proceso a través de la colocación de un pestillo. El hilo o proceso comprueba el valor del pestillo en puntos estratégicos de su ejecución.

1.3.6. Patrones de diseño, reglas y creatividad

La presencia combinada de todos los elementos de un patrón y cualidades es lo que hace que los patrones sean más que simples heurísticos, reglas o algoritmos. Los heurísticos y principios participan frecuentemente en los objetivos y/o razón fundamental de un patrón, pero solamente son elementos de un patrón. Además de esto como Cope escribe en “*Software Design Patterns: Common Questions and Answers*”:

Las reglas no son comúnmente soportadas por una razón fundamental, no ponen un contexto. Una regla podría ser parte de la solución en la descripción de un patrón, pero una regla como solución no es aún suficiente ni necesaria. Los

patrones no son diseños para ser ejecutados o analizados por ordenadores, como uno podría imaginar cierto para las reglas: los patrones tienen que ser ejecutados por arquitectos con perspicacia, sabiduría, experiencia, y un sentido de la estética.

Un patrón es el proceso que genera una solución, pero podría generar un inmenso número de soluciones variantes (imaginablemente sin repetir la misma solución dos veces). El elemento humano de los patrones es que principalmente contribuye a su variabilidad y adaptabilidad, y normalmente necesita un gran grado de creatividad en sus aplicaciones y combinaciones. Así que, como el proceso de arquitectura y diseño son propósitos creativos, así también es la aplicación de patrones. En el mismo libro mencionado anteriormente, Cope dice:

Si el diseño es codificado en patrones, ¿la creatividad necesaria será grande? ¿Podemos reemplazar nosotros los caros diseños con los programadores menos sofisticados que se guían por patrones? La respuesta es que la creatividad es todavía necesaria para dar forma a los patrones en un contexto dado. También como los modistas ajustan a la medida un patrón a un cliente individual, y quizás algún suceso específico cuando el vestido esta puesto, así los diseñadores deben ser creativos cuando utilizan patrones. Los patrones canalizan la creatividad; no la reemplazan ni la reducen.

1.3.7. Patrones y algoritmos

La sección titulada patrones de diseño, reglas y creatividad también se aplica en gran parte a los algoritmos y sus estructuras de datos. Ciertamente, los algoritmos y estructuras de datos pueden ser empleados en la implementación de uno o varios patrones, pero los algoritmos y las estructuras de datos generalmente solucionan problemas computacionales más concretos como ordenación y búsqueda. Los patrones son típicamente interesantes en arquitecturas de gran extensión, que tienen efectos de gran escala. Los patrones de diseño solucionan a la gente y desarrolladores cuestiones como mantenibilidad, reusabilidad, comunicabilidad y encapsulación.

Los algoritmos y estructuras de datos son normalmente relacionados casi exclusivamente con el contexto de optimización o tiempo, o algunos otros aspectos de la computación compleja y recursos en uso. Tratan de encontrar la más compacta y eficiente solución que realiza alguna computación importante o almacenar y recordar sus resultados.

Hay un gran número de libros de algoritmos y estructuras de datos que proporcionan el código fuente y análisis para estructuras como los árboles AVL o los árboles desplegados. Pero en muchos de estos mismos libros hay una pequeña mención de cómo implementar estas estructuras de un modo que proporcione mantenibilidad, adaptabilidad y reutilización. Mirando el tiempo computacional no soluciona el conjunto de efectos que afectan a las arquitecturas e implementaciones tan bien como los usuarios.

Esto es porque los algoritmos y estructuras de datos tienden a ser más concretos que los patrones: porque ellos solucionan mayormente cuestiones de complejidad computacional, y no tanto las cuestiones subyacentes de la gente que utiliza y construye software. Los patrones fundamentalmente solucionan cuestiones más complejas que las simples cuestiones de eficiencia/memoria hardware y software.

Por supuesto los desarrolladores de software necesitan conocer ambos y encontrar las arquitecturas apropiadas y con las soluciones apropiadas para los problemas computacionales. Así serán tan necesarios los patrones como los algoritmos y las estructuras de datos (y su uso conjunto).

1.3.8. Patrones y frameworks

Relacionados con los patrones de diseño están los frameworks. Un framework es una mini arquitectura que proporciona la estructura genérica y el comportamiento para una familia de abstracciones a lo largo de un contexto de metáforas que especifican su colaboración y es utilizado en un dominio dado.

El framework codifica el contexto en una clase de máquina virtual mientras hace las abstracciones abiertas. Un framework no es generalmente una aplicación completa sino que a menudo le falta la funcionalidad de una aplicación específica. Una aplicación, en cambio puede ser construida con uno o más frameworks insertando dicha funcionalidad. Una posible definición de framework es: *conjunto de clases cooperantes que hace reusable un diseño para una clase específica de software*. Un framework proporciona una guía arquitectónica para dividir el diseño en clases y definir sus responsabilidades y colaboraciones. Un framework dicta la arquitectura de la aplicación. Define el conjunto de la estructura, su partición en clases y objetos, la colaboración entre ellos y el hilo de control.

La diferencia entre un framework y una librería ordinaria de programación es que el framework utiliza un flujo de control invertido con respecto a los clientes.

Los frameworks pueden incluir patrones de diseño. Sin embargo hay que distinguirlos los frameworks son ejecutables mientras que los patrones de diseño representan un conocimiento o experiencia sobre software. En realidad, un framework se puede ver como la implementación de un sistema de patrones de diseño. Los frameworks son de naturaleza física, mientras que los patrones son de naturaleza lógica: los frameworks son la realización física de uno o varios patrones de soluciones software; los patrones son las instrucciones de cómo implementar estas soluciones.

Las principales diferencias entre patrones de diseño y frameworks son:

1. **Los patrones de diseño son más abstractos que los frameworks.** Los frameworks pueden ser codificados, pero solo los *ejemplos* de patrones de diseño pueden ser codificados. Una característica de los frameworks es que pueden ser escritos bajo lenguajes de programación y no solamente estudiados para ejecutarse y

reutilizarse directamente. En cambio, los patrones de diseño tienen que ser implementados cada vez que son usados. Los patrones de diseño también explican el objetivo, los cambios y las consecuencias de un diseño.

2. **Los patrones de diseño son elementos arquitectónicos más pequeños que los frameworks.** Un framework contiene varios patrones de diseño, lo contrario nunca es cierto.
3. **Los patrones de diseño están menos especializados que los frameworks.** Los frameworks siempre tienen un dominio particular de aplicación. En cambio, los patrones de diseño pueden ser utilizados en cualquier tipo de aplicación.

1.3.9. Donde usar los patrones

Vamos a ver algunos casos en los que los patrones de diseño se muestran como una solución efectiva para resolver problemas de diseño:



ENCONTRAR LOS OBJETOS APROPIADOS

La parte difícil del diseño orientado a objetos es descomponer un sistema en objetos. La tarea es difícil porque muchos factores entran en juego: encapsulación, granularidad, dependencia, flexibilidad, realización, evolución, reusabilidad. Todos ellos influyen en la descomposición, a menudo de forma conflictiva.

Las metodologías de diseño orientadas a objetos tienen diferentes aproximaciones. Puedes escribir la declaración de un problema, simplemente sacando los nombres y los verbos, y creando las correspondientes clases y operaciones. O puedes enfocarlo sobre las colaboraciones y responsabilidades del sistema. O puedes modelar el mundo real y trasladar los objetos encontrados durante el análisis al diseño. Siempre habrá desacuerdos sobre que aproximación es la mejor.

Muchos objetos en un diseño vienen del modelo de análisis. Pero los diseños orientados a objetos a menudo terminan en clases que no tienen homólogos en el mundo real. Algunos de estos son clases de bajo nivel como los arrays. Otros son de nivel más alto. Por ejemplo el patrón *Composite* introducen una abstracción para tratar uniformemente los objetos que no tienen un homólogo físico. El modelado estricto del mundo real nos guía a un sistema que refleja las realidades de hoy pero no necesariamente las de mañana. Las abstracciones que emergen durante el diseño son la clave para hacer un diseño flexible.

Los patrones de diseño te ayudan a identificar las abstracciones menos obvias y los objetos que pueden capturarlas. Por ejemplo, los objetos que representan un algoritmo no ocurren en la naturaleza, aunque son una parte crucial de los diseños flexibles. El patrón *Strategy* describe como implementar una familia de algoritmos intercambiables. El patrón *State* representa cada estado de una entidad como un objeto. Estos objetos son raramente encontrados durante el análisis e incluso en las primeras etapas del diseño; sino que son descubiertas más tarde en el transcurso de realizar un diseño más flexible y reutilizable.

DETERMINAR EL NÚMERO Y TAMAÑO DE LOS OBJETOS

Los objetos pueden variar enormemente en tamaño y número. ¿Cómo decidimos como debería ser un objeto?

Los patrones de diseño también solucionan esta cuestión. El patrón *Facade* describe como representar subsistemas completos como objetos, y el patrón *Flyweight* como soportar un enorme número de objetos de pequeño tamaño. Otros patrones de diseño describen formas de descomponer un objeto en objetos más pequeños. El *Abstract Factory* y el *Builder* proporcionan objetos cuyas únicas responsabilidades son crear otros objetos. El *Visitor* y el *Command* proporcionan objetos cuyas únicas responsabilidades son las de implementar una petición sobre otro objeto o grupo de objetos.

ESPECIFICAR LAS INTERFACES DE LOS OBJETOS

Cada operación declarada en un objeto especifica el nombre de la operación, los objetos que tiene como parámetros, y el valor que retorna la operación. Esto se conoce como la signatura de la operación. El conjunto de todas las signaturas definidas en un objeto se llama interface del objeto. Un interface de un objeto caracteriza el conjunto completo de peticiones que pueden ser enviadas al objeto. Un tipo es un nombre utilizado para denotar un interface particular.

Los interfaces son fundamentales en los sistemas orientados a objetos. Los objetos solamente se conocen a través de sus interfaces. No hay forma de conocer nada acerca de un objeto o preguntarle nada sin ser a través de su interface. Un interface de un objeto no dice nada sobre su implementación, diferentes objetos tienen libertad para implementar las peticiones de forma diferente. Esto significa que dos objetos que tienen implementaciones completamente diferentes pueden tener interfaces idénticos.

Los patrones de diseño te ayudan a definir interfaces para identificar sus elementos claves y los tipos de datos que consiguen enviar a través de un interface. Un patrón de diseño también puede decirnos que no poner en el interface.

Los patrones de diseño también especifican relaciones entre interfaces. En particular, ellos a menudo requieren algunas clases que tengan interfaces similares, o meter restricciones sobre los interfaces de algunas clases. Por ejemplo, el *Decorator* y el *Proxy* requieren que los interfaces de objetos Decorator y Proxy sean idénticos para los objetos Decorados y Proxies. En el *Visitor*, el interface Visitor debe reflejar todas las clases de objetos que los visitor pueden visitar.

ESPECIFICAR LA IMPLEMENTACIÓN DE LOS OBJETOS

1. **Herencia de clases frente a herencia de interfaces:** Muchos patrones de diseño dependen de esta distinción. Por ejemplo, los objetos en el patrón *Chain of Responsibility* tienen que tener un interface común, pero normalmente no comparten una implementación común. En el patrón *Composite*, el componente define un interface común, pero el compuesto a menudo define una implementación común. Los patrones *Command*, *State*, y *Strategy* son a menudo implementados con clases abstractas que son interfaces puros.
2. **La programación hacia un interface, no una implementación:** Cuando la herencia es utilizada cuidadosamente, todas las clases derivan

de una clase abstracta compartiendo su interface. Esto implica que una subclase sólo añade o sobrescribe operaciones y no esconde operaciones de la clase padre. Todas las subclases pueden luego responder a las peticiones del interface de esta clase abstracta, siendo todas ellas subtipos de la clase abstracta.

La manipulación de objetos solamente en términos del interface definido por clases abstractas tiene dos beneficios:

- Los clientes permanecen sin enterarse de los tipos específicos de objetos que usan, los objetos son adheridos al interface que los clientes esperan.
- Los clientes permanecen sin enterarse de las clases que implementan esos objetos. Los clientes solamente conocen las clases abstractas definidas en el interface.

La reducción de las dependencias de implementación entre subsistemas nos conduce al siguiente principio del diseño orientado a objetos reutilizable: *Programar un interface, no una implementación*. No declarar variables que sean instancias de una clase particular. En cambio consigna solamente un interface definido por una clase abstracta. Esto es un tema común en los patrones de diseño.

Si tienes que instanciar clases concretas (esto es, especificar una implementación particular) en alguna parte de tu sistema, los patrones de creación (*Abstract Factory*, *Builder*, *Factory Method*, *Prototype*, y *Singleton*) te permiten hacerlo. Para abstraer el proceso de creación del objeto, estos patrones te dan diferentes caminos para asociar un interface con su implementación transparentemente a la instanciación. Los patrones de creación aseguran que tu sistema esta escrito en términos de interfaces, no de implementaciones.

HACER FUNCIONAR LOS MECANISMOS DE REUTILIZACIÓN

Mucha gente puede comprender conceptos como objetos, interfaces, clases, y herencia. El desafío consiste en aplicarlos para construir software flexible y reutilizable, y los patrones de diseño pueden enseñarte como hacerlo.

1. Herencia frente a composición: Reuso de caja blanca o caja negra:

Las dos técnicas más comunes para reutilizar funcionalidades en un sistema orientado a objetos son la herencia de clases y la composición de objetos.

La herencia de clases nos permite definir la implementación de una clase en términos de otra. La reutilización mediante subclases suele ser referida como reutilizar la caja blanca. El término “caja blanca” se refiere a la visibilidad: con la herencia el interactuar de la clase padre es a menudo visible para las subclases.

La composición de objetos es una alternativa a la herencia de clases. Aquí las nuevas funcionalidades son obtenidas ensamblando o componiendo objetos para conseguir funcionalidades más complejas. La composición de objetos requiere que los objetos sean compuestos teniendo interfaces bien definidos. Este estilo de reutilización se llama reutilizar la caja negra,

porque los detalles internos de los objetos no son visibles. Los objetos parecen solamente “cajas negras”.

La herencia y la composición tienen cada una sus ventajas y sus desventajas. La herencia de clases se define estáticamente en tiempo de compilación y es fácil de utilizar. La composición de objetos se define dinámicamente en tiempo de ejecución a través de objetos que adquieren referencias a otros objetos.

La composición de objetos tiene otro efecto sobre el diseño de un sistema. Favorecer la composición sobre la herencia de clases nos ayuda a conservar la encapsulación de cada clase. La jerarquía de clases permanecerá pequeña y será menos probable que se formen estructuras inmanejables. Por otro lado, un diseño basado en la composición de objetos tendrá más objetos, y el comportamiento del sistema dependerá de sus interrelaciones en lugar de estar definido en una clase.

Esto nos guía hacia el segundo principio del diseño orientado a objetos: *Favorecer la composición de objetos sobre la herencia de clases.*

Idealmente, no se debería tener que crear nuevos componentes para realizar la reutilización. Debería ser posible conseguir todas las funcionalidades necesarias sólo por ensamblamiento de componentes existentes a través de la composición de objetos. Pero esto raramente ocurre, porque el conjunto de componentes disponibles nunca es lo suficientemente rico en la práctica. Reutilizar mediante la herencia hace más fácil crear nuevos componentes que pueden ser compuestos con otros. Por consiguiente, la herencia y la composición trabajan juntas.

No obstante, la experiencia dice que los diseñadores emplean demasiado la herencia como una técnica de reutilización, y los diseños, algunas veces, son más reutilizables y más simples utilizando más la composición de objetos. La composición de objetos es aplicada una y otra vez en los patrones de diseño.

2. **Delegación:** La delegación es un modo de hacer composición tan poderoso para la reutilización como la herencia. En la delegación dos objetos están involucrados en el manejo de una petición: un objeto receptor delega operaciones a su delegado. Esto indica que una clase mantiene una referencia a una instancia de otra clase.

La principal ventaja de la delegación es que hace más fácil componer comportamientos en tiempo de ejecución y cambiar la forma en que ellos están compuestos.

Varios patrones de diseño utilizan delegación. Los patrones *State*, *Strategy*, y *Visitor* dependen de esto. En el patrón *State*, un objeto delega la petición a un objeto estado que representa su estado actual. En el patrón *Strategy*, un objeto delega una petición específica a un objeto que representa la estrategia para llevar a cabo esa petición. Un objeto solamente tendrá un estado, pero éste puede tener muchas estrategias para peticiones diferentes. El propósito de ambos patrones es cambiar el comportamiento de un objeto cambiando los objetos sobre los cuales delegar la petición. En el patrón *Visitor*, la operación que consigue

realizarse sobre cada elemento de una estructura de objetos es siempre delegada al objeto *Visitor*.

Otros patrones utilizan la delegación no tan fuertemente. El patrón *Mediator* introduce un objeto para mediar la comunicación entre otros objetos. Algunas veces el objeto *Mediator* implementa operaciones simplemente para enviarlas a otros objetos; otras veces pasa una referencia a él mismo y por consiguiente utiliza la verdadera delegación. El patrón *Chain of Responsibility* maneja peticiones para enviarlas desde un objeto a otros a través de una cadena de objetos. Algunas veces esta petición lleva consigo una referencia al objeto original que recibió la petición, en cuyo caso el patrón utiliza delegación. El patrón *Bridge* desacopla una abstracción de su implementación. Si la abstracción y una implementación particular están estrechamente relacionadas, luego la abstracción podría delegar operaciones a esa implementación.

La delegación es un ejemplo extremo de la composición de objetos. Esto permite que se pueda sustituir siempre la herencia con la composición de objetos como mecanismo para reutilizar código.

- 3. Herencia frente a clases genéricas:** Las clases genéricas es otra técnica para reutilizar funcionalidades. Esta técnica nos permite definir un tipo sin especificar los otros tipos que utiliza. Los tipos no especificados son suministrados como parámetros en el punto de uso. Por ejemplo, una lista puede ser parametrizada por el tipo de elementos que contiene.

Las clases genéricas nos proporcionan una tercera forma (en adición a la herencia de clases y a la composición de objetos) para componer comportamientos en sistemas orientados a objetos. Muchos diseños pueden ser implementados utilizando alguna de estas tres técnicas.

RELACIONAR LAS ESTRUCTURAS QUE VA A HABER EN TIEMPO DE COMPILACIÓN Y EN TIEMPO DE EJECUCIÓN

La estructura de un programa orientado a objetos en tiempo de ejecución normalmente tiene poco parecido con su estructura del código. La estructura del código se termina en el tiempo de compilación y son clases en relaciones de herencia fijas. El tiempo de ejecución de un programa consiste en cambios rápidos de la comunicación entre objetos. En realidad las dos estructuras son ampliamente independientes.

Con tanta disparidad entre las estructuras en tiempo de ejecución y en tiempo de compilación esta claro que el código no revelará nada sobre cómo trabajará un sistema. Las estructuras en tiempo de ejecución del sistema deben ser impuestas por el diseñador más que por el lenguaje. Las relaciones entre los objetos y sus tipos deben ser diseñadas con mucho cuidado, porque determinan como de buena o mala es la estructura en tiempo de ejecución.

Muchos patrones de diseño capturan explícitamente la distinción entre estructuras en tiempo de compilación y en tiempo de ejecución. Los patrones *Composite* y *Decorator* son especialmente útiles para construir estructuras complejas en tiempo de ejecución. El patrón *Observer* se involucra en estructuras en tiempo de ejecución que son difíciles de comprender a menos que se conozca el patrón. El

patrón *Chain of Responsibility* también resulta en la comunicación de patrones que la herencia no revela. En general, las estructuras en tiempo de ejecución no son claras desde el código hasta que no se comprenden los patrones.

RAZONES DE CAMBIO

Cada patrón de diseño permite que algunos aspectos de la estructura del sistema varíen independientemente de otros aspectos, haciendo por tanto sistemas más robustos para un tipo particular de cambio. Alternativas eficientes para volver a diseñar la aplicación en los siguientes casos:

1. **Crear un objeto mediante la especificación de una clase:** especificando un nombre de clase cuando se crea un objeto hace que te lleve a una implementación particular en vez de a una interfaz. Este compromiso puede complicar futuros cambios. Para evitarlo, crea los objetos indirectamente.

Patrones de diseño: *Abstract Factory, Factory Method, Prototype*.

2. **Dependencia de operaciones específicas:** al especificar una operación te compromete a una única forma de satisfacer la petición que no se puede cambiar. Para evitarlo, peticiones fuertemente codificadas, se realiza tan fácil como cambiar la forma en que se satisface una petición tanto en tiempo de compilación como en tiempo de ejecución.

Patrones de diseño: *Chain of Responsibility, Command*.

3. **Dependencia de la plataforma hardware y/o software:** el software dependiente de un hardware concreto es difícil de portar a otras plataformas. Incluso podría ser difícil de mantenerlo en su plataforma original. Por eso es importante que el diseño del sistema límite sus dependencias con la plataforma.

Patrones de diseño: *Abstract Factory, Bridge*.

4. **Dependencia de la implementación o representación de un objeto:** los clientes que conocen la representación interna de un objeto pueden necesitar cambios si se modifica la representación interna de dicho objeto.

Patrones de diseño: *Abstract Factory, Bridge, Proxy*.

5. **Dependencias de algoritmos:** los algoritmos son a menudo extendidos, optimizados y reemplazados durante el desarrollo y la reutilización. Los objetos que dependen de un algoritmo tendrán que ser modificados cuando el algoritmo cambie.

Patrones de diseño: *Builder, Iterator, Strategy, Template Method, Visitor*.

6. **Gran acoplamiento entre clases:** al cambiar o eliminar una clase se deben cambiar muchas clases relacionadas, por lo que el sistema es difícil de entender, de mantener y de portar a otra plataforma. La pérdida de acoplamiento incrementa la probabilidad de que una clase pueda ser reutilizada por sí misma y que un sistema sea mas entendible, portable, modificable y extensible. Los patrones de diseño utilizan técnicas tales como el acoplamiento abstracto para promover la pérdida de acoplamiento en los sistemas.

Patrones de diseño: *Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer.*

7. **Extensión de la funcionalidad mediante herencia:** para definir una subclase se necesita un conocimiento de la clase padre con lo que se puede romper la encapsulación. La composición de objetos en general y la delegación en particular proveen alternativas flexibles a la herencia para combinar comportamientos. Pueden añadirse nuevas funcionalidades a una aplicación mediante la composición de objetos existentes mejor que definiendo nuevas subclases de clases existentes. Por otro lado, la gran utilización de la composición de objetos puede dar diseños más difíciles de entender. Muchos patrones de diseño producen diseños en los cuales se puede introducir una nueva funcionalidad definiendo solamente una subclase y componiendo sus instancias con algunas existentes.

Patrones de diseño: *Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy.*

8. **Imposibilidad de alterar las clases de forma conveniente:** a veces hay que modificar clases que no se pueden modificar convenientemente. Quizás necesitas el código fuente y no lo tienes (como puede ser el caso de una librería comercial). O a lo mejor algún cambio podría requerir la modificación de muchas subclases existentes. Los patrones de diseño ofrecen formas para modificar clases en estas circunstancias.

Patrones de diseño: *Adapter, Decorator, Visitor.*

1.3.10. Seleccionar un patrón de diseño

De entre los diferentes patrones de diseño que existen puede ser complicado la elección de uno para el diseño de una aplicación. Entre las pautas que hay que seguir a la hora de seleccionar un patrón se pueden indicar las siguientes:

1. **Observar como el patrón de diseño resuelve los problemas de diseño** (explicado en el apartado anterior, apartado 1.3.9).
2. **Revisar las secciones de “Objetivo”** (apartado 1.3.5). Puedes utilizar la clasificación mostrada en forma de tabla en el apartado 1.3.4 para facilitar la búsqueda.
3. **Estudiar la interrelación entre los diferentes patrones.** Estudiar estas relaciones puede guiarte directamente al patrón correcto o grupo de patrones.
4. **Estudiar los patrones con un propósito similar.** Hay patrones que son muy parecidos estudiar sus similitudes y sus diferencias te ayudará en la elección del patrón que mejor se adapte a tu problema.
5. **Examinar las razones de cambio.** Mirar las causas de rediseño que están en el apartado 1.3.9 para ver si tu problema esta involucrado en una o más de una. Luego mirar los patrones que te ayudan a evitar las causas de rediseño.
6. **Considerar lo que se debería cambiar en el diseño concreto.** Este apartado es el opuesto al anterior, es decir al enfocado sobre las razones de cambio. En lugar de considerar que podría forzar un cambio en un diseño, considera lo que quieres que

sea capaz de cambiar sin rediseñarlo. El objetivo aquí es encapsular el concepto que varía, un fin de muchos patrones de diseño. Son aspectos del diseño que los patrones de diseño permiten que varíen independientemente, por lo tanto puedes hacer cambios sin rediseñar.

1.3.11. Como usar un patrón de diseño

Una vez seleccionado el patrón de diseño que se va a aplicar hay que tener en cuenta las siguientes claves para utilizar el patrón:

- ✱ **Leer el patrón de diseño por encima.** Prestar una atención particular a las secciones “Aplicabilidad” y “Consecuencias” para asegurarse de que el patrón es correcto para tu problema.
- ✱ **Observar la estructura, los elementos que participan en el patrón y las colaboraciones entre ellos.** Asegurarse de que comprendes las clases y objetos del patrón y como se relacionan.
- ✱ **Mirar la sección “Código del ejemplo” para ver un ejemplo concreto del patrón en código.** Estudiar el código te enseñará como implementar el patrón.
- ✱ **Escoger nombres significativos de los elementos que participan en el patrón para el contexto de la aplicación.** Los nombres de los elementos de un patrón son normalmente demasiado abstractos para aparecer directamente en una aplicación. No obstante, es muy útil incorporar los nombres de los participantes en el nombre que aparece en la aplicación. Esto te ayudará a tener el patrón más explícito en la implementación.
- ✱ **Definir las clases.** Declarar sus interfaces, establecer sus relaciones de herencia, y definir las variables instanciadas que representan datos y referencias de objetos. Identificar las clases existentes en tu aplicación que el patrón afectará y modificará adecuadamente.
- ✱ **Definir nombres específicos para las operaciones en dicha aplicación.** Otra vez, los nombres dependen generalmente de la aplicación. Utilizar las responsabilidades y colaboraciones asociadas con cada operación como guía.
- ✱ **Implementar las operaciones que conllevan responsabilidad y colaboración en el patrón.** La sección “Implementación” te ofrece una guía en la implementación. Los ejemplos de la sección “Código del ejemplo” también te ayudan.

Estas son sólo unas directrices para empezar. Luego desarrollarás tu propia forma de trabajar con patrones de diseño.

Antes de terminar este apartado sobre *como usar patrones de diseño*, habría que decir algo sobre *como no usar los patrones de diseño*. Los patrones de diseño no se deben de aplicar indiscriminadamente. A menudo consiguen flexibilidad y variabilidad introduciendo niveles adicionales de tortuosidad, y pueden complicar un diseño y/o mayores costes de realización. Un patrón de diseño solamente debería ser aplicado cuando la flexibilidad que permite es

necesaria. La sección “Consecuencias” es de mucha ayuda para evaluar los beneficios y obligaciones de un patrón.

1.3.12. El futuro de los patrones

La gran popularidad que han ido adquiriendo los patrones hacen que la comunidad software los utilice y los soporte comúnmente. Existen tres grupos de personas que usan patrones para documentar sus procesos de desarrollo de software en la forma de un lenguaje de patrones. Varias notaciones usadas por las Tecnologías Orientadas a Objetos han añadido soporte para el modelado y representación de patrones de diseño. Muchas herramientas de desarrollo de software han añadido un soporte similar. Algunos proyectos de investigación están intentando codificar patrones de diseño con el propósito de generar código fuente. Algunas librerías comerciales de software proporcionan implementación de algunos patrones de diseño (Java utiliza unos pocos en sus librerías estándar). Se piensa que en poco tiempo los lenguajes de programación introducirán sintaxis para representar patrones como unidades de construcción de software.

Hay una especulación a cerca de si los patrones sustituirán algún día a los programadores. Esta misma especulación ya ha aparecido en algunas ocasiones con la introducción de nuevas tecnologías. Los lenguajes y las herramientas que se usan para solucionar problemas software evolucionarán, pero se seguirá necesitando desarrolladores con la misma evolución.

Aunque la habilidad de codificar patrones como componentes de software pueda llegar a ser importante, más importante será el conocimiento de cómo y cuando aplicar y combinar patrones, en conjunción con la habilidad de usar un vocabulario de nombres de patrones para comunicarse con otros desarrolladores.

1.4. ANTIPATRONES

Como en la mayor parte de las disciplinas también existen malos usos que están más o menos extendidos, y que por lo tanto no aportan soluciones efectivas a la resolución de problemas.

Si un patrón es una buena práctica, entonces un antipatrón es una mala práctica. Hay dos nociones de antipatrones:

1. Aquellos que describen una mala solución a un problema que da como resultado una mala situación.
2. Aquellos que describen como salir de una mala situación y convertirla en una buena solución.

Los antipatrones deben ser valorados porque a menudo es tan importante ver y entender malas soluciones como ver y entender las buenas.

Brown, Malveau, MacCormick y Mowbray han escrito un libro sobre los antipatrones que intenta identificar y diagnosticar varias anti-soluciones para corregirlas y prevenirlas. Los antipatrones más útiles serán aquellos que permiten rehacer un buen diseño a partir de uno malo.

Se puede definir entonces un patrón de la siguiente forma:

“Forma literaria que describe una solución comúnmente dada a un problema que genera consecuencias negativas decididamente”.

Estos antipatrones se pueden encontrar en muchas disciplinas. Ateniéndonos a la ingeniería del software se pueden clasificar los diferentes tipos de antipatrones de la siguiente forma:

Antipatrones de desarrollo de software:

1. The Blob (“clases gigantes”).
2. Lava Flow (“código muerto”).
3. Funcional Decomposition (“diseño no orientado a objetos”).
4. Poltergeists (“no se sabe bien lo que hacen algunas clases”).
5. Golden hammer (“para un martillo todo son clavos”).
6. Spaghetti code (“muchos if o switch”).
7. Cut-and-paste programming (“cortar y pegar código”).

Antipatrones de arquitectura de software:

1. Stovepipe enterprise (“aislamiento en la empresa”).
2. Stovepipe system (“aislamiento entre sistemas”).
3. Vendor Lock-In (“arquitectura dependiente de producto”).
4. Architecture by implication.
5. Design by committee (“navaja suiza”).
6. Reinvent the Wheel (“reinventar la rueda”).

Antipatrones de gestión de proyectos software:

1. Analysis paralysis.
2. Death by planning.
3. Corncob (“personas problemáticas”).
4. Irrational management.
5. Project mismanagement.

De todos estos patrones los más relacionados con los patrones de diseño son los de desarrollo de software por lo que habrá que evitarlos en la medida de lo posible. Todos los antipatrones tienen al lado una aclaración que permite intuir a lo que se refieren.

1.5. CONCLUSIÓN

Todas las disciplinas de la ingeniería desarrolladas tienen un compendio colectivo de observación, experimentación de “buenas prácticas” y “lecciones aprendidas” para resolver conocidos problemas de ingeniería. Los grandes ingenieros no solo diseñan sus productos de acuerdo estrictamente a los principios de la matemática y la ciencia. Ellos deben adaptar sus soluciones para realizar los cambios y compromisos óptimos entre las soluciones conocidas, los principios, y las restricciones que encuentran los incrementos y cambios demandados de coste, tiempo, calidad y necesidades de los clientes. Los patrones nos ayudan a conseguir la formación para identificar que es constante y reconocer los cambios incesantes. En este sentido, los patrones realizan la convergencia de las interacciones dinámicas de componentes en configuraciones estables, que se solucionan a través de sistemas exitosos.

Los patrones representan la experiencia, la cual, a través de su asimilación, imparte la perspicacia y el conocimiento de los diseñadores expertos. Los patrones ayudan a amoldar la visión compartida de la arquitectura, y colección de estilos. Si queremos diseñar software que se desenvuelve en una disciplina de ingeniería desarrollada, entonces está provee las “mejores prácticas” y “lecciones aprendidas” que deben ser suficientemente y formalmente documentadas, compiladas, escrutadas y ampliamente difundidas como patrones (y antipatrones). Una vez que una solución ha sido expresada en forma de patrón, está podría ser aplicada y reaplicada a otros contextos, y facilitar ampliamente la reutilización a través de todos los utensilios del espectro de la ingeniería de software tales como: análisis, arquitecturas, diseños, implementaciones, algoritmos y estructuras de datos, tests, planes y organización de estructuras.

Los patrones son herramientas válidas para capturar y comunicar los conocimientos y experiencia adquiridos para proporcionar calidad y productividad software, para solucionar las cuestiones fundamentales en el desarrollo de software. Al utilizar estas herramientas estamos mejor adaptados a solucionar retos como “comunicación del conocimiento de arquitecturas entre diseñadores; acomodando un nuevo paradigma de diseño o estilo arquitectónico; resolviendo objetivos no funcionales tales como la reutilización, portabilidad, y extensibilidad; y evitando errores y dificultades del diseño, que ha sido aprendido tradicionalmente solamente a través de las experiencia”.

Los patrones exponen conocimiento sobre la construcción de software que ha sido fruto de muchos expertos durante muchos años. Todo trabajo sobre patrones debería concentrarse en aplicarse ampliamente estos valiosos recursos. Todo diseñador de software debería ser capaz de utilizar patrones correctamente cuando construye sistemas software. Una vez archivados, seremos capaces de ver la inteligencia humana que reflejan los patrones, cada patrón por individual y todos los patrones en su conjunto.

2. EXPLICACIÓN DETALLADA DE LOS PATRONES DE DISEÑO UTILIZADOS EN LOS EJEMPLOS

2.1. SINGLETON [GoF95]

2.1.1. Objetivo

Garantiza que solamente se crea una instancia de la clase y provee un punto de acceso global a él. Todos los objetos que utilizan una instancia de esa clase usan la misma instancia.

2.1.2. Aplicabilidad

El patrón Singleton se puede utilizar en los siguientes casos:

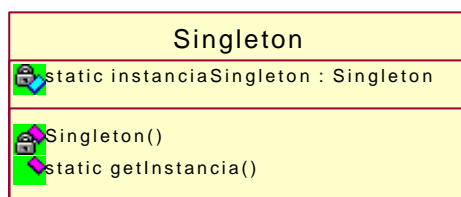
- Debe haber exactamente una instancia de la clase.
- La única instancia de la clase debe ser accesible para todos los clientes de esa clase.

2.1.3. Solución

El patrón Singleton es relativamente simple, ya que sólo involucra una clase.

Una clase Singleton tiene una variable static que se refiere a la única instancia de la clase que quieres usar. Esta instancia es creada cuando la clase es cargada en memoria. Tú deberías de implementar la clase de tal forma que se prevenga que otras clases puedan crear instancias adicionales de una clase Singleton. Esto significa que debes asegurarte de que todos los constructores de la clase son privados.

Para acceder a la única instancia de una clase Singleton, la clase proporciona un método static, normalmente llamado *getInstancia*, el cual retorna una referencia a la única instancia de la clase.



Clase Singleton.

2.1.4. Consecuencias

- Existe exactamente una instancia de una clase Singleton.
- Otras clases que quieran una referencia a la única instancia de la clase Singleton conseguirán esa instancia llamando al método static *getInstancia* de la clase. Controla el acceso a la única instancia.
- Tener subclases de una clase Singleton es complicado y resultan clases imperfectamente encapsuladas. Para hacer subclases de una clase Singleton, deberías tener un constructor que no sea privado. También, si quieres definir una subclase de una clase Singleton que sea también Singleton, querrás que la subclase sobrescriba el método *getInstancia* de la clase Singleton. Esto no será posible, ya que métodos como *getInstancia* deben ser static. Java no permite sobrescribir los métodos static.

2.1.5. Implementación

Para forzar el carácter de una clase Singleton, debes codificar la clase de tal forma que prevengas que otras clases creen directamente instancias de la clase. El modo de realizarlo es declarar todos los constructores de la clase privados. Tener cuidado de declarar al menos un constructor privado. Si una clase no declara ningún constructor, entonces un constructor público es automáticamente generado por ella.

Una variación común del patrón Singleton ocurre en situaciones donde la instancia de un Singleton podría no ser necesaria. En estas situaciones, puedes posponer la creación de la instancia hasta la primera llamada a *getInstancia*.

Otra variación sobre el patrón Singleton descende del hecho de que la política de instanciación de una clase está encapsulada en el método de la clase *getInstancia*. Debido a esto, es posible variar la política de creación. Una posible política es tener el método *getInstancia* que retorne alternativamente una de las dos instancias o crear periódicamente una nueva instancia para ser retornada por *getInstancia*. Es decir se puede permitir un número variable de instancias.

2.1.6. Usos en el API de Java

En el API de Java la clase *java.lang.Runtime* es una clase Singleton, tiene exactamente

una única instancia de la clase. No tiene constructores públicos. Para conseguir una referencia a su única instancia, otras clases deben llamar a su método static *getRuntime*.

2.1.7. Patrones relacionados

Puedes utilizar el patrón Singleton con muchos otros patrones. En particular, es a menudo utilizado con los patrones Abstract Factory, Builder, y Prototype.

El patrón Singleton tiene alguna similitud con el patrón Cache Management. Un Singleton es funcionalmente similar a un Cache que contiene solamente un objeto.

2.2. ITERATOR [GoF95]

2.2.1. Objetivo

Define un interface que declara métodos para acceder secuencialmente a los objetos de una colección. Una clase que accede a una colección solamente a través de un interface independiente de la clase que implementa el interface.

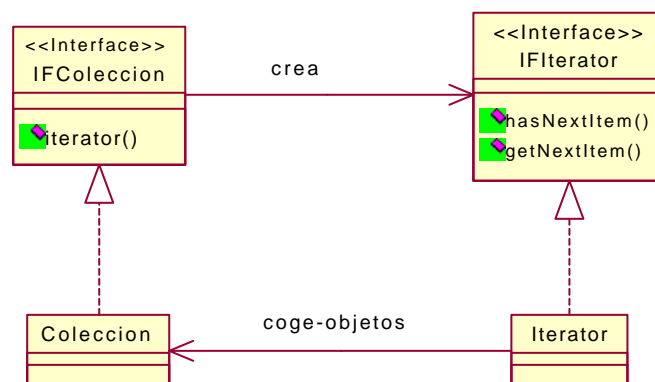
2.2.2. Aplicabilidad

El patrón Iterator se puede utilizar en los siguientes casos:

- Una clase necesita acceder al contenido de una colección sin llegar a ser dependiente de la clase que es utilizada para implementar la colección, es decir sin tener que exponer su representación interna.
- Una clase necesita un modo uniforme de acceder al contenido de varias colecciones.
- Cuando se necesita soportar múltiples recorridos de una colección.

2.2.3. Solución

El diagrama de clases que muestra la organización de las clases e interfaces que participan en el patrón Iterator es el siguiente:



Patrón Iterator.

A continuación están las descripciones de los papeles que juegan las clases e interfaces en la organización citada anteriormente:

Coleccion Una clase en este papel encapsula una colección de objetos o valores.

IFIterator	Un interface en este papel define los métodos que acceden secuencialmente a los objetos que están encapsulados en un objeto <i>Coleccion</i> .
Iterator	Una clase en este papel implementa un interface <i>IFIterator</i> . Sus instancias proporcionan acceso secuencial a los contenidos del objeto <i>Coleccion</i> asociado con el objeto <i>Iterator</i> .
IFColeccion	Las clases <i>Coleccion</i> normalmente toman responsabilidad para crear sus propios objetos <i>Iterator</i> . Es conveniente tener un modo consistente de preguntar a un objeto <i>Coleccion</i> para crear un objeto <i>Iterator</i> por sí mismo. Para proporcionar esta consistencia, todas las clases <i>Coleccion</i> implementan un interface <i>IFColeccion</i> que declara un método para crear objetos <i>Iterator</i> .

2.2.4. Consecuencias

Es posible acceder a una colección de objetos sin conocer el código de los objetos.

Utilizando varios objetos *Iterator*, es simple tener y manejar varios recorridos al mismo tiempo, puesto que cada *Iterator* mantiene la pista de su recorrido.

Es posible para una clase *Coleccion* proporcionar diferentes tipos de objetos *Iterator* que recorran la colección en diferentes modos. Por ejemplo, una clase *Coleccion* que mantiene una asociación entre la clave de los objetos y el valor de los objetos podría tener diferentes métodos para crear *Iterators* que recorran sólo la clave de los objetos y crear *Iterators* que recorran sólo el valor de los objetos.

Las clases *Iterator* simplifican el código de las colecciones, ya que el código de los recorridos se encuentra en los *Iterators* y no en las colecciones.

2.2.5. Implementación

El interface *IFIterator* mostrado en el diagrama de clases del patrón Iterator contiene un conjunto mínimo de métodos. Es común para los interfaces *IFIterator* definir métodos adicionales cuando son útiles y son soportados por las clases *Coleccion*. En adición a los métodos para preguntar la existencia y coger el siguiente elemento de la colección, son comunes los siguientes métodos:

- Para preguntar la existencia y coger el elemento anterior de la colección.
- Moverse al primero o al último elemento de la colección.
- Coger el número de elementos del recorrido.

En muchos casos, un algoritmo de recorrido de una clase *Iterator* requiere el acceso a la estructura de datos interna de una clase *Coleccion*. Por esta razón, las clases *Iterator* son a menudo implementadas como una clase interna privada de la clase *Coleccion*.

Un *Iterator Null* es un *Iterator* que no retorna objetos. Su método *hasNext* siempre retorna false. Los *Iterators Null* son normalmente implementados como una simple clase que implementa el apropiado interface *IFIterator*. La utilización de *Iterators Null* puede simplificar la implementación de clases *Coleccion* y otras clases *Iterator* para eliminar la necesidad de que algún código necesitare el manejo especial de un recorrido null.

Las modificaciones para un objeto *Coleccion* mientras un *Iterator* recorre su contenido puede causar problemas. Si las provisiones no son realizadas de acuerdo con tales modificaciones, un *Iterator* podría retornar un conjunto inconsistente de resultados. Tales potenciales inconsistencias incluyen saltarse objetos o retornar el mismo dos veces.

El modo más simple de manejar las modificaciones de una colección durante un recorrido por un *Iterator* es considerar que el *Iterator* es invalido después de la modificación. Puedes implementar esto teniendo cada una de las clases *Coleccion* métodos que modifiquen una colección incrementando un contador cuando ellos modifican la colección. Los objetos *Iterator* pueden luego detectar un cambio en su colección subyacente notando un cambio en su contador de cambios. Si uno de los métodos del objeto *Iterator* es llamado y nota que la colección subyacente ha cambiado, luego puede lanzar una excepción.

Una forma más robusta de manejar las modificaciones de una colección durante un recorrido por un *Iterator* es asegurarse de que el *Iterator* retorna un conjunto consistente de resultados. Hay varias formas de hacerlo. Por ejemplo tener una copia completa de la colección, normalmente es la técnica menos deseable porque es la más cara en tiempo y memoria.

2.2.6. Usos en el API de Java

Las clases colección en el paquete *java.util* siguen el patrón *Iterator*. El interface *java.util.Collection* juega el papel de *IFColeccion*. El paquete incluye un número de clases que implementan el *java.util.Collection*.

El interface *java.util.Iterator* juega el papel del *IFIterator*. Las clases en el paquete que implementan *java.util.Collection* definen las clases internas privadas que implementan *java.util.Iterator* y juegan el papel *Iterator*.

2.2.7. Patrones relacionados

Adapter El patrón *Iterator* es una forma especializada del patrón *Adapter* para acceder secuencialmente a los contenidos de una colección de objetos.

Factory Method Algunas clases *Coleccion* podrían usar el patrón *Factory Method* para determinar que tipo de *Iterator* instanciar.

Null Object *Iterators* Null son algunas veces utilizados para implementar el patrón Null Object.

2.3. STRATEGY [GoF95]

2.3.1. Objetivo

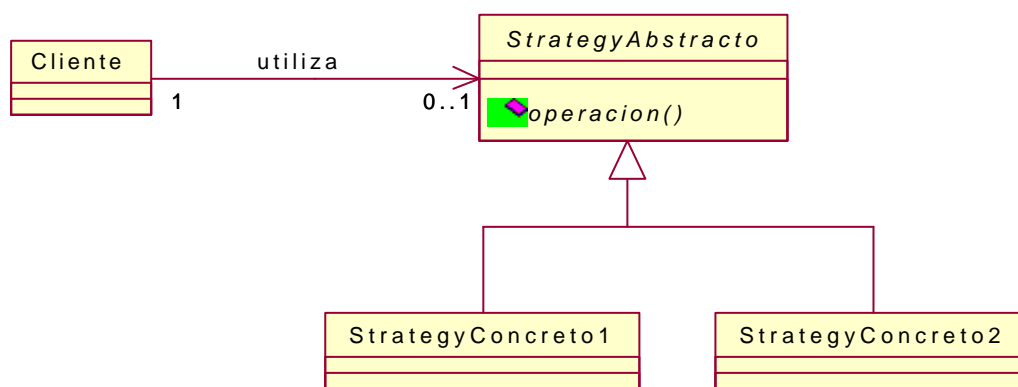
Encapsula algoritmos relacionados en clases que son subclasses de una superclase común. Esto permite la selección de un algoritmo que varía según el objeto y también le permite la variación en el tiempo.

2.3.2. Aplicabilidad

- Un programa tiene que proporcionar múltiples variantes de un algoritmo o comportamiento.
- Puedes encapsular las variantes de comportamiento en clases separadas que proporcionan un modo consistente de acceder a los comportamientos.
- Poner estos comportamientos en clases separadas significa que las clases que utilizan estos comportamientos no necesitan conocer nada sobre como están implementados. Estas clases tienen una superclase común o interfaz que permite que las clases que las utilizan ignoren como seleccionar un comportamiento o que comportamiento es seleccionado.

2.3.3. Solución

El diagrama de clases que muestra la organización de las clases que participan en el patrón Strategy es el siguiente:



Patrón Strategy.

A continuación están las descripciones de los papeles que juegan las clases en la organización citada anteriormente:

- Cliente** Una clase en el papel de *Cliente* delega una operación a una clase abstracta o interface. Esto hace que sin conocer la clase actual del objeto delegue la operación hacia o como esa clase implementa la operación.
- StrategyAbstracto** Una clase en este papel proporciona una forma común para acceder a la operación encapsulada por sus subclases. También se puede utilizar un interface para este papel.
- StrategyConcreto1, StrategyConcreto2** Las clases en este papel implementan diferentes implementaciones de la operación que la clase *Cliente* delega.

El patrón Strategy siempre sucede con un mecanismo para determinar el objeto *StrategyConcreto* que el objeto *Cliente* utilizará. Sin embargo, el mecanismo actual varía tanto que ningún mecanismo particular es incluido en el patrón.

2.3.4. Consecuencias

El patrón Strategy permite que los comportamientos de los objetos *Cliente* sean determinados dinámicamente sobre un objeto base.

El patrón Strategy simplifica los objetos *Cliente* para deducirlos de alguna responsabilidad para seleccionar comportamientos o implementaciones de comportamientos alternativos. Esto simplifica el código de los objetos *Cliente* eliminando las expresiones *if* y *switch*. En algunos casos, esto puede incrementar también la velocidad de los objetos *Cliente* porque ellos no necesitan perder tiempo seleccionado un comportamiento.

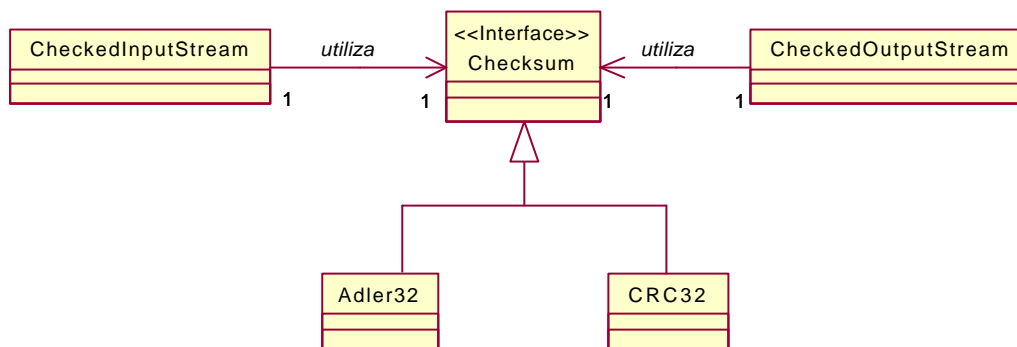
2.3.5. Implementación

Es común para las clases *StrategyConcreto* compartir algún comportamiento común. Deberías factorizar el comportamiento común que comparten en una superclase común.

Podría haber situaciones en las cuales ninguno de los comportamientos encapsulados en las clases *StrategyConcreto* es apropiado. Una forma común de manejar estas situaciones es que el objeto *Cliente* tenga un null en lugar de una referencia a un objeto *Strategy*. Esto significa tener que chequear por un null antes de llamar a un método de un objeto *Strategy*. Si la estructura de un objeto *Cliente* tiene este inconveniente, considerará utilizar el patrón Null Object.

2.3.6. Usos en el API de Java

El paquete `java.util.zip` contiene algunas clases que utilizan el patrón Strategy. Las clases `CheckedInputStream` y `CheckedOutputStream` utilizan ambas el patrón Strategy para procesar checksums sobre streams de bytes. Estas dos clases participan ambas como clases *Cliente*. Los constructores para ambas clases tienen un argumento *Checksum*. *Checksum* es un interface que participa en el papel de *StrategyAbstracto*. Las dos clases que implementan el interface *Checksum* son: `Adler32` y `CRC32`. Estas clases participan en el papel de *StrategyConcreto*. El siguiente diagrama representa las relaciones entre estas clases:



Clases relacionadas con Checksum.

2.3.7. Patrones relacionados

Adapter El patrón Adapter es estructuralmente similar al patrón Strategy. La diferencia esta en el objetivo. El patrón Adapter permite a un objeto *Cliente* sacar su función pretendida originalmente mediante llamadas a métodos de objetos que implementa un interface particular. El patrón Strategy proporciona objetos que implementa un interface particular con el propósito de alterar o determinar el comportamiento de un objeto *Cliente*.

Flyweight Si hay muchos objetos *Cliente*, los objetos *StrategyConcreto* pueden estar mejor implementados como Flyweights.

Null Object El patrón Strategy es a menudo utilizado con el patrón Null Object.

Template Method El patrón Template Method maneja comportamientos alternativos a través de subclases más que a través de delegación.

2.4. OBSERVER [GoF95]

2.4.1. Objetivo

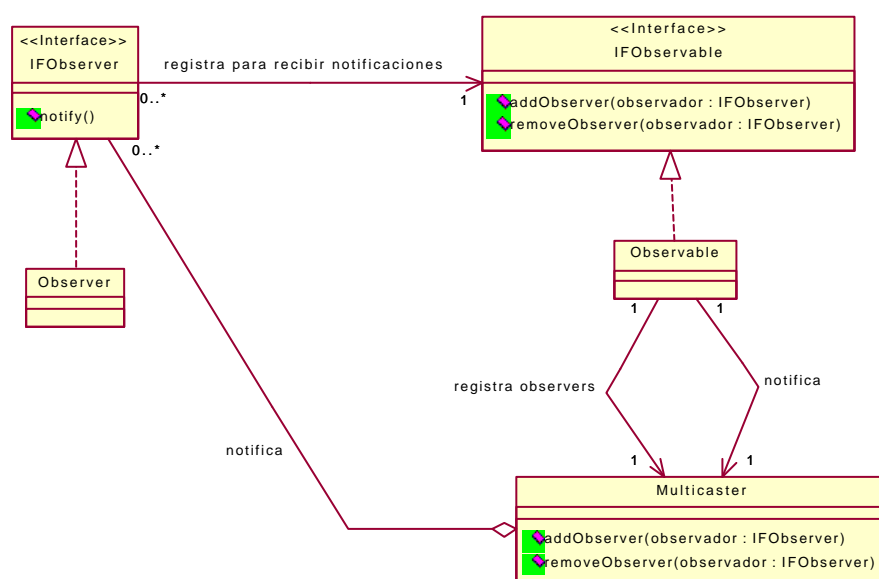
Permite a los objetos captar dinámicamente las dependencias entre objetos, de tal forma que un objeto notificará a los objetos dependientes de él cuando cambia su estado, siendo actualizados automáticamente.

2.4.2. Aplicabilidad

- Cuando una abstracción tiene dos aspectos, uno dependiente de otro. Una instancia de una clase necesitará notificar a otros objetos cuando cambia su estado. La instancia de la otra clase necesitará ser notificada cuando un objeto sobre el que tiene dependencia cambia su estado. Encapsular estos aspectos en objetos diferentes permite variarlos y reutilizarlos independientemente.
- Cuando tienes una relación de dependencia de uno a muchos que puede requerir que un objeto notifique a múltiples objetos que dependen de él cuando cambia su estado.

2.4.3. Solución

El diagrama de clases que muestra la organización de las clases e interfaces que participan en el patrón Observer es el siguiente:



Patrón Observer.

A continuación están las descripciones de los papeles que juegan las clases e interfaces en la organización citada anteriormente:

- IFObserver** Un interface en este papel define un método que es normalmente llamado *notifica* o *actualiza*. Un objeto *Observable* llama a este método para proporcionar una notificación de que su estado ha cambiado, pasándole los argumentos apropiados. En muchos casos, una referencia al objeto *Observable* es uno de los argumentos que permite al método conocer que objeto proporciona la notificación.
- Observer** Las instancias de clases en este papel implementan el interface *IFObserver* y reciben notificaciones de cambio de estado de los objetos *Observable*.
- IFObservable** Los objetos *Observable* implementan un interface en este papel. El interface define dos métodos que permiten a los objetos *Observer* registrarse y desregistrarse para recibir notificaciones.
- Observable** Una clase en este papel implementa el interface *IFObservable*. Sus instancias son las responsables de manejar la inscripción de objetos *IFObserver* que quieren recibir notificaciones de cambios de estado. Sus instancias son también responsables de distribuir las notificaciones. La clase *Observable* no implementa directamente estas responsabilidades. En lugar de eso, delega estas responsabilidades a un objeto *Multicaster*.
- Multicaster** Las instancias de una clase en este papel manejan la inscripción de objetos *IFObserver* y enviarles notificaciones por medio de un objeto *Observable*. La delegación de estas responsabilidades hacia una clase *Multicaster* permite que sus implementaciones sean reutilizadas por todas las clases *Observable* que implementan el mismo interface *IFObservable* o enviar notificaciones a objetos que implementan el mismo interface *IFObserver*.

La siguiente figura resume las colaboraciones entre los objetos que participan en el patrón Observer:

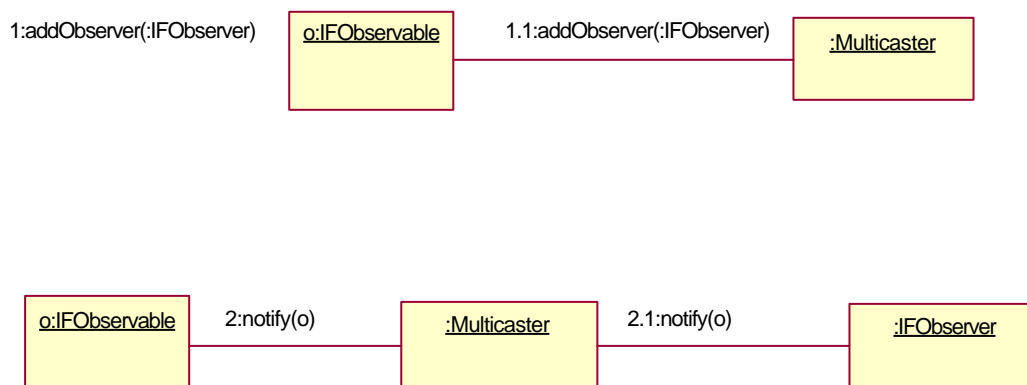


Diagrama de colaboración.

Una descripción más detallada de las interacciones mostrada en la figura anterior es la siguiente:

1. Los objetos que implementan un interface *IObserver* son pasados al método *addObserver* de un objeto *ObsevableIF*.
 - 1.1.El objeto *IObservable* delega la llamada a *addObserver* a su objeto asociado *Multicaster*. Este añade el objeto *IObservable* a la colección de objetos *IObserver* que él mantiene.
2. El objeto *IObservable* necesita notificar a otros objetos que son dependientes de él que su estado ha cambiado. Este objeto inicializa la notificación llamando al método *notify* de su objeto asociado *Multicaster*.
 - 2.1.El objeto *Multicaster* llama al método *notify* de cada uno de los objetos *IObserver* de su colección.

2.4.4. Consecuencias

El patrón Observer permite a un objeto enviar notificaciones a otros objetos sin que los objetos emisores o los objetos receptores de las notificaciones sean conscientes de las clases de los otros.

Hay algunas situaciones en las que el patrón Observer puede tener imprevistos e indeseables resultados:

- El envío de notificaciones puede durar bastante tiempo si un objeto tiene un gran número de objetos a los que enviar notificaciones. Esto puede suceder porque un objeto tiene muchos observadores registrados directamente para recibir sus notificaciones. Esto también puede suceder porque un objeto tiene muchos observadores indirectamente porque sus notificaciones son en cascada por otros objetos.
- Un problema más serio ocurre si hay dependencias en ciclo. Los objetos llaman cada uno a los métodos *notify* de los otros hasta que se llena la pila y se lanza una excepción *StackOverflowError*. Pensando seriamente, este problema puede ser fácilmente solucionado añadiendo una variable booleana interno a una de las clases involucradas en el ciclo que detecte una notificación recursiva, sería algo así:

```
private boolean inNotify=false;
public void notify (IObservable source)
{
    if (inNotify)
        return;
    inNotify=true;
    .....
```

```
inNotify=false;  
} //notify (IObservable)
```

Cuando un objeto *Observer* recibe una notificación, este conoce que objeto ha cambiado, pero no conoce de que forma ha cambiado. Evitar requerir un objeto *Observer* para determinar que atributos de un objeto *IObservable* ha cambiado. Esto es más simple para un observador actuar sobre todos los atributos de los objetos *IObservable* más bien que ir al problema de determinar cuales han cambiado y luego actuar sobre ellos.

2.4.5. Implementación

Un objeto *Observable* normalmente pasa una referencia a sí mismo como parámetro aun método *notify* de un objeto *Observer*. En la mayoría de los casos, el objeto *Observer* necesita acceder a los atributos del objeto *Observable* para actuar sobre la notificación. Aquí están algunos modos de proporcionar este acceso:

- Añadir métodos al interface *IObservable* para retornar valores de los atributos. Esta es normalmente la mejor solución. Sin embargo, solamente funciona si todas las clases que implementan el interface *IObservable* tiene un conjunto común de atributos para que los objetos *Observer* actúen sobre las notificaciones.
- Puedes tener múltiples interfaces *IObservable*, con cada uno proporcionas acceso a los atributos para que los objetos *Observer* actúen sobre las notificaciones. Para hacer este trabajo, los interfaces *IObservable* deben declarar una versión de su método *notify* para cada uno de los interfaces *ObservableIF*. Está no es una solución muy buena.
- Puedes pasar los atributos que necesitan los objetos *IObservable* como parámetros a sus métodos *notify*. La principal desventaja de esta solución es que requiere objetos *Observable* que conozcan bastante sobre los objetos *IObservable* para proporcionarlos con los valores correctos de los atributos.
- Puedes distribuir con el interface *IObservable* y pasar los objetos *Observable* a los objetos *IObservable* como instancias de su clase actual. Esto implica sobrecargar el método *notify* del interface *IObservable*, así que hay un método *notify* para cada clase *Observable* que enviará notificaciones a los objetos *IObservable*. La principal desventaja de este enfoque es que las clases *Observer* deben conocer las clases *Observable* que distribuirán notificaciones a sus instancias y conocer como sacar los atributos que necesita de ellas. Por otro lado, si solamente una clase *Observable* enviará notificaciones a las clases *Observer*, entonces está es la mejor solución. Esto no añade complejidad a ninguna de las clases. Sustituye una dependencia sobre un único interface por una dependencia sobre una única clase. Entonces esto simplifica el diseño eliminando el interface *IObservable*.

Otra simplificación que se hace frecuentemente en el patrón *Observer* es eliminar la clase *Multicaster*. Si una clase *Observable* es la única clase que envía notificaciones a los objetos que implementan un interface particular, entonces no hay necesidad para utilizar una clase *Multicaster*. Otra razón de no tener una clase *Multicaster* es que un objeto *Observable*

nunca tendrá que enviar notificaciones a más de un objeto. En este caso, el manejo y envío de notificaciones a objetos *Observer* es tan simple que un clase *Multicaster* añade más complejidad que beneficio.

Podría no ser necesario o útil notificar a los objetos *Observer* de cada cambio a un objeto *Observable*. Si este es el caso, puedes evitar las innecesarias notificaciones de cambios de estado y esperar hasta que un lote de cambios este completo para enviar las notificaciones. Si otro objeto realiza cambios en un estado de objeto *Observable*, entonces proporcionar una única notificación para un lote de cambios es más complicado. Tendrás que añadir un método a la clase de objetos *Observable* que otros objetos puedan llamar para indicar el comienzo de un lote de cambios de estado. Cuando un cambio de estado es parte de un lote, este no debería causar que el objeto envíe ninguna notificación a sus observers registrados. También tendrás que añadir un método a la clase de objetos *Observable* que otros objetos llamen para indicar el fin del lote de cambios de estado. Cuando este método es llamado, si ningún cambio de estado ha ocurrido desde el principio del lote, el objeto debería enviar notificaciones a sus observers registrados.

Si múltiples objetos inicializarán cambios de estado de un objeto *Observable*, entonces determinara el fin de un lote de cambios podría ser más complicado. Una buena forma de manejar esta complejidad es añadir un objeto adicional que coordine los cambios de estado inicializados por otros objetos y comprendiendo su lógica bien es suficiente para determinar el fin de un lote de cambios.

El patrón Observer es normalmente utilizado para notificar a otros objetos que un estado de un objeto ha cambiado. Una variación común de esto es definir un interface *IFObservable* alterno que permita a los objetos que demandan que ellos reciban una notificación antes de un cambio de estado. La razón normal para enviar notificaciones de cambios de estado después de un cambio de estado es para permitir que el cambio se propague a otros objetos. La razón normal para enviar una notificación antes de un cambio de estado es que otros objetos pueden vetar un cambio de estado. La forma normal de implementar esto es teniendo un objeto que lanza una excepción para prevenir un cambio de estado pensado.

2.4.6. Usos en el API de Java

El modelo de delegación de eventos en Java es una forma especializada del patrón Observer. Las clases cuyas instancias pueden ser fuente de eventos participan en el papel de *Observable*. Los interfaces escuchadores de eventos participan en el papel *IFObserver*. Las clases que implementan los interfaces escuchadores de eventos participan en el papel *Observer*. Como hay un número de clases que envían varias subclases de *java.awt.AwtEvent* a sus escuchadores, hay una clase *Multicaster* que ellos utilizan llamada *java.awt.AWTEventMulticaster*.

2.4.7. Patrones relacionados

Adapter El patrón Adapter puede ser utilizado para permitir a los objetos que no implementen el interface requerido participar en el patrón Observer para recibir notificaciones.

Delegation El patrón Observer utiliza el patrón Delegation.

Mediator El patrón Mediator es utilizado algunas veces para coordinar cambios de estado inicializados por múltiples objetos a un objeto *Observable*.

2.5. PROTOTYPE [GoF95]

2.5.1. Objetivo

Permite a un objeto crear objetos personalizados sin conocer su clase exacta o los detalles de cómo crearlos. Su tarea es dar objetos prototipo a un objeto que inicializa la creación de objetos. El objeto de creación e inicialización crea objetos mandando a sus objetos prototipo que hagan una copia de si mismos.

2.5.2. Aplicabilidad

- Un sistema debe ser capaz de crear objetos sin conocer su clase exacta, como son creados, o que datos representan.
- Las clases serán instanciadas sin ser conocidas por el sistema hasta el tiempo de ejecución.
- Los siguientes métodos para la creación de una gran variedad de objetos son indeseables:

Las clases que inicializan la creación de objetos crean directamente los objetos. Esto hace que ellos sean conscientes y dependientes de un gran número de otras clases.

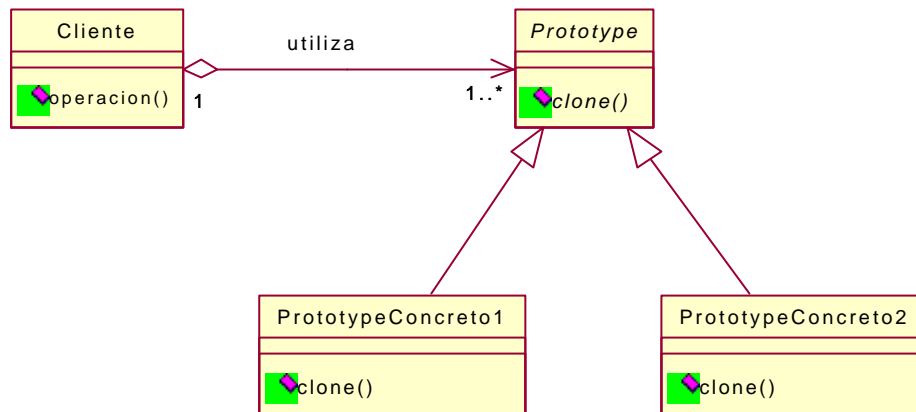
Las clases que inicializan la creación de objetos crean los objetos indirectamente a través de una clase factory method. Un factory method que es capaz de crear una gran variedad de objetos puede ser muy grande y difícil de mantener.

Las clases que inicializan la creación de objetos crean los objetos indirectamente a través de una clase abstracta factory. Para que un abstracta factory sea capaz de crear una gran variedad de objetos debe tener una gran variedad de clases factory concretas en una jerarquía semejante a las clases que deben ser instanciadas.

Los objetos diferentes que un sistema debe crear deben ser instancias de la misma clase que contienen diferentes estados de información o datos.

2.5.3. Solución

El diagrama de clases que muestra la organización de las clases e interfaces que participan en el patrón Prototype es el siguiente:



Patrón Prototype.

A continuación están las descripciones de los papeles que juegan las clases e interfaces en la organización citada anteriormente:

- | | |
|---|---|
| Cliente | La clase <i>Cliente</i> crea nuevos objetos pidiendo al prototipo que se clone. |
| Prototype | Declara la interface del objeto que se clona. Suele ser una clase abstracta. |
| PrototypeConcreto1, PrototypeConcreto2 | Las clases en este papel implementan una operación por medio de la clonación de sí mismo. |

2.5.4. Consecuencias

- Un programa puede dinámicamente añadir y borrar objetos prototipo en tiempo de ejecución. Esta es una ventaja que no ofrece ninguno de los otros patrones de creación.
- Esconde los nombres de los productos específicos al cliente.
- El objeto cliente puede también ser capaz de crear nuevos objetos de tipos prototipo. Se pueden especificar nuevos objetos prototipo variando los existentes.
- La clase *Cliente* es independiente de las clases exactas de los objetos prototipo que utiliza. También, la clase *Cliente* no necesita conocer los detalles de cómo construir los objetos prototipo.

- Los objetos de prototipo concretos implementan el interface *Prototype* , de esta forma el patrón Prototype se asegura de que los objetos prototipo proporcionan un conjunto consistente de métodos para que los objetos clientes los utilicen.
- Se puede configurar una aplicación cargando clases dinámicamente.
- No hay necesidad de organizar los objetos prototipos en ningún orden de jerarquía de clases.
- Las subclasses son reducidas.

2.5.5. Implementación

Todas las clases en Java heredan un método de la clase *Object* llamado *clone*. Un método *clone* de un objeto retorna una copia de ese objeto. Esto solamente se hace para instancias de clases que dan permiso para ser clonadas. Una clase da permiso para que su instancia sea clonada si, y solo si, ella implementa el interface *Cloneable*.

Si va a variar el número de prototipos se puede utilizar un administrador de prototipos.

Cómo implementar la operación *clone* de los objetos prototipo es otra importante característica de la implementación. Hay dos estrategias básicas para implementar la operación *clone*:

1. **Copia superficial** significa que las variables de los objetos clonados contienen los mismos valores que las variables del objeto original y que todas las referencias al objeto son a los mismos objetos. En otras palabras, la copia superficial copia solamente el objeto que será clonado, no los objetos a los que se refiere.
2. **Copia profunda** significa que las variables de los objetos clonados contienen los mismos valores que las variables del objeto original, excepto que estas variables que se refieren a objetos realizan copias de los objetos referenciados por el objeto original. En otras palabras, la copia profunda copia el objeto que será clonado y los objetos a los que se referencia. Implementar la copia profunda puede ser delicado. Necesitarás decidir si quieres hacer copia profunda o superficial de los objetos copiados indirectamente. También necesitarás ser cuidadoso con el manejo de las referencias circulares.

La copia superficial es más fácil de implementar porque todas las clases heredan un método *clone* de la clase *Object* que hace esto. Sin embargo, a menos que una clase de objetos implemente el interface *Cloneable*, el método *clone* rechazará trabajar. Si todos los objetos prototipo que tu programas utilizarán clonarse a sí mismos por copia superficial, puedes ahorrar tiempo declarando un interface *Prototype* que extienda el interface *Cloneable*. De esta forma todas las clases que implementen el interface *Prototype* también implementarán el interface *Cloneable*.

2.5.6. Usos en el API de Java

El patrón Prototype es la esencia de los JavaBeans. Los JavaBeans son instancias de clases que conforma ciertas convenciones de nombres. Las convenciones de nombres permiten a un bean crear un programa conociendo como particularizarlos. Después de que un objeto bean ha sido particularizado para su uso en una aplicación, el objeto es guardado en un fichero para ser cargado por la aplicación mientras se esta ejecutando. Salvar un objeto a un fichero para ser cargado más tarde por otra aplicación es un modo de posponer en el tiempo la creación de objetos.

2.5.7. Patrones relacionados

Composite El patrón Prototype es utilizado a menudo con el patrón Composite.

Abstract Factory El patrón Abstract Factory puede ser una buena alternativa al patrón Prototype donde los cambios dinámicos que el patrón Prototype permite para los objetos prototipo no son necesarios. Pueden competir en su objetivo, pero también pueden colaborar entre sí.

Facade La clase cliente normalmente actúa comúnmente como un facade que separa las otras clases que participan en el patrón Prototype del resto del programa.

Factory Method El patrón Factory Method puede ser una alternativa al patrón Prototype cuando la paleta de objetos prototipo nunca contiene más de un objeto.

Decorator El patrón Prototype es utilizado a menudo con el patrón Decorator.

2.6. COMPOSITE [GoF95]

2.6.1. Objetivo

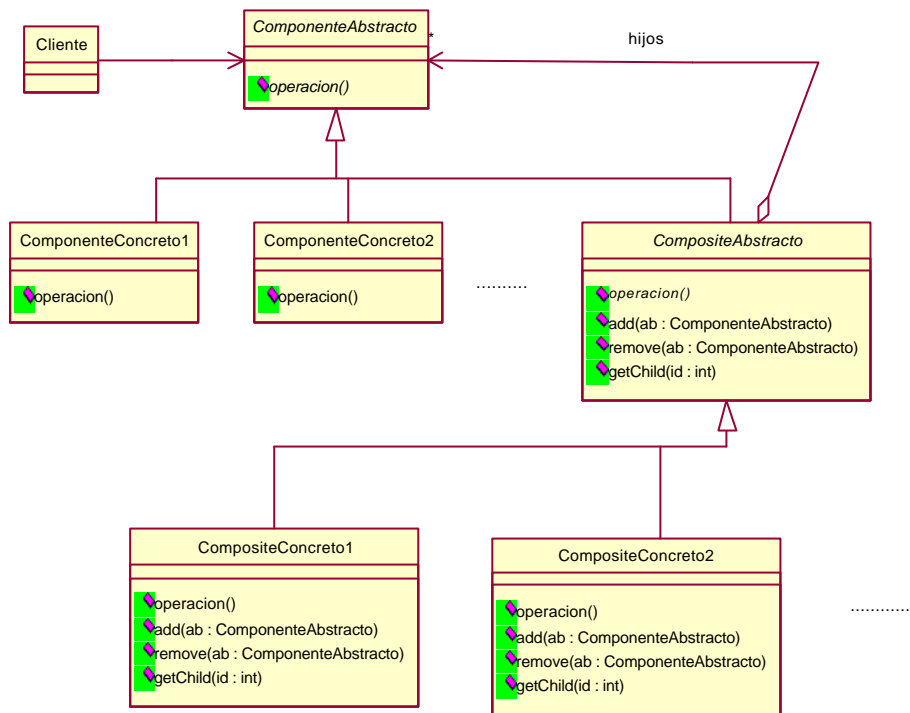
Permite construir objetos complejos mediante composición recursiva de objetos similares. El patrón Composite también permite que los objetos del árbol sean manipulados por un manejador consistente, para requerir todos los objetos hay una superclase o un interfaz común. Permite a los clientes tratar de la misma manera tanto a objetos individuales como a compuestos.

2.6.2. Aplicabilidad

- Tienes un objeto complejo que quieres descomponer en una jerarquía de objetos.
- Quieres minimizar la complejidad del conjunto de la jerarquía minimizando el número de tipos diferentes de objetos hijo en el árbol que los objetos compuestos necesitan para formarse.
- Los clientes no conocen la diferencia entre los objetos inidividuales y los grupos.

2.6.3. Solución

Minimiza la complejidad de un objeto composite organizado en jerarquias parte-todo proporcionando una superclase para todos los objetos de la jerarquía y una superclase abstracta para todos los composites de la jerarquia. A continuación se muestra el diagrama de clases del patrón Composite:



Patrón Composite.

Ahora mostramos las descripciones de las clases que participan en este patrón:

ComponenteAbstracto El *ComponenteAbstracto* es una clase abstracta y la superclase común de todos los objetos que están en el árbol de objetos para formar un objeto composite. Los objetos composite normalmente tratan a los objetos que ellos contienen como instancias del *ComponenteAbstracto*. Los clientes de los objetos compositen los tratan normalmente como instancias del *ComponenteAbstracto*.

ComponenteConcreto1, ComponenteConcreto2, Las instancias de estas clases son utilizadas como hojas en la organización del árbol. Una hoja no tiene hijos. Definen el comportamiento de los objetos primitivos.

CompositeAbstracto El *CompositeAbstracto* es la superclase abstracta de todos los objetos composite que participan en el patrón Composite. Define el comportamiento de los objetos que tienen hijos. Almacena los hijos. El *CompositeAbstracto* define y proporciona las implementaciones por defecto para los métodos para manejar los componentes de los objetos composite. El método *add* añade un componente a un objeto composite. El método *remove* borra un componente del

objeto composite. El método *getChild* retorna una referencia a un objeto componente de un objeto composite.

CompositeConcreto1, CompositeConcreto2, ... Las instancias de estas clases son objetos composite que utilizan otras instancias del *ComponenteAbstracto*. Implementan las operaciones con los hijos en la interfaz.

Cliente Manipula los objetos en la composición a través de la interfaz de la clase *ComponenteAbstracto*.

La clase abstracta composite *CompositeAbstracto* no es necesaria cuando sólo hay una clase concreta composite *CompositeConcreto*.

2.6.4. Consecuencias

- Hay una jerarquía de objetos tan complejos como se requieran.
- El cliente es simple. Los objetos cliente de un *ComponenteAbstracto* pueden tratar simplemente con un *ComponenteAbstracto*, sin tener que conocer a ninguna de las subclases del *ComponenteAbstracto*.
- Si un cliente llama a un método de un *ComponenteAbstracto* que es supuestamente realizar una operación y el objeto *ComponenteAbstracto* es un objeto *CompositeAbstracto*, entonces delegará esta operación a los objetos *ComponenteAbstracto* que lo constituyen. Análogamente, si un objeto cliente llama a un método de un objeto *ComponenteAbstracto* que no es un *CompositeAbstracto* y el método requiere alguna información contextual, entonces el *ComponenteAbstracto* delegará la petición de información contextual a su padre.
- El principal beneficio del patrón Composite es que permite a los clientes de un objeto composite y a los objetos que lo constituyen desconocer la clase específica de los objetos con los que tratan.
- Se simplifica la adición de nuevos objetos.
- El diseño es más general.
- Se dificulta el control de los tipos de composiciones válidas. Deben hacerse chequeos en tiempo de ejecución.

2.6.5. Implementación

Si las clases que participan en el patrón Composite implementan alguna operación por delegación de estas a sus objetos padre, entonces el mejor modo para conservar velocidad y simplicidad es teniendo en cada instancia del *ComponenteAbstracto* una referencia a su padre. Es importante implementar el puntero al padre de una forma que se asegure la consistencia entre padre e hijo. Siempre debería ser que un *ComponenteAbstracto* identifique a un *CompositeAbstracto* como su padre, si y sólo si, el *CompositeAbstracto* lo

identifica como uno de sus hijos. La mejor forma para obligar esto es modificar las referencias al padre y al hijo solamente en la clase *CompositeAbstracto* añadiendo y quitando métodos.

Compartir componentes entre muchos padres utilizando el patrón Flyweight es una forma de ahorrar memoria. Sin embargo, es difícil compartir componentes correctamente manteniendo las referencias a los padres.

La clase *CompositeAbstracto* podría proporcionar una implementación por defecto para manejar los hijos de los objetos composite. Sin embargo, es muy común que los composite concretos sobrescriban esta implementación.

El patrón Composite es algunas veces implementado con clases que no comparten una superclase común, pero comparten un interface común.

Si una clase concreta composite delega una operación a los objetos que lo constituyen, entonces cachear el resultado de la operación podría mejorar el rendimiento. Si una clase concreta composite cachea el resultado de una operación, entonces es importante que los objetos que constituyen el composite notifiquen al objeto composite que pueden ser invalidados sus valores cacheados.

2.6.6. Usos en el API de Java

El paquete *java.awt.swing* contiene un buen ejemplo del patrón Composite. Su clase *Component* cubre el papel del *ComponenteAbstracto*. Su clase *Container* cubre el papel del *CompositeAbstracto*. Tiene un número de clases en el papel *ComponenteConcreto*, incluyendo *Label*, *TextField*, y *Button*. Entre las clases en el papel *CompositeConcreto* se incluyen *Panel*, *Frame*, y *Dialog*.

2.6.7. Patrones relacionados

Chain of Responsibility El patrón Chain of Responsibility puede ser combinado con el patrón Composite para añadir links del hijo al padre (para propagar responsabilidades hacia arriba), de tal forma que, los hijos puedan conseguir información sobre un progenitor sin tener que conocer que progenitor proporciona la información.

Decorator Generalmente se usan juntos.

Flyweight Para compartir componentes, cuando no haya referencias a los padres.

Iterator Para recorrer los hijos en un composite.

Vistor Se puede utilizar el patrón Visitor para encapsular operaciones en una clase simple, que de lo contrario podría propagarse a través de muchas clases y agregar comportamiento a las clases del patrón Composite..

2.7. DECORATOR [GoF95]

2.7.1. Objetivo

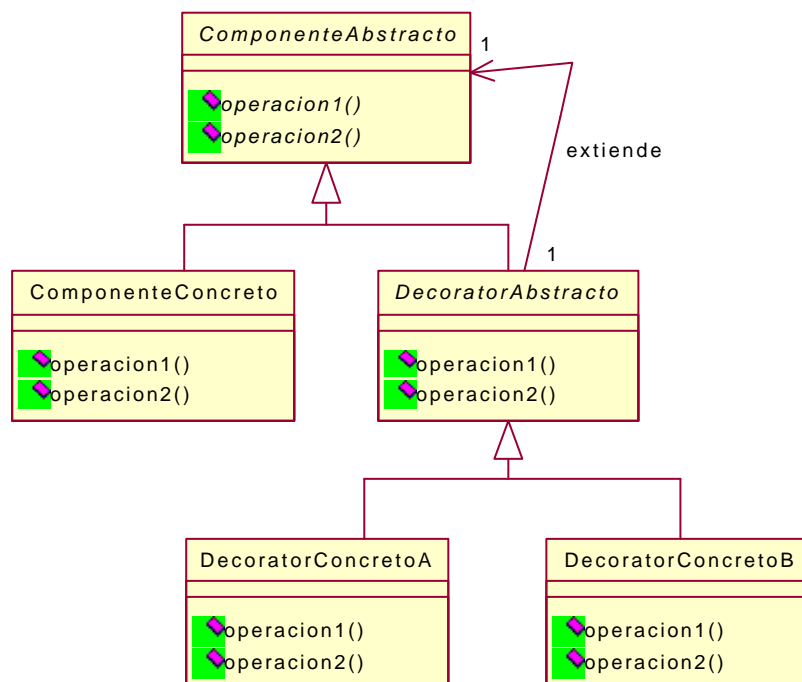
Extiende la funcionalidad de un objeto dinámicamente de tal modo que es transparente a sus clientes, utilizando una instancia de una subclase de la clase original que delega las operaciones al objeto original. Provee una alternativa muy flexible para agregar funcionalidad a una clase.

2.7.2. Aplicabilidad

- Hay una necesidad de extender la funcionalidad de una clase, pero no hay razones para extenderlo a través de la herencia.
- Hay la necesidad de extender dinámicamente la funcionalidad de un objeto y quizás quitar la funcionalidad extendida.

2.7.3. Solución

El diagrama de clases que muestra la estructura general del patrón Decorator es el siguiente:



Patrón Decorator.

A continuación están las descripciones de los papeles que juegan las clases de la figura anterior en el patrón Decorator:

ComponenteAbstracto Una clase abstracta en este papel es la superclase común de todos los objetos componente pueden potencialmente ser extendidos a través del patrón Decorator. En algunos casos los objetos componente que son extendidos no tienen una superclase común pero implementan un interface común. En este caso, el interface común ocupa el lugar de la clase abstracta.

ComponenteConcreto El patrón Decorator extiende las clases en este papel utilizando objetos que delegan a instancias de una clase *ComponenteConcreto*.

DecoratorAbstracto La clase abstracta en este papel es la superclase común para las clases decorator. Las instancias de esta clase tienen responsabilidad para mantener una referencia a los objetos componente que delegan en los objetos decorator. Esta clase también a menudo sobrescribe todos los métodos que hereda de la clase *ComponenteAbstracto*, así que, las instancias simplemente llaman al método del objeto componente que delega en el objeto decorator. Esta implementación por defecto proporciona exactamente el comportamiento necesario por los métodos cuyo comportamiento no está siendo extendido.

DecoratorConcretoA, DecoratorConcretoB, Estas clases concretas decorator extienden el comportamiento de los métodos que ellas heredan de la clase *DecoratorAbstracto* de cualquier forma necesaria.

2.7.4. Consecuencias

El patrón Decorator proporciona mayor flexibilidad que la herencia. Esto te permite alterar dinámicamente el comportamiento de los objetos individuales añadiendo y borrando decorators. La herencia, por otra parte, determina la naturaleza de todas las instancias de una clase estáticamente.

Utilizando diferentes combinaciones de unos pocos tipos distintos de objetos decorator, puedes crear muchas combinaciones distintas de comportamientos. Para crear esos diferentes tipos de comportamiento con la herencia se requiere que definas muchas clases distintas.

La flexibilidad de los objetos decorator los hace más propensos a errores que la herencia. Por ejemplo, es posible combinar objetos decorator de diferentes formas que no funcionen, o crear referencias circulares entre los objetos decorator.

Utilizando el patrón Decorator generalmente se tienen menos clases que utilizando la herencia. Teniendo menos clases se simplifica el diseño y la implementación de los programas. Por otro lado, utilizando el patrón Decorator normalmente se tienen más objetos. El gran número de objetos puede realizar el debugging (encontrar y reparar errores de un programa) más complicado.

Una última dificultad asociada al patrón Decorator es que un objeto decorator no es un objeto componente, por lo que se pierde la identidad del objeto. Los objetos componente se esconden detrás de los objetos decorator.

2.7.5. Implementación

La mayoría de las implementaciones del patrón Decorator son más sencillas que el caso general. Aquí están algunas de las simplificaciones más comunes:

- Si solamente hay una clase *ComponenteConcreto* y ninguna clase *ComponenteAbstracto*, entonces la clase *DecoratorAbstracto* es normalmente una subclase de la clase *ComponenteConcreto*.
- A menudo el patrón Decorator es utilizado para delegar a un único objeto. En este caso, no hay necesidad de tener la clase *DecoratorAbstracto* para mantener una colección de referencias. Sólo conservando una única referencia es suficiente.
- Si solamente hay una clase decorator, entonces no hay necesidad de tener una clase *DecoratorAbstracto* separada. Se puede fusionar las responsabilidades de la clase *DecoratorAbstracto* con otras de la clase concrete decorator. Podría también ser razonable prescindir de la clase *DecoratorAbstracto* si hay dos clases concretas decorator, pero no si hay más de dos.

2.7.6. Patrones relacionados

Delegation El patrón Decorator es una forma estructurada de aplicar el patrón Delegation.

Filter El patrón Filter es una versión especializada del patrón Decorator enfocada en la manipulación de stream de datos.

Strategy El patrón Decorator es útil para organizar las cosas que suceden antes o después de que se llaman a los métodos de otro objeto. Si quieres planificar diferentes cosas que ocurren en medio de las llamadas a un método considera utilizar el patrón Strategy.

Template Method El patrón Template Method es otra alternativa al patrón Decorator que permite variar el comportamiento en medio de una llamada a un método en lugar de antes o después.

Adapter Un decorador solo cambia las responsabilidades del objeto, no su interface. El patrón Adapter cambia el interface.

Composite Este patrón esta relacionado con el patrón Decorator, pero no es lo mismo. No está pensado para la agregación de objetos.

2.8. FACTORY METHOD [GoF95]

2.8.1. Objetivo

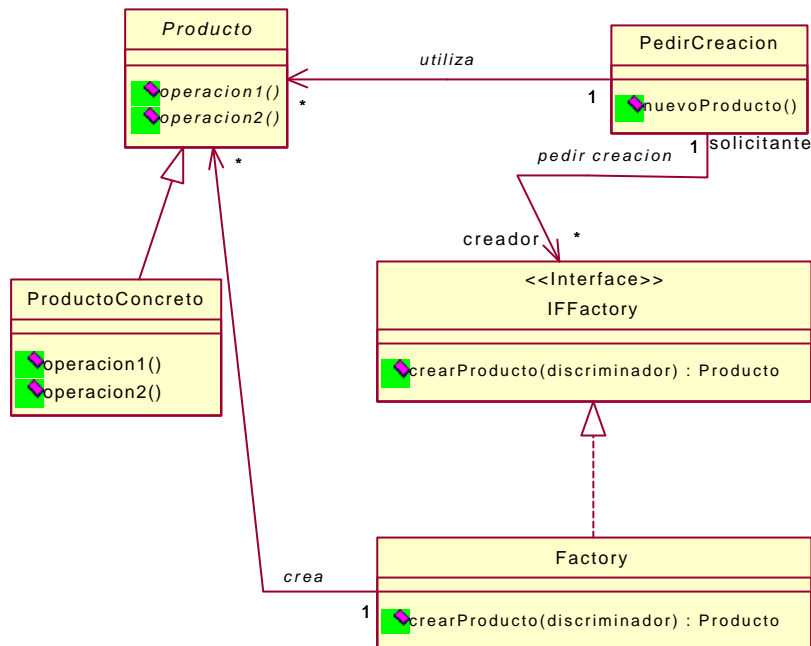
Tu escribes una clase para reutilizar con tipos arbitrarios de datos. Tu organizas esta clase de tal forma que se puedan instanciar otras clases sin depender de ninguna de las clases que puedan ser instanciadas. La clase reutilizable es capaz de permanecer independiente de las otras clases instanciadas delegando la elección de que clase instanciar a otros objetos y referirse a los objetos recién creados a través de un interface común. Define una interface para crear un objeto, dejando a las subclasses decidir el tipo específico. Permite delegar la responsabilidad de instanciación a las subclasses.

2.8.2. Aplicabilidad

- Debería ser posible organizar una clase de tal forma que se puedan crear objetos que hereden de una clase dada o que implementen un interface dado. Para ser reutilizable la clase debería crear objetos sin conocer que subclasses de una clase dada están disponibles o que clases que implementan un interface dado estan disponibles. Este conocimiento de uso específico debería de proceder de una clase de uno específico separada.
- Una clase no puede anticipar la clase de objetos que debe crear.
- Una clase quiere a sus clases derivadas para especificar los objetos a crear.
- Las clases delegan la responsabilidad a una de varias subclasses y se quiere localizar que subclase es la delegada.

2.8.3. Solución

El diagrama de clases que muestra la organización de las clases e interfaces que participan en el patrón Factory Method es el siguiente:



Patrón Factory Method.

A continuación están las descripciones de los papeles que juegan las clases e interfaces en la organización citada anteriormente:

- Producto** Una clase en este papel es una superclase abstracta de objetos producidos por el patrón Factory Method. Define la interfaz de los objetos que crea el patrón Factory Method.
- ProductoConcreto** Esta es alguna de las clases concretas que son instanciadas por los objetos que participan en el patrón Factory Method. Implementa la interfaz del producto. Si estas clases no comparten una lógica común, entonces el papel del producto puede ser realizado por un interface en lugar de una clase abstracta.
- PedirCreacion** Esta clase es una clase independiente que se necesita para crear clases específicas. Esto lo hace indirectamente a través de una instancia de una clase *Factory*.
- IFFactory** Es un interface. Los objetos que crean objetos producto por petición de objetos de la clase *PedirCreacion* deben implementar este interface. Los interfaces de este tipo declaran un método que es llamado por un objeto de la clase *PedirCreacion* para crear objetos de productos concretos. Este método tiene argumentos que son necesarios para deducir la clase a instanciar.
- Factory** Esta es la clase que implementa adecuadamente el interface *IFFactory* y tiene un método para crear objetos *ProductoConcreto*.

2.8.4. Consecuencias

Las primeras consecuencias de utilizar el patrón Factory Method son las siguientes:

- La clase *PedirCreacion* es independiente de la clase de los objetos de los productos concretos que son actualmente creados.
- El conjunto de clases Producto que pueden ser instanciadas podría cambiar dinámicamente.

2.8.5. Implementación

En situaciones donde todos los productos concretos son conocidos por anticipado, el interface del producto podría no ser necesario. En estas situaciones, el único beneficio de utilizar el patrón Factory Method es que la clase *PedirCreacion* se mantiene independiente de las clases de los productos concretos actuales instanciadas. Esta forma de trabajar es que la clase *PedirCreacion* se refiere directamente a un objeto factory. Este objeto factory tiene un método *crearProducto* implementado con la lógica necesaria para instanciar la clase correcta del producto concreto.

Si cada clase que implementa un interface producto crea solamente un tipo de producto concreto, entonces el método *crearProducto* definido por el interface podría no necesitar ningún parámetro. Sin embargo, si los objetos factory requieren crear muchos tipos de objetos producto entonces su método *crearProducto* necesita tener los parámetros necesarios para deducir que clase producto instanciar. Parametrizar los métodos *crearProducto* a menudo es algo así:

```
Image crearImagen (String ext)
{
    if (ext.equals("gif"))
        return new GIFImage();
    if (ext.equals("jpeg"))
        return new JPEGImage();
    ....
} //crearImagen(String)
```

Esta secuencia de código de palabras clave *if* funciona bien para métodos *crearProducto* que tiene un conjunto fijo de clases producto que instanciar. Para escribir un método *crearProducto* que maneje un variable o gran número de clase producto puedes utilizar varios objetos que indiquen que clase instanciar como claves en una tabla hash, con objetos *java.lang.reflect.Constructor* por valores. Utilizando esta técnica, buscas el valor de un argumento en la tabla hash y utilizas el objeto Constructor que esta como su valor en la tabla hash para instanciar el objeto deseado.

Otro punto que ilustra el segmento de código es que los métodos de creación son un lugar razonable para encontrar palabras clave *switch* o cadenas de palabras *if*. En muchas situaciones, la presencia de palabras *switch* o cadenas de palabras *if* en el código indican que un método debería ser implementado como un método polimórfico. Los métodos de creación no pueden ser implementados utilizando polimorfismo, porque el polimorfismo solamente funciona después de que un objeto ha sido creado.

Para muchas implementaciones del patrón Factory Method, los argumentos que los métodos *crearProducto* de los objetos factory tienen un conjunto de valores predeterminados. A menudo es conveniente para la clase *Factory* definir nombres simbólicos para cada uno de estos valores predeterminados. Las clases que piden a la clase *Factory* que cree objetos pueden utilizar estas constantes que definen los nombres simbólicos para especificar el tipo de objeto que debe ser creado.

2.8.6. Usos en el API de Java

El API de Java utiliza el patrón Factory Method en diferentes lugares para permitir la integración del entorno del applet con el programa que lo contiene. Por ejemplo, cada objeto *URL* tiene un objeto *URLConnection* asociado. Puedes utilizar los objetos *URLConnection* para leer los bytes de una URL. Los objetos *URLConnection* también tienen un método llamado *getContent* que retorna el contenido del paquete URL en un tipo apropiado de objeto. Por ejemplo, si la URL contiene un fichero *gif*, entonces el método *getContent* del objeto *URLConnection* retorna un objeto *Image*.

Los objetos *URLConnection* juegan el papel de pedir la creación del patrón Factory Method. Estos objetos delegan el trabajo del método *getContent* a un objeto *ContentHandler*. *ContentHandler* es una clase abstracta que sirve como una clase *Producto* que sabe como manejar un tipo específico de contenido. La forma en que un objeto *URLConnection* consigue un objeto *ContentHandler* es a través de un objeto *ContentHandlerFactory*. La clase *ContentHandlerFactory* es una clase abstracta que participa en el patrón Factory Method como un interface *IFFactory*. La clase *URLConnection* también tiene un método llamado *setContentHandlerFactory*. Los programas que contienen applets llaman a ese método para proporcionar un objeto factory utilizado por todos los objetos *URLConnection*.

2.8.7. Patrones relacionados

Abstract Factory El patrón Factory Method es útil para construir objetos individuales para un propósito específico sin que la petición de construcción conozca las clases específicas que serán instanciadas. Si necesitas crear un conjunto de algunos objetos relacionados o dependientes, entonces el patrón Abstract Factory es más apropiado para este uso.

Template Method El patrón Factory Method es a menudo utilizado con el patrón Template Method.

Prototype El patrón Prototype proporciona un modo alternativo para un objeto de trabajar con otros objetos sin conocer los detalles de su construcción. El patrón Prototype especifica la clase de objetos a crear usando una instancia prototípica, y creando los objetos mediante una copia de este prototipo. Este patrón no necesita derivar del interface *IFactory*. Sin embargo puede requerir una operación de inicialización en la clase *Producto*, esta operación no la necesita el patrón Factory Method.

2.9. STATE [GoF95]

2.9.1. Objetivo

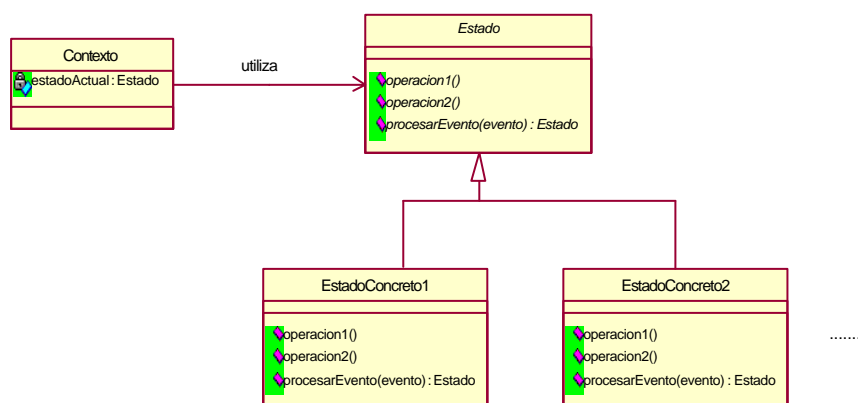
Encapsula los estados de un objeto como objetos separados, cada pertenencia en una subclase separada de una clase abstracta estado. Permite a un objeto cambiar su comportamiento cuando su estado interno cambia. El objeto parecerá haber cambiado de clase.

2.9.2. Aplicabilidad

- El comportamiento de un objeto es determinado por un estado interno que cambia en respuesta a los eventos. Debe cambiar su comportamiento en tiempo de ejecución cuando cambie su estado.
- La organización de la lógica que maneja un estado de un objeto debería ser capaz de cambiar a muchos estados sin llegar a ser una gran cantidad de código inmanejable.
- Las operaciones tienen sentencias condicionales largas que dependen del estado del objeto. Este estado es representado por una constante enumerada. A menudo, varias operaciones contendrán esta misma estructura condicional. Este patrón sitúa cada rama de la sentencia condicional en una clase separada. Esto permite tratar el estado de un objeto como un objeto que puede variar de forma independiente a otros objetos.

2.9.3. Solución

El diagrama de clases que muestra la organización de clases que participan en el patrón State es el siguiente:



Patrón State.

A continuación están las descripciones de los papeles que juegan las clases en la organización citada anteriormente:

- Contexto** *Contexto* es una clase cuyas instancias exponen el comportamiento de los estados. Define la interfaz para los clientes. Las instancias de la clase *Contexto* determinan su estado actual manteniendo una referencia a una instancia de una subclase concreta de la clase *Estado*. La subclase de la clase *Estado* determina el estado.
- Estado** La clase *Estado* es la superclase de todas las clases utilizadas para representar el estado de los objetos contexto. Define una interfaz para encapsular el comportamiento asociado con un estado particular del contexto.
- EstadoConcreto1, EstadoConcreto2,** Estas clases son subclases concretas de la clase *Estado*. Cada subclase implementa un comportamiento asociado con un estado del contexto.

La clase *Contexto* delega peticiones a estados específicos al objeto EstadoConcreto actual. Un contexto se puede pasar a sí mismo como argumento al objeto estado. Esto permite al objeto estado acceder al contexto si fuera necesario.

La clase *Contexto* es la interfaz primaria para los clientes que pueden configurar un contexto con objetos estado. Una vez que se configura el contexto los clientes no tienen que trabajar directamente con los objetos estado.

2.9.4. Consecuencias

El código para cada estado está en su propia clase. Esta organización hace más fácil añadir nuevos estados definiendo nuevas clases sin consecuencias no intencionadas. Por esta razón, el patrón State funciona muy bien para pocos y muchos estados.

Una alternativa al uso del patrón State es usar valores de datos para definir estados internos y tener operaciones en la clase *Contexto* que comprueben los datos explícitamente. Pero en este caso tendríamos instrucciones condicionales en la implementación del *Contexto*. Añadir un estado supondría cambiar varias operaciones que complican el mantenimiento.

El patrón State evita este problema pero introduce otro, ya que distribuye el comportamiento por diferentes estados. Esto aumenta el número de clases y es menos compacto que una clase simple. Sin embargo esta distribución es buena si hay muchos estados evitando sentencias condicionales largas. Estas sentencias, al igual que los métodos largos no son deseables. Son monolíticas y tienden a hacer el código poco explícito lo que lo convierte en difícil de entender y modificar. El patrón State ofrece una forma mejor de estructurar el código de estados específicos. Al encapsular cada transición de estados y acciones en una clase se eleva la idea de un estado ejecución para todos los estados de los objetos.

Hace las transiciones de estados más explícitas. Cuando un objeto define su estado actual sólo en términos de valores de datos internos las transacciones de estado no tienen representación explícita, sino que se muestran como asignaciones entre variables. Al introducir objetos separados para diferentes estados hace a las transacciones más explícitas. Además los objetos pueden proteger al *Contexto* de estados internos inconsistentes ya que las transiciones de estado parecen ser atómicas para los clientes de los objetos estado. Un cliente llama al método *procesarEvento* del estado actual y retorna el nuevo estado al cliente.

Los objetos que representan estados no paramétricos, no instancian variables (el estado se representa en su tipo), pueden ser compartidos como singletons si no hay necesidad de crear una instancia directa de la clase Estado.

2.9.5. Implementación

El patrón State no especifica quien define el criterio para las transiciones. Si el criterio es fijo puede ser implementado en el contexto. Sin embargo, es más flexible y apropiado permitir a las subclases de la clase *Estado* especificar su estado sucesor y cuando se hace la transición. Esto requiere añadir una interfaz a la clase *Contexto* para permitir a los objetos estado poner el estado actual al contexto.

Descentralizar la lógica de las transiciones hace más fácil modificar y extender dicha lógica definiendo nuevas subclases *Estado*. Una desventaja de esta descentralización es que una subclase *Estado* conocerá al menos una de las otras lo que introduce dependencia entre subclases.

Otra forma de imponer esta estructura es usar una tabla que mapee las transiciones. La ventaja principal de las tablas es su regularidad ya que se puede modificar el criterio de transición modificando datos en vez de código. Sin embargo tiene alguna desventaja:

Es menos eficiente que una llamada a una función, hace la transición menos explícita y más difícil de entender, es difícil añadir acciones para acompañar las transiciones de estados. El patrón State modela comportamientos específicos mientras que la tabla se centra en definir transiciones de estados.

2.9.6. Patrones relacionados

Flyweight Se puede utilizar el patrón Flyweight para compartir objetos estado para soportar un gran número de objetos de forma eficiente. Este patrón explica cuando y como se puede compartir un estado.

Mediator Puedes utilizar el patrón State con el patrón Mediator cuando implementas los interfaces de usuario.

Singleton Se pueden implementar estados utilizando el patrón Singleton.

2.10. TEMPLATE METHOD [GoF95]

2.10.1. Objetivo

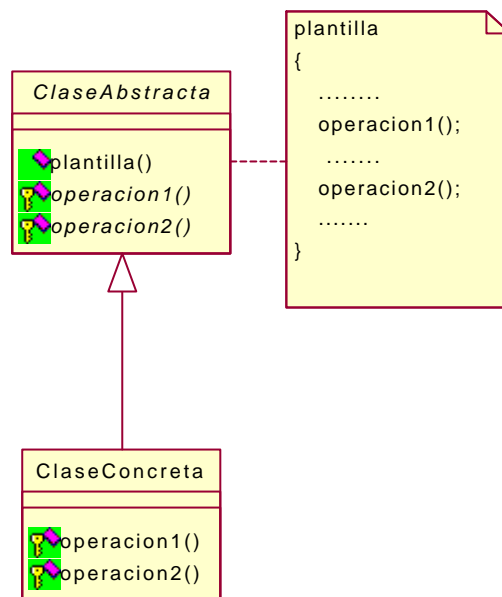
Escribe una clase abstracta que contiene parte de la lógica necesaria para realizar su finalidad. Organiza la clase de tal forma que sus métodos concretos llaman a un método abstracto donde la lógica buscada tendría que aparecer. Facilita la lógica buscada en métodos de subclases que sobrescriben a los métodos abstractos. Define un esqueleto de un algoritmo, delegando algunos pasos a las subclases. Permite redefinir ciertos pedazos del algoritmo sin cambiar su estructura.

2.10.2. Aplicabilidad

- Tienes que diseñar una clase que será utilizada en muchos programas. Pensando que todas las responsabilidades de la clase serán siempre las mismas, y que algunas porciones de su comportamiento serán diferentes en cada programa que la utiliza. Para asegurarse de que los programadores que utilizan la clase suministran la lógica a todas las responsabilidades específicas del programa, implementas la clase como una clase abstracta y las responsabilidades específicas del programa como métodos abstractos. Para utilizar la clase, un programador debe definir una subclase concreta. El compilador insistirá en que se sobrescriban todos los métodos abstractos de la superclase, por lo tanto forzará al programador a proporcionar el código para todas las responsabilidades específicas de la aplicación. Es decir este patrón se utiliza para implementar las partes invariantes de un algoritmo una vez y dejar a las subclases implementar el comportamiento que puede variar.
- Tienes un diseño existente que contiene un número de clases que hacen cosas similares. Para minimizar la duplicación de código, factorizas estas clases, identificando que tienen en común y que es diferente. Organizas las clases de tal forma que sus diferencias estén contenidas completamente en métodos discretos con nombres comunes. El resto del código de las clases debería ser un código común que puedes poner en una nueva superclase común. Defines los métodos que encapsulan la lógica especializada en la superclase como métodos abstractos. El compilador obliga a que las nuevas subclases de esta clase mantengan la misma organización que las otras subclases.

2.10.3. Solución

El diagrama de clases que muestra la organización de las clases que participan en el patrón Template Method es el siguiente:



Patrón Template Method.

A continuación están las descripciones de los papeles que juegan las clases e interfaces en la organización citada anteriormente.

- ClaseAbstracta** Una clase en este papel tienen un método concreto que contiene la lógica de alto nivel (el esqueleto del algoritmo) de la clase. Este es el método *plantilla* indicado en el diagrama. Este método llama a otros métodos, definidos en la clase *ClaseAbstracta* como métodos abstractos, que invocan a la lógica de bajo nivel que varía para cada subclase de la clase *ClaseAbstracta*.
- ClaseConcreta** Una clase en este papel es una subclase concreta de una clase *ClaseAbstracta*. La clase *ClaseConcreta* sobrescribe los métodos abstractos definidos en su superclase y proporciona la lógica necesaria (los pasos del algoritmo) para completar la lógica del método *plantilla*.

2.10.4. Consecuencias

Un programador que escribe una subclase de una clase *ClaseAbstracta* es obligado a sobrescribir estos métodos que deben ser sobrescritos para completar la lógica de la superclase. Una clase *ClaseAbstracta* bien diseñada tienen una estructura que proporciona un programador como guía para suministrar la estructura básica de sus subclases.

Esta técnica es fundamental para reutilizar código. Es especialmente importante en librerías de clases porque son los medios usados para obtener puntos comunes en estas librerías.

El patrón Template Method nos conduce a una estructura de control invertida que a veces se conoce con el nombre de “*el principio de Hollywood*” que es “*No nos llames, nosotros te llamaremos*”. Viene a indicar como una clase base llama a las operaciones de una subclase y no al revés.

2.10.5. Implementación

Una clase *ClaseAbstracta* proporciona la guía para forzar a los programadores a sobrescribir los métodos abstractos con la intención de proporcionar la lógica que rellena los huecos de la lógica de su método *plantilla*. Puedes proporcionar una estructura adicional para tener métodos adicionales en la clase *ClaseAbstracta* que las subclases puedan sobrescribir para proporcionar lógica suplementaria u opcional. Por ejemplo, consiedera una clase reutilizable.

Los métodos que pueden ser sobrescritos opcionalmente para proporcionar funcionalidad adicional o personalizada son llamados *métodos anzuelo*. Para realizarlo más fácil para un programador debe ser consciente de los métodos anzuelo que proporciona una clase, puedes aplicar una convención de nombres a los métodos anzuelo. Dos de las convenciones más comunes de nombres para métodos anzuelo son empezar los nombres de los métodos anzuelo con un prefijo *do-* o finalizar los estos métodos con el sufijo *-Hook*.

2.10.6. Patrones relacionados

Strategy El patrón Strategy modifica la lógica de los objetos individuales. El patrón Template Method modifica la lógica de una clase entera.

2.11. COMMAND [GoF95]

2.11.1. Objetivo

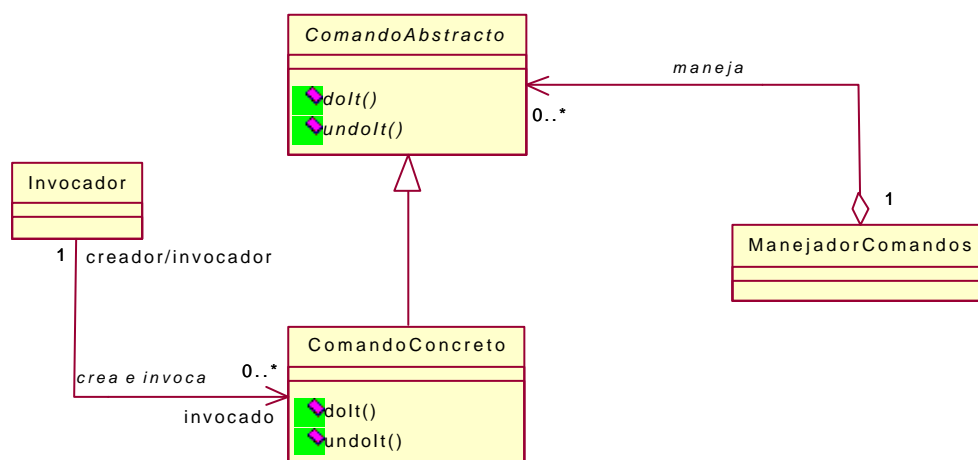
Encapsula una operación en un objeto, permitiendo parametrizar operaciones, de tal forma que se pueda controlar su selección y secuencia, ponerlas en la cola, deshacerlas y manipularlas.

2.11.2. Aplicabilidad

- Necesitas controlar la secuenciación, selección, o cronometraje de la ejecución de los comandos.
- Un tipo particular de manipulación de comandos que motiva el uso del patrón Command es el manejo de deshacer y rehacer.
- Soporte constante para mantener un log (anotación de las actividades que se producen en un ordenador) y la recuperación ante fallos. Desde que utilizas un log constante retrocederás a los efectos de los comandos previamente ejecutados. Un log constante puede ser incorporado en un mecanismo de manejo de transacciones que permite a los comandos deshacerse si una transacción es abortada.

2.11.3. Solución

El diagrama de clases que muestra la organización de las clases que participan en el patrón Command es el siguiente:



Patrón Command.

A continuación están las descripciones de los papeles que juegan las clases en la organización citada anteriormente:

- ComandoAbstracto** Una clase en este papel es la superclase de las clases que encapsulan los comandos. Sólo define un método abstracto *doIt* que otras clases llaman para ejecutar los comandos encapsulados por sus subclases. Si se requiere soporte deshacer, una clase *ComandoAbstracto* también define un método *undoIt* que deshace los efectos de la última llamada al método *doIt*.
- ComandoConcreto** Las clases en este papel son las clases concretas que encapsulan un comando específico. Otras clases invocan el comando a través de una llamada al método *doIt* de la clase. La lógica deshacer para el comando es invocada a través de una llamada al método *undoIt* de la clase. Los constructores de los objetos normalmente proporcionan algún parámetro que requieren los comandos. Muchos comandos necesitan al menos un parámetro, que es el objeto sobre el que actúa el comando. Por ejemplo, un comando salvar un objeto al disco requiere que el objeto que sea salvado sea pasado al constructor del objeto comando.
- Invocador** Una clase en este papel crea un objeto de comando concreto si este necesita llamar a un comando. Podría llamar a estos métodos *doIt* de los objetos o dejar que el objeto *manejadorComandos* lo haga.
- ManejadorComandos** Una clase *ManejadorComandos* es la responsable de manejar una colección de objetos comando creados por un objeto invocador. La responsabilidad específica de una clase *ManejadorComandos* puede incluir manejar los comandos de deshacer y rehacer, comandos secuenciales, y comandos de planificación. Las clases *ManejadorComandos* normalmente son independientes de las aplicaciones en las cuales ellas son utilizadas y pueden ser muy reutilizables.

2.11.4. Consecuencias

- El objeto que invoca a un comando no es el mismo objeto que ejecuta el comando. Esta separación proporciona flexibilidad en la planificación y secuenciación de comandos. Plasmar los comandos como objetos significa que ellos pueden ser agrupados, delegarse, y por supuesto manipularse como cualquier otro tipo de objeto. Se pueden componer comandos (Composite).
- La capacidad de agrupar y controlar la secuenciación de los comandos significa que puedes utilizar el patrón Command como la base de un mecanismo que soporta las ordenes del teclado. Este es un mecanismo que graba una secuencia de comandos y

les permite utilizarlos más tarde. El patrón Command también puede ser utilizado para crear otros tipos de patrones compuestos.

- Añadir nuevos comandos normalmente es más fácil porque no se rompe ninguna dependencia.

2.11.5. Implementación

Hay unas pocas cuestiones que considerar cuando implementas el patrón Command. La primera y posiblemente la más importante es decidir que comandos habrá. Si los comandos son emitidos por un interface de usuario que proporciona un nivel de los comandos del usuario, entonces un modo muy natural de identificar las clases de los comandos concretos es tener un clase comando concreto para cada nivel de los comandos de usuario. Si introduces esta estrategia y hay algún comando del usuario particularmente complejo, entonces habrá igualmente clases comando complejas. Para evitar poner demasiada complejidad en una clase, podrías querer implementar los comandos más complejos del usuario con varias clases command.

Si el número de comandos del usuario es muy grande, entonces podrías seguir la estrategia de implementarlas con combinaciones de objetos comandos. Esta estrategia podría permitirte implementar un gran número de comandos externos con un pequeño número de clases comando.

Otra cuestión de la implementación a considerar es la captura de la información de estado necesaria para los comandos deshacer. Para ser capaz de deshacer los efectos de un comando, es necesario guardar el estado de los objetos sobre el cual se opera para ser capaz de restaurar ese estado.

Podría haber comandos que no permitan deshacerse porque ellos necesitan guardar una gran cantidad de información del estado. Podría haber comandos que nunca pueden ser deshechos porque no es posible restaurar el estado de estos cambios. Los comandos que involucran el borrado de ficheros están dentro de esta categoría.

El objeto manejadorComandos debería ser consciente cuando es ejecutado un comando que no se puede deshacer. Hay un número de razones para esto:

- Suponiendo que un objeto manejadorComandos es responsable de inicializar la ejecución de comandos. Si esté es consciente de que el comando no se podrá deshacer antes de que esté es ejecutado, entonces se puede proporcionar un mecanismo común para avisar al usuario de que un comando no puede deshacerse va a ser ejecutado. Cuando se avisa al usuario, se puede ofrecer al usuario también la opción de no ejecutar el comando.
- Mantener una historia de un comando con el fin de deshacerlo consume memoria y algunas veces otros recursos. Después de ejecutar el comando que no puede deshacerse, la historia de los comandos que no es disponible puede ser borrada.

Mantener la historia de un comando después de ejecutar un comando que no se puede deshacer es un gasto de recursos.

- La mayoría de los interfaces de usuario de los programas que tienen un comando deshacer tienen un menú item que los usuarios pueden utilizar para permitir el comando deshacer. Los buenos interfaces de usuario evitan sorpresas a los usuarios. Responden a un comando deshacer, con una notificación de que el último comando no puede ser deshecho sorprendiendo al usuario que espera que un comando deshacer pueda ser llevado a cabo. Un modo de evitar esta sorpresa es que el objeto que maneja los comandos tenga habilitado o deshabilitado la opción del menú deshacer si el ultimo comando ejecutado puede deshacerse o no.

Puedes simplificar el patrón si no necesitas soportar la operación deshacer. Si la operación deshacer no es requerida, entonces la clase *ComandoAbstracto* no necesita definir el método *undoIt*.

Hay una extensión común del patrón Command utilizada cuando los comandos están públicos en un interface de usuario. El propósito de esta extensión es evitar juntar los componentes del interface de usuario a los objetos comando específicos o incluso requerir los componentes de los interfaces de usuario para conocer las clase concretas comando. La extensión consiste en incrustar el nombre de un comando en los componentes del interface de usuario y luego utilizar un objeto factory method para crear los objetos comando.

Invocar los comandos a través de un comando factory proporciona un estrato de in dirección que puede ser muy útil. La in dirección permite expedir varios objetos comando para compartir transparentemente el mismo objeto comando. Más importante aún, la in dirección hace más fácil tener menús y barras de herramientas al gusto del cliente.

2.11.6. Usos en el API de Java

El núcleo del API de Java no tiene ningún buen ejemplo del patrón Command. Sin embargo, contiene algunos soportes para la extensión de la GUI del patrón Command. Sus clases *Button* y *MenuItem* tienen métodos llamados *getActionCommand* y *setActionCommand* que puedes utilizar para conseguir y dar el nombre de un comando asociado con el botón o menú item.

2.11.7. Patrones relacionados

Factory Method El patrón Factory Method es utilizado algunas veces para proporcionar un estrato de in dirección entre el interface del usuario y las clases comandos.

Little Language Puedes utilizar el patrón Command para ayudarte a implementar un patrón Little Language.

Marker Interface Puedes utilizar el patrón Marker Interface con el patrón Command para implementar el proceso deshacer/rehacer.

Snapshot Si quieres proporcionar un granulado mecanismo deshacer que guarde el estado entero de un objeto mas que un calculo comando por comando de cómo reconstruir los estados previos, puedes utilizar el patrón Snapshot.

Template Method El patrón Template Method puede ser utilizado para implementar la lógica deshacer de alto nivel del patrón Command.

Composite Puede ser utilizado para macro-comandos.

Prototype Se usa cuando se copian comandos a una lista histórica.

2.12. ADAPTER [GoF95]

2.12.1. Objetivo

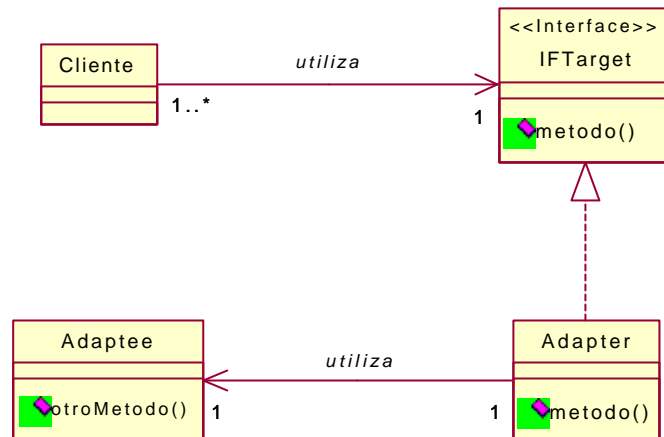
Una clase Adapter implementa un interfaz que conoce a sus clientes y proporciona acceso a una instancia de una clase que no conoce a sus clientes, es decir convierte la interfaz de una clase en una interfaz que el cliente espera. Un objeto Adapter proporciona la funcionalidad prometida por un interfaz sin tener que conocer que clase es utilizada para implementar ese interfaz. Permite trabajar juntas a dos clases con interfaces incompatibles.

2.12.2. Aplicabilidad

- Quieres utilizar una clase que llame a un método a través de una interface, pero quieres utilizarlo con una clase que no implementa ese interface. Modificar esa clase que implementa el interface no es una opción por un par de razones:
 1. No tienes el código fuente de la clase.
 2. La clase es una clase de propósito general, y es inapropiado para ella implementar un interface par un propósito específico.
- Quieres determinar dinámicamente que métodos de otros objetos llama un objeto. Quieres realizarlo sin que el objeto llamado tenga conocimientos de la otra clase de objetos.

2.12.3. Solución

Supón que tienes una clase que llama a un método a través de un interface. Quieres que una instancia de esa clase llame a un método de un objeto que no implementa el interface. Puedes planificar que la instancia realice la llamada a través de un objeto adapter que implementa el interface con un método que llama a un método del objeto que no implementa el interface. El diagrama de clases que demuestra esta organización es el siguiente:



Patrón Adapter.

A continuación están las descripciones de los papeles que juegan las clases e interfaces en la organización citada anteriormente:

Cliente	Esta es una clase que llama a un método de otra clase a través de un interface en regla que necesita no asumir que los objetos que lo llaman al método pertenecen a un clase específica.
IFTarget	Este interface declara el método que la clase <i>Cliente</i> llama.
Adapter	Esta clase implementa el interface <i>IFTarget</i> . Implementa el método que el cliente llama para llamar a un método de la clase <i>Adaptee</i> , la cual no implementa el interface <i>IFTarget</i> .
Adaptee	Esta clase no implementa el método del interface <i>IFTarget</i> pero tiene un método que quiere llamar la clase <i>Cliente</i> .

2.12.4. Consecuencias

- El cliente y las clases *Adaptee* permanecen independientes unas de las otras.
- El patrón Adapter introduce una in dirección adicional en un programa . Como cualquier otra in dirección, contribuye a la dificultad implicada en la compresión del programa.
- Puedes utilizar una clase *Adapter* para determinar cuales de los métodos de un objeto llama otro objeto.

2.12.5. Implementación

La implementación de una clase *Adapter* es más bien sencilla. Sin embargo, una cuestión que deberías considerar cuando implementes el patrón Adapter es como los objetos adapter conocen que instancia de la clase *Adaptee* llamar. Hay dos métodos:

1. Pasar una referencia a los objetos cliente como parámetro a los constructores de los objetos adapter o a uno de sus métodos. Esto permite al objeto adapter ser utilizado

con cualquier instancia o posiblemente muchas instancias de la clase *Adaptee*. Este método es ilustrado en el siguiente ejemplo:

```
class ClienteFacturaAdapter implements IFDireccion
{
    private Cliente miCliente;

    /**
     *
     */

    public ClienteFacturaAdapter (Cliente cliente)
    {
        miCliente=cliente;
    } //constructor

    /**
     *
     */

    public String getDireccion1()
    {
        return miCliente.getFacturaDireccion1();
    } //getDireccion1()

    /**
     *
     */

    public void setDireccion1(String direccion1)
    {
        miCliente.setFacturaDireccion1(direccion1);
    } //setDireccion1()

    /**
     *
     */

} //class ClienteFacturaAdapter
```

2. Hacer la clase *Adapter* una clase interna de la clase *Adaptee*. Esto simplifica la asociación entre el objeto adapter y el objeto adaptee haciendolo automático. También hace la asociación inflexible. Este método es ilustrado en el siguiente ejemplo:

```
MenuItem salir=new MenuItem ("Salir");
salir.addActionListener(new ActionListener()
{
    public void actionPerformed
        (ActionEvent
evt)
    {
        close();
    } //actionPerformed(ActionEvent)
});
```

2.12.6. Usos en el API de Java

Una forma muy común de utilizar para utilizar las clases *Adapter* con el API de Java es para manejar eventos, como esto:

```

Button ok=new Button("OK");
ok.addActionListener(new ActionListener()
{
    public void actionPerformed
        (ActionEvent
evt)
    {
        doIt();
    } //actionPerformed(ActionEvent)
});
add(ok);
    
```

El ejemplo del código anterior crea una instancia de una clase anónima que implementa el interface *ActionListener*. El objeto *Button* llama al método *actionPerformed* de la clase cuando el botón es presionado. Este patrón codificado es muy común para el código que maneja eventos.

El API de Java no incluye ningún objeto adapter público que este listo para utilizar. Incluye clases tales como *java.awt.event.WindowAdapter* que pretenden ser heredadas más que utilizarse directamente. La idea es que hay algunos interfaces escuchadores de eventos, tales como *WindowListener*, que declaran varios métodos. En muchos casos no todos los métodos necesitan ser implementados. El interface *WindowListener* declara ocho métodos que son llamados para proporcionar notificación sobre los ocho tipos diferentes de eventos de la ventana. A menudo uno o dos de estos tipos de eventos son de interés. Los métodos que corresponden a los eventos que no son de interés normalmente están dando implementaciones de no hacer nada. La clase *WindowAdapter* implementa el interface *WindowListener* e implementa los ocho métodos con implementaciones de no hacer nada. Una clase *Adapter* que hereda la clase *WindowAdapter* necesita implementar solamente los métodos que corresponden a los eventos que son de interés. Hereda las implementaciones de no hacer nada para el resto. Por ejemplo:

```

addWindowListener(new WindowAdapter()
{
    public void windowClosing
        (WindowEvent
evt)
    {
        exit();
    } //windowClosing(WindowEvent)
});
    
```

En este código de ejemplo, la clase *Adapter* anónima es una subclase de la clase *WindowAdapter*. Implementa solamente el método *windowClosing*. Hereda las implementaciones de no hacer nada para los otros siete métodos de la clase *WindowAdapter*.

2.12.7. Patrones relacionados

Facade La clase Adapter proporciona un objeto que actúa como un intermediario para llamar a los métodos entre los objetos cliente y *uno de los otros* objetos que no conocen a los objetos cliente. El patrón Facade proporciona un objeto que actúa como un intermediario para llamar a los métodos entre los objetos cliente y *múltiples* objetos que no conocen a los objetos cliente.

Iterator El patrón Iterator es una versión especializada del patrón Adapter para acceder secuencialmente al contenido de una colección de objetos.

Proxy El patrón Proxy, como el patrón Adapter, utiliza un objeto que es un sustituto por otro objeto. Sin embargo, un objeto proxy tiene el mismo interface que el objeto por el que es sustituido.

Strategy El patrón Strategy es estructuralmente similar al patrón Adapter. La diferencia está en el objetivo. El patrón Adapter permite a un objeto cliente exportar su función deseada originalmente para llamar al método de los objetos que implementan un interface particular. El patrón Strategy proporciona objetos que implementan un interface particular con el propósito de alterar o determinar el comportamiento de un objeto cliente.

2.13. MEDIATOR [GoF95]

2.13.1. Objetivo

Define un objeto que encapsula la forma en la cual interactúan otros. Utiliza un objeto para coordinar cambios de estado entre otros objetos. Colocando la lógica en un objeto para manejar cambios de estado de otros objetos, en lugar de distribuir la lógica sobre los otros objetos, dando como resultado una implementación mas cohesiva de la lógica y decrementando las relaciones entre otros objetos. Promueve el acoplamiento débil.

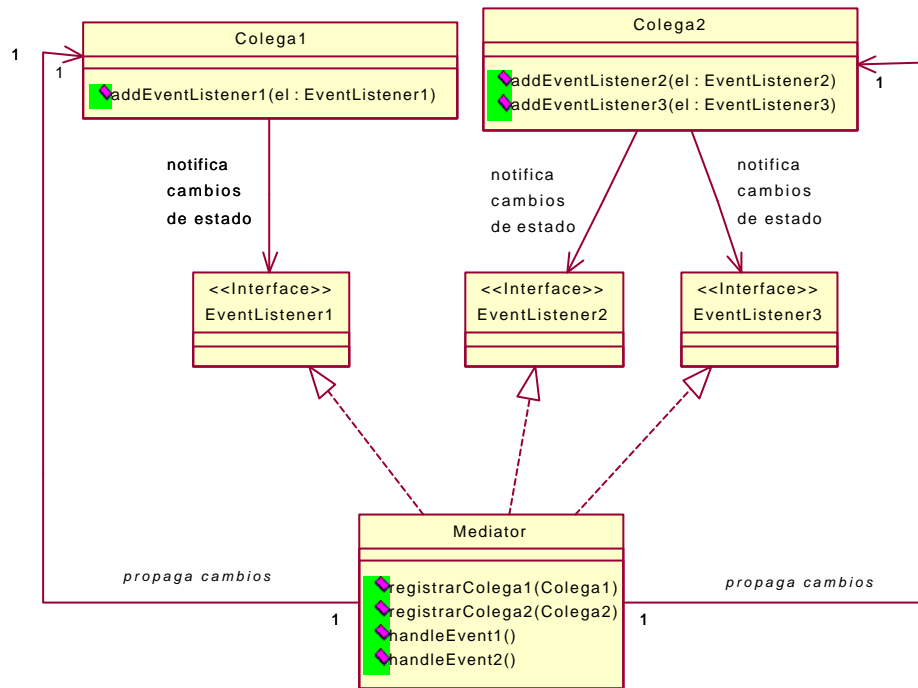
2.13.2. Aplicabilidad

Utiliza el patrón Mediator cuando:

- Tienes un conjunto de objetos relacionados y la mayoría de los objetos están involucrados en muchas relaciones de dependencia.
- Te das cuenta al definir las subclases que los objetos individuales pueden participar en relaciones de dependencia.
- Las clases son difíciles de reutilizar porque su función básica esta entrelazada con relaciones de dependencia.

2.13.3. Solución

El diagrama de clases que muestra la organización de las clases e interfaces que participan en el patrón Mediator, en el caso general, es el siguiente:



Patrón Mediator.

A continuación están las descripciones de los papeles que juegan las clases e interfaces en la organización citada anteriormente:

Colega1, Colega2, .. Las instancias de las clases en estos papeles tienen el estado relacionado a las dependencias. Hay dos tipos de dependencias:

- Un tipo de dependencia requiere un objeto para conseguir la aprobación de otros objetos antes de hacer tipos específicos de cambios de estado.
- El otro tipo de dependencia requiere un objeto para notificar a otros objetos después de que este ha hecho un tipo específico de cambios de estado.

Ambos tipos de dependencias son manejadas de un modo similar. Las instancias de *Colega1*, *Colega2*, están asociadas con un objeto mediator. Cuando ellos quieren conseguir la aprobación anterior para un cambio de estado, llaman a un método del objeto Mediator. El método del objeto Mediator realiza cuidadoso el resto.

EventListener1, EventListener2, .. Los interfaces en este papel permiten a las clases *Colega1*, *Colega2*, ... conseguir un nivel alto de reusabilidad. Ellos hacen esto para permitir que estas clases estén despreocupadas de que ellos están trabajando con un objeto mediator. Cada uno de estos interfaces define métodos

Mediator

relacionados con un tipo particular de evento. Para proporcionar notificaciones de estado los objetos colegas llaman al método apropiado del interface apropiado sin conocer la clase de los objetos mediator que implementa el método.

Las instancias de las clases en el papel *Mediator* tienen la lógica para procesar las notificaciones de estado de los objetos colega1, colega2, Las clases *Mediator* implementan uno o más de un interface *EventListener*. Los objetos colega1, colega2, ... llaman a métodos declarados por los interfaces *EventListener* para informar al objeto mediator de cambios de estado. El objeto mediator entonces realiza la lógica que es apropiada. Para notificar de cambios de estado pensados, normalmente incluirá la aprobación o desaprobación del cambio. Para notificar los cambios de estado completados, normalmente incluirá la propagación de las notificaciones a otros objetos.

Las clases Mediator tienen métodos que pueden ser llamados para asociarlos con objetos colega1, colega2, ... Estos métodos son indicados en el diagrama como *registrarColega1*, *registrarColega2*,A ellos se les pasa un objeto apropiado colega y generalmente llaman a uno o a más de uno de sus metodos *add....Listener* para informar al objeto colega que debería informar al objeto mediator de los cambios de estado.

Los medios que los objetos colega1, colega2,.... proporcionan permiten a otros objetos expresar su interés en los cambios de estado y el mecanismo para proporcionar notificaciones de estos cambios de estado normalmente se asemeja al modelo de delegación de eventos de Java.

2.13.4. Consecuencias

- La mayoría de la complejidad involucrada en el manejo de dependencias se cambia de otros objetos al objeto mediator. Esto hace que los otros objetos sean más fáciles de implementar y mantener.
- Poner toda la dependencia de la lógica para un conjunto de objetos relacionados en un lugar puede hacer incomprensible la dependencia lógica fácilmente. Si la clase *Mediator* llega a ser demasiado grande, entonces dividirlo en piezas más pequeñas puede hacerlo más comprensible.
- Utilizar un objeto mediator normalmente significa que no hay necesidad de heredar de las clases *Colega* solo para implementar su manejo de dependencias.
- Las clases *Colega* son más reutilizables porque su núcleo de funcionalidades no esta entrelazado con el código de manejo de dependencias. El código de manejo de dependencias tiende a ser específico de una aplicación.

- Como el código de manejo de dependencias normalmente es específico de la aplicación, la clase *Mediator* no es normalmente reutilizable.

2.13.5. Implementación

En muchos casos, un objeto es responsable de crear todos los objetos colega y su objeto mediator. Este objeto típicamente representa un frame o un diálogo y es un contenedor para los objetos que crea. Cuando hay un único objeto responsable de crear todos los objetos colega y su objeto mediator, la clase *Mediator* es normalmente declarada como un miembro privado de esa clase. Limitando la visibilidad de una clase *Mediator* incrementas la robustez del programa.

Hay algunas decisiones que tendrías que realizar cuando implementes el patrón Mediator. Una de estas decisiones es si un objeto mediator mantendrá su propio modelo interno de los estados de los objetos colega o buscará el estado de cada objeto cuando necesite conocer el estado del objeto.

Si realizas el primer método, entonces el objeto mediator empieza asumiendo el estado inicial de todos los objetos colega de los que es responsable. Tendrá una variable instancia para cada objeto colega. Un objeto mediator impone el valor inicial para cada una de estas variables instancia que será lo que supone el estado inicial de los correspondientes objetos colega. Cuando un objeto colega notifica al objeto mediator que su estado ha cambiado, el objeto mediator cambia los valores de sus variables instancia para colocar el nuevo estado. Después de que el objeto mediator actualiza su variables instancia, utiliza los valores de estas variables instancia para tomar decisiones si es que estas necesitan ser tomadas.

Si realizas el segundo método, el objeto mediator no somete a juicio el modelo de estados de los objetos colega con sus variables instancia. En lugar de esto, cuando un objeto colega le notifique de un cambio de estado, el objeto mediator tomará las decisiones necesarias a realizar para buscar el estado de cada objeto colega sobre el cual debe basar sus decisiones.

La mayoría de la gente que implementa por primera vez el patrón Mediator piensa en el primer método. Sin embargo, en la mayoría de los casos, el segundo método es la mejor opción. La desventaja del primer método es que es posible que el objeto mediator este equivocado sobre el estado de un objeto colega. Para que un objeto mediator consiga correctamente el modelo de estados de los objetos colega podría requerir tener código adicional para imitar la lógica de los objetos colega. Si un programador de mantenimiento modifica después una de las clase *Colega*, la modificación podría misteriosamente romper la clase *Mediator*.

La ventaja del segundo método es su simplicidad. El objeto mediator nunca puede estar equivocado sobre el estado de un objeto colega. Esto hace más fácil la implementación y el mantenimiento. Sin embargo, si buscas los estados completos de los objetos colega se consume demasiado tiempo, entonces el método de tener el objeto mediator el modelo de estados de los objetos colega podría ser más práctico.

2.13.6. Usos en el API de Java

Los ejemplos del patrón Mediator que ocurren en el API de Java tienen una pequeña diferencia de las clases *Mediator* que tienes probablemente en código. Esto es porque las clases *Mediator* normalmente contienen código de aplicaciones específicas, y las clases del API de Java son aplicaciones independientes.

Los interfaces de usuario gráficos basados en Java pueden ser contruidos desde objetos que son instancias de subclases de *java.awt.swing.Jcomponent*. Los objetos *Jcomponent* utilizan una instancia de una subclase de *java.awt.swing.FocusManager* como un mediator. Si los objetos *Jcomponent* están asociados con un objeto *FocusManager*, entonces ellos llaman a su método *processKeyEvent* cuando ellos reciben un *KeyEvent*. El propósito de un objeto *FocusManager* es reconocer los golpes de teclado que podría causar un objeto diferente *Jcomponent* para recibir el área que recibe de entrada y hacerlo también.

La forma en que los objetos *Jcomponent* utilizan un objeto *FocusManager* difiere del patrón Mediator descrito aquí en dos cosas:

1. Los objetos *Jcomponent* solamente pasan los *KeyEvents* a los objetos *FocusManager*. La mayoría de las clases *Mediator* que escribes tendrán que manejar más de un tipo de evento.
2. Los objetos *Jcomponent* no acceden a los objetos *FocusManager* a través de un interface. Ellos se refieren directamente a la clase *FocusManager*. Teniendo los objetos *Jcomponent* referidos a un objeto *FocusManager* a través de un interface proporcionaría una organización más flexible. Aparentemente, los diseñadores del API de Java creyeron esto, porque la interacción entre los objetos *Jcomponent* y los objetos *FocusManager* es de bajo nivel, no hay necesidad de esa flexibilidad.

2.13.7. Patrones relacionados

Adapter Las clases *Mediator* a menudo utilizan objetos adapter para recibir notificaciones de cambios de estado.

Interface El patrón Mediator utiliza el patrón Interface para mantener las clases *Colega* independientes de la clase *Mediator*.

Observer El patrón Observer tiene una gran parte del modelo de delegación de eventos de Java. Si quieres utilizar el patrón Mediator en un contexto para el cual crees que el modelo de eventos de Java es inapropiado, puedes sustituir el patrón Observer.

2.14. NULL OBJECT [Woolf97]

2.14.1. Objetivo

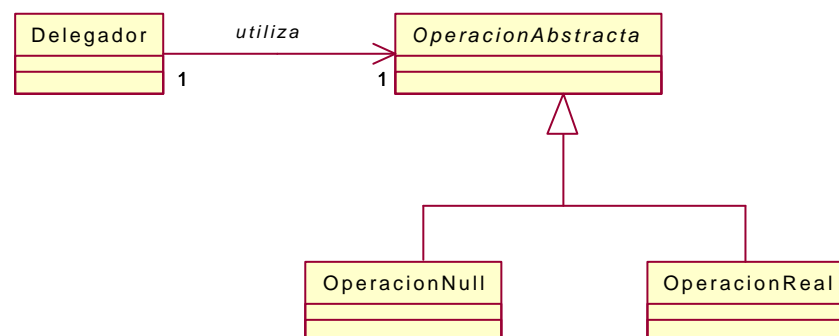
Facilita una alternativa para utilizar null para indicar la ausencia de un objeto al que delegue una operación. Utilizando null para indicar la ausencia de cada uno de los objetos requeridos se realiza una pregunta para ver si es null antes de cada llamada a los métodos de otros objetos. En lugar de utilizar null, el patrón Null Object utiliza una referencia a un objeto que no hace nada.

2.14.2. Aplicabilidad

- Una clase delega una operación a otra clase. La clase que delega normalmente no presta atención sobre como la otra clase implementa la operación. Sin embargo, algunas veces se requiere que la operación sea implementada sin hacer nada.
- Si quieres que la clase que delega la operación la delegue en todos los casos, incluyendo el caso de no hacer nada. No quieres que el caso de no hacer nada requiera ningún código especial en la clase que delega.

2.14.3. Solución

El diagrama de clases que muestra la organización de las clases que participan en el patrón Null Object es el siguiente:



Patrón Null Object.

A continuación están las descripciones de los papeles que juegan las clase en la organización citada anteriormente:

- Delegador** Una clase en este papel participa en el patrón Null Object para delegar una operación a una clase abstracta o a un interface. Se realiza la delegación sin tener responsabilidad sobre el caso de no hacer nada de una operación. Esta simplicidad da por hecho que el objeto en el que delega encapsulará el comportamiento correcto, incluso si este es no hacer nada.
- OperacionAbstracta** Una clase en el papel *Delegador* delega una operación a una clase en el papel *OperacionAbstracta*. Las clases en este papel no tienen porque ser necesariamente abstractas. Un interface puede también cumplir este papel.
- OperacionReal** Las clases en este papel implementan la operación que la clase *Delegador* delega a la clase *OperacionAbstracta*.
- OperacionNull** Las clases en este papel proporcionan una implementación para no hacer nada de la operación que la clase *Delegador* delega a la clase *OperacionAbstracta*.

2.14.4. Consecuencias

- El patrón Null Object desahoga una clase que delega una operación a otra clase con la responsabilidad de implementar la versión de no hacer nada de esa operación. Esto tiene como consecuencia un código más simple que no tiene que tener un test por *null* antes de llamar al método que implementa la operación delegada. Resulta un código más fiable porque el patrón Null Object elimina algunos casos de crear bugs (errores en los programas) por omitir el test por *null* en el código.
- El comportamiento de no hacer nada encapsulado por una clase en el papel *OperacionNull* es reutilizable, si hay un comportamiento consistente de no hacer nada que funciona para todas las clases *Delegador*.
- El patrón Null Object incrementa el número de clases en el programa. Si no hay todavía una clase o interface en el papel de *OperacionAbstracta*, entonces el patrón Null Object podría introducir más complejidad al introducir clases adicionales que lo alejan de la simplificación del código.

2.14.5. Implementación

Es frecuente el caso de que instancias de las clases *OperacionNull* no contengan información de instancias específicas. Cuando se esta en este caso, puedes ahorrar tiempo y memoria implementando la clase *OperacionNull* como una clase singleton.

2.14.6. Patrones relacionados

Singleton Si las instancias de una clase *OperacionNull* no contienen información de instancias específicas, entonces puedes ahorrar tiempo y memoria implementando la clase *OperacionNull* como una clase singleton.

Strategy El patrón Null Object se utiliza frecuentemente con el patrón Strategy.

3. EJEMPLOS DE APLICACIONES UTILIZANDO PATRONES DE DISEÑO

Una vez realizada la introducción a los patrones y más concretamente a los patrones de diseño vamos a ver su gran aplicación en ejemplos concretos.

Los patrones de diseño son de una ayuda invaluable para el desarrollo orientado a objetos de software de calidad.

El conjunto de patrones permiten crear software reusable. Los patrones son más que una solución, son un núcleo de soluciones, que se ofrecen como sugerencias más que como normas.

Los **patrones de diseño** se deben de aplicar en la práctica para comprenderlos y conocerlos, por ello, a partir de ahora, el desarrollo del proyecto consistirá en la utilización de patrones de diseño en la realización de programas, con **un fin didáctico**, explicando más detalladamente los patrones utilizados, cómo y porqué, así como la implementación de dichos ejemplos en uno de los últimos lenguajes de programación que han aparecido y que mayor futuro tienen, debido a sus numerosas ventajas, el **lenguaje Java**.

Para la realización de los ejemplos se ha utilizado el Kit de Desarrollo de Java, el **JDK 1.2.2**. Como entorno de desarrollo se ha utilizado el **FreeJava 3.0**.

El código de cada uno de los ejemplos se mostrará de forma completa en diferentes anexos.

3.1. APPLET PARA ESCUCHAR AUDIO

3.1.1. ¿En qué consiste la aplicación?

Se trata de un applet que nos permita garantizar que no se escucha más de un audio clip al mismo tiempo. Si un applet contiene dos partes de código que independientemente ponen audio clips, entonces es posible que ambos se escuchen al mismo tiempo. Cuando dos audio clips se escuchan al mismo tiempo, los resultados son de confusión al mezclarse los sonidos.

Para evitar la indeseable situación de escuchar dos audio clips a la vez, la clase que maneja los audio clips debe parar un audio clip antes empezar a poner otro. El modo de diseñar una clase que implemente esta política es una clase que garantice que sólo hay una instancia de esa clase compartida por todos los objetos que usan la clase. Si todas las peticiones de escuchar audio clips se hacen a través del mismo objeto, entonces es sencillo para ese objeto parar un audio clip antes de empezar otro.



Forma del applet al ejecutarlo.

3.1.2. Patrones de diseño utilizados

Esta aplicación es un ejemplo del patrón Singleton. La clase *ManejadorAudio* es la clase Singleton.

El constructor de la clase *ManejadorAudio* es privado. Esto previene que otras clases creen directamente una instancia de la clase *ManejadorAudio*. En cambio para conseguir una instancia de la clase *ManejadorAudio*, otras clases deben de llamar a su método *getInstancia*. El método *getInstancia* es un método static que siempre retorna la misma instancia de la clase *ManejadorAudio*. La instancia que retorna es la instancia referida a su variable privada static *instancia*.

El resto de los métodos de la clase *ManejadorAudio* son responsables del control de los audio clips. La clase *ManejadorAudio* tiene una instancia privada de una variable llamada *audioAnterior*, la cual es inicializada a null y luego referida al último audio clip escuchado. Antes de escuchar un nuevo audio clip, la instancia de la clase *ManejadorAudio* para el audio clip referido por la variable *audioAnterior*. Esto garantiza que el audio clip anterior finalice antes de que el siguiente audio clip empiece.

Si todos los audio clips se ponen a través del único objeto de la clase *ManejadorAudio*, nunca habrá más de un audio clip sonando al mismo tiempo.

3.1.3. Diagrama de clases

Esta aplicación consta de dos clases: *ManejadorAudio* y *AudioPlayer*. Su diagrama de clases es el siguiente:

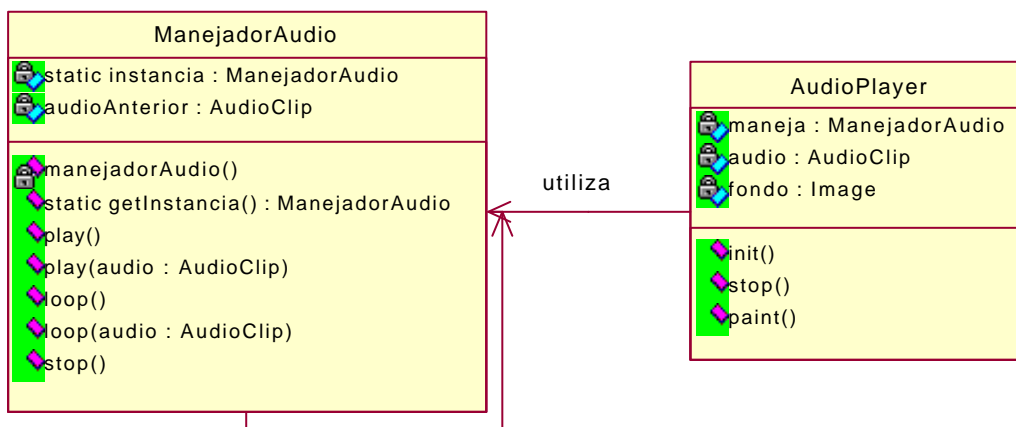


Diagrama de clases del applet para escuchar audio.

3.1.4. Código en Java

La clase que implementa el patrón Singleton es la clase *ManejadorAudio*, que implementa el interface *AudioClip* del API de Java para poder escuchar los audios. Para ello esta clase crea el único objeto de la clase *ManejadorAudio* que va a existir. De esta forma evitamos que se puedan escuchar dos audio clips a la vez, siempre y cuando se use para reproducirlos el único objeto de la clase *ManejadorAudio*, llamado *instancia*, el cual es privado y static.

```
import java.applet.AudioClip;

/**
 * Esta clase es utilizada para evitar que dos audio clips
 * suenen a la vez. La clase tiene una sola instancia a la
 * que se puede acceder a través de su metodo getInstancia.
 * Cuando pones audio clips a traves de ese objeto, este detiene
 * el ultimo audio clip que se esta escuchando antes de empezar
 * el nuevo audio demandado. Si todos los audio clips se ponen
 * a través del objeto instancia entonces nunca habrá mas
 * de un audio clip sonando al mismo tiempo.
 */
public class ManejadorAudio implements AudioClip
{

    private static ManejadorAudio instancia=new ManejadorAudio();
    private AudioClip audioAnterior; //la petición anterior de audio

    .....
}
```

Hay que declarar un constructor privado para que otras clases no puedan crear más objetos.

```
/**
 * Definimos un constructor privado, por lo tanto no se generará
un
 * constructor publico por defecto.
 */
private ManejadorAudio()
{
} //ManejadorAudio()

.....
}
```

Para acceder al único objeto instancia que existe de la clase *ManejadorAudio* se hace a través del método *getInstancia*.

```
/**
 * Retorna una referencia a la unica instancia de la clase
 * ManejadorAudio
 * @return ManejadorAudio la unica instancia de la clase
 * ManejadorAudio
 */
```

```
public static ManejadorAudio getInstancia()
{
    return instancia;
} //getInstancia()

.....
```

Cuando trabajas a través de este objeto, se para el audio clip que esta sonando antes de que empiece a escucharse otro.

```
/**
 * Para la ejecución del audio que se esta escuchando y empieza el
 * audio pasado como parametro.
 * @param audio el nuevo audio a escuchar.
 */
public void play(AudioClip audio)
{
    if (audioAnterior!=null)
        audioAnterior.stop();
    if (audio!=null)
    {
        audioAnterior=audio;
        audio.play();
    }
} //play(AudioClip)

.....
```

La clase *AudioPlayer* es la que utiliza la única instancia de la clase, que obtiene a través del método *getInstancia* de la clase *ManejadorAudio*, para escuchar los diferentes audio clips.

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

/**
 * Esta clase es la que implementa el applet. Desde aqui se
 * controlan los audios a traves de la clase ManejadorAudio.
 */
public class AudioPlayer extends Applet
{
    ManejadorAudio maneja=ManejadorAudio.getInstancia();
    /*maneja es la referencia al unico objeto que existe de la
    clase ManejadorAudio que lo obtiene a través del metodo
    getInstancia de la clase ManejadorAudio*/

    .....
```

```
.....

/**
 * Esta clase controla los eventos sobre los botones.
 */
class HandlerBotones implements ActionListener
```

```

{
/*****/

/**
 * Este metodo dicta lo que se ha de hacer cuando se pulsa
 * alguno de los tres botones: play, stop y loop.
 */
public void actionPerformed(ActionEvent e)
{
    String s = e.getActionCommand();
    if("Play".equals(s))
        maneja.play(audio);
    else if("Stop".equals(s))
        maneja.stop();
    else if("Loop".equals(s))
        maneja.loop(audio);
} //actionPerformed

/*****/

} //class HndlerBotones

/*****/
/*****/
/**
 * Esta clase controla los eventos sobre los radio botones.
 */
class HandlerCheckbox implements ItemListener
{
/*****/

/**
 * Este método dicta lo que se hace cuando se selecciona alguno
 * de los dos radio botones: audio1 y audio2.
 */
public void itemStateChanged (ItemEvent e)
{
    String s;
    Graphics g;
    Checkbox checkbox=(Checkbox) e.getItemSelectable();
    if(checkbox.getState())
    {
        s=checkbox.getLabel();
        if ("Audio1".equals(s))
        {
            audio=getAudioClip(getCodeBase(),"audio/guitarra.au");
            maneja.play(audio);
            fondo= getImage(getCodeBase(),"paris.jpg");
            repaint();
        }
        else
        {
            audio=getAudioClip(getCodeBase(), "audio/spacemusic.au");
            maneja.play(audio);
            fondo=getImage(getCodeBase(),"playa.jpg");
            repaint();
        }
    }
}
}

```

```

        } //itemStateChanged

/*****

        } //classHandlerCheckbox
        .....
    
```

El código completo se muestra en el Anexo A.

3.1.5. Funcionamiento de la aplicación

La aplicación se puede ejecutar al abrir en un visor de páginas web, que ejecute java, el fichero *AudioPlayer.html* o también ejecutando el fichero *run.bat* que esta en el directorio *Applet Audio* bajo MS-DOS.

Cuando se ha ejecutado el applet se verá en pantalla una ventana similar a la figura que aparece en el apartado 3.1.1. Esta ventana tiene tres botones: *Play*, *Stop* y *Loop* y dos radio botones *Audio1* y *Audio2*. Para poder escuchar algo hay que seleccionar en primer lugar uno de los dos radio botones. Una vez seleccionado alguno de los dos radio botones aparecerá en pantalla una foto y comenzará a sonar un audio asociado con el radio botón seleccionado. En cualquier momento se puede cambiar la selección del audio con lo cual cambiará el sonido.

Una vez que se ha seleccionado alguno de los dos radio botones se puede controlar el audio con alguno de los tres botones que aparecen en pantalla según la función deseada por el usuario:

- *Play* -> comienza de nuevo el audio seleccionado.
- *Stop* -> para el audio seleccionado si esté esta sonando.
- *Loop* -> comienza de nuevo el audio seleccionado y lo repite continuamente.

Si se selecciona el botón radio *Audio2* comenzará a sonar el audio2 y la situación de la pantalla será algo así:

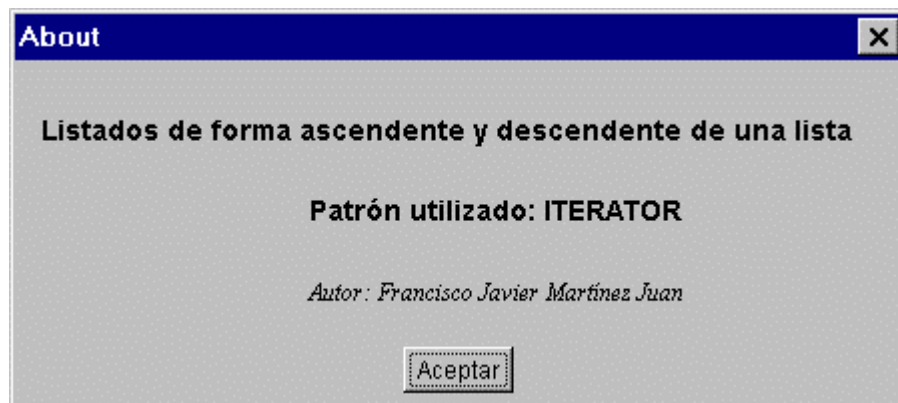


Forma del applet al seleccionar el botón radio de audio2.

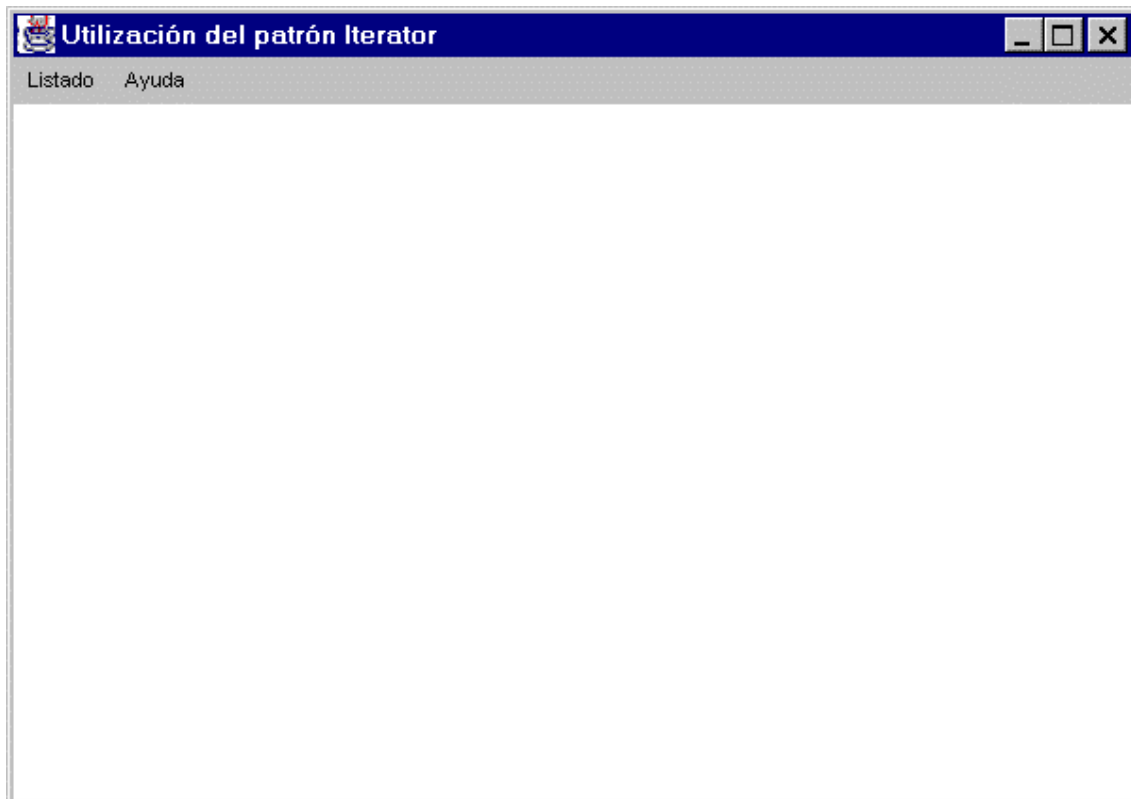
3.2. LISTADOS DE FORMA ASCENDENTE Y DESCENDENTE DE UNA LISTA

3.2.1. ¿En qué consiste la aplicación?

Esta aplicación consiste en realizar el recorrido de una lista de forma ascendente y descendente mostrando por pantalla el listado de cada uno de los recorridos.



Cuadro de Diálogo About.



Forma de la aplicación al ejecutarla.

La lista contiene diez elementos, cada uno de ellos es un entero y corresponden a números del 1 al 10, ambos incluidos.

3.2.2. Patrones de diseño utilizados

Esta aplicación es un ejemplo del patrón Iterator. Este patrón se utiliza para realizar los recorridos, de forma ascendente y descendente, para poder mostrar los respectivos listados de los elementos de la lista por pantalla.

El patrón Iterator ya está implementado en el API de Java, por lo tanto en esta aplicación se utiliza esta ventaja.

Para implementar la colección de elementos se utiliza la clase *LinkedList* del paquete *java.util* del API de Java. Esta clase tiene un método llamado *listIterator* que devuelve un interface *IFIterator* llamado *ListIterator* que permite recorrer la lista en cada dirección. La clase en el paquete *java.util* que implementa la colección define la clase interna privada que implementa el *java.util.ListIterator* y juega el papel *Iterator*.

3.2.3. Diagrama de clases

Esta aplicación consta de tres clases: *Coleccion*, *ListalIterator* y *DialogAbout*. Su diagrama de clases es el siguiente:

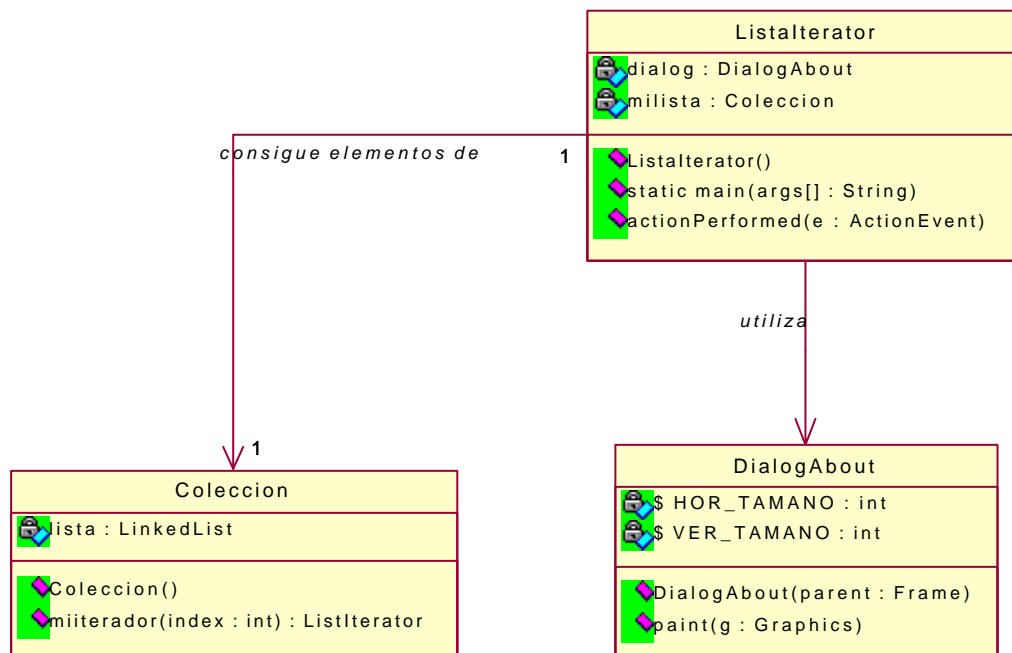


Diagrama de clases.

3.2.4. Código en Java

La clase *Coleccion* implementa la colección de elementos, que en este caso es una lista de tipo *LinkedList* que contiene 10 elementos enteros del 1 al 10 ambos incluidos. La clase *LinkedList* pertenece al paquete *java.util* del API de Java e implementa una lista enlazada.

```
import java.util.*;

/**
 * Esta clase implementa la coleccion de elementos, que en este caso
 * sera una lista de tipo LinkedList que contiene 10 elementos enteros
 * del 1 al 10 ambos incluidos.
 */
public class Coleccion
{
    /**
     * Creamos una lista enlazada llamada lista de tipo LinkedList
     */
    private LinkedList lista=new LinkedList();

    /**
     * Este es el constructor de la clase y lo que hace es insertar en la
     * lista los 10 enteros del 1 al 10 ambos incluidos.
     */
    public Coleccion()
    {
        for(int i=1;i<11;i++)
        {
            lista.add(new Integer(i));
        }
    }
}

//Coleccion
.....
```

La clase *Coleccion* retorna en el método *miiterator* un interface *ListIterator* que permite recorrer la lista en cada dirección y modificar la lista durante el recorrido. Este interface se obtiene a través del método *listIterator* de la clase *LinkedList* pasándole como parámetro la posición específica de la lista por la que empezar el recorrido.

```
/**
 * Retorna un ListIterator que es un interface IFIterator para listas
 * que permite recorrer la lista en cada direccion y modificar la
 * lista durante el recorrido. Este interface se obtiene a través del
 * metodo listIterator de la clase LinkedList pasandole como
 * parámetro La posicion especifica de la lista por la que empezar el
 * recorrido.
 * @param index indica la posición de la lista por la que empezar el
 * recorrido.
 * @return ListIterator es un interface de tipo ListIterator.
 */
public ListIterator miiterator(int index)
```

```
{
    return lista.listIterator(index);
} //miiterator
.....
```

La clase *ListaIterator* es la que implementa la ventana que sale en la pantalla al ejecutar la aplicación y lo que se debe realizar al seleccionar las opciones del menú.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/**
 * Esta clase hereda de Frame y es la que implementa la ventana que
 * sale en la pantalla al ejecutar la aplicación y las opciones que se
 * deben realizar al pulsar las opciones del menu.
 */
public class ListaIterator extends Frame implements ActionListener
{
    private DialogAbout dialog;
    private Coleccion milista=new Coleccion();//crea la lista de 10
    elementos.
    private Label recorrido=new Label();
    private Label nodos=new Label();
    private Label num=new Label();
    .....
}
```

El recorrido de la lista a través del interface *ListIterator* que devuelve el método *miiterator* de la clase *Coleccion* se realiza en el método *actionPerformed* de la clase *ListaIterator* cuando se seleccionan las opciones del Ascendente y Descendente del menú Listado.

```
/**
 * Este método dicta lo que se ha de hacer cada vez que se
 * selecciona alguna opción del menu.
 */
public void actionPerformed (ActionEvent e)
{
    String seleccion=e.getActionCommand();

    Integer numero=new Integer(0);
    int nume;

    if ("Salir".equals(seleccion))//item Salir
    {
        System.exit(0);
    }
    else if ("Ascendente".equals(seleccion))//item Ascendente
    {
        recorrido.setText("RECORRIDO ASCENDENTE");
        nodos.setText("Nodo1  Nodo2  Nodo3  Nodo4  Nodo5
Nodo6  Nodo7  Nodo8  Nodo9  Nodo10");
        nodos.setBounds(50,200,550,30);
        add(nodos);
    }
}
```

```

        //conseguimos el iterador para recorrer la lista
ascendentemente
        ListIterator ascendente=milista.miiterator(0);
        num.setText("");
        //recorremos la lista ascendentemente mientras existan
elementos
        while (ascendente.hasNext())
        {
            numero=(Integer)ascendente.next();
            nume=numero.intValue();
            num.setText(num.getText()+"
"+Integer.toString(nume));
        }
        num.setBounds(30,225,550,30);
        add(num);
    }
    else if ("Descendente".equals(seleccion))//item descendente
    {
        recorrido.setText("RECORRIDO DESCENDENTE");
        nodos.setText("Nodo10   Nodo9   Nodo8   Nodo7   Nodo6
Nodo5   Nodo4   Nodo3   Nodo2   Nodo1");
        nodos.setBounds(50,200,550,30);
        add(nodos);
        //conseguimos el iterador para recorrer la lista
descendentemente
        ListIterator descendente=milista.miiterator(10);
        num.setText("");
        //recorremos la lista descendentemente mientras existan
elementos
        while (descendente.hasPrevious())
        {
            numero=(Integer)descendente.previous();
            nume=numero.intValue();
            num.setText(num.getText()+"
"+Integer.toString(nume));
        }
        num.setBounds(30,225,550,30);
        add(num);
    }
    else if ("About".equals(seleccion))//item About
    {
        int resolucionPantallaAlto=this.getSize().height;
        int resolucionPantallaAncho=this.getSize().width;
        dialog=new DialogAbout(this);
        dialog.setLocation(resolucionPantallaAncho/3,resolucionPantallaAlto/2)
        ;
        dialog.show();
    }
} //actionPerformed(ActionEvent)
.....

```

La clase *DialogAbout* implementa el cuadro de diálogo que sale por pantalla cuando se selecciona la opción About del menú Ayuda. Este cuadro es el que aparece en la primera figura del apartado 3.2.1.

El código completo se muestra en el Anexo B.

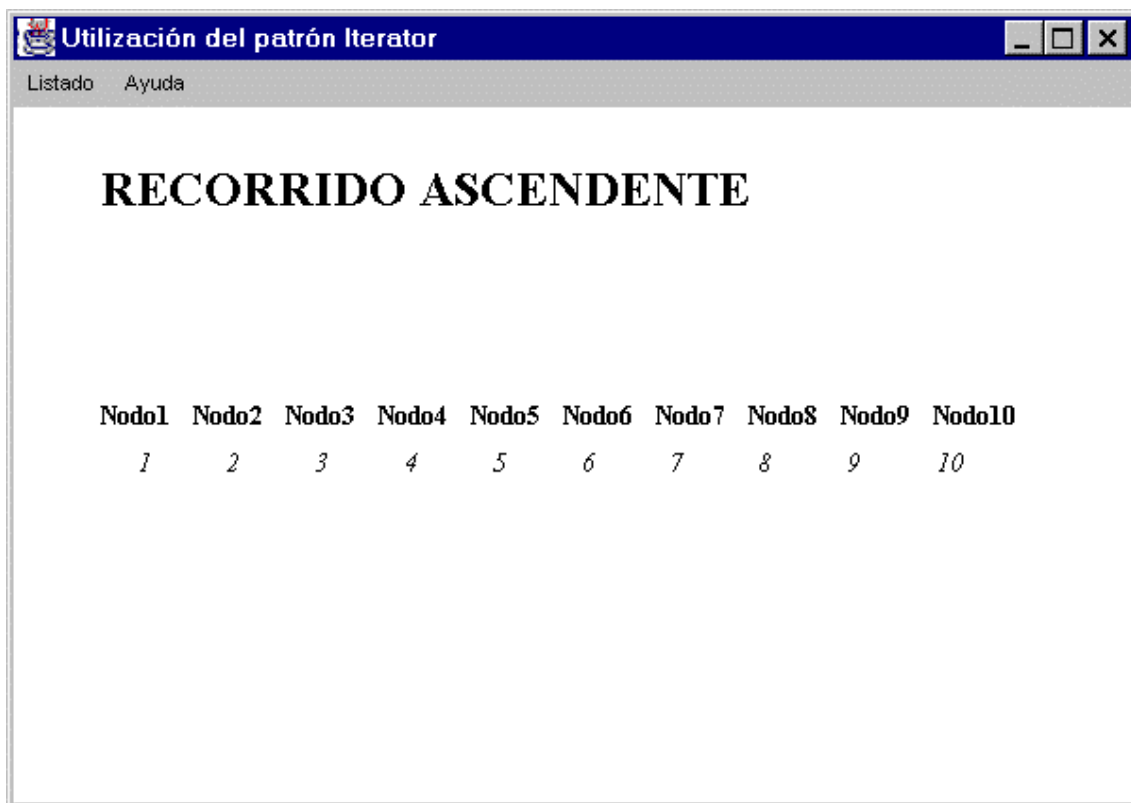
3.2.5. Funcionamiento de la aplicación

La aplicación se puede ejecutar ejecutando el fichero *run.bat* que esta en el directorio *Lista Iterator* bajo MS-DOS.

Cuando se ha ejecutado la aplicación se verá en pantalla una ventana similar a la segunda figura que aparece en el apartado 3.2.1.

El manejo de esta aplicación es muy sencillo. Para realizar el listado ascendente de los diez elementos que tiene la lista se selecciona la opción del menú Listado Ascendente. Para realizar el listado descendente de los diez elementos que tiene la lista se selecciona la opción del menú Listado Descendente. Si lo que se quiere es ver el cuadro de diálogo que aparece en la primera figura del apartado 3.2.1. se selecciona la opción About del menú Ayuda.

Si se selecciona la opción Ascendente la situación de la pantalla será algo así:

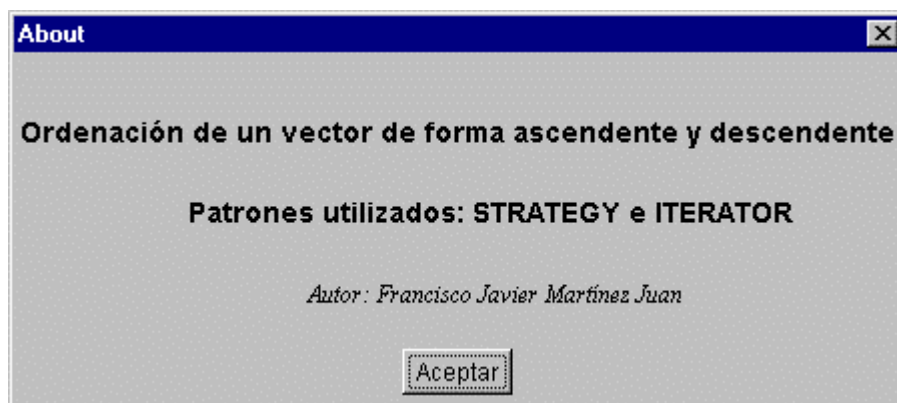


Forma de la aplicación al elegir la opción del menú *Listado Ascendente*.

3.3. ORDENACIÓN DE UN VECTOR DE FORMA ASCENDENTE Y DESCENDENTE

3.3.1. ¿En qué consiste la aplicación?

Esta aplicación consiste en la ordenación de un vector de forma ascendente y descendente por el método de la Burbuja, mostrando por pantalla el vector ordenado.

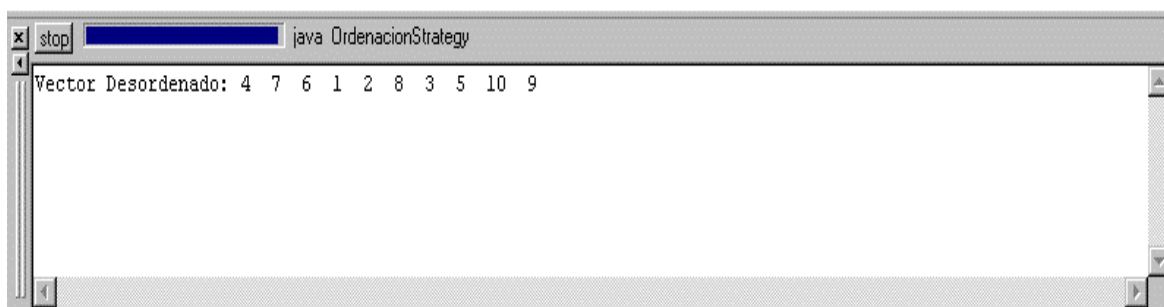


Cuadro de Diálogo About.



Forma de la aplicación al ejecutarla.

El vector contiene diez elementos, cada uno de ellos es un entero y corresponden a números del 1 al 10, ambos incluidos. En el vector no hay números repetidos. Cuando el vector esta desordenado la situación del vector sería esta:



Vector desordenado.

3.3.2. Patrones de diseño utilizados

Esta aplicación es un ejemplo de los patrones Strategy e Iterator. El patrón Strategy se utiliza para encapsular algoritmos relacionados en clases que son subclases de una superclase común. Esto permite la selección y utilización de un algoritmo que varía según el objeto utilizado. El patrón Iterator se utiliza para realizar los recorridos del vector, para poder mostrar el orden de los elementos del vector por pantalla.

El patrón Strategy tiene un interface llamado *Ordenacion* que proporciona una forma común para acceder al algoritmo de ordenación encapsulado en sus subclases, es decir en las clases que implementan el interface *Ordenacion*, que en este caso son: *AlgBurbujaAsc* y *AlgBurbujaDesc*. También existe otra clase llamada *Algoritmos* que es la que controla la selección y uso del objeto particular Strategy. Es decir mediante esta clase se selecciona el algoritmo a utilizar, en nuestro caso *AlgBurbujaAsc* o *AlgBurbujaDesc* y se llama a su método ordenar correspondiente.

El patrón Iterator ya esta implementado en el API de Java, por lo tanto en esta aplicación se utiliza esta ventaja.

Para implementar la colección de elementos se utiliza la clase *Vector* del paquete *java.util* del API de Java. Esta clase tiene un método llamado *listIterator* que devuelve un interface *IFIterator* llamado *ListIterator* que permite recorrer el vector. La clase en el paquete *java.util* que implementa la colección define la clase interna privada que implementa el *java.util.ListIterator* y juega el papel *Iterator*. También se utiliza otro interface *IFIterator* llamado *Enumeration* que también sirve para recorrer el vector. Este interface *Enumeration* lo devuelve un método llamado *elements* de la clase *Vector*.

3.3.3. Diagrama de clases

Esta aplicación consta de un interface y seis clases. El interface es *Ordenacion* y las clases son: *AlgBurbujaAsc*, *AlgBurbujaDesc*, *Algoritmos*, *Coleccion*, *OrdencionStrategy* y *DialogAbout*. Su diagrama de clases es el siguiente:

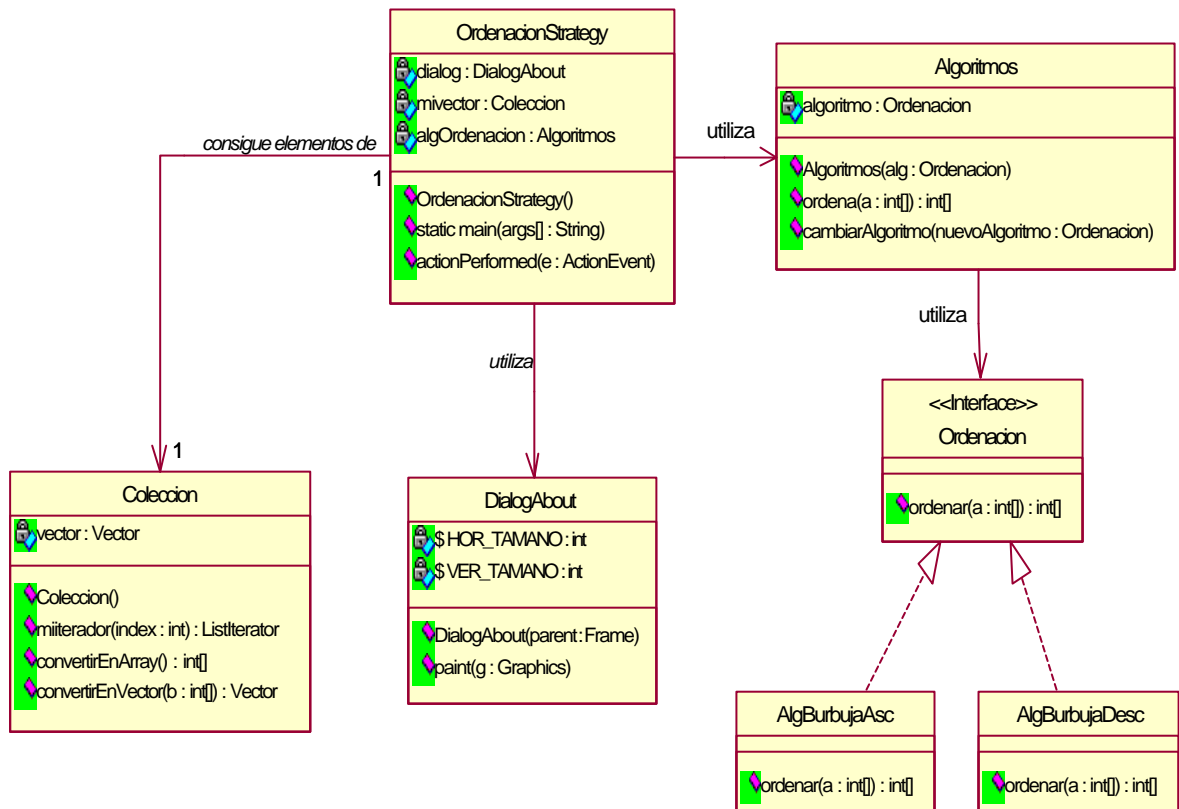


Diagrama de clases.

3.3.4. Código en Java

El interface *Ordenacion* juega el papel de la clase *StrategyAbstracto* del patrón de diseño Strategy. Este interface proporciona una forma común para acceder al algoritmo de ordenación encapsulado en sus subclases, es decir en las clases que implementan el interface, que en esta caso son: *AlgBurbujaAsc* y *AlgBurbujaDesc*.

```

/**
 * Este interface proporciona una forma común para acceder al
 * algoritmo de ordenación encapsulado en sus subclases, es decir
 * en las clases que implementan el interface.
 */
interface Ordenacion

```

```
{
    /**
     * Este método es el que se utiliza para ejecutar cualquiera
     * de los algoritmos de ordenación y será implementado por el
     * algoritmo de ordenación determinado.
     */
    public int[] ordenar(int a[]);
} //interface Ordenacion

/*****
/*****FIN DEL INTERFACE*****/
```

Las clases *AlgBurbujaAsc* y *AlgBurbujaDesc* juegan el papel de las clases *StrategyConcreto1* y *StrategyConcreto2* del patrón Strategy y son los algoritmos de ordenación de la Burbuja ascendente y descendente respectivamente.

```
/**
 * Ordena un vector por el metodo de la Burbuja ascendente.
 */

class AlgBurbujaAsc implements Ordenacion
{
    /*****/

    /**
     * Ordena un vector por el método de la Burbuja descendente.
     * @param a[] el vector a ordenar.
     * @return int[] el vector ordenado.
     */
    public int[] ordenar(int a[])
    {
        for (int i = 1; i <= a.length; ++i)
            for (int j = (a.length) - 1; j >= i; --j)
            {
                if (a[j-1] > a[j])
                {
                    int T = a[j-1];
                    a[j-1] = a[j];
                    a[j] = T;
                }
            }
        return a;
    } //ordenar

    /*****/

} //class AlgBurbujaAsc

/*****/
/*****FIN DE LA CLASE*****/
```

```

/**
 * Ordena un vector por el metodo de la Burbuja descendentemente.
 */

class AlgBurbujaDesc implements Ordenacion
{
    /**
     * Ordena un vector por el método de la burbuja descendentemente.
     * @param a[] el vector a ordenar.
     * @return int[] el vector ordenado.
     */
    public int[] ordenar(int a[])
    {
        for (int i = 1; i <= a.length; ++i)
            for (int j = (a.length) -1; j>= i; --j)
            {
                if (a[j-1]< a[j])
                {
                    int T = a[j-1];
                    a[j-1] = a[j];
                    a[j] = T;
                }
            }
        return a;
    } //ordenar

    /**
     *
     */
} //class AlgBurbujaDesc

/**
 *
 */
/**FIN DE LA CLASE**

```

La clase *Algoritmos* controla la selección y uso del objeto particular strategy. Es decir mediante esta clase se selecciona el algoritmo a utilizar en este caso *AlgBurbujaAsc* o *AlgBurbujaDesc* y se llama a su método ordenar correspondiente. Esta clase juega el papel de la clase *Cliente* del patrón Strategy.

```

/**
 * Esta clase controla la selección y uso del objeto particular
 * strategy. Es decir mediante esta clase se selecciona el algoritmo
 * a utilizar en nuestro caso AlgBurbujaAsc o AlgBurbujaDesc y se
 * llama a su método ordenar correspondiente.
 */
public class Algoritmos
{
    private Ordenacion algoritmo; //referencia al interface Ordenacion.

    /**
     *
     */

```

```

/**
 * Constructor de la clase.
 * @param alg es un objeto de tipo ordenacion con el cual
 * seleccionamos el algoritmo que queremos utilizar.
 */
public Algoritmos(Ordenacion alg)
{
    algoritmo=alg;
} //Algoritmos

/*****

/**
 * Con este método llamamos al método ordenar del algoritmo elegido.
 * @param a[] el vector a ordenar.
 * @return int[] el vector ordenado.
 */
public int[] ordena (int[] a)
{
    return algoritmo.ordenar(a);
} //ordena

/*****

/**
 * Este método sirve para cambiar la selección del algoritmo.
 * @param nuevoAlgoritmo el nuevo algoritmo seleccionado.
 */
public void cambiarAlgoritmo (Ordenacion nuevoAlgoritmo)
{
    algoritmo=nuevoAlgoritmo;
} //cambiarAlgoritmo

/*****

} //class Algoritmos

/*****
/*****FIN DE LA CLASE*****/

```

La clase *Coleccion* implementa la colección de elementos, que en este caso es un vector de tipo *Vector* que contiene 10 elementos enteros desordenados ({4, 7, 6, 1, 2, 8, 3, 5, 10, 9}). La clase *Vector* pertenece al paquete *java.util* del API de Java e implementa un array de objetos.

La clase *Coleccion* retorna en el método *miiterator* un interface *ListIterator* que permite recorrer el vector. Este interface se obtiene a través del método *listIterator* de la clase *Vector* pasándole como parámetro la posición específica del vector por la que empezar el recorrido. Para la clase *Vector* también existe en el API de Java otro interface de tipo *IFIterator* llamado *Enumeration* que se obtiene a través del método *elements* de la clase *Vector*.

```
import java.util.*;
```

```

/**
 * Esta clase implementa la coleccion de elementos, que en este caso
 * sera un vector de tipo Vector que contiene 10 elementos enteros
 * desordenados {4,7,6,1,2,8,3,5,10,9}
 */
public class Coleccion
{
    private static Vector vector=new Vector(9);

    /**
     * Este es el constructor de la clase y lo que hace es insertar en el
     * vector los 10 enteros desordenados y se imprime por pantalla el
     * contenido del vector.
     */
    public Coleccion()
    {
        vector.clear();
        vector.addElement(new Integer(4));
        vector.addElement(new Integer(7));
        vector.addElement(new Integer(6));
        vector.addElement(new Integer(1));
        vector.addElement(new Integer(2));
        vector.addElement(new Integer(8));
        vector.addElement(new Integer(3));
        vector.addElement(new Integer(5));
        vector.addElement(new Integer(10));
        vector.addElement(new Integer(9));
        // vector={4,7,6,1,2,8,3,5,10,9}
        System.out.print("Vector Desordenado: ");
        for (Enumeration e = vector.elements() ; e.hasMoreElements() ; )
        {
            Integer a;
            a=(Integer)e.nextElement();
            System.out.print(a.intValue()+" ");

        }
        System.out.println();
    } //Coleccion
}
.....

```

```

/**
 * Retorna un ListIterator que es un interface IFilterator para
 * vectores que permite recorrer el vector. Este interface se obtiene
 * a través del metodo listIterator de la clase Vector pasandole como
 * parámetro La posicion especifica del vector por la que empezar
 * el recorrido.
 * @param index indica la posición del vector por la que empezar el
 * recorrido.
 * @return ListIterator es un interface de tipo ListIterator.
 */
public ListIterator miiterator(int index)
{
    return vector.listIterator(index);
}

```

```
//miiterator
```

La clase OrdenacionStrategy es la que implementa la ventana que sale en la pantalla al ejecutar la aplicación y lo que se debe realizar al seleccionar las opciones del menú.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/**
 * Esta clase hereda de Frame y es la que implementa la ventana que
 * sale en la pantalla al ejecutar la aplicación y las opciones que se
 * deben realizar al pulsar las opciones del menu.
 */
public class OrdenacionStrategy extends Frame implements
ActionListener
{
    private DialogAbout dialog;
    private Coleccion mivector;
    private Algoritmos algOrdenacion;//para seleccionar y utilizar el
                                   //algoritmo deseado.

    private Label ordenacion=new Label();
    private Label nodos=new Label();
    private Label num=new Label();

    .....
```

La ordenación del vector se hace a través de la clase *Algoritmos*, utilizando el método *ordena* de dicha clase . Una vez ordenado el vector se muestra el vector por pantalla a través del interface *ListIterator* que devuelve el método *miiterator* de la clase *Coleccion*. Esto se realiza en el método *ActionPerformed* de la clase *OrdenacionStrategy* cuando se seleccionan las opciones Ascendente y descendente del menú Ordenación.

```
/**
 * Este método dicta lo que se ha de hacer cada vez que se
 * selecciona alguna opción del menu.
 */
public void actionPerformed (ActionEvent e)
{
    String seleccion=e.getActionCommand();
    int[] miarray=new int[10];
    Integer numero=new Integer(0);
    int nume;

    if ("Salir".equals(seleccion))//item Salir
    {
        System.exit(0);
    }
    else if ("Ascendente".equals(seleccion))//item Ascendente
    {
        ordenacion.setText("ORDENACIÓN ASCENDENTE");
    }
}
```

```

        nodos.setText("Nodo1  Nodo2  Nodo3  Nodo4  Nodo5
Nodo6  Nodo7  Nodo8  Nodo9  Nodo10");
        nodos.setBounds(50,200,550,30);
        add(nodos);
        mivector=new Coleccion();//crea el vector de 10 elementos.

        //Seleccionamos el algoritmo deseado.
        algOrdenacion=new Algoritmos(new AlgBurbujaAsc());
        //Y utilizamos el algoritmo deseado.
        miarray=algOrdenacion.ordena(mivector.convertirEnArray());
        mivector.convertirEnVector(miarray);

        //conseguimos el iterador para recorrer la lista
ascendentemente
        ListIterator ascendente=mivector.miiterator(0);
        num.setText("");
        //recorremos la lista ascendentemente mientras existan
elementos
        while (ascendente.hasNext())
        {
            numero=(Integer)ascendente.next();
            nume=numero.intValue();
            num.setText(num.getText()+"
"+Integer.toString(nume));
        }
        num.setBounds(30,225,550,30);
        add(num);
    }
    else if ("Descendente".equals(seleccion))//item descendente
    {
        ordenacion.setText("ORDENACIÓN DESCENDENTE");
        nodos.setText("Nodo1  Nodo2  Nodo3  Nodo4  Nodo5
Nodo6  Nodo7  Nodo8  Nodo9  Nodo10");
        nodos.setBounds(50,200,550,30);
        add(nodos);
        mivector=new Coleccion();//crea el vector de 10 elementos.

        //Seleccionamos el algoritmo deseado.
        algOrdenacion=new Algoritmos(new AlgBurbujaDesc());
        // Y utilizamos el algoritmo deseado.
        miarray=algOrdenacion.ordena(mivector.convertirEnArray());
        mivector.convertirEnVector(miarray);

        //conseguimos el iterador para recorrer la lista
ascendentemente
        ListIterator ascendente=mivector.miiterator(0);
        num.setText("");
        //recorremos la lista ascendentemente mientras existan
elementos
        while (ascendente.hasNext())
        {
            numero=(Integer)ascendente.next();
            nume=numero.intValue();
            num.setText(num.getText()+"
"+Integer.toString(nume));
        }
        num.setBounds(30,225,550,30);
        add(num);
    }

```

```

    }
    else if ("About".equals(seleccion))//item About
    {
        int resolutionPantallaAlto=this.getSize().height;
        int resolutionPantallaAncho=this.getSize().width;
        dialog=new DialogAbout(this);

dialog.setLocation(resolucionPantallaAncho/3,resolucionPantallaAlto/2)
;
        dialog.show();
    }
} //actionPerformed(ActionEvent)

.....

```

La clase *DialogAbout* implementa el cuadro de diálogo que sale por pantalla cuando se selecciona la opción About del menú Ayuda. Este cuadro es el que aparece en la primera figura del apartado 3.3.1.

El código completo se muestra en el Anexo C.

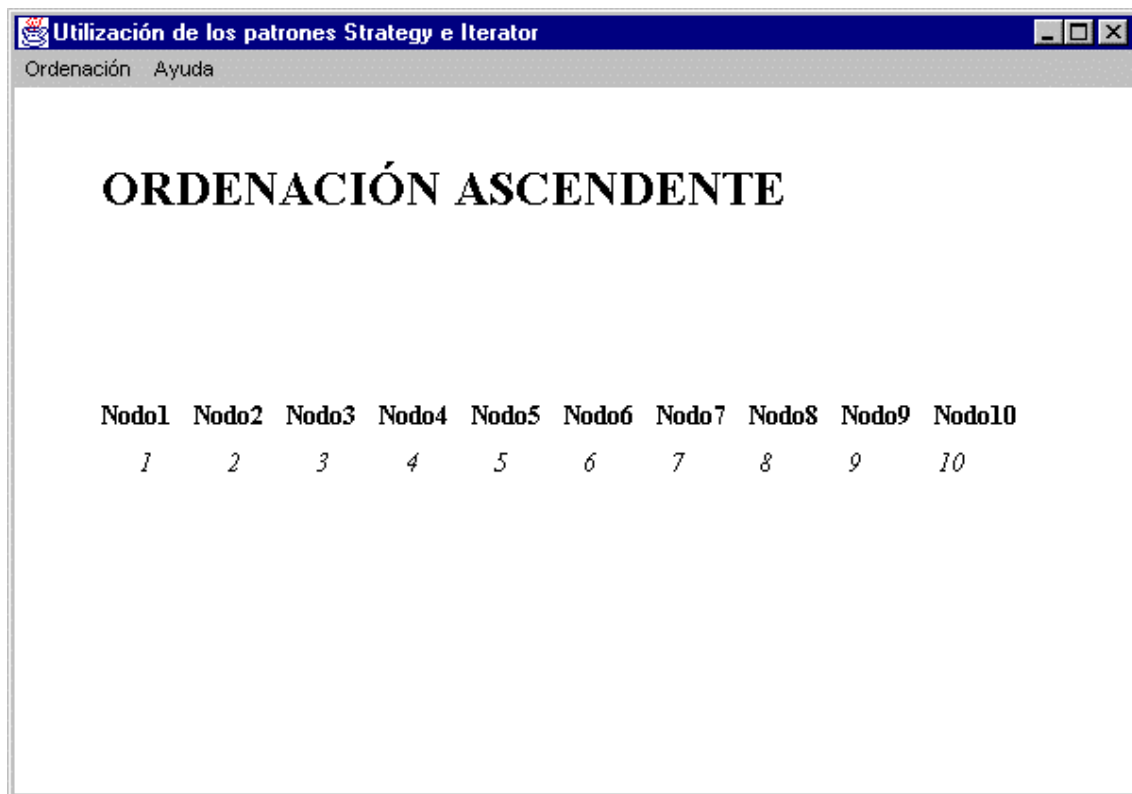
3.3.5. Funcionamiento de la aplicación

La aplicación se puede ejecutar ejecutando el fichero *run.bat* que esta en el directorio *Ordenacion Strategy* bajo MS-DOS.

Cuando se ha ejecutado la aplicación se verá en pantalla una ventana similar a la segunda figura que aparece en el apartado 3.3.1.

El manejo de esta aplicación es muy sencillo. Para realizar la ordenación ascendente de los diez elementos que tiene el vector se selecciona la opción del menú Ordenación Ascendente. Para realizar la ordenación descendente de los diez elementos que tiene el vector se selecciona la opción del menú Ordenación Descendente. Si lo que se quiere es ver el cuadro de diálogo que aparece en la primera figura del apartado 3.3.1. se selecciona la opción About del menú Ayuda.

Si se selecciona la opción Ascendente la situación de la pantalla será algo así:



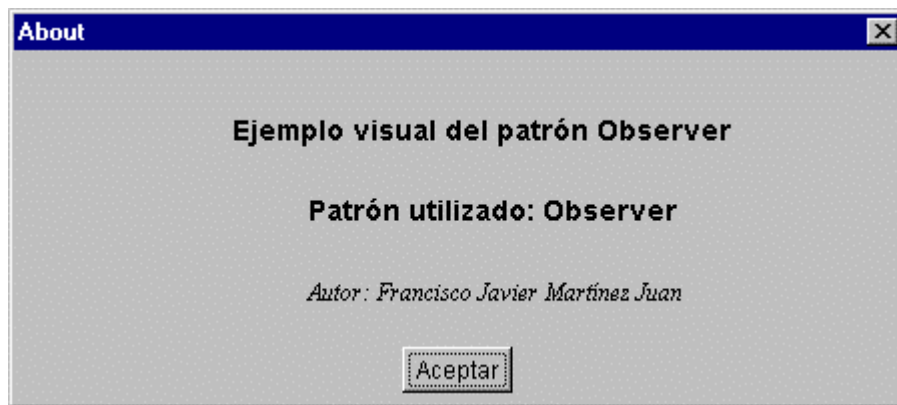
Forma de la aplicación al elegir la opción del menú *Ordenación Ascendente*.

3.4. EJEMPLO VISUAL DEL PATRÓN OBSERVER

3.4.1. ¿En qué consiste la aplicación?

Esta aplicación consiste en la representación del estado interno de una clase, que en este caso consiste en un valor entero, de diferentes formas. La aplicación también permite al usuario introducir un nuevo valor para alterar el estado interno, es decir el valor del entero.

Al realizar una modificación del estado interno se notificará este suceso a todos los objetos que estén como observadores de ese estado, con lo cual todos estos objetos cambiarán su vista al cambiar el estado interno (el valor del entero).



Cuadro de Diálogo About.

También se puede crear cualquier número de observadores, del valor entero, de dos tipos diferentes: Observador de Texto y Observador de Barra con sólo pulsar los respectivos botones que aparecen en la pantalla al ejecutar la aplicación.

El Observador de Texto es algo así:



El Observador de Barra es algo así:



Forma de la aplicación al ejecutarla.

3.4.2. Patrones de diseño utilizados

Esta aplicación es un ejemplo del patrón Observer. El patrón Observer permite captar dinámicamente las dependencias entre objetos, de tal forma que un objeto notificará a los objetos dependientes de él cuando cambia su estado, siendo actualizados automáticamente.

Así pues, en nuestro caso cada vez que se cambie el estado interno, es decir, el valor del entero, se actualizarán automáticamente todos los observadores (observers) que existan de ese estado. Al decir todos los observadores serán tanto los Observadores de Texto como los Observadores de Barra y tantos como existan de cada tipo.

El API de Java proporciona parte del patrón Observer. En el paquete *java.util* hay un interface llamado *Observer* (juega el papel *IFObserver*) y una clase llamada *Observable* (juega el papel *Observable*):

- **Observer:** Es cualquier objeto que desee ser notificado cuando el estado de otro objeto sea alterado.

- **Observable:** Es cualquier objeto cuyo estado puede representar interés y sobre el cual otro objeto ha demostrado ese interés.

Este interface *Observer* y esta clase *Observable* son útiles en cualquier sistema en que se necesite que algunos objetos sean notificados cuando ocurran cambios en otros objetos.

A continuación vamos a exponer los métodos que contienen:

Observer

public void update (Observable obs, Object obj)

Llamada cuando se produce un cambio en el estado del objeto Observable.

Observable

public void addObserver (Observer obs)

Añade un observador a la lista interna de observadores.

public void deleteObserver (Observer obs)

Borra un observador de la lista interna de observadores.

public void deleteObservers()

Borra todos los observadores de la lista interna.

public int countObserver()

Devuelve el número de observadores en la lista interna.

protected void setChanged()

Levanta el flag interno que indica que el Observable ha cambiado de estado.

protected void clearChanged()

Baja el flag interno que indica que el Observable ha cambiado de estado.

protected boolean hasChanged()

Devuelve un valor booleano indicando si el Observable ha cambiado.

public void notifyObservers()

Comprueba el flag interno para ver si el Observable ha cambiado de estado y lo notifica a todos los observadores.

```
public void notifyObservers (Object obj)
```

Comprueba el flag interno para ver si el Observable ha cambiado de estado y lo notifica a todos los observadores. Les pasa el objeto especificado en la llamada para que lo usen los observadores en su método `update()`.

Las clases que encapsulan a los observadores: *TextoObservador* y *BarraObservador* implementan el interface *Observer*.

La clase *ValorObservable* es la que encapsula el valor del entero que puede cambiar de estado hereda de la clase *Observable*.

3.4.3. Diagrama de clases

Esta aplicación consta de cinco clases: *ValorObservable*, *TextoObservador*, *BarraObservador*, *DialogAbout* y *PatronObserver*. Su diagrama de clases es el siguiente:

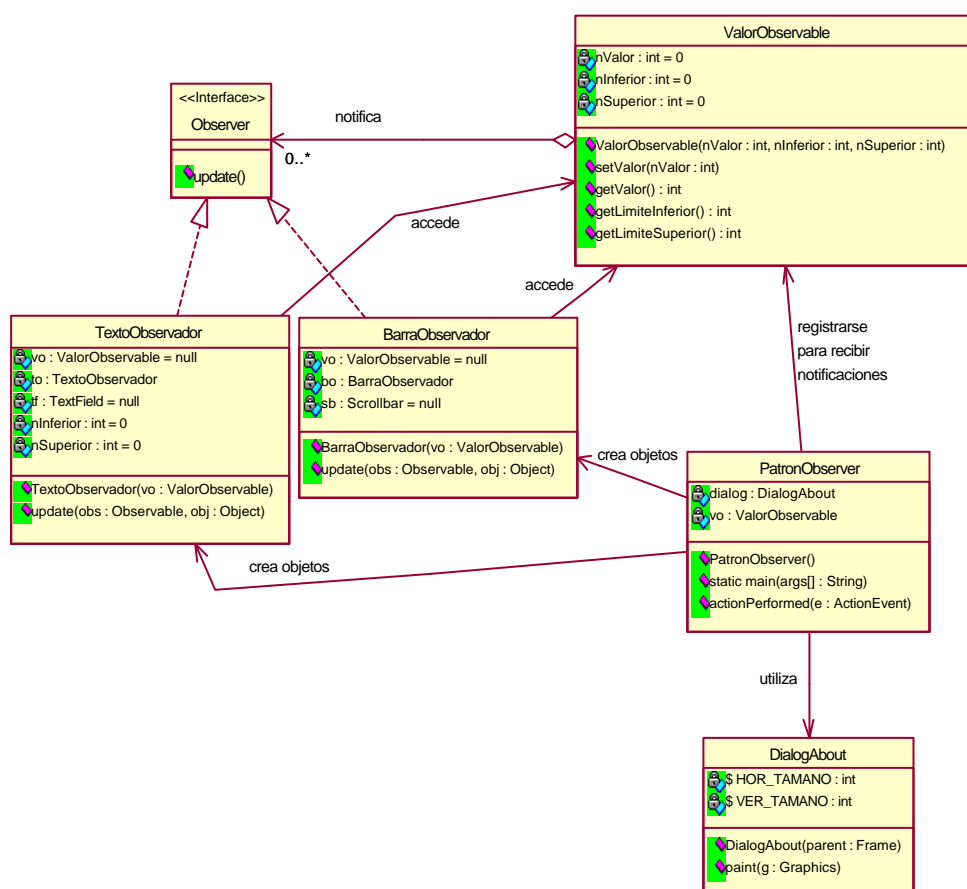


Diagrama de clases.

En el diagrama de clases del patrón Observer había una clase *Multicaster* que aquí no aparece y esto es debido a que se ha utilizado una simplificación citada en el apartado 2.4.6 (Implementación) ya que solo existe una única clase Observable que envía notificaciones.

3.4.4. Código en Java

La clase *ValorObservable* hereda de la clase Observable del paquete *java.util* del API de Java. Esta clase juega el papel de la clase *Observable* del patrón Observer.

```
import java.util.Observable;

/**
 * Esta clase representa la clase Observable y es la que
 * notifica a los observadores (Observers) que estan
 * pendientes de los cambios de estado de esta clase, que
 * su estado se ha visto alterado.
 */
public class ValorObservable extends Observable
{
    private int nValor = 0;
    private int nInferior = 0;
    private int nSuperior = 0;

    /*****

    /**
     * Constructor al que indicamos el valor en que comenzamos
     * y los limites inferior y superior que no deben
     * sobrepasarse.
     */
    public ValorObservable( int nValor,int nInferior,int nSuperior )
    {
        this.nValor = nValor;
        this.nInferior = nInferior;
        this.nSuperior = nSuperior;
    } //ValorObservable(int, int, int)

    .....
}
```

Como la clase Observable ya implementa todos los métodos necesarios para proporcionar el funcionamiento de tipo Observador/Observable, la clase derivada solamente necesita proporcionar acceso al estado interno del objeto Observable.

En la clase *ValorObservable* el estado interno es capturado en el entero *nValor*. A este valor se accede y se modifica solamente a través de sus métodos públicos. Si el valor cambia, el objeto invoca a su propio método *setChanged()* para indicar que ha cambiado el estado de la clase *Observable* ha cambiado. Luego invoca a su propio método *notifyObservers()* para actualizar a todos los observadores registrados (*Observers*).

```

/**
 * Fija el valor que le pasamos y notifica a los observadores que
 * estan pendientes del cambio de estado de los objetos de esta
 * clase, que su estado se ha visto alterado.
 */
public void setValor(int nValor)
{
    this.nValor = nValor;
    setChanged();
    notifyObservers();
} //setValor(int)

/*****

/**
 * Devuelve el valor actual que tiene el objeto.
 */
public int getValor()
{
    return( nValor );
} //getValor

.....

```

Las clases *TextoObservador* y *BarraObservador* implementan el interface *Observer* del paquete *java.util* del API de Java. Este interface juega el papel de la clase *IObserver* del patrón *Observer* y las clases *TextoObservador* y *BarraObservador* juegan el papel *Observer* del patrón *Observer*, es decir son observadores de la clase *ValorObservable*.

```

import java.awt.*;
import java.awt.event.*;
import java.util.Observer;
import java.util.Observable;

/**
 * Esta clase representa un observador (Observer) que es un TextField,
 * que es un campo de entrada de texto de una sola linea.
 * Esta linea de texto aparecerá en una ventana independiente.
 * Se crea un nuevo objeto de esta clase, es decir una nueva
 * línea de entrada de texto cada vez que se pulse el botón
 * "Nuevo Observador Texto".
 */
public class TextoObservador extends Frame implements Observer
{
    private ValorObservable vo = null;
    private TextoObservador to;
    private TextField tf = null;
    private Label l = null;
    private int nInferior = 0;
    private int nSuperior = 0;

    .....

```

```

import java.awt.*;
import java.awt.event.*;

```



```
import java.util.Observer;
import java.util.Observable;

/**
 * Esta clase representa un observador (Observer) que es una barra de
 * desplazamiento. Esta barra de desplazamiento aparecerá
 * en una ventana independiente. Se crea un nuevo objeto
 * de esta clase es decir una nueva barra de desplazamiento
 * cada vez que se pulse el botón "Nuevo Observador Barra".
 */
public class BarraObservador extends Frame implements Observer
{
    private ValorObservable vo = null;
    private BarraObservador bo;
    private Scrollbar sb = null;
    .....
}
```

Las clases que implementan el interface Observer (*TextoObservador* y *BarraObservador*) tienen un método *update()*. Este método será llamado siempre que el Observable (clase *ValorObservable*) cambie de estado, que anuncia este cambio llamando a su método *notifyObservers()*. Los observadores actualizan el valor del objeto que están observando, para reflejar el valor actual del objeto. También pueden modificar el estado de la clase *ValorObservable*.

La clase *TextoObservador* en el método *update()* actualiza el valor del objeto que esta observando en su campo de texto, para reflejar el valor actual del objeto.

```
/**
 * Actualizamos el valor del objeto que estamos observando
 * en nuestro campo de texto, para reflejar el valor actual
 * del objeto.
 */
public void update( Observable obs, Object obj )
{
    if( obs == vo )
        tf.setText( String.valueOf( vo.getValor() ) );
} //update(Observable, Object)
.....
}
```

Esta clase también puede modificar el estado de la clase *ValorObservable*.

```
/**
 * Esta clase maneja los events que se producen en el campo
 * de texto.
 */
class ManejadorTextField implements ActionListener
{
    /**
     * Controlamos el evento que proviene del campo de texto,
     * cuando se introduce un valor numerico.
     */
    .....
}
```

```

    */
    public void actionPerformed((ActionEvent e)
    {
        int n = 0;
        boolean bValido = false;

        try
        {
            n = Integer.parseInt( tf.getText() );
            bValido = true;
        }
        catch( NumberFormatException nfe )
        {
            bValido = false;
        }

        // Comprobamos que no se sobrepasen los limites que hemos
        // fijado
        if( n < nInferior || n > nSuperior )
            bValido = false;

        if( bValido )
        {
            vo.setValor( n );
            l.setText( "" );
        }
        else
            l.setText( "Valor no válido -> inténtelo de nuevo:
["+nInferior+"-"+nSuperior+"]" );
    } //actionPerformed(ActionEvent)

    /*****

    */ //class ManejadorTextField

    .....

```

Al cerrarse la ventana del Observador de Texto hay que borrar el observador de la lista interna de observadores de la clase *ValorObservable*.

```

    /**
     * Esta clase maneja los eventos que se producen sobre la
     * ventana del campo de texto.
     */
    class WindowEventHandler extends WindowAdapter
    {
        /*****

        /**
         * Controlamos el cierre de la ventana, para eliminar el
         * objeto que hemos creado antes de hacer desaparecer
         * la ventana.
         */
        public void windowClosing( WindowEvent e )
        {
            vo.deleteObserver(to);

```

```

        dispose();
    } //windowClosing(WindowEvent)

    /**
    */
} //class WindowEventHandler

.....

```

La clase *BarraObservador* en el método *update()* actualiza el valor del objeto que esta observando en la barra de desplazamiento, para reflejar el valor actual del objeto.

```

/**
 * Actualizamos el valor que corresponde al número que
 * actualmente tiene el objeto que estamos observando.
 */
public void update( Observable obs, Object obj )
{
    if( obs == vo )
        sb.setValue( vo.getValor() );
} //update(Observable, Object)

.....

```

Esta clase también puede modificar el estado de la clase *ValorObservable*.

```

/**
 * Esta clase maneja los eventos que se producen sobre la
 * barra de desplazamiento.
 */
class ManejadorScrolling implements AdjustmentListener
{
    /**
    * Manejamos los eventos que se producen al manipular la
    * barra. Cuando desplazamos el marcador de la barra, vamos
    * actualizando el valor del objeto observable y presentamos
    * el valor que se ha adquirido.
    */
    public void adjustmentValueChanged( AdjustmentEvent e)
    {
        if( e.UNIT_INCREMENT == AdjustmentEvent.UNIT_INCREMENT )
        {
            vo.setValor( sb.getValue() );
        }
        else if( e.UNIT_DECREMENT == AdjustmentEvent.UNIT_DECREMENT )
        {
            vo.setValor( sb.getValue() );
        }
        else if( e.BLOCK_INCREMENT == AdjustmentEvent.BLOCK_INCREMENT )
        {
            vo.setValor( sb.getValue() );
        }
        else if( e.BLOCK_DECREMENT == AdjustmentEvent.BLOCK_DECREMENT )
        {

```

```

        vo.setValor( sb.getValue() );
    }
    else if( e.TRACK == AdjustmentEvent.TRACK )
    {
        vo.setValor( sb.getValue() );
    }
} //adjustValueChanged(AdjustmentEvent)

/*****

} //class ManejadorScrolling

```

Al cerrarse la ventana del Observador de Barra hay que borrar el observador de la lista interna de observadores de la clase *ValorObservable*.

```

/**
 * Esta clase maneja los eventos sobre la ventana de la
 * barra de desplazamiento.
 */
class WindowEventHandler extends WindowAdapter
{
/*****

/**
 * Controlamos el cierre de la ventana, para eliminar el
 * objeto que hemos creado antes de hacer desaparecer
 * la ventana.
 */
public void windowClosing( WindowEvent e )
{
    vo.deleteObserver(bo);
    dispose();
} //windowClosing(WindowEvent)

/*****
} //class WindowEventHandler

```

La clase *PatronObserver* es la que implementa la ventana que sale por pantalla al ejecutar la aplicación y lo que se debe realizar al seleccionar las opciones del menú o pulsar los botones.

```

import java.awt.*;
import java.awt.event.*;
import java.util.Observable;

/**
 * Esta clase hereda de Frame y es la que implementa la ventana que
 * sale en la pantalla al ejecutar la aplicación y las opciones que se
 * deben realizar al pulsar los botones y las opciones del menú.
 */
public class PatronObserver extends Frame implements ActionListener
{
    private DialogAbout dialog;

    private Button bObservTexto;

```

```
private Button bObservBarra;
private ValorObservable vo;

private Label titulo=new Label();
```

Al pulsar los botones se crean nuevos observadores, ya sean de texto o de barra, y se añaden a la lista interna de observadores de la clase *ValorObservable*, llamando a su método *addObserver()* del objeto observable, para que el objeto Observable les pueda notificar cuando su estado se altere.

```
/**
 * Este método dicta lo que se ha de hacer cada vez que se
 * selecciona alguna opción del menú o se pulsa alguno de los
 * botones.
 */
public void actionPerformed (ActionEvent e)
{
    String s=e.getActionCommand();

    if( "Nuevo Observador Texto".equals(s) )//botón Nuevo
Observador Texto
    {
        TextoObservador textoObserv = new TextoObservador( vo );
        vo.addObserver( textoObserv );
    }
    else if("Nuevo Observador Barra".equals(s))//botón Nuevo
Observador Barra
    {
        BarraObservador barraObserv = new BarraObservador( vo );
        vo.addObserver( barraObserv );
    }

    if ( "Salir".equals(s) )//item Salir
    {
        System.exit(0);
    }
    else if ( "About".equals(s) )//item About
    {
        int resolutionPantallaAlto=this.getSize().height;
        int resolutionPantallaAncho=this.getSize().width;
        dialog=new DialogAbout( this );

        dialog.setLocation(resolutionPantallaAncho/3,resolutionPantallaAlto/2)
        ;

        dialog.show();
    }
} //actionPerformed(ActionEvent)
```

En la siguiente secuencia, vamos a describir como se realiza la interacción entre un Observador y un objeto Observable, durante la ejecución del programa:

1. En primer lugar el usuario manipula un elemento del interface de usuario. De esta forma se realiza un cambio de estado en el Observable, a través de uno de sus métodos públicos de acceso; en nuestro caso, llama a *setValor()*.
2. El método público de acceso modifica el dato privado, ajusta el estado interno y llama al método *setChanged()* para indicar que su estado ha cambiado. Luego llama al método *notifyObservers()* para notificar a los observadores que su estado ha cambiado.
3. Se llama a los métodos *update()* de cada Observador, indicando que hay un cambio en el estado del objeto que estaban observando. El Observador accede entonces a los datos del Observable a través del método público del Observable y se actualiza.

La clase *DialogAbout* implementa el cuadro de diálogo que sale por pantalla cuando se selecciona la opción *About* del menú *Ayuda*. Este cuadro es el que aparece en la primera figura del apartado 3.4.1.

El código completo se muestra en el Anexo D.

3.4.5. Funcionamiento de la aplicación

La aplicación se puede ejecutar ejecutando el fichero *run.bat* que esta en el directorio *Observer* bajo MS-DOS.

Cuando se ha ejecutado la aplicación se verá en pantalla una ventana similar a la cuarta figura que aparece en el apartado 3.4.1.

Al pulsar sobre los botones que aparecen en la ventana nos aparecerán otras ventanas en la parte superior izquierda de la pantalla que dependiendo del botón pulsado serán unas ventanas parecidas a las que aparecen en la segunda y tercera figura del apartado 3.4.1.

Cada vez que se pulse un botón aparecerá una nueva ventana, la cual podemos colocarla en cualquier parte de la pantalla de tal forma que se puedan ver el cambio automático de todas las ventanas observadoras al realizar un cambio del valor del entero.

Para cambiar el valor del entero sobre una ventana de Observador de Texto nos colocamos en el campo de texto introducimos el valor del entero deseado y le damos a intro. Una vez realizado este cambio todas las ventanas observadoras habrán cambiado al nuevo valor.

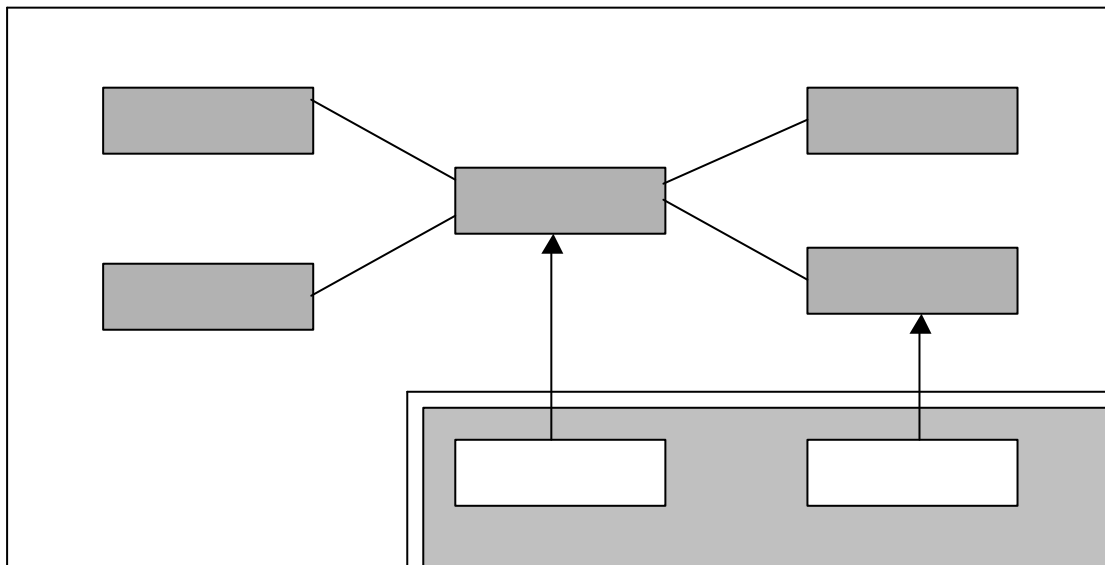
Para cambiar el valor del entero sobre una ventana de Observador de Barra se hace a través de movimientos del marcador de la barra por acción del usuario. La posición del marcador en la barra representa el valor actual que corresponde con el estado del entero. Al realizar los cambios sobre el Observador de Barra se verá como todas las ventanas observadoras son actualizadas automáticamente.

3.5. JHOTDRAW 5.1

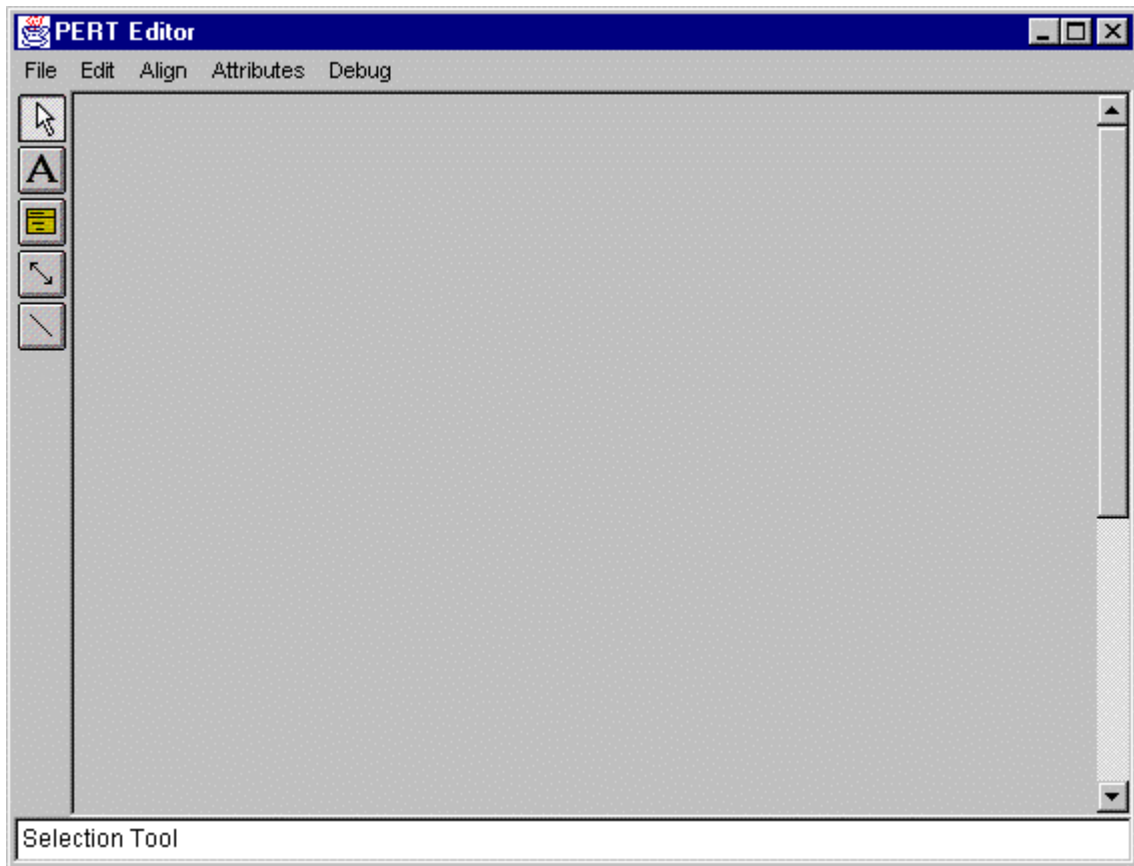
3.5.1. ¿En qué consiste la aplicación?

JHotDraw es un framework especializado para editores gráficos. JHotDraw es un framework para editores de manipulación directa con las siguientes características: distintos tipos de figuras, manipulación directa de las figuras, conexión de figuras, herramientas, actualización en pantalla eficiente, applets y aplicaciones. JHotDraw es un editor gráfico que tiene que dibujar sin saber lo que dibuja.

Un framework es un diseño reutilizable de un programa o parte de un programa expresado como un conjunto de clases, que tiene un comportamiento y hace una cosa por omisión, por ejemplo un editor gráfico. Como todo software, es una mezcla de lo concreto y lo abstracto. Los frameworks son diseños reutilizables no solo en código, son más abstractos que la mayoría del software, lo cual hace que documentarlos sea más difícil. Los frameworks son diseñados por expertos en un dominio particular y luego utilizados por no expertos.



En el gráfico anterior los rectángulos más oscuros representarían las clases que implementa el framework. Los rectángulos en blanco serían lo que implementamos nosotros para dar una funcionalidad y utilidad al framework, por ejemplo para hacer gráficos PERT (el PERT es un diagrama de programación de tareas, es un instrumento para la toma de decisiones que nos permite la planificación, programación y control de proyectos que requieren la coordinación de un elevado número de actividades entre las cuales existen relaciones de precedencia y que consumen unos tiempos y recursos limitados).



Forma de la aplicación Pert al ejecutarla.

Un framework es una colaboración de clases adaptables que definen una solución para un problema dado:

- Define las abstracciones fundamentales y sus interfaces.
- Establece las interacciones entre los objetos.
- Adaptación: redefinición.
- Soluciones por defecto.

Un framework se caracteriza por su capacidad de reutilización tanto a nivel de arquitectura y diseño como a nivel de código. Un framework tiene que ser muy flexible y fácil de modificar para seguir creciendo, es decir ha de tener máxima flexibilidad. Un framework separa muy bien el interfaz y el diseño.

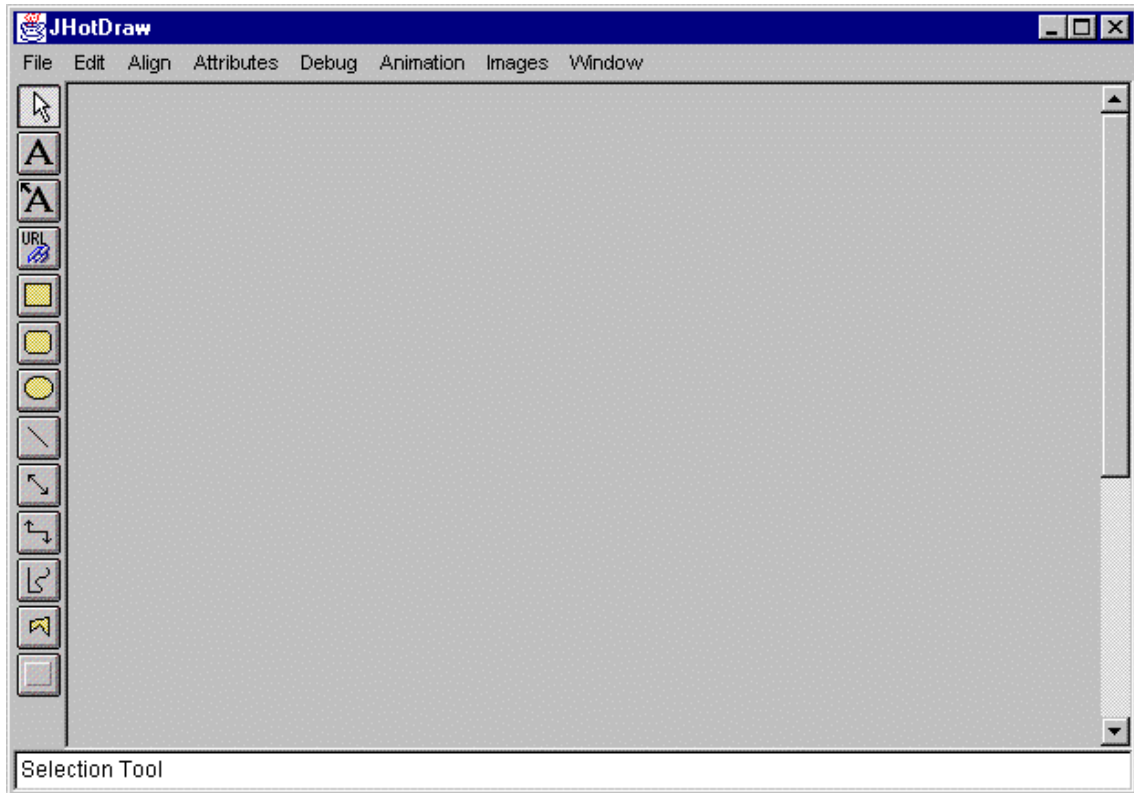
Un framework crece por agregación o por herencia pero no reimplementando código.

La arquitectura JHotDraw se ha inspirado en:

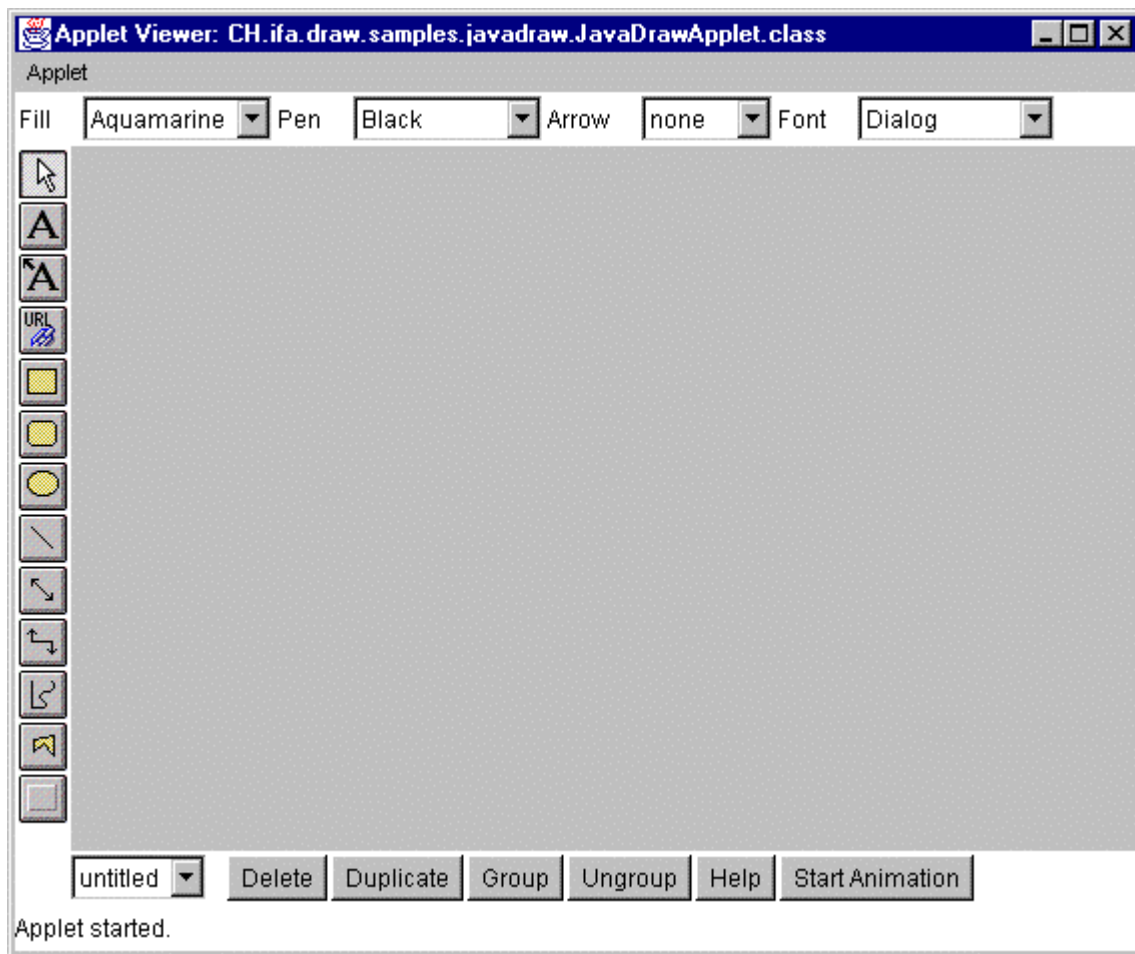
- HotDraw: Es un framework de dibujo desarrollado en Smaltalk. Fue desarrollado por Kent Beck y Ward Cunningham.
- ET++: Es un framework de una aplicación comprensible y portable, y una librería de clases para C++. Fue desarrollado por Andre Weinand y Erich Gamma.

JHotDraw ha sido desarrollado por Thomas Eggenschwiler y Erich Gamma y fue presentado en el OOPSLA97. Fue creado para un seminario como ejemplo de la aplicación de patrones de diseño en la creación de Frameworks, pero sus ideas son directamente aplicables a las aplicaciones profesionales.

Han colaborado también personas como Uwe Steinmueller, Doug Lea, Kent Beck, y Ward Cunningham.



Forma de la aplicación JavaDraw al ejecutarla.



Forma del applet JavaDraw al ejecutarlo.

3.5.2. Diagrama de clases general

El diagrama de clases que muestra las clases más importantes y su relaciones es el siguiente:

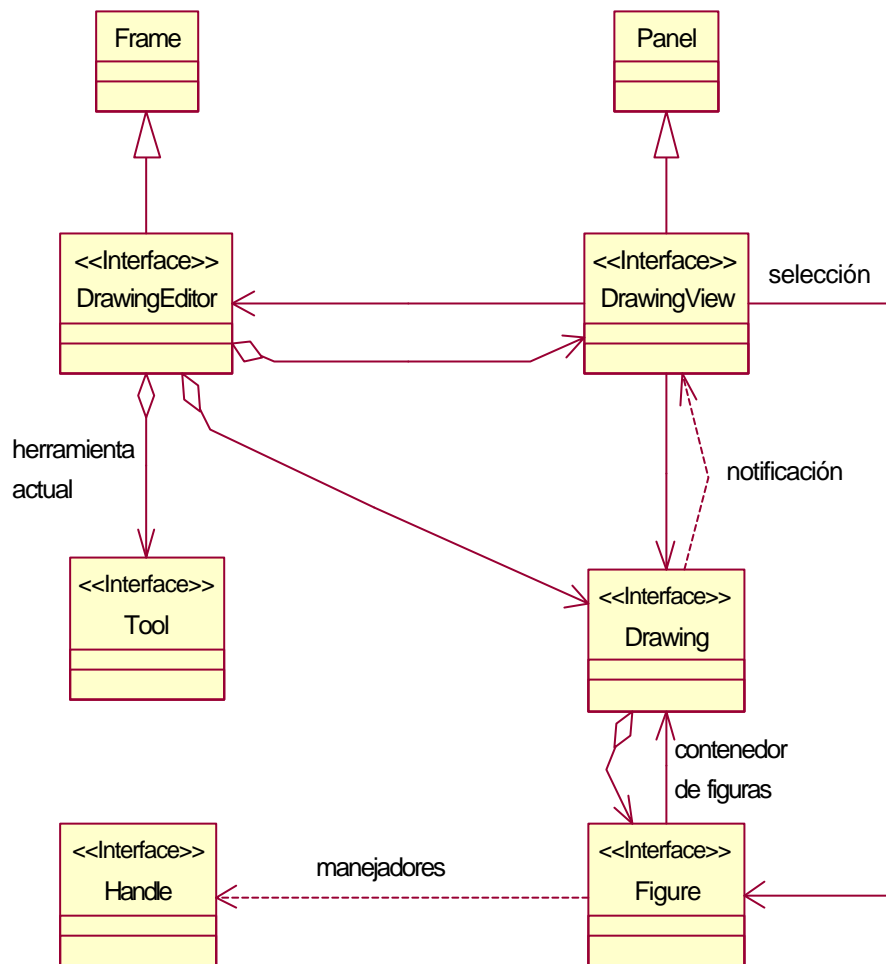


Diagrama de clases general.

El interface DrawingEditor define el interface para coordinar los diferentes objetos que participan en un editor de dibujo. Desacopla los participantes de un editor de dibujo.

DrawingView dibuja un Drawing y escucha sus cambios. Recibe entradas del usuario y las delega a la herramienta actual.

Drawing es un contenedor para figuras. Drawing envía los eventos DrawingChanged a DrawingChangeListeners cuando una parte de su area fue modificada.

Figure es el interface de una figura gráfica. Una figura conoce su caja de representación (display box) y puede dibujarse ella misma. Una figura puede estar compuesta de varias figuras. Para interactuar y manipular con una figura proporciona Handles y Connectors.

Una figura tiene un conjunto de manejadores que manipulan su forma o sus atributos. Una figura tiene uno o más conectores que definen como localizar y posicionar un punto de conexión.

Las figuras pueden tener un conjunto abierto de atributos. Un atributo es identificado por un string.

Las implementaciones por defecto para el interface Figure son proporcionadas por el AbstractFigure.

Los Handles son utilizados para cambiar una figura por manipulación directa. Los manejadores conocen sus propias figuras y proporcionan métodos para localizar y ubicar el manejo sobre la figura y seguir la pista a los cambios. Los Handles adaptan las operaciones que manipulan una figura a un interface común.

El interface Tool define un modo de presentación visual de dibujos. Todos los eventos de entrada sobre la representación visual del dibujo son enviados a su herramienta actual.

Las herramientas informan a su editor cuando se han realizado con una interacción llamando al método toolDone() del editor. Las herramientas son creadas una vez y reutilizadas. Son inicializadas y desinicializadas con activate() y deactivate() respectivamente.

3.5.3. Patrones de diseño utilizados y código en Java

El JHotDraw es un framework que utiliza patrones de diseño. Los patrones de diseño son problemas orientados no soluciones orientadas. Cada patrón describe como solucionar una pequeña parte de un gran problema de diseño.

Así pues los patrones de diseño proporcionan un diseño flexible (Strategy, State, Decorator,...), independiente y extensible. Permiten también la integración en una infraestructura existente (Adapter). Con los patrones se pretende separar de una clase unas características, una separación entre interfaz y diseño.

Para desarrollar frameworks es muy importante la separación de diseño y código. Esta separación hace más fácil la creación o reutilización de código.

Por lo tanto se separa el diseño de la implementación mediante interfaces.

Es importante tener unas normas para nombrar las clases, sobretodo si el programa es grande. Las normas de asignación de nombres a las clases e interfaces en el JHotDraw son las siguientes:

- Nombre simple para los interfaces. Una palabra que exprese su propósito.
- Nombres compuestos para las subclases.
- Las implementaciones por defecto:
 - Abstract<X>: Las clases abstractas como AbstractFigure proporcionan una implementación por defecto pero necesitan ser redefinidas.

- ❑ **Standard<X>:** Las clases estándar como `StandardDrawing` implementan un interface del framework y puede ser reusadas directamente.

La documentación de un framework tiene tres propósitos, y los patrones de diseño te ayudan a cumplir cada uno de ellos. La documentación debe describir:

- El propósito del framework.
- Cómo utilizar el framework.
- Los detalles del diseño del framework.

Los patrones de diseño son la forma más apropiada para enseñar como utilizar un framework, un conjunto de patrones de diseño pueden reunir los tres propósitos para documentar un framework.

3.5.3.1. Abstracciones fundamentales

FIGURAS

Las figuras pueden ser simples y complejas y pueden tener un número indeterminado de atributos.

Las responsabilidades de las figuras son: dibujarse, saber su posición, y tamaño y moverse. Una figura conoce su caja de representación (display box) y puede dibujarse ella misma. Una figura puede estar compuesta de varias figuras. Para interactuar y manipular con una figura proporciona Handles y Connectors.

Una figura tiene un conjunto de manejadores que manipulan su forma o sus atributos. Una figura tiene uno o más conectores que definen como localizar y posicionar un punto de conexión.

Las figuras pueden tener un conjunto abierto de atributos. Un atributo es identificado por un string.

La abstracción de las figuras debe ser independiente de cualquier detalle de implementación.

El interface de una figura gráfica es:

```
/*
 * @(#)Figure.java 5.1
 *
 */

package CH.ifa.draw.framework;

import CH.ifa.draw.util.*;
import java.awt.*;
import java.util.*;
```

```
import java.io.Serializable;

/**
 * The interface of a graphical figure. A figure knows
 * its display box and can draw itself. A figure can be
 * composed of several figures. To interact and manipulate
 * with a figure it can provide Handles and Connectors.<p>
 * A figure has a set of handles to manipulate its shape or
 * attributes.
 * A figure has one or more connectors that define how
 * to locate a connection point.<p>
 * Figures can have an open ended set of attributes.
 * An attribute is identified by a string.<p>
 * Default implementations for the Figure interface are provided
 * by AbstractFigure.
 *
 * @see Handle
 * @see Connector
 * @see AbstractFigure
 */

public interface Figure
    extends Storable, Cloneable, Serializable {

    /**
     * Moves the Figure to a new location.
     * @param x the x delta
     * @param y the y delta
     */
    public void moveBy(int dx, int dy);

    /**
     * Changes the display box of a figure. This method is
     * always implemented in figure subclasses.
     * It only changes
     * the displaybox and does not announce any changes. It
     * is usually not called by the client. Clients typically call
     * displayBox to change the display box.
     * @param origin the new origin
     * @param corner the new corner
     * @see #displayBox
     */
    public void basicDisplayBox(Point origin, Point corner);

    /**
     * Changes the display box of a figure. Clients usually
     * invoke this method. It changes the display box
     * and announces the corresponding changes.
     * @param origin the new origin
     * @param corner the new corner
     * @see #displayBox
     */
    public void displayBox(Point origin, Point corner);

    /**
     * Gets the display box of a figure
     * @see #basicDisplayBox
     */
}
```

```

public Rectangle displayBox();

/**
 * Draws the figure.
 * @param g the Graphics to draw into
 */
public void draw(Graphics g);

/**
 * Returns the handles used to manipulate
 * the figure. Handles is a Factory Method for
 * creating handle objects.
 *
 * @return a Vector of handles
 * @see Handle
 */
public Vector handles();

/**
 * Gets the size of the figure
 */
public Dimension size();

/**
 * Gets the figure's center
 */
public Point center();

/**
 * Checks if the Figure should be considered as empty.
 */
public boolean isEmpty();

/**
 * Returns an Enumeration of the figures contained in this figure
 */
public FigureEnumeration figures();

/**
 * Returns the figure that contains the given point.
 */
public Figure findFigureInside(int x, int y);

/**
 * Checks if a point is inside the figure.
 */
public boolean containsPoint(int x, int y);

/**
 * Returns a Clone of this figure
 */
public Object clone();

/**
 * Changes the display box of a figure. This is a
 * convenience method. Implementors should only
 * have to override basicDisplayBox
 * @see #displayBox

```

```
    */
    public void displayBox(Rectangle r);

    /**
     * Checks whether the given figure is contained in this figure.
     */
    public boolean includes(Figure figure);

    /**
     * Decomposes a figure into its parts. A figure is considered
     * as a part of itself.
     */
    public FigureEnumeration decompose();

    /**
     * Sets the Figure's container and registers the container
     * as a figure change listener. A figure's container can be
     * any kind of FigureChangeListener. A figure is not restricted
     * to have a single container.
     */
    public void addToContainer(FigureChangeListener c);

    /**
     * Removes a figure from the given container and unregisters
     * it as a change listener.
     */
    public void removeFromContainer(FigureChangeListener c);

    /**
     * Gets the Figure's listeners.
     */
    public FigureChangeListener listener();

    /**
     * Adds a listener for this figure.
     */
    public void addFigureChangeListener(FigureChangeListener l);

    /**
     * Removes a listener for this figure.
     */
    public void removeFigureChangeListener(FigureChangeListener l);

    /**
     * Releases a figure's resources. Release is called when
     * a figure is removed from a drawing. Informs the listeners that
     * the figure is removed by calling figureRemoved.
     */
    public void release();

    /**
     * Invalidates the figure. This method informs its listeners
     * that its current display box is invalid and should be
     * refreshed.
     */
    public void invalidate();

    /**
```



```

* Informes that a figure is about to change such that its
* display box is affected.
* Here is an example of how it is used together with changed()
* <pre>
* public void move(int x, int y) {
*     willChange();
*     // change the figure's location
*     changed();
* }
* </pre>
* @see #invalidate
* @see #changed
*/
public void willChange();

/**
 * Informes that a figure has changed its display box.
 * This method also triggers an update call for its
 * registered observers.
 * @see #invalidate
 * @see #willChange
 */
public void changed();

/**
 * Checks if this figure can be connected
 */
public boolean canConnect();

/**
 * Gets a connector for this figure at the given location.
 * A figure can have different connectors at different locations.
 */
public Connector connectorAt(int x, int y);

/**
 * Sets whether the connectors should be visible.
 * Connectors can be optionally visible. Implement
 * this method and react on isVisible to turn the
 * connectors on or off.
 */
public void connectorVisibility(boolean isVisible);

/**
 * Returns the connection inset. This is only a hint that
 * connectors can use to determine the connection location.
 * The inset defines the area where the display box of a
 * figure should not be connected.
 */
public Insets connectionInsets();

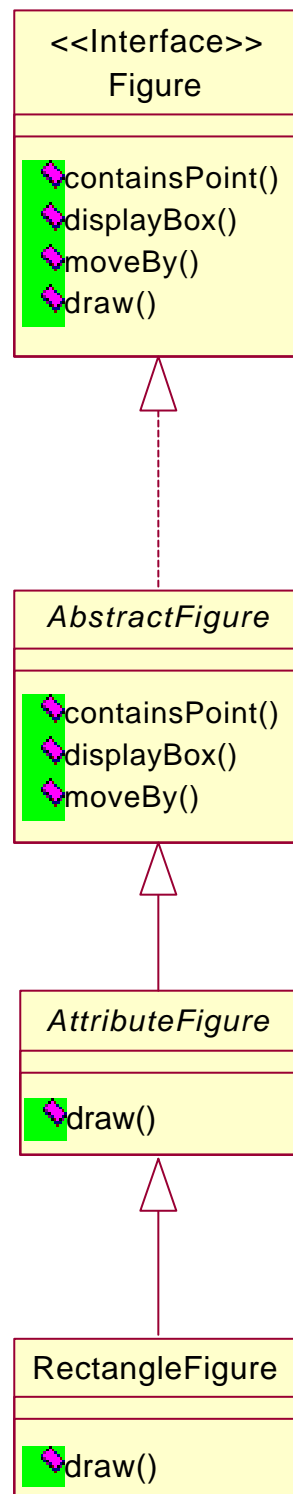
/**
 * Returns the locator used to located connected text.
 */
public Locator connectedTextLocator(Figure text);

```

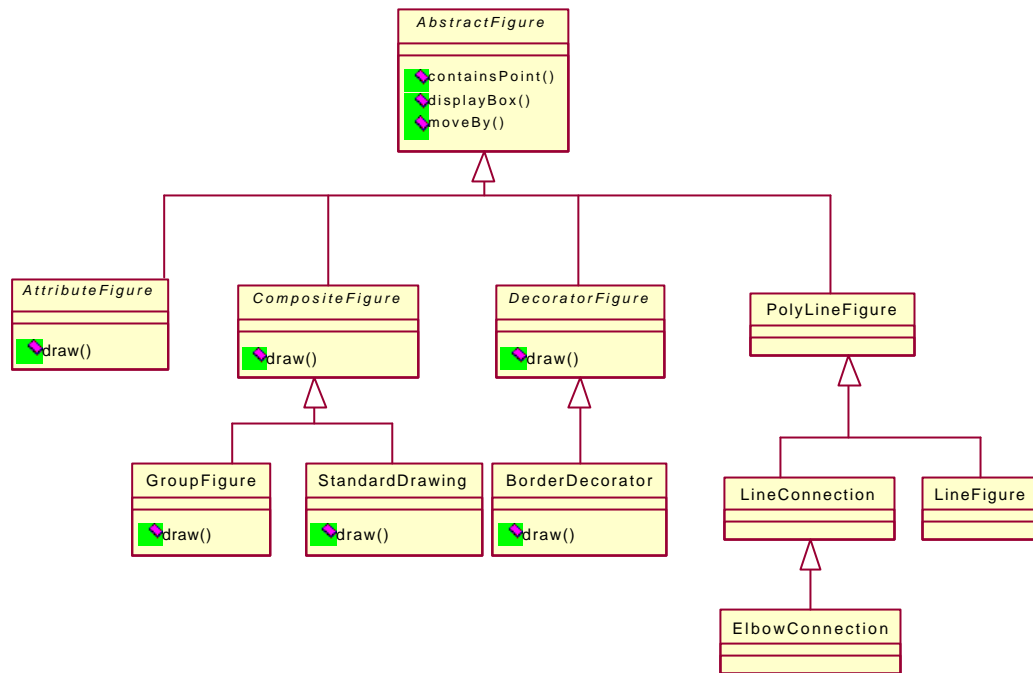
```
/**
 * Returns the named attribute or null if a
 * a figure doesn't have an attribute.
 * All figures support the attribute names
 * FillColor and FrameColor
 */
public Object getAttribute(String name);

/**
 * Sets the named attribute to the new value
 */
public void setAttribute(String name, Object value);
}
```

Las implementaciones por defecto para el interface Figure son proporcionadas por el AbstractFigure. Esta clase abstracta ofrece un comportamiento por defecto. La clase AbstractFigure utiliza el patrón de diseño **Template Method** para implementar un comportamiento por defecto e invariante para las subclases figuras. Este patrón lo explicaremos detalladamente más adelante.

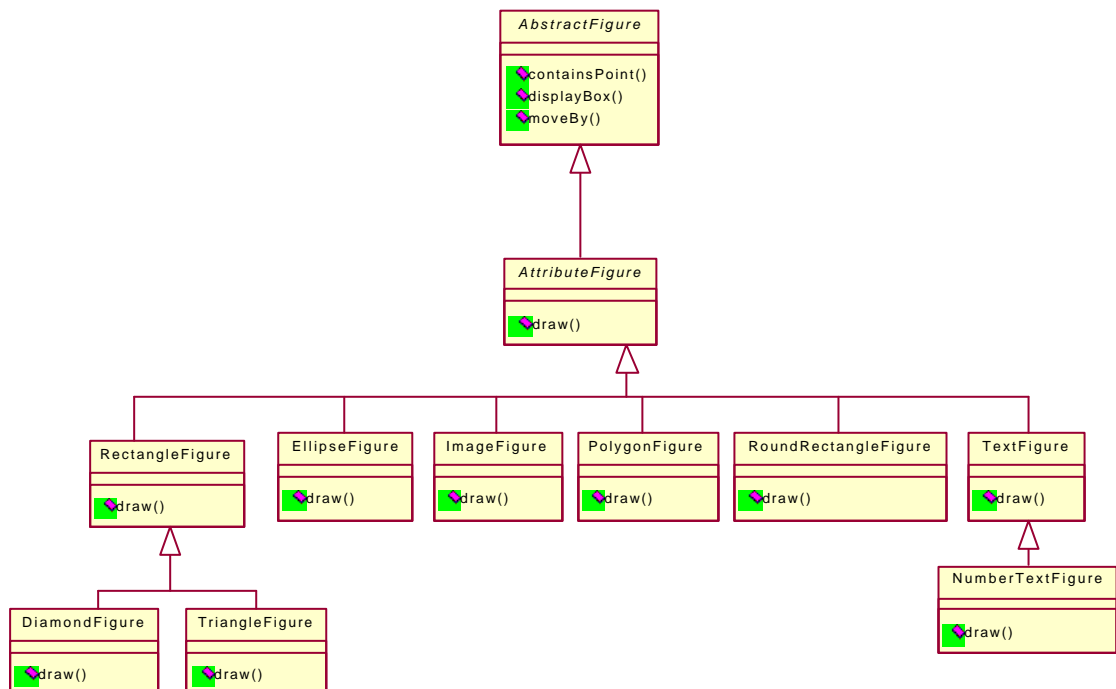


Jerarquía Figure.



Jerarquía AbstractFigure.

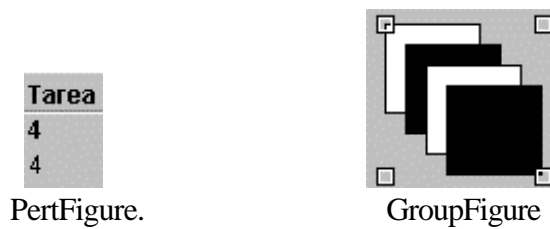
Una figura puede tener un conjunto no limitado de atributos. Los atributos son almacenados en un diccionario implementado por FigureAttributes.



Jerarquía de figuras.

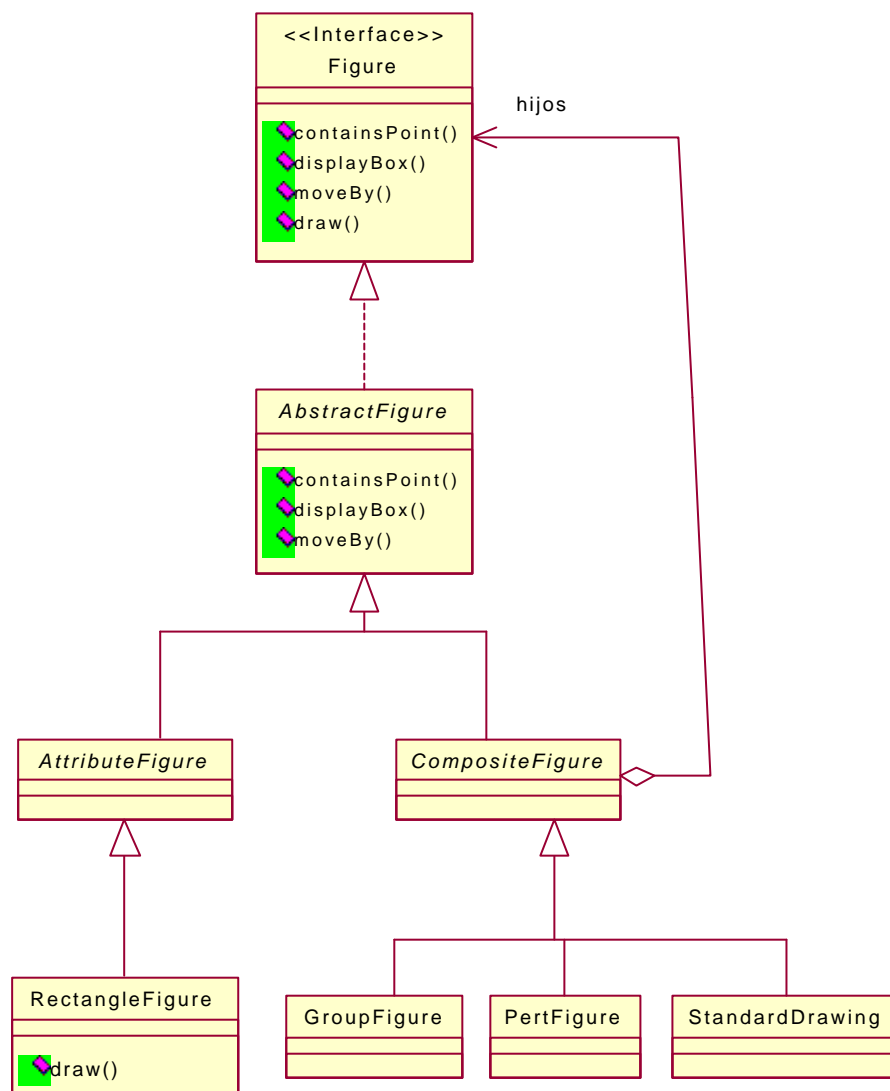
FIGURAS COMPUESTAS

Las figuras complicadas como una `PertFigure` están compuestas de simples figuras. Una tarea se muestra como una caja con una línea para el nombre de la tarea, una línea para la duración de la tarea, y una línea para el tiempo final más temprano de la tarea. Solamente el nombre de la tarea y su duración son editables, es decir se pueden cambiar. El tiempo final es calculado automáticamente por el PERT.



Las figuras complejas como `PertFigure` y `GroupFigure` son subclases de `CompositeFigure`. Una `CompositeFigure` es una figura con otras figuras como componentes. Una `CompositeFigure` no define ningún comportamiento layout. Este está dispuesto por subclases que ponen en orden los contenedores de figuras.

Es decir para las figuras compuestas se utiliza el **patrón Composite**. La `CompositeFigure` permite tratar una composición de figuras como una simple figura.



Utilización del patrón Composite.

Figure y AbstractFigure realizan el papel de *ComponenteAbstracto* del patrón Composite. CompositeFigure realiza el papel *CompositeAbstracto*. GroupFigure, PertFigure y StandardDrawing realizan el papel de *CompositeConcretos*. El papel de *ComponenteConcretos* lo realizan EllipseFigure, Imagefigure, PolygonFigure, RectangleFigure, DiamondFigure, TriangleFigure, RoundRectangleFigure, TextFigure, NumberTextFigure, BorderDecorator, PolyLineFigure, LineConnection, LineFigure, ElbowConnection.

```

public abstract class AbstractFigure implements Figure {
    .....

    /**
     * Returns an Enumeration of the figures contained in this figure.
     */
}

```

```

    * @see CompositeFigure
    */
    public FigureEnumeration figures() {
        Vector figures = new Vector(1);
        figures.addElement(this);
        return new FigureEnumerator(figures);
    }

    .....

    /**
     * Checks whether the given figure is contained in this figure.
     */
    public boolean includes(Figure figure) {
        return figure == this;
    }

    .....

    /**
     * Decomposes a figure into its parts. It returns a Vector
     * that contains itself.
     * @return an Enumeration for a Vector with itself as the
     * only element.
     */
    public FigureEnumeration decompose() {
        Vector figures = new Vector(1);
        figures.addElement(this);
        return new FigureEnumerator(figures);
    }

    .....
}

```

El código de la clase CompositeFigure sería el siguiente:

```

/*
 * @(#)CompositeFigure.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;
import java.awt.*;
import java.util.*;
import java.io.*;

/**
 * A Figure that is composed of several figures. A CompositeFigure
 * doesn't define any layout behavior. It is up to subclassers to
 * arrange the contained figures.
 * <hr>
 * <b>Design Patterns</b><P>
 * 

```

```

* <b><a href=../pattlets/sld012.htm>Composite</a></b><br>
* CompositeFigure enables to treat a composition of figures like
* a single figure.<br>
* @see Figure
*/

public abstract class CompositeFigure
    extends AbstractFigure
    implements FigureChangeListener {

    /**
     * The figures that this figure is composed of
     * @see #add
     * @see #remove
     */
    protected Vector fFigures;

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = 7408153435700021866L;
    private int compositeFigureSerializedDataVersion = 1;

    protected CompositeFigure() {
        fFigures = new Vector();
    }

    /**
     * Adds a figure to the list of figures. Initializes the
     * the figure's container.
     */
    public Figure add(Figure figure) {
        if (!fFigures.contains(figure)) {
            fFigures.addElement(figure);
            figure.addToContainer(this);
        }
        return figure;
    }

    /**
     * Adds a vector of figures.
     * @see #add
     */
    public void addAll(Vector newFigures) {
        Enumeration k = newFigures.elements();
        while (k.hasMoreElements())
            add((Figure) k.nextElement());
    }

    /**
     * Removes a figure from the composite.
     * @see #removeAll
     */
    public Figure remove(Figure figure) {
        if (fFigures.contains(figure)) {
            figure.removeFromContainer(this);
            fFigures.removeElement(figure);
        }
    }
}

```



```

        return figure;
    }

    /**
     * Removes a vector of figures.
     * @see #remove
     */
    public void removeAll(Vector figures) {
        Enumeration k = figures.elements();
        while (k.hasMoreElements())
            remove((Figure) k.nextElement());
    }

    /**
     * Removes all children.
     * @see #remove
     */
    public void removeAll() {
        FigureEnumeration k = figures();
        while (k.hasMoreElements()) {
            Figure figure = k.nextFigure();
            figure.removeFromContainer(this);
        }
        fFigures.removeAllElements();
    }

    /**
     * Removes a figure from the figure list, but
     * doesn't release it. Use this method to temporarily
     * manipulate a figure outside of the drawing.
     */
    public synchronized Figure orphan(Figure figure) {
        fFigures.removeElement(figure);
        return figure;
    }

    /**
     * Removes a vector of figures from the figure's list
     * without releasing the figures.
     * @see orphan
     */
    public void orphanAll(Vector newFigures) {
        Enumeration k = newFigures.elements();
        while (k.hasMoreElements())
            orphan((Figure) k.nextElement());
    }

    /**
     * Replaces a figure in the drawing without
     * removing it from the drawing.
     */
    public synchronized void replace(Figure figure, Figure
replacement) {
        int index = fFigures.indexOf(figure);
        if (index != -1) {
            replacement.addToContainer(this);    // will invalidate
                                                //figure
            figure.changed();
        }
    }

```

```

        fFigures.setElementAt(replacement, index);
    }
}

/**
 * Sends a figure to the back of the drawing.
 */
public synchronized void sendToBack(Figure figure) {
    if (fFigures.contains(figure)) {
        fFigures.removeElement(figure);
        fFigures.insertElementAt(figure, 0);
        figure.changed();
    }
}

/**
 * Brings a figure to the front.
 */
public synchronized void bringToFront(Figure figure) {
    if (fFigures.contains(figure)) {
        fFigures.removeElement(figure);
        fFigures.addElement(figure);
        figure.changed();
    }
}

/**
 * Draws all the contained figures
 * @see Figure#draw
 */
public void draw(Graphics g) {
    FigureEnumeration k = figures();
    while (k.hasMoreElements())
        k.nextFigure().draw(g);
}

/**
 * Gets a figure at the given index.
 */
public Figure figureAt(int i) {
    return (Figure)fFigures.elementAt(i);
}

/**
 * Returns an Enumeration for accessing the contained figures.
 * The figures are returned in the drawing order.
 */
public final FigureEnumeration figures() {
    return new FigureEnumerator(fFigures);
}

/**
 * Gets number of child figures.
 */
public int figureCount() {
    return fFigures.size();
}

```

```

/**
 * Returns an Enumeration for accessing the contained figures
 * in the reverse drawing order.
 */
public final FigureEnumeration figuresReverse() {
    return new ReverseFigureEnumerator(fFigures);
}

/**
 * Finds a top level Figure. Use this call for hit detection that
 * should not descend into the figure's children.
 */
public Figure findFigure(int x, int y) {
    FigureEnumeration k = figuresReverse();
    while (k.hasMoreElements()) {
        Figure figure = k.nextFigure();
        if (figure.containsPoint(x, y))
            return figure;
    }
    return null;
}

/**
 * Finds a top level Figure that intersects the given rectangle.
 */
public Figure findFigure(Rectangle r) {
    FigureEnumeration k = figuresReverse();
    while (k.hasMoreElements()) {
        Figure figure = k.nextFigure();
        Rectangle fr = figure.displayBox();
        if (r.intersects(fr))
            return figure;
    }
    return null;
}

/**
 * Finds a top level Figure, but supresses the passed
 * in figure. Use this method to ignore a figure
 * that is temporarily inserted into the drawing.
 * @param x the x coordinate
 * @param y the y coordinate
 * @param without the figure to be ignored during
 * the find.
 */
public Figure findFigureWithout(int x, int y, Figure without) {
    if (without == null)
        return findFigure(x, y);
    FigureEnumeration k = figuresReverse();
    while (k.hasMoreElements()) {
        Figure figure = k.nextFigure();
        if (figure.containsPoint(x, y) &&
!figure.includes(without))
            return figure;
    }
    return null;
}

```

```

/**
 * Finds a top level Figure that intersects the given rectangle.
 * It supresses the passed
 * in figure. Use this method to ignore a figure
 * that is temporarily inserted into the drawing.
 */
public Figure findFigure(Rectangle r, Figure without) {
    if (without == null)
        return findFigure(r);
    FigureEnumeration k = figuresReverse();
    while (k.hasMoreElements()) {
        Figure figure = k.nextFigure();
        Rectangle fr = figure.displayBox();
        if (r.intersects(fr) && !figure.includes(without))
            return figure;
    }
    return null;
}

/**
 * Finds a figure but descends into a figure's
 * children. Use this method to implement <i>click-through</i>
 * hit detection, that is, you want to detect the inner most
 * figure containing the given point.
 */
public Figure findFigureInside(int x, int y) {
    FigureEnumeration k = figuresReverse();
    while (k.hasMoreElements()) {
        Figure figure = k.nextFigure().findFigureInside(x, y);
        if (figure != null)
            return figure;
    }
    return null;
}

/**
 * Finds a figure but descends into a figure's
 * children. It supresses the passed
 * in figure. Use this method to ignore a figure
 * that is temporarily inserted into the drawing.
 */
public Figure findFigureInsideWithout(int x, int y, Figure
without) {
    FigureEnumeration k = figuresReverse();
    while (k.hasMoreElements()) {
        Figure figure = k.nextFigure();
        if (figure != without) {
            Figure found = figure.findFigureInside(x, y);
            if (found != null)
                return found;
        }
    }
    return null;
}

/**
 * Checks if the composite figure has the argument as one of
 * its children.

```

```

    */
    public boolean includes(Figure figure) {
        if (super.includes(figure))
            return true;

        FigureEnumeration k = figures();
        while (k.hasMoreElements()) {
            Figure f = k.nextFigure();
            if (f.includes(figure))
                return true;
        }
        return false;
    }

    /**
     * Moves all the given figures by x and y. Doesn't announce
     * any changes. Subclassers override
     * basicMoveBy. Clients usually call moveBy.
     * @see moveBy
     */
    protected void basicMoveBy(int x, int y) {
        FigureEnumeration k = figures();
        while (k.hasMoreElements())
            k.nextFigure().moveBy(x,y);
    }

    /**
     * Releases the figure and all its children.
     */
    public void release() {
        super.release();
        FigureEnumeration k = figures();
        while (k.hasMoreElements()) {
            Figure figure = k.nextFigure();
            figure.release();
        }
    }

    /**
     * Propagates the figureInvalidated event to my listener.
     * @see FigureChangeListener
     */
    public void figureInvalidated(FigureChangeEvent e) {
        if (listener() != null)
            listener().figureInvalidated(e);
    }

    /**
     * Propagates the removeFromDrawing request up to the container.
     * @see FigureChangeListener
     */
    public void figureRequestRemove(FigureChangeEvent e) {
        if (listener() != null)
            listener().figureRequestRemove(new
FigureChangeEvent(this));
    }

    /**

```

```

    * Propagates the requestUpdate request up to the container.
    * @see FigureChangeListener
    */
    public void figureRequestUpdate(FigureChangeEvent e) {
        if (listener() != null)
            listener().figureRequestUpdate(e);
    }

    public void figureChanged(FigureChangeEvent e) {
    }

    public void figureRemoved(FigureChangeEvent e) {
    }

    /**
     * Writes the contained figures to the StorableOutput.
     */
    public void write(StorableOutput dw) {
        super.write(dw);
        dw.writeInt(fFigures.size());
        Enumeration k = fFigures.elements();
        while (k.hasMoreElements())
            dw.writeStorable((Storable) k.nextElement());
    }

    /**
     * Reads the contained figures from StorableInput.
     */
    public void read(StorableInput dr) throws IOException {
        super.read(dr);
        int size = dr.readInt();
        fFigures = new Vector(size);
        for (int i=0; i<size; i++)
            add((Figure)dr.readStorable());
    }

    private void readObject(ObjectInputStream s)
        throws ClassNotFoundException, IOException {

        s.defaultReadObject();

        FigureEnumeration k = figures();
        while (k.hasMoreElements()) {
            Figure figure = k.nextFigure();
            figure.addToContainer(this);
        }
    }
}

```

En la clase anterior también se utiliza el patrón **Iterator** ya que se utilizan los interfaces `FigureEnumeration` y `Enumeration` que juegan el papel *IFIterator* de dicho patrón para recorrer el vector de figuras que forman el `CompositeFigure`.

```

/**
 * Adds a vector of figures.
 * @see #add

```

```

    */
    public void addAll(Vector newFigures) {
        Enumeration k = newFigures.elements();
        while (k.hasMoreElements())
            add((Figure) k.nextElement());
    }

```

```

    FigureEnumeration k = figures();
    while (k.hasMoreElements()) {
        Figure figure = k.nextFigure();
        figure.addToContainer(this);
    }

```

La clase PertFigure sería así:

```

/*
 * @(#)PertFigure.java 5.1
 *
 */

package CH.ifa.draw.samples.pert;

import java.awt.*;
import java.util.*;
import java.io.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.util.*;

public class PertFigure extends CompositeFigure {
    .....

    private void initialize() {
        fPostTasks = new Vector();
        fPreTasks = new Vector();
        fDisplayBox = new Rectangle(0, 0, 0, 0);

        Font f = new Font("Helvetica", Font.PLAIN, 12);
        Font fb = new Font("Helvetica", Font.BOLD, 12);

        TextFigure name = new TextFigure();
        name.setFont(fb);
        name.setText("Task");
        //name.setAttribute("TextColor", Color.white);
        add(name);

        NumberTextFigure duration = new NumberTextFigure();
        duration.setValue(0);
        duration.setFont(fb);
        add(duration);

        NumberTextFigure end = new NumberTextFigure();
        end.setValue(0);
    }

```

```

        end.setFont(f);
        end.setReadOnly(true);
        add(end);
    }

    .....
}

```

ATRIBUTOS

Una figura puede tener un conjunto no limitado de atributos. Se tendrían demasiados métodos de acceso. Muchos de los atributos son específicos de un tipo de figura, por ejemplo el tipo de letra, el ancho de la línea, etc. Algunos atributos son específicos de una figura concreta, por ejemplo el atributo URL.

La clase `AttributeFigure` da la implementación de una figura cuyo estado se almacena en un diccionario.

En Java todos los objetos heredan de la clase `Object` por lo tanto podemos poner como parámetro de un método un objeto de tipo `Object` y un método puede devolver un objeto de tipo `Object`. Por lo tanto se define un interface genérico de acceso a pares {clave, valor}:

```

void setAttribute (String name, Object value)

Object getAttribute (String name)

```

Los atributos son almacenados en un diccionario implementado por `FigureAttributes`.

El código de la clase `AttributeFigure` es:

```

/*
 * @(#)AttributeFigure.java 5.1
 *
 */

package CH.ifa.draw.figures;

import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;

import java.awt.*;
import java.util.*;
import java.io.*;

/**
 * A figure that can keep track of an open ended set of attributes.
 * The attributes are stored in a dictionary implemented by
 * FigureAttributes.
 *
 * @see Figure
 */

```



```

* @see Handle
* @see FigureAttributes
*/
public abstract class AttributeFigure extends AbstractFigure {

    /**
     * The attributes of a figure. Each figure can have
     * an open ended set of attributes. Attributes are
     * identified by name.
     * @see #getAttribute
     * @see #setAttribute
     */
    private FigureAttributes fAttributes;

    /**
     * The default attributes associated with a figure.
     * If a figure doesn't have an attribute set, a default
     * value from this shared attribute set is returned.
     * @see #getAttribute
     * @see #setAttribute
     */
    private static FigureAttributes fgDefaultAttributes = null;

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = -10857585979273442L;
    private int attributeFigureSerializedDataVersion = 1;

    protected AttributeFigure() { }

    /**
     * Draws the figure in the given graphics. Draw is a template
     * method calling drawBackground followed by drawFrame.
     */
    public void draw(Graphics g) {
        Color fill = getFillColor();
        if (!ColorMap.isTransparent(fill)) {
            g.setColor(fill);
            drawBackground(g);
        }
        Color frame = getFrameColor();
        if (!ColorMap.isTransparent(frame)) {
            g.setColor(frame);
            drawFrame(g);
        }
    }

    /**
     * Draws the background of the figure.
     * @see #draw
     */
    protected void drawBackground(Graphics g) {
    }

    /**
     * Draws the frame of the figure.
     * @see #draw
    
```

```

    */
    protected void drawFrame(Graphics g) {
    }

    /**
     * Gets the fill color of a figure. This is a convenience
     * method.
     * @see getAttribute
     */
    public Color getFillColor() {
        return (Color) getAttribute("FillColor");
    }

    /**
     * Gets the frame color of a figure. This is a convenience
     * method.
     * @see getAttribute
     */
    public Color getFrameColor() {
        return (Color) getAttribute("FrameColor");
    }

    //---- figure attributes -----

    private static void initializeAttributes() {
        fgDefaultAttributes = new FigureAttributes();
        fgDefaultAttributes.set("FrameColor", Color.black);
        fgDefaultAttributes.set("FillColor", new Color(0x70DB93));
        fgDefaultAttributes.set("TextColor", Color.black);
        fgDefaultAttributes.set("ArrowMode", new Integer(0));
        fgDefaultAttributes.set("FontName", "Helvetica");
        fgDefaultAttributes.set("FontSize", new Integer(12));
        fgDefaultAttributes.set("FontStyle", new
Integer(Font.PLAIN));
    }

    /**
     * Gets a the default value for a named attribute
     * @see getAttribute
     */
    public static Object getDefaultAttribute(String name) {
        if (fgDefaultAttributes == null)
            initializeAttributes();
        return fgDefaultAttributes.get(name);
    }

    /**
     * Returns the named attribute or null if a
     * a figure doesn't have an attribute.
     * All figures support the attribute names
     * FillColor and FrameColor
     */
    public Object getAttribute(String name) {
        if (fAttributes != null) {
            if (fAttributes.hasDefined(name))
                return fAttributes.get(name);
        }
        return getDefaultAttribute(name);
    }

```

```

    }

    /**
     * Sets the named attribute to the new value
     */
    public void setAttribute(String name, Object value) {
        if (fAttributes == null)
            fAttributes = new FigureAttributes();
        fAttributes.set(name, value);
        changed();
    }

    /**
     * Stores the Figure to a StorableOutput.
     */
    public void write(StorableOutput dw) {
        super.write(dw);
        if (fAttributes == null)
            dw.writeString("no_attributes");
        else {
            dw.writeString("attributes");
            fAttributes.write(dw);
        }
    }

    /**
     * Reads the Figure from a StorableInput.
     */
    public void read(StorableInput dr) throws IOException {
        super.read(dr);
        String s = dr.readString();
        if (s.toLowerCase().equals("attributes")) {
            fAttributes = new FigureAttributes();
            fAttributes.read(dr);
        }
    }
}

```

La clase abstracta AttributeFigure utiliza el patrón de diseño **Template Method**. El método draw de dicha clase es un template method llama a drawBackground y a drawFrame. AttributeFigure juega el papel de *ClaseAbstracta* del patrón Template Method. Las clases que heredan de AttributeFigure juegan el papel de *ClaseConcreta*.

```

    /**
     * Draws the figure in the given graphics. Draw is a template
     * method calling drawBackground followed by drawFrame.
     */
    public void draw(Graphics g) {
        Color fill = getFillColor();
        if (!ColorMap.isTransparent(fill)) {
            g.setColor(fill);
            drawBackground(g);
        }
        Color frame = getFrameColor();
        if (!ColorMap.isTransparent(frame)) {

```

```

        g.setColor(frame);
        drawFrame(g);
    }
}

/**
 * Draws the background of the figure.
 * @see #draw
 */
protected void drawBackground(Graphics g) {
}

/**
 * Draws the frame of the figure.
 * @see #draw
 */
protected void drawFrame(Graphics g) {
}

```

FigureAttributes es un contenedor para los atributos de una figura. Los atributos son guardados como parejas {clave, valor}.

```

/**
 * @(#)FigureAttributes.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.util.*;
import java.awt.Color;
import java.io.IOException;
import java.io.Serializable;

import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

/**
 * A container for a figure's attributes. The attributes are stored
 * as key/value pairs.
 *
 * @see Figure
 */

public class FigureAttributes
    extends Object
    implements Cloneable, Serializable {

    private Hashtable fMap;

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = -
6886355144423666716L;
    private int figureAttributesSerializedDataVersion = 1;

```

```

/**
 * Constructs the FigureAttributes.
 */
public FigureAttributes() {
    fMap = new Hashtable();
}

/**
 * Gets the attribute with the given name.
 * @returns attribute or null if the key is not defined
 */
public Object get(String name) {
    return fMap.get(name);
}

/**
 * Sets the attribute with the given name and
 * overwrites its previous value.
 */
public void set(String name, Object value) {
    fMap.put(name, value);
}

/**
 * Tests if an attribute is defined.
 */
public boolean hasDefined(String name) {
    return fMap.containsKey(name);
}

/**
 * Clones the attributes.
 */
public Object clone() {
    try {
        FigureAttributes a = (FigureAttributes) super.clone();
        a.fMap = (Hashtable) fMap.clone();
        return a;
    } catch (CloneNotSupportedException e) {
        throw new InternalError();
    }
}

/**
 * Reads the attributes from a StorableInput.
 * FigureAttributes store the following types directly:
 * Color, Boolean, String, Int. Other attribute types
 * have to implement the Storable interface or they
 * have to be wrapped by an object that implements Storable.
 * @see Storable
 * @see #write
 */
public void read(StorableInput dr) throws IOException {
    String s = dr.readString();
    if (!s.toLowerCase().equals("attributes"))
        throw new IOException("Attributes expected");
}

```

```

        fMap = new Hashtable();
        int size = dr.readInt();
        for (int i=0; i<size; i++) {
            String key = dr.readString();
            String valtype = dr.readString();
            Object val = null;
            if (valtype.equals("Color"))
                val = new Color(dr.readInt(), dr.readInt(),
dr.readInt());
            else if (valtype.equals("Boolean"))
                val = new Boolean(dr.readString());
            else if (valtype.equals("String"))
                val = dr.readString();
            else if (valtype.equals("Int"))
                val = new Integer(dr.readInt());
            else if (valtype.equals("Storable"))
                val = dr.readStorable();
            else if (valtype.equals("UNKNOWN"))
                continue;

            fMap.put(key, val);
        }
    }

    /**
     * Writes the attributes to a StorableInput.
     * FigureAttributes store the following types directly:
     * Color, Boolean, String, Int. Other attribute types
     * have to implement the Storable interface or they
     * have to be wrapped by an object that implements Storable.
     * @see Storable
     * @see #write
     */
    public void write(StorableOutput dw) {
        dw.writeString("attributes");

        dw.writeInt(fMap.size());    // number of attributes
        Enumeration k = fMap.keys();
        while (k.hasMoreElements()) {
            String s = (String) k.nextElement();
            dw.writeString(s);
            Object v = fMap.get(s);
            if (v instanceof String) {
                dw.writeString("String");
                dw.writeString((String) v);
            } else if (v instanceof Color) {
                dw.writeString("Color");
                dw.writeInt(((Color)v).getRed());
                dw.writeInt(((Color)v).getGreen());
                dw.writeInt(((Color)v).getBlue());
            } else if (v instanceof Boolean) {
                dw.writeString("Boolean");
                if (((Boolean)v).booleanValue())
                    dw.writeString("TRUE");
                else
                    dw.writeString("FALSE");
            } else if (v instanceof Integer) {
                dw.writeString("Int");
            }
        }
    }
}

```

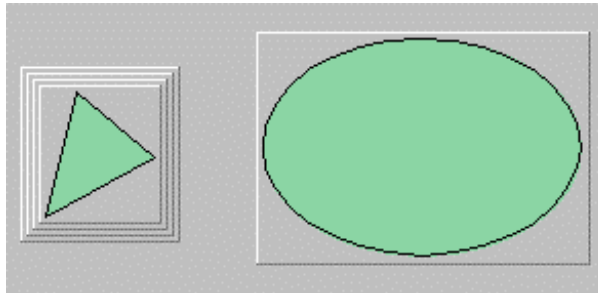
```

        dw.writeInt(((Integer)v).intValue());
    } else if (v instanceof Storable) {
        dw.writeString("Storable");
        dw.writeStorable((Storable)v);
    } else {
        System.out.println(v);
        dw.writeString("UNKNOWN");
    }
}
}
}
}

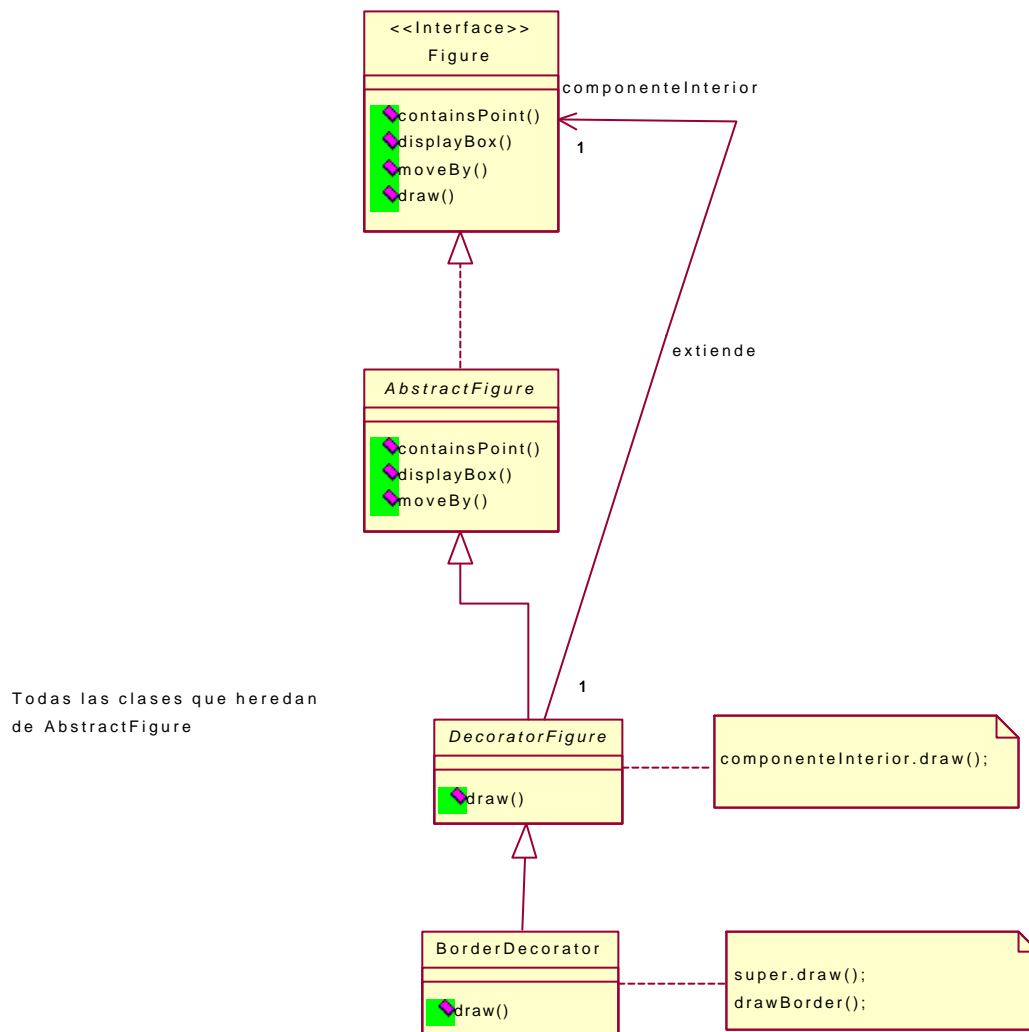
```

PATRÓN DECORATOR

DecoratorFigure puede ser utilizado para decorar otras figuras con decoraciones como bordes, sombras,.... Decorator envia todos sus métodos a las figuras que contiene. Las subclases pueden selectivamente sobrescribir estos métodos, extender y filtrar sus comportamientos. Es decir se añaden funcionalidades sin modificar las clases.



Para ello se utiliza el patrón **Decorator**. Así pues DecoratorFigure realiza el papel de *DecoratorAbstracto* de dicho patrón. La clase BorderDecorator realiza el papel de *DecoratorConcreto*. Figure y AbstractFigure realizan el papel *ComponenteAbstracto*. Y todas las figuras, las clases que heredan de AbstractFigure, realizan el papel de *ComponenteConcreto*.



Utilización del Patrón Decorator.

El código de DecoratorFigure es el siguiente:

```

/*
 * @(#)DecoratorFigure.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

import java.awt.*;
import java.util.*;
import java.io.*;

/**
 * DecoratorFigure can be used to decorate other figures with
 * decorations like borders. Decorator forwards all the

```



```

* methods to their contained figure. Subclasses can selectively
* override these methods to extend and filter their behavior.
* <hr>
* <b>Design Patterns</b><P>
* 
* <b><a href=../pattlets/sld014.htm>Decorator</a></b><br>
* DecoratorFigure is a decorator.
*
* @see Figure
*/

public abstract class DecoratorFigure
    extends AbstractFigure
    implements FigureChangeListener {

    /**
     * The decorated figure.
     */
    protected Figure fComponent;

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = 8993011151564573288L;
    private int decoratorFigureSerializedDataVersion = 1;

    public DecoratorFigure() { }

    /**
     * Constructs a DecoratorFigure and decorates the passed in
     * figure.
     */
    public DecoratorFigure(Figure figure) {
        decorate(figure);
    }

    /**
     * Forwards the connection insets to its contained figure..
     */
    public Insets connectionInsets() {
        return fComponent.connectionInsets();
    }

    /**
     * Forwards the canConnect to its contained figure..
     */
    public boolean canConnect() {
        return fComponent.canConnect();
    }

    /**
     * Forwards containsPoint to its contained figure.
     */
    public boolean containsPoint(int x, int y) {
        return fComponent.containsPoint(x, y);
    }

    /**

```

```

    * Decorates the given figure.
    */
    public void decorate(Figure figure) {
        fComponent = figure;
        fComponent.addToContainer(this);
    }

    /**
     * Removes the decoration from the contained figure.
     */
    public Figure peelDecoration() {
        fComponent.removeFromContainer(this); //??? set the container
                                           // to the listener()?
        return fComponent;
    }

    /**
     * Forwards displayBox to its contained figure.
     */
    public Rectangle displayBox() {
        return fComponent.displayBox();
    }

    /**
     * Forwards basicDisplayBox to its contained figure.
     */
    public void basicDisplayBox(Point origin, Point corner) {
        fComponent.basicDisplayBox(origin, corner);
    }

    /**
     * Forwards draw to its contained figure.
     */
    public void draw(Graphics g) {
        fComponent.draw(g);
    }

    /**
     * Forwards findFigureInside to its contained figure.
     */
    public Figure findFigureInside(int x, int y) {
        return fComponent.findFigureInside(x, y);
    }

    /**
     * Forwards handles to its contained figure.
     */
    public Vector handles() {
        return fComponent.handles();
    }

    /**
     * Forwards includes to its contained figure.
     */
    public boolean includes(Figure figure) {
        return (super.includes(figure) ||
        fComponent.includes(figure));
    }

```

```

/**
 * Forwards moveBy to its contained figure.
 */
public void moveBy(int x, int y) {
    fComponent.moveBy(x, y);
}

/**
 * Forwards basicMoveBy to its contained figure.
 */
protected void basicMoveBy(int x, int y) {
    // this will never be called
}

/**
 * Releases itself and the contained figure.
 */
public void release() {
    super.release();
    fComponent.removeFromContainer(this);
    fComponent.release();
}

/**
 * Propagates invalidate up the container chain.
 * @see FigureChangeListener
 */
public void figureInvalidated(FigureChangeEvent e) {
    if (listener() != null)
        listener().figureInvalidated(e);
}

public void figureChanged(FigureChangeEvent e) {
}

public void figureRemoved(FigureChangeEvent e) {
}

/**
 * Propagates figureRequestUpdate up the container chain.
 * @see FigureChangeListener
 */
public void figureRequestUpdate(FigureChangeEvent e) {
    if (listener() != null)
        listener().figureRequestUpdate(e);
}

/**
 * Propagates the removeFromDrawing request up to the container.
 * @see FigureChangeListener
 */
public void figureRequestRemove(FigureChangeEvent e) {
    if (listener() != null)
        listener().figureRequestRemove(new
FigureChangeEvent(this));
}

```

```

/**
 * Forwards figures to its contained figure.
 */
public FigureEnumeration figures() {
    return fComponent.figures();
}

/**
 * Forwards decompose to its contained figure.
 */
public FigureEnumeration decompose() {
    return fComponent.decompose();
}

/**
 * Forwards setAttribute to its contained figure.
 */
public void setAttribute(String name, Object value) {
    fComponent.setAttribute(name, value);
}

/**
 * Forwards getAttribute to its contained figure.
 */
public Object getAttribute(String name) {
    return fComponent.getAttribute(name);
}

/**
 * Returns the locator used to located connected text.
 */
public Locator connectedTextLocator(Figure text) {
    return fComponent.connectedTextLocator(text);
}

/**
 * Returns the Connector for the given location.
 */
public Connector connectorAt(int x, int y) {
    return fComponent.connectorAt(x, y);
}

/**
 * Forwards the connector visibility request to its component.
 */
public void connectorVisibility(boolean isVisible) {
    fComponent.connectorVisibility(isVisible);
}

/**
 * Writes itself and the contained figure to the StorableOutput.
 */
public void write(StorableOutput dw) {
    super.write(dw);
    dw.writeStorable(fComponent);
}

/**

```

```

    * Reads itself and the contained figure from the StorableInput.
    */
    public void read(StorableInput dr) throws IOException {
        super.read(dr);
        decorate((Figure)dr.readStorable());
    }

    private void readObject(ObjectInputStream s)
        throws ClassNotFoundException, IOException {

        s.defaultReadObject();

        fComponent.addToContainer(this);
    }
}

```

BorderDecorator decora una Figure arbitraria con un borde.

```

/*
 * @(#)BorderDecorator.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.util.*;

import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;

/**
 * BorderDecorator decorates an arbitrary Figure with
 * a border.
 */
public class BorderDecorator extends DecoratorFigure {

    /*
     * Serialization support.
     */
    private static final long serialVersionUID = 1205601808259084917L;
    private int borderDecoratorSerializedDataVersion = 1;

    public BorderDecorator() { }
    public BorderDecorator(Figure figure) {
        super(figure);
    }

    private Point border() {
        return new Point(3,3);
    }

    /**
     * Draws a the figure and decorates it with a border.
     */
    public void draw(Graphics g) {
        Rectangle r = displayBox();
    }
}

```

```

        super.draw(g);
        g.setColor(Color.white);
        g.drawLine(r.x, r.y, r.x, r.y + r.height);
        g.drawLine(r.x, r.y, r.x + r.width, r.y);
        g.setColor(Color.gray);
        g.drawLine(r.x + r.width, r.y, r.x + r.width, r.y + r.height);
        g.drawLine(r.x, r.y + r.height, r.x + r.width, r.y +
r.height);
    }

    /**
     * Gets the displaybox including the border.
     */
    public Rectangle displayBox() {
        Rectangle r = fComponent.displayBox();
        r.grow(border().x, border().y);
        return r;
    }

    /**
     * Invalidates the figure extended by its border.
     */
    public void figureInvalidated(FigureChangeEvent e) {
        Rectangle rect = e.getInvalidatedRectangle();
        rect.grow(border().x, border().y);
        super.figureInvalidated(new FigureChangeEvent(e.getFigure(),
rect));
    }

    public Insets connectionInsets() {
        Insets i = super.connectionInsets();
        i.top -= 3;
        i.bottom -= 3;
        i.left -= 3;
        i.right -= 3;
        return i;
    }
}

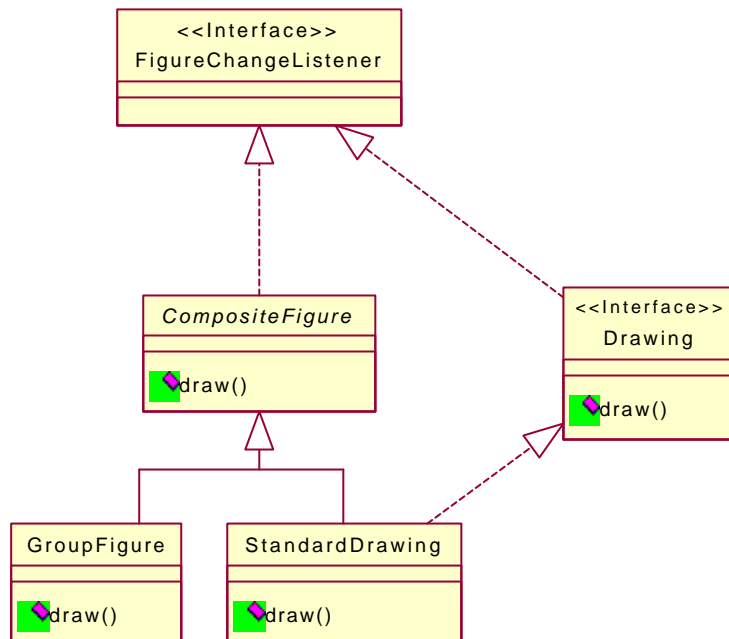
```

DIBUJOS

Drawing es un contenedor para figuras, es decir es un conjunto de Figures.

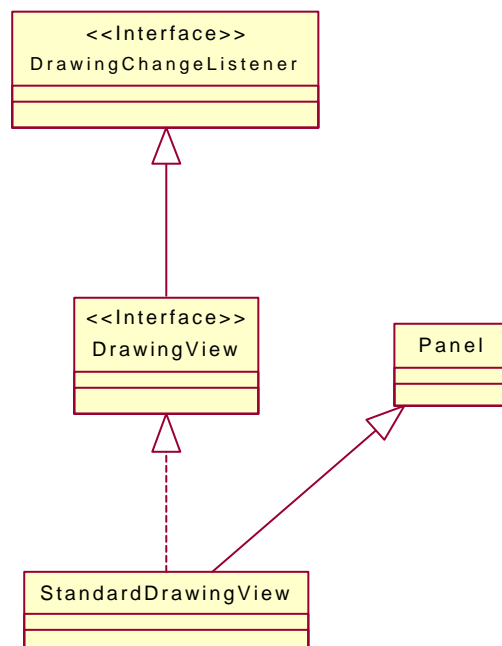
Drawing envía los eventos DrawingChanged a DrawingChangeListeners cuando una parte de su area fue modificada.

Drawing utiliza el patrón Observer para desacoplar el Drawing de sus vistas y permitir varias vistas.



Jerarquía Drawing-CompositeFigure.

`DrawingView` dibuja un dibujo (`Drawing`) y escucha sus cambios. Recibe entradas del usuario y las delega a la herramienta actual. La selección actual se encuentra en `DrawingView`.



Jerarquía DrawingView.

`DrawingView` utiliza los siguientes patrones de diseño:

Observer `DrawingView` observa dibujos que cambian a través del interface `DrawingChangeListener`.

State `DrawingView` juega el papel del *Contexto* en el patrón State. La herramienta es el *Estado*.

Strategy `DrawingView` es el *Cliente* en el patrón Strategy como corresponde al `UpdateStrategy`. `DrawingView` es el *Cliente* para el `PointConstrainer`.

El código del interface `DrawingView` es el siguiente:

```

/*
 * @(#)DrawingView.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.image.ImageObserver;
import java.awt.*;
import java.util.*;
import java.io.*;
import CH.ifa.draw.util.*;

/**
 * DrawingView renders a Drawing and listens to its changes.
 * It receives user input and delegates it to the current tool.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld026.htm>Observer</a></b><br>
 * DrawingView observes drawing for changes via the DrawingListener
 interface.<br>
 * 
 * <b><a href=../pattlets/sld032.htm>State</a></b><br>
 * DrawingView plays the role of the StateContext in
 the State pattern. Tool is the State.<br>
 * 
 * <b><a href=../pattlets/sld034.htm>Strategy</a></b><br>
 * DrawingView is the StrategyContext in the Strategy pattern
 with regard to the UpdateStrategy. <br>
 * DrawingView is the StrategyContext for the PointConstrainer.
 *
 * @see Drawing
 * @see Painter
 * @see Tool
 */

public interface DrawingView extends ImageObserver,
DrawingChangeListener {

    /**
     * Sets the view's editor.
     */
    public void setEditor(DrawingEditor editor);

```



```

/**
 * Gets the current tool.
 */
public Tool tool();

/**
 * Gets the drawing.
 */
public Drawing drawing();

/**
 * Sets and installs another drawing in the view.
 */
public void setDrawing(Drawing d);

/**
 * Gets the editor.
 */
public DrawingEditor editor();

/**
 * Adds a figure to the drawing.
 * @return the added figure.
 */
public Figure add(Figure figure);

/**
 * Removes a figure from the drawing.
 * @return the removed figure
 */
public Figure remove(Figure figure);

/**
 * Adds a vector of figures to the drawing.
 */
public void addAll(Vector figures);

/**
 * Gets the size of the drawing.
 */
public Dimension getSize();

/**
 * Gets the minimum dimension of the drawing.
 */
public Dimension getMinimumSize();

/**
 * Gets the preferred dimension of the drawing..
 */
public Dimension getPreferredSize();

/**
 * Sets the current display update strategy.
 * @see UpdateStrategy
 */
public void setDisplayUpdate(Painter updateStrategy);

```

```

/**
 * Gets the currently selected figures.
 * @return a vector with the selected figures. The vector
 * is a copy of the current selection.
 */
public Vector selection();

/**
 * Gets an enumeration over the currently selected figures.
 */
public FigureEnumeration selectionElements();

/**
 * Gets the currently selected figures in Z order.
 * @see #selection
 * @return a vector with the selected figures. The vector
 * is a copy of the current selection.
 */
public Vector selectionZOrdered();

/**
 * Gets the number of selected figures.
 */
public int selectionCount();

/**
 * Adds a figure to the current selection.
 */
public void addToSelection(Figure figure);

/**
 * Adds a vector of figures to the current selection.
 */
public void addToSelectionAll(Vector figures);

/**
 * Removes a figure from the selection.
 */
public void removeFromSelection(Figure figure);

/**
 * If a figure isn't selected it is added to the selection.
 * Otherwise it is removed from the selection.
 */
public void toggleSelection(Figure figure);

/**
 * Clears the current selection.
 */
public void clearSelection();

/**
 * Gets the current selection as a FigureSelection. A
 * FigureSelection can be cut, copied, pasted.
 */
public FigureSelection getFigureSelection();

/**

```

```

    * Finds a handle at the given coordinates.
    * @return the hit handle, null if no handle is found.
    */
    public Handle findHandle(int x, int y);

    /**
     * Gets the position of the last click inside the view.
     */
    public Point lastClick();

    /**
     * Sets the current point constrainer.
     */
    public void setConstrainer(PointConstrainer p);

    /**
     * Gets the current grid setting.
     */
    public PointConstrainer getConstrainer();

    /**
     * Checks whether the drawing has some accumulated damage
     */
    public void checkDamage();

    /**
     * Repair the damaged area
     */
    public void repairDamage();

    /**
     * Paints the drawing view. The actual drawing is delegated to
     * the current update strategy.
     * @see Painter
     */
    public void paint(Graphics g);

    /**
     * Creates an image with the given dimensions
     */
    public Image createImage(int width, int height);

    /**
     * Gets a graphic to draw into
     */
    public Graphics getGraphics();

    /**
     * Gets the background color of the DrawingView
     */
    public Color getBackground();

    /**
     * Gets the background color of the DrawingView
     */
    public void setBackground(Color c);

    /**

```

```

    * Draws the contents of the drawing view.
    * The view has three layers: background, drawing, handles.
    * The layers are drawn in back to front order.
    */
    public void drawAll(Graphics g);

    /**
     * Draws the currently active handles.
     */
    public void drawHandles(Graphics g);

    /**
     * Draws the drawing.
     */
    public void drawDrawing(Graphics g);

    /**
     * Draws the background. If a background pattern is set it
     * is used to fill the background. Otherwise the background
     * is filled in the background color.
     */
    public void drawBackground(Graphics g);

    /**
     * Sets the cursor of the DrawingView
     */
    public void setCursor(Cursor c);

    /**
     * Freezes the view by acquiring the drawing lock.
     * @see Drawing#lock
     */
    public void freezeView();

    /**
     * Unfreezes the view by releasing the drawing lock.
     * @see Drawing#unlock
     */
    public void unfreezeView();
}

```

La clase StandardDrawingView es un Panel sobre el que se dibujan los dibujos (Drawings). Además obtiene los eventos de teclado y del ratón y se los pasa al editor (DrawingEditor) para que los maneje.

La clase StandardDrawingView es la implementación estándar de un interface DrawingView.

```

/*
 * @(#)StandardDrawingView.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.awt.event.*;

```

```
import java.util.*;
import java.io.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

/**
 * The standard implementation of DrawingView.
 * @see DrawingView
 * @see Painter
 * @see Tool
 */

public class StandardDrawingView
    extends Panel
    implements DrawingView,
               MouseListener,
               MouseMotionListener,
               KeyListener {

    /**
     * The DrawingEditor of the view.
     * @see #tool
     * @see #setStatus
     */
    transient private DrawingEditor fEditor;

    /**
     * The shown drawing.
     */
    private Drawing fDrawing;

    /**
     * the accumulated damaged area
     */
    private transient Rectangle fDamage = null;

    /**
     * The list of currently selected figures.
     */
    transient private Vector fSelection;

    /**
     * The shown selection handles.
     */
    transient private Vector fSelectionHandles;

    /**
     * The preferred size of the view
     */
    private Dimension fViewSize;

    /**
     * The position of the last mouse click
     * inside the view.
     */
    private Point fLastClick;

    /**
```

```

    * A vector of optional backgrounds. The vector maintains
    * a list a view painters that are drawn before the contents,
    * that is in the background.
    */
    private Vector fBackgrounds = null;

    /**
     * A vector of optional foregrounds. The vector maintains
     * a list a view painters that are drawn after the contents,
     * that is in the foreground.
     */
    private Vector fForegrounds = null;

    /**
     * The update strategy used to repair the view.
     */
    private Painter fUpdateStrategy;

    /**
     * The grid used to constrain points for snap to
     * grid functionality.
     */
    private PointConstrainer fConstrainer;

    /*
     * Serialization support. In JavaDraw only the Drawing is
     * serialized. However, for beans support StandardDrawingView
     * supports serialization
     */
    private static final long serialVersionUID = -
3878153366174603336L;
    private int drawingViewSerializedDataVersion = 1;

    /**
     * Constructs the view.
     */
    public StandardDrawingView(DrawingEditor editor, int width, int
height) {
        fEditor = editor;
        fViewSize = new Dimension(width,height);
        fLastClick = new Point(0, 0);
        fConstrainer = null;
        fSelection = new Vector();
        setDisplayUpdate(new BufferedUpdateStrategy());
        setBackground(Color.lightGray);

        addMouseListener(this);
        addMouseMotionListener(this);
        addKeyListener(this);
    }

    /**
     * Sets the view's editor.
     */
    public void setEditor(DrawingEditor editor) {
        fEditor = editor;
    }

```

```
/**
 * Gets the current tool.
 */
public Tool tool() {
    return fEditor.tool();
}

/**
 * Gets the drawing.
 */
public Drawing drawing() {
    return fDrawing;
}

/**
 * Sets and installs another drawing in the view.
 */
public void setDrawing(Drawing d) {
    clearSelection();

    if (fDrawing != null)
        fDrawing.removeDrawingChangeListener(this);

    fDrawing = d;
    if (fDrawing != null)
        fDrawing.addDrawingChangeListener(this);
    checkMinimumSize();
    repaint();
}

/**
 * Gets the editor.
 */
public DrawingEditor editor() {
    return fEditor;
}

/**
 * Adds a figure to the drawing.
 * @return the added figure.
 */
public Figure add(Figure figure) {
    return drawing().add(figure);
}

/**
 * Removes a figure from the drawing.
 * @return the removed figure
 */
public Figure remove(Figure figure) {
    return drawing().remove(figure);
}

/**
 * Adds a vector of figures to the drawing.
 */
public void addAll(Vector figures) {
    FigureEnumeration k = new FigureEnumeration(figures);
```

```

        while (k.hasMoreElements())
            add(k.nextFigure());
    }

    /**
     * Gets the minimum dimension of the drawing.
     */
    public Dimension getMinimumSize() {
        return fViewSize;
    }

    /**
     * Gets the preferred dimension of the drawing..
     */
    public Dimension getPreferredSize() {
        return getMinimumSize();
    }

    /**
     * Sets the current display update strategy.
     * @see UpdateStrategy
     */
    public void setDisplayUpdate(Painter updateStrategy) {
        fUpdateStrategy = updateStrategy;
    }

    /**
     * Gets the currently selected figures.
     * @return a vector with the selected figures. The vector
     * is a copy of the current selection.
     */
    public Vector selection() {
        // protect the vector with the current selection
        return (Vector)fSelection.clone();
    }

    /**
     * Gets an enumeration over the currently selected figures.
     */
    public FigureEnumeration selectionElements() {
        return new FigureEnumerator(fSelection);
    }

    /**
     * Gets the currently selected figures in Z order.
     * @see #selection
     * @return a vector with the selected figures. The vector
     * is a copy of the current selection.
     */
    public Vector selectionZOrdered() {
        Vector result = new Vector(fSelection.size());
        FigureEnumeration figures = drawing().figures();

        while (figures.hasMoreElements()) {
            Figure f= figures.nextFigure();
            if (fSelection.contains(f)) {
                result.addElement(f);
            }
        }
    }

```



```

    }
    return result;
}

/**
 * Gets the number of selected figures.
 */
public int selectionCount() {
    return fSelection.size();
}

/**
 * Adds a figure to the current selection.
 */
public void addToSelection(Figure figure) {
    if (!fSelection.contains(figure)) {
        fSelection.addElement(figure);
        fSelectionHandles = null;
        figure.invalidate();
        selectionChanged();
    }
}

/**
 * Adds a vector of figures to the current selection.
 */
public void addToSelectionAll(Vector figures) {
    FigureEnumeration k = new FigureEnumerator(figures);
    while (k.hasMoreElements())
        addToSelection(k.nextFigure());
}

/**
 * Removes a figure from the selection.
 */
public void removeFromSelection(Figure figure) {
    if (fSelection.contains(figure)) {
        fSelection.removeElement(figure);
        fSelectionHandles = null;
        figure.invalidate();
        selectionChanged();
    }
}

/**
 * If a figure isn't selected it is added to the selection.
 * Otherwise it is removed from the selection.
 */
public void toggleSelection(Figure figure) {
    if (fSelection.contains(figure))
        removeFromSelection(figure);
    else
        addToSelection(figure);
    selectionChanged();
}

/**
 * Clears the current selection.

```

```

    */
    public void clearSelection() {
        Figure figure;

        FigureEnumeration k = selectionElements();

        while (k.hasMoreElements())
            k.nextFigure().invalidate();
        fSelection = new Vector();
        fSelectionHandles = null;
        selectionChanged();
    }

    /**
     * Gets an enumeration of the currently active handles.
     */
    private Enumeration selectionHandles() {
        if (fSelectionHandles == null) {
            fSelectionHandles = new Vector();
            FigureEnumeration k = selectionElements();
            while (k.hasMoreElements()) {
                Figure figure = k.nextFigure();
                Enumeration kk = figure.handles().elements();
                while (kk.hasMoreElements())
                    fSelectionHandles.addElement(kk.nextElement());
            }
        }
        return fSelectionHandles.elements();
    }

    /**
     * Gets the current selection as a FigureSelection. A
     * FigureSelection can be cut, copied, pasted.
     */
    public FigureSelection getFigureSelection() {
        return new FigureSelection(selectionZOrdered());
    }

    /**
     * Finds a handle at the given coordinates.
     * @return the hit handle, null if no handle is found.
     */
    public Handle findHandle(int x, int y) {
        Handle handle;

        Enumeration k = selectionHandles();
        while (k.hasMoreElements()) {
            handle = (Handle) k.nextElement();
            if (handle.containsPoint(x, y))
                return handle;
        }
        return null;
    }

    /**
     * Informs that the current selection changed.
     * By default this event is forwarded to the
     * drawing editor.
     */

```

```

    */
    protected void selectionChanged() {
        fEditor.selectionChanged(this);
    }

    /**
     * Gets the position of the last click inside the view.
     */
    public Point lastClick() {
        return fLastClick;
    }

    /**
     * Sets the grid spacing that is used to constrain points.
     */
    public void setConstrainer(PointConstrainer c) {
        fConstrainer = c;
    }

    /**
     * Gets the current constrainer.
     */
    public PointConstrainer getConstrainer() {
        return fConstrainer;
    }

    /**
     * Constrains a point to the current grid.
     */
    protected Point constrainPoint(Point p) {
        // constrain to view size
        Dimension size = getSize();
        //p.x = Math.min(size.width, Math.max(1, p.x));
        //p.y = Math.min(size.height, Math.max(1, p.y));
        p.x = Geom.range(1, size.width, p.x);
        p.y = Geom.range(1, size.height, p.y);

        if (fConstrainer != null )
            return fConstrainer.constrainPoint(p);
        return p;
    }

    /**
     * Handles mouse down events. The event is delegated to the
     * currently active tool.
     * @return whether the event was handled.
     */
    public void mousePressed(MouseEvent e) {
        requestFocus(); // JDK1.1
        Point p = constrainPoint(new Point(e.getX(), e.getY()));
        fLastClick = new Point(e.getX(), e.getY());
        tool().mouseDown(e, p.x, p.y);
        checkDamage();
    }

    /**
     * Handles mouse drag events. The event is delegated to the
     * currently active tool.

```

```

    * @return whether the event was handled.
    */
    public void mouseDragged(MouseEvent e) {
        Point p = constrainPoint(new Point(e.getX(), e.getY()));
        tool().mouseDrag(e, p.x, p.y);
        checkDamage();
    }

    /**
     * Handles mouse move events. The event is delegated to the
     * currently active tool.
     * @return whether the event was handled.
     */
    public void mouseMoved(MouseEvent e) {
        tool().mouseMove(e, e.getX(), e.getY());
    }

    /**
     * Handles mouse up events. The event is delegated to the
     * currently active tool.
     * @return whether the event was handled.
     */
    public void mouseReleased(MouseEvent e) {
        Point p = constrainPoint(new Point(e.getX(), e.getY()));
        tool().mouseUp(e, p.x, p.y);
        checkDamage();
    }

    /**
     * Handles key down events. Cursor keys are handled
     * by the view the other key events are delegated to the
     * currently active tool.
     * @return whether the event was handled.
     */
    public void keyPressed(KeyEvent e) {
        int code = e.getKeyCode();
        if ((code == KeyEvent.VK_BACK_SPACE) || (code ==
KeyEvent.VK_DELETE)) {
            Command cmd = new DeleteCommand("Delete", this);
            cmd.execute();
        } else if (code == KeyEvent.VK_DOWN || code == KeyEvent.VK_UP
||
            code == KeyEvent.VK_RIGHT || code == KeyEvent.VK_LEFT) {
            handleCursorKey(code);
        } else {
            tool().keyDown(e, code);
        }
        checkDamage();
    }

    /**
     * Handles cursor keys by moving all the selected figures
     * one grid point in the cursor direction.
     */
    protected void handleCursorKey(int key) {
        int dx = 0, dy = 0;
        int stepX = 1, stepY = 1;
        // should consider Null Object.
    }

```

```

        if (fConstrainer != null) {
            stepX = fConstrainer.getStepX();
            stepY = fConstrainer.getStepY();
        }

        switch (key) {
        case KeyEvent.VK_DOWN:
            dy = stepY;
            break;
        case KeyEvent.VK_UP:
            dy = -stepY;
            break;
        case KeyEvent.VK_RIGHT:
            dx = stepX;
            break;
        case KeyEvent.VK_LEFT:
            dx = -stepX;
            break;
        }
        moveSelection(dx, dy);
    }

    private void moveSelection(int dx, int dy) {
        FigureEnumeration figures = selectionElements();
        while (figures.hasMoreElements())
            figures.nextFigure().moveBy(dx, dy);
        checkDamage();
    }

    /**
     * Refreshes the drawing if there is some accumulated damage
     */
    public synchronized void checkDamage() {
        Enumeration each = drawing().drawingChangeListeners();
        while (each.hasMoreElements()) {
            Object l = each.nextElement();
            if (l instanceof DrawingView) {
                ((DrawingView)l).repairDamage();
            }
        }
    }

    public void repairDamage() {
        if (fDamage != null) {
            repaint(fDamage.x, fDamage.y, fDamage.width,
fDamage.height);
            fDamage = null;
        }
    }

    public void drawingInvalidated(DrawingChangeEvent e) {
        Rectangle r = e.getInvalidatedRectangle();
        if (fDamage == null)
            fDamage = r;
        else
            fDamage.add(r);
    }
}

```

```

public void drawingRequestUpdate(DrawingChangeEvent e) {
    repairDamage();
}

/**
 * Updates the drawing view.
 */
public void update(Graphics g) {
    paint(g);
}

/**
 * Paints the drawing view. The actual drawing is delegated to
 * the current update strategy.
 * @see Painter
 */
public void paint(Graphics g) {
    fUpdateStrategy.draw(g, this);
}

/**
 * Draws the contents of the drawing view.
 * The view has three layers: background, drawing, handles.
 * The layers are drawn in back to front order.
 */
public void drawAll(Graphics g) {
    boolean isPrinting = g instanceof PrintGraphics;
    drawBackground(g);
    if (fBackgrounds != null && !isPrinting)
        drawPainters(g, fBackgrounds);
    drawDrawing(g);
    if (fForegrounds != null && !isPrinting)
        drawPainters(g, fForegrounds);
    if (!isPrinting)
        drawHandles(g);
}

/**
 * Draws the currently active handles.
 */
public void drawHandles(Graphics g) {
    Enumeration k = selectionHandles();
    while (k.hasMoreElements())
        ((Handle) k.nextElement()).draw(g);
}

/**
 * Draws the drawing.
 */
public void drawDrawing(Graphics g) {
    fDrawing.draw(g);
}

/**
 * Draws the background. If a background pattern is set it
 * is used to fill the background. Otherwise the background
 * is filled in the background color.

```

```

    */
    public void drawBackground(Graphics g) {
        g.setColor(getBackground());
        g.fillRect(0, 0, getBounds().width, getBounds().height);
    }

    private void drawPainters(Graphics g, Vector v) {
        for (int i = 0; i < v.size(); i++)
            ((Painter)v.elementAt(i)).draw(g, this);
    }

    /**
     * Adds a background.
     */
    public void addBackground(Painter painter) {
        if (fBackgrounds == null)
            fBackgrounds = new Vector(3);
        fBackgrounds.addElement(painter);
        repaint();
    }

    /**
     * Removes a background.
     */
    public void removeBackground(Painter painter) {
        if (fBackgrounds != null)
            fBackgrounds.removeElement(painter);
        repaint();
    }

    /**
     * Removes a foreground.
     */
    public void removeForeground(Painter painter) {
        if (fForegrounds != null)
            fForegrounds.removeElement(painter);
        repaint();
    }

    /**
     * Adds a foreground.
     */
    public void addForeground(Painter painter) {
        if (fForegrounds == null)
            fForegrounds = new Vector(3);
        fForegrounds.addElement(painter);
        repaint();
    }

    /**
     * Freezes the view by acquiring the drawing lock.
     * @see Drawing#lock
     */
    public void freezeView() {
        drawing().lock();
    }

    /**

```

```

    * Unfreezes the view by releasing the drawing lock.
    * @see Drawing#unlock
    */
    public void unfreezeView() {
        drawing().unlock();
    }

    private void readObject(ObjectInputStream s)
        throws ClassNotFoundException, IOException {

        s.defaultReadObject();

        fSelection = new Vector(); // could use lazy initialization
                                   //instead

        if (fDrawing != null)
            fDrawing.addDrawingChangeListener(this);
    }

    private void checkMinimumSize() {
        FigureEnumeration k = drawing().figures();
        Dimension d = new Dimension(0, 0);
        while (k.hasMoreElements()) {
            Rectangle r = k.nextFigure().displayBox();
            d.width = Math.max(d.width, r.x+r.width);
            d.height = Math.max(d.height, r.y+r.height);
        }
        if (fViewSize.height < d.height || fViewSize.width < d.width)
        {
            fViewSize.height = d.height+10;
            fViewSize.width = d.width+10;
            setSize(fViewSize);
        }
    }

    public boolean isFocusTraversable() {
        return true;
    }

    // listener methods we are not interested in
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mouseClicked(MouseEvent e) {}
    public void keyTyped(KeyEvent e) {}
    public void keyReleased(KeyEvent e) {}
}

```

3.5.3.2. Herramientas

DrawingView recibe los eventos del interface de usuario. La respuesta a estos eventos depende de la herramienta activa en ese momento.

Hay que tener en cuenta que diferentes tipos de editores pueden requerir distintos tipos de herramientas por lo tanto no debe limitarse su funcionalidad y debe ser sencillo definir nuevas herramientas.

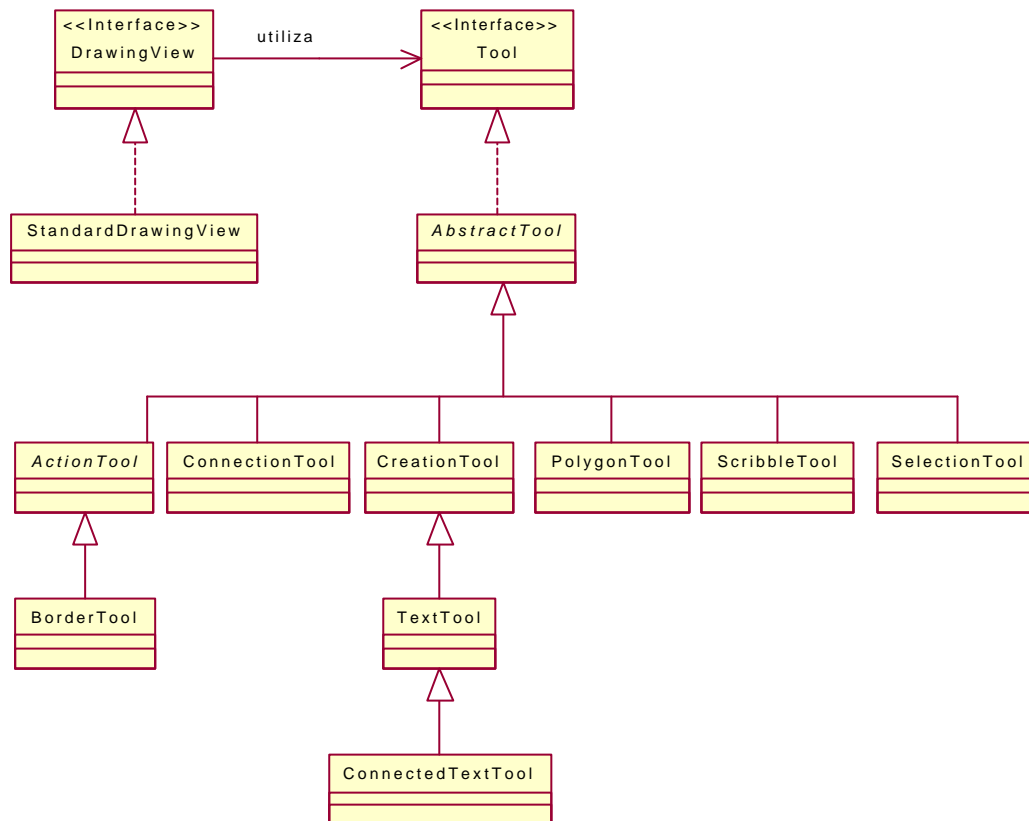
Una idea errónea que se nos podría ocurrir sería esta:

```
class DrawingView
{
    void mouseDown(...)
    {
        switch (tool)
        {
            case SELECTION: ...
                break;
            case CIRCLE: ....
                break;
        }
    }
    void mouseDrag(...)
    {
        switch (tool)
        {
            case SELECTION: ...
                break;
            case CIRCLE: ....
                break;
        }
    }
}
```

Esta idea es errónea porque es tedioso añadir nuevas herramientas y además se impide la reutilización. Esta idea sería el antipatrón de desarrollo de software Spaghetti code, que es tener muchos if o switch.

La reacción del editor ante los eventos depende del estado en el que se encuentre el editor. Por lo tanto para solucionar este problema se utiliza el patrón **State**. El patrón State permite a un objeto cambiar su comportamiento cuando cambias sus atributos internos. Por encapsulación todos los estados especifican su comportamiento.

El interface DrawingView y su implementación estándar StandardDrawingView juegan el papel *Contexto* del patrón State. El interface Tool y la clase abstracta AbstractTool juegan el papel *Estado*. Las herramientas ActionTool, ConnectionTool, CreationTool, PolygonTool, ScribbleTool, SelectionTool juegan el papel de *EstadoConcreto*.



Utilización del patrón State.

Una herramienta define un modo de presentación visual de dibujos. Todos los eventos de entrada sobre la representación visual del dibujo son enviados a su herramienta actual.

Las herramientas informan a su editor cuando se han realizado con una interacción llamando al método `toolDone()` del editor. Las herramientas son creadas una vez y reutilizadas. Son inicializadas y desinicializadas con `activate()` y `deactivate()` respectivamente.

```

/*
 * @(#)Tool.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.*;
import java.awt.event.MouseEvent;
import java.awt.event.KeyEvent;

/**
 * A tool defines a mode of the drawing view. All input events
 * targeted to the drawing view are forwarded to its current tool.<p>
 * Tools inform their editor when they are done with an interaction
 * by calling the editor's toolDone() method.
 * The Tools are created once and reused. They
 * are initialized/deinitialized with activate()/deactivate().
 */

```

```

* <hr>
* <b>Design Patterns</b><P>
* 
* <b><a href=../pattlets/sld032.htm>State</a></b><br>
* Tool plays the role of the State. In encapsulates all state
* specific behavior. DrawingView plays the role of the StateContext.
* @see DrawingView
*/

public interface Tool {

    /**
     * Activates the tool for the given view. This method is called
     * whenever the user switches to this tool. Use this method to
     * reinitialize a tool.
     */
    public void activate();

    /**
     * Deactivates the tool. This method is called whenever the user
     * switches to another tool. Use this method to do some clean-up
     * when the tool is switched. Subclassers should always call
     * super.deactivate.
     */
    public void deactivate();

    /**
     * Handles mouse down events in the drawing view.
     */
    public void mouseDown(MouseEvent e, int x, int y);

    /**
     * Handles mouse drag events in the drawing view.
     */
    public void mouseDrag(MouseEvent e, int x, int y);

    /**
     * Handles mouse up in the drawing view.
     */
    public void mouseUp(MouseEvent e, int x, int y);

    /**
     * Handles mouse moves (if the mouse button is up).
     */
    public void mouseMove(MouseEvent evt, int x, int y);

    /**
     * Handles key down events in the drawing view.
     */
    public void keyDown(KeyEvent evt, int key);
}

```

```

package CH.ifa.draw.standard;

import java.awt.*;

```

```

import java.awt.event.*;
import java.util.*;
import java.io.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

/**
 * The standard implementation of DrawingView.
 * @see DrawingView
 * @see Painter
 * @see Tool
 */

public class StandardDrawingView
    extends Panel
    implements DrawingView,
               MouseListener,
               MouseMotionListener,
               KeyListener {

    .....

    /**
     * The list of currently selected figures.
     */
    transient private Vector fSelection;

    /**
     * The shown selection handles.
     */
    transient private Vector fSelectionHandles;

    .....

    /**
     * Constructs the view.
     */
    public StandardDrawingView(DrawingEditor editor, int width, int
height) {
        fEditor = editor;
        fViewSize = new Dimension(width,height);
        fLastClick = new Point(0, 0);
        fConstrainer = null;
        fSelection = new Vector();
        setDisplayUpdate(new BufferedUpdateStrategy());
        setBackground(Color.lightGray);

        addMouseListener(this);
        addMouseMotionListener(this);
        addKeyListener(this);
    }

    .....

    /**
     * Gets the current tool.
     */
    public Tool tool() {

```

```

        return fEditor.tool();
    }

    .....

    /**
     * Handles mouse down events. The event is delegated to the
     * currently active tool.
     * @return whether the event was handled.
     */
    public void mousePressed(MouseEvent e) {
        requestFocus(); // JDK1.1
        Point p = constrainPoint(new Point(e.getX(), e.getY()));
        fLastClick = new Point(e.getX(), e.getY());
        tool().mouseDown(e, p.x, p.y);
        checkDamage();
    }

    /**
     * Handles mouse drag events. The event is delegated to the
     * currently active tool.
     * @return whether the event was handled.
     */
    public void mouseDragged(MouseEvent e) {
        Point p = constrainPoint(new Point(e.getX(), e.getY()));
        tool().mouseDrag(e, p.x, p.y);
        checkDamage();
    }

    /**
     * Handles mouse move events. The event is delegated to the
     * currently active tool.
     * @return whether the event was handled.
     */
    public void mouseMoved(MouseEvent e) {
        tool().mouseMove(e, e.getX(), e.getY());
    }

    /**
     * Handles mouse up events. The event is delegated to the
     * currently active tool.
     * @return whether the event was handled.
     */
    public void mouseReleased(MouseEvent e) {
        Point p = constrainPoint(new Point(e.getX(), e.getY()));
        tool().mouseUp(e, p.x, p.y);
        checkDamage();
    }

    /**
     * Handles key down events. Cursor keys are handled
     * by the view the other key events are delegated to the
     * currently active tool.
     * @return whether the event was handled.
     */
    public void keyPressed(KeyEvent e) {
        int code = e.getKeyCode();

```

```

        if ((code == KeyEvent.VK_BACK_SPACE) || (code ==
KeyEvent.VK_DELETE)) {
            Command cmd = new DeleteCommand("Delete", this);
            cmd.execute();
        } else if (code == KeyEvent.VK_DOWN || code == KeyEvent.VK_UP
||
            code == KeyEvent.VK_RIGHT || code == KeyEvent.VK_LEFT) {
            handleCursorKey(code);
        } else {
            tool().keyDown(e, code);
        }
        checkDamage();
    }

    .....
}

```

HERRAMIENTAS DE CREACIÓN

Muchas de las herramientas de creación se diferencian únicamente en la figura a crear.

Para crear estas herramientas se puede pensar en utilizar el patrón Factory Method o el patrón Prototype.

La ventaja del patrón Prototype es que no hace falta una clase Herramienta por cada figura. Una sola clase puede valer para todas.

La ventaja del patrón Factory Method es que la figura no necesita saber clonarse.

JHotDraw utiliza para esta misión el patrón **Prototype**.

CreationTool es una herramienta que crea nuevas figuras. La figura que se crea es especificada por un Prototype. CreationTool crea nuevas figuras clonando un prototipo. CreationTool juega el papel del *PrototypeConcreto* del patrón Prototype.

```

/*
 * @(#)CreationTool.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.awt.event.MouseEvent;

import CH.ifa.draw.framework.*;

/**
 * A tool to create new figures. The figure to be
 * created is specified by a prototype.
 *
 * <hr>

```

```

* <b>Design Patterns</b><P>
* 
* <b><a href=../pattlets/sld029.htm>Prototype</a></b><br>
* CreationTool creates new figures by cloning a prototype.
* <hr>
* @see Figure
* @see Object#clone
*/

public class CreationTool extends AbstractTool {

    /**
     * the anchor point of the interaction
     */
    private Point    fAnchorPoint;

    /**
     * the currently created figure
     */
    private Figure   fCreatedFigure;

    /**
     * the prototypical figure that is used to create new figures.
     */
    private Figure   fPrototype;

    /**
     * Initializes a CreationTool with the given prototype.
     */
    public CreationTool(DrawingView view, Figure prototype) {
        super(view);
        fPrototype = prototype;
    }

    /**
     * Constructs a CreationTool without a prototype.
     * This is for subclasses overriding createFigure.
     */
    protected CreationTool(DrawingView view) {
        super(view);
        fPrototype = null;
    }

    /**
     * Sets the cross hair cursor.
     */
    public void activate() {
        view().setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
    }

    /**
     * Creates a new figure by cloning the prototype.
     */
    public void mouseDown(MouseEvent e, int x, int y) {
        fAnchorPoint = new Point(x,y);
    }

```

```

        fCreatedFigure = createFigure();
        fCreatedFigure.displayBox(fAnchorPoint, fAnchorPoint);
        view().add(fCreatedFigure);
    }

    /**
     * Creates a new figure by cloning the prototype.
     */
    protected Figure createFigure() {
        if (fPrototype == null)
            throw new HJDError("No prototype defined");
        return (Figure) fPrototype.clone();
    }

    /**
     * Adjusts the extent of the created figure
     */
    public void mouseDrag(MouseEvent e, int x, int y) {
        fCreatedFigure.displayBox(fAnchorPoint, new Point(x,y));
    }

    /**
     * Checks if the created figure is empty. If it is, the figure
     * is removed from the drawing.
     * @see Figure#isEmpty
     */
    public void mouseUp(MouseEvent e, int x, int y) {
        if (fCreatedFigure.isEmpty())
            drawing().remove(fCreatedFigure);
        fCreatedFigure = null;
        editor().toolDone();
    }

    /**
     * Gets the currently created figure
     */
    protected Figure createdFigure() {
        return fCreatedFigure;
    }
}

```

NUEVAS HERRAMIENTAS

Una vez hecha una nueva figura se desea obtener su herramienta. Tenemos las siguientes opciones:

1. Dar una semántica adecuada al método clone que sirva como prototipo (generalmente vale con la implementación por defecto). De esta manera no hace falta implementar ninguna herramienta nueva. Esta opción esta basada en la clonacion.

```

    /**
     * the prototypical figure that is used to create new figures.
     */
    private Figure fPrototype;

```

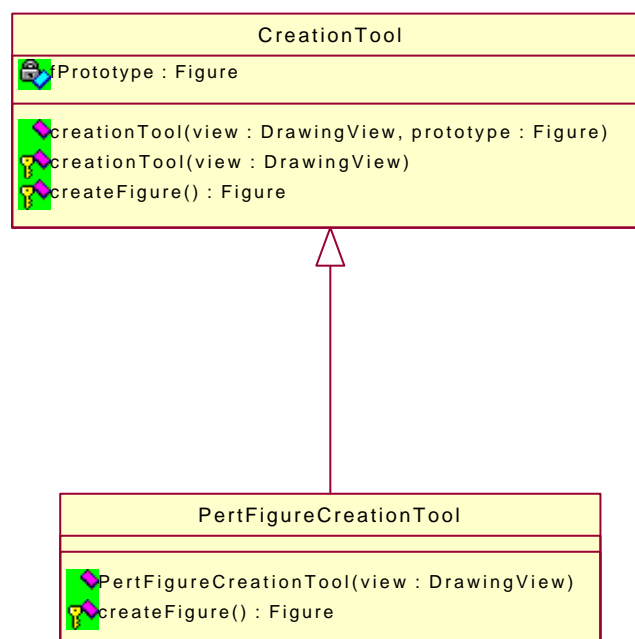


```
/**
 * Initializes a CreationTool with the given prototype.
 */
public CreationTool(DrawingView view, Figure prototype) {
    super(view);
    fPrototype = prototype;
}
```

La forma de utilizar esta opción sería:

```
tool=new CreationTool(view(), new RectangleFigure());
tool=new CreationTool(view(), new EllipseFigure());
tool=new CreationTool(view(), new CircleFigure());
```

- Utilizando el patrón de diseño **Factory Method** como se hace en la clase PertFigureCreationTool del ejemplo PERT. Una versión más eficiente de la herramienta de creación Pert que no esta basada en la clonación.



Jerarquía PertFigureCreationTool.

```
/*
 * @(#)PertFigureCreationTool.java 5.1
 *
 */

package CH.ifa.draw.samples.pert;

import java.awt.*;

import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
```

```
/**
 * A more efficient version of the generic Pert creation
 * tool that is not based on cloning.
 */

public class PertFigureCreationTool extends CreationTool {

    public PertFigureCreationTool(DrawingView view) {
        super(view);
    }

    /**
     * Creates a new PertFigure.
     */
    protected Figure createFigure() {
        return new PertFigure();
    }
}
```

El constructor de la clase PertFigureCreationTool llama al constructor CreationTool(view) de su superclase (CreationTool). Esto funciona porque la subclase PertFigureCreationTool sobrescribe el método createFigure.

```
/**
 * Constructs a CreationTool without a prototype.
 * This is for subclasses overriding createFigure.
 */
protected CreationTool(DrawingView view) {
    super(view);
    fPrototype = null;
}
```

La forma de utilizar esta opción sería:

```
tool=new PertFigureCreationTool(view());
```

HERRAMIENTA DE SELECCIÓN

Herramienta para seleccionar y manipular figuras.

- Selecciona las figuras pulsando un botón.
- Selecciona las figuras incluidas en un área.
- Mueve las figuras.
- Las cambia de tamaño.

Una idea errónea que se nos podría ocurrir sería esta:

```
class SelectionTool implements Tool
{
    public void mouseDrag (MouseEvent x, y)
    {
        if (estabaArrastrandoHandle)
```

```

        {
            <código cambiar tamaño>
        }
    else if (estabaMoviendoFigura)
    {
        <código mover figura>
    }
    if (estabaSeleccionandoArea)
    {
        <hacer rubberbanding>
    }
    else .....
}

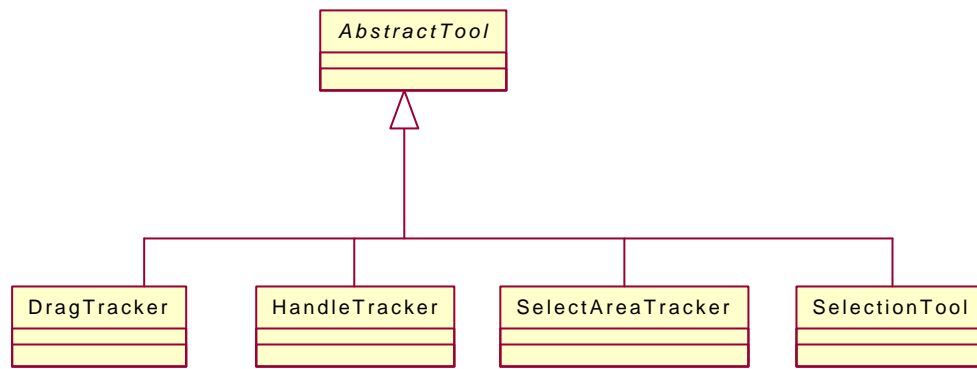
public void mouseUp (MouseEvent x, y)
{
    if (estabaArrastrandoHandle)
    {
        <código cambiar tamaño>
    }
    else if (estabaMoviendoFigura)
    {
        <código mover figura>
    }
    if (estabaSeleccionandoArea)
    {
        <hacer rubberbanding>
    }
    else .....
}
}

```

Esta idea es errónea porque tiene demasiado código mezclado y es difícil de implementar y depurar, y se impide la reutilización. Esta idea sería el antipatrón de desarrollo de software Spaghetti code, que es tener muchos if o switch.

La acción a realizar por la herramienta de selección depende del estado en el que se encuentre: moviendo, cambiando el tamaño y seleccionando un área.

Por lo tanto se utiliza el patrón de diseño **State**. SelectionTool realiza el papel de *Contexto* del patrón State. AbstractTool realiza el papel de *Estado*. DragTracker, HandleTracker, SelectAreaTracker realizan el papel de *EstadoConcreto*. Los diferentes estados son manejados por diferentes herramientas (DragTracker, HandleTracker, SelectAreaTracker), es decir SelectionTool delega el comportamiento específico del estado a su herramienta hijo actual.



Jerarquía SelectionTool.

```

/*
 * @(#)SelectionTool.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.awt.event.MouseEvent;
import java.util.Vector;
import CH.ifa.draw.framework.*;

/**
 * Tool to select and manipulate figures.
 * A selection tool is in one of three states, e.g., background
 * selection, figure selection, handle manipulation. The different
 * states are handled by different child tools.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld032.htm>State</a></b><br>
 * SelectionTool is the StateContext and child is the State.
 * The SelectionTool delegates state specific
 * behavior to its current child tool.
 * <hr>
 */

public class SelectionTool extends AbstractTool {

    private Tool fChild = null;

    public SelectionTool(DrawingView view) {
        super(view);
    }

    /**
     * Handles mouse down events and starts the corresponding tracker.
     */
    public void mouseDown(MouseEvent e, int x, int y)
    {
        // on Windows NT: AWT generates additional mouse down events
        // when the left button is down && right button is clicked.
        // To avoid dead locks we ignore such events
    }
}

```

```

        if (fChild != null)
            return;

        view().freezeView();

        Handle handle = view().findHandle(e.getX(), e.getY());
        if (handle != null) {
            fChild = createHandleTracker(view(), handle);
        }
        else {
            Figure figure = drawing().findFigure(e.getX(), e.getY());
            if (figure != null) {
                fChild = createDragTracker(view(), figure);
            }
            else {
                if (!e.isShiftDown()) {
                    view().clearSelection();
                }
                fChild = createAreaTracker(view());
            }
        }
        fChild.mouseDown(e, x, y);
    }

    /**
     * Handles mouse drag events. The events are forwarded to the
     * current tracker.
     */
    public void mouseDrag(MouseEvent e, int x, int y) {
        if (fChild != null) // JDK1.1 doesn't guarantee mouseDown,
                           // mouseDrag, mouseUp
            fChild.mouseDrag(e, x, y);
    }

    /**
     * Handles mouse up events. The events are forwarded to the
     * current tracker.
     */
    public void mouseUp(MouseEvent e, int x, int y) {
        view().unfreezeView();
        if (fChild != null) // JDK1.1 doesn't guarantee mouseDown,
                           // mouseDrag, mouseUp
            fChild.mouseUp(e, x, y);
        fChild = null;
    }

    /**
     * Factory method to create a Handle tracker. It is used to track
     * a handle.
     */
    protected Tool createHandleTracker(DrawingView view, Handle
handle) {
        return new HandleTracker(view, handle);
    }

    /**
     * Factory method to create a Drag tracker. It is used to drag a
     * figure.

```

```

    */
    protected Tool createDragTracker(DrawingView view, Figure f) {
        return new DragTracker(view, f);
    }

    /**
     * Factory method to create an area tracker. It is used to select
     * an area.
     */
    protected Tool createAreaTracker(DrawingView view) {
        return new SelectAreaTracker(view);
    }
}

```

Esta clase también utiliza el patrón **Factory Method** para crear un `HandleTracker`, `DragTracker` y `SelectAreaTracker`.

```

    /**
     * Factory method to create a Handle tracker. It is used to track
     * a handle.
     */
    protected Tool createHandleTracker(DrawingView view, Handle
handle) {
        return new HandleTracker(view, handle);
    }

    /**
     * Factory method to create a Drag tracker. It is used to drag a
     * figure.
     */
    protected Tool createDragTracker(DrawingView view, Figure f) {
        return new DragTracker(view, f);
    }

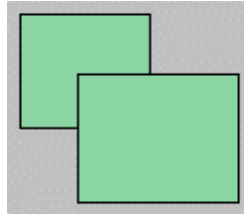
    /**
     * Factory method to create an area tracker. It is used to select
     * an area.
     */
    protected Tool createAreaTracker(DrawingView view) {
        return new SelectAreaTracker(view);
    }
}

```

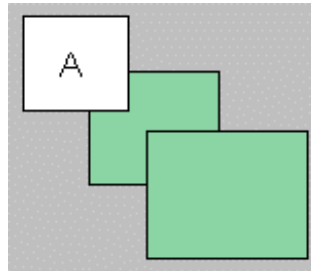
3.5.3.3. Actualización de pantalla

Para actualizar la pantalla hay que tener en cuenta algunos detalles:

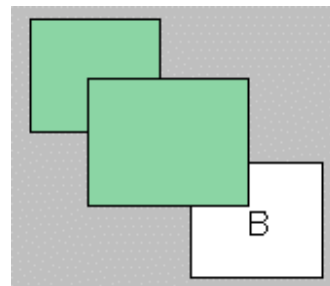
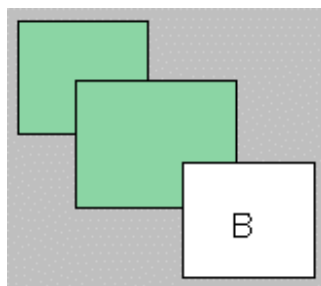
Si se mueve una figura desde A hasta B (`moveBy`):



- La figura no puede limpiar la zona que ocupa la figura en A porque no sabe lo que puede haber debajo de ella, ya que podría haber otra figura.



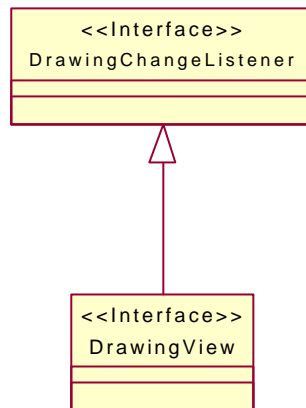
- La figura no debe dibujarse en B sin más, ya que puede haber otra figura encima y no puede machacar la figura que esta al frente.



El DrawingView es el que debe realizar la actualización:

- Representa al lugar donde se dibujan las figuras.
- Sabe cual es su color de fondo.
- Sabe donde están el resto de las figuras y uqe partes se solapan.

Cuando una herramienta manipula una figura, para que el DrawingView la actualice en pantalla se utiliza el patrón **Observer**. DrawingView observa los dibujos que cambian a través del interface DrawingChangeListener.



`DrawingChangeListener` es un escuchador interesado en los cambios del `Drawing`.

```

/*
 * @(#)DrawingChangeListener.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.Rectangle;
import java.util.EventListener;

/**
 * Listener interested in Drawing changes.
 */
public interface DrawingChangeListener extends EventListener {

    /**
     * Sent when an area is invalid
     */
    public void drawingInvalidated(DrawingChangeEvent e);

    /**
     * Sent when the drawing wants to be refreshed
     */
    public void drawingRequestUpdate(DrawingChangeEvent e);
}

```

`DrawingChangeEvent` es el evento pasado a los `DrawingChangeListeners`.

```

/*
 * @(#)DrawingChangeEvent.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.Rectangle;
import java.util.EventObject;

/**

```



```

* The event passed to DrawingChangeListeners.
*
*/
public class DrawingChangeEvent extends EventObject {

    private Rectangle fRectangle;

    /**
     * Constructs a drawing change event.
     */
    public DrawingChangeEvent(Drawing source, Rectangle r) {
        super(source);
        fRectangle = r;
    }

    /**
     * Gets the changed drawing
     */
    public Drawing getDrawing() {
        return (Drawing)getSource();
    }

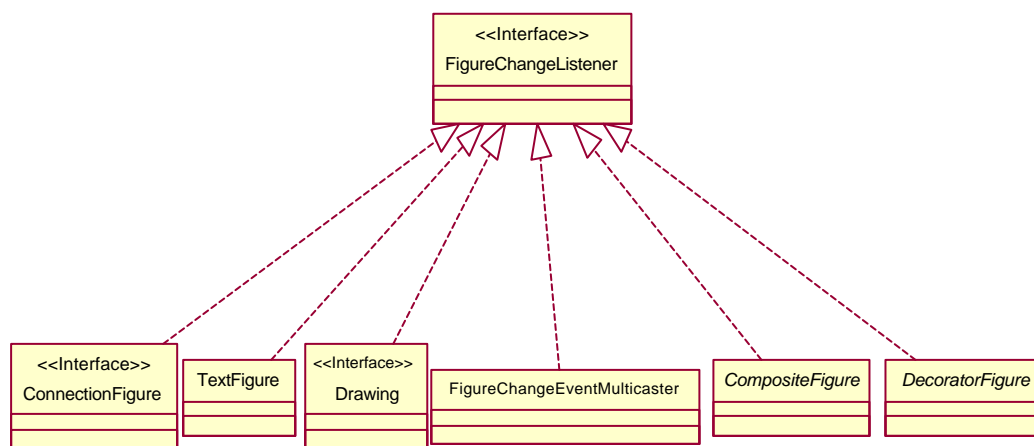
    /**
     * Gets the changed rectangle
     */
    public Rectangle getInvalidatedRectangle() {
        return fRectangle;
    }
}

```

El Listener de una Figure es su Drawing. Este a su vez utiliza el patrón **Observer** para notificar a sus DrawingView.

El DrawingView se limita a calcular la unión de las áreas a repintar.

El patrón Observer en Drawing se utiliza para desacoplar el Drawing de sus vistas y capacitar múltiples vistas.



FigureChangeListener es el escuchador interesado en los cambios sobre Figure (sobre Drawing).

```

/*
 * @(#)FigureChangeListener.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.Rectangle;
import java.util.EventListener;

/**
 * Listener interested in Figure changes.
 */
public interface FigureChangeListener extends EventListener {

    /**
     * Sent when an area is invalid
     */
    public void figureInvalidated(FigureChangeEvent e);

    /**
     * Sent when a figure changed
     */
    public void figureChanged(FigureChangeEvent e);

    /**
     * Sent when a figure was removed
     */
    public void figureRemoved(FigureChangeEvent e);

    /**
     * Sent when requesting to remove a figure.
     */
    public void figureRequestRemove(FigureChangeEvent e);

    /**
     * Sent when an update should happen.
     */
    public void figureRequestUpdate(FigureChangeEvent e);
}

```

FigureChangeEvent es el evento FigureChange pasado a los FigureChangeListeners.

```

/*
 * @(#)FigureChangeEvent.java 5.1
 *
 */

package CH.ifa.draw.framework;

```

```
import java.awt.Rectangle;
import java.util.EventObject;

/**
 * FigureChange event passed to FigureChangeListeners.
 *
 */
public class FigureChangeEvent extends EventObject {

    private Rectangle fRectangle;
    private static final Rectangle fgEmptyRectangle = new
Rectangle(0, 0, 0, 0);

    /**
     * Constructs an event for the given source Figure. The rectangle
     * is the area to be invalidated.
     */
    public FigureChangeEvent(Figure source, Rectangle r) {
        super(source);
        fRectangle = r;
    }

    public FigureChangeEvent(Figure source) {
        super(source);
        fRectangle = fgEmptyRectangle;
    }

    /**
     * Gets the changed figure
     */
    public Figure getFigure() {
        return (Figure)getSource();
    }

    /**
     * Gets the changed rectangle
     */
    public Rectangle getInvalidatedRectangle() {
        return fRectangle;
    }
}
```

La figura avisa cuando se necesite repintar un área.

```
/*
 * @(#)AbstractFigure.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

import java.awt.*;
import java.util.*;
import java.io.*;
```

```

/**
 * AbstractFigure provides default implementations for
 * the Figure interface.
 *
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld036.htm>Template Method</a></b><br>
 * Template Methods implement default and invariant behavior for
 * figure subclasses.
 * <hr>
 *
 * @see Figure
 * @see Handle
 */

public abstract class AbstractFigure implements Figure {

    /**
     * The listeners for a figure's changes.
     * @see #invalidate
     * @see #changed
     * @see #willChange
     */
    private transient FigureChangeListener fListener;

    .....

    protected AbstractFigure() { }

    /**
     * Moves the figure by the given offset.
     */
    public void moveBy(int dx, int dy) {
        willChange();
        basicMoveBy(dx, dy);
        changed();
    }

    /**
     * Moves the figure. This is the
     * method that subclasses override. Clients usually
     * call displayBox.
     * @see moveBy
     */
    protected abstract void basicMoveBy(int dx, int dy);

    /**
     * Changes the display box of a figure. Clients usually
     * call this method. It changes the display box
     * and announces the corresponding change.
     * @param origin the new origin
     * @param corner the new corner
     * @see displayBox
     */
    public void displayBox(Point origin, Point corner) {
        willChange();
    }

```

```

        basicDisplayBox(origin, corner);
        changed();
    }

    /**
     * Sets the display box of a figure. This is the
     * method that subclasses override. Clients usually
     * call displayBox.
     * @see displayBox
     */
    public abstract void basicDisplayBox(Point origin, Point corner);
    .....

    /**
     * Adds a listener for this figure.
     */
    public void addFigureChangeListener(FigureChangeListener l) {
        fListener = FigureChangeEventMulticaster.add(fListener, l);
    }

    /**
     * Removes a listener for this figure.
     */
    public void removeFigureChangeListener(FigureChangeListener l) {
        fListener = FigureChangeEventMulticaster.remove(fListener, l);
    }

    /**
     * Gets the figure's listeners.
     */
    public FigureChangeListener listener() {
        return fListener;
    }
    .....

    /**
     * Invalidates the figure. This method informs the listeners
     * that the figure's current display box is invalid and should be
     * refreshed.
     */
    public void invalidate() {
        if (fListener != null) {
            Rectangle r = displayBox();
            r.grow(Handle.HANDLESIZE, Handle.HANDLESIZE);
            fListener.figureInvalidated(new FigureChangeEvent(this,
r));
        }
    }

    /**
     * Informes that a figure is about to change something that
     * affects the contents of its display box.
     *
     * @see Figure#willChange
     */
    public void willChange() {

```

```
        invalidate();
    }

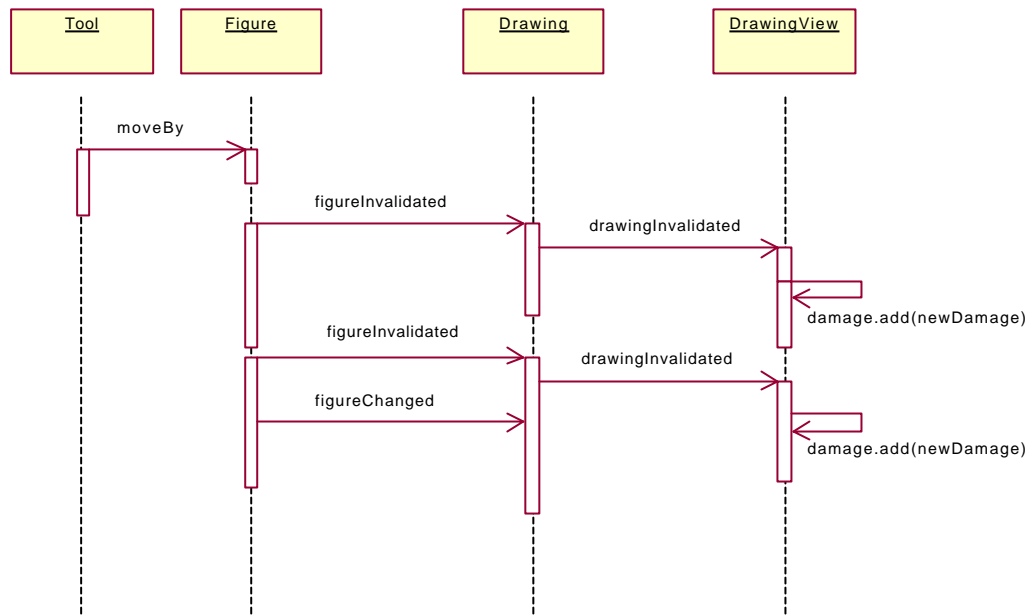
    /**
     * Informs that a figure changed the area of its display box.
     *
     * @see FigureChangeEvent
     * @see Figure#changed
     */
    public void changed() {
        invalidate();
        if (fListener != null)
            fListener.figureChanged(new FigureChangeEvent(this));
    }

    .....
}
```

El método `changed` se utiliza para lanzar el evento `figureChanged`. Indica que ha cambiado su `displayBox`. Se puede usar para saber cuando ha cambiado de tamaño o posición una figura. Por ejemplo un `CompositeFigure` (`PertFigure`) lo puede utilizar para saber si uno de sus componentes (un texto) ha cambiado de tamaño y reajustar su ancho o alto (actualizar el layout de sus subcomponentes). Lo utilizan también las `ConnectionFigure` (líneas de conexión) para saber si se ha movido alguna de las figuras a las que están conectadas para mover el extremo con ellas.

El método `invalidate` lanza el evento `figureInvalidated`. Indica que hay que repintar el `displayBox`. Se utiliza para ir añadiendo áreas a repintar al `DrawingView`. Este mecanismo de repintado es interno entre la figura y su `DrawingView`. No es necesario escuchar este evento. Es exclusivamente para temas de repintado. Si se desea saber si una figura ha cambiado hay que usar `figureChanged`. De la misma manera, cuando se implemente una nueva figura hay que usar `changed` en vez de `invalidate`. A `invalidate` ya se encarga de llamarlo en los lugares adecuados la clase base de la figura cuando sea necesario.

Generalmente las figuras se limitan a añadir zonas dañadas. Si alguna como la `ImageFigure` quiere asegurarse además de que se actualice en el momento en vez de esperar a `checkDamage` entonces tiene que lanzar este evento en vez de `figureInvalidated`.



Cualquier método de la clase figura que necesite un reflejo en pantalla necesita llamar a invalidate (move, displayBox, etc).

Cuando una nueva figura redefina alguno de estos métodos deberá recordar invocar a invalidate y changed en la nueva implementación.

Para que no sea necesario recordar esto se utiliza el patrón de diseño **Template Method** en la clase AbstractFigure que implementa un comportamiento por defecto e invariante para las subclases figuras (subclases de AbstractFigure). Template Method define el esqueleto de un algoritmo.

AbstractFigure realiza el papel de *ClaseAbstracta* en el patrón Template Method y todas las clases que heredan de AbstractFigure juegan el papel de *ClaseConcreta*.

```

/*
 * @(#)AbstractFigure.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

import java.awt.*;
import java.util.*;
import java.io.*;

/**
 * AbstractFigure provides default implementations for
 * the Figure interface.
 */

```

```

* <hr>
* <b>Design Patterns</b><P>
* 
* <b><a href=../pattlets/sld036.htm>Template Method</a></b><br>
* Template Methods implement default and invariant behavior for
* figure subclasses.
* <hr>
*
* @see Figure
* @see Handle
* /

public abstract class AbstractFigure implements Figure {

    /**
     * The listeners for a figure's changes.
     * @see #invalidate
     * @see #changed
     * @see #willChange
     */
    private transient FigureChangeListener fListener;

    .....

    protected AbstractFigure() { }

    /**
     * Moves the figure by the given offset.
     */
    public void moveBy(int dx, int dy) {
        willChange();
        basicMoveBy(dx, dy);
        changed();
    }

    /**
     * Moves the figure. This is the
     * method that subclassers override. Clients usually
     * call displayBox.
     * @see moveBy
     */
    protected abstract void basicMoveBy(int dx, int dy);

    /**
     * Changes the display box of a figure. Clients usually
     * call this method. It changes the display box
     * and announces the corresponding change.
     * @param origin the new origin
     * @param corner the new corner
     * @see displayBox
     */
    public void displayBox(Point origin, Point corner) {
        willChange();
        basicDisplayBox(origin, corner);
        changed();
    }

    /**

```



```

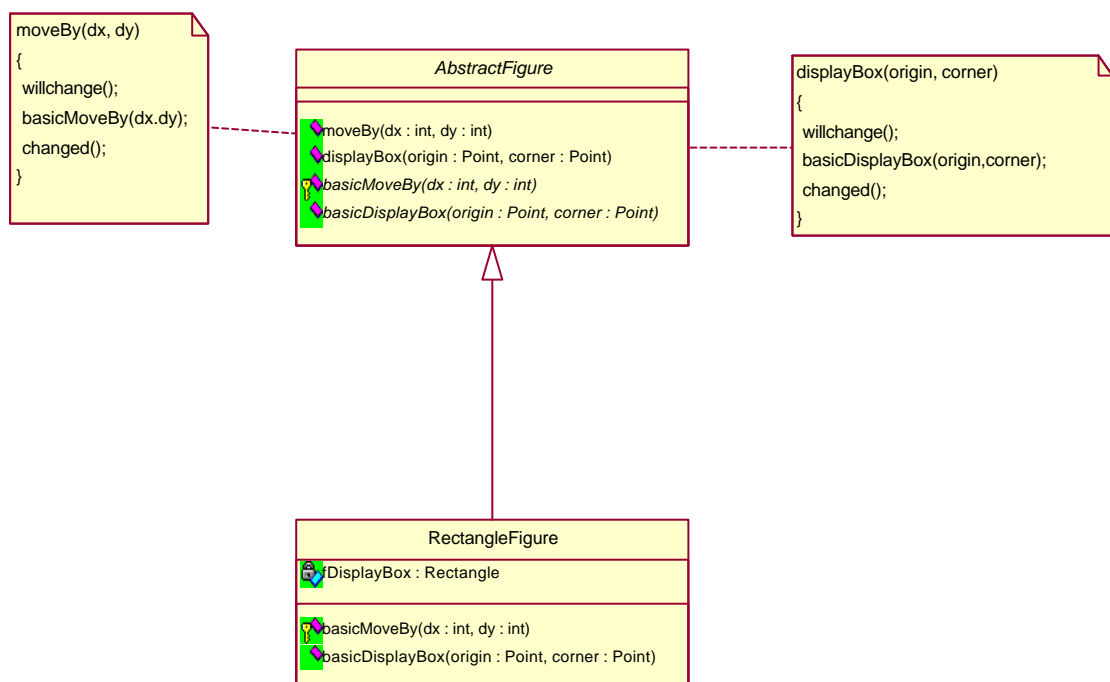
    * Sets the display box of a figure. This is the
    * method that subclasses override. Clients usually
    * call displayBox.
    * @see displayBox
    */
    public abstract void basicDisplayBox(Point origin, Point corner);

    .....

}

```

Como ejemplo del uso del patrón Template Method en una figura concreta vamos a ver la clase RectangleFigure.



Utilización del patrón Template Method.

```

/*
 * @(#)RectangleFigure.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.io.IOException;
import java.util.Vector;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.*;

/**

```

```
    * A rectangle figure.
    */
public class RectangleFigure extends AttributeFigure {

    private Rectangle    fDisplayBox;

    /*
     * Serialization support.
     */
    private static final long serialVersionUID = 184722075881789163L;
    private int rectangleFigureSerializedDataVersion = 1;

    public RectangleFigure() {
        this(new Point(0,0), new Point(0,0));
    }

    public RectangleFigure(Point origin, Point corner) {
        basicDisplayBox(origin,corner);
    }

    public void basicDisplayBox(Point origin, Point corner) {
        fDisplayBox = new Rectangle(origin);
        fDisplayBox.add(corner);
    }

    public Vector handles() {
        Vector handles = new Vector();
        BoxHandleKit.addHandles(this, handles);
        return handles;
    }

    public Rectangle displayBox() {
        return new Rectangle(
            fDisplayBox.x,
            fDisplayBox.y,
            fDisplayBox.width,
            fDisplayBox.height);
    }

    protected void basicMoveBy(int x, int y) {
        fDisplayBox.translate(x,y);
    }

    public void drawBackground(Graphics g) {
        Rectangle r = displayBox();
        g.fillRect(r.x, r.y, r.width, r.height);
    }

    public void drawFrame(Graphics g) {
        Rectangle r = displayBox();
        g.drawRect(r.x, r.y, r.width-1, r.height-1);
    }

    //-- store / load -----

    public void write(StorableOutput dw) {
        super.write(dw);
        dw.writeInt(fDisplayBox.x);
    }
}
```

```

        dw.writeInt(fDisplayBox.y);
        dw.writeInt(fDisplayBox.width);
        dw.writeInt(fDisplayBox.height);
    }

    public void read(StorableInput dr) throws IOException {
        super.read(dr);
        fDisplayBox = new Rectangle(
            dr.readInt(),
            dr.readInt(),
            dr.readInt(),
            dr.readInt());
    }
}

```

El DrawingView va acumulando áreas que necesitan actualización. Si se actualizan muy frecuentemente en pantalla baja el rendimiento y si se actualizan con poca frecuencia el usuario percibirá desajustes.

JHotDraw actualiza a la “velocidad del usuario”. Después de cada evento de usuario actualiza todas las áreas de una sola vez.

```

package CH.ifa.draw.standard;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

/**
 * The standard implementation of DrawingView.
 * @see DrawingView
 * @see Painter
 * @see Tool
 */

public class StandardDrawingView
    extends Panel
    implements DrawingView,
               MouseListener,
               MouseMotionListener,
               KeyListener {
    .....

    /**
     * the accumulated damaged area
     */
    private transient Rectangle fDamage = null;
}

```

```

    * The list of currently selected figures.
    */
    transient private Vector fSelection;

.....

    /**
     * Constructs the view.
     */
    public StandardDrawingView(DrawingEditor editor, int width, int
height) {
        fEditor = editor;
        fViewSize = new Dimension(width,height);
        fLastClick = new Point(0, 0);
        fConstrainer = null;
        fSelection = new Vector();
        setDisplayUpdate(new BufferedUpdateStrategy());
        setBackground(Color.lightGray);

        addMouseListener(this);
        addMouseMotionListener(this);
        addKeyListener(this);
    }

.....

    /**
     * Handles mouse down events. The event is delegated to the
     * currently active tool.
     * @return whether the event was handled.
     */
    public void mousePressed(MouseEvent e) {
        requestFocus(); // JDK1.1
        Point p = constrainPoint(new Point(e.getX(), e.getY()));
        fLastClick = new Point(e.getX(), e.getY());
        tool().mouseDown(e, p.x, p.y);
        checkDamage();
    }

    /**
     * Handles mouse drag events. The event is delegated to the
     * currently active tool.
     * @return whether the event was handled.
     */
    public void mouseDragged(MouseEvent e) {
        Point p = constrainPoint(new Point(e.getX(), e.getY()));
        tool().mouseDrag(e, p.x, p.y);
        checkDamage();
    }

    /**
     * Handles mouse move events. The event is delegated to the
     * currently active tool.
     * @return whether the event was handled.
     */
    public void mouseMoved(MouseEvent e) {
        tool().mouseMove(e, e.getX(), e.getY());
    }

```

```

/**
 * Handles mouse up events. The event is delegated to the
 * currently active tool.
 * @return whether the event was handled.
 */
public void mouseReleased(MouseEvent e) {
    Point p = constrainPoint(new Point(e.getX(), e.getY()));
    tool().mouseUp(e, p.x, p.y);
    checkDamage();
}

/**
 * Handles key down events. Cursor keys are handled
 * by the view the other key events are delegated to the
 * currently active tool.
 * @return whether the event was handled.
 */
public void keyPressed(KeyEvent e) {
    int code = e.getKeyCode();
    if ((code == KeyEvent.VK_BACK_SPACE) || (code ==
KeyEvent.VK_DELETE)) {
        Command cmd = new DeleteCommand("Delete", this);
        cmd.execute();
    } else if (code == KeyEvent.VK_DOWN || code == KeyEvent.VK_UP
||
        code == KeyEvent.VK_RIGHT || code == KeyEvent.VK_LEFT) {
        handleCursorKey(code);
    } else {
        tool().keyDown(e, code);
    }
    checkDamage();
}

.....

private void moveSelection(int dx, int dy) {
    FigureEnumeration figures = selectionElements();
    while (figures.hasMoreElements())
        figures.nextFigure().moveBy(dx, dy);
    checkDamage();
}

/**
 * Refreshes the drawing if there is some accumulated damage
 */
public synchronized void checkDamage() {
    Enumeration each = drawing().drawingChangeListeners();
    while (each.hasMoreElements()) {
        Object l = each.nextElement();
        if (l instanceof DrawingView) {
            ((DrawingView)l).repairDamage();
        }
    }
}

public void repairDamage() {
    if (fDamage != null) {

```

```

        repaint(fDamage.x, fDamage.y, fDamage.width,
fDamage.height);
        fDamage = null;
    }
}

public void drawingInvalidated(DrawingChangeEvent e) {
    Rectangle r = e.getInvalidatedRectangle();
    if (fDamage == null)
        fDamage = r;
    else
        fDamage.add(r);
}

public void drawingRequestUpdate(DrawingChangeEvent e) {
    repairDamage();
}

.....
}

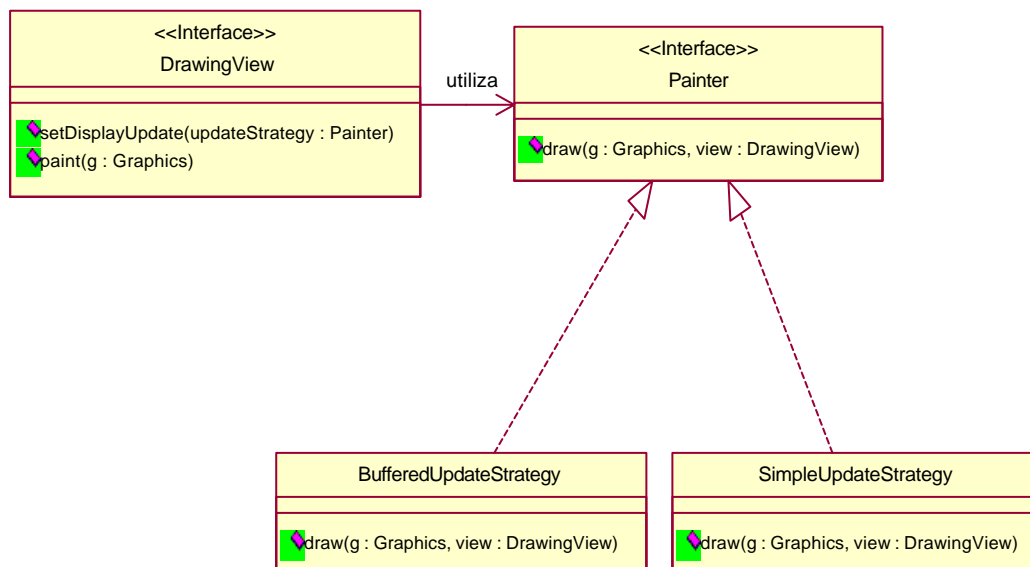
```

JHotDraw imprime todas las figuras del área dañada de abajo hacia arriba.

Para actualizar hay dos opciones:

1. Con buffer: No requiere borrar el fondo y evita parpadeos.
2. Sin buffer o actualización directa: Es más sencillo y consume menos recursos.

JHotDraw utiliza cualquiera de las dos. Para ello Painter define el interface para dibujar un estrato en un DawingView. Painter utiliza el patrón **Strategy**. El DrawingView juega el papel del *Cliente* en el patrón Strategy. El interface Painter juega el papel de *StrategyAbstracto*. BufferedUpdateStrategy y SimpleUpdateStrategy juegan el papel de *StrategyConcreto*.



Utilización del patrón Strategy.

```

public class StandardDrawingView
    extends Panel
    implements DrawingView,
               MouseListener,
               MouseMotionListener,
               KeyListener {

    .....

    /**
     * The update strategy used to repair the view.
     */
    private Painter fUpdateStrategy;

    .....

    /**
     * Constructs the view.
     */
    public StandardDrawingView(DrawingEditor editor, int width, int
height) {
        fEditor = editor;
        fViewSize = new Dimension(width,height);
        fLastClick = new Point(0, 0);
        fConstrainer = null;
        fSelection = new Vector();
        setDisplayUpdate(new BufferedUpdateStrategy());
        setBackground(Color.lightGray);

        addMouseListener(this);
        addMouseMotionListener(this);
        addKeyListener(this);
    }

    .....

    /**
     * Sets the current display update strategy.
     * @see UpdateStrategy
     */
    public void setDisplayUpdate(Painter updateStrategy) {
        fUpdateStrategy = updateStrategy;
    }

    .....

    /**
     * Updates the drawing view.
     */
    public void update(Graphics g) {
        paint(g);
    }

    /**
     * Paints the drawing view. The actual drawing is delegated to
     * the current update strategy.

```

```

    * @see Painter
    */
    public void paint(Graphics g) {
        fUpdateStrategy.draw(g, this);
    }

    .....
}

```

El interface Painter encapsula un algoritmo para dibujar algo en el DrawingView.

```

/*
 * @(#)Painter.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.*;
import java.io.Serializable;

/**
 * Painter defines the interface for drawing a layer
 * into a DrawingView.<p>
 *
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld034.htm>Strategy</a></b><br>
 * Painter encapsulates an algorithm to render something in
 * the DrawingView. The DrawingView plays the role of the
 * StrategyContext.
 * <hr>
 * @see DrawingView
 */

public interface Painter extends Serializable {

    /**
     * Draws into the given DrawingView.
     */
    public void draw(Graphics g, DrawingView view);

}

```

BufferedUpdateStrategy implementa una estrategia de actualización que primero dibuja una vista en un buffer y seguidamente copia el buffer al DrawingView.

```

/*
 * @(#)BufferedUpdateStrategy.java 5.1
 *
 */

package CH.ifa.draw.standard;

```



```
import java.awt.*;
import java.awt.image.*;
import CH.ifa.draw.framework.*;

/**
 * The BufferedUpdateStrategy implements an update
 * strategy that first draws a view into a buffer
 * followed by copying the buffer to the DrawingView.
 * @see DrawingView
 */

public class BufferedUpdateStrategy
    implements Painter {

    /**
     * The offscreen image
     */
    transient private Image fOffscreen;
    private int fImagewidth = -1;
    private int fImageheight = -1;

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = 6489532222954612824L;
    private int bufferedUpdateSerializedDataVersion = 1;

    /**
     * Draws the view contents.
     */
    public void draw(Graphics g, DrawingView view) {
        // create the buffer if necessary
        Dimension d = view.getSize();
        if ((fOffscreen == null) || (d.width != fImagewidth)
            || (d.height != fImageheight)) {
            fOffscreen = view.createImage(d.width, d.height);
            fImagewidth = d.width;
            fImageheight = d.height;
        }

        // let the view draw on offscreen buffer
        Graphics g2 = fOffscreen.getGraphics();
        view.drawAll(g2);

        g.drawImage(fOffscreen, 0, 0, view);
    }
}
```

El SimpleUpdateStrategy implementa una estrategia de actualización que redibuja directamente un DrawingView.

```
/*
 * @(#)SimpleUpdateStrategy.java 5.1
 *
 */
```

```

package CH.ifa.draw.standard;

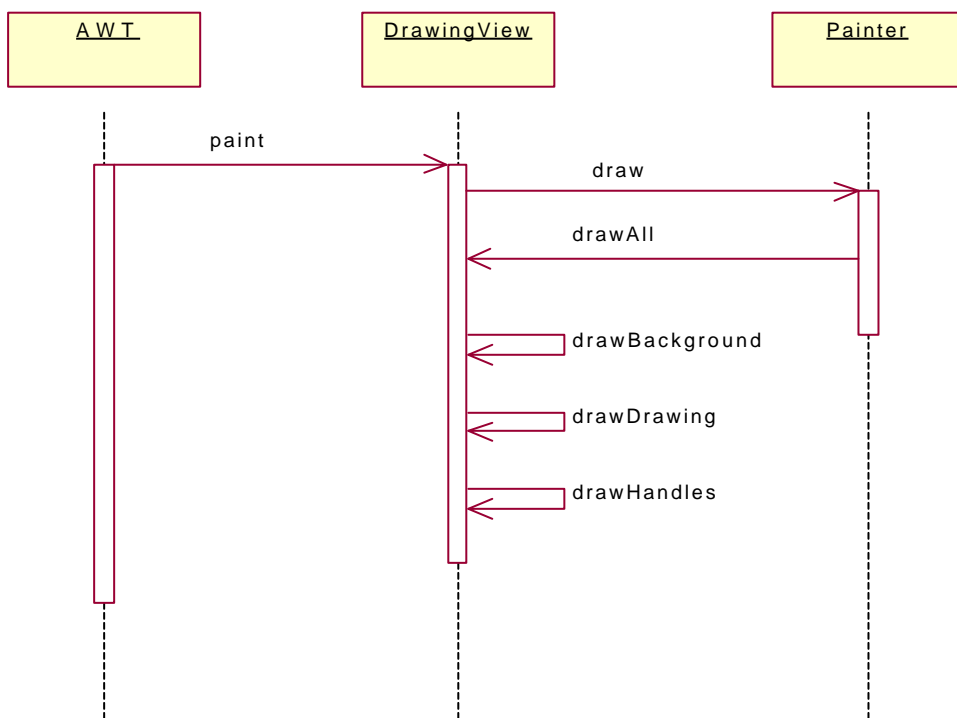
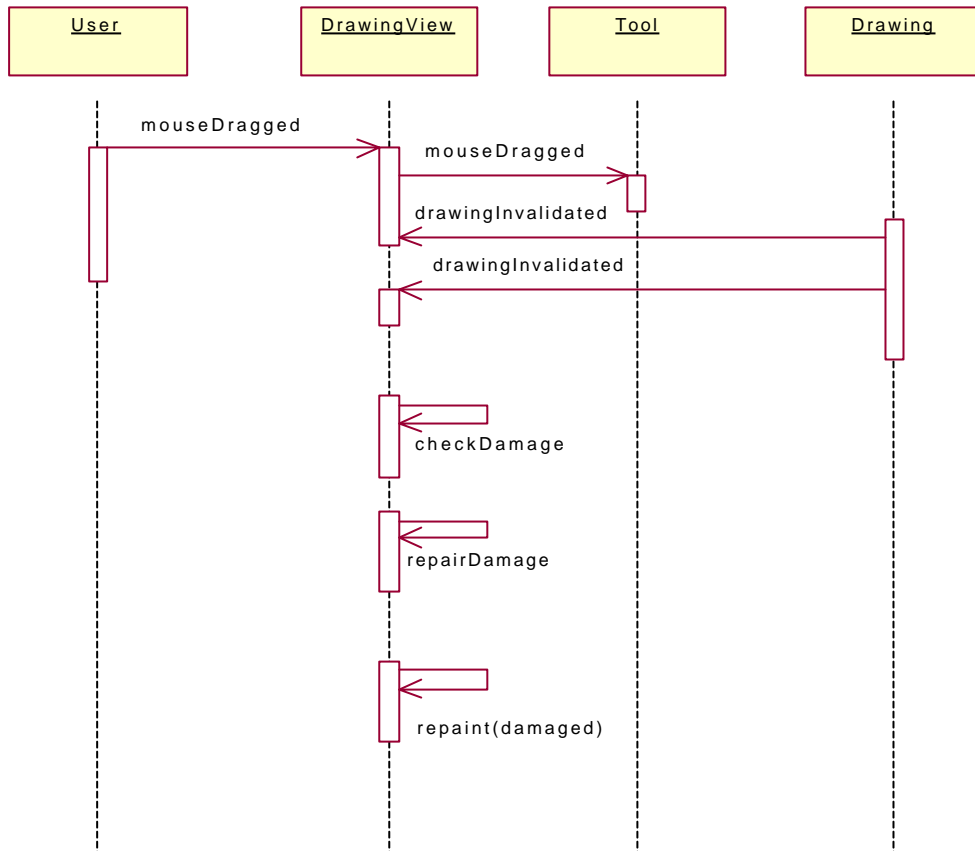
import java.awt.*;
import CH.ifa.draw.framework.*;

/**
 * The SimpleUpdateStrategy implements an update
 * strategy that directly redraws a DrawingView.
 * @see DrawingView
 */
public class SimpleUpdateStrategy implements Painter {

    /**
     * Serialization support. In JavaDraw only the Drawing is
     * serialized.
     * However, for beans support SimpleUpdateStrategy supports
     * serialization
     */
    private static final long serialVersionUID = -
7539925820692134566L;

    /**
     * Draws the view contents.
     */
    public void draw(Graphics g, DrawingView view) {
        view.drawAll(g);
    }
}

```



3.5.3.4. Manipulación de figuras

Los requisitos que se deben cumplir para la manipulación de figuras son: cambio de tamaño, rotaciones, manipulación de vértices, inicio de conexiones, cambio de inicio y final de la conexión, cualquier otra manipulación específica de una nueva figura, etc...

El problema es que cada figura tendrá una forma distinta de ser manipulada:

```
displayBox(x, y, width, height)
rotate(angle)
movePoint(index, newX, newY)
new Connection()
setConnectionStart(...) / setConnectionEnd(...)
.....
```

Una idea errónea que se nos podría ocurrir sería esta:

```
class HandleTracker
{
    void mouseDrag(x,y)
    {
        type=fig.handleType(x,y);
        if (type==ROTATION)
        {
            fig.rotate(...);
        }
        else if (type==MOVE)
        {
            fig.move(...);
        }
        else if (type==CONNECTION)
        {
            line.setEnd(...);
        }
        else .....
    }
}
```

Esta idea es errónea porque tiene demasiado código mezclado y es difícil de implementar y depurar, y se impide la reutilización. Esta idea sería el antipatrón de desarrollo de software Spaghetti code, que es tener muchos if o switch.

Otra aproximación podría ser la siguiente:

```
interface Figure
{
    .....

    int getHandle (x,y);
}
```

```

.....

void invokeStart( int handleIndex, x, y);
void invokeStep( int handleIndex, x, y);
void invokeEnd( int handleIndex, x, y);

.....

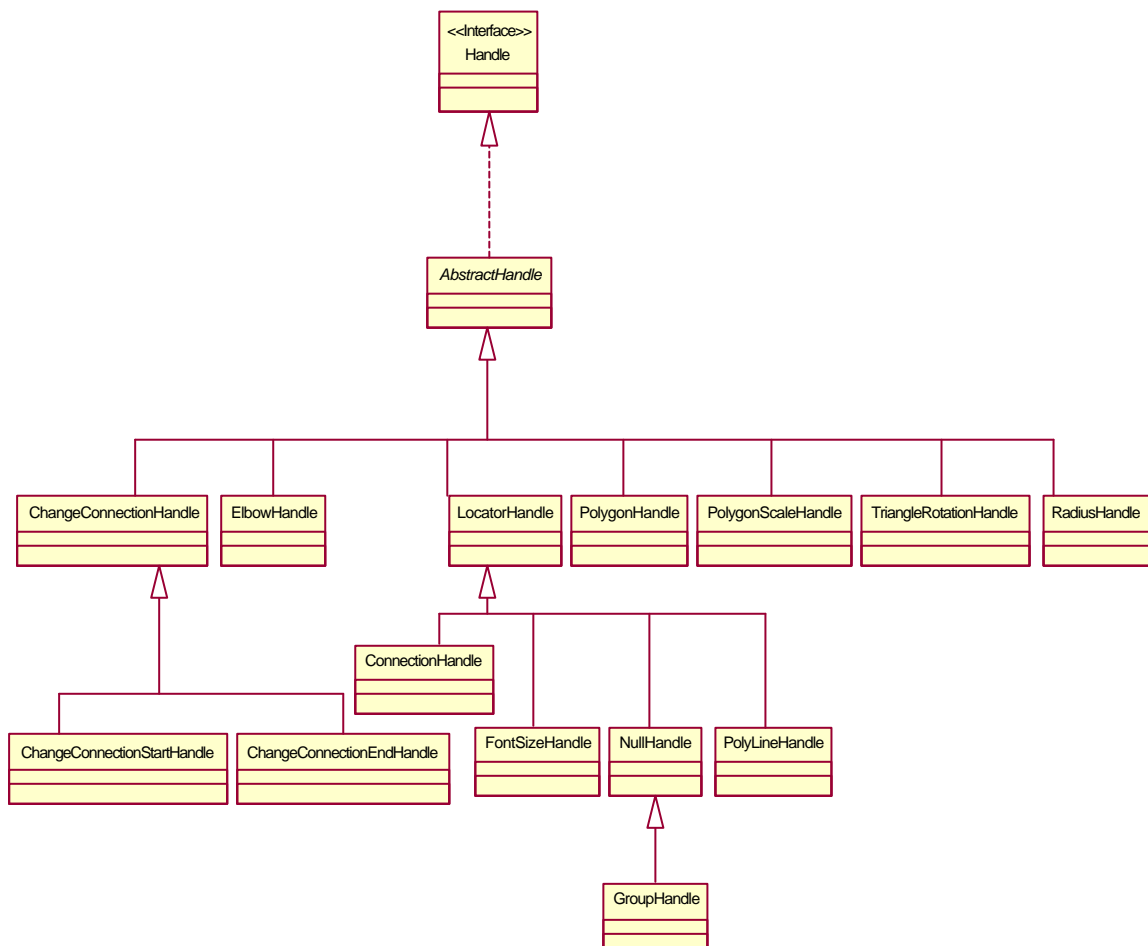
void drawHandles();

.....

}

```

Se desea que la misma herramienta de selección pueda manipular figuras que tienen distintos interfaces. Por lo tanto la solución más adecuada es utilizar el patrón **Adapter**.



Jerarquía Handle.

Los Handles son utilizados para cambiar una figura por manipulación directa. Los manejadores conocen sus propias figuras y proporcionan métodos para localizar y ubicar el manejo sobre la figura y seguir la pista a los cambios.

Los Handles adaptan las operaciones que manipulan una figura a un interface común. El interface Handle juega el papel de **IFTarget** del patrón Adapter.

```

/*
 * @(#)Handle.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.*;

/**
 * Handles are used to change a figure by direct manipulation.
 * Handles know their owning figure and they provide methods to
 * locate the handle on the figure and to track changes.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld004.htm>Adapter</a></b><br>
 * Handles adapt the operations to manipulate a figure to a common
 interface.
 *
 * @see Figure
 */
public interface Handle {

    public static final int HANDLESIZE = 8;

    /**
     * Locates the handle on the figure. The handle is drawn
     * centered around the returned point.
     */
    public abstract Point locate();

    /**
     * @deprecated As of version 4.1,
     * use invokeStart(x, y, drawingView)
     * Tracks the start of the interaction. The default implementation
     * does nothing.
     * @param x the x position where the interaction started
     * @param y the y position where the interaction started
     */
    public void invokeStart(int x, int y, Drawing drawing);

    /**
     * @deprecated As of version 4.1,
     * use invokeStart(x, y, drawingView)
     * Tracks the start of the interaction. The default implementation
     * does nothing.
     * @param x the x position where the interaction started
     * @param y the y position where the interaction started
     * @param view the handles container
     */
    public void invokeStart(int x, int y, DrawingView view);

```

```

/**
 * @deprecated As of version 4.1,
 * use invokeStep(x, y, anchorX, anchorY, drawingView)
 *
 * Tracks a step of the interaction.
 * @param dx x delta of this step
 * @param dy y delta of this step
 */
public void invokeStep (int dx, int dy, Drawing drawing);

/**
 * Tracks a step of the interaction.
 * @param x the current x position
 * @param y the current y position
 * @param anchorX the x position where the interaction started
 * @param anchorY the y position where the interaction started
 */
public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view);

/**
 * Tracks the end of the interaction.
 * @param x the current x position
 * @param y the current y position
 * @param anchorX the x position where the interaction started
 * @param anchorY the y position where the interaction started
 */
public void invokeEnd(int x, int y, int anchorX, int anchorY,
DrawingView view);

/**
 * @deprecated As of version 4.1,
 * use invokeEnd(x, y, anchorX, anchorY, drawingView).
 *
 * Tracks the end of the interaction.
 */
public void invokeEnd (int dx, int dy, Drawing drawing);

/**
 * Gets the handle's owner.
 */
public Figure owner();

/**
 * Gets the display box of the handle.
 */
public Rectangle displayBox();

/**
 * Tests if a point is contained in the handle.
 */
public boolean containsPoint(int x, int y);

/**
 * Draws this handle.
 */
public void draw(Graphics g);
}

```

La clase `AbstractHandle` esta realizando también el papel *IFTarget* del patrón Adapter. La clase `AbstractHandle` proporciona una implementación por defecto para el interface `Handle`.

```

/*
 * @(#)AbstractHandle.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.framework.*;
import java.awt.*;

/**
 * AbstractHandle provides default implementation for the
 * Handle interface.
 *
 * @see Figure
 * @see Handle
 */
public abstract class AbstractHandle implements Handle {

    /**
     * The standard size of a handle.
     */
    public static final int HANDLESIZE = 8;
    private Figure fOwner;

    /**
     * Initializes the owner of the figure.
     */
    public AbstractHandle(Figure owner) {
        fOwner = owner;
    }

    /**
     * Locates the handle on the figure. The handle is drawn
     * centered around the returned point.
     */
    public abstract Point locate();

    /**
     * @ deprecated As of version 4.1,
     * use invokeStart(x, y, drawingView)
     * Tracks the start of the interaction. The default implementation
     * does nothing.
     * @param x the x position where the interaction started
     * @param y the y position where the interaction started
     */
    public void invokeStart(int x, int y, Drawing drawing) { }

    /**
     * @ deprecated As of version 4.1,
     * use invokeStart(x, y, drawingView)

```



```

    * Tracks the start of the interaction. The default implementation
    * does nothing.
    * @param x the x position where the interaction started
    * @param y the y position where the interaction started
    * @param view the handles container
    */
    public void invokeStart(int x, int y, DrawingView view) {
        invokeStart(x, y, view.drawing());
    }

    /**
     * @ deprecated As of version 4.1,
     * use invokeStep(x, y, anchorX, anchorY, drawingView)
     *
     * Tracks a step of the interaction.
     * @param dx x delta of this step
     * @param dy y delta of this step
     */
    public void invokeStep (int dx, int dy, Drawing drawing) { }

    /**
     * Tracks a step of the interaction.
     * @param x the current x position
     * @param y the current y position
     * @param anchorX the x position where the interaction started
     * @param anchorY the y position where the interaction started
     */
    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        invokeStep(x-anchorX, y-anchorY, view.drawing());
    }

    /**
     * Tracks the end of the interaction.
     * @param x the current x position
     * @param y the current y position
     * @param anchorX the x position where the interaction started
     * @param anchorY the y position where the interaction started
     */
    public void invokeEnd(int x, int y, int anchorX, int anchorY,
DrawingView view) {
        invokeEnd(x-anchorX, y-anchorY, view.drawing());
    }

    /**
     * @deprecated As of version 4.1,
     * use invokeEnd(x, y, anchorX, anchorY, drawingView).
     *
     * Tracks the end of the interaction.
     */
    public void invokeEnd (int dx, int dy, Drawing drawing) { }

    /**
     * Gets the handle's owner.
     */
    public Figure owner() {
        return fOwner;
    }
}

```

```

/**
 * Gets the display box of the handle.
 */
public Rectangle displayBox() {
    Point p = locate();
    return new Rectangle(
        p.x - HANDLESIZE / 2,
        p.y - HANDLESIZE / 2,
        HANDLESIZE,
        HANDLESIZE);
}

/**
 * Tests if a point is contained in the handle.
 */
public boolean containsPoint(int x, int y) {
    return displayBox().contains(x, y);
}

/**
 * Draws this handle.
 */
public void draw(Graphics g) {
    Rectangle r = displayBox();

    g.setColor(Color.white);
    g.fillRect(r.x, r.y, r.width, r.height);

    g.setColor(Color.black);
    g.drawRect(r.x, r.y, r.width, r.height);
}
}

```

Como ejemplo de una clase que herede de `AbstractHandle` vamos a ver el código de la clase `PolygonHandle` que es un manejador para un nodo sobre un polígono.

```

/*
 * Fri Feb 28 07:47:13 1997  Doug Lea  (dl at gee)
 * Based on PolyLineHandle
 */

package CH.ifa.draw.contrib;

import java.awt.*;
import java.util.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.standard.*;

/**
 * A handle for a node on the polygon.
 */
public class PolygonHandle extends AbstractHandle {

```

```

private int fIndex;
private Locator fLocator;

/**
 * Constructs a polygon handle.
 * @param owner the owning polygon figure.
 * @l the locator
 * @index the index of the node associated with this handle.
 */
public PolygonHandle(PolygonFigure owner, Locator l, int index) {
    super(owner);
    fLocator = l;
    fIndex = index;
}

public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
    myOwner().setPointAt(new Point(x, y), fIndex);
}

public void invokeEnd (int x, int y, int anchorX, int anchorY,
DrawingView view) {
    myOwner().smoothPoints();
}

public Point locate() {
    return fLocator.locate(owner());
}

private PolygonFigure myOwner() {
    return (PolygonFigure)owner();
}
}

```

Otro ejemplo de una clase que herede de AbstractHandle vamos a ver el código de la clase TriangleRotationHandle que es un Handle para rotar una TriangleFigure.

Fijaros que esta clase utiliza los métodos invokeStart, invokeStep e invokeEnd que estan marcados como deprecated.

```

/*
 * Sun Mar 2 19:15:28 1997 Doug Lea (dl at gee)
 * Based on RadiusHandle
 */

package CH.ifa.draw.contrib;

import java.awt.*;
import java.util.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.standard.*;

```

```

/**
 * A Handle to rotate a TriangleFigure
 */
class TriangleRotationHandle extends AbstractHandle {

    private Point fOrigin = null;
    private Point fCenter = null;

    public TriangleRotationHandle(TriangleFigure owner) {
        super(owner);
    }

    public void invokeStart(int x, int y, Drawing drawing) {
        fCenter = owner().center();
        fOrigin = getOrigin();
    }

    public void invokeStep (int dx, int dy, Drawing drawing) {
        double angle = Math.atan2(fOrigin.y + dy - fCenter.y,
                                   fOrigin.x + dx - fCenter.x);
        ((TriangleFigure)(owner())).rotate(angle);
    }

    public void invokeEnd (int dx, int dy, Drawing drawing) {
        fOrigin = null;
        fCenter = null;
    }

    public Point locate() {
        return getOrigin();
    }

    Point getOrigin() { // find a nice place to put handle
        // almost same code as PolygonScaleHandle
        Polygon p = ((TriangleFigure)(owner())).polygon();
        Point first = new Point(p.xpoints[0], p.ypoints[0]);
        Point ctr = owner().center();
        double len = Geom.length(first.x, first.y, ctr.x, ctr.y);
        if (len == 0) // best we can do?
            return new Point(first.x - HANDLESIZE/2, first.y +
                              HANDLESIZE/2);

        double u = HANDLESIZE / len;
        if (u > 1.0) // best we can do?
            return new Point((first.x * 3 + ctr.x)/4, (first.y * 3 +
ctr.y)/4);
        else
            return new Point((int)(first.x * (1.0 - u) + ctr.x * u),
                              (int)(first.y * (1.0 - u) + ctr.y * u));
    }

    public void draw(Graphics g) {
        Rectangle r = displayBox();

        g.setColor(Color.yellow);
        g.fillOval(r.x, r.y, r.width, r.height);

        g.setColor(Color.black);
    }
}

```

```

        g.drawOval(r.x, r.y, r.width, r.height);
    }
}

```

Para obtener los handles de una Figure tenemos el método abstracto handles del interface Figure que devuelve un Vector de manejadores utilizados para manipular algunos de los atributos de la figura. El método handles es un **Factory Method** para crear objetos handle.

```

/*
 * @(#)Figure.java 5.1
 *
 */

package CH.ifa.draw.framework;

import CH.ifa.draw.util.*;
import java.awt.*;
import java.util.*;
import java.io.Serializable;

/**
 * The interface of a graphical figure. A figure knows
 * its display box and can draw itself. A figure can be
 * composed of several figures. To interact and manipulate
 * with a figure it can provide Handles and Connectors.<p>
 * A figure has a set of handles to manipulate its shape or
 * attributes.
 * A figure has one or more connectors that define how
 * to locate a connection point.<p>
 * Figures can have an open ended set of attributes.
 * An attribute is identified by a string.<p>
 * Default implementations for the Figure interface are provided
 * by AbstractFigure.
 *
 * @see Handle
 * @see Connector
 * @see AbstractFigure
 */

public interface Figure
    extends Storable, Cloneable, Serializable {

    .....

    /**
     * Returns the handles used to manipulate
     * the figure. Handles is a Factory Method for
     * creating handle objects.
     *
     * @return a Vector of handles
     * @see Handle
     */
    public Vector handles();

    .....
}

```

}

Al implementar una nueva figura hay que redefinir el método handles para devolver los objetos que se desea que manipulen la figura.

```

/*
 * @(#)PolyLineFigure.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.util.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.*;

/**
 * A poly line figure consists of a list of points.
 * It has an optional line decoration at the start and end.
 *
 * @see LineDecoration
 */
public class PolyLineFigure extends AbstractFigure {

    public final static int ARROW_TIP_NONE    = 0;
    public final static int ARROW_TIP_START   = 1;
    public final static int ARROW_TIP_END     = 2;
    public final static int ARROW_TIP_BOTH    = 3;

    protected Vector      fPoints;
    protected LineDecoration fStartDecoration = null;
    protected LineDecoration fEndDecoration  = null;
    protected Color        fFrameColor      = Color.black;

    .....

    public PolyLineFigure() {
        fPoints = new Vector(4);
    }

    public PolyLineFigure(int size) {
        fPoints = new Vector(size);
    }

    public PolyLineFigure(int x, int y) {
        fPoints = new Vector();
        fPoints.addElement(new Point(x, y));
    }

    .....

    public Vector handles() {
        Vector handles = new Vector(fPoints.size());
    }

```

```

        for (int i = 0; i < fPoints.size(); i++)
            handles.addElement(new PolyLineHandle(this, locator(i),
i));
        return handles;
    }

    .....
}

```

```

/*
 * @(#)SelectionTool.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.awt.event.MouseEvent;
import java.util.Vector;
import CH.ifa.draw.framework.*;

/**
 * Tool to select and manipulate figures.
 * A selection tool is in one of three states, e.g., background
 * selection, figure selection, handle manipulation. The different
 * states are handled by different child tools.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld032.htm>State</a></b><br>
 * SelectionTool is the StateContext and child is the State.
 * The SelectionTool delegates state specific
 * behavior to its current child tool.
 * <hr>
 */

public class SelectionTool extends AbstractTool {

    private Tool fChild = null;

    public SelectionTool(DrawingView view) {
        super(view);
    }

    /**
     * Handles mouse down events and starts the corresponding tracker.
     */
    public void mouseDown(MouseEvent e, int x, int y)
    {
        // on Windows NT: AWT generates additional mouse down events
        // when the left button is down && right button is clicked.
        // To avoid dead locks we ignore such events
        if (fChild != null)
            return;
    }
}

```

```

        view().freezeView();

        Handle handle = view().findHandle(e.getX(), e.getY());
        if (handle != null) {
            fChild = createHandleTracker(view(), handle);
        }
        else {
            Figure figure = drawing().findFigure(e.getX(), e.getY());
            if (figure != null) {
                fChild = createDragTracker(view(), figure);
            }
            else {
                if (!e.isShiftDown()) {
                    view().clearSelection();
                }
                fChild = createAreaTracker(view());
            }
        }
        fChild.mouseDown(e, x, y);
    }

    /**
     * Handles mouse drag events. The events are forwarded to the
     * current tracker.
     */
    public void mouseDrag(MouseEvent e, int x, int y) {
        if (fChild != null) // JDK1.1 doesn't guarantee mouseDown,
                           // mouseDrag, mouseUp
            fChild.mouseDrag(e, x, y);
    }

    /**
     * Handles mouse up events. The events are forwarded to the
     * current tracker.
     */
    public void mouseUp(MouseEvent e, int x, int y) {
        view().unfreezeView();
        if (fChild != null) // JDK1.1 doesn't guarantee mouseDown,
                           // mouseDrag, mouseUp
            fChild.mouseUp(e, x, y);
        fChild = null;
    }

    /**
     * Factory method to create a Handle tracker. It is used to track
     * a handle.
     */
    protected Tool createHandleTracker(DrawingView view, Handle
handle) {
        return new HandleTracker(view, handle);
    }

    /**
     * Factory method to create a Drag tracker. It is used to drag a
     * figure.
     */
    protected Tool createDragTracker(DrawingView view, Figure f) {
        return new DragTracker(view, f);
    }

```



```

    }

    /**
     * Factory method to create an area tracker. It is used to select
     * an area.
     */
    protected Tool createAreaTracker(DrawingView view) {
        return new SelectAreaTracker(view);
    }
}

```

Un ejemplo de una clase en el papel de *Cliente* del patrón Adapter sería la clase `HandleTracker`. `HandleTracker` implementa interacciones con los manejadores de una figura.

```

/*
 * @(#)HandleTracker.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.awt.event.MouseEvent;
import CH.ifa.draw.framework.*;

/**
 * HandleTracker implements interactions with the handles
 * of a Figure.
 *
 * @see SelectionTool
 */
public class HandleTracker extends AbstractTool {

    private Handle fAnchorHandle;

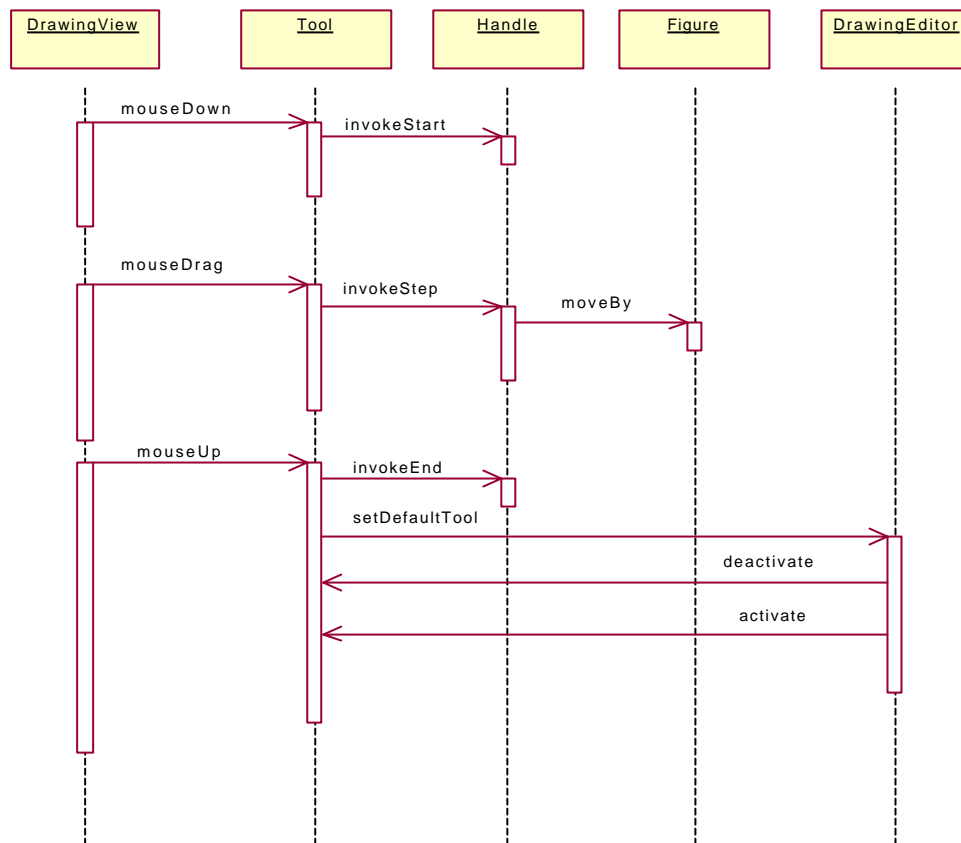
    public HandleTracker(DrawingView view, Handle anchorHandle) {
        super(view);
        fAnchorHandle = anchorHandle;
    }

    public void mouseDown(MouseEvent e, int x, int y) {
        super.mouseDown(e, x, y);
        fAnchorHandle.invokeStart(x, y, view());
    }

    public void mouseDrag(MouseEvent e, int x, int y) {
        super.mouseDrag(e, x, y);
        fAnchorHandle.invokeStep(x, y, fAnchorX, fAnchorY, view());
    }

    public void mouseUp(MouseEvent e, int x, int y) {
        super.mouseDrag(e, x, y);
        fAnchorHandle.invokeEnd(x, y, fAnchorX, fAnchorY, view());
    }
}

```

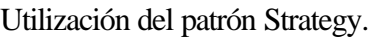


Habría un problema si un handle fuese responsable de manipular la figura y de saber su propia posición. Por cada handle con acción y posición distinta habría una nueva clase. Los handles con la misma acción pero distinta posición también necesitarían distintas clases: NullHandle, ConnectionHandle, GroupHandle, FontSizeHandle, etc. GroupHandle deriva de NullHandle y cambia el dibujo del Handle. ConnectionHandle es el Handle que permite iniciar conexiones como en el ejemplo Pert. En los ejemplos anteriores habría que hacer nueve clases NullHandle y lo mismo con las demas. En el caso de GroupHandle sólo se quiere cambiar el aspecto del NullHandle. Este aspecto habría que hacerlo con nuevas clases, ya que habría nueve NullHandles (una por cada posición).

Para evitar ese problema se han separado las dos responsabilidades del handle:

- ✓ Manipular la figura (invokeStart/invokeStep/invokeEnd).
- ✓ Saber su colocación en la figura.

Por ello se utiliza el patrón **Strategy**. Locator encapsula la estrategia para localizar una manipulación sobre una figura. LocatorHandle realiza el papel de *Cliente* del patrón Strategy. El interface Locator y la clase abstracta AbstractLocator jugarían el papel de *StrategyAbstracto*. Las clases OffsetLocator y RelativeLocator jugarían el papel de *StrategyConcreto*.



```

/*
 * @(#)Locator.java 5.1
 *
 */

package CH.ifa.draw.framework;

import CH.ifa.draw.util.Storable;
import java.awt.*;
import java.io.Serializable;

/**
 * Locators can be used to locate a position on a figure.<p>
 *
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld034.htm>Strategy</a></b><br>
 * Locator encapsulates the strategy to locate a handle on a figure.
 */

public interface Locator extends Storable, Serializable, Cloneable {

    /**
     * Locates a position on the passed figure.
     * @return a point on the figure.
     */
    public Point locate(Figure owner);
}

```

AbstractLocator proporciona una implementación por defecto para el interface Locator.

```

/*
 * @(#)AbstractLocator.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

import java.awt.*;
import java.util.*;
import java.io.IOException;

/**
 * AbstractLocator provides default implementations for
 * the Locator interface.
 *
 * @see Locator
 * @see Handle
 */

public abstract class AbstractLocator
    implements Locator, Storable, Cloneable {

    protected AbstractLocator() {
    }

    public Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
    }

    /**
     * Stores the arrow tip to a StorableOutput.
     */
    public void write(StorableOutput dw) {
    }

    /**
     * Reads the arrow tip from a StorableInput.
     */
    public void read(StorableInput dr) throws IOException {
    }
}

```

OffsetLocator es un locator para desplazar otro Locator.

```

/*
 * @(#)OffsetLocator.java 5.1

```

```

*
*/

package CH.ifa.draw.standard;

import java.awt.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.*;

/**
 * A locator to offset another Locator.
 * @see Locator
 */
public class OffsetLocator extends AbstractLocator {

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = 2679950024611847621L;
    private int offsetLocatorSerializedDataVersion = 1;

    private Locator fBase;
    private int fOffsetX;
    private int fOffsetY;

    public OffsetLocator() {
        fBase = null;
        fOffsetX = 0;
        fOffsetY = 0;
    }

    public OffsetLocator(Locator base) {
        this();
        fBase = base;
    }

    public OffsetLocator(Locator base, int offsetX, int offsetY) {
        this(base);
        fOffsetX = offsetX;
        fOffsetY = offsetY;
    }

    public Point locate(Figure owner) {
        Point p = fBase.locate(owner);
        p.x += fOffsetX;
        p.y += fOffsetY;
        return p;
    }

    public void moveBy(int dx, int dy) {
        fOffsetX += dx;
        fOffsetY += dy;
    }

    public void write(StorableOutput dw) {
        super.write(dw);
        dw.writeInt(fOffsetX);
    }
}

```

```

        dw.writeInt(fOffsetY);
        dw.writeStorable(fBase);
    }

    public void read(StorableInput dr) throws IOException {
        super.read(dr);
        fOffsetX = dr.readInt();
        fOffsetY = dr.readInt();
        fBase = (Locator)dr.readStorable();
    }
}

```

RelativeLocator es un locator que especifica un punto que es relativo a los bordes de una figura. Un RelativeLocator es un objeto que tiene como parámetros dos reales entre 0 y 1 (en el constructor) y una figura (en el método locate). Cuando se le pide su posición dentro de la figura lo que hace es multiplicar el ancho y el alto por los reales, obteniendo una posición relativa dentro de la figura. A esto le suma la x e y de la figura. Por lo tanto un RelativeLocator permite establecer una posición dentro de la figura.

Esta clase también tiene métodos estáticos para generar los RelativeLocators más habituales (norte, sur, esquinas, etc). Estos métodos se limitan a un new cambiando los dos reales en la llamada al constructor. Por ejemplo el método center() devuelve un new RelativeLocator(0.5,0.5).

```

/*
 * @(#)RelativeLocator.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.*;

/**
 * A locator that specifies a point that is relative to the bounds
 * of a figure.
 * @see Locator
 */
public class RelativeLocator extends AbstractLocator {
    /*
     * Serialization support.
     */
    private static final long serialVersionUID = 2619148876087898602L;
    private int relativeLocatorSerializedDataVersion = 1;

    double fRelativeX;
    double fRelativeY;

    public RelativeLocator() {
        fRelativeX = 0.0;
        fRelativeY = 0.0;
    }
}

```

```
}

public RelativeLocator(double relativeX, double relativeY) {
    fRelativeX = relativeX;
    fRelativeY = relativeY;
}

public Point locate(Figure owner) {
    Rectangle r = owner.displayBox();
    return new Point(
        r.x + (int)(r.width*fRelativeX),
        r.y + (int)(r.height*fRelativeY)
    );
}

public void write(StorableOutput dw) {
    super.write(dw);
    dw.writeDouble(fRelativeX);
    dw.writeDouble(fRelativeY);
}

public void read(StorableInput dr) throws IOException {
    super.read(dr);
    fRelativeX = dr.readDouble();
    fRelativeY = dr.readDouble();
}

static public Locator east() {
    return new RelativeLocator(1.0, 0.5);
}

/**
 * North.
 */
static public Locator north() {
    return new RelativeLocator(0.5, 0.0);
}

/**
 * West.
 */
static public Locator west() {
    return new RelativeLocator(0.0, 0.5);
}

/**
 * North east.
 */
static public Locator northEast() {
    return new RelativeLocator(1.0, 0.0);
}

/**
 * North west.
 */
static public Locator northWest() {
    return new RelativeLocator(0.0, 0.0);
}
```

```

/**
 * South.
 */
static public Locator south() {
    return new RelativeLocator(0.5, 1.0);
}

/**
 * South east.
 */
static public Locator southEast() {
    return new RelativeLocator(1.0, 1.0);
}

/**
 * South west.
 */
static public Locator southWest() {
    return new RelativeLocator(0.0, 1.0);
}

/**
 * Center.
 */
static public Locator center() {
    return new RelativeLocator(0.5, 0.5);
}
}

```

Un LocatorHandle implementa un Handle para delegar la localización pedida a un objeto locator.

```

/*
 * @(#)LocatorHandle.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.Point;
import CH.ifa.draw.framework.*;

/**
 * A LocatorHandle implements a Handle by delegating the location
 * requests to
 * a Locator object.
 *
 * @see Locator
 */

public class LocatorHandle extends AbstractHandle {

    private Locator    fLocator;

```



```

/**
 * Initializes the LocatorHandle with the given Locator.
 */
public LocatorHandle(Figure owner, Locator l) {
    super(owner);
    fLocator = l;
}

/**
 * Locates the handle on the figure by forwarding the request
 * to its figure.
 */
public Point locate() {
    return fLocator.locate(owner());
}
}

```

El Handle delega en el Locator la responsabilidad de saber su posición. Cada Locator representa a una posición de la figura.

Todos los Handle que tengan la misma posición (incluso en distintas figuras) podrán compartir el mismo Locator.

```

public class CenterLocatorExample
{
    public Point locate (Figure owner)
    {
        Rectangle r=owner.displayBox();
        return new Point(r.x+r.width/2, r.y+r.height/2);
    }
}

```

Un ejemplo de una clase que utiliza la clase RelativeLocator sería la clase PertFigure.

```

/*
 * @(#)PertFigure.java 5.1
 *
 */

package CH.ifa.draw.samples.pert;

import java.awt.*;
import java.util.*;
import java.io.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.util.*;

public class PertFigure extends CompositeFigure {
    private static final int BORDER = 3;
    private Rectangle fDisplayBox;
    private Vector fPreTasks;
    private Vector fPostTasks;
}

```

```

.....

    public Vector handles() {
        Vector handles = new Vector();
        handles.addElement(new NullHandle(this,
RelativeLocator.northWest()));
        handles.addElement(new NullHandle(this,
RelativeLocator.northEast()));
        handles.addElement(new NullHandle(this,
RelativeLocator.southWest()));
        handles.addElement(new NullHandle(this,
RelativeLocator.southEast()));
        handles.addElement(new ConnectionHandle(this,
RelativeLocator.east(),
                                new PertDependency())
                                );
        return handles;
    }

.....
}

```

Los Handles de redimensionado si tienen distintas acciones en distintas posiciones entonces ha de haber una clase por Handle, pero no hay que rehacer esas clases para cada nueva figura: reutilización.

La clase BoxHandleKit es un conjunto de métodos de utilidad para crear Handles para las localizaciones comunes sobre una representación visual (display box) de una figura. Tiene métodos estáticos para poner un handle en una posición (north, south, etc), métodos para poner de una vez las cuatro esquinas (addCornerHandles) y métodos para poner esquinas y puntos cardinales (addHandles).

Esta clase tiene una clase por cada handle (8 en total). Son LocatorHandle que ponen en su constructor el RelativeLocator adecuado y que además implementan un invokeStep adecuado para modificar el displayBox de la figura.

```

/*
 * @(#)BoxHandleKit.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.framework.*;
import java.awt.*;
import java.util.Vector;

/**
 * A set of utility methods to create Handles for the common
 * locations on a figure's display box.
 * @see Handle
 */

```

```
// TBD: use anonymous inner classes (had some problems with JDK 1.1)

public class BoxHandleKit {

    /**
     * Fills the given Vector with handles at each corner of a
     * figure.
     */
    static public void addCornerHandles(Figure f, Vector handles) {
        handles.addElement(southEast(f));
        handles.addElement(southWest(f));
        handles.addElement(northEast(f));
        handles.addElement(northWest(f));
    }

    /**
     * Fills the given Vector with handles at each corner
     * and the north, south, east, and west of the figure.
     */
    static public void addHandles(Figure f, Vector handles) {
        addCornerHandles(f, handles);
        handles.addElement(south(f));
        handles.addElement(north(f));
        handles.addElement(east(f));
        handles.addElement(west(f));
    }

    static public Handle south(Figure owner) {
        return new SouthHandle(owner);
    }

    static public Handle southEast(Figure owner) {
        return new SouthEastHandle(owner);
    }

    static public Handle southWest(Figure owner) {
        return new SouthWestHandle(owner);
    }

    static public Handle north(Figure owner) {
        return new NorthHandle(owner);
    }

    static public Handle northEast(Figure owner) {
        return new NorthEastHandle(owner);
    }

    static public Handle northWest(Figure owner) {
        return new NorthWestHandle(owner);
    }

    static public Handle east(Figure owner) {
        return new EastHandle(owner);
    }

    static public Handle west(Figure owner) {
        return new WestHandle(owner);
    }
}
```

```

}

class NorthEastHandle extends LocatorHandle {
    NorthEastHandle(Figure owner) {
        super(owner, RelativeLocator.northEast());
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        Rectangle r = owner().displayBox();
        owner().displayBox(
            new Point(r.x, Math.min(r.y + r.height, y)),
            new Point(Math.max(r.x, x), r.y + r.height)
        );
    }
}

class EastHandle extends LocatorHandle {
    EastHandle(Figure owner) {
        super(owner, RelativeLocator.east());
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        Rectangle r = owner().displayBox();
        owner().displayBox(
            new Point(r.x, r.y), new Point(Math.max(r.x, x), r.y +
r.height)
        );
    }
}

class NorthHandle extends LocatorHandle {
    NorthHandle(Figure owner) {
        super(owner, RelativeLocator.north());
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        Rectangle r = owner().displayBox();
        owner().displayBox(
            new Point(r.x, Math.min(r.y + r.height, y)),
            new Point(r.x + r.width, r.y + r.height)
        );
    }
}

class NorthWestHandle extends LocatorHandle {
    NorthWestHandle(Figure owner) {
        super(owner, RelativeLocator.northWest());
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        Rectangle r = owner().displayBox();
        owner().displayBox(
            new Point(Math.min(r.x + r.width, x), Math.min(r.y +
r.height, y)),

```

```

        new Point(r.x + r.width, r.y + r.height)
    );
}
}

class SouthEastHandle extends LocatorHandle {
    SouthEastHandle(Figure owner) {
        super(owner, RelativeLocator.southEast());
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        Rectangle r = owner().displayBox();
        owner().displayBox(
            new Point(r.x, r.y),
            new Point(Math.max(r.x, x), Math.max(r.y, y))
        );
    }
}

class SouthHandle extends LocatorHandle {
    SouthHandle(Figure owner) {
        super(owner, RelativeLocator.south());
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        Rectangle r = owner().displayBox();
        owner().displayBox(
            new Point(r.x, r.y),
            new Point(r.x + r.width, Math.max(r.y, y))
        );
    }
}

class SouthWestHandle extends LocatorHandle {
    SouthWestHandle(Figure owner) {
        super(owner, RelativeLocator.southWest());
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        Rectangle r = owner().displayBox();
        owner().displayBox(
            new Point(Math.min(r.x + r.width, x), r.y),
            new Point(r.x + r.width, Math.max(r.y, y))
        );
    }
}

class WestHandle extends LocatorHandle {
    WestHandle(Figure owner) {
        super(owner, RelativeLocator.west());
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        Rectangle r = owner().displayBox();

```

```

        owner().displayBox(
            new Point(Math.min(r.x + r.width, x), r.y),
            new Point(r.x + r.width, r.y + r.height)
        );
    }
}

```

Un ejemplo de utilización de la clase BoxHandleKit es la clase RoundedRectangleFigure, en la que se puede ver que con una sola línea (BoxHandleKit) pone todos los handles y se olvida de ellos. Ya se encargan de todo ellos solos.

```

/*
 * @(#)RoundRectangleFigure.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.io.IOException;
import java.util.Vector;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.*;

/**
 * A round rectangle figure.
 * @see RadiusHandle
 */
public class RoundRectangleFigure extends AttributeFigure {

    .....

    public Vector handles() {
        Vector handles = new Vector();
        BoxHandleKit.addHandles(this, handles);

        handles.addElement(new RadiusHandle(this));

        return handles;
    }

    .....
}

```

Extensión de JHotDraw: incorporación de Handles a una nueva figura

Si los RelativeLocator permitían poner un handle en cualquier lugar (reutilizar posicionamiento de los handles), los del BoxHandleKit permiten manipular cualquier figura a través de su displayBox (reutilización de acción). De esta manera, al implementar un nuevo handle se plantea lo siguiente:

- ♦ Al añadir una nueva figura hay que redefinir el metodo handles.
- ♦ Si se desean los Handles de redimensionado utilizar la clase BoxHandleKit.
- ♦ Si la acción y la localización del nuevo Handle ya la contempla alguna clase entonces combinarlas.

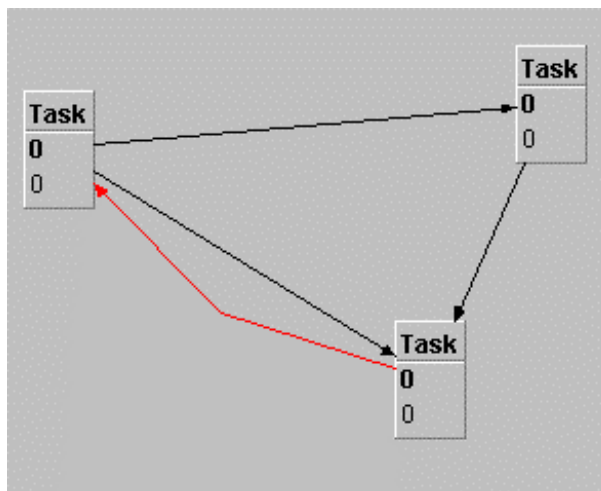
```
handles.addElement(new NullHandle(this, RelativeLocator.northWest()));
```

- ♦ Si la acción es específica de la figura redefinir LocatorHandle.
- ♦ Si la localización es específica de la figura redefinir AbstractHandle.

3.5.3.5. Conexión de figuras

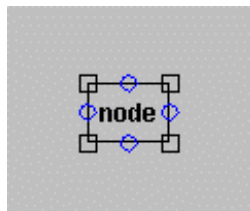
Los requisitos que se han de cumplir en la conexión de figuras son:

- Independencia de la implementación de las figuras.
- Control de conexiones inválidas.

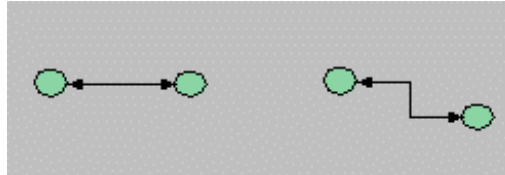


Relación circular en rojo.

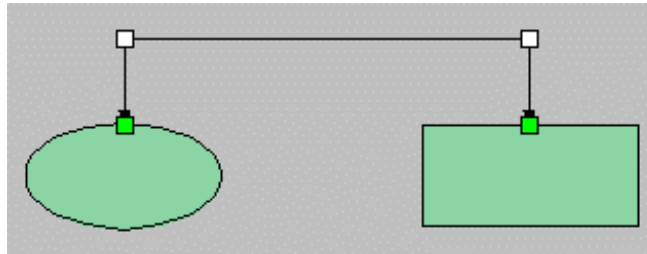
- Conexión en cualquier punto de la figura o en puntos concretos.



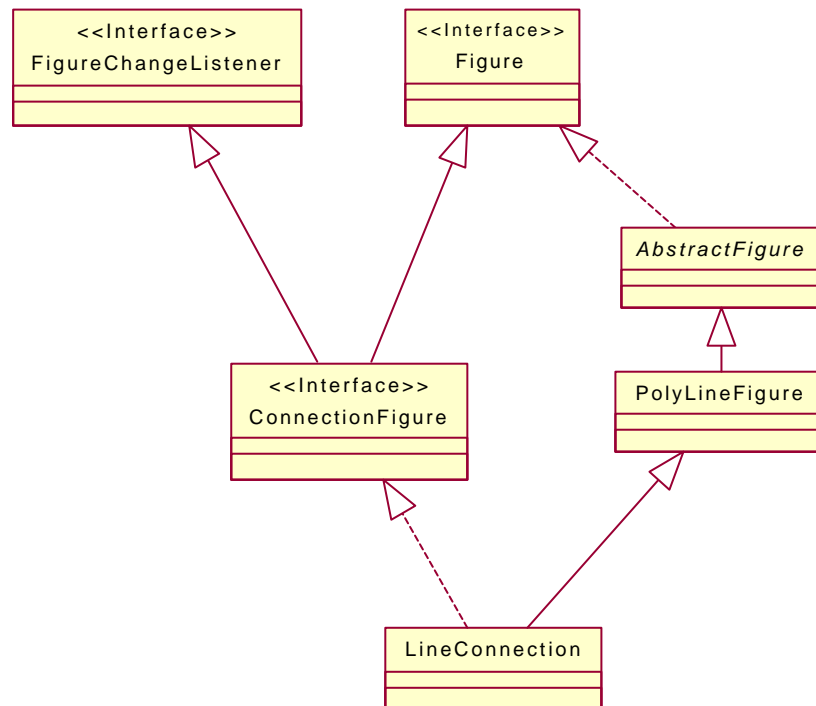
- Reacción ante los cambios en las figuras conectadas.
- Distintos tipos de conexiones (recta, segmentos, arcos,...).



- Las conexiones deben poder manipularse (handles).



Una ConnectionFigure es una Figure que relaciona dos figuras. Una ConnectionFigure puede tener varios segmentos.



Jerarquía ConnectionFigure.

Las Figures que conectan Connectors proporcionan Figures. Un ConnectionFigure conoce su Connector del principio y del final. Utiliza los Connectors para localizar sus puntos de conexión.

El código del interface ConnectionFigure es el siguiente:

```

/*
 * @(#)ConnectionFigure.java 5.1
 */

```



```

*/

package CH.ifa.draw.framework;

import java.awt.Point;
import java.io.Serializable;
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

/**
 * Figures to connect Connectors provided by Figures.
 * A ConnectionFigure knows its start and end Connector.
 * It uses the Connectors to locate its connection points.<p>
 * A ConnectionFigure can have multiple segments. It provides
 * operations to split and join segments.
 *
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld034.htm>Strategy</a></b><br>
 * Strategy is used encapsulate the algorithm to locate the connection
 point.
 * ConnectionFigure is the Strategy context and Connector is the
 Strategy.<br>
 * 
 * <b><a href=../pattlets/sld026.htm>Observer</a></b><br>
 * Observer is used to track changes of the connected figures. A
 connection
 * figure registers itself as listeners or observers of the source and
 * target connector.
 * <hr>
 *
 * @see Connector
 */

public interface ConnectionFigure extends Figure, FigureChangeListener
{

    /**
     * Sets the start Connector of the connection.
     * @param figure the start figure of the connection
     */
    public void connectStart(Connector start);

    /**
     * Sets the end Connector of the connection.
     * @param figure the end figure of the connection
     */
    public void connectEnd(Connector end);

    /**
     * Updates the connection
     */
    public void updateConnection();

    /**
     * Disconnects the start figure from the dependent figure
     */

```

```

public void disconnectStart();

/**
 * Disconnects the end figure from the dependent figure
 */
public void disconnectEnd();

/**
 * Gets the start Connector
 */
public Connector start();

/**
 * Gets the end Connector.
 */
public Connector end();

/**
 * Checks if two figures can be connected. Implement this method
 * to constrain the allowed connections between figures.
 */
public boolean canConnect(Figure start, Figure end);

/**
 * Checks if the ConnectionFigure connects the same figures.
 */
public boolean connectsSame(ConnectionFigure other);

/**
 * Sets the start point.
 */
public void startPoint(int x, int y);

/**
 * Sets the end point.
 */
public void endPoint(int x, int y);

/**
 * Gets the start point.
 */
public Point startPoint();

/**
 * Gets the end point.
 */
public Point endPoint();

/**
 * Sets the position of the point at the given position
 */
public void setPointAt(Point p, int index);

/**
 * Gets the Point at the given position
 */
public Point pointAt(int index);

```

```
/**
 * Gets the number of points or nodes of the connection
 */
public int pointCount();

/**
 * Splits the hit segment.
 * @param x, y the position where the figure should be split
 * @return the index of the splitting point
 */
public int splitSegment(int x, int y);

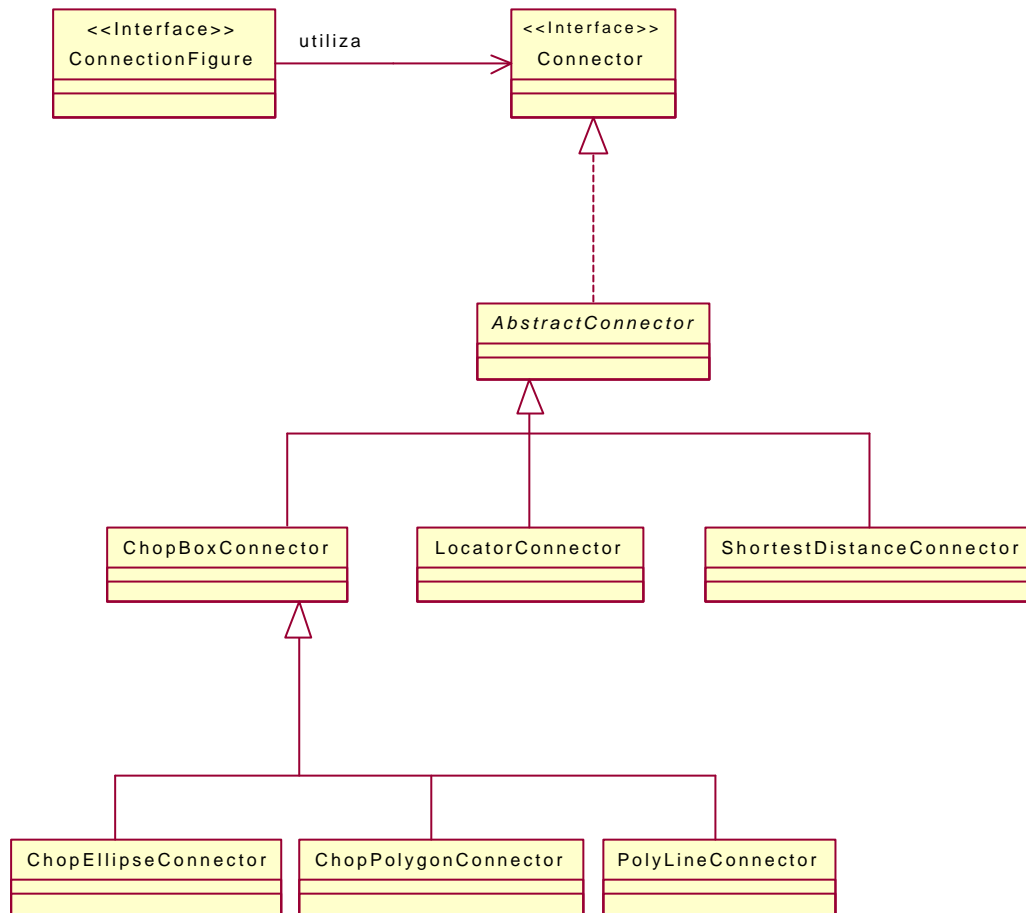
/**
 * Joins the hit segments.
 * @param x, y the position where the figure should be joined.
 * @return whether the segment was joined
 */
public boolean joinSegments(int x, int y);
}
```

Al crear una conexión entre dos figuras podemos tener diferentes algoritmos para cacular donde colocar las coordenadas de sus extremos.

- El centro de cada figura.
- La mínima distancia entre sus áreas.
- Los puntos de corte de la recta que una los centros y el displayBox.
- Los puntos cardinales de la figura.
-

El posicionamiento es relativo por lo tanto al mover una de las figuras de la conexión no se pierde la conexión pero se tiene que ver cuales son los nuevos puntos de conexión o si son los mismos que antes.

Para solucionar este problema se utiliza el patrón **Strategy**. Este patrón es utilizado para encapsular el algoritmo para localizar el punto de conexión. *ConnectionFigure* juega el papel de *Cliente* del patrón *Strategy*. El interface *Connector* y la clase abstracta *AbstractConnectores* juegan el papel de *StrategyAbstracto*. Y *ChopBoxConnector*, *LocatorConnector* y *ShortestDistanceConnector* juegan el papel de *StrategyConcreto*.



Un Conector es un punto de conexión de la figura. Las `ConnectionFigure` están ancladas a un Conector en cada uno de sus extremos. A la línea se le pasa un Conector para cada extremo. Un Conector sabe cuáles son las coordenadas en las cuales debe realizarse la conexión con la figura, además de cómo dibujarse y su área de detección.

Los Connectors saben cómo posicionar un punto de conexión sobre una figura. Un Connector conoce su propia figura y puede determinar el punto de inicio y el punto final de una figura de conexión dada. Un conector tiene una zona de representación (display box) que describe el área de una figura de la cual es responsable. Un conector puede estar visible pero puede no estarlo.

La línea escucha los eventos de `Change` de las dos figuras que conecta. Si alguna de ellas se mueve entonces le pide a su Connector la nueva posición y se repinta.

Connector implementa la estrategia para determinar los puntos de conexión.

Factory Method Los Connectors son creados por el factory method de la `Figure` `connectorAt`.

```

/*
 * @(#)Connector.java 5.1

```

```

*
* /

package CH.ifa.draw.framework;

import java.awt.*;
import java.io.Serializable;

import CH.ifa.draw.util.*;

/**
 * Connectors know how to locate a connection point on a figure.
 * A Connector knows its owning figure and can determine either
 * the start or the endpoint of a given connection figure. A connector
 * has a display box that describes the area of a figure it is
 * responsible for. A connector can be visible but it doesn't have
 * to be.<br>
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld004.htm>Strategy</a></b><br>
 * Connector implements the strategy to determine the connections
 * points.<br>
 * 
 * <b><a href=../pattlets/sld016.htm>Factory Method</a></b><br>
 * Connectors are created by the Figure's factory method connectorAt.
 * <hr>
 *
 * @see Figure#connectorAt
 * @see ConnectionFigure
 */
public interface Connector extends Serializable, Storable {

    /**
     * Finds the start point for the connection.
     */
    public abstract Point findStart(ConnectionFigure connection);

    /**
     * Finds the end point for the connection.
     */
    public abstract Point findEnd(ConnectionFigure connection);

    /**
     * Gets the connector's owner.
     */
    public abstract Figure owner();

    /**
     * Gets the display box of the connector.
     */
    public abstract Rectangle displayBox();

    /**
     * Tests if a point is contained in the connector.
     */
    public abstract boolean containsPoint(int x, int y);

```

```

/**
 * Draws this connector. Connectors don't have to be visible
 * and it is OK leave this method empty.
 */
public abstract void draw(Graphics g);
}

```

En la ConnectionFigure en vez de dar coordenadas absolutas (startPoint, endPoint) es más flexible dar un Conector a cada extremo.

```

/*
 * @(#)ConnectionFigure.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.Point;
import java.io.Serializable;
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

public interface ConnectionFigure extends Figure, FigureChangeListener
{
    .....

    /**
     * Sets the start Connector of the connection.
     * @param figure the start figure of the connection
     */
    public void connectStart(Connector start);

    /**
     * Sets the end Connector of the connection.
     * @param figure the end figure of the connection
     */
    public void connectEnd(Connector end);

    .....

    /**
     * Gets the start Connector
     */
    public Connector start();

    /**
     * Gets the end Connector.
     */
    public Connector end();

    .....
}

```

El interface Figure es el que sabe en que puntos puede realizarse una conexión.

Una figura tiene uno o más conectores que definen como localizar y posicionar un punto de conexión.

```

/*
 * @(#)Figure.java 5.1
 *
 */

package CH.ifa.draw.framework;

import CH.ifa.draw.util.*;
import java.awt.*;
import java.util.*;
import java.io.Serializable;

public interface Figure
    extends Storable, Cloneable, Serializable {

    .....

    /**
     * Checks if this figure can be connected
     */
    public boolean canConnect();

    /**
     * Gets a connector for this figure at the given location.
     * A figure can have different connectors at different locations.
     */
    public Connector connectorAt(int x, int y);

    /**
     * Sets whether the connectors should be visible.
     * Connectors can be optionally visible. Implement
     * this method and react on isVisible to turn the
     * connectors on or off.
     */
    public void connectorVisibility(boolean isVisible);

    /**
     * Returns the connection inset. This is only a hint that
     * connectors can use to determine the connection location.
     * The inset defines the area where the display box of a
     * figure should not be connected.
     */
    public Insets connectionInsets();

    /**
     * Returns the locator used to located connected text.
     */
    public Locator connectedTextLocator(Figure text);
}

```

Un ejemplo de su utilización sería este:

```

/*
 * @(#)AbstractFigure.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

import java.awt.*;
import java.util.*;
import java.io.*;

public abstract class AbstractFigure implements Figure {
    .....

    /**
     * Checks if this figure can be connected. By default
     * AbstractFigures can be connected.
     */
    public boolean canConnect() {
        return true;
    }

    /**
     * Returns the connection inset. The connection inset
     * defines the area where the display box of a
     * figure can't be connected. By default the entire
     * display box can be connected.
     */
    public Insets connectionInsets() {
        return new Insets(0, 0, 0, 0);
    }

    /**
     * Returns the Figures connector for the specified location.
     * By default a ChopBoxConnector is returned.
     * @see ChopBoxConnector
     */
    public Connector connectorAt(int x, int y) {
        return new ChopBoxConnector(this);
    }

    /**
     * Sets whether the connectors should be visible.
     * By default they are not visible and
     */
    public void connectorVisibility(boolean isVisible) {
    }

    /**

```



```

    * Returns the locator used to located connected text.
    */
    public Locator connectedTextLocator(Figure text) {
        return RelativeLocator.center();
    }

    .....
}

```

Un ChopBoxConnector localiza puntos de conexión para realizar la conexión entre los centros de las dos figuras de la zona de representación.

```

/*
 * @(#)ChopBoxConnector.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.Geom;

/**
 * A ChopBoxConnector locates connection points by
 * chopping the connection between the centers of the
 * two figures at the display box.
 * @see Connector
 */
public class ChopBoxConnector extends AbstractConnector {

    /*
     * Serialization support.
     */
    private static final long serialVersionUID = -
1461450322712345462L;

    public ChopBoxConnector() { // only used for Storable
implementation
    }

    public ChopBoxConnector(Figure owner) {
        super(owner);
    }

    public Point findStart(ConnectionFigure connection) {
        Figure startFigure = connection.start().owner();
        Rectangle r2 = connection.end().displayBox();
        Point r2c = null;

        if (connection.pointCount() == 2)
            r2c = new Point(r2.x + r2.width/2, r2.y + r2.height/2);
        else
            r2c = connection.pointAt(1);
    }
}

```

```

        return chop(startFigure, r2c);
    }

    public Point findEnd(ConnectionFigure connection) {
        Figure endFigure = connection.end().owner();
        Rectangle r1 = connection.start().displayBox();
        Point r1c = null;

        if (connection.pointCount() == 2)
            r1c = new Point(r1.x + r1.width/2, r1.y + r1.height/2);
        else
            r1c = connection.pointAt(connection.pointCount()-2);

        return chop(endFigure, r1c);
    }

    protected Point chop(Figure target, Point from) {
        Rectangle r = target.displayBox();
        return Geom.angleToPoint(r, (Geom.pointToAngle(r, from)));
    }
}

```

Un Connector debe saber su posición, tamaño y dibujo.

Un LocatorConnector independiza la posición del conector (delega en connector) de las otras responsabilidades (dibujo y tamaño).

Un LocatorConnector localiza puntos de conexión con la ayuda de un Locator. Soporta la definición de puntos de conexión para localizaciones semánticas.

```

/*
 * @(#)LocatorConnector.java 5.1
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.*;

/**
 * A LocatorConnector locates connection points with
 * the help of a Locator. It supports the definition
 * of connection points to semantic locations.
 * @see Locator
 * @see Connector
 */
public class LocatorConnector extends AbstractConnector {

    /**
     * The standard size of the connector. The display box
     * is centered around the located point.
     */
    public static final int SIZE = 8;
}

```

```

private Locator fLocator;

/*
 * Serialization support.
 */
private static final long serialVersionUID = 5062833203337604181L;
private int locatorConnectorSerializedDataVersion = 1;

public LocatorConnector() { // only used for Storable
    fLocator = null;
}

public LocatorConnector(Figure owner, Locator l) {
    super(owner);
    fLocator = l;
}

protected Point locate(ConnectionFigure connection) {
    return fLocator.locate(owner());
}

/**
 * Tests if a point is contained in the connector.
 */
public boolean containsPoint(int x, int y) {
    return displayBox().contains(x, y);
}

/**
 * Gets the display box of the connector.
 */
public Rectangle displayBox() {
    Point p = fLocator.locate(owner());
    return new Rectangle(
        p.x - SIZE / 2,
        p.y - SIZE / 2,
        SIZE,
        SIZE);
}

/**
 * Draws this connector.
 */
public void draw(Graphics g) {
    Rectangle r = displayBox();

    g.setColor(Color.blue);
    g.fillOval(r.x, r.y, r.width, r.height);
    g.setColor(Color.black);
    g.drawOval(r.x, r.y, r.width, r.height);
}

/**
 * Stores the arrow tip to a StorableOutput.
 */
public void write(StorableOutput dw) {
    super.write(dw);
}

```

```

        dw.writeStorable(fLocator);
    }

    /**
     * Reads the arrow tip from a StorableInput.
     */
    public void read(StorableInput dr) throws IOException {
        super.read(dr);
        fLocator = (Locator)dr.readStorable();
    }
}

```

Como ejemplo de utilización de LocatorConnector se muestra NodeFigure.

```

/*
 * @(#)NodeFigure.java 5.1
 *
 */

package CH.ifa.draw.samples.net;

import java.awt.*;
import java.util.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.util.*;

public class NodeFigure extends TextFigure {

    private static final int BORDER = 6;
    private Vector          fConnectors;
    private boolean         fConnectorsVisible;

    public NodeFigure() {
        initialize();
        fConnectors = null;
    }

    public Rectangle displayBox() {
        Rectangle box = super.displayBox();
        int d = BORDER;
        box.grow(d, d);
        return box;
    }

    public boolean containsPoint(int x, int y) {
        // add slop for connectors
        if (fConnectorsVisible) {
            Rectangle r = displayBox();
            int d = LocatorConnector.SIZE/2;
            r.grow(d, d);
            return r.contains(x, y);
        }
    }
}

```

```

        return super.containsPoint(x, y);
    }

    private void drawBorder(Graphics g) {
        Rectangle r = displayBox();
        g.setColor(getFrameColor());
        g.drawRect(r.x, r.y, r.width-1, r.height-1);
    }

    public void draw(Graphics g) {
        super.draw(g);
        drawBorder(g);
        drawConnectors(g);
    }

    public Vector handles() {
        ConnectionFigure prototype = new LineConnection();
        Vector handles = new Vector();
        handles.addElement(new ConnectionHandle(this,
RelativeLocator.east(), prototype));
        handles.addElement(new ConnectionHandle(this,
RelativeLocator.west(), prototype));
        handles.addElement(new ConnectionHandle(this,
RelativeLocator.south(), prototype));
        handles.addElement(new ConnectionHandle(this,
RelativeLocator.north(), prototype));

        handles.addElement(new NullHandle(this,
RelativeLocator.southEast()));
        handles.addElement(new NullHandle(this,
RelativeLocator.southWest()));
        handles.addElement(new NullHandle(this,
RelativeLocator.northEast()));
        handles.addElement(new NullHandle(this,
RelativeLocator.northWest()));
        return handles;
    }

    private void drawConnectors(Graphics g) {
        if (fConnectorsVisible) {
            Enumeration e = connectors().elements();
            while (e.hasMoreElements())
                ((Connector) e.nextElement()).draw(g);
        }
    }

    /**
     */
    public void connectorVisibility(boolean isVisible) {
        fConnectorsVisible = isVisible;
        invalidate();
    }

    /**
     */
    public Connector connectorAt(int x, int y) {
        return findConnector(x, y);
    }

```

```

/**
 */
private Vector connectors() {
    if (fConnectors == null)
        createConnectors();
    return fConnectors;
}

private void createConnectors() {
    fConnectors = new Vector(4);
    fConnectors.addElement(new LocatorConnector(this,
RelativeLocator.north() ));
    fConnectors.addElement(new LocatorConnector(this,
RelativeLocator.south() ));
    fConnectors.addElement(new LocatorConnector(this,
RelativeLocator.west() ));
    fConnectors.addElement(new LocatorConnector(this,
RelativeLocator.east() ));
}

private Connector findConnector(int x, int y) {
    // return closest connector
    long min = Long.MAX_VALUE;
    Connector closest = null;
    Enumeration e = connectors().elements();
    while (e.hasMoreElements()) {
        Connector c = (Connector)e.nextElement();
        Point p2 = Geom.center(c.displayBox());
        long d = Geom.length2(x, y, p2.x, p2.y);
        if (d < min) {
            min = d;
            closest = c;
        }
    }
    return closest;
}

private void initialize() {
    setText("node");
    Font fb = new Font("Helvetica", Font.BOLD, 12);
    setFont(fb);
    createConnectors();
}
}

```

Resumen

Al implementar una nueva figura

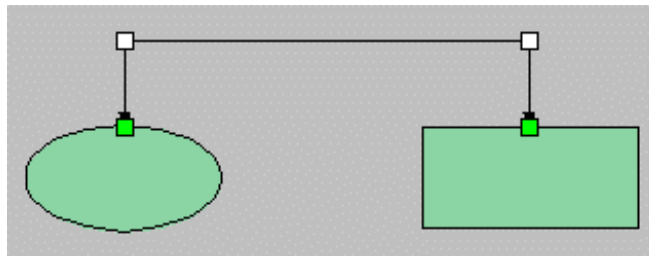
- ◆ Si no se redefine connectorAt se obtiene la conexión estándar mediante un ChopBoxConnector (AbstractFigure).
- ◆ Si se desean indicar puntos de conexión fijos se debe redefinir connectorAt y devolver el conector adecuado (NodeFigure).

- ◆ Si se desea cambiar la apariencia de los conectores se deberá derivar de `LocatorConnector` y redefinir `draw`.
- ◆ Si se desea cambiar además el posicionamiento habrá que derivar del interface `Locator`.

Al mover una de las dos figuras que participan en la conexión se actualiza la `ConnectionFigure`, para ello se utiliza el patrón de diseño **Observer**. El `Observer` es utilizado para ver los cambios de las figuras conectadas. Una `ConnectionFigure` se registra ella misma como escuchadores (listeners) o observadores (observers) de los conectores fuente y destino.

Las implementaciones estándar de JHotDraw relacionadas con la conexión de figuras son:

- `LineConnection`: Es la figura de conexión más habitual. Se implementa mediante una `PolyLineFigure`. Permite indicar los adornos de extremos de la conexión.



`LineDecoration` decora el punto del principio y del final de una line o poly line figure. `LineDecoration` es la clase base para las diferentes decoraciones de la línea.

```

/*
 * @(#)LineDecoration.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.io.Serializable;

import CH.ifa.draw.util.Storable;

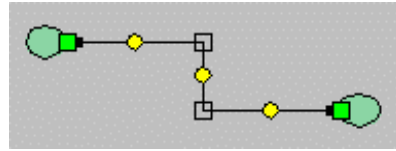
/**
 * Decorate the start or end point of a line or poly line figure.
 * LineDecoration is the base class for the different line
 * decorations.
 * @see PolyLineFigure
 */
public interface LineDecoration
    extends Storable, Cloneable, Serializable {

    /**
     * Draws the decoration in the direction specified by the two
     * points.
     */
}

```

```
public abstract void draw(Graphics g, int x1, int y1, int x2, int y2);
}
```

- ❑ ElbowConnection: Implementa una conexión entre figuras (ConnectionFigure) formada a base de segmentos verticales y horizontales.



- ❑ ConnectionTool: Es la herramienta que permite crear las conexiones. Vale para cualquier ConnectionFigure. No es necesario crear una herramienta distinta para cada tipo de conexión.

```
tool=new ConnectionTool(view(), new LineConnection());
tool=new ConnectionTool(view(), new ElbowConnection());
```

Resumen. Creación de nuevos tipos de conexiones

- ◆ Si simplemente se desean conectar figuras sin ningún tipo de semántica utilizar CreationTool con el tipo de conexión deseada (LineConnection, ElbowConnection, ...).
- ◆ Si se desea algún tipo de acción asociada a la conexión (validación, conexión, desconexión, etc.) derivar del tipo de conexión deseado y redefinir los métodos adecuados.

```
/*
 * @(#)PertDependency.java 5.1
 *
 */

package CH.ifa.draw.samples.pert;

import java.awt.*;
import java.util.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.standard.*;

public class PertDependency extends LineConnection {
    /*
     * Serialization support.
     */
    private static final long serialVersionUID = -
7959500008698525009L;
    private int pertDependencySerializedDataVersion = 1;
```



```

public PertDependency() {
    setEndDecoration(new ArrowTip());
    setStartDecoration(null);
}

public void handleConnect(Figure start, Figure end) {
    PertFigure source = (PertFigure)start;
    PertFigure target = (PertFigure)end;
    if (source.hasCycle(target)) {
        setAttribute("FrameColor", Color.red);
    } else {
        target.addPreTask(source);
        source.addPostTask(target);
        source.notifyPostTasks();
    }
}

public void handleDisconnect(Figure start, Figure end) {
    PertFigure source = (PertFigure)start;
    PertFigure target = (PertFigure)end;
    if (target != null) {
        target.removePreTask(source);
        target.updateDurations();
    }
    if (source != null)
        source.removePostTask(target);
}

public boolean canConnect(Figure start, Figure end) {
    return (start instanceof PertFigure && end instanceof
PertFigure);
}

public Vector handles() {
    Vector handles = super.handles();
    // don't allow to reconnect the starting figure
    handles.setElementAt(
        new NullHandle(this, PolyLineFigure.locator(0)), 0);
    return handles;
}
}

```

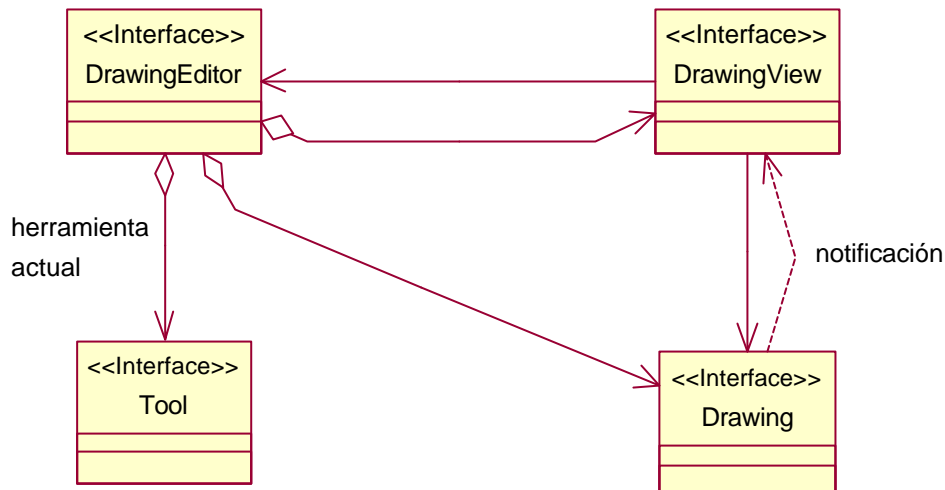
- ◆ Si se desea crear un nuevo tipo de conexión (arcos) se deberá implementar partiendo de ConnectionFigure. En este caso se podrá seguir usando ConnectionTool si la nueva conexión se crea como las habituales (pinchar, arrastrar y conectar). En caso contrario habrá que crear su propia ConnectionTool (derivando de AbstractTool).

3.5.3.6. Entorno de ejecución

Un editor JHotDraw consiste en DrawingView, Drawing, Tools, Menus, Botones (paletas).

Se desea tener un coordinador de todos los objetos citados anteriormente y este coordinador debe valer tanto para aplicaciones como para applets.

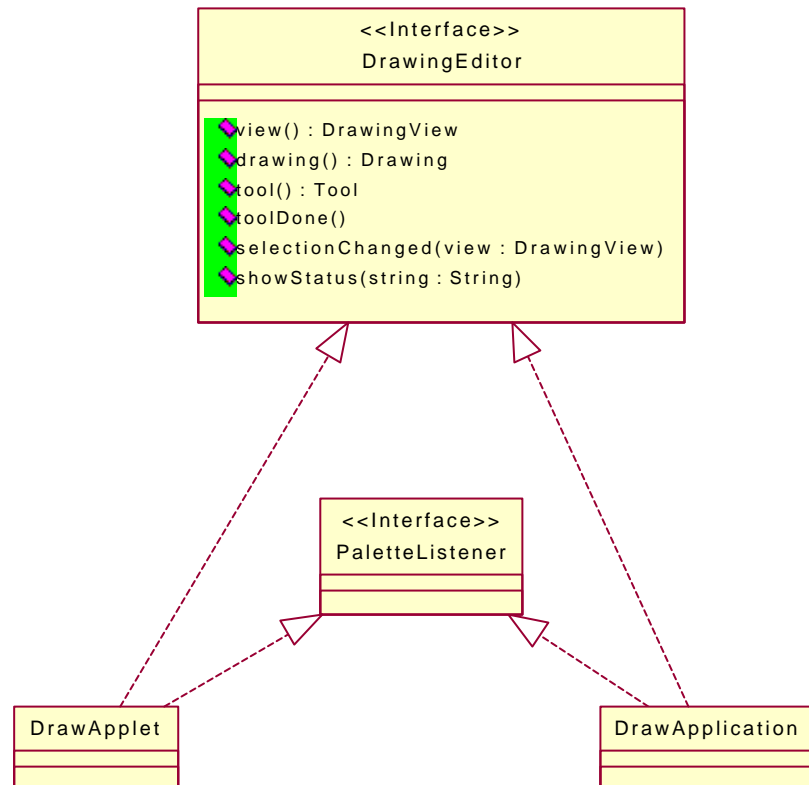
Para ello se utiliza el patrón de diseño **Mediator**. El interface DrawingEditor junto con sus implementaciones por defecto DrawApplet y DrawApplication juegan el papel de *Mediator* en el patrón Mediator. Desacopla los participantes de un editor de dibujo. DrawingView, Drawing y Tool juegan el papel de *Colega*.



Utilización del patrón Mediator.

DrawingEditor define el interface para coordinar los diferentes objetos que participan en un editor de dibujo.

El editor se basa en DrawingEditor independientemente de si el programa principal es una aplicación o un applet.



Jerarquía DrawingEditor.

```

/*
 * @(#)DrawingEditor.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.*;

/**
 * DrawingEditor defines the interface for coordinating
 * the different objects that participate in a drawing editor.
 *
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld022.htm>Mediator</a></b><br>
 * DrawingEditor is the mediator. It decouples the participants
 * of a drawing editor.
 *
 * @see Tool
 * @see DrawingView
 * @see Drawing
 */
public interface DrawingEditor {

    /**
     * Gets the editor's drawing view.
     */

```

```

    */
    DrawingView view();

    /**
     * Gets the editor's drawing.
     */
    Drawing    drawing();

    /**
     * Gets the editor's current tool.
     */
    Tool       tool();

    /**
     * Informs the editor that a tool has done its interaction.
     * This method can be used to switch back to the default tool.
     */
    void       toolDone();

    /**
     * Informs that the current selection has changed.
     * Override this method to handle selection changes.
     */
    void       selectionChanged(DrawingView view);

    /**
     * Shows a status message in the editor's user interface
     */
    void       showStatus(String string);
}

```

El interface PaletteListener es para manejar eventos sobre la paleta.

```

/*
 * @(#)PaletteListener.java 5.1
 *
 */

package CH.ifa.draw.util;

/**
 * Interface for handling palette events.
 *
 * @see PaletteButton
 */

public interface PaletteListener {
    /**
     * The user selected a palette entry. The selected button is
     * passed as an argument.
     */
    void paletteUserSelected(PaletteButton button);

    /**
     * The user moved the mouse over the palette entry.
     */
}

```

```
void paletteUserOver(PaletteButton button, boolean inside);
}
```

JHotDraw ya viene con una implementación por defecto del DrawingEditor tanto para aplicaciones (DrawApplication) como para applets (DrawApplet).

DrawApplication y DrawApplet se encargan de crear las herramientas, menús y paletas más comunes.

Si el usuario quiere crear un nuevo editor adaptando alguno de los elementos de DrawApplication o DrawApplet puede utilizar los factory methods (patrón **Factory Method**), que hay en las dos clases, con comportamiento por defecto que pueden ser redefinidos.

- createEditMenu()
- createFileMenu()
- createFontMenu()
- createFontSizeMenu()
- createFontStyleMenu()
- createMenus(MenuBar)
- createSelectionTool()
- createStatusLine()
- createToolButton(String, String, Tool)
- createToolPalette()
- createTools(Panel)
-

```
/*
 * @(#)DrawApplication.java 5.1
 *
 */

package CH.ifa.draw.application;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.util.*;

/**
 * DrawApplication defines a standard presentation for
 * standalone drawing editors. The presentation is
 * customized in subclasses.
 * The application is started as follows:
 * <pre>
 * public static void main(String[] args) {
 *     MayDrawApp window = new MyDrawApp();
 *     window.open();
 * }
 * </pre>
```

```

*/

public class DrawApplication
    extends Frame
    implements DrawingEditor, PaletteListener {

    private Drawing          fDrawing;
    private Tool             fTool;
    private Iconkit          fIconkit;

    private TextField        fStatusLine;
    private StandardDrawingView fView;
    private ToolButton       fDefaultToolButton;
    private ToolButton       fSelectedToolButton;

    private String           fDrawingFilename;
    static String            fgUntitled = "untitled";

    // the image resource path
    private static final String fgDrawPath = "/CH/ifa/draw/";
    public static final String IMAGES = fgDrawPath+"images/";

    /**
     * The index of the file menu in the menu bar.
     */
    public static final int    FILE_MENU = 0;
    /**
     * The index of the edit menu in the menu bar.
     */
    public static final int    EDIT_MENU = 1;
    /**
     * The index of the alignment menu in the menu bar.
     */
    public static final int    ALIGNMENT_MENU = 2;
    /**
     * The index of the attributes menu in the menu bar.
     */
    public static final int    ATTRIBUTES_MENU = 3;

    /**
     * Constructs a drawing window with a default title.
     */
    public DrawApplication() {
        super("JHotDraw");
    }

    /**
     * Constructs a drawing window with the given title.
     */
    public DrawApplication(String title) {
        super(title);
    }

    /**
     * Opens the window and initializes its contents.
     * Clients usually only call but don't override it.
     */
}

```

```

public void open() {
    fIconkit = new Iconkit(this);
    setLayout(new BorderLayout());

    fView = createDrawingView();
    Component contents = createContents(fView);
    add("Center", contents);
    //add("Center", fView);

    Panel tools = createToolPalette();
    createTools(tools);
    add("West", tools);

    fStatusLine = createStatusLine();
    add("South", fStatusLine);

    MenuBar mb = new MenuBar();
    createMenus(mb);
    setMenuBar(mb);

    initDrawing();
    Dimension d = defaultSize();
    setSize(d.width, d.height);

    addListeners();

    show();
}

/**
 * Registers the listeners for this window
 */
protected void addListeners() {
    addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent event) {
                exit();
            }
        }
    );
}

private void initDrawing() {
    fDrawing = createDrawing();
    fDrawingFilename = fgUntitled;
    fView.setDrawing(fDrawing);
    toolDone();
}

/**
 * Creates the standard menus. Clients override this
 * method to add additional menus.
 */
protected void createMenus(MenuBar mb) {
    mb.add(createFileMenu());
    mb.add(createEditMenu());
    mb.add(createAlignmentMenu());
}

```

```

        mb.add(createAttributesMenu());
        mb.add(createDebugMenu());
    }

    /**
     * Creates the file menu. Clients override this
     * method to add additional menu items.
     */
    protected Menu createFileMenu() {
        Menu menu = new Menu("File");
        MenuItem mi = new MenuItem("New", new MenuShortcut('n'));
        mi.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    promptNew();
                }
            }
        );
        menu.add(mi);

        mi = new MenuItem("Open...", new MenuShortcut('o'));
        mi.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    promptOpen();
                }
            }
        );
        menu.add(mi);

        mi = new MenuItem("Save As...", new MenuShortcut('s'));
        mi.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    promptSaveAs();
                }
            }
        );
        menu.add(mi);

        mi = new MenuItem("Save As Serialized...");
        mi.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    promptSaveAsSerialized();
                }
            }
        );
        menu.add(mi);
        menu.addSeparator();
        mi = new MenuItem("Print...", new MenuShortcut('p'));
        mi.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    print();
                }
            }
        );
    }

```



```

        menu.add(mi);
        menu.addSeparator();
        mi = new MenuItem("Exit");
        mi.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    exit();
                }
            }
        );
        menu.add(mi);
        return menu;
    }

    /**
     * Creates the edit menu. Clients override this
     * method to add additional menu items.
     */
    protected Menu createEditMenu() {
        CommandMenu menu = new CommandMenu("Edit");
        menu.add(new CutCommand("Cut", fView), new MenuShortcut('x'));
        menu.add(new CopyCommand("Copy", fView), new MenuShortcut('c'));
        menu.add(new PasteCommand("Paste", fView), new
MenuShortcut('v'));
        menu.addSeparator();
        menu.add(new DuplicateCommand("Duplicate", fView), new
MenuShortcut('d'));
        menu.add(new DeleteCommand("Delete", fView));
        menu.addSeparator();
        menu.add(new GroupCommand("Group", fView));
        menu.add(new UngroupCommand("Ungroup", fView));
        menu.addSeparator();
        menu.add(new SendToBackCommand("Send to Back", fView));
        menu.add(new BringToFrontCommand("Bring to Front", fView));
        return menu;
    }

    /**
     * Creates the alignment menu. Clients override this
     * method to add additional menu items.
     */
    protected Menu createAlignmentMenu() {
        CommandMenu menu = new CommandMenu("Align");
        menu.add(new ToggleGridCommand("Toggle Snap to Grid", fView, new
Point(4,4)));
        menu.addSeparator();
        menu.add(new AlignCommand("Lefts", fView, AlignCommand.LEFTS));
        menu.add(new AlignCommand("Centers", fView,
AlignCommand.CENTERS));
        menu.add(new AlignCommand("Rights", fView,
AlignCommand.RIGHTS));
        menu.addSeparator();
        menu.add(new AlignCommand("Tops", fView, AlignCommand.TOPS));
        menu.add(new AlignCommand("Middles", fView,
AlignCommand.MIDDLES));
        menu.add(new AlignCommand("Bottoms", fView,
AlignCommand.BOTTOMS));
        return menu;
    }

```

```

    }

    /**
     * Creates the debug menu. Clients override this
     * method to add additional menu items.
     */
    protected Menu createDebugMenu() {
        Menu menu = new Menu("Debug");

        MenuItem mi = new MenuItem("Simple Update");
        mi.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    fView.setDisplayUpdate(new SimpleUpdateStrategy());
                }
            }
        );
        menu.add(mi);

        mi = new MenuItem("Buffered Update");
        mi.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    fView.setDisplayUpdate(new
BufferedUpdateStrategy());
                }
            }
        );
        menu.add(mi);
        return menu;
    }

    /**
     * Creates the attributes menu and its submenus. Clients override
     * this method to add additional menu items.
     */
    protected Menu createAttributesMenu() {
        Menu menu = new Menu("Attributes");
        menu.add(createColorMenu("Fill Color", "FillColor"));
        menu.add(createColorMenu("Pen Color", "FrameColor"));
        menu.add(createArrowMenu());
        menu.addSeparator();
        menu.add(createFontMenu());
        menu.add(createFontSizeMenu());
        menu.add(createFontStyleMenu());
        menu.add(createColorMenu("Text Color", "TextColor"));
        return menu;
    }

    /**
     * Creates the color menu.
     */
    protected Menu createColorMenu(String title, String attribute) {
        CommandMenu menu = new CommandMenu(title);
        for (int i=0; i<ColorMap.size(); i++)
            menu.add(
                new ChangeAttributeCommand(
                    ColorMap.name(i),

```

```

        attribute,
        ColorMap.color(i),
        fView
    )
    );
    return menu;
}

/**
 * Creates the arrows menu.
 */
protected Menu createArrowMenu() {
    CommandMenu menu = new CommandMenu("Arrow");
    menu.add(new ChangeAttributeCommand("none", "ArrowMode",
new Integer(PolyLineFigure.ARROW_TIP_NONE), fView));
    menu.add(new ChangeAttributeCommand("at Start", "ArrowMode",
new Integer(PolyLineFigure.ARROW_TIP_START), fView));
    menu.add(new ChangeAttributeCommand("at End", "ArrowMode",
new Integer(PolyLineFigure.ARROW_TIP_END), fView));
    menu.add(new ChangeAttributeCommand("at Both", "ArrowMode",
new Integer(PolyLineFigure.ARROW_TIP_BOTH), fView));
    return menu;
}

/**
 * Creates the fonts menus. It installs all available fonts
 * supported by the toolkit implementation.
 */
protected Menu createFontMenu() {
    CommandMenu menu = new CommandMenu("Font");
    String fonts[] = Toolkit.getDefaultToolkit().getFontList();
    for (int i = 0; i < fonts.length; i++)
        menu.add(new ChangeAttributeCommand(fonts[i], "FontName",
fonts[i], fView));
    return menu;
}

/**
 * Creates the font style menu with entries (Plain, Italic, Bold).
 */
protected Menu createFontStyleMenu() {
    CommandMenu menu = new CommandMenu("Font Style");
    menu.add(new ChangeAttributeCommand("Plain", "FontStyle", new
Integer(Font.PLAIN), fView));
    menu.add(new ChangeAttributeCommand("Italic", "FontStyle", new
Integer(Font.ITALIC), fView));
    menu.add(new ChangeAttributeCommand("Bold", "FontStyle", new
Integer(Font.BOLD), fView));
    return menu;
}

/**
 * Creates the font size menu.
 */
protected Menu createFontSizeMenu() {
    CommandMenu menu = new CommandMenu("Font Size");
    int sizes[] = { 9, 10, 12, 14, 18, 24, 36, 48, 72 };
    for (int i = 0; i < sizes.length; i++) {

```

```

        menu.add(
            new ChangeAttributeCommand(
                Integer.toString(sizes[i]),
                "FontSize",
                new Integer(sizes[i]), fView)
        );
    }
    return menu;
}

/**
 * Creates the tool palette.
 */
protected Panel createToolPalette() {
    Panel palette = new Panel();
    palette.setBackground(Color.lightGray);
    palette.setLayout(new PaletteLayout(2,new Point(2,2)));
    return palette;
}

/**
 * Creates the tools. By default only the selection tool is added.
 * Override this method to add additional tools.
 * Call the inherited method to include the selection tool.
 * @param palette the palette where the tools are added.
 */
protected void createTools(Panel palette) {
    Tool tool = createSelectionTool();

    fDefaultToolButton = createToolButton(IMAGES+"SEL", "Selection
Tool", tool);
    palette.add(fDefaultToolButton);
}

/**
 * Creates the selection tool used in this editor. Override to use
 * a custom selection tool.
 */
protected Tool createSelectionTool() {
    return new SelectionTool(view());
}

/**
 * Creates a tool button with the given image, tool, and text
 */
protected ToolButton createToolButton(String iconName, String
toolName, Tool tool) {
    return new ToolButton(this, iconName, toolName, tool);
}

/**
 * Creates the drawing view used in this application.
 * You need to override this method to use a DrawingView
 * subclass in your application. By default a standard
 * DrawingView is returned.
 */
protected StandardDrawingView createDrawingView() {
    Dimension d = getDrawingViewSize();

```

```

        return new StandardDrawingView(this, d.width, d.height);
    }

    /**
     * Override to define the dimensions of the drawing view.
     */
    protected Dimension getDrawingViewSize() {
        return new Dimension(400, 600);
    }

    /**
     * Creates the drawing used in this application.
     * You need to override this method to use a Drawing
     * subclass in your application. By default a standard
     * Drawing is returned.
     */
    protected Drawing createDrawing() {
        return new StandardDrawing();
    }

    /**
     * Creates the contents component of the application
     * frame. By default the DrawingView is returned in
     * a ScrollPane.
     */
    protected Component createContents(StandardDrawingView view) {
        ScrollPane sp = new ScrollPane();
        Adjustable vadjust = sp.getVAdjustable();
        Adjustable hadjust = sp.getHAdjustable();
        hadjust.setUnitIncrement(16);
        vadjust.setUnitIncrement(16);

        sp.add(view);
        return sp;
    }

    /**
     * Sets the drawing to be edited.
     */
    public void setDrawing(Drawing drawing) {
        fView.setDrawing(drawing);
        fDrawing = drawing;
    }

    /**
     * Gets the default size of the window.
     */
    protected Dimension defaultSize() {
        return new Dimension(430, 406);
    }

    /**
     * Creates the status line.
     */
    protected TextField createStatusLine() {
        TextField field = new TextField("No Tool", 40);
        field.setEditable(false);
        return field;
    }

```

```

    }

    /**
     * Handles a user selection in the palette.
     * @see PaletteListener
     */
    public void paletteUserSelected(PaletteButton button) {
        ToolButton toolButton = (ToolButton) button;
        setTool(toolButton.tool(), toolButton.name());
        setSelected(toolButton);
    }

    /**
     * Handles when the mouse enters or leaves a palette button.
     * @see PaletteListener
     */
    public void paletteUserOver(PaletteButton button, boolean inside)
    {
        ToolButton toolButton = (ToolButton) button;
        if (inside)
            showStatus(toolButton.name());
        else
            showStatus(fSelectedToolButton.name());
    }

    /**
     * Gets the current drawing.
     * @see DrawingEditor
     */
    public Drawing drawing() {
        return fDrawing;
    }

    /**
     * Gets the current tool.
     * @see DrawingEditor
     */
    public Tool tool() {
        return fTool;
    }

    /**
     * Gets the current drawing view.
     * @see DrawingEditor
     */
    public DrawingView view() {
        return fView;
    }

    /**
     * Sets the default tool of the editor.
     * @see DrawingEditor
     */
    public void toolDone() {
        if (fDefaultToolButton != null) {
            setTool(fDefaultToolButton.tool(),
fDefaultToolButton.name());
            setSelected(fDefaultToolButton);
        }
    }

```

```

    }

}

/**
 * Handles a change of the current selection. Updates all
 * menu items that are selection sensitive.
 * @see DrawingEditor
 */
public void selectionChanged(DrawingView view) {
    MenuBar mb = getMenuBar();
    CommandMenu editMenu = (CommandMenu)mb.getMenu(EDIT_MENU);
    editMenu.setEnabled();
    CommandMenu alignmentMenu =
(CommandMenu)mb.getMenu(ALIGNMENT_MENU);
    alignmentMenu.setEnabled();
}

/**
 * Shows a status message.
 * @see DrawingEditor
 */
public void showStatus(String string) {
    fStatusLine.setText(string);
}

private void setTool(Tool t, String name) {
    if (fTool != null)
        fTool.deactivate();
    fTool = t;
    if (fTool != null) {
        fStatusLine.setText(name);
        fTool.activate();
    }
}

private void setSelected(ToolButton button) {
    if (fSelectedToolButton != null)
        fSelectedToolButton.reset();
    fSelectedToolButton = button;
    if (fSelectedToolButton != null)
        fSelectedToolButton.select();
}

/**
 * Exits the application. You should never override this method
 */
public void exit() {
    destroy();
    setVisible(false); // hide the Frame
    dispose(); // tell windowing system to free resources
    System.exit(0);
}

/**
 * Handles additional clean up operations. Override to destroy
 * or release drawing editor resources.
 */
protected void destroy() {

```

```

    }

    /**
     * Resets the drawing to a new empty drawing.
     */
    public void promptNew() {
        initDrawing();
    }

    /**
     * Shows a file dialog and opens a drawing.
     */
    public void promptOpen() {
        FileDialog dialog = new FileDialog(this, "Open File...",
        FileDialog.LOAD);
        dialog.show();
        String filename = dialog.getFile();
        if (filename != null) {
            filename = stripTrailingAsterisks(filename);
            String dirname = dialog.getDirectory();
            loadDrawing(dirname + filename);
        }
        dialog.dispose();
    }

    /**
     * Shows a file dialog and saves drawing.
     */
    public void promptSaveAs() {
        toolDone();
        String path = getSavePath("Save File...");
        if (path != null) {
            if (!path.endsWith(".draw"))
                path += ".draw";
            saveAsStorableOutput(path);
        }
    }

    /**
     * Shows a file dialog and saves drawing.
     */
    public void promptSaveAsSerialized() {
        toolDone();
        String path = getSavePath("Save File...");
        if (path != null) {
            if (!path.endsWith(".ser"))
                path += ".ser";
            saveAsObjectOutput(path);
        }
    }

    /**
     * Prints the drawing.
     */
    public void print() {
        fTool.deactivate();
        PrintJob printJob = getToolkit().getPrintJob(this, "Print
        Drawing", null);
    }

```



```

        if (printJob != null) {
            Graphics pg = printJob.getGraphics();

            if (pg != null) {
                fView.printAll(pg);
                pg.dispose(); // flush page
            }
            printJob.end();
        }
        fTool.activate();
    }

    private String getSavePath(String title) {
        String path = null;
        FileDialog dialog = new FileDialog(this, title,
FileDialog.SAVE);
        dialog.show();
        String filename = dialog.getFile();
        if (filename != null) {
            filename = stripTrailingAsterisks(filename);
            String dirname = dialog.getDirectory();
            path = dirname + filename;
        }
        dialog.dispose();
        return path;
    }

    private String stripTrailingAsterisks(String filename) {
        // workaround for bug on NT
        if (filename.endsWith("*. *"))
            return filename.substring(0, filename.length() - 4);
        else
            return filename;
    }

    private void saveAsStorableOutput(String file) {
        // TBD: should write a MIME header
        try {
            FileOutputStream stream = new FileOutputStream(file);
            StorableOutput output = new StorableOutput(stream);
            output.writeStorable(fDrawing);
            output.close();
        } catch (IOException e) {
            showStatus(e.toString());
        }
    }

    private void saveAsObjectOutput(String file) {
        // TBD: should write a MIME header
        try {
            FileOutputStream stream = new FileOutputStream(file);
            ObjectOutputStream output = new ObjectOutputStream(stream);
            output.writeObject(fDrawing);
            output.close();
        } catch (IOException e) {
            showStatus(e.toString());
        }
    }

```

```

    }

    private void loadDrawing(String file) {
        toolDone();
        String type = guessType(file);
        if (type.equals("storable"))
            readFromStorableInput(file);
        else if (type.equals("serialized"))
            readFromObjectInput(file);
        else
            showStatus("Unknown file type");
    }

    private void readFromStorableInput(String file) {
        try {
            FileInputStream stream = new FileInputStream(file);
            StorableInput input = new StorableInput(stream);
            fDrawing.release();
            fDrawing = (Drawing)input.readStorable();
            fView.setDrawing(fDrawing);
        } catch (IOException e) {
            initDrawing();
            showStatus("Error: " + e);
        }
    }

    private void readFromObjectInput(String file) {
        try {
            FileInputStream stream = new FileInputStream(file);
            ObjectInput input = new ObjectInputStream(stream);
            fDrawing.release();
            fDrawing = (Drawing)input.readObject();
            fView.setDrawing(fDrawing);
        } catch (IOException e) {
            initDrawing();
            showStatus("Error: " + e);
        } catch (ClassNotFoundException e) {
            initDrawing();
            showStatus("Class not found: " + e);
        }
    }

    private String guessType(String file) {
        if (file.endsWith(".draw"))
            return "storable";
        if (file.endsWith(".ser"))
            return "serialized";
        return "unknown";
    }
}

```

Un ejemplo de cómo crear un nuevo editor adaptando la clase DrawApplication utilizando los factory methods sería este:

```
/*
```

```

* @(#)PertApplication.java 5.1
*
*/

package CH.ifa.draw.samples.pert;

import java.awt.*;
import java.util.*;
import java.io.*;
import java.net.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.application.*;

public class PertApplication extends DrawApplication {

    static private final String PERTIMAGES =
"/CH/ifa/draw/samples/pert/images/";

    PertApplication() {
        super("PERT Editor");
    }

    protected void createTools(Panel palette) {
        super.createTools(palette);

        Tool tool;
        tool = new TextTool(view(), new TextFigure());
        palette.add(createToolButton(IMAGES+"TEXT", "Text Tool",
tool));

        // the generic but slower version
        //tool = new CreationTool(new PertFigure());
        //palette.add(createToolButton(PERTIMAGES+"PERT", "Task Tool",
//tool));

        tool = new PertFigureCreationTool(view());
        palette.add(createToolButton(PERTIMAGES+"PERT", "Task Tool",
tool));

        tool = new ConnectionTool(view(), new PertDependency());
        palette.add(createToolButton(IMAGES+"CONN", "Dependency Tool",
tool));

        tool = new CreationTool(view(), new LineFigure());
        palette.add(createToolButton(IMAGES+"Line", "Line Tool",
tool));
    }

    //-- main -----

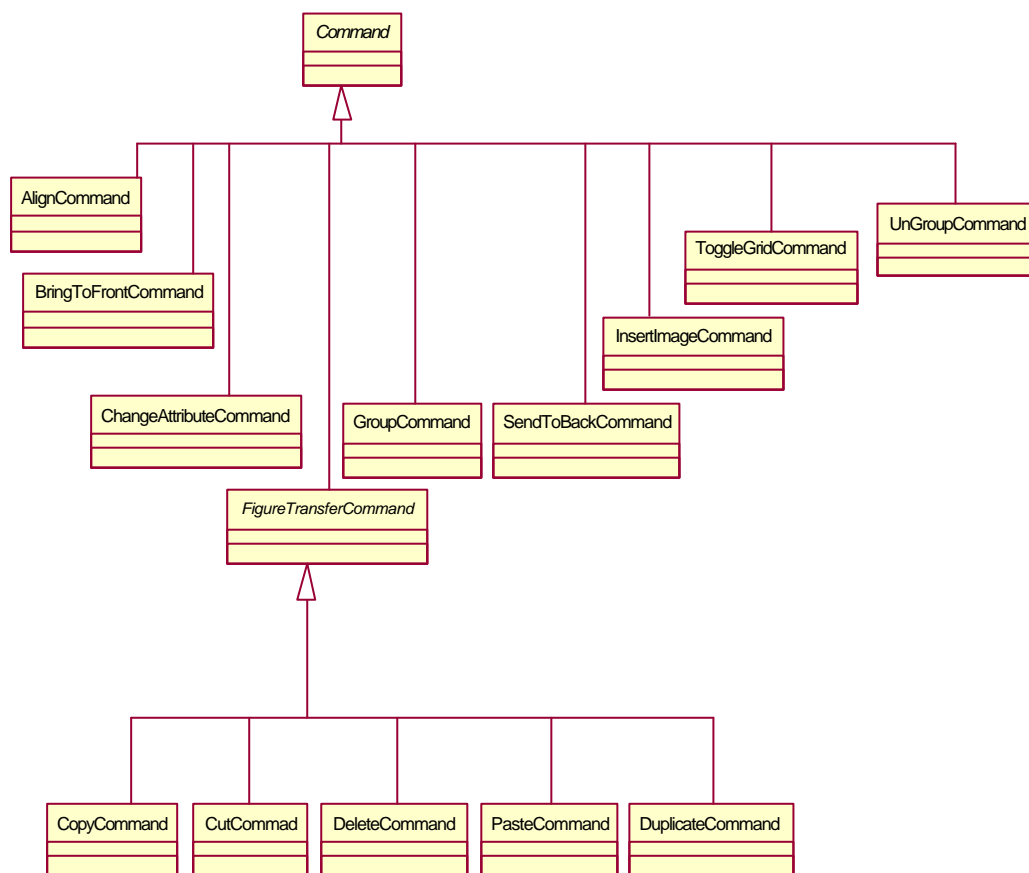
    public static void main(String[] args) {
        PertApplication pert = new PertApplication();
        pert.open();
    }
}

```

Muchas operaciones pueden invocarse desde varios sitios: menús, atajos de teclado, botones y muchas operaciones son las mismas para todos los editores (cut, paste, delete, align, order, ...)

Para no duplicar código (mala solución: antipatrón Cut-and-paste programming”cortar y pegar código”) y para no tener que repetir las mismas operaciones en cada nuevo editor (reutilización) se utiliza el patrón de diseño **Command**. La clase abstracta Command realiza el papel de *ComandoAbstracto* del patrón Command. Todas las clases que heredan de Command juegan el papel de *ComandoConcreto*.

Este patrón Command en JHotDraw no soporta la operación undo (deshacer).



Utilización del patrón Command.

Los comandos encapsulan una acción para ser ejecutada. Los comandos tienen un nombre y pueden ser utilizados en conjunción con los componentes de los interfaces del usuario *Command enabled*.

```

/*
 * @(#)Command.java 5.1
 *
 */

```

```

package CH.ifa.draw.util;

import java.awt.*;
import java.util.*;

/**
 * Commands encapsulate an action to be executed. Commands have
 * a name and can be used in conjunction with <i>Command enabled</i>
 * ui components.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld010.htm>Command</a></b><br>
 * Command is a simple instance of the command pattern without undo
 * support.
 * <hr>
 *
 * @see CommandButton
 * @see CommandMenu
 * @see CommandChoice
 */
public abstract class Command {

    private String fName;

    /**
     * Constructs a command with the given name.
     */
    public Command(String name) {
        fName = name;
    }

    /**
     * Executes the command.
     */
    public abstract void execute();

    /**
     * Tests if the command can be executed.
     */
    public boolean isExecutable() {
        return true;
    }

    /**
     * Gets the command name.
     */
    public String name() {
        return fName;
    }
}

```

Como ejemplo de una subclase de la clase Command vamos a mostrar el código de la clase GroupCommand.

GroupCommand es un comando para agrupar la selección en un GroupFigure.

```
/*
 * @(#)GroupCommand.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.util.*;
import CH.ifa.draw.util.Command;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;

/**
 * Command to group the selection into a GroupFigure.
 *
 * @see GroupFigure
 */
public class GroupCommand extends Command {

    private DrawingView fView;

    /**
     * Constructs a group command.
     * @param name the command name
     * @param view the target view
     */
    public GroupCommand(String name, DrawingView view) {
        super(name);
        fView = view;
    }

    public void execute() {
        Vector selected = fView.selectionZOrdered();
        Drawing drawing = fView.drawing();
        if (selected.size() > 0) {
            fView.clearSelection();
            drawing.orphanAll(selected);

            GroupFigure group = new GroupFigure();
            group.addAll(selected);
            fView.addToSelection(drawing.add(group));
        }
        fView.checkDamage();
    }

    public boolean isExecutable() {
        return fView.selectionCount() > 0;
    }
}
```

De esta manera las operaciones más habituales se pueden utilizar directamente: CutCommand, CopyCommand, PasteCommand, DeleteCommand, DuplicateCommand, GroupCommand, UngroupCommand, ChangeAttributeCommand, AlignCommand, etc.

JHotDraw incluye componentes que ejecutan Commands: CommandMenu, CommandChoice y CommandButton.

CommandMenu es un Command que habilita el menú. Al seleccionar una opción del menú se ejecuta el comando correspondiente.

```

/*
 * @(#)CommandMenu.java 5.1
 *
 */

package CH.ifa.draw.util;

import java.awt.*;
import java.util.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

/**
 * A Command enabled menu. Selecting a menu item
 * executes the corresponding command.
 *
 * @see Command
 */
public class CommandMenu
    extends Menu implements ActionListener {

    private Vector    fCommands;

    public CommandMenu(String name) {
        super(name);
        fCommands = new Vector(10);
    }

    /**
     * Adds a command to the menu. The item's label is
     * the command's name.
     */
    public synchronized void add(Command command) {
        MenuItem m = new MenuItem(command.name());
        m.addActionListener(this);
        add(m);
        fCommands.addElement(command);
    }

    /**
     * Adds a command with the given short cut to the menu. The item's
     * label is the command's name.
     */
    public synchronized void add(Command command, MenuShortcut
shortcut) {
        MenuItem m = new MenuItem(command.name(), shortcut);
        m.setName(command.name());
        m.addActionListener(this);
        add(m);
    }
}

```

```

        fCommands.addElement(command);
    }

    public synchronized void remove(Command command) {
        System.out.println("not implemented");
    }

    public synchronized void remove(MenuItem item) {
        System.out.println("not implemented");
    }

    /**
     * Changes the enabling/disabling state of a named menu item.
     */
    public synchronized void enable(String name, boolean state) {
        for (int i = 0; i < getItemCount(); i++) {
            MenuItem item = getItem(i);
            if (name.equals(item.getLabel())) {
                item.setEnabled(state);
                return;
            }
        }
    }

    public synchronized void checkEnabled() {
        int j = 0;
        for (int i = 0; i < getItemCount(); i++) {
            MenuItem item = getItem(i);
            // ignore separators
            // a separator has a hyphen as its label
            if (item.getLabel().equals("-"))
                continue;
            Command cmd = (Command)fCommands.elementAt(j);
            item.setEnabled(cmd.isExecutable());
            j++;
        }
    }

    /**
     * Executes the command.
     */
    public void actionPerformed(ActionEvent e) {
        int j = 0;
        Object source = e.getSource();
        for (int i = 0; i < getItemCount(); i++) {
            MenuItem item = getItem(i);
            // ignore separators
            // a separator has a hyphen as its label
            if (item.getLabel().equals("-"))
                continue;
            if (source == item) {
                Command cmd = (Command)fCommands.elementAt(j);
                cmd.execute();
                break;
            }
            j++;
        }
    }
}

```



```
}
```

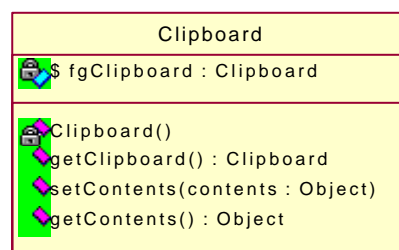
Un ejemplo de cómo se utiliza la clase CommandMenu lo podemos ver en el método createEditMenu de la clase DrawApplication.

```
/**
 * Creates the edit menu. Clients override this
 * method to add additional menu items.
 */
protected Menu createEditMenu() {
    CommandMenu menu = new CommandMenu("Edit");
    menu.add(new CutCommand("Cut", fView), new MenuShortcut('x'));
    menu.add(new CopyCommand("Copy", fView), new MenuShortcut('c'));
    menu.add(new PasteCommand("Paste", fView), new
MenuShortcut('v'));
    menu.addSeparator();
    menu.add(new DuplicateCommand("Duplicate", fView), new
MenuShortcut('d'));
    menu.add(new DeleteCommand("Delete", fView));
    menu.addSeparator();
    menu.add(new GroupCommand("Group", fView));
    menu.add(new UngroupCommand("Ungroup", fView));
    menu.addSeparator();
    menu.add(new SendToBackCommand("Send to Back", fView));
    menu.add(new BringToFrontCommand("Bring to Front", fView));
    return menu;
}
```

3.5.3.7. Más patrones de diseño

PATRÓN SINGLETON (Clipboard)

La clase Clipboard es un portapapeles global. Esta clase implementa el patrón de diseño **Singleton**. Es un singleton que puede ser utilizado para almacenar y conseguir los contenidos del portapapeles.



Utilización del patrón Singleton.

Al utilizar el patrón Singleton existe exactamente una única instancia de la clase Clipboard.

```
/*
 * @(#)Clipboard.java 5.1
 */
```

```

*/

package CH.ifa.draw.util;

import java.awt.*;
import java.util.*;

/**
 * A temporary replacement for a global clipboard.
 * It is a singleton that can be used to store and
 * get the contents of the clipboard.
 */
public class Clipboard {
    static Clipboard fgClipboard = new Clipboard();

    /**
     * Gets the clipboard.
     */
    static public Clipboard getClipboard() {
        return fgClipboard;
    }

    private Object fContents;

    private Clipboard() {
    }

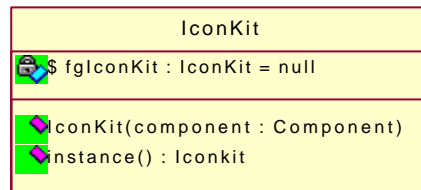
    /**
     * Sets the contents of the clipboard.
     */
    public void setContents(Object contents) {
        fContents = contents;
    }

    /**
     * Gets the contents of the clipboard.
     */
    public Object getContents() {
        return fContents;
    }
}

```

PATRÓN SINGLETON (Iconkit)

La clase IconKit soporta la compartición de imágenes. Mantiene un mapa de nombres de imagen y sus correspondiente imágenes. IconKit también soporta la carga de una colección de imágenes de un modo sincronizado. La resolución de un nombre de la ruta (path) para una imagen se delega al DrawingEditor.



Utilización del patrón Singleton.

Esta clase implementa el patrón de diseño **Singleton** es decir la clase IconKit es un singleton.

```

/*
 * @(#)Iconkit.java 5.1
 *
 */

package CH.ifa.draw.util;

import java.awt.*;
import java.awt.image.ImageProducer;
import java.util.*;

/**
 * The Iconkit class supports the sharing of images. It maintains
 * a map of image names and their corresponding images.
 *
 * Iconkit also supports to load a collection of images in
 * synchronized way.
 * The resolution of a path name to an image is delegated to the
 * DrawingEditor.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld031.htm>Singleton</a></b><br>
 * The Iconkit is a singleton.
 * <hr>
 */
public class Iconkit {
    private Hashtable      fMap;
    private Vector         fRegisteredImages;
    private Component      fComponent;
    private final static int ID = 123;
    private static Iconkit fgIconkit = null;
    private static boolean fgDebug = false;

    /**
     * Constructs an Iconkit that uses the given editor to
     * resolve image path names.
     */
    public Iconkit(Component component) {
        fMap = new Hashtable(53);
        fRegisteredImages = new Vector(10);
        fComponent = component;
        fgIconkit = this;
    }

    /**

```

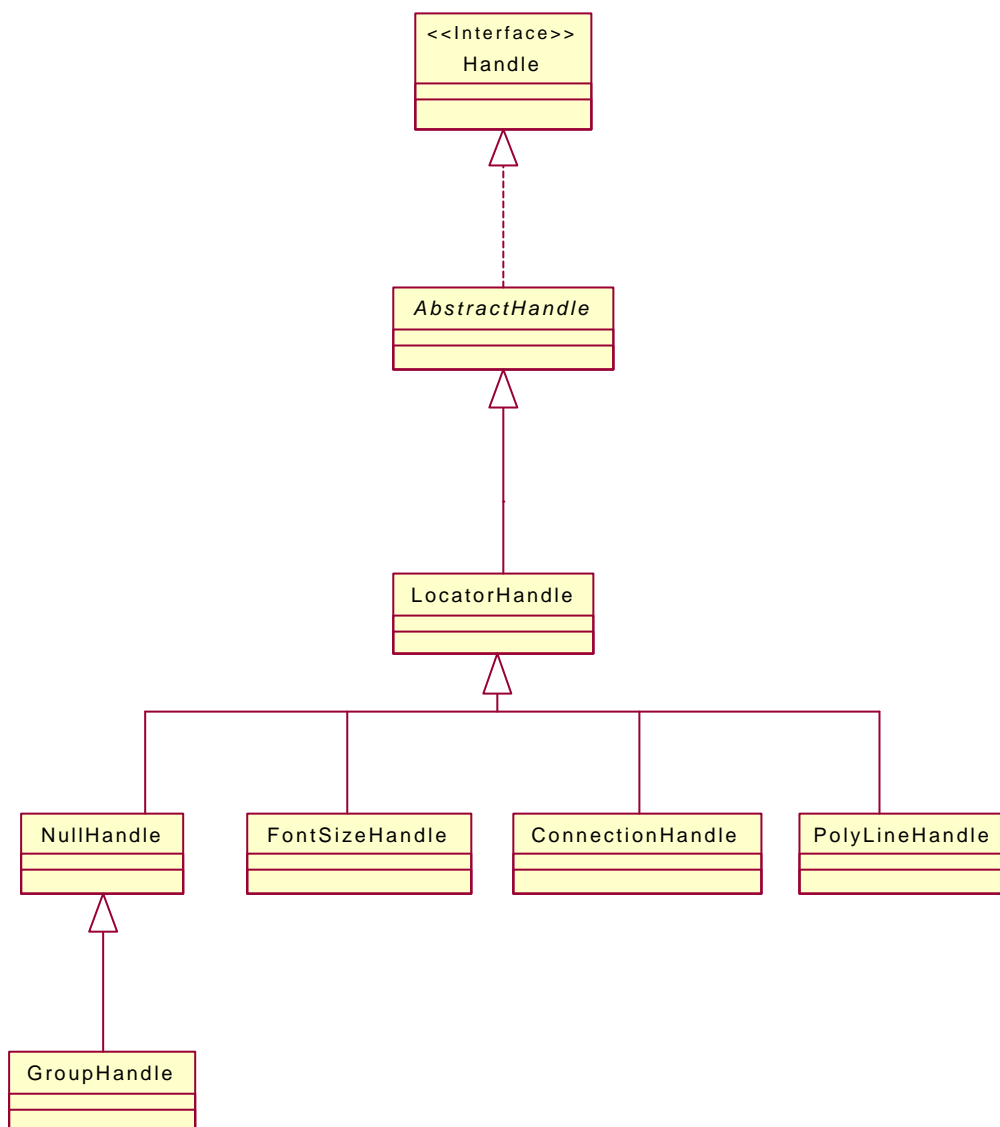
```
    * Gets the single instance
    */
    public static Iconkit instance() {
        return fgIconkit;
    }

    .....

}
```

PATRÓN NULL OBJECT

El patrón de diseño **Null Object** permite tratar los manejadores que no hacen nada del mismo modo que otros manejadores. NullHandle y GroupHandle juegan el papel de *OperacionNull* del patrón Null Object. Los otros tipos de manejadores: FontSizeHandle, ConnectionHandle y PolyLineHanle juegan el papel de *OperacionReal*. La clase LocatorHandle y sus superiores juegan el papel de *OperacionAbstracta*.



Utilización del patrón Null Object.

NullHandle es un manejador que no cambia la figura que lo posee. Su único propósito es mostrar que una figura esta seleccionada.

```

/*
 * @(#)NullHandle.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import CH.ifa.draw.framework.*;

/**
 * A handle that doesn't change the owned figure. Its only purpose is
 * to show feedback that a figure is selected.
 */

```

```

* <hr>
* <b>Design Patterns</b><P>
* 
* <b>NullObject</b><br>
* NullObject enables to treat handles that don't do
* anything in the same way as other handles.
*
*/
public class NullHandle extends LocatorHandle {

    /**
     * The handle's locator.
     */
    protected Locator fLocator;

    public NullHandle(Figure owner, Locator locator) {
        super(owner, locator);
    }

    /**
     * Draws the NullHandle. NullHandles are drawn as a
     * red framed rectangle.
     */
    public void draw(Graphics g) {
        Rectangle r = displayBox();

        g.setColor(Color.black);
        g.drawRect(r.x, r.y, r.width, r.height);
    }
}

```

Vamos a comparar el NullHandle con otro tipo de Handle como por ejemplo FontSizeHandle. FontSizeHandle es un Handle para cambiar el tamaño de la fuente por manipulación directa.

```

/*
 * @(#)FontSizeHandle.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;

/**
 * A Handle to change the font size by direct manipulation.
 */
public class FontSizeHandle extends LocatorHandle {

    private Font    fFont;
    private int     fSize;

    public FontSizeHandle(Figure owner, Locator l) {
        super(owner, l);
    }
}

```

```

    }

    public void invokeStart(int x, int y, DrawingView view) {
        TextFigure textOwner = (TextFigure) owner();
        fFont = textOwner.getFont();
        fSize = fFont.getSize();
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        TextFigure textOwner = (TextFigure) owner();
        int newSize = fSize + y-anchorY;
        textOwner.setFont(new Font(fFont.getName(), fFont.getStyle(),
newSize) );
    }

    public void draw(Graphics g) {
        Rectangle r = displayBox();

        g.setColor(Color.yellow);
        g.fillOval(r.x, r.y, r.width, r.height);

        g.setColor(Color.black);
        g.drawOval(r.x, r.y, r.width, r.height);
    }
}

```

GroupHandle es una clase que deriva de NullHandle y cambia el dibujo del handle para una GroupFigure.

```

/*
 * @(#)GroupHandle.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.NullHandle;

/**
 * A Handle for a GroupFigure.
 */
final class GroupHandle extends NullHandle {

    public GroupHandle(Figure owner, Locator locator) {
        super(owner, locator);
    }

    /**
     * Draws the Group handle.
     */
    public void draw(Graphics g) {
        Rectangle r = displayBox();
    }
}

```

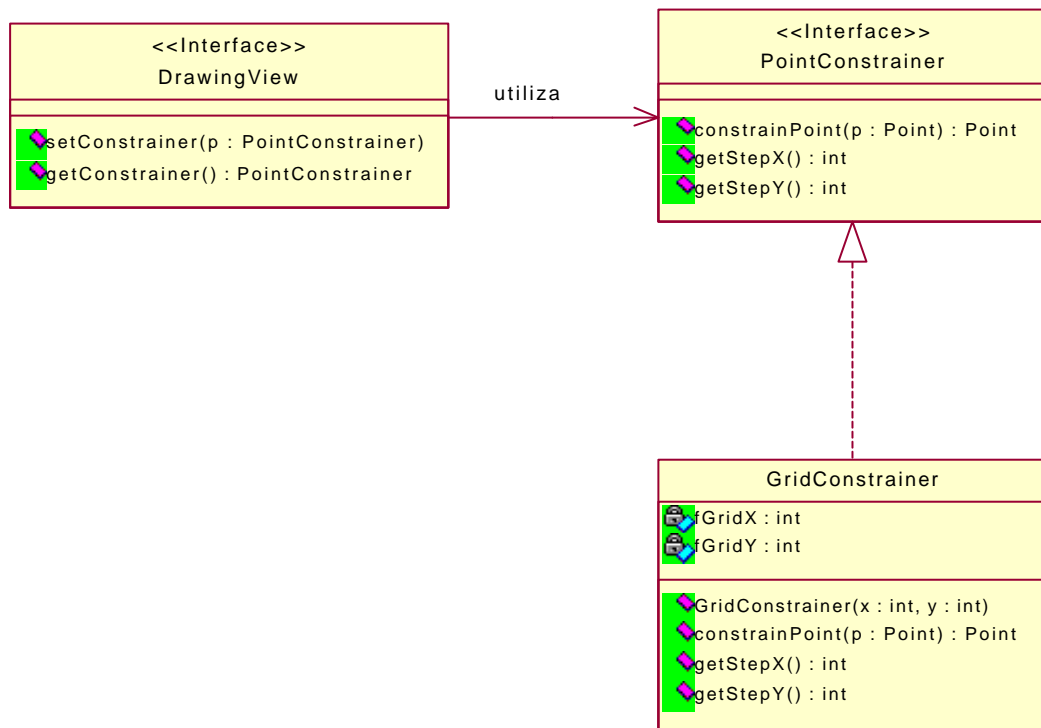
```

        g.setColor(Color.black);
        g.drawRect(r.x, r.y, r.width, r.height);
        r.grow(-1,-1);
        g.setColor(Color.white);
        g.drawRect(r.x, r.y, r.width, r.height);
    }
}

```

PATRÓN STRATEGY (PointConstrainer)

PointConstrainer es un interface que puede utilizarse para implementar diferentes tipos de grids (cuadrículas). Para ello se utiliza el patrón de diseño **Strategy**. El interface DrawingView juega el papel de *Cliente* del patrón Strategy. El interface PointConstrainer juega el papel de *StrategyAbstracto* y la clase GridConstrainer el papel de *StrategyConcreto*.



Utilización del patrón Strategy.

```

/*
 * @(#)PointConstrainer.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.*;

/**
 * Interface to constrain a Point. This can be used to implement
 * different kinds of grids.
 */

```



```

* <hr>
* <b>Design Patterns</b><P>
* 
* <b><a href=../pattlets/sld034.htm>Strategy</a></b><br>
* DrawingView is the StrategyContext.<br>
*
* @see DrawingView
*/

public interface PointConstrainer {
    /**
     * Constrains the given point.
     * @return constrained point.
     */
    public Point constrainPoint(Point p);

    /**
     * Gets the x offset to move an object.
     */
    public int getStepX();

    /**
     * Gets the y offset to move an object.
     */
    public int getStepY();
}

```

La clase GridConstrainer obliga a un punto de tal forma que caiga sobre una grid.

```

/*
 * @(#)GridConstrainer.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.io.Serializable;

import CH.ifa.draw.framework.PointConstrainer;

/**
 * Constrains a point such that it falls on a grid.
 *
 * @see DrawingView
 */

public class GridConstrainer implements PointConstrainer, Serializable
{
    private int fGridX;
    private int fGridY;

    public GridConstrainer(int x, int y) {

```

```

        fGridX = Math.max(1, x);
        fGridY = Math.max(1, y);
    }

    /**
     * Constrains the given point.
     * @return constrained point.
     */
    public Point constrainPoint(Point p) {
        p.x = ((p.x+fGridX/2) / fGridX) * fGridX;
        p.y = ((p.y+fGridY/2) / fGridY) * fGridY;
        return p;
    }

    /**
     * Gets the x offset to move an object.
     */
    public int getStepX() {
        return fGridX;
    }

    /**
     * Gets the y offset to move an object.
     */
    public int getStepY() {
        return fGridY;
    }
}

```

El interface `DrawingView` es el que hace de *Cliente* del patrón Strategy.

```

/*
 * @(#)DrawingView.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.image.ImageObserver;
import java.awt.*;
import java.util.*;
import java.io.*;
import CH.ifa.draw.util.*;

/**
 * DrawingView renders a Drawing and listens to its changes.
 * It receives user input and delegates it to the current tool.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld026.htm>Observer</a></b><br>
 * DrawingView observes drawing for changes via the DrawingListener
 interface.<br>
 * 
 * <b><a href=../pattlets/sld032.htm>State</a></b><br>
 * DrawingView plays the role of the StateContext in
 * the State pattern. Tool is the State.<br>

```

```

* 
* <b><a href=../pattlets/sld034.htm>Strategy</a></b><br>
* DrawingView is the StrategyContext in the Strategy pattern
* with regard to the UpdateStrategy. <br>
* DrawingView is the StrategyContext for the PointConstrainer.
*
* @see Drawing
* @see Painter
* @see Tool
* /

public interface DrawingView extends ImageObserver,
DrawingChangeListener {

.....

    /**
     * Sets the current point constrainer.
     */
    public void setConstrainer(PointConstrainer p);

    /**
     * Gets the current grid setting.
     */
    public PointConstrainer getConstrainer();

.....
}

```

PATRÓN PROTOTYPE (ConnectionHandle)

ConnectionHandle es un manejador que conecta figuras. El objeto connection que se crea es especificado por un Prototype. Por lo tanto se utiliza el patrón de diseño **Prototype**.

ConnectionHandle crea la conexión clonando un prototipo.

```

/*
 * @(#)ConnectionHandle.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;

import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.Geom;

/**
 * A handle to connect figures.
 * The connection object to be created is specified by a prototype.
 * <hr>
 * <b>Design Patterns</b><P>

```

```

* 
* <b><a href=../pattlets/sld029.htm>Prototype</a></b><br>
* ConnectionHandle creates the connection by cloning a prototype.
* <hr>
*
* @see ConnectionFigure
* @see Object#clone
*/

public class ConnectionHandle extends LocatorHandle {

    /**
     * the currently created connection
     */
    private ConnectionFigure fConnection;

    /**
     * the prototype of the connection to be created
     */
    private ConnectionFigure fPrototype;

    /**
     * the current target
     */
    private Figure fTarget = null;

    /**
     * Constructs a handle with the given owner, locator, and
     * connection prototype
     */
    public ConnectionHandle(Figure owner, Locator l, ConnectionFigure
prototype) {
        super(owner, l);
        fPrototype = prototype;
    }

    .....

    /**
     * Creates the ConnectionFigure. By default the figure prototype
     * is cloned.
     */
    protected ConnectionFigure createConnection() {
        return (ConnectionFigure)fPrototype.clone();
    }

    .....
}

```

PATRÓN PROTOTYPE (ConnectionTool)

ConnectionTool es una herramienta que puede ser utilizada para conectar figuras, para dividir conexiones, y para unir dos segmentos de una conexión. Las ConnectionTools cambian la visibilidad de los Connectors cuando entra una figura. El objeto connection que se crea es especificado por un Prototype. Por lo tanto se utiliza el patrón de diseño **Prototype**.

ConnectionTool crea la conexión clonando un prototipo.

```

/*
 * @(#)ConnectionTool.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.awt.event.MouseEvent;
import java.util.*;

import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.Geom;

/**
 * A tool that can be used to connect figures, to split
 * connections, and to join two segments of a connection.
 * ConnectionTools turns the visibility of the Connectors
 * on when it enters a figure.
 * The connection object to be created is specified by a prototype.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld029.htm>Prototype</a></b><br>
 * ConnectionTools creates the connection by cloning a prototype.
 * <hr>
 *
 * @see ConnectionFigure
 * @see Object#clone
 */

public class ConnectionTool extends AbstractTool {

    /**
     * the anchor point of the interaction
     */
    private Connector fStartConnector;
    private Connector fEndConnector;
    private Connector fConnectorTarget = null;

    private Figure fTarget = null;

    /**
     * the currently created figure
     */

```

```

private ConnectionFigure fConnection;

.....

/**
 * the prototypical figure that is used to create new
 * connections.
 */
private ConnectionFigure fPrototype;

public ConnectionTool(DrawingView view, ConnectionFigure
prototype) {
    super(view);
    fPrototype = prototype;
}

.....

/**
 * Creates the ConnectionFigure. By default the figure prototype
 * is cloned.
 */
protected ConnectionFigure createConnection() {
    return (ConnectionFigure)fPrototype.clone();
}

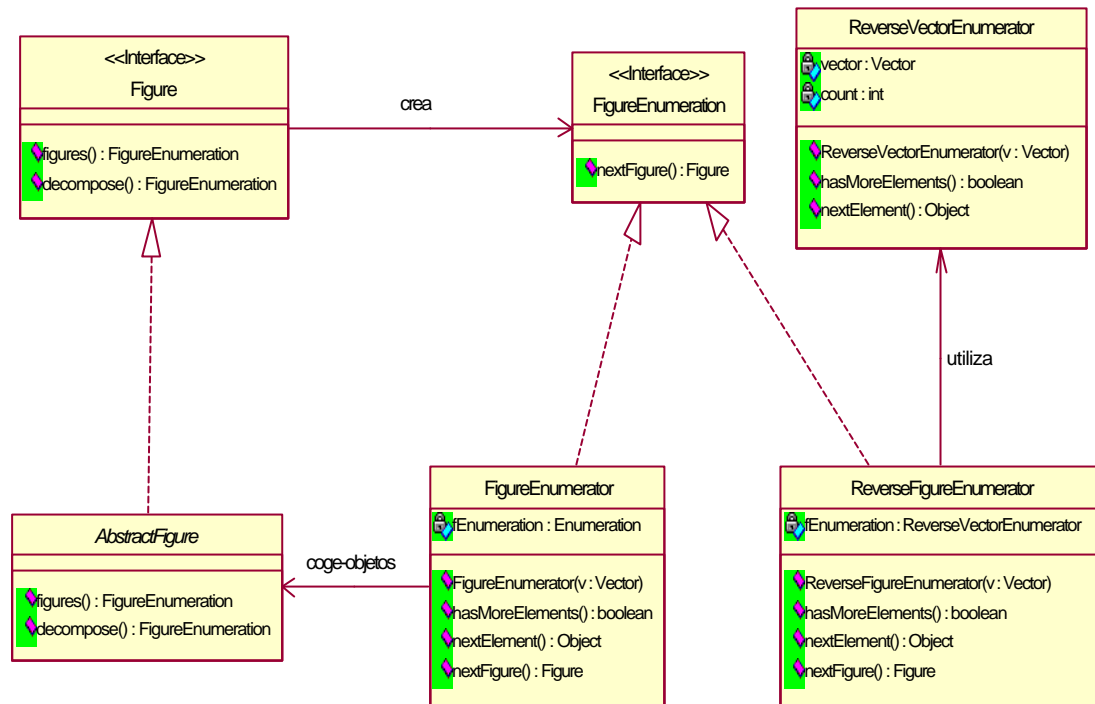
.....
}

```

PATRÓN ITERATOR

Este patrón ya se ha nombrado en el apartado 3.5.3.1 (Abstracciones fundamentales) pero aquí se explica más detalladamente.

El patrón de diseño **Iterator** se utiliza para acceder secuencialmente a los objetos de una colección. *FigureEnumeration* juega el papel de *IFIterator* del patrón de diseño *Iterator*. *FigureEnumerator* y *ReverseFigureEnumerator*. Las clases *FigureEnumerator* y *ReverseFigureEnumerator* juegan el papel *Iterator*. El interface *Figure* juega el papel de *IFColeccion*. La clase abstracta *AbstractFigure* juega el papel de *Coleccion*.



Utilización del patrón Iterator.

El interface FigureEnumeration es un interface de Enumeration (hereda del interface *IFIterator* java.util Enumeration) que accede a Figures. Proporciona un método nextFigure, que se esconde bajo la función del código del cliente.

```

/*
 * @(#)FigureEnumeration.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.util.*;

/**
 * Interface for Enumerations that access Figures.
 * It provides a method nextFigure, that hides the down casting
 * from client code.
 */
public interface FigureEnumeration extends Enumeration {
    /**
     * Returns the next element of the enumeration. Calls to this
     * method will enumerate successive elements.
     * @exception NoSuchElementException If no more elements exist.
     */
    public Figure nextFigure();
}

```

La clase FigureEnumerator es una Enumeration para un Vector de Figures.

```

/*
 * @(#)FigureEnumerator.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.framework.*;
import java.util.*;

/**
 * An Enumeration for a Vector of Figures.
 */
public final class FigureEnumerator implements FigureEnumeration {
    Enumeration fEnumeration;

    public FigureEnumerator(Vector v) {
        fEnumeration = v.elements();
    }

    /**
     * Returns true if the enumeration contains more elements; false
     * if its empty.
     */
    public boolean hasMoreElements() {
        return fEnumeration.hasMoreElements();
    }

    /**
     * Returns the next element of the enumeration. Calls to this
     * method will enumerate successive elements.
     * @exception NoSuchElementException If no more elements exist.
     */
    public Object nextElement() {
        return fEnumeration.nextElement();
    }

    /**
     * Returns the next element of the enumeration. Calls to this
     * method will enumerate successive elements.
     * @exception NoSuchElementException If no more elements exist.
     */
    public Figure nextFigure() {
        return (Figure)fEnumeration.nextElement();
    }
}

```

ReverseFigureEnumeration es un Enumeration que enumera un vector de figuras desde atrás (size-1) hacia delante (0). Esta clase utiliza la clase ReverseVectorEnumerator que es una clase que implementa el **IFIterator** java.util.Enumeration.

```

/*

```



```

* @(#)ReverseFigureEnumerator.java 5.1
*
*/

package CH.ifa.draw.standard;

import java.util.*;
import CH.ifa.draw.util.ReverseVectorEnumerator;
import CH.ifa.draw.framework.*;

/**
 * An Enumeration that enumerates a vector of figures back (size-1) to
 * front (0).
 */
public final class ReverseFigureEnumerator implements
FigureEnumeration {
    ReverseVectorEnumerator fEnumeration;

    public ReverseFigureEnumerator(Vector v) {
        fEnumeration = new ReverseVectorEnumerator(v);
    }

    /**
     * Returns true if the enumeration contains more elements; false
     * if its empty.
     */
    public boolean hasMoreElements() {
        return fEnumeration.hasMoreElements();
    }

    /**
     * Returns the next element of the enumeration. Calls to this
     * method will enumerate successive elements.
     * @exception NoSuchElementException If no more elements exist.
     */
    public Object nextElement() {
        return fEnumeration.nextElement();
    }

    /**
     * Returns the next element casted as a figure of the enumeration.
     * Calls to this method will enumerate successive elements.
     * @exception NoSuchElementException If no more elements exist.
     */
    public Figure nextFigure() {
        return (Figure)fEnumeration.nextElement();
    }
}

```

```

/*
 * @(#)Figure.java 5.1
 *
*/

package CH.ifa.draw.framework;

import CH.ifa.draw.util.*;

```

```
import java.awt.*;
import java.util.*;
import java.io.Serializable;

/**
 * The interface of a graphical figure. A figure knows
 * its display box and can draw itself. A figure can be
 * composed of several figures. To interact and manipulate
 * with a figure it can provide Handles and Connectors.<p>
 * A figure has a set of handles to manipulate its shape or
 * attributes.
 * A figure has one or more connectors that define how
 * to locate a connection point.<p>
 * Figures can have an open ended set of attributes.
 * An attribute is identified by a string.<p>
 * Default implementations for the Figure interface are provided
 * by AbstractFigure.
 *
 * @see Handle
 * @see Connector
 * @see AbstractFigure
 */

public interface Figure
    extends Storable, Cloneable, Serializable {

    .....

    /**
     * Checks whether the given figure is contained in this figure.
     */
    public boolean includes(Figure figure);

    /**
     * Decomposes a figure into its parts. A figure is considered
     * as a part of itself.
     */
    public FigureEnumeration decompose();

    .....
}
```

```
/*
 * @(#)AbstractFigure.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

import java.awt.*;
import java.util.*;
import java.io.*;
```

```

/**
 * AbstractFigure provides default implementations for
 * the Figure interface.
 *
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld036.htm>Template Method</a></b><br>
 * Template Methods implement default and invariant behavior for
 * figure subclasses.
 * <hr>
 *
 * @see Figure
 * @see Handle
 */

public abstract class AbstractFigure implements Figure {

    .....

    /**
     * Returns an Enumeration of the figures contained in this figure.
     * @see CompositeFigure
     */
    public FigureEnumeration figures() {
        Vector figures = new Vector(1);
        figures.addElement(this);
        return new FigureEnumerator(figures);
    }

    .....

    /**
     * Decomposes a figure into its parts. It returns a Vector
     * that contains itself.
     * @return an Enumeration for a Vector with itself as the
     * only element.
     */
    public FigureEnumeration decompose() {
        Vector figures = new Vector(1);
        figures.addElement(this);
        return new FigureEnumerator(figures);
    }

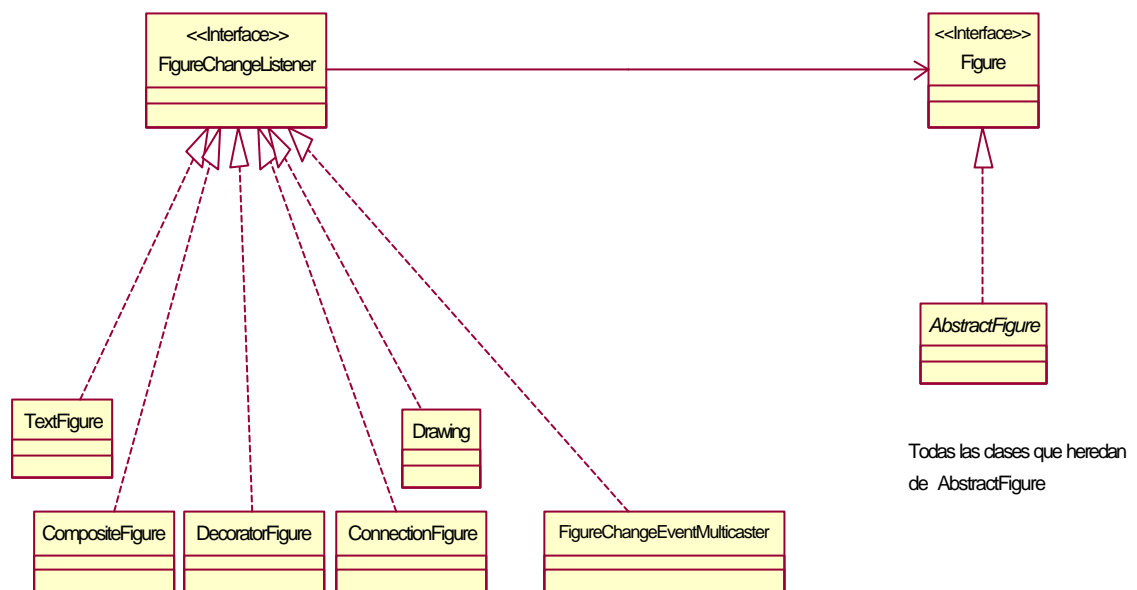
    .....
}

```

PATRÓN OBSERVER

Este patrón se utiliza en Drawing, DrawingView y ConnectionFigure y ya se ha hablado de él en el apartado 3.5.3.3 (Actualización de pantalla). En este apartado se hace el análisis de la utilización de este patrón de otra forma.

El patrón de diseño **Observer** es utilizado para desacoplar el Drawing de sus vistas y capacitar múltiples vistas.



Jerarquía FigureChangeListener.

El interface FigureChangeListener es un escuchador interesado en los cambios sobre Figure.

```

/*
 * @(#)FigureChangeListener.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.Rectangle;
import java.util.EventListener;

/**
 * Listener interested in Figure changes.
 */
public interface FigureChangeListener extends EventListener {

    /**
     * Sent when an area is invalid
     */
    public void figureInvalidated(FigureChangeEvent e);

    /**
     * Sent when a figure changed
     */
    public void figureChanged(FigureChangeEvent e);

    /**
     * Sent when a figure was removed
     */
}

```

```

    */
    public void figureRemoved(FigureChangeEvent e);

    /**
     * Sent when requesting to remove a figure.
     */
    public void figureRequestRemove(FigureChangeEvent e);

    /**
     * Sent when an update should happen.
     */
    public void figureRequestUpdate(FigureChangeEvent e);
}

```

Drawing envía los eventos DrawingChanged a DrawingChangeListeners cuando una parte de su area fue modificada.

```

/*
 * @(#)Drawing.java 5.1
 *
 */

package CH.ifa.draw.framework;

import CH.ifa.draw.util.*;
import java.awt.*;
import java.util.*;
import java.io.*;

/**
 * Drawing is a container for figures.
 * <p>
 * Drawing sends out DrawingChanged events to DrawingChangeListeners
 * whenever a part of its area was invalidated.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld026.htm>Observer</a></b><br>
 * The Observer pattern is used to decouple the Drawing from its views
 and
 * to enable multiple views.<hr>
 *
 * @see Figure
 * @see DrawingView
 * @see FigureChangeListener
 */

public interface Drawing
    extends Storable, FigureChangeListener, Serializable {

    .....

    /**
     * Adds a listener for this drawing.

```

```

    */
    public void addDrawingChangeListener(DrawingChangeListener
listener);

    /**
     * Removes a listener from this drawing.
     */
    public void removeDrawingChangeListener(DrawingChangeListener
listener);

    .....

    /**
     * Gets the listeners of a drawing.
     */
    public Enumeration drawingChangeListeners();

    /**
     * Invalidates a rectangle and merges it with the
     * existing damaged area.
     */
    public void figureInvalidated(FigureChangeEvent e);

    /**
     * Forces an update of the drawing change listeners.
     */
    public void figureRequestUpdate(FigureChangeEvent e);

    /**
     * Handles a removeFrfigureRequestRemove request that
     * is passed up the figure container hierarchy.
     * @see FigureChangeListener
     */
    public void figureRequestRemove(FigureChangeEvent e);

    /**
     * Acquires the drawing lock.
     */
    public void lock();

    /**
     * Releases the drawing lock.
     */
    public void unlock();
}

```

FigureChangeEventMulticaster maneja una lista de FigureChangeListeners que serán notificados de específicos FigureChangeEvents.

```

/*
 * @(#)FigureChangeEventMulticaster.java 5.1
 *
 */
package CH.ifa.draw.standard;

```

```
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;

/**
 * Manages a list of FigureChangeListeners to be notified of
 * specific FigureChangeEvents.
 */

public class FigureChangeEventMulticaster extends
    AWTEventMulticaster implements FigureChangeListener {

    public FigureChangeEventMulticaster(EventListener a, EventListener
b) {
        super(a, b);
    }

    public void figureInvalidated(FigureChangeEvent e) {
        ((FigureChangeListener)a).figureInvalidated(e);
        ((FigureChangeListener)b).figureInvalidated(e);
    }

    public void figureRequestRemove(FigureChangeEvent e) {
        ((FigureChangeListener)a).figureRequestRemove(e);
        ((FigureChangeListener)b).figureRequestRemove(e);
    }

    public void figureRequestUpdate(FigureChangeEvent e) {
        ((FigureChangeListener)a).figureRequestUpdate(e);
        ((FigureChangeListener)b).figureRequestUpdate(e);
    }

    public void figureChanged(FigureChangeEvent e) {
        ((FigureChangeListener)a).figureChanged(e);
        ((FigureChangeListener)b).figureChanged(e);
    }

    public void figureRemoved(FigureChangeEvent e) {
        ((FigureChangeListener)a).figureRemoved(e);
        ((FigureChangeListener)b).figureRemoved(e);
    }

    public static FigureChangeListener add(FigureChangeListener a,
FigureChangeListener b) {
        return (FigureChangeListener)addInternal(a, b);
    }

    public static FigureChangeListener remove(FigureChangeListener l,
FigureChangeListener oldl) {
        return (FigureChangeListener) removeInternal(l, oldl);
    }
}
```

```

protected EventListener remove(EventListener old1)
{
    if (old1 == a)
        return b;
    if (old1 == b)
        return a;
    EventListener a2 = removeInternal((FigureChangeListener)a,
old1);
    EventListener b2 = removeInternal((FigureChangeListener)b,
old1);
    if (a2 == a && b2 == b)
        return this;
    else
        return addInternal((FigureChangeListener)a2,
(FigureChangeListener)b2);
}

protected static EventListener addInternal(FigureChangeListener a,
FigureChangeListener b) {
    if (a == null) return b;
    if (b == null) return a;
    return new FigureChangeEventMulticaster(a, b);
}

protected static EventListener removeInternal(EventListener l,
EventListener old1) {
    if (l == old1 || l == null) {
        return null;
    } else if (l instanceof FigureChangeEventMulticaster) {
        return ((FigureChangeEventMulticaster)l).remove(old1);
    } else {
        return l;    // it's not here
    }
}
}

```

```

/*
 * @(#)Figure.java 5.1
 *
 */

package CH.ifa.draw.framework;

import CH.ifa.draw.util.*;
import java.awt.*;
import java.util.*;
import java.io.Serializable;

/**
 * The interface of a graphical figure. A figure knows
 * its display box and can draw itself. A figure can be
 * composed of several figures. To interact and manipulate
 * with a figure it can provide Handles and Connectors.<p>
 * A figure has a set of handles to manipulate its shape or
 * attributes.
 */

```



```

* A figure has one or more connectors that define how
* to locate a connection point.<p>
* Figures can have an open ended set of attributes.
* An attribute is identified by a string.<p>
* Default implementations for the Figure interface are provided
* by AbstractFigure.
*
* @see Handle
* @see Connector
* @see AbstractFigure
*/

public interface Figure
    extends Storable, Cloneable, Serializable {

    .....

    /**
     * Sets the Figure's container and registers the container
     * as a figure change listener. A figure's container can be
     * any kind of FigureChangeListener. A figure is not restricted
     * to have a single container.
     */
    public void addToContainer(FigureChangeListener c);

    /**
     * Removes a figure from the given container and unregisters
     * it as a change listener.
     */
    public void removeFromContainer(FigureChangeListener c);

    /**
     * Gets the Figure's listeners.
     */
    public FigureChangeListener listener();

    /**
     * Adds a listener for this figure.
     */
    public void addFigureChangeListener(FigureChangeListener l);

    /**
     * Removes a listener for this figure.
     */
    public void removeFigureChangeListener(FigureChangeListener l);

    /**
     * Releases a figure's resources. Release is called when
     * a figure is removed from a drawing. Informs the listeners that
     * the figure is removed by calling figureRemoved.
     */
    public void release();

    /**
     * Invalidates the figure. This method informs its listeners
     * that its current display box is invalid and should be
     * refreshed.
     */
}

```

```

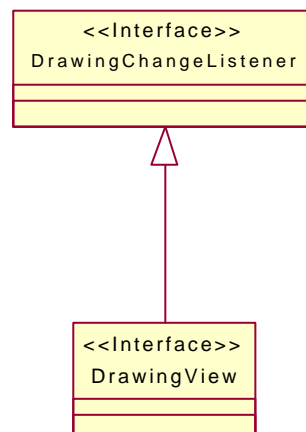
public void invalidate();

/**
 * Informes that a figure is about to change such that its
 * display box is affected.
 * Here is an example of how it is used together with changed()
 * <pre>
 * public void move(int x, int y) {
 *     willChange();
 *     // change the figure's location
 *     changed();
 * }
 * </pre>
 * @see #invalidate
 * @see #changed
 */
public void willChange();

/**
 * Informes that a figure has changed its display box.
 * This method also triggers an update call for its
 * registered observers.
 * @see #invalidate
 * @see #willChange
 */
public void changed();

.....
}

```



Jerarquía DrawingChangeListener.

El interface DrawingChangeListener es un escuchador interesado en los cambios del Drawing.

```

/*
 * @(#)DrawingChangeListener.java 5.1
 *
 */

```

```

package CH.ifa.draw.framework;

import java.awt.Rectangle;
import java.util.EventListener;

/**
 * Listener interested in Drawing changes.
 */
public interface DrawingChangeListener extends EventListener {

    /**
     * Sent when an area is invalid
     */
    public void drawingInvalidated(DrawingChangeEvent e);

    /**
     * Sent when the drawing wants to be refreshed
     */
    public void drawingRequestUpdate(DrawingChangeEvent e);
}

```

DrawingView observa drawings que cambian a través del interface DrawingChangeListener.

```

/*
 * @(#)DrawingView.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.image.ImageObserver;
import java.awt.*;
import java.util.*;
import java.io.*;
import CH.ifa.draw.util.*;

/**
 * DrawingView renders a Drawing and listens to its changes.
 * It receives user input and delegates it to the current tool.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld026.htm>Observer</a></b><br>
 * DrawingView observes drawing for changes via the DrawingListener
 interface.<br>
 * 
 * <b><a href=../pattlets/sld032.htm>State</a></b><br>
 * DrawingView plays the role of the StateContext in
 the State pattern. Tool is the State.<br>
 * 
 * <b><a href=../pattlets/sld034.htm>Strategy</a></b><br>
 * DrawingView is the StrategyContext in the Strategy pattern
 with regard to the UpdateStrategy. <br>
 * DrawingView is the StrategyContext for the PointConstrainer.
 *
 * @see Drawing
 * @see Painter

```

```

* @see Tool
*/

public interface DrawingView extends ImageObserver,
DrawingChangeListener {

.....

    /**
     * Sets the current display update strategy.
     * @see UpdateStrategy
     */
    public void setDisplayUpdate(Painter updateStrategy);

.....

    /**
     * Checks whether the drawing has some accumulated damage
     */
    public void checkDamage();

    /**
     * Repair the damaged area
     */
    public void repairDamage();

    /**
     * Paints the drawing view. The actual drawing is delegated to
     * the current update strategy.
     * @see Painter
     */
    public void paint(Graphics g);

.....

    /**
     * Gets the background color of the DrawingView
     */
    public void setBackground(Color c);

    /**
     * Draws the contents of the drawing view.
     * The view has three layers: background, drawing, handles.
     * The layers are drawn in back to front order.
     */
    public void drawAll(Graphics g);

    /**
     * Draws the currently active handles.
     */
    public void drawHandles(Graphics g);

    /**
     * Draws the drawing.
     */
    public void drawDrawing(Graphics g);

```

```

/**
 * Draws the background. If a background pattern is set it
 * is used to fill the background. Otherwise the background
 * is filled in the background color.
 */
public void drawBackground(Graphics g);

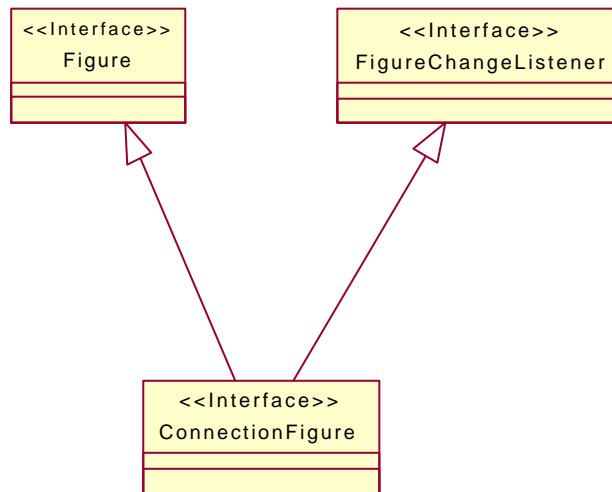
.....

/**
 * Freezes the view by acquiring the drawing lock.
 * @see Drawing#lock
 */
public void freezeView();

/**
 * Unfreezes the view by releasing the drawing lock.
 * @see Drawing#unlock
 */
public void unfreezeView();
}

```

El patrón Observer también se utiliza en el interface ConnectionFigure para ver los cambios de las figuras conectadas. Una figura connection registra ella misma los escuchadores (listeners) y observadores (observers) de los conectores fuente y destino. Si se mueve alguna de las figuras a las que está conectada la línea escucha los ChangeEvent y se actualiza.



Jerarquía ConnectionFigure.

ConnectionFigure es Figure por lo cual se pueden mover, tienen handles, etc. Las Figures que conectan Connectors proporcionan Figures. Un ConnectionFigure conoce su Connector del principio y del final. Utiliza los Connectors para localizar sus puntos de conexión.

```

/*
 * @(#)ConnectionFigure.java 5.1
 *
 */

```

```

package CH.ifa.draw.framework;

import java.awt.Point;
import java.io.Serializable;
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

/**
 * Figures to connect Connectors provided by Figures.
 * A ConnectionFigure knows its start and end Connector.
 * It uses the Connectors to locate its connection points.<p>
 * A ConnectionFigure can have multiple segments. It provides
 * operations to split and join segments.
 *
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld034.htm>Strategy</a></b><br>
 * Strategy is used encapsulate the algorithm to locate the connection
 * point.
 * ConnectionFigure is the Strategy context and Connector is the
 * Strategy.<br>
 * 
 * <b><a href=../pattlets/sld026.htm>Observer</a></b><br>
 * Observer is used to track changes of the connected figures. A
 * connection connection figure registers itself as listeners or
 * observers of the source and target connector.
 * <hr>
 *
 * @see Connector
 */

public interface ConnectionFigure extends Figure, FigureChangeListener
{

    /**
     * Sets the start Connector of the connection.
     * @param figure the start figure of the connection
     */
    public void connectStart(Connector start);

    /**
     * Sets the end Connector of the connection.
     * @param figure the end figure of the connection
     */
    public void connectEnd(Connector end);

    /**
     * Updates the connection
     */
    public void updateConnection();

    /**
     * Disconnects the start figure from the dependent figure
     */
    public void disconnectStart();

    /**

```

```

    * Disconnects the end figure from the dependent figure
    */
    public void disconnectEnd();

    /**
     * Gets the start Connector
     */
    public Connector start();

    /**
     * Gets the end Connector.
     */
    public Connector end();

    /**
     * Checks if two figures can be connected. Implement this method
     * to constrain the allowed connections between figures.
     */
    public boolean canConnect(Figure start, Figure end);

    /**
     * Checks if the ConnectionFigure connects the same figures.
     */
    public boolean connectsSame(ConnectionFigure other);

    .....
}

```

3.5.3.8. Conclusiones

- ◆ Los patrones establecieron las estructuras del diseño del JHotDraw. Si se unen todos los patrones de diseño utilizados forman casi la totalidad de la aplicación.
- ◆ No siempre es inmediato saber que patrón utilizar.
 - A veces varios son aplicables (Prototype, Factory Method).
 - Implementar los patrones es fácil; lo difícil es saber cuando y dónde hay que implementarlos.
- ◆ El diseño sigue siendo iterativo.
- ◆ Los patrones ayudan en la documentación (pattlets). Para explicar el interface Locator basta con decir que juega el papel de *StrategyAbstracto* del patrón de diseño Strategy.
- ◆ Los patrones de diseño y Java son una combinación muy potente **“pero ningún lenguaje puede reducir la importancia del diseño”**. Kent Beck y Erich Gamma.
- ◆ Java se adapta muy bien a los patrones de diseño.

La documentación de un framework tiene tres propósitos, y los patrones de diseño te ayudan a cumplir cada uno de ellos. La documentación debe describir:

- El propósito del framework.
- Cómo utilizar el framework.

- Los detalles del diseño del framework.

Los patrones de diseño son la forma más apropiada para enseñar como utilizar un framework, un conjunto de patrones de diseño pueden reunir los tres propósitos para documentar un framework.

El código completo del JHotDraw y sus ejemplos se muestra en el Anexo E.

3.5.4. Funcionamiento de la aplicación

Como parte del framework JHotDraw se incluyen un número de ejemplos de aplicaciones y applets que demuestran varios aspectos del JHotDraw.

3.5.4.1. JavaDrawApp

Este ejemplo ilustra varias herramientas y figuras estándar proporcionadas con el JHotDraw.

Una descripción de sus características sería la siguiente:

JavaDraw proporciona herramientas, manejadores, y comandos para manipular un dibujo.

Herramientas

Las herramientas que están a la izquierda crean o manipulan figuras.



Selecciona, mueve o cambia de tamaño una figura; la tecla shift mantiene la selección. Arrastrando la herramienta de selección por el fondo seleccionas las figuras que están dentro del rectángulo que se forma al arrastrar el ratón.



Crea una nueva o edita una figura de texto existente.



Crea una nueva figura de texto y la adjunta a la figura seleccionada. Una vez que la figura de texto esta adjuntada permanecerá conectada a la figura.



Edita un atributo URL de una figura; la URL así asociada con una figura se seguirá cuando la figura es seleccionada en el applet viewer.



Crea una figura rectángulo.



Crea una figura rectángulo redondeado.



Crea una figura elipse.



Crea una figura línea.



Crea o ajusta las conexiones entre figuras. Una conexión se separa en segmentos arrastrando un punto de la conexión. Para unir dos segmentos hacer clic en el punto final de un segmento.



Crea una conexión en forma de codo, segmentos horizontales y verticales, y ofrece las mismas características que la herramienta de conexión ordinaria.



Crea una figura garabato; arrastrando el ratón con el botón pulsado crea una línea, mientras que haciendo clicks con el ratón se produce una figura de una línea de muchos segmentos (polyline).



Crea una figura polígono.

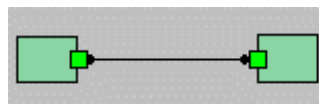


Decora la figura pulsada con el ratón con un borde.

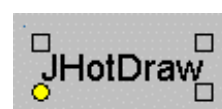
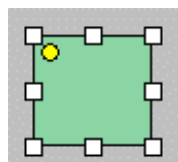
Manejadores

Cuando las figuras están seleccionadas se proporcionan diferentes manejadores para manipularlas.

- Los manejadores rellenos de blanco permiten la manipulación de la forma de una figura.
- Los manejadores de un círculo azul es un manejador de conexión. Permite crear conexiones entre figuras.
- Los manejadores enmarcados de negro son manejadores null. Ellos solamente muestran el estado de selección de una figura pero no soportan ninguna manipulación.
- Los manejadores de verde se muestran sobre las conexiones. Permiten reconectarlas con otras figuras.



- Los manejadores de amarillo permiten cambiar el tamaño de la fuente de una figura de texto o el tamaño del radio de un rectángulo redondeado.



Comandos

Los comandos son invocados desde la barra de botones en la parte inferior (applet) o desde la barra del menú en la parte de arriba (aplicación).

Los comandos estándar que son proporcionados son: cut, copy, paste, duplicate, delete, group, ungroup, bring to front, and send to back.

Atributos

Los atributos, tales como color de relleno, color del texto o punta de la flecha, pueden cambiarse a través del menú Attributes (aplicación) o de los menús popup (applet).

La animación que soporta el JavaDraw es adjuntada a una figura con un AnimationDecorator.

Soporta adjuntar URLs a las figuras lo cual es implementado por URLTool.

La aplicación se puede ejecutar así:

```
java CH.ifa.draw.samples.javadraw.JavaDrawApp
```

Esta línea se ha puesto en el fichero *run.bat*, así pues basta con ejecutar dicho fichero bajo MS-DOS.

Cuando se ha ejecutado la aplicación se verá en pantalla una ventana similar a la tercera figura que aparece en el apartado 3.5.1.

El directorio *drawings* incluye algunos ejemplos de dibujos. Estos dibujos se pueden cargar con el comando Open del menú File. Tu CLASSPATH tiene que incluir el path absoluto del directorio del JHotDraw de lo contrario cuando lo ejecutes tendrás problemas al cargar las clases.

3.5.4.2. JavaDrawApplet

El JavaDrawApplet es la versión applet del JavaDrawApp. Cuando es un applet el dibujo no se puede salvar. Un ejemplo de dibujo puede ser cargado desde el ítem de elección (choice) de carga del applet.

El applet se puede ejecutar abriendo la página *JavaDrawApplet.html* en el Internet Explorer con compilador de java, ya que si se abre con el Netscape Communicator no se verá la paleta de herramientas. También se puede abrir con el appletviewer:

```
appletviewer JavaDrawApplet.html
```

Esta línea se ha puesto en el fichero *runapplet.bat*, así pues basta con ejecutar dicho fichero bajo MS-DOS.

Cuando se ha ejecutado la aplicación se verá en pantalla una ventana similar a la cuarta figura que aparece en el apartado 3.5.1.

3.5.4.3. JavaDrawViewer

El JavaDrawViewer permite ver un dibujo JavaDraw. Si una figura tiene una URL adjunta puedes seguirla seleccionando la figura con el ratón.

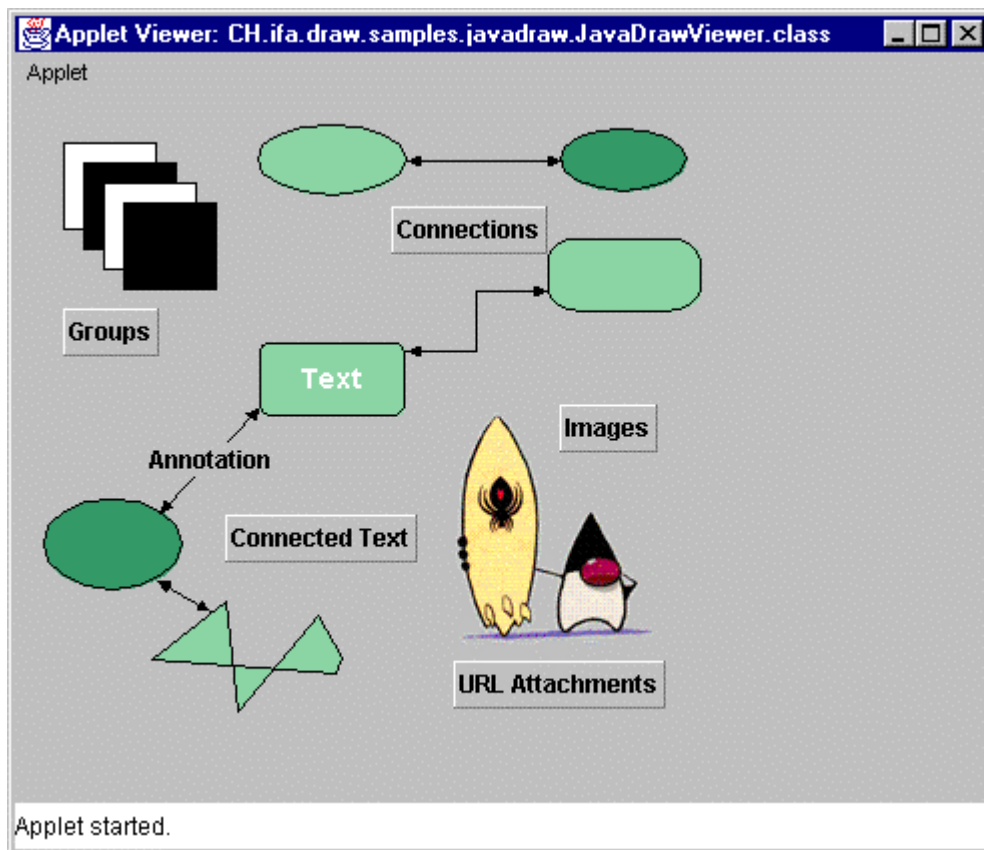
El ejemplo ilustra como crear una presentación mínima sin una paleta de herramientas ni otros adornos del interface de usuario.

El applet se puede ejecutar abriendo la página *JavaDrawViewer.html* en el Internet Explorer con compilador de java, ya que si se abre con el Netscape Communicator no se verá nada ya que se producirá un error al ejecutar el applet. También se puede abrir con el `appletviewer`:

appletviewer JavaDrawViewer.html

Esta línea se ha puesto en el fichero *runviewerapplet.bat*, así pues basta con ejecutar dicho fichero bajo MS-DOS.

Cuando se ha ejecutado la aplicación se verá en pantalla una ventana similar a esta:.



Forma del applet JavaDrawViewer al ejecutarlo.

3.5.4.4. PertApp

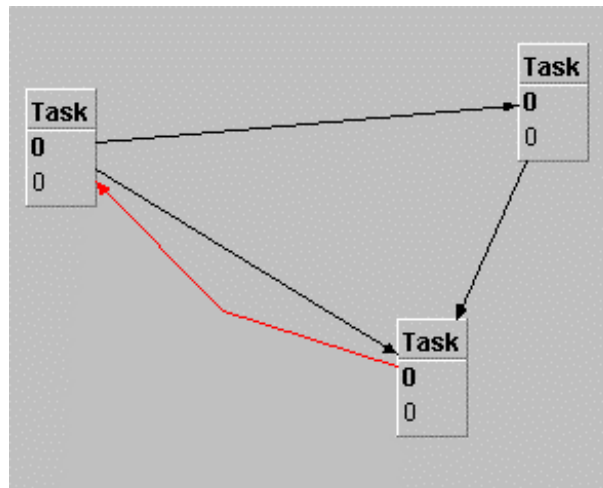
PertApp es un simple editor de dibujo que soporta el manejo de proyectos PERT. Con PertApp puedes crear tareas y definir sus dependencias. PertApp tiene el cuidado de rastrear el tiempo final de cada tarea.

Una descripción de sus características sería la siguiente.

PERT proporciona herramientas, manejadores, y comandos para manipular un diagrama PERT.

Una tarea se muestra como una caja con una línea para el nombre de la tarea, una línea para la duración de la tarea, y una línea para el tiempo final más temprano de la tarea. Solamente el nombre de la tarea y su duración son editables, es decir se pueden cambiar. El tiempo final es calculado automáticamente por el PERT.

Las dependencias circulares se muestran en rojo.



Relación circular en rojo.

Herramientas

La paleta de herramientas que esta a la izquierda ofrece varias herramientas que crean nuevas figuras o manipulan las existentes.



Selecciona, mueve o cambia de tamaño una figura; la techa shift mantiene la selección. Arrastrando la herramienta de selección por el fondo seleccionas las figuras que están dentro del rectángulo que se forma al arrastrar el ratón.



Crea una nueva tarea.



Crea una nueva o edita una figura de texto existente.



Crea o ajusta las conexiones entre figuras. Una conexión se separa en segmentos arrastrando un punto de la conexión. Para unir dos segmentos hacer clic en el punto final de un segmento.

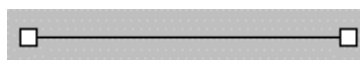


Crea una figura linea.

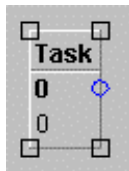
Manejadores

Cuando las figuras están seleccionadas se proporcionan diferentes manejadores para manipularlas.

- Los manejadores rellenados de blanco permiten la manipulación de la forma de una figura.



- Los manejadores de un círculo azul es un manejador de conexión. Permite crear conexiones entre figuras.
- Los manejadores enmarcados de negro son manejadores null. Ellos solamente muestran el estado de selección de una figura pero no soportan ninguna manipulación.



- Los manejadores de verde se muestran sobre las conexiones. Permiten reconectarlas con otras figuras.
- Los manejadores de amarillo permiten cambiar el tamaño de la fuente de una figura de texto o el tamaño del radio de un rectángulo redondeado.

Comandos

Los comandos son invocados desde la barra de botones en la parte inferior (applet) o desde la barra del menú en la parte de arriba (aplicación).

Los comandos estándar que son proporcionados son: cut, copy, paste, duplicate, delete, group, ungroup, bring to front, and send to back.

Atributos

Los atributos, tales como color de relleno, color del texto o punta de la flecha, pueden cambiarse a través del menú Attributes (aplicación) o de los menús popup (applet).

Este ejemplo ilustra como crear figuras más complejas y como utilizar las conexiones. Las conexiones pueden ser creadas con un ConnectionTool o con ConnectionHandles.

La aplicación se puede ejecutar así:

```
java CH.ifa.draw.samples.pert.PertApplication
```

Esta línea se ha puesto en el fichero *runpert.bat*, así pues basta con ejecutar dicho fichero bajo MS-DOS.

Cuando se ha ejecutado la aplicación se verá en pantalla una ventana similar a la segunda figura que aparece en el apartado 3.5.1.

El directorio *drawings* incluye algunos ejemplos de dibujos. Estos dibujos se pueden cargar con el comando Open del menú File. Tu CLASSPATH tiene que incluir el path absoluto del directorio del JHotDraw de lo contrario cuando lo ejecutes tendrás problemas al cargar las clases.

3.5.4.5. PertApplet

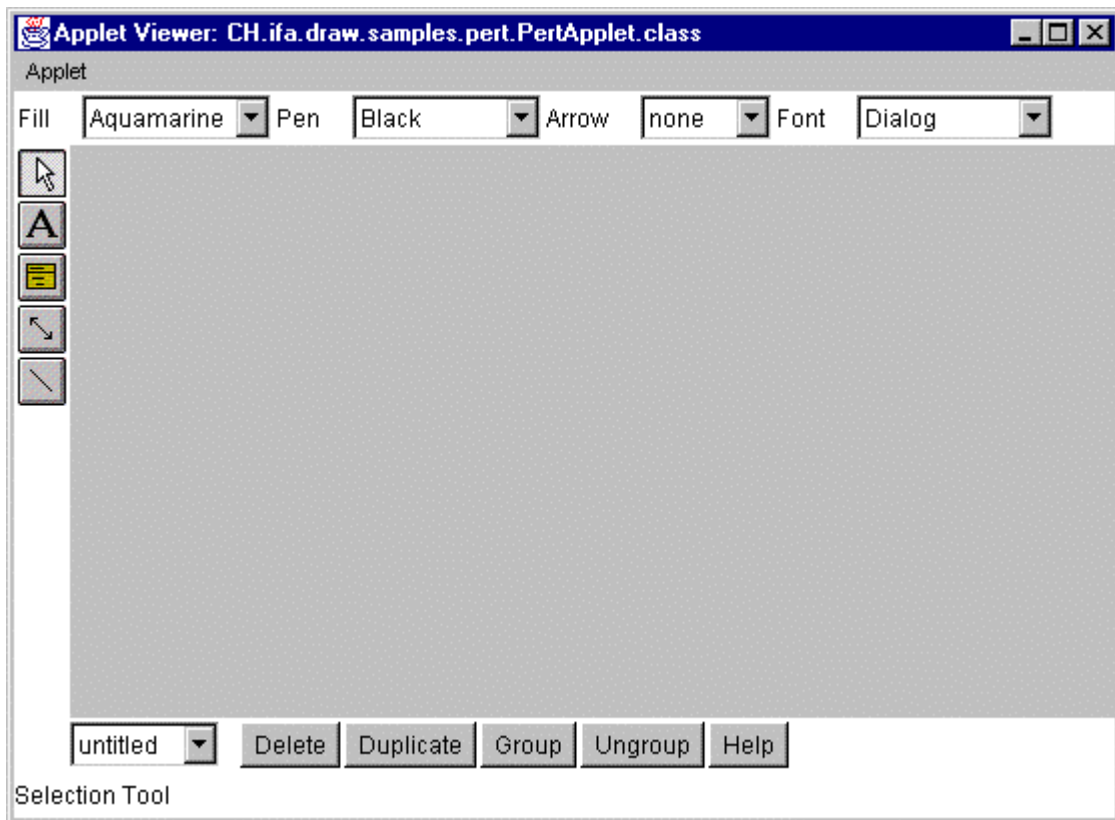
El PertApplet es la versión applet del PertApp. Cuando es un applet el dibujo no se puede salvar. Un ejemplo de dibujo puede ser cargado desde el ítem de elección (choice) de carga del applet.

El applet se puede ejecutar abriendo la página *PertApplet.html* en el Internet Explorer con compilador de java, ya que si se abre con el Netscape Communicator no se verá la paleta de herramientas. También se puede abrir con el appletviewer:

appletviewer PertApplet.html

Esta línea se ha puesto en el fichero *runpertapplet.bat*, así pues basta con ejecutar dicho fichero bajo MS-DOS.

Cuando se ha ejecutado la aplicación se verá en pantalla una ventana similar a esta:



Forma del applet Pert al ejecutarlo.

3.5.4.6. NothingApp

Este es el ejemplo más sencillo de JhotDraw. Este ejemplo puede ser utilizado como punto para empezar el desarrollo de nuestra propia aplicación. Los ejemplos utilizan unas

herramientas estándar para crear un simple editor de dibujo. Esto solamente es cerca de una página de código:

```

/*
 * @(#)NothingApp.java 5.1
 *
 */

package CH.ifa.draw.samples.nothing;

import java.awt.*;
import java.util.*;
import java.io.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.contrib.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.application.*;

public class NothingApp extends DrawApplication {

    NothingApp() {
        super("Nothing");
    }

    protected void createTools(Palette palette) {
        super.createTools(palette);

        Tool tool = new TextTool(view(), new TextFigure());
        palette.add(createToolButton(IMAGES+"TEXT", "Text Tool",
tool));

        tool = new CreationTool(view(), new RectangleFigure());
        palette.add(createToolButton(IMAGES+"RECT", "Rectangle Tool",
tool));

        tool = new CreationTool(view(), new RoundRectangleFigure());
        palette.add(createToolButton(IMAGES+"RRECT", "Round Rectangle
Tool", tool));

        tool = new CreationTool(view(), new EllipseFigure());
        palette.add(createToolButton(IMAGES+"ELLIPSE", "Ellipse Tool",
tool));

        tool = new CreationTool(view(), new LineFigure());
        palette.add(createToolButton(IMAGES+"LINE", "Line Tool",
tool));

        tool = new PolygonTool(view());
        palette.add(createToolButton(IMAGES+"POLYGON", "Polygon Tool",
tool));

        tool = new ConnectionTool(view(), new LineConnection());
        palette.add(createToolButton(IMAGES+"CONN", "Connection Tool",
tool));
    }
}

```



```

        tool = new ConnectionTool(view(), new ElbowConnection());
        palette.add(createToolButton(IMAGES+"OCONN", "Elbow Connection
Tool", tool));
    }

    //-- main -----

    public static void main(String[] args) {
        DrawApplication window = new NothingApp();
        window.open();
    }
}

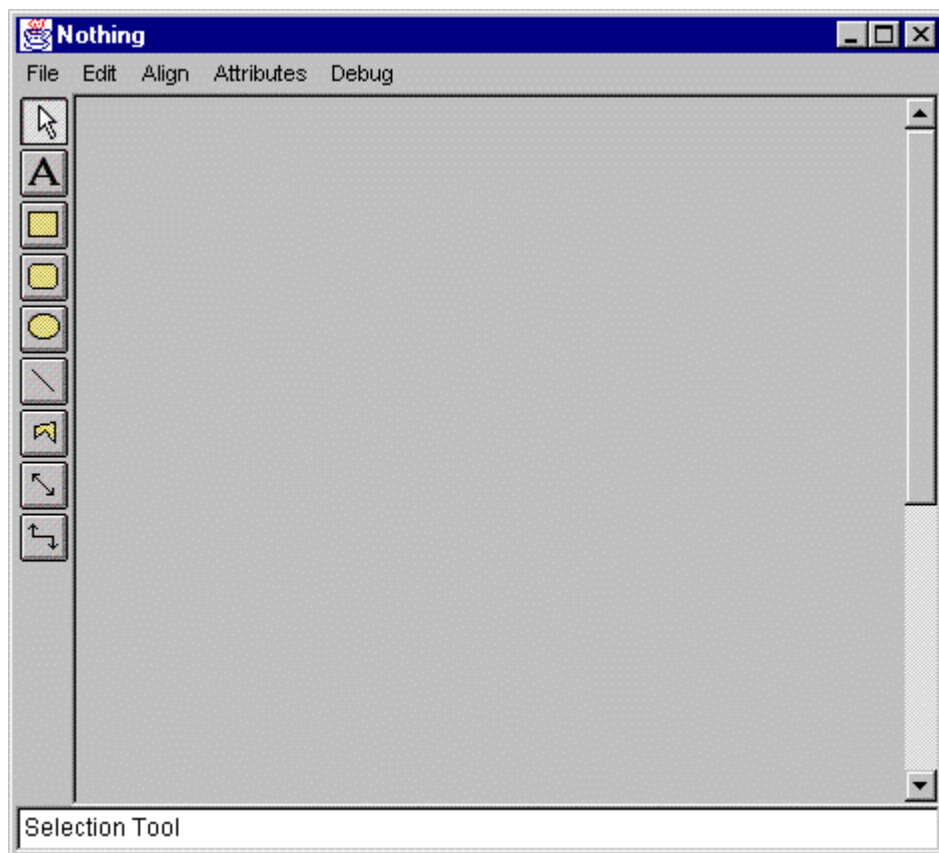
```

La aplicación se puede ejecutar así:

java CH.ifa.draw.samples.nothing.NothingApp

Esta línea se ha puesto en el fichero *runnothing.bat*, así pues basta con ejecutar dicho fichero bajo MS-DOS.

El directorio *drawings* incluye algunos ejemplos de dibujos. Estos dibujos se pueden cargar con el comando Open del menú File. Tu CLASSPATH tiene que incluir el path absoluto del directorio del JHotDraw de lo contrario cuando lo ejecutes tendrás problemas al cargar las clases. Cuando se ha ejecutado la aplicación se verá en pantalla una ventana similar a esta:



Forma de la aplicación Nothing al ejecutarla.

3.5.4.7. NothingApplet

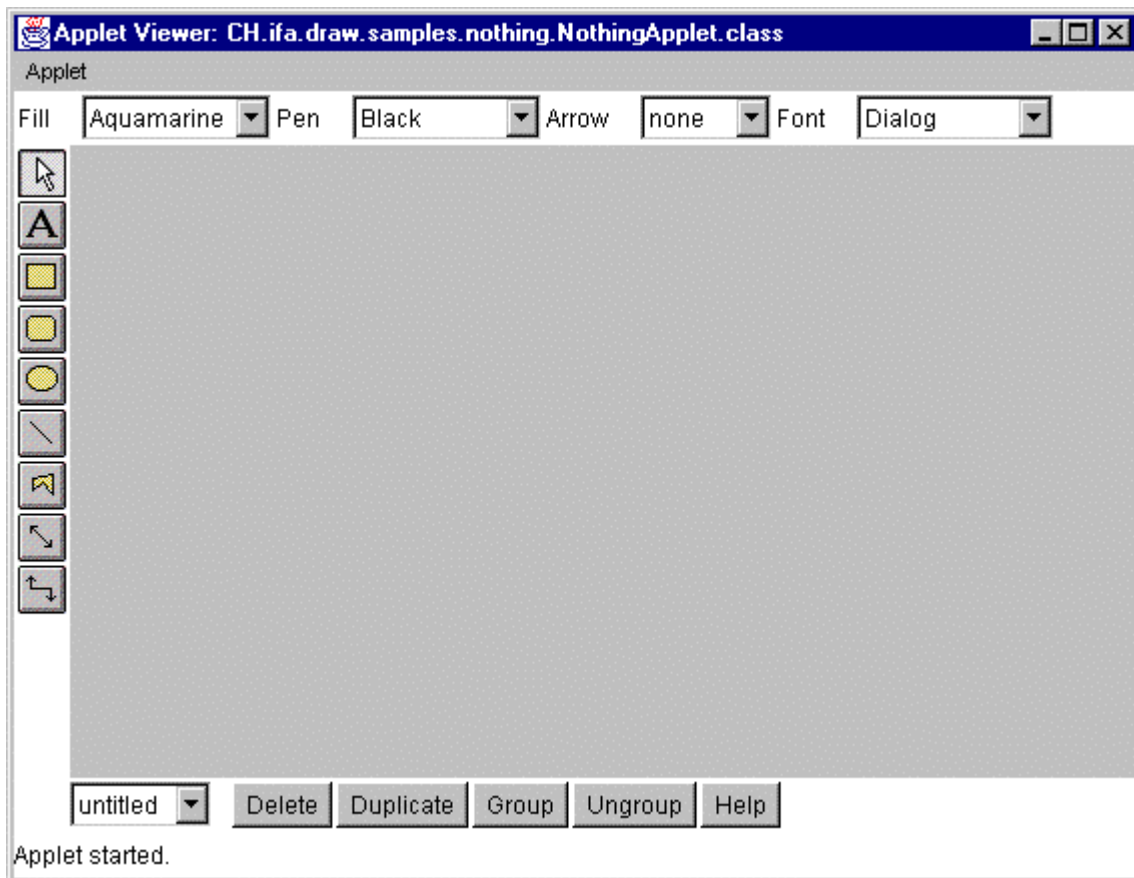
El NothingApplet es la versión applet del NothingApp. Cuando es un applet el dibujo no se puede salvar. Un ejemplo de dibujo puede ser cargado desde el ítem de elección (choice) de carga del applet.

El applet se puede ejecutar abriendo la página *NothingApplet.html* en el Internet Explorer con compilador de java, ya que si se abre con el Netscape Communicator no se verá la paleta de herramientas. También se puede abrir con el `appletviewer`:

appletviewer NothingApplet.html


Esta línea se ha puesto en el fichero *runnothingapplet.bat*, así pues basta con ejecutar dicho fichero bajo MS-DOS.

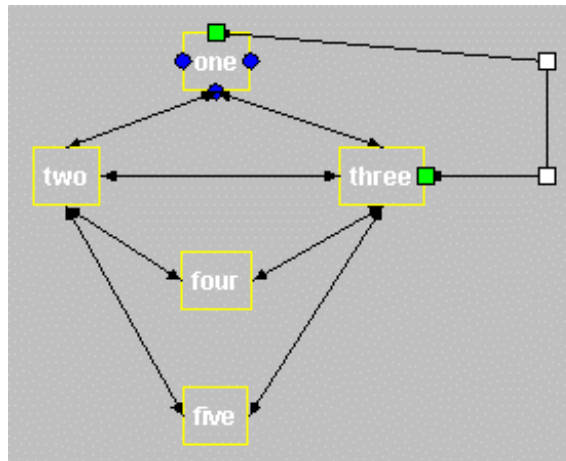
Cuando se ha ejecutado la aplicación se verá en pantalla una ventana similar a esta:



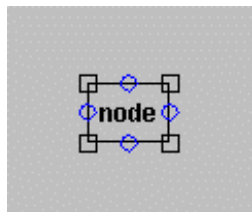
Forma del applet Nothing al ejecutarlo.

3.5.4.8. NetApp

NetApp es un simple editor de redes. Permite la creación de nodos . Cuando se selecciona un nodo se visualizan un conjunto de manejadores que pueden ser utilizados para conectarlo con otros nodos. Los nodos son conectados en localizaciones específicas. Las potenciales localizaciones son resaltadas cuando el ratón se mueve dentro de un nodo arrastrando una línea de conexión. Este ejemplo ilustra la utilización de LocatorConnectors.



Se esta arrastrando la conexión sobre el nodo.



El nodo esta seleccionado.

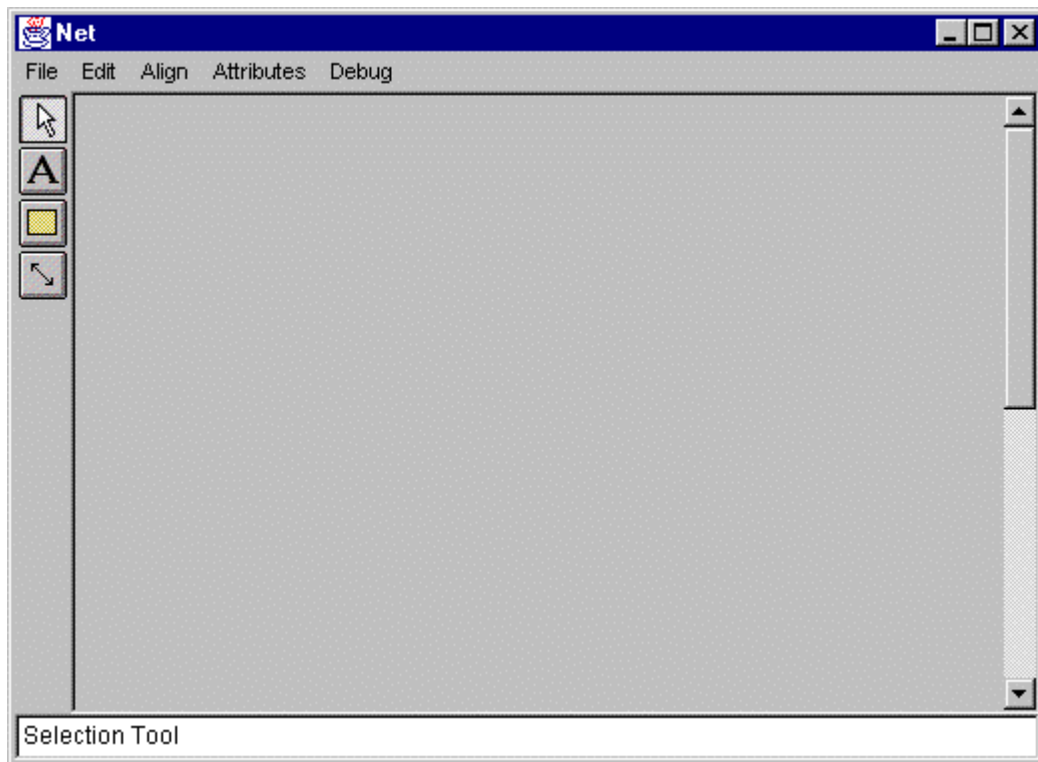
La aplicación se puede ejecutar así:

```
java CH.ifa.draw.samples.net.NetApp
```

Esta línea se ha puesto en el fichero *runnet.bat*, así pues basta con ejecutar dicho fichero bajo MS-DOS.

En el directorio *drawings* puedes encontrar un ejemplo net. Estos dibujos se pueden cargar con el comando Open del menú File. Tu CLASSPATH tiene que incluir el path absoluto del directorio del JHotDraw de lo contrario cuando lo ejecutes tendrás problemas al cargar las clases.

Cuando se ha ejecutado la aplicación se verá en pantalla una ventana similar a esta:



Forma de la aplicación Net al ejecutarla.

4. CONCLUSIONES FINALES

Este proyecto tiene como objetivo servir como punto de inicio para introducirse en el mundo de los **patrones de diseño**. La guía consta de una introducción teórica a los patrones. A continuación se realiza la explicación detallada de los patrones utilizados en los ejemplos.

Después de esta parte teórica comienza la aplicación de los patrones de diseño explicados anteriormente en ejemplos. Los ejemplos van avanzando en dificultad, es decir se empieza por aplicaciones sencillas. Los cuatro primeros ejemplos son aplicaciones muy sencillas, ya que el objetivo fundamental es comprender como, donde y porque se utilizan los patrones de diseño.

El último ejemplo, el más complejo de todos, es el **JHotDraw** que es un framework realizado por profesionales expertos en los patrones de diseño. El JHotDraw utiliza un gran número de patrones, por lo que la comprensión detallada llevará su tiempo. Este ejemplo es muy bueno, ya que los patrones han sido aplicados correctamente y contiene varias aplicaciones de un mismo patrón y bastantes patrones diferentes, y se trata de una aplicación de gran utilidad y directamente aplicable en el trabajo profesional.

Todas las aplicaciones realizadas están detalladamente explicadas. También se proporciona, en los diferentes anexos, el código completo de cada una de las aplicaciones para que la comprensión sea total.

El lenguaje de programación utilizado para la realización de este tutorial de patrones de diseño ha sido el lenguaje Java por las siguientes características:

- ◆ Java es un lenguaje de programación muy moderno, ha sido uno de los últimos lenguajes de programación que han aparecido.
- ◆ Java se ha convertido en muy poco tiempo en uno de los lenguajes más utilizados y que mayor futuro tienen.
- ◆ Java se adapta muy bien a los patrones de diseño.
- ◆ Los patrones de diseño y Java son una combinación muy potente.

La utilización de los patrones de diseño proporciona numerosas ventajas tales como:

- ◆ Los patrones establecen las estructuras del diseño de las aplicaciones.
- ◆ Los patrones de diseño proporcionan un diseño flexible, independiente y extensible.
- ◆ Los patrones de diseño separan de una clase unas características, es decir hacen una separación entre interfaz y diseño. Esta separación hace más fácil la creación o reutilización de código.
- ◆ El diseño sigue siendo iterativo.
- ◆ Los patrones ayudan en la documentación. Al utilizar patrones de diseño es más fácil documentar los detalles del diseño y como reutilizar la aplicación.

- ◆ Al utilizar patrones de diseño sabemos que son diseños muy efectivos y eficientes, ampliamente demostrados por la experiencia de otras personas y que nos ayudan a construir software correctamente.

Con la realización de este proyecto he aprendido muchas cosas entre las cuales destacó las siguientes:

- ◆ Aprendizaje de que son, para que sirven, donde se utilizan y como los patrones de diseño de diseño.
- ◆ Aplicación práctica de los patrones de diseño.
- ◆ Aprendizaje de que es, cómo se utiliza, porque tiene ese diseño un framework.
- ◆ Utilización conjunta framework-patrones de diseño.
- ◆ Comprensión y explicación del framework JHotDraw con todos los patrones de diseño que utiliza.
- ◆ Aprendizaje del lenguaje Java.

Así pues como conclusión final:

Los patrones de diseño son muy útiles y, si no lo son todavía demasiado, serán el futuro de la programación y la construcción de software.

5. BIBLIOGRAFÍA

“*A pattern Language: Towns/Building/Construction*” de Christopher Alexander, Sara Ishikawa, Murria silverstein, Max Jacobson, Ingrid Fiksdahl-King y Shlomo Angel, (1977), Oxford University Press. [AIS77].

“*The Timeless Way of Building*” de Christopher Alexander (1979), Oxford University Press.

“*Using Pattern Languages for Object-Oriented Programs*” de Ward Cunningham y Kent Beck (1987).

“*Advanced C++ Programming Styles and Idioms*” de Jim Coplien (1991).

“*Design Patterns: Elements of Reusable Object-Oriented Software*” (*Gang of Four*, [GoF]) de Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides (1994), Addison-Wesley Professional Computing Series. [GoF95].

“*Patterns and Software: Essential Concepts and Terminology*” de Brad Appleton.

“*Patterns in Java (volume 1)*” de Mark Grand (1998), Wiley Computer Publishing. [Grand98].

“*Patterns in Java (volume 2)*” de Mark Grand (1999), Wiley Computer Publishing.

“*Thinking in Patterns with Java*” de Bruce Eckel. President, MindView, Inc.

“*Anti Patterns. Refactoring software, Architectures and Proyects in Crisis*” de W. J. Brown, R. C. Malveau, H.W. “Skip” McCormick III, T. J. Mowbray (1998), Wiley Computer Publishing.

“*Java 1.2 al descubierto*” de Jaime Jaworski (1999), Prentice Hall.

“*Aprenda Java como si estuviera en primero*” de Javier García de Jalón, José Ignacio Rodríguez, Iñigo Mingo, Aitor Imaz, Alfonso Brazález, Alberto Larzabal, Jesús Calleja y Jon García (1999). Universidad de Navarra.

ANEXO A: Código completo en Java del apartado 3.1.4

La aplicación del apartado 3.1 (Applet para escuchar audio) consta de dos clases y una página web donde va inmerso el applet para poder ejecutar la aplicación:

- Clase ManejadorAudio.
- Clase AudioPlayer.
- Página web AudioPlayer

Clase ManejadorAudio

Se encuentra en el fichero ManejadorAudio.java y su contenido es el siguiente:

```
import java.applet.AudioClip;

/**
 * Esta clase es utilizada para evitar que dos audio clips
 * suenen a la vez. La clase tiene una sola instancia a la
 * que se puede acceder a través de su metodo getInstancia.
 * Cuando pones audio clips a traves de ese objeto, este detiene
 * el ultimo audio clip que se esta escuchando antes de empezar
 * el nuevo audio demandado. Si todos los audio clips se ponen
 * a través del objeto instancia entonces nunca habrá mas
 * de un audio clip sonando al mismo tiempo.
 */
public class ManejadorAudio implements AudioClip
{

    private static ManejadorAudio instancia=new ManejadorAudio();
    private AudioClip audioAnterior; //la petición anterior de audio

    /*****

    /**
     * Definimos un constructor privado, por lo tanto no se generará
un
     * constructor publico por defecto.
     */
    private ManejadorAudio()
    {
    }//ManejadorAudio()

    /*****
    ** /

    /**
```

```

    * Retorna una referencia a la unica instancia de la clase
    * ManejadorAudio
    * @return ManejadorAudio la unica instancia de la clase
    * ManejadorAudio
    */
    public static ManejadorAudio getInstancia()
    {
        return instancia;
    } //getInstancia()

    /*****

    /**
     * Empieza la ejecución del audio seleccionado. Cada vez que se
    llama
     * a este metodo el audio comienza desde el principio.
     */
    public void play()
    {
        if (audioAnterior!=null)
            audioAnterior.play();
    } //play()

    /*****

    /**
     * Para la ejecución del audio que se esta escuchando y empieza el
     * audio pasado como parametro.
     * @param audio el nuevo audio a escuchar.
     */
    public void play(AudioClip audio)
    {
        if (audioAnterior!=null)
            audioAnterior.stop();
        if (audio!=null)
        {
            audioAnterior=audio;
            audio.play();
        }
    } //play(AudioClip)

    /*****

    /**
     * Se escucha el audio en un bucle.
     */
    public void loop()
    {
        if (audioAnterior!=null)
            audioAnterior.loop();
    } //loop()

    /*****

    /**
     * Para la ejecución del audio que se esta escuchando y empieza el

```

```

    * audio pasado como parametro en un bucle.
    * @param audio el nuevo audio a escuchar en un bucle.
    */
    public void loop (AudioClip audio)
    {
        if (audioAnterior!=null)
            audioAnterior.stop();
        if (audio!=null)
        {
            audioAnterior= audio;
            audio.loop();
        }
    } //loop(AudioClip)

    /*****

    /**
    * Para la ejecucion del audio.
    */
    public void stop()
    {
        if (audioAnterior != null)
            audioAnterior.stop();
    } //stop

    } //class ManejadorAudio

    /*****
    /*****FIN DE LA CLASE*****/

```

Clase AudioPlayer

Se encuentra en el fichero AudioPlayer.java y su contenido es el siguiente:

```

import java.awt.*;
import java.applet.*;
import java.awt.event.*;

/**
 * Esta clase es la que implementa el applet. Desde aqui se
 * controlan los audios a traves de la clase ManejadorAudio.
 */
public class AudioPlayer extends Applet
{
    ManejadorAudio maneja=ManejadorAudio.getInstancia();
    /*maneja es la referencia al unico objeto que existe de la
    clase ManejadorAudio que lo obtiene a través del metodo
    getInstancia de la clase ManejadorAudio*/
    private AudioClip audio;
    private Image fondo;

    /*****

    /**

```

```

* Este método es el que inicializa el applet y muestra
* en la pantalla los botones y los radio botones.
*/
public void init()
{
    setLayout(new BorderLayout());
    Panel botones= new Panel();
    Button botonPlay = new Button("Play");
    Button botonStop = new Button("Stop");
    Button botonLoop = new Button("Loop");
    botonPlay.addActionListener(new HandlerBotones());
    botonStop.addActionListener(new HandlerBotones());
    botonLoop.addActionListener(new HandlerBotones());
    botones.add(botonPlay);
    botones.add(botonStop);
    botones.add(botonLoop);
    add("South",botones);

    Panel radioboton= new Panel();
    CheckboxGroup grupo = new CheckboxGroup();
    Checkbox radiol= new Checkbox("Audio1",grupo,false);
    Checkbox radio2= new Checkbox("Audio2",grupo,false);
    radiol.addItemListener (new HandlerCheckbox());
    radio2.addItemListener (new HandlerCheckbox());
    radioboton.add(radiol);
    radioboton.add(radio2);
    add("East", radioboton);
} //init

/*****

/**
 * Este metodo se ejecuta cuando se para el applet y lo
 * que hace es parar el audio si esta sonando.
 */
public void stop()
{
    maneja.stop();
} //stop

/*****

/**
 * Esta clase es la que pinta en la pantalla, pone el nombre
 * AUDIO PLAYER y visualiza una foto si esta asignada a la
 * variable fondo de tipo image.
 */
public void paint(Graphics g)
{
    Font tipofuente= new Font("SansSerif", Font.BOLD, 25);
    g.setFont(tipofuente);
    g.setColor(Color.blue);
    g.drawString("AUDIO PLAYER (Patrón Singleton)", 30, 30);
    if (fondo!=null)
        g.drawImage(fondo,31,35,this);
} //paint

/****

```

```

/*****/

/**
 * Esta clase controla los eventos sobre los botones.
 */
class HandlerBotones implements ActionListener
{
/*****/

/**
 * Este metodo dicta lo que se ha de hacer cuando se pulsa
 * alguno de los tres botones: play, stop y loop.
 */
public void actionPerformed(ActionEvent e)
{
    String s = e.getActionCommand();
    if("Play".equals(s))
        maneja.play(audio);
    else if("Stop".equals(s))
        maneja.stop();
    else if("Loop".equals(s))
        maneja.loop(audio);
} //actionPerformed

/*****/

} //class HndlerBotones

/*****/
/*****/

/**
 * Esta clase controla los eventos sobre los radio botones.
 */
class HandlerCheckbox implements ItemListener
{
/*****/

/**
 * Este método dicta lo que se hace cuando se selecciona alguno
 * de los dos radio botones: audio1 y audio2.
 */
public void itemStateChanged (ItemEvent e)
{
    String s;
    Graphics g;
    Checkbox checkbox=(Checkbox) e.getItemSelectable();
    if(checkbox.getState())
    {
        s=checkbox.getLabel();
        if ("Audio1".equals(s))
        {
            audio=getAudioClip(getCodeBase(),"audio/guitarra.au");
            maneja.play(audio);
            fondo= getImage(getCodeBase(),"paris.jpg");
            repaint();
        }
        else
        {

```

```

        audio=getAudioClip(getCodeBase(), "audio/spacemusic.au");
        maneja.play(audio);
        fondo=getImage(getCodeBase(),"playa.jpg");
        repaint();
    }
}
} //itemStateChanged

/*****

} //class HandlerCheckbox

/

/

/

} // class AudioPlayer

/
** /
/*****FIN DE LA
CLASE*****/

```

Página web AudioPlayer

Se encuentra en el fichero AudioPlayer.html y su contenido es el siguiente:

```

<HTML>
  <HEAD>
    <TITLE> Audio Player</TITLE>
  </HEAD>
  <BODY>
    <APPLET CODE="AudioPlayer.class" WIDTH="600" HEIGHT="400">
      </APPLET>
  </BODY>
</HTML>

```

ANEXO B: Código completo en Java del apartado 3.2.4

La aplicación del apartado 3.2 (Listados de forma ascendente y descendente de una lista) consta de tres clases :

- Clase Coleccion.
- Clase ListaIterator
- Clase DialogAbout.

Clase Coleccion

Se encuentra en el fichero Coleccion.java y su contenido es el siguiente:

```
import java.util.*;

/**
 * Esta clase implementa la coleccion de elementos, que en este caso
 * sera una lista de tipo LinkedList que contiene 10 elementos enteros
 * del 1 al 10 ambos incluidos.
 */
public class Coleccion
{
    /**
     * Creamos una lista enlazada llamada lista de tipo LinkedList
     */
    private LinkedList lista=new LinkedList();

    /**
     * Este es el constructor de la clase y lo que hace es insertar en la
     * lista los 10 enteros del 1 al 10 ambos incluidos.
     */
    public Coleccion()
    {
        for(int i=1;i<11;i++)
        {
            lista.add(new Integer(i));
        }
    }

    /**
     * Retorna un ListIterator que es un interface IFlterator para listas
     */
}
```

```

* que permite recorrer la lista en cada direccion y modificar la
* lista durante el recorrido. Este interface se obtiene a través del
* metodo listIterator de la clase LinkedList pasandole como
* parámetro La posicion especifica de la lista por la que empezar el
* recorrido.
* @param index indica la posición de la lista por la que empezar el
* recorrido.
* @return ListIterator es un interface de tipo ListIterator.
*/
public ListIterator miiterator(int index)
{
    return lista.listIterator(index);
} //miiterator

/*****

*/ //class Coleccion

/*****
/*****FIN DE LA CLASE*****/

```

Clase Listalterator

Se encuentra en el fichero ListaIterator.java y su contenido es el siguiente:

```

import java.awt.*;
import java.awt.event.*;
import java.util.*;

/**
 * Esta clase hereda de Frame y es la que implementa la ventana que
 * sale en la pantalla al ejecutar la aplicación y las opciones que se
 * deben realizar al pulsar las opciones del menu.
 */
public class ListaIterator extends Frame implements ActionListener
{
    private DialogAbout dialog;
    private Coleccion milista=new Coleccion();//crea la lista de 10
    elementos.
    private Label recorrido=new Label();
    private Label nodos=new Label();
    private Label num=new Label();

    /*****

    /**
     * Es el constructor de la clase y se encarga de realizar el menu.
     */
    public ListaIterator()
    {
        super("Utilización del patrón Iterator");
    }

```



```

Toolkit tk= Toolkit.getDefaultToolkit();
Dimension d=tk.getScreenSize();
int resolucionPantallaAlto=d.height;
int resolucionPantallaAncho=d.width;
setSize(resolucionPantallaAncho/2, resolucionPantallaAlto/2);
setLocation(resolucionPantallaAncho/8,resolucionPantallaAlto/8);
//creacion de menus
MenuBar mbar= new MenuBar();
setMenuBar(mbar);
//Menu Listado
Menu listadoMenu= new Menu("Listado");
mbar.add(listadoMenu);
//opciones del menu listado
//opcion orden ascendente
MenuItem ascendenteItem=new MenuItem("Ascendente", new
MenuShortcut('a'));
ascendenteItem.addActionListener(this);
listadoMenu.add(ascendenteItem);
//opcion orden descendente
MenuItem descendenteItem=new MenuItem("Descendente", new
MenuShortcut('d'));
descendenteItem.addActionListener(this);
listadoMenu.add(descendenteItem);
listadoMenu.add("-");
//opcion salir
MenuItem salirItem=new MenuItem("Salir", new MenuShortcut('s'));
salirItem.addActionListener(this);
listadoMenu.add(salirItem);

//Menu Ayuda
Menu AyudaMenu= new Menu("Ayuda");
mbar.add(AyudaMenu);
//opciones del menu Ayuda
//opcion About
MenuItem aboutItem=new MenuItem("About", new MenuShortcut('b'));
aboutItem.addActionListener(this);
AyudaMenu.add(aboutItem);

addWindowListener (new WindowAdapter()
{
    public void windowClosing(WindowEvent e)
    {
        dispose();
        System.exit(0);
    }
});
setLayout(null);
recorrido.setBounds(50,75,400,40);
recorrido.setFont(new Font("SansSerif",Font.BOLD,26));
add(recorrido);
nodos.setFont(new Font("SansSerif",Font.BOLD,14));
num.setFont(new Font("SansSerif",Font.ITALIC,14));
} //ListaIterator

/*****

/**
 * Es el método principal main de la clase, crea un objeto de esta

```

```

    * clase y la visualiza por pantalla con el tamaño deseado.
    */
    public static void main(String args[])
    {
        ListaIterator f = new ListaIterator();
        f.setSize(600,400);
        f.show();
    } //main(String)

    /*****

    /**
     * Este método dicta lo que se ha de hacer cada vez que se
     * selecciona alguna opción del menu.
     */
    public void actionPerformed (ActionEvent e)
    {
        String seleccion=e.getActionCommand();

        Integer numero=new Integer(0);
        int nume;

        if ("Salir".equals(seleccion))//item Salir
        {
            System.exit(0);
        }
        else if ("Ascendente".equals(seleccion))//item Ascendente
        {
            recorrido.setText("RECORRIDO ASCENDENTE");
            nodos.setText("Nodo1  Nodo2  Nodo3  Nodo4  Nodo5
Nodo6  Nodo7  Nodo8  Nodo9  Nodo10");
            nodos.setBounds(50,200,550,30);
            add(nodos);
            //conseguimos el iterador para recorrer la lista
            ascendente
            ListIterator ascendente=millista.miiterator(0);
            num.setText("");
            //recorremos la lista ascendente mientras existan
            elementos
            while (ascendente.hasNext())
            {
                numero=(Integer)ascendente.next();
                nume=numero.intValue();
                num.setText(num.getText()+"
"+Integer.toString(nume));
            }
            num.setBounds(30,225,550,30);
            add(num);
        }
        else if ("Descendente".equals(seleccion))//item descendente
        {
            recorrido.setText("RECORRIDO DESCENDENTE");
            nodos.setText("Nodo10  Nodo9  Nodo8  Nodo7  Nodo6
Nodo5  Nodo4  Nodo3  Nodo2  Nodo1");
            nodos.setBounds(50,200,550,30);
            add(nodos);
            //conseguimos el iterador para recorrer la lista
            descendente

```

```

        ListIterator descendente=milista.miiterator(10);
        num.setText("");
        //recorremos la lista descendentemente mientras existan
elementos
        while (descendente.hasPrevious())
        {
            numero=(Integer)descendente.previous();
            nume=numero.intValue();
            num.setText(num.getText()+"
"+Integer.toString(nume));
        }
        num.setBounds(30,225,550,30);
        add(num);

    }
    else if ("About".equals(seleccion))//item About
    {
        int resolucionPantallaAlto=this.getSize().height;
        int resolucionPantallaAncho=this.getSize().width;
        dialog=new DialogAbout(this);

        dialog.setLocation(resolucionPantallaAncho/3,resolucionPantallaAlto/2)
        ;
        dialog.show();
    }
    }//actionPerformed(ActionEvent)

/*****/

} //class ListaIterator

/*****/
/*****FIN DE LA CLASE*****/

```

Clase DialogAbout

Se encuentra en el fichero DialogAbout.java y su contenido es el siguiente:

```

import java.awt.*;
import java.awt.event.*;

/**
 * Esta clase implementa el cuadro de dialogo que sale por pantalla
 * al pulsar en el menu Ayuda el item About o las teclas control+b
 */
public class DialogAbout extends Dialog
{
    static int HOR_TAMANO = 450;//tamaño en horizontal del cuadro
    static int VER_TAMANO = 200;//tamaño en vertical del cuadro

/*****/

```

```

/**
 * Es el constructor de la clase.
 */
public DialogAbout( Frame parent )
{
    super( parent,"About",true);
    this.setResizable( false );
    setBackground( Color.lightGray);
    setLayout( new BorderLayout() );
    Panel p = new Panel();
    Button b=new Button("Aceptar");
    b.addActionListener(new DialogHandler());
    p.add(b);
    add( "South",p );
    setSize( HOR_TAMANO,VER_TAMANO );
    addWindowListener(new WindowAdapter()
        {public void windowClosing (WindowEvent e)
        {
            dispose();
        }
    });
} //DialogAbout(Frame)

/*****

/**
 * Este método pinta en el cuadro las frases que aparecen en
 * él, en la posición deseada y con el color y fuentes
 * deseados.
 */
public void paint( Graphics g )
{
    g.setColor( Color.black );
    Font f=new Font("SansSerif",Font.BOLD,14);
    g.setFont(f);
    g.drawString( "Listados de forma ascendente y descendente
de una lista",
        (HOR_TAMANO/HOR_TAMANO)+15,VER_TAMANO/3 );
    g.drawString( "Patrón utilizado: ITERATOR",
        HOR_TAMANO/3,VER_TAMANO/3+40 );
    Font f1=new Font("TimesRoman",Font.ITALIC,12);
    g.setFont(f1);
    g.drawString( "Autor: Francisco Javier Martínez Juan",
        HOR_TAMANO/3,VER_TAMANO/3+80 );

} //paint(Graphics)

/*****

/*****

/**
 * Esta clase es la que maneja los eventos que ocurren sobre el
 * cuadro de dialogo.
 */
class DialogHandler implements ActionListener

```

```

{

/*****

/**
 * Este método cierra el cuadro al pulsar el boton Aceptar.
 */
public void actionPerformed (ActionEvent e)
{
    dispose();
} //actionPerformed(ActionEvent)

/*****
} //class DialogHandler

/*****
/*****

} //class DialogAbout

/*****
/*****FIN DE LA CLASE*****/

```


ANEXO C: Código completo en Java del apartado 3.3.4

La aplicación del apartado 3.3 (Ordenación de un vector de forma ascendente y descendente) consta de un interface y seis clases :

- Interface Ordenacion.
- Clase AlgBurbujaAsc.
- Clase AlgBurbujaDesc.
- Clase Algoritmos.
- Clase Coleccion.
- Clase OrdenacionStrategy.
- Clase DialogAbout.

Interface Ordenacion

Se encuentra en el fichero Ordenacion.java y su contenido es el siguiente:

```
/**
 * Este interface proporciona una forma común para acceder al
 * algoritmo de ordenación encapsulado en sus subclases, es decir
 * en las clases que implementan el interface.
 */
interface Ordenacion
{
    /**
     * Este método es el que se utiliza para ejecutar cualquiera
     * de los algoritmos de ordenación y será implementado por el
     * algoritmo de ordenación determinado.
     */
    public int[] ordenar(int a[]);
} //interface Ordenacion

/*****
/*****FIN DEL INTERFACE*****/
```

Clase AlgBurbujaAsc

Se encuentra en el fichero AlgBurbujaAsc.java y su contenido es el siguiente:

```
/**

 * Ordena un vector por el metodo de la Burbuja ascendente.
```

```

*/

class AlgBurbujaAsc implements Ordenacion
{

    /**
     * Ordena un vector por el método de la Burbuja descendente.
     * @param a[] el vector a ordenar.
     * @return int[] el vector ordenado.
     */
    public int[] ordenar(int a[])
    {
        for (int i = 1; i <= a.length; ++i)
            for (int j = (a.length) - 1; j >= i; --j)
            {
                if (a[j-1] > a[j])
                {
                    int T = a[j-1];
                    a[j-1] = a[j];
                    a[j] = T;
                }
            }
        return a;
    } //ordenar

}

} //class AlgBurbujaAsc

}

} //*****FIN DE LA CLASE*****

```

Clase AlgBurbujaDesc

Se encuentra en el fichero AlgBurbujaDesc.java y su contenido es el siguiente:

```

/**
 * Ordena un vector por el metodo de la Burbuja descendente.
 */

class AlgBurbujaDesc implements Ordenacion
{

    /**
     * Ordena un vector por el método de la burbuja descendente.
     * @param a[] el vector a ordenar.
     * @return int[] el vector ordenado.
     */

```



```

    */
    public int[] ordenar(int a[])
    {
        for (int i = 1; i <= a.length; ++i)
            for (int j = (a.length) -1; j>= i; --j)
            {
                if (a[j-1]< a[j])
                {
                    int T = a[j-1];
                    a[j-1] = a[j];
                    a[j] = T;
                }
            }
        return a;
    } //ordenar

    /*****

} //class AlgBurbujaDesc

    /*****
    /*****FIN DE LA CLASE*****/

```

Clase Algoritmos

Se encuentra en el fichero Algoritmos.java y su contenido es el siguiente:

```

/**
 * Esta clase controla la selección y uso del objeto particular
 * strategy. Es decir mediante esta clase se selecciona el algoritmo
 * a utilizar en nuestro caso AlgBurbujaAsc o AlgBurbujaDesc y se
 * llama a su método ordenar correspondiente.
 */
public class Algoritmos
{
    private Ordenacion algoritmo; //referencia al interface Ordenacion.

    /*****

    /**
     * Constructor de la clase.
     * @param alg es un objeto de tipo ordenacion con el cual
     * seleccionamos el algoritmo que queremos utilizar.
     */
    public Algoritmos(Ordenacion alg)
    {
        algoritmo=alg;
    } //Algoritmos

    /*****

    /**
     * Con este método llamamos al método ordenar del algoritmo elegido.

```

```

* @param a[] el vector a ordenar.
* @return int[] el vector ordenado.
*/
public int[] ordena (int[] a)
{
    return algoritmo.ordenar(a);
} //ordena

/*****

/**
 * Este método sirve para cambiar la selección del algoritmo.
 * @param nuevoAlgoritmo el nuevo algoritmo seleccionado.
 */
public void cambiarAlgoritmo (Ordenacion nuevoAlgoritmo)
{
    algoritmo=nuevoAlgoritmo;
} //cambiarAlgoritmo

/*****

} //class Algoritmos

/*****
/*****FIN DE LA CLASE*****/

```

Clase Coleccion

Se encuentra en el fichero Coleccion.java y su contenido es el siguiente:

```

import java.util.*;

/**
 * Esta clase implementa la coleccion de elementos, que en este caso
 * sera un vector de tipo Vector que contiene 10 elementos enteros
 * desordenados {4,7,6,1,2,8,3,5,10,9}
 */
public class Coleccion
{
    private static Vector vector=new Vector(9);

    /*****

    /**
     * Este es el constructor de la clase y lo que hace es insertar en el
     * vector los 10 enteros desordenados y se imprime por pantalla el
     * contenido del vector.
     */
    public Coleccion()
    {
        vector.clear();
        vector.addElement(new Integer(4));
        vector.addElement(new Integer(7));
    }
}

```

```

vector.addElement(new Integer(6));
vector.addElement(new Integer(1));
vector.addElement(new Integer(2));
vector.addElement(new Integer(8));
vector.addElement(new Integer(3));
vector.addElement(new Integer(5));
vector.addElement(new Integer(10));
vector.addElement(new Integer(9));
// vector={4,7,6,1,2,8,3,5,10,9}
System.out.print("Vector Desordenado: ");
for (Enumeration e = vector.elements() ; e.hasMoreElements() ; )
{
    Integer a;
    a=(Integer)e.nextElement();
    System.out.print(a.intValue()+" ");

}
System.out.println();
} //Coleccion

/*****

/**
 * Este método pasa el vector de tipo Vector (objetos Integer) a
 * un array de elementos int (tipo básico entero) para poder
 * ordenarlo.
 * @return int[] es el array con los elementos int del vector.
 */
public int[] convertirEnArray ()
{
    Integer a;
    int[] b=new int[10];
    int i=0;
    for (Enumeration e = vector.elements() ; e.hasMoreElements() ; )
    {
        a=(Integer)e.nextElement();
        b[i]=a.intValue();
        i++;
    }

    return b;
} //convertirEnArray

/*****

/**
 * Este método pasa el array b[] pasado como parametro al vector de
 * tipo Vector (objetos Integer) para recorrerlo con un iterator.
 * @param b[] es el array de elementos de tipo int.
 * @return Vector es el vector de elementos Integer.
 */
public Vector convertirEnVector (int b[])
{
    Integer a;

    vector.clear();//eliminamos todos los elementos del vector.

```

```

        for(int i=0;i<b.length;i++)
        {
            vector.addElement(new Integer(b[i]));
        }
        return vector;
    } //convertirEnVector

/*****

/**
 * Retorna un ListIterator que es un interface IFIterator para
 * vectores que permite recorrer el vector. Este interface se obtiene
 * a través del metodo listIterator de la clase Vector pasandole como
 * parámetro La posicion especifica del vector por la que empezar
 * el recorrido.
 * @param index indica la posición del vector por la que empezar el
 * recorrido.
 * @return ListIterator es un interface de tipo ListIterator.
 */
public ListIterator miiterator(int index)
{
    return vector.listIterator(index);
} //miiterator

/*****

} //class Coleccion

/*****
/*****FIN DE LA CLASE*****/

```

Clase OrdenacionStrategy

Se encuentra en el fichero OrdenacionStrategy.java y su contenido es el siguiente:

```

import java.awt.*;
import java.awt.event.*;
import java.util.*;

/**
 * Esta clase hereda de Frame y es la que implementa la ventana que
 * sale en la pantalla al ejecutar la aplicación y las opciones que se
 * deben realizar al pulsar las opciones del menu.
 */
public class OrdenacionStrategy extends Frame implements
ActionListener
{
    private DialogAbout dialog;
    private Coleccion mivector;
    private Algoritmos algOrdenacion; //para seleccionar y utilizar el
                                     //algoritmo deseado.
    private Label ordenacion=new Label();

```

```

private Label nodos=new Label();
private Label num=new Label();

/*****

/**
 * Es el constructor de la clase y se encarga de realizar el menu.
 */
public OrdenacionStrategy()
{
    super("Utilización de los patrones Strategy e Iterator");
    Toolkit tk= Toolkit.getDefaultToolkit();
    Dimension d=tk.getScreenSize();
    int resolucionPantallaAlto=d.height;
    int resolucionPantallaAncho=d.width;
    setSize(resolucionPantallaAncho/2, resolucionPantallaAlto/2);
    setLocation(resolucionPantallaAncho/8,
resolucionPantallaAlto/8);
    //creacion de menus
    MenuBar mbar= new MenuBar();
    setMenuBar(mbar);
    //Menu Ordenación
    Menu ordenacionMenu= new Menu("Ordenación");
    mbar.add(ordenacionMenu);
    //opciones del menu Ordenacion
    //opcion orden ascendente
    MenuItem ascendenteItem=new MenuItem("Ascendente", new
MenuShortcut('a'));
    ascendenteItem.addActionListener(this);
    ordenacionMenu.add(ascendenteItem);
    //opcion orden descendente
    MenuItem descendenteItem=new MenuItem("Descendente", new
MenuShortcut('d'));
    descendenteItem.addActionListener(this);
    ordenacionMenu.add(descendenteItem);
    ordenacionMenu.add("-");
    //opcion salir
    MenuItem salirItem=new MenuItem("Salir", new MenuShortcut('s'));
    salirItem.addActionListener(this);
    ordenacionMenu.add(salirItem);

    //Menu Ayuda
    Menu AyudaMenu= new Menu("Ayuda");
    mbar.add(AyudaMenu);
    //opciones del menu Ayuda
    //opcion About
    MenuItem aboutItem=new MenuItem("About", new MenuShortcut('b'));
    aboutItem.addActionListener(this);
    AyudaMenu.add(aboutItem);

    addWindowListener (new WindowAdapter()
    {
        public void windowClosing(WindowEvent e)
        {
            dispose();
            System.exit(0);
        }
    }
}

```

```

    });
    setLayout(null);
    ordenacion.setBounds(50,75,400,40);
    ordenacion.setFont(new Font("SansSerif",Font.BOLD,26));
    add(ordenacion);
    nodos.setFont(new Font("SansSerif",Font.BOLD,14));
    num.setFont(new Font("SansSerif",Font.ITALIC,14));
} //OrdenacionStrategy

/*****

/**
 * Es el método principal main de la clase, crea un objeto de esta
 * clase y la visualiza por pantalla con el tamaño deseado.
 */
public static void main(String args[])
{
    OrdenacionStrategy f = new OrdenacionStrategy();
    f.setSize(600,400);
    f.show();
} //main(String)

/*****

/**
 * Este método dicta lo que se ha de hacer cada vez que se
 * selecciona alguna opción del menu.
 */
public void actionPerformed (ActionEvent e)
{
    String seleccion=e.getActionCommand();
    int[] miarray=new int[10];
    Integer numero=new Integer(0);
    int nume;

    if ("Salir".equals(seleccion))//item Salir
    {
        System.exit(0);
    }
    else if ("Ascendente".equals(seleccion))//item Ascendente
    {
        ordenacion.setText("ORDENACIÓN ASCENDENTE");
        nodos.setText("Nodo1  Nodo2  Nodo3  Nodo4  Nodo5
Nodo6  Nodo7  Nodo8  Nodo9  Nodo10");
        nodos.setBounds(50,200,550,30);
        add(nodos);
        mivector=new Coleccion();//crea el vector de 10 elementos.

        //Seleccionamos el algoritmo deseado.
        algOrdenacion=new Algoritmos(new AlgBurbujaAsc());
        //Y utilizamos el algoritmo deseado.
        miarray=algOrdenacion.ordena(mivector.convertirEnArray());
        mivector.convertirEnVector(miarray);

        //conseguimos el iterador para recorrer la lista
        ascendente
        ListIterator ascendente=mivector.miiterator(0);
        num.setText("");

```

```

//recorremos la lista ascendentemente mientras existan
elementos
while (ascendente.hasNext())
{
    numero=(Integer)ascendente.next();
    nume=numero.intValue();
    num.setText(num.getText()+"
"+Integer.toString(nume));
}
num.setBounds(30,225,550,30);
add(num);
}
else if ("Descendente".equals(seleccion))//item descendente
{
    ordenacion.setText("ORDENACIÓN DESCENDENTE");
    nodos.setText("Nodo1  Nodo2  Nodo3  Nodo4  Nodo5
Nodo6  Nodo7  Nodo8  Nodo9  Nodo10");
    nodos.setBounds(50,200,550,30);
    add(nodos);
    mivector=new Coleccion();//crea el vector de 10 elementos.

    //Seleccionamos el algoritmo deseado.
    algOrdenacion=new Algoritmos(new AlgBurbujaDesc());
    // Y utilizamos el algoritmo deseado.
    miarray=algOrdenacion.ordena(mivector.convertirEnArray());
    mivector.convertirEnVector(miarray);

    //conseguimos el iterador para recorrer la lista
ascendentemente
ListIterator ascendente=mivector.miiterator(0);
num.setText("");
//recorremos la lista ascendentemente mientras existan
elementos
while (ascendente.hasNext())
{
    numero=(Integer)ascendente.next();
    nume=numero.intValue();
    num.setText(num.getText()+"
"+Integer.toString(nume));
}
num.setBounds(30,225,550,30);
add(num);
}
else if ("About".equals(seleccion))//item About
{
    int resolutionPantallaAlto=this.getSize().height;
    int resolutionPantallaAncho=this.getSize().width;
    dialog=new DialogAbout(this);

    dialog.setLocation(resolutionPantallaAncho/3,resolutionPantallaAlto/2)
;
    dialog.show();
}
} //actionPerformed(ActionEvent)

/*****/

```

```
//class OrdenacionStrategy

/*****
/*****FIN DE LA CLASE*****/
```

Clase DialogAbout

Se encuentra en el fichero DialogAbout.java y su contenido es el siguiente:

```
import java.awt.*;
import java.awt.event.*;

/**
 * Esta clase implementa el cuadro de dialogo que sale por pantalla
 * al pulsar en el menu Ayuda el item About o las teclas control+b
 */
public class DialogAbout extends Dialog
{
    static int HOR_TAMANO = 450; //tamaño en horizontal del cuadro
    static int VER_TAMANO = 200; //tamaño en vertical del cuadro

    /**
     * Es el constructor de la clase.
     */
    public DialogAbout( Frame parent )
    {
        super( parent, "About", true );
        this.setResizable( false );
        setBackground( Color.lightGray );
        setLayout( new BorderLayout() );
        Panel p = new Panel();
        Button b = new Button( "Aceptar" );
        b.addActionListener( new DialogHandler() );
        p.add( b );
        add( "South", p );
        setSize( HOR_TAMANO, VER_TAMANO );
        addWindowListener( new WindowAdapter()
        {
            public void windowClosing ( WindowEvent e )
            {
                dispose();
            }
        } );
    } //DialogAbout( Frame )

    /**
     * Este método pinta en el cuadro las frases que aparecen en
     * él, en la posición deseada y con el color y fuentes
     */
}
```



```

        * deseados.
        */
    public void paint( Graphics g )
    {
        g.setColor( Color.black );
        Font f=new Font( "SansSerif",Font.BOLD,14);
        g.setFont(f);
        g.drawString( "Ordenación de un vector de forma ascendente
y descendente",
            (HOR_TAMANO/HOR_TAMANO)+5,VER_TAMANO/3 );
        g.drawString( "Patrones utilizados: STRATEGY e ITERATOR",
            HOR_TAMANO/5,VER_TAMANO/3+40 );
        Font f1=new Font( "TimesRoman",Font.ITALIC,12);
        g.setFont(f1);
        g.drawString( "Autor: Francisco Javier Martínez Juan",
            HOR_TAMANO/3,VER_TAMANO/3+80 );

    }//paint(Graphics)

/*****
/*****

/**
 * Esta clase es la que maneja los eventos que ocurren sobre el
 * cuadro de dialogo.
 */
class DialogHandler implements ActionListener
{

/*****

/**
 * Este método cierra el cuadro al pulsar el boton Aceptar.
 */
    public void actionPerformed (ActionEvent e)
    {
        dispose();
    }//actionPerformed(ActionEvent)

/*****
    }//class DialogHandler

/*****
/*****

}//class DialogAbout

/*****
/*****FIN DE LA CLASE*****/

```


ANEXO D: Código completo en Java del apartado 3.4.4

La aplicación del apartado 3.4 (Ejemplo visual del patrón Observer) consta de cinco clases:

- Clase ValorObservable.
- Clase TextoObservador.
- Clase BarraObservador.
- Clase PatronObserver.
- Clase DialogAbout.

Clase ValorObservable

Se encuentra en el fichero ValorObservable.java y su contenido es el siguiente:

```
import java.util.Observable;

/**
 * Esta clase representa la clase Observable y es la que
 * notifica a los observadores (Observers) que estan
 * pendientes de los cambios de estado de esta clase, que
 * su estado se ha visto alterado.
 */
public class ValorObservable extends Observable
{
    private int nValor = 0;
    private int nInferior = 0;
    private int nSuperior = 0;

    /**
     * Constructor al que indicamos el valor en que comenzamos
     * y los limites inferior y superior que no deben
     * sobrepasarse.
     */
    public ValorObservable( int nValor,int nInferior,int nSuperior )
    {
        this.nValor = nValor;
        this.nInferior = nInferior;
        this.nSuperior = nSuperior;
    }

    /**
     * Fija el valor que le pasamos y notifica a los observadores que
     * estan pendientes del cambio de estado de los objetos de esta

```

```

* clase, que su estado se ha visto alterado.
*/
public void setValor(int nValor)
{
    this.nValor = nValor;
    setChanged();
    notifyObservers();
} //setValor(int)

/*****

/**
 * Devuelve el valor actual que tiene el objeto.
 */
public int getValor()
{
    return( nValor );
} //getValor

/*****

/**
 * Devuelve el limite inferior del rango de valores en los que se
 * ha de mover el objeto.
 */
public int getLimiteInferior()
{
    return( nInferior );
} //getLimiteInferior

/*****

/**
 * Devuelve el limite superior del rango de valores en los que se
 * ha de mover el objeto.
 */
public int getLimiteSuperior()
{
    return( nSuperior );
} //getLimiteSuperior

/*****

} //class ValorObservable

/*****
/
/*****FIN DE LA CLASE*****/

```

Clase TextoObservador

Se encuentra en el fichero TextoObservador.java y su contenido es el siguiente:

```
import java.awt.*;
import java.awt.event.*;
import java.util.Observer;
import java.util.Observable;

/**
 * Esta clase representa un observador (Observer) que es un TextField,
 * que es un campo de entrada de texto de una sola linea.
 * Esta linea de texto aparecerá en una ventana independiente.
 * Se crea un nuevo objeto de esta clase, es decir una nueva
 * línea de entrada de texto cada vez que se pulse el botón
 * "Nuevo Observador Texto".
 */
public class TextoObservador extends Frame implements Observer
{
    private ValorObservable vo = null;
    private TextoObservador to;
    private TextField tf = null;
    private Label l = null;
    private int nInferior = 0;
    private int nSuperior = 0;

    /**
     * Constructor en el que creamos una ventana donde poder
     * introducir el valor numerico que represente la
     * cantidad que corresponda al objeto observable.
     */
    public TextoObservador( ValorObservable vo )
    {
        super("Observador de Texto");
        this.vo = vo;
        setLayout( new GridLayout( 0,1 ) );
        nInferior = vo.getLimiteInferior();
        nSuperior = vo.getLimiteSuperior();

        tf = new TextField( String.valueOf( vo.getValor() ) );
        tf.addActionListener(new ManejadorTextField());
        add( tf );

        l = new Label();
        add( l );
        setSize(300,100);
        addWindowListener (new WindowEventHandler());
        show();
    } //TextoObservador(ValorObservable)
}
```

```

/**
 * Actualizamos el valor del objeto que estamos observando
 * en nuestro campo de texto, para reflejar el valor actual
 * del objeto.
 */
public void update( Observable obs,Object obj )
{
    if( obs == vo )
        tf.setText( String.valueOf( vo.getValor() ) );
} //update(Observable, Object)

/*****
/*****

/**
 * Esta clase maneja los events que se producen en el campo
 * de texto.
 */
class ManejadorTextField implements ActionListener
{
    /*****

    /**
     * Controlamos el evento que proviene del campo de texto,
     * cuando se introduce un valor numerico.
     */
    public void actionPerformed( ActionEvent e)
    {
        int n = 0;
        boolean bValido = false;

        try
        {
            n = Integer.parseInt( tf.getText() );
            bValido = true;
        }
        catch( NumberFormatException nfe )
        {
            bValido = false;
        }

        // Comprobamos que no se sobrepasen los limites que hemos
        // fijado
        if( n < nInferior || n > nSuperior )
            bValido = false;

        if( bValido )
        {
            vo.setValor( n );
            l.setText( "" );
        }
        else
            l.setText( "Valor no válido -> inténtelo de nuevo:
["+nInferior+"-"+nSuperior+"]" );
    } //actionPerformed(ActionEvent)

    /*****

```

```
//class ManejadorTextField

/*****
/*****

/**
 * Esta clase maneja los eventos que se producen sobre la
 * ventana del campo de texto.
 */
class WindowEventHandler extends WindowAdapter
{
/*****

/**
 * Controlamos el cierre de la ventana, para eliminar el
 * objeto que hemos creado antes de hacer desaparecer
 * la ventana.
 */
public void windowClosing( WindowEvent e )
{
    vo.deleteObserver(to);
    dispose();
} //windowClosing(WindowEvent)

/*****
} //class WindowEventHandler

/*****
/*****

} //class TextoObservador

/*****
/*****FIN DE LA CLASE*****/
```

Clase BarraObservador

Se encuentra en el fichero BarraObservador.java y su contenido es el siguiente:

```
import java.awt.*;
import java.awt.event.*;
import java.util.Observer;
import java.util.Observable;

/**
 * Esta clase representa un observador (Observer) que es una barra de
 * desplazamiento. Esta barra de desplazamiento aparecerá
 * en una ventana independiente. Se crea un nuevo objeto
 * de esta clase es decir una nueva barra de desplazamiento
 * cada vez que se pulse el botón "Nuevo Observador Barra".
 */
public class BarraObservador extends Frame implements Observer
{
    private ValorObservable vo = null;
```

```

private BarraObservador bo;
private Scrollbar sb = null;

/*****

/**
 * Constructor que presenta una ventanita con la barra de
 * desplazamiento para poder ir alterando los valores del
 * objeto observable.
 */
public BarraObservador( ValorObservable vo )
{
    super( "Observador de Barra" );
    this.vo = vo;
    setLayout( new GridLayout( 0,1 ) );
    sb = new Scrollbar( Scrollbar.HORIZONTAL,vo.getValor(),1,
        vo.getLimiteInferior(),vo.getLimiteSuperior() );
    sb.addAdjustmentListener(new ManejadorScrolling());
    add( sb );
    setSize(300,100);
    addWindowListener (new WindowEventHandler());
    show();
} //BarraObservador(ValorObservable)

/*****

/**
 * Actualizamos el valor que corresponde al número que
 * actualmente tiene el objeto que estamos observando.
 */
public void update( Observable obs,Object obj )
{
    if( obs == vo )
        sb.setValue( vo.getValor() );
} //update(Observable, Object)

/*****
/*****

/**
 * Esta clase maneja los eventos que se producen sobre la
 * barra de desplazamiento.
 */
class ManejadorScrolling implements AdjustmentListener
{
    /*****

    /**
     * Manejamos los eventos que se producen al manipular la
     * barra. Cuando desplazamos el marcador de la barra, vamos
     * actualizando el valor del objeto observable y presentamos
     * el valor que se ha adquirido.
     */
    public void adjustmentValueChanged( AdjustmentEvent e)
    {
        if(e.UNIT_INCREMENT==AdjustmentEvent.UNIT_INCREMENT )
        {
            vo.setValor( sb.getValue() );
        }
    }
}

```



```

        }
        else if( e.UNIT_DECREMENT == AdjustmentEvent.UNIT_DECREMENT )
        {
            vo.setValor( sb.getValue() );
        }
        else if( e.BLOCK_INCREMENT == AdjustmentEvent.BLOCK_INCREMENT )
        {
            vo.setValor( sb.getValue() );
        }
        else if( e.BLOCK_DECREMENT == AdjustmentEvent.BLOCK_DECREMENT )
        {
            vo.setValor( sb.getValue() );
        }
        else if( e.TRACK == AdjustmentEvent.TRACK )
        {
            vo.setValor( sb.getValue() );
        }
    } //adjustmentValueChanged(AdjustmentEvent)

    /*****

    */ //class ManejadorScrolling

    /*****
    /*****

    /**
     * Esta clase maneja los eventos sobre la ventana de la
     * barra de desplazamiento.
     */
    class WindowEventHandler extends WindowAdapter
    {
        /*****

        /**
         * Controlamos el cierre de la ventana, para eliminar el
         * objeto que hemos creado antes de hacer desaparecer
         * la ventana.
         */
        public void windowClosing( WindowEvent e )
        {
            vo.deleteObserver(bo);
            dispose();
        } //windowClosing(WindowEvent)

        /*****

    } //class WindowEventHandler

    /*****
    /*****

    } //class BarraObservador

    /*****
    /*****FIN DE LA CLASE*****/

```

Clase PatronObserver

Se encuentra en el fichero PatronObserver.java y su contenido es el siguiente:

```
import java.awt.*;
import java.awt.event.*;
import java.util.Observable;

/**
 * Esta clase hereda de Frame y es la que implementa la ventana que
 * sale en la pantalla al ejecutar la aplicación y las opciones que se
 * deben realizar al pulsar los botones y las opciones del menú.
 */
public class PatronObserver extends Frame implements ActionListener
{
    private DialogAbout dialog;

    private Button bObservTexto;
    private Button bObservBarra;
    private ValorObservable vo;

    private Label titulo=new Label();

    /**
     * Es el constructor de la clase y se encarga de realizar el menu.
     */
    public PatronObserver()
    {
        super("Utilización del patrón Observer");
        Toolkit tk= Toolkit.getDefaultToolkit();
        Dimension d=tk.getScreenSize();
        int resolucionPantallaAlto=d.height;
        int resolucionPantallaAncho=d.width;
        setSize(resolucionPantallaAncho/2, resolucionPantallaAlto/2);
        setLocation(resolucionPantallaAncho/8,
resolucionPantallaAlto/8);
        setBackground( Color.white );
        //creacion de menus
        MenuBar mbar= new MenuBar();
        setMenuBar(mbar);
        //Menu Salir
        Menu salirMenu= new Menu("Salir");
        mbar.add(salirMenu);
        //opciones del menu Salir
        //opcion salir
        MenuItem salirItem=new MenuItem("Salir", new MenuShortcut('s'));
        salirItem.addActionListener(this);
        salirMenu.add(salirItem);

        //Menu Ayuda
        Menu AyudaMenu= new Menu("Ayuda");
        mbar.add(AyudaMenu);
    }
}
```

```
//opciones del menu Ayuda
//opcion About
MenuItem aboutItem=new MenuItem("About", new MenuShortcut('b'));
aboutItem.addActionListener(this);
AyudaMenu.add(aboutItem);

setLayout(null);
titulo.setBounds(150,75,300,40);
titulo.setFont(new Font("SansSerif",Font.BOLD,26));
titulo.setText("PATRÓN OBSERVER");
add(titulo);

bObservTexto = new Button( "Nuevo Observador Texto" );
bObservBarra = new Button( "Nuevo Observador Barra" );
vo = new ValorObservable( 0,0,1000 );
bObservTexto.setBounds( 125,300,150,30);
bObservBarra.setBounds(300,300,150,30);
bObservTexto.setBackground( Color.lightGray );
bObservBarra.setBackground( Color.lightGray );
add( bObservTexto );
bObservTexto.addActionListener(this);
add( bObservBarra );
bObservBarra.addActionListener(this);

addWindowListener (new WindowAdapter()
{
    public void windowClosing(WindowEvent e)
    {
        dispose();
        System.exit(0);
    }
});

} //PatronObserver

/*****
**
* Es el método principal main de la clase, crea un objeto de esta
* clase y la visualiza por pantalla con el tamaño deseado.
*/
public static void main(String args[])
{
    PatronObserver f = new PatronObserver();
    f.setSize(600,400);
    f.show();
} //main(String)

/*****
**
* Este método dicta lo que se ha de hacer cada vez que se
* selecciona alguna opción del menú o se pulsa alguno de los
* botones.
*/
public void actionPerformed (ActionEvent e)
{
    String s=e.getActionCommand();
```

```

        if( "Nuevo Observador Texto".equals(s) )//botón Nuevo
Observador Texto
        {
            TextoObservador textoObserv = new TextoObservador( vo );
            vo.addObserver( textoObserv );
        }
        else if("Nuevo Observador Barra".equals(s))//botón Nuevo
Observador Barra
        {
            BarraObservador barraObserv = new BarraObservador( vo );
            vo.addObserver( barraObserv );
        }

        if ("Salir".equals(s))//item Salir
        {
            System.exit(0);
        }
        else if ("About".equals(s))//item About
        {
            int resolucioPantallaAlto=this.getSize().height;
            int resolucioPantallaAncho=this.getSize().width;
            dialog=new DialogAbout( this );

dialog.setLocation(resolucioPantallaAncho/3,resolucioPantallaAlto/2)
;
            dialog.show();
        }
    } //actionPerformed(ActionEvent)

/*****/

} //class PatronObserver

/*****/
/*****FIN DE LA CLASE*****/

```

Clase DialogAbout

Se encuentra en el fichero DialogAbout.java y su contenido es el siguiente:

```

import java.awt.*;
import java.awt.event.*;

/**
 * Esta clase implementa el cuadro de dialogo que sale por pantalla
 * al pulsar en el menu Ayuda el item About o las teclas control+b
 */
public class DialogAbout extends Dialog
{
    static int HOR_TAMANO = 450;//tamaño en horizontal del cuadro
    static int VER_TAMANO = 200;//tamaño en vertical del cuadro

```

```

/*****/

/**
 * Es el constructor de la clase.
 */
public DialogAbout( Frame parent )
{
    super( parent,"About",true);
    this.setResizable( false );
    setBackground( Color.lightGray);
    setLayout( new BorderLayout() );
    Panel p = new Panel();
    Button b=new Button("Aceptar");
    b.addActionListener(new DialogHandler());
    p.add(b);
    add( "South",p );
    setSize( HOR_TAMANO,VER_TAMANO );
    addWindowListener(new WindowAdapter()
        {public void windowClosing (WindowEvent e)
        {
            dispose();
        }
    });
} //DialogAbout(Frame)

/*****/

/**
 * Este método pinta en el cuadro las frases que aparecen en
 * él, en la posición deseada y con el color y fuentes
 * deseados.
 */
public void paint( Graphics g )
{
    g.setColor( Color.black );
    Font f=new Font("SansSerif",Font.BOLD,14);
    g.setFont(f);
    g.drawString( "Ejemplo visual del patrón Observer",
        HOR_TAMANO/4,VER_TAMANO/3 );
    g.drawString( "Patrón utilizado: Observer",
        HOR_TAMANO/3,VER_TAMANO/3+40 );
    Font f1=new Font("TimesRoman",Font.ITALIC,12);
    g.setFont(f1);
    g.drawString( "Autor: Francisco Javier Martínez Juan",
        HOR_TAMANO/3,VER_TAMANO/3+80 );

} //paint(Graphics)

/*****/

/*****/

/**
 * Esta clase es la que maneja los eventos que ocurren sobre el
 * cuadro de dialogo.

```

```

    */
    class DialogHandler implements ActionListener
    {

        /*****

        /**
         * Este método cierra el cuadro al pulsar el boton Aceptar.
         */
        public void actionPerformed (ActionEvent e)
        {
            dispose();
        } //actionPerformed(ActionEvent)

        /*****
        } //class DialogHandler

        /*****
        /*****

    } //class DialogAbout

    /*****
    /*****FIN DE LA CLASE*****/

```

ANEXO E: Código completo en Java del JHotDraw 5.1

La aplicación del apartado 3.5 (JHotDraw 5.1) consta de once paquetes:

- Paquete CH.ifa.draw.util.
- Paquete CH.ifa.draw.framework.
- Paquete CH.ifa.draw.standard.
- Paquete CH.ifa.draw.figures.
- Paquete CH.ifa.draw.contrib.
- Paquete CH.ifa.draw.applet.
- Paquete CH.ifa.draw.application.
- Paquete CH.ifa.draw.samples.javadraw.
- Paquete CH.ifa.draw.samples.pert.
- Paquete CH.ifa.draw.samples.nothing.
- Paquete CH.ifa.draw.samples.net.

Paquete CH.ifa.draw.util

Este paquete proporciona utilidades que pueden ser utilizadas independientemente del JHotDraw.

Este paquete contiene lo siguiente:

- Interface Animatable.
- Interface PaletteListener.
- Interface Storable.
- Clase Clipboard.
- Clase ColorMap.
- Clase Command.
- Clase CommandButton.
- Clase CommandChoice.
- Clase CommandMenu.
- Clase Filler.
- Clase FloatingTextField.
- Clase Geom.
- Clase Iconkit.
- Clase PaletteButton.
- Clase PaletteIcon.
- Clase PaletteLayout.
- Clase ReverseVectorEnumerator.
- Clase StorableInput.
- Clase StorableOutput.

Interface Animatable

Animatable define un interface simple de animación.

Se encuentra en el fichero Animatable.java y su contenido es el siguiente:

```
/*
 * @(#)Animatable.java 5.1
 *
 */

package CH.ifa.draw.util;

/**
 * Animatable defines a simple animation interface
 */
public interface Animatable {
    /**
```

```
    * Perform a step of the animation.
    */
    void animationStep();
}
```

Interface PaletteListener

Interface para manejar eventos sobre la paleta.

Se encuentra en el fichero PaletteListener.java y su contenido es el siguiente:

```
/*
 * @(#)PaletteListener.java 5.1
 *
 */

package CH.ifa.draw.util;

/**
 * Interface for handling palette events.
 *
 * @see PaletteButton
 */

public interface PaletteListener {
    /**
     * The user selected a palette entry. The selected button is
     * passed as an argument.
     */
    void paletteUserSelected(PaletteButton button);

    /**
     * The user moved the mouse over the palette entry.
     */
    void paletteUserOver(PaletteButton button, boolean inside);
}
```

Interface Storable

Interface que es utilizado por StorableInput y StorableOutput para guardar y recuperar objetos. Los objetos que implementan este interface y que son recuperados por StorableInput tienen que proporcionar un constructor por defecto sin argumentos.

Se encuentra en el fichero Storable.java y su contenido es el siguiente:

```
/*
 * @(#)Storable.java 5.1
 *
 */
```

```
package CH.ifa.draw.util;

import java.io.*;

/**
 * Interface that is used by StorableInput and StorableOutput
 * to flatten and resurrect objects. Objects that implement
 * this interface and that are resurrected by StorableInput
 * have to provide a default constructor with no arguments.
 *
 * @see StorableInput
 * @see StorableOutput
 */
public interface Storable {
    /**
     * Writes the object to the StorableOutput.
     */
    public void write(StorableOutput dw);

    /**
     * Reads the object from the StorableInput.
     */
    public void read(StorableInput dr) throws IOException;
}
```

Clase Clipboard

Un suplente temporal para un portapapeles global. Es un singleton que puede ser utilizado para almacenar y conseguir los contenidos del portapapeles.

Se encuentra en el fichero Clipboard.java y su contenido es el siguiente:

```
/*
 * @(#)Clipboard.java 5.1
 *
 */

package CH.ifa.draw.util;

import java.awt.*;
import java.util.*;

/**
 * A temporary replacement for a global clipboard.
 * It is a singleton that can be used to store and
 * get the contents of the clipboard.
 *
 */
public class Clipboard {
    static Clipboard fgClipboard = new Clipboard();

    /**
     * Gets the clipboard.
     */
}
```

```

static public Clipboard getClipboard() {
    return fgClipboard;
}

private Object fContents;

private Clipboard() {
}

/**
 * Sets the contents of the clipboard.
 */
public void setContents(Object contents) {
    fContents = contents;
}

/**
 * Gets the contents of the clipboard.
 */
public Object getContents() {
    return fContents;
}
}

```

Clase ColorMap

Un mapa que es rellenado con algunos colores estándar. Los colores se pueden buscar por su nombre o por su índice.

Se encuentra en el fichero ColorMap.java y su contenido es el siguiente:

```

/*
 * @(#)ColorMap.java 5.1
 *
 */

package CH.ifa.draw.util;

import java.awt.Color;

class ColorEntry {
    public String fName;
    public Color fColor;

    ColorEntry(String name, Color color) {
        fColor = color;
        fName = name;
    }
}

/**
 * A map that is filled with some standard colors. The colors

```

```

* can be looked up by name or index.
*
*/

public class ColorMap
    extends Object {

    static ColorEntry fMap[] = {
        new ColorEntry("Black",          Color.black),
        new ColorEntry("Blue",           Color.blue),
        new ColorEntry("Green",          Color.green),
        new ColorEntry("Red",            Color.red),
        new ColorEntry("Pink",           Color.pink),
        new ColorEntry("Magenta",        Color.magenta),
        new ColorEntry("Orange",         Color.orange),
        new ColorEntry("Yellow",         Color.yellow),
        new ColorEntry("New Tan",        new Color(0xEBC79E)),
        new ColorEntry("Aquamarine",     new Color(0x70DB93)),
        new ColorEntry("Sea Green",      new Color(0x238E68)),
        new ColorEntry("Dark Gray",      Color.darkGray),
        new ColorEntry("Light Gray",     Color.lightGray),
        new ColorEntry("White",          Color.white),
        // there is no support for alpha values so we use a
        // special value
        // to represent a transparent color
        new ColorEntry("None",          new Color(0xFFC79E))
    };

    public static int size() {
        return fMap.length;
    }

    public static Color color(int index) {
        if (index < size() && index >= 0)
            return fMap[index].fColor;

        throw new ArrayIndexOutOfBoundsException("Color index: " +
index);
    }

    public static Color color(String name) {
        for (int i = 0; i < fMap.length; i++)
            if (fMap[i].fName.equals(name))
                return fMap[i].fColor;

        return Color.black;
    }

    public static String name(int index) {
        if (index < size() && index >= 0)
            return fMap[index].fName;

        throw new ArrayIndexOutOfBoundsException("Color index: " +
index);
    }

    public static int colorIndex(Color color) {

```

```

        for (int i=0; i<fMap.length; i++)
            if (fMap[i].fColor.equals(color))
                return i;
        return 0;
    }

    public static boolean isTransparent(Color color) {
        return color.equals(color("None"));
    }
}

```

Clase Command

Los comandos encapsulan una acción para ser ejecutada. Los comandos tienen un nombre y pueden ser utilizados en conjunción con los componentes de los interfaces del usuario *Command enabled*.

Patrones de diseño

Command Command es una simple instancia del patrón Command sin soportar la operación undo (deshacer).

Se encuentra en el fichero Command.java y su contenido es el siguiente:

```

/*
 * @(#)Command.java 5.1
 *
 */

package CH.ifa.draw.util;

import java.awt.*;
import java.util.*;

/**
 * Commands encapsulate an action to be executed. Commands have
 * a name and can be used in conjunction with Command enabled
 * ui components.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld010.htm>Command</a></b><br>
 * Command is a simple instance of the command pattern without undo
 * support.
 * <hr>
 *
 * @see CommandButton
 * @see CommandMenu
 * @see CommandChoice
 */
public abstract class Command {

```

```

private String fName;

/**
 * Constructs a command with the given name.
 */
public Command(String name) {
    fName = name;
}

/**
 * Executes the command.
 */
public abstract void execute();

/**
 * Tests if the command can be executed.
 */
public boolean isExecutable() {
    return true;
}

/**
 * Gets the command name.
 */
public String name() {
    return fName;
}
}

```

Clase CommandButton

Un Command habilita un botón. Al hacer clic sobre el botón se ejecuta el comando.

Se encuentra en el fichero CommandButton.java y su contenido es el siguiente:

```

/**
 * @(#)CommandButton.java 5.1
 */

package CH.ifa.draw.util;

import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.util.*;

/**
 * A Command enabled button. Clicking the button executes the command.
 *
 * @see Command
 */

public class CommandButton

```

```

        extends Button implements ActionListener {

    private Command    fCommand;

    /**
     * Initializes the button with the given command.
     * The command's name is used as the label.
     */
    public CommandButton(Command command) {
        super(command.name());
        fCommand = command;
        addActionListener(this);
    }

    /**
     * Executes the command. If the command's name was changed
     * as a result of the command the button's label is updated
     * accordingly.
     */
    public void actionPerformed(ActionEvent e) {
        fCommand.execute();
        if (!getLabel().equals(fCommand.name())) {
            setLabel(fCommand.name());
        }
    }
}

```

Clase CommandChoice

Un Command habilita la elección. Al seleccionar una opción se ejecuta el comando correspondiente.

Se encuentra en el fichero CommandChoice.java y su contenido es el siguiente:

```

/*
 * @(#)CommandChoice.java 5.1
 *
 */

package CH.ifa.draw.util;

import java.awt.*;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.util.*;

/**
 * A Command enabled choice. Selecting a choice executes the
 * corresponding command.
 *
 * @see Command
 */
public class CommandChoice
    extends Choice implements ItemListener {

```



```

private Vector    fCommands;

public CommandChoice() {
    fCommands = new Vector(10);
    addItemListener(this);
}

/**
 * Adds a command to the menu.
 */
public synchronized void addItem(Command command) {
    addItem(command.name());
    fCommands.addElement(command);
}

/**
 * Executes the command.
 */
public void itemStateChanged(ItemEvent e) {
    Command command =
(Command)fCommands.elementAt(getSelectedIndex());
    command.execute();
}
}

```

Clase CommandMenu

Un Command habilita el menú. Al seleccionar una opción del menú se ejecuta el comando correspondiente.

Se encuentra en el fichero CommandMenu.java y su contenido es el siguiente:

```

/*
 * @(#)CommandMenu.java 5.1
 *
 */

package CH.ifa.draw.util;

import java.awt.*;
import java.util.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

/**
 * A Command enabled menu. Selecting a menu item
 * executes the corresponding command.
 *
 * @see Command
 */
public class CommandMenu
    extends Menu implements ActionListener {

```

```

private Vector fCommands;

public CommandMenu(String name) {
    super(name);
    fCommands = new Vector(10);
}

/**
 * Adds a command to the menu. The item's label is
 * the command's name.
 */
public synchronized void add(Command command) {
    MenuItem m = new MenuItem(command.name());
    m.addActionListener(this);
    add(m);
    fCommands.addElement(command);
}

/**
 * Adds a command with the given short cut to the menu. The item's
 * label is the command's name.
 */
public synchronized void add(Command command, MenuShortcut
shortcut) {
    MenuItem m = new MenuItem(command.name(), shortcut);
    m.setName(command.name());
    m.addActionListener(this);
    add(m);
    fCommands.addElement(command);
}

public synchronized void remove(Command command) {
    System.out.println("not implemented");
}

public synchronized void remove(MenuItem item) {
    System.out.println("not implemented");
}

/**
 * Changes the enabling/disabling state of a named menu item.
 */
public synchronized void enable(String name, boolean state) {
    for (int i = 0; i < getItemCount(); i++) {
        MenuItem item = getItem(i);
        if (name.equals(item.getLabel())) {
            item.setEnabled(state);
            return;
        }
    }
}

public synchronized void checkEnabled() {
    int j = 0;
    for (int i = 0; i < getItemCount(); i++) {
        MenuItem item = getItem(i);
        // ignore separators
        // a separator has a hyphen as its label
    }
}

```

```

        if (item.getLabel().equals("-"))
            continue;
        Command cmd = (Command)fCommands.elementAt(j);
        item.setEnabled(cmd.isExecutable());
        j++;
    }
}

/**
 * Executes the command.
 */
public void actionPerformed(ActionEvent e) {
    int j = 0;
    Object source = e.getSource();
    for (int i = 0; i < getItemCount(); i++) {
        MenuItem item = getItem(i);
        // ignore separators
        // a separator has a hyphen as its label
        if (item.getLabel().equals("-"))
            continue;
        if (source == item) {
            Command cmd = (Command)fCommands.elementAt(j);
            cmd.execute();
            break;
        }
        j++;
    }
}
}

```

Clase Filler

Un componente que puede ser utilizado para reservar espacio en blanco en un layout.

Se encuentra en el fichero Filler.java y su contenido es el siguiente:

```

/*
 * @(#)Filler.java 5.1
 *
 */

package CH.ifa.draw.util;

import java.awt.*;

/**
 * A component that can be used to reserve white space in a layout.
 */
public class Filler
    extends Canvas {

    private int    fWidth;
    private int    fHeight;
    private Color  fBackground;
}

```

```

public Filler(int width, int height) {
    this(width, height, null);
}

public Filler(int width, int height, Color background) {
    fWidth = width;
    fHeight = height;
    fBackground = background;
}

public Dimension getMinimumSize() {
    return new Dimension(fWidth, fHeight);
}

public Dimension getPreferredSize() {
    return getMinimumSize();
}

public Color getBackground() {
    if (fBackground != null)
        return fBackground;
    return super.getBackground();
}
}

```

Clase FloatingTextField

Un revestimiento en un campo de texto que es utilizado para editar una TextFigure. Un FloatingTextField requiere dos pasos de inicialización: En un primer paso se crea el revestimiento y en un segundo paso se puede posicionarse.

Se encuentra en el fichero FloatingTextField.java y su contenido es el siguiente:

```

/*
 * @(#)FloatingTextField.java 5.1
 *
 */

package CH.ifa.draw.util;

import java.awt.*;
import java.awt.event.*;

/**
 * A text field overlay that is used to edit a TextFigure.
 * A FloatingTextField requires a two step initialization:
 * In a first step the overlay is created and in a
 * second step it can be positioned.
 *
 * @see TextFigure
 */

```

```

public class FloatingTextField extends Object {

    private TextField    fEditWidget;
    private Container    fContainer;

    public FloatingTextField() {
        fEditWidget = new TextField(20);
    }

    /**
     * Creates the overlay for the given Component.
     */
    public void createOverlay(Container container) {
        createOverlay(container, null);
    }

    /**
     * Creates the overlay for the given Container using a
     * specific font.
     */
    public void createOverlay(Container container, Font font) {
        container.add(fEditWidget, 0);
        if (font != null)
            fEditWidget.setFont(font);
        fContainer = container;
    }

    /**
     * Adds an action listener
     */
    public void addActionListener(ActionListener listener) {
        fEditWidget.addActionListener(listener);
    }

    /**
     * Remove an action listener
     */
    public void removeActionListener(ActionListener listener) {
        fEditWidget.removeActionListener(listener);
    }

    /**
     * Positions the overlay.
     */
    public void setBounds(Rectangle r, String text) {
        fEditWidget.setText(text);
        fEditWidget.setBounds(r.x, r.y, r.width, r.height);
        fEditWidget.setVisible(true);
        fEditWidget.selectAll();
        fEditWidget.requestFocus();
    }

    /**
     * Gets the text contents of the overlay.
     */
    public String getText() {
        return fEditWidget.getText();
    }
}

```

```

    }

    /**
     * Gets the preferred size of the overlay.
     */
    public Dimension getPreferredSize(int cols) {
        return fEditWidget.getPreferredSize(cols);
    }

    /**
     * Removes the overlay.
     */
    public void endOverlay() {
        fContainer.requestFocus();
        if (fEditWidget == null)
            return;
        fEditWidget.setVisible(false);
        fContainer.remove(fEditWidget);

        Rectangle bounds = fEditWidget.getBounds();
        fContainer.repaint(bounds.x, bounds.y, bounds.width,
        bounds.height);
    }
}

```

Clase Geom

Algunas utilidades de geometría.

Se encuentra en el fichero Geom.java y su contenido es el siguiente:

```

/*
 * @(#)Geom.java 5.1
 *
 */

package CH.ifa.draw.util;

import java.awt.Rectangle;
import java.awt.Point;

/**
 * Some geometric utilities.
 */
public class Geom {

    private Geom() {}; // never instantiated

    /**
     * Tests if a point is on a line.
     */
    static public boolean lineContainsPoint(int x1, int y1,
                                           int x2, int y2,
                                           int px, int py) {

```

```

Rectangle r = new Rectangle(new Point(x1, y1));
r.add(x2, y2);

r.grow(2,2);
if (! r.contains(px,py))
    return false;

double a, b, x, y;

if (x1 == x2)
    return (Math.abs(px - x1) < 3);

if (y1 == y2)
    return (Math.abs(py - y1) < 3);

a = (double)(y1 - y2) / (double)(x1 - x2);
b = (double)y1 - a * (double)x1;
x = (py - b) / a;
y = a * px + b;

return (Math.min(Math.abs(x - px), Math.abs(y - py)) < 4);
}

static public final int NORTH = 1;
static public final int SOUTH = 2;
static public final int WEST = 3;
static public final int EAST = 4;

/**
 * Returns the direction NORTH, SOUTH, WEST, EAST from
 * one point to another one.
 */
static public int direction(int x1, int y1, int x2, int y2) {
    int direction = 0;
    int vx = x2 - x1;
    int vy = y2 - y1;

    if (vy < vx && vx > -vy)
        direction = EAST;
    else if (vy > vx && vy > -vx)
        direction = NORTH;
    else if (vx < vy && vx < -vy)
        direction = WEST;
    else
        direction = SOUTH;
    return direction;
}

static public Point south(Rectangle r) {
    return new Point(r.x + r.width / 2, r.y + r.height);
}

static public Point center(Rectangle r) {
    return new Point(r.x + r.width / 2, r.y + r.height/2);
}

static public Point west(Rectangle r) {

```

```

        return new Point(r.x, r.y + r.height/ 2);
    }

    static public Point east(Rectangle r) {
        return new Point(r.x+r.width, r.y + r.height/ 2);
    }

    static public Point north(Rectangle r) {
        return new Point(r.x+r.width/2, r.y);
    }

    /**
     * Constains a value to the given range.
     * @return the constrained value
     */
    static public int range(int min, int max, int value) {
        if (value < min)
            value = min;
        if (value > max)
            value = max;
        return value;
    }

    /**
     * Gets the square distance between two points.
     */
    static public long length2(int x1, int y1, int x2, int y2) {
        return (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1);
    }

    /**
     * Gets the distance between to points
     */
    static public long length(int x1, int y1, int x2, int y2) {
        return (long)Math.sqrt(length2(x1, y1, x2, y2));
    }

    /**
     * Gets the angle of a point relative to a rectangle.
     */
    static public double pointToAngle(Rectangle r, Point p) {
        int px = p.x - (r.x+r.width/2);
        int py = p.y - (r.y+r.height/2);
        return Math.atan2(py*r.width, px*r.height);
    }

    /**
     * Gets the point on a rectangle that corresponds to the given
    angle.
     */
    static public Point angleToPoint(Rectangle r, double angle) {
        double si = Math.sin(angle);
        double co = Math.cos(angle);
        double e = 0.0001;

        int x= 0, y= 0;
        if (Math.abs(si) > e) {
            x= (int) ((1.0 + co/Math.abs(si))/2.0 * r.width);

```



```

        x= range(0, r.width, x);
    } else if (co >= 0.0)
        x= r.width;
    if (Math.abs(co) > e) {
        y= (int) ((1.0 + si/Math.abs(co))/2.0 * r.height);
        y= range(0, r.height, y);
    } else if (si >= 0.0)
        y= r.height;
    return new Point(r.x + x, r.y + y);
}

/**
 * Converts a polar to a point
 */
static public Point polarToPoint(double angle, double fx, double
fy) {
    double si = Math.sin(angle);
    double co = Math.cos(angle);
    return new Point((int)(fx*co+0.5), (int)(fy*si+0.5));
}

/**
 * Gets the point on an oval that corresponds to the given angle.
 */
static public Point ovalAngleToPoint(Rectangle r, double angle) {
    Point center = Geom.center(r);
    Point p = Geom.polarToPoint(angle, r.width/2, r.height/2);
    return new Point(center.x + p.x, center.y + p.y);
}

/**
 * Standard line intersection algorithm
 * Return the point of intersection if it exists, else null
 */
// from Doug Lea's PolygonFigure
static public Point intersect(int xa, // line 1 point 1 x
                             int ya, // line 1 point 1 y
                             int xb, // line 1 point 2 x
                             int yb, // line 1 point 2 y
                             int xc, // line 2 point 1 x
                             int yc, // line 2 point 1 y
                             int xd, // line 2 point 2 x
                             int yd) { // line 2 point 2 y

    // source: http://vision.dai.ed.ac.uk/andrewfg/c-g-a-faq.html
    // eq: for lines AB and CD
    //      (YA-YC)(XD-XC)-(XA-XC)(YD-YC)
    // r = ----- (eqn 1)
    //      (XB-XA)(YD-YC)-(YB-YA)(XD-XC)
    //
    //      (YA-YC)(XB-XA)-(XA-XC)(YB-YA)
    // s = ----- (eqn 2)
    //      (XB-XA)(YD-YC)-(YB-YA)(XD-XC)
    // XI = XA + r(XB-XA)
    // YI = YA + r(YB-YA)

    double denom = ((xb - xa) * (yd - yc) - (yb - ya) * (xd - xc));

```

```

double rnum = ((ya - yc) * (xd - xc) - (xa - xc) * (yd - yc));

if (denom == 0.0) { // parallel
    if (rnum == 0.0) { // coincident; pick one end of first line
        if ((xa < xb && (xb < xc || xb < xd)) ||
            (xa > xb && (xb > xc || xb > xd)))
            return new Point(xb, yb);
        else
            return new Point(xa, ya);
    }
    else
        return null;
}

double r = rnum / denom;

double snum = ((ya - yc) * (xb - xa) - (xa - xc) * (yb - ya));
double s = snum / denom;

if (0.0 <= r && r <= 1.0 && 0.0 <= s && s <= 1.0) {
    int px = (int)(xa + (xb - xa) * r);
    int py = (int)(ya + (yb - ya) * r);
    return new Point(px, py);
}
else
    return null;
}
}

```

Clase IconKit

La clase IconKit soporta la compartición de imágenes. Mantiene un mapa de nombres de imagen y sus correspondiente imágenes. IconKit también soporta la carga de una colección de imágenes de un modo sincronizado. La resolución de un nombre de la ruta (path) para una imagen se delega al DrawingEditor.

Patrones de diseño

Singleton El IconKit es un singleton.

Se encuentra en el fichero IconKit.java y su contenido es el siguiente:

```

/*
 * @(#)Iconkit.java 5.1
 *
 */

package CH.ifa.draw.util;

import java.awt.*;
import java.awt.image.ImageProducer;
import java.util.*;

/**
 * The Iconkit class supports the sharing of images. It maintains
 * a map of image names and their corresponding images.
 *
 * Iconkit also supports to load a collection of images in
 * synchronized way.
 * The resolution of a path name to an image is delegated to the
 * DrawingEditor.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld031.htm>Singleton</a></b><br>
 * The Iconkit is a singleton.
 * <hr>
 */
public class Iconkit {
    private Hashtable      fMap;
    private Vector         fRegisteredImages;
    private Component      fComponent;
    private final static int ID = 123;
    private static Iconkit fgIconkit = null;
    private static boolean fgDebug = false;

    /**
     * Constructs an Iconkit that uses the given editor to
     * resolve image path names.
     */
    public Iconkit(Component component) {
        fMap = new Hashtable(53);
        fRegisteredImages = new Vector(10);
        fComponent = component;
        fgIconkit = this;
    }

    /**
     * Gets the single instance
     */
    public static Iconkit instance() {
        return fgIconkit;
    }

    /**
     * Loads all registered images.

```

```

    * @see #registerImage
    */
    public void loadRegisteredImages(Component component) {
        if (fRegisteredImages.size() == 0)
            return;

        MediaTracker tracker = new MediaTracker(component);

        // register images with MediaTracker
        Enumeration k = fRegisteredImages.elements();
        while (k.hasMoreElements()) {
            String fileName = (String) k.nextElement();
            if (basicGetImage(fileName) == null) {
                tracker.addImage(loadImage(fileName), ID);
            }
        }
        fRegisteredImages.removeAllElements();

        // block until all images are loaded
        try {
            tracker.waitForAll();
        } catch (Exception e) { }
    }

    /**
     * Registers an image that is then loaded together with
     * the other registered images by loadRegisteredImages.
     * @see #loadRegisteredImages
     */
    public void registerImage(String fileName) {
        fRegisteredImages.addElement(fileName);
    }

    /**
     * Registers and loads an image.
     */
    public Image registerAndLoadImage(Component component, String
fileName) {
        registerImage(fileName);
        loadRegisteredImages(component);
        return getImage(fileName);
    }

    /**
     * Loads an image with the given name.
     */
    public Image loadImage(String filename) {
        if (fMap.containsKey(filename))
            return (Image) fMap.get(filename);
        Image image = loadImageResource(filename);
        if (image != null)
            fMap.put(filename, image);
        return image;
    }

    public Image loadImageResource(String resourcename) {
        Toolkit toolkit = Toolkit.getDefaultToolkit();
        try {

```

```

        java.net.URL url = getClass().getResource(resourcename);
        if (fgDebug)
            System.out.println(resourcename);
        return toolkit.createImage((ImageProducer)
url.getContent());
    } catch (Exception ex) {
        return null;
    }
}

/**
 * Gets the image with the given name. If the image
 * can't be found it tries it again after loading
 * all the registered images.
 */
public Image getImage(String filename) {
    Image image = basicGetImage(filename);
    if (image != null)
        return image;
    // load registered images and try again
    loadRegisteredImages(fComponent);
    // try again
    return basicGetImage(filename);
}

private Image basicGetImage(String filename) {
    if (fMap.containsKey(filename))
        return (Image) fMap.get(filename);
    return null;
}
}

```

Clase PaletteButton

Una paleta de botones está en uno de los tres estados del botón. Utiliza el interface palette listener para notificar sobre los cambios de estado.

Se encuentra en el fichero PaletteButton.java y su contenido es el siguiente:

```

/*
 * @(#)PaletteButton.java 5.1
 *
 */

package CH.ifa.draw.util;

import java.awt.*;
import java.awt.event.*;

/**
 * A palette button is a three state button. The states are normal
 * pressed and selected. It uses to the palette listener interface
 * to notify about state changes.
 */

```

```

* @see PaletteListener
* @see PaletteLayout
*/

public abstract class PaletteButton
    extends Canvas
    implements MouseListener, MouseMotionListener {

    static final int NORMAL = 1;
    static final int PRESSED = 2;
    static final int SELECTED = 3;

    private PaletteListener fListener;
    private int fState;
    private int fOldState;

    /**
     * Constructs a PaletteButton.
     * @param listener the listener to be notified.
     */
    public PaletteButton(PaletteListener listener) {
        fListener = listener;
        fState = fOldState = NORMAL;
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    public abstract void paintBackground(Graphics g);
    public abstract void paintNormal(Graphics g);
    public abstract void paintPressed(Graphics g);
    public abstract void paintSelected(Graphics g);

    public Object value() {
        return null;
    }

    public String name() {
        return "";
    }

    public void reset() {
        fState = NORMAL;
        repaint();
    }

    public void select() {
        fState = SELECTED;
        repaint();
    }

    public void mousePressed(MouseEvent e) {
        fOldState = fState;
        fState = PRESSED;
        repaint();
    }

    public void mouseDragged(MouseEvent e) {

```

```

        if (contains(e.getX(),e.getY()))
            fState = PRESSED;
        else
            fState = fOldState;
        repaint();
    }

    public void mouseReleased(MouseEvent e) {
        fState = fOldState;
        repaint();
        if (contains(e.getX(),e.getY()))
            fListener.paletteUserSelected(this);
    }

    public void mouseMoved(MouseEvent e) {
        fListener.paletteUserOver(this, true);
    }

    public void mouseExited(MouseEvent e) {
        if (fState == PRESSED) // JDK1.1 on Windows sometimes loses
mouse released
            mouseDragged(e);
        fListener.paletteUserOver(this, false);
    }

    public void mouseClicked(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}

    public void update(Graphics g) {
        paint(g);
    }

    public void paint(Graphics g) {
        paintBackground(g);

        switch (fState) {
        case PRESSED:
            paintPressed(g);
            break;
        case SELECTED:
            paintSelected(g);
            break;
        case NORMAL:
        default:
            paintNormal(g);
            break;
        }
    }
}

```

Clase Palettelcon

Un icono de tres estados que puede ser utilizado en Palettes.

Se encuentra en el fichero PaletteIcon.java y su contenido es el siguiente:

```
/*
 * @(#)PaletteIcon.java 5.1
 *
 */

package CH.ifa.draw.util;

import java.awt.*;

/**
 * A three state icon that can be used in Palettes.
 *
 * @see PaletteButton
 */

public class PaletteIcon extends Object {

    Image      fNormal;
    Image      fPressed;
    Image      fSelected;
    Dimension  fSize;

    public PaletteIcon(Dimension size, Image normal, Image pressed,
Image selected) {
        fSize = size;
        fNormal = normal;
        fPressed = pressed;
        fSelected = selected;
    }

    public Image normal() { return fNormal; }
    public Image pressed() { return fPressed; }
    public Image selected() { return fSelected; }

    public int getWidth() { return fSize.width; }
    public int getHeight() { return fSize.height; }

}
```

Clase PaletteLayout

Un manejador layout del cliente para la paleta.

Se encuentra en el fichero PaletteLayout.java y su contenido es el siguiente:


```

/*
 * @(#)PaletteLayout.java 5.1
 *
 */

package CH.ifa.draw.util;

import java.awt.*;

/**
 * A custom layout manager for the palette.
 * @see PaletteButton
 */
public class PaletteLayout
    implements LayoutManager {

    private int          fGap;
    private Point        fBorder;
    private boolean      fVerticalLayout;

    /**
     * Initializes the palette layout.
     * @param gap the gap between palette entries.
     */
    public PaletteLayout(int gap) {
        this(gap, new Point(0,0), true);
    }

    public PaletteLayout(int gap, Point border) {
        this(gap, border, true);
    }

    public PaletteLayout(int gap, Point border, boolean vertical) {
        fGap = gap;
        fBorder = border;
        fVerticalLayout = vertical;
    }

    public void addLayoutComponent(String name, Component comp) {
    }

    public void removeLayoutComponent(Component comp) {
    }

    public Dimension preferredLayoutSize(Container target) {
        return minimumLayoutSize(target);
    }

    public Dimension minimumLayoutSize(Container target) {
        Dimension dim = new Dimension(0, 0);
        int nmembers = target.getComponentCount();

        for (int i = 0 ; i < nmembers ; i++) {
            Component m = target.getComponent(i);
            if (m.isVisible()) {
                Dimension d = m.getMinimumSize();

```

```

        if (fVerticalLayout) {
            dim.width = Math.max(dim.width, d.width);
            if (i > 0)
                dim.height += fGap;
            dim.height += d.height;
        } else {
            dim.height = Math.max(dim.height, d.height);
            if (i > 0)
                dim.width += fGap;
            dim.width += d.width;
        }
    }
}

Insets insets = target.getInsets();
dim.width += insets.left + insets.right;
dim.width += 2 * fBorder.x;
dim.height += insets.top + insets.bottom;
dim.height += 2 * fBorder.y;
return dim;
}

public void layoutContainer(Container target) {
    Insets insets = target.getInsets();
    int nmembers = target.getComponentCount();
    int x = insets.left + fBorder.x;
    int y = insets.top + fBorder.y;

    for (int i = 0 ; i < nmembers ; i++) {
        Component m = target.getComponent(i);
        if (m.isVisible()) {
            Dimension d = m.getMinimumSize();
            m.setBounds(x, y, d.width, d.height);
            if (fVerticalLayout) {
                y += d.height;
                y += fGap;
            } else {
                x += d.width;
                x += fGap;
            }
        }
    }
}
}
}
}
}

```

Clase ReverseVectorEnumerator

Un Enumeration que enumera un vector desde atrás (size-1) hacia delante (0).

Se encuentra en el fichero ReverseVectorEnumerator.java y su contenido es el siguiente:

```

/*
 * @(#)ReverseVectorEnumerator.java 5.1
 */

```

```

*/
package CH.ifa.draw.util;

import java.util.*;

/**
 * An Enumeration that enumerates a vector back (size-1) to front (0).
 */
public class ReverseVectorEnumerator
implements Enumeration {

    Vector vector;
    int count;

    public ReverseVectorEnumerator(Vector v) {
        vector = v;
        count = vector.size() - 1;
    }

    public boolean hasMoreElements() {
        return count >= 0;
    }

    public Object nextElement() {
        if (count >= 0) {
            return vector.elementAt(count--);
        }
        throw new NoSuchElementException("ReverseVectorEnumerator");
    }
}

```

Clase StorableInput

Un stream de entrada que puede ser utilizado para recuperar objetos Storable. StorableInput preserva la identidad del objeto de los objetos guardados.

Se encuentra en el fichero StorableInput.java y su contenido es el siguiente:

```

/*
 * @(#)StorableInput.java 5.1
 *
 */

package CH.ifa.draw.util;

import java.util.*;
import java.io.*;
import java.awt.Color;

/**
 * An input stream that can be used to resurrect Storable objects.
 * StorableInput preserves the object identity of the stored objects.
 */

```

```

*
* @see Storable
* @see StorableOutput
*/

public class StorableInput
    extends Object {

    private StreamTokenizer fTokenizer;
    private Vector          fMap;

    /**
     * Initializes a Storable input with the given input stream.
     */
    public StorableInput(InputStream stream) {
        Reader r = new BufferedReader(new InputStreamReader(stream));
        fTokenizer = new StreamTokenizer(r);
        fMap = new Vector();
    }

    /**
     * Reads and resurrects a Storable object from the input stream.
     */
    public Storable readStorable() throws IOException {
        Storable storable;
        String s = readString();

        if (s.equals("NULL"))
            return null;

        if (s.equals("REF")) {
            int ref = readInt();
            return (Storable) retrieve(ref);
        }

        storable = (Storable) makeInstance(s);
        map(storable);
        storable.read(this);
        return storable;
    }

    /**
     * Reads a string from the input stream.
     */
    public String readString() throws IOException {
        int token = fTokenizer.nextToken();
        if (token == StreamTokenizer.TT_WORD || token == '"') {
            return fTokenizer.sval;
        }

        String msg = "String expected in line: " +
fTokenizer.lineno();
        throw new IOException(msg);
    }

    /**
     * Reads an int from the input stream.
     */

```

```

        public int readInt() throws IOException {
            int token = fTokenizer.nextToken();
            if (token == StreamTokenizer.TT_NUMBER)
                return (int) fTokenizer.nval;

            String msg = "Integer expected in line: " +
fTokenizer.lineno();
            throw new IOException(msg);
        }

        /**
         * Reads a color from the input stream.
         */
        public Color readColor() throws IOException {
            return new Color(readInt(), readInt(), readInt());
        }

        /**
         * Reads a double from the input stream.
         */
        public double readDouble() throws IOException {
            int token = fTokenizer.nextToken();
            if (token == StreamTokenizer.TT_NUMBER)
                return fTokenizer.nval;

            String msg = "Double expected in line: " +
fTokenizer.lineno();
            throw new IOException(msg);
        }

        /**
         * Reads a boolean from the input stream.
         */
        public boolean readBoolean() throws IOException {
            int token = fTokenizer.nextToken();
            if (token == StreamTokenizer.TT_NUMBER)
                return ((int) fTokenizer.nval) == 1;

            String msg = "Integer expected in line: " +
fTokenizer.lineno();
            throw new IOException(msg);
        }

        private Object makeInstance(String className) throws IOException {
            try {
                Class cl = Class.forName(className);
                return cl.newInstance();
            } catch (NoSuchMethodError e) {
                throw new IOException("Class " + className
                    + " does not seem to have a no-arg constructor");
            } catch (ClassNotFoundException e) {
                throw new IOException("No class: " + className);
            } catch (InstantiationException e) {
                throw new IOException("Cannot instantiate: " + className);
            } catch (IllegalAccessException e) {
                throw new IOException("Class ( " + className + ") not
accessible");
            }
        }
    
```

```

    }

    private void map(Storable storable) {
        if (!fMap.contains(storable))
            fMap.addElement(storable);
    }

    private Storable retrieve(int ref) {
        return (Storable) fMap.elementAt(ref);
    }
}

```

Clase StorableOutput

Un stream de salida que puede ser utilizado para guardar objetos Storable. StorableOutput preserva la identidad del objeto de los objetos guardados.

Se encuentra en el fichero StorableOutput.java y su contenido es el siguiente:

```

/*
 * @(#)StorableOutput.java 5.1
 *
 */

package CH.ifa.draw.util;

import java.util.*;
import java.io.*;
import java.awt.Color;

/**
 * An output stream that can be used to flatten Storable objects.
 * StorableOutput preserves the object identity of the stored objects.
 *
 * @see Storable
 * @see StorableInput
 */

public class StorableOutput extends Object {

    private PrintWriter    fStream;
    private Vector          fMap;
    private int             fIndent;

    /**
     * Initializes the StorableOutput with the given output stream.
     */
    public StorableOutput(OutputStream stream) {
        fStream = new PrintWriter(stream);
        fMap = new Vector();
        fIndent = 0;
    }
}

```

```

/**
 * Writes a storable object to the output stream.
 */
public void writeStorable(Storable storable) {
    if (storable == null) {
        fStream.print("NULL");
        space();
        return;
    }

    if (mapped(storable)) {
        writeRef(storable);
        return;
    }

    incrementIndent();
    startNewLine();
    map(storable);
    fStream.print(storable.getClass().getName());
    space();
    storable.write(this);
    space();
    decrementIndent();
}

/**
 * Writes an int to the output stream.
 */
public void writeInt(int i) {
    fStream.print(i);
    space();
}

public void writeColor(Color c) {
    writeInt(c.getRed());
    writeInt(c.getGreen());
    writeInt(c.getBlue());
}

/**
 * Writes an int to the output stream.
 */
public void writeDouble(double d) {
    fStream.print(d);
    space();
}

/**
 * Writes an int to the output stream.
 */
public void writeBoolean(boolean b) {
    if (b)
        fStream.print(1);
    else
        fStream.print(0);
    space();
}

```

```

/**
 * Writes a string to the output stream. Special characters
 * are quoted.
 */
public void writeString(String s) {
    fStream.print(' ');
    for(int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        switch(c) {
            case '\n': fStream.print('\\n'); fStream.print(' ');
break;
            case '"' : fStream.print('\\'); fStream.print('"');
break;
            case '\\': fStream.print('\\'); fStream.print('\\');
break;
            case '\t': fStream.print('\\t'); fStream.print(' ');
break;
            default: fStream.print(c);
        }

        }
    fStream.print(' ');
    space();
}

/**
 * Closes a storable output stream.
 */
public void close() {
    fStream.close();
}

private boolean mapped(Storable storable) {
    return fMap.contains(storable);
}

private void map(Storable storable) {
    if (!fMap.contains(storable))
        fMap.addElement(storable);
}

private void writeRef(Storable storable) {
    int ref = fMap.indexOf(storable);

    fStream.print("REF");
    space();
    fStream.print(ref);
    space();
}

private void incrementIndent() {
    fIndent += 4;
}

private void decrementIndent() {
    fIndent -= 4;
    if (fIndent < 0) fIndent = 0;
}
}

```



```
private void startNewLine() {  
    fStream.println();  
    for (int i=0; i<fIndent; i++)  
        space();  
}  
  
private void space() {  
    fStream.print(' ');  
}  
}
```


Paquete CH.ifa.draw.framework

El paquete framework incluye las clases e interfaces que definen el framework JHotDraw. Este paquete no proporciona ninguna clase de implementación concreta.

Este paquete contiene lo siguiente:

- Interface ConnectionFigure.
- Interface Connector.
- Interface Drawing.
- Interface DrawingChangeListener.
- Interface DrawingEditor.
- Interface DrawingView.
- Interface Figure.
- Interface FigureChangeListener.
- Interface FigureEnumeration.
- Interface Handle.
- Interface Locator.
- Interface Painter.
- Interface PointConstrainer.
- Interface Tool.
- Clase DrawingChangeEvent.
- Clase FigureChangeEvent.
- Clase FigureSelection.
- Error HJDError.

Interface ConnectionFigure

Las Figures que conectan Connectors proporcionan Figures. Un ConnectionFigure conoce su Connector del principio y del final. Utiliza los Connectors para localizar sus puntos de conexión.

Patrones de diseño

Strategy Es utilizado para encapsular el algoritmo para localizar el punto de conexión. ConnectionFigure es el *Cliente* y Connector es el *StrategyAbstracto*.

Observer El Observer es utilizado para ver los cambios de las figuras conectadas. Una figure connection se registra ella misma como escuchadores (listeners) o observadores (observers) de los conectores fuente y destino.

Se encuentra en el fichero ConnectionFigure.java y su contenido es el siguiente:

```

/*
 * @(#)ConnectionFigure.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.Point;
import java.io.Serializable;
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

/**
 * Figures to connect Connectors provided by Figures.
 * A ConnectionFigure knows its start and end Connector.
 * It uses the Connectors to locate its connection points.<p>
 * A ConnectionFigure can have multiple segments. It provides
 * operations to split and join segments.
 *
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld034.htm>Strategy</a></b><br>
 * Strategy is used encapsulate the algorithm to locate the connection
 point.
 * ConnectionFigure is the Strategy context and Connector is the
 Strategy.<br>
 * 
 * <b><a href=../pattlets/sld026.htm>Observer</a></b><br>
 * Observer is used to track changes of the connected figures. A
 connection
 * figure registers itself as listeners or observers of the source and
 * target connector.
 * <hr>
 *
 * @see Connector
 */

public interface ConnectionFigure extends Figure, FigureChangeListener
{

    /**
     * Sets the start Connector of the connection.
     * @param figure the start figure of the connection
     */
    public void connectStart(Connector start);

    /**
     * Sets the end Connector of the connection.
     * @param figure the end figure of the connection
     */
    public void connectEnd(Connector end);

    /**
     * Updates the connection
     */
}

```

```

public void updateConnection();

/**
 * Disconnects the start figure from the dependent figure
 */
public void disconnectStart();

/**
 * Disconnects the end figure from the dependent figure
 */
public void disconnectEnd();

/**
 * Gets the start Connector
 */
public Connector start();

/**
 * Gets the end Connector.
 */
public Connector end();

/**
 * Checks if two figures can be connected. Implement this method
 * to constrain the allowed connections between figures.
 */
public boolean canConnect(Figure start, Figure end);

/**
 * Checks if the ConnectionFigure connects the same figures.
 */
public boolean connectsSame(ConnectionFigure other);

/**
 * Sets the start point.
 */
public void startPoint(int x, int y);

/**
 * Sets the end point.
 */
public void endPoint(int x, int y);

/**
 * Gets the start point.
 */
public Point startPoint();

/**
 * Gets the end point.
 */
public Point endPoint();

/**
 * Sets the position of the point at the given position
 */
public void setPointAt(Point p, int index);

```

```
/**
 * Gets the Point at the given position
 */
public Point pointAt(int index);

/**
 * Gets the number of points or nodes of the connection
 */
public int pointCount();

/**
 * Splits the hit segment.
 * @param x, y the position where the figure should be split
 * @return the index of the splitting point
 */
public int splitSegment(int x, int y);

/**
 * Joins the hit segments.
 * @param x, y the position where the figure should be joined.
 * @return whether the segment was joined
 */
public boolean joinSegments(int x, int y);
}
```

Interface Connector

Los Connectors saben como posicionar un punto de conexión sobre una figura. Un Connector conoce su propia figura y puede determinar el punto de inicio y el punto final de una figura de conexión dada. Un conector tiene una zona de representación (display box) que describe el area de una figura de la cual es responsable. Un conector puede estar visible pero puede no estarlo.

Patrones de diseño

Strategy Connector implementa el strategy para determinar los puntos de conexión.

Factory Method Los Connectors son creados por el factory method de la Figure connectorAt.

Se encuentra en el fichero Connector.java y su contenido es el siguiente:

```
/*
 * @(#)Connector.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.*;
```

```
import java.io.Serializable;

import CH.ifa.draw.util.*;

/**
 * Connectors know how to locate a connection point on a figure.
 * A Connector knows its owning figure and can determine either
 * the start or the endpoint of a given connection figure. A connector
 * has a display box that describes the area of a figure it is
 * responsible for. A connector can be visible but it doesn't have
 * to be.<br>
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld004.htm>Strategy</a></b><br>
 * Connector implements the strategy to determine the connections
 * points.<br>
 * 
 * <b><a href=../pattlets/sld016.htm>Factory Method</a></b><br>
 * Connectors are created by the Figure's factory method connectorAt.
 * <hr>
 *
 * @see Figure#connectorAt
 * @see ConnectionFigure
 */
public interface Connector extends Serializable, Storable {

    /**
     * Finds the start point for the connection.
     */
    public abstract Point findStart(ConnectionFigure connection);

    /**
     * Finds the end point for the connection.
     */
    public abstract Point findEnd(ConnectionFigure connection);

    /**
     * Gets the connector's owner.
     */
    public abstract Figure owner();

    /**
     * Gets the display box of the connector.
     */
    public abstract Rectangle displayBox();

    /**
     * Tests if a point is contained in the connector.
     */
    public abstract boolean containsPoint(int x, int y);

    /**
     * Draws this connector. Connectors don't have to be visible
     * and it is OK leave this method empty.
     */
    public abstract void draw(Graphics g);
}
```

```
}
```

Interface Drawing

Drawing es un contenedor para figuras.

Drawing envía los eventos `DrawingChanged` a `DrawingChangeListeners` cuando una parte de su area fue modificada.

Patrones de diseño

Observer El patrón Observer se utiliza para desacoplar el Drawing de sus vistas y permitir varias vistas.

Se encuentra en el fichero `Drawing.java` y su contenido es el siguiente:

```
/*
 * @(#)Drawing.java 5.1
 *
 */

package CH.ifa.draw.framework;

import CH.ifa.draw.util.*;
import java.awt.*;
import java.util.*;
import java.io.*;

/**
 * Drawing is a container for figures.
 * <p>
 * Drawing sends out DrawingChanged events to DrawingChangeListeners
 * whenever a part of its area was invalidated.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld026.htm>Observer</a></b><br>
 * The Observer pattern is used to decouple the Drawing from its views
 and
 * to enable multiple views.<hr>
 *
 * @see Figure
 * @see DrawingView
 * @see FigureChangeListener
 */

public interface Drawing
    extends Storable, FigureChangeListener, Serializable {

    /**
     * Releases the drawing and its contained figures.
     */
}
```



```

public void release();

/**
 * Returns an enumeration to iterate in
 * Z-order back to front over the figures.
 */
public FigureEnumeration figures();

/**
 * Returns an enumeration to iterate in
 * Z-order front to back over the figures.
 */
public FigureEnumeration figuresReverse();

/**
 * Finds a top level Figure. Use this call for hit detection that
 * should not descend into the figure's children.
 */
public Figure findFigure(int x, int y);

/**
 * Finds a top level Figure that intersects the given rectangle.
 */
public Figure findFigure(Rectangle r);

/**
 * Finds a top level Figure, but supresses the passed
 * in figure. Use this method to ignore a figure
 * that is temporarily inserted into the drawing.
 * @param x the x coordinate
 * @param y the y coordinate
 * @param without the figure to be ignored during
 * the find.
 */
public Figure findFigureWithout(int x, int y, Figure without);

/**
 * Finds a top level Figure that intersects the given rectangle.
 * It supresses the passed
 * in figure. Use this method to ignore a figure
 * that is temporarily inserted into the drawing.
 */
public Figure findFigure(Rectangle r, Figure without);

/**
 * Finds a figure but descends into a figure's
 * children. Use this method to implement <i>click-through</i>
 * hit detection, that is, you want to detect the inner most
 * figure containing the given point.
 */
public Figure findFigureInside(int x, int y);

/**
 * Finds a figure but descends into a figure's
 * children. It supresses the passed
 * in figure. Use this method to ignore a figure
 * that is temporarily inserted into the drawing.
 * @param x the x coordinate

```

```

    * @param y the y coordinate
    * @param without the figure to be ignored during
    * the find.
    */
    public Figure findFigureInsideWithout(int x, int y, Figure
without);

    /**
    * Adds a listener for this drawing.
    */
    public void addDrawingChangeListener(DrawingChangeListener
listener);

    /**
    * Removes a listener from this drawing.
    */
    public void removeDrawingChangeListener(DrawingChangeListener
listener);

    /**
    * Gets the listeners of a drawing.
    */
    public Enumeration drawingChangeListeners();

    /**
    * Adds a figure and sets its container to refer
    * to this drawing.
    * @return the figure that was inserted.
    */
    public Figure add(Figure figure);

    /**
    * Adds a vector of figures.
    */
    public void addAll(Vector newFigures);

    /**
    * Removes the figure from the drawing and releases it.
    */
    public Figure remove(Figure figure);

    /**
    * Removes a figure from the figure list, but
    * doesn't release it. Use this method to temporarily
    * manipulate a figure outside of the drawing.
    */
    public Figure orphan(Figure figure);

    /**
    * Removes a vector of figures from the figure's list
    * without releasing the figures.
    * @see orphan
    */
    public void orphanAll(Vector newFigures);

    /**
    * Removes a vector of figures .
    * @see remove

```

```
    */
    public void removeAll(Vector figures);

    /**
     * Replaces a figure in the drawing without
     * removing it from the drawing.
     */
    public void replace(Figure figure, Figure replacement);

    /**
     * Sends a figure to the back of the drawing.
     */
    public void sendToBack(Figure figure);

    /**
     * Brings a figure to the front.
     */
    public void bringToFront(Figure figure);

    /**
     * Draws all the figures back to front.
     */
    public void draw(Graphics g);

    /**
     * Invalidates a rectangle and merges it with the
     * existing damaged area.
     */
    public void figureInvalidated(FigureChangeEvent e);

    /**
     * Forces an update of the drawing change listeners.
     */
    public void figureRequestUpdate(FigureChangeEvent e);

    /**
     * Handles a removeFrfigureRequestRemove request that
     * is passed up the figure container hierarchy.
     * @see FigureChangeListener
     */
    public void figureRequestRemove(FigureChangeEvent e);

    /**
     * Acquires the drawing lock.
     */
    public void lock();

    /**
     * Releases the drawing lock.
     */
    public void unlock();
}
```

Interface DrawingChangeListener

Es un escuchador interesado en los cambios del Drawing.

Se encuentra en el fichero DrawingChangeListener.java y su contenido es el siguiente:

```
/*
 * @(#)DrawingChangeListener.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.Rectangle;
import java.util.EventListener;

/**
 * Listener interested in Drawing changes.
 */
public interface DrawingChangeListener extends EventListener {

    /**
     * Sent when an area is invalid
     */
    public void drawingInvalidated(DrawingChangeEvent e);

    /**
     * Sent when the drawing wants to be refreshed
     */
    public void drawingRequestUpdate(DrawingChangeEvent e);
}
```

Interface DrawingEditor

Define el interface para coordinar los diferentes objetos que participan en un editor de dibujo.

Patrones de diseño

Mediator DrawingEditor es el *Mediator*. Desacopla los participantes de un editor de dibujo.

Se encuentra en el fichero DrawingEditor.java y su contenido es el siguiente:

```
/*
 * @(#)DrawingEditor.java 5.1
 *
 */

package CH.ifa.draw.framework;
```

```
import java.awt.*;

/**
 * DrawingEditor defines the interface for coordinating
 * the different objects that participate in a drawing editor.
 *
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld022.htm>Mediator</a></b><br>
 * DrawingEditor is the mediator. It decouples the participants
 * of a drawing editor.
 *
 * @see Tool
 * @see DrawingView
 * @see Drawing
 */
public interface DrawingEditor {

    /**
     * Gets the editor's drawing view.
     */
    DrawingView view();

    /**
     * Gets the editor's drawing.
     */
    Drawing drawing();

    /**
     * Gets the editor's current tool.
     */
    Tool tool();

    /**
     * Informs the editor that a tool has done its interaction.
     * This method can be used to switch back to the default tool.
     */
    void toolDone();

    /**
     * Informs that the current selection has changed.
     * Override this method to handle selection changes.
     */
    void selectionChanged(DrawingView view);

    /**
     * Shows a status message in the editor's user interface
     */
    void showStatus(String string);
}
```

Interface DrawingView

DrawingView dibuja un Drawing y escucha sus cambios. Recibe entradas del usuario y las delega a la herramienta actual.

Patrones de diseño

Observer DrawingView observa dibujos que cambian a través del interface DrawingChangeListener.

State DrawingView juega el papel del *Contexto* en el patrón State. La herramienta es el *Estado*.

Strategy DrawingView es el *Cliente* en el patrón Strategy como corresponde al UpdateStrategy. DrawingView es el *Cliente* para el PointConstrainer.

Se encuentra en el fichero DrawingView.java y su contenido es el siguiente:

```

/*
 * @(#)DrawingView.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.image.ImageObserver;
import java.awt.*;
import java.util.*;
import java.io.*;
import CH.ifa.draw.util.*;

/**
 * DrawingView renders a Drawing and listens to its changes.
 * It receives user input and delegates it to the current tool.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld026.htm>Observer</a></b><br>
 * DrawingView observes drawing for changes via the DrawingListener
 interface.<br>
 * 
 * <b><a href=../pattlets/sld032.htm>State</a></b><br>
 * DrawingView plays the role of the StateContext in
 the State pattern. Tool is the State.<br>
 * 
 * <b><a href=../pattlets/sld034.htm>Strategy</a></b><br>
 * DrawingView is the StrategyContext in the Strategy pattern
 with regard to the UpdateStrategy. <br>
 * DrawingView is the StrategyContext for the PointConstrainer.
 */

```

```

* @see Drawing
* @see Painter
* @see Tool
*/

public interface DrawingView extends ImageObserver,
DrawingChangeListener {

    /**
     * Sets the view's editor.
     */
    public void setEditor(DrawingEditor editor);

    /**
     * Gets the current tool.
     */
    public Tool tool();

    /**
     * Gets the drawing.
     */
    public Drawing drawing();

    /**
     * Sets and installs another drawing in the view.
     */
    public void setDrawing(Drawing d);

    /**
     * Gets the editor.
     */
    public DrawingEditor editor();

    /**
     * Adds a figure to the drawing.
     * @return the added figure.
     */
    public Figure add(Figure figure);

    /**
     * Removes a figure from the drawing.
     * @return the removed figure
     */
    public Figure remove(Figure figure);

    /**
     * Adds a vector of figures to the drawing.
     */
    public void addAll(Vector figures);

    /**
     * Gets the size of the drawing.
     */
    public Dimension getSize();

    /**
     * Gets the minimum dimension of the drawing.

```

```

    */
    public Dimension getMinimumSize();

    /**
     * Gets the preferred dimension of the drawing..
     */
    public Dimension getPreferredSize();

    /**
     * Sets the current display update strategy.
     * @see UpdateStrategy
     */
    public void setDisplayUpdate(Painter updateStrategy);

    /**
     * Gets the currently selected figures.
     * @return a vector with the selected figures. The vector
     * is a copy of the current selection.
     */
    public Vector selection();

    /**
     * Gets an enumeration over the currently selected figures.
     */
    public FigureEnumeration selectionElements();

    /**
     * Gets the currently selected figures in Z order.
     * @see #selection
     * @return a vector with the selected figures. The vector
     * is a copy of the current selection.
     */
    public Vector selectionZOrdered();

    /**
     * Gets the number of selected figures.
     */
    public int selectionCount();

    /**
     * Adds a figure to the current selection.
     */
    public void addToSelection(Figure figure);

    /**
     * Adds a vector of figures to the current selection.
     */
    public void addToSelectionAll(Vector figures);

    /**
     * Removes a figure from the selection.
     */
    public void removeFromSelection(Figure figure);

    /**
     * If a figure isn't selected it is added to the selection.
     * Otherwise it is removed from the selection.
     */

```



```

public void toggleSelection(Figure figure);

/**
 * Clears the current selection.
 */
public void clearSelection();

/**
 * Gets the current selection as a FigureSelection. A
 * FigureSelection can be cut, copied, pasted.
 */
public FigureSelection getFigureSelection();

/**
 * Finds a handle at the given coordinates.
 * @return the hit handle, null if no handle is found.
 */
public Handle findHandle(int x, int y);

/**
 * Gets the position of the last click inside the view.
 */
public Point lastClick();

/**
 * Sets the current point constrainer.
 */
public void setConstrainer(PointConstrainer p);

/**
 * Gets the current grid setting.
 */
public PointConstrainer getConstrainer();

/**
 * Checks whether the drawing has some accumulated damage
 */
public void checkDamage();

/**
 * Repair the damaged area
 */
public void repairDamage();

/**
 * Paints the drawing view. The actual drawing is delegated to
 * the current update strategy.
 * @see Painter
 */
public void paint(Graphics g);

/**
 * Creates an image with the given dimensions
 */
public Image createImage(int width, int height);

/**
 * Gets a graphic to draw into

```

```

    */
    public Graphics getGraphics();

    /**
     * Gets the background color of the DrawingView
     */
    public Color getBackground();

    /**
     * Gets the background color of the DrawingView
     */
    public void setBackground(Color c);

    /**
     * Draws the contents of the drawing view.
     * The view has three layers: background, drawing, handles.
     * The layers are drawn in back to front order.
     */
    public void drawAll(Graphics g);

    /**
     * Draws the currently active handles.
     */
    public void drawHandles(Graphics g);

    /**
     * Draws the drawing.
     */
    public void drawDrawing(Graphics g);

    /**
     * Draws the background. If a background pattern is set it
     * is used to fill the background. Otherwise the background
     * is filled in the background color.
     */
    public void drawBackground(Graphics g);

    /**
     * Sets the cursor of the DrawingView
     */
    public void setCursor(Cursor c);

    /**
     * Freezes the view by acquiring the drawing lock.
     * @see Drawing#lock
     */
    public void freezeView();

    /**
     * Unfreezes the view by releasing the drawing lock.
     * @see Drawing#unlock
     */
    public void unfreezeView();
}

```

Interface Figure

El interface de una figura gráfica. Una figura conoce su caja de representación (display box) y puede dibujarse ella misma. Una figura puede estar compuesta de varias figuras. Para interactuar y manipular con una figura proporciona Handles y Connectors.

Una figura tiene un conjunto de manejadores que manipulan su forma o sus atributos. Una figura tiene uno o más conectores que definen como localizar y posicionar un punto de conexión.

Las figuras pueden tener un conjunto abierto de atributos. Un atributo es identificado por un string.

Las implementaciones por defecto para el interface Figure son proporcionadas por el AbstractFigure.

Se encuentra en el fichero Figure.java y su contenido es el siguiente:

```
/*
 * @(#)Figure.java 5.1
 *
 */

package CH.ifa.draw.framework;

import CH.ifa.draw.util.*;
import java.awt.*;
import java.util.*;
import java.io.Serializable;

/**
 * The interface of a graphical figure. A figure knows
 * its display box and can draw itself. A figure can be
 * composed of several figures. To interact and manipulate
 * with a figure it can provide Handles and Connectors.<p>
 * A figure has a set of handles to manipulate its shape or
 * attributes.
 * A figure has one or more connectors that define how
 * to locate a connection point.<p>
 * Figures can have an open ended set of attributes.
 * An attribute is identified by a string.<p>
 * Default implementations for the Figure interface are provided
 * by AbstractFigure.
 *
 * @see Handle
 * @see Connector
 * @see AbstractFigure
 */

public interface Figure
    extends Storable, Cloneable, Serializable {
```

```

/**
 * Moves the Figure to a new location.
 * @param x the x delta
 * @param y the y delta
 */
public void moveBy(int dx, int dy);

/**
 * Changes the display box of a figure. This method is
 * always implemented in figure subclasses.
 * It only changes
 * the displaybox and does not announce any changes. It
 * is usually not called by the client. Clients typically call
 * displayBox to change the display box.
 * @param origin the new origin
 * @param corner the new corner
 * @see #displayBox
 */
public void basicDisplayBox(Point origin, Point corner);

/**
 * Changes the display box of a figure. Clients usually
 * invoke this method. It changes the display box
 * and announces the corresponding changes.
 * @param origin the new origin
 * @param corner the new corner
 * @see #displayBox
 */
public void displayBox(Point origin, Point corner);

/**
 * Gets the display box of a figure
 * @see #basicDisplayBox
 */
public Rectangle displayBox();

/**
 * Draws the figure.
 * @param g the Graphics to draw into
 */
public void draw(Graphics g);

/**
 * Returns the handles used to manipulate
 * the figure. Handles is a Factory Method for
 * creating handle objects.
 *
 * @return a Vector of handles
 * @see Handle
 */
public Vector handles();

/**
 * Gets the size of the figure
 */
public Dimension size();

/**

```

```

    * Gets the figure's center
    */
    public Point center();

    /**
     * Checks if the Figure should be considered as empty.
     */
    public boolean isEmpty();

    /**
     * Returns an Enumeration of the figures contained in this figure
     */
    public FigureEnumeration figures();

    /**
     * Returns the figure that contains the given point.
     */
    public Figure findFigureInside(int x, int y);

    /**
     * Checks if a point is inside the figure.
     */
    public boolean containsPoint(int x, int y);

    /**
     * Returns a Clone of this figure
     */
    public Object clone();

    /**
     * Changes the display box of a figure. This is a
     * convenience method. Implementors should only
     * have to override basicDisplayBox
     * @see #displayBox
     */
    public void displayBox(Rectangle r);

    /**
     * Checks whether the given figure is contained in this figure.
     */
    public boolean includes(Figure figure);

    /**
     * Decomposes a figure into its parts. A figure is considered
     * as a part of itself.
     */
    public FigureEnumeration decompose();

    /**
     * Sets the Figure's container and registers the container
     * as a figure change listener. A figure's container can be
     * any kind of FigureChangeListener. A figure is not restricted
     * to have a single container.
     */
    public void addToContainer(FigureChangeListener c);

    /**
     * Removes a figure from the given container and unregisters

```

```

    * it as a change listener.
    */
    public void removeFromContainer(FigureChangeListener c);

    /**
     * Gets the Figure's listeners.
     */
    public FigureChangeListener listener();

    /**
     * Adds a listener for this figure.
     */
    public void addFigureChangeListener(FigureChangeListener l);

    /**
     * Removes a listener for this figure.
     */
    public void removeFigureChangeListener(FigureChangeListener l);

    /**
     * Releases a figure's resources. Release is called when
     * a figure is removed from a drawing. Informs the listeners that
     * the figure is removed by calling figureRemoved.
     */
    public void release();

    /**
     * Invalidates the figure. This method informs its listeners
     * that its current display box is invalid and should be
     * refreshed.
     */
    public void invalidate();

    /**
     * Informes that a figure is about to change such that its
     * display box is affected.
     * Here is an example of how it is used together with changed()
     * <pre>
     * public void move(int x, int y) {
     *     willChange();
     *     // change the figure's location
     *     changed();
     * }
     * </pre>
     * @see #invalidate
     * @see #changed
     */
    public void willChange();

    /**
     * Informes that a figure has changed its display box.
     * This method also triggers an update call for its
     * registered observers.
     * @see #invalidate
     * @see #willChange
     */
    public void changed();

```

```

/**
 * Checks if this figure can be connected
 */
public boolean canConnect();

/**
 * Gets a connector for this figure at the given location.
 * A figure can have different connectors at different locations.
 */
public Connector connectorAt(int x, int y);

/**
 * Sets whether the connectors should be visible.
 * Connectors can be optionally visible. Implement
 * this method and react on isVisible to turn the
 * connectors on or off.
 */
public void connectorVisibility(boolean isVisible);

/**
 * Returns the connection inset. This is only a hint that
 * connectors can use to determine the connection location.
 * The inset defines the area where the display box of a
 * figure should not be connected.
 */
public Insets connectionInsets();

/**
 * Returns the locator used to located connected text.
 */
public Locator connectedTextLocator(Figure text);

/**
 * Returns the named attribute or null if a
 * figure doesn't have an attribute.
 * All figures support the attribute names
 * FillColor and FrameColor
 */
public Object getAttribute(String name);

/**
 * Sets the named attribute to the new value
 */
public void setAttribute(String name, Object value);
}

```

Interface FigureChangeListener

Escuchador interesado en los cambios sobre Figure.

Se encuentra en el fichero FigureChangeListener.java y su contenido es el siguiente:

```

/*
 * @(#)FigureChangeListener.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.Rectangle;
import java.util.EventListener;

/**
 * Listener interested in Figure changes.
 *
 */
public interface FigureChangeListener extends EventListener {

    /**
     * Sent when an area is invalid
     */
    public void figureInvalidated(FigureChangeEvent e);

    /**
     * Sent when a figure changed
     */
    public void figureChanged(FigureChangeEvent e);

    /**
     * Sent when a figure was removed
     */
    public void figureRemoved(FigureChangeEvent e);

    /**
     * Sent when requesting to remove a figure.
     */
    public void figureRequestRemove(FigureChangeEvent e);

    /**
     * Sent when an update should happen.
     */
    public void figureRequestUpdate(FigureChangeEvent e);
}

```

Interface FigureEnumeration

Interface de Enumeration que accede a Figures. Proporciona un método nextFigure, que se esconde bajo la función del código del cliente.

Se encuentra en el fichero FigureEnumeration.java y su contenido es el siguiente:

```

/*
 * @(#)FigureEnumeration.java 5.1
 *

```



```

*/

package CH.ifa.draw.framework;

import java.util.*;

/**
 * Interface for Enumerations that access Figures.
 * It provides a method nextFigure, that hides the down casting
 * from client code.
 */
public interface FigureEnumeration extends Enumeration {
    /**
     * Returns the next element of the enumeration. Calls to this
     * method will enumerate successive elements.
     * @exception NoSuchElementException If no more elements exist.
     */
    public Figure nextFigure();
}

```

Interface Handle

Handles son utilizados para cambiar una figura por manipulación directa. Los manejadores conocen sus propias figuras y proporcionan métodos para localizar y ubicar el manejo sobre la figura y seguir la pista a los cambios.

Patrones de diseño

Adapter Los Handles adaptan las operaciones que manipulan una figura a un interface común.

Se encuentra en el fichero Handle.java y su contenido es el siguiente:

```

/*
 * @(#)Handle.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.*;

/**
 * Handles are used to change a figure by direct manipulation.
 * Handles know their owning figure and they provide methods to
 * locate the handle on the figure and to track changes.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld004.htm>Adapter</a></b><br>
 * Handles adapt the operations to manipulate a figure to a common
 interface.

```

```

*
* @see Figure
*/
public interface Handle {

    public static final int HANDLESIZE = 8;

    /**
     * Locates the handle on the figure. The handle is drawn
     * centered around the returned point.
     */
    public abstract Point locate();

    /**
     * @deprecated As of version 4.1,
     * use invokeStart(x, y, drawingView)
     * Tracks the start of the interaction. The default implementation
     * does nothing.
     * @param x the x position where the interaction started
     * @param y the y position where the interaction started
     */
    public void invokeStart(int x, int y, Drawing drawing);

    /**
     * @deprecated As of version 4.1,
     * use invokeStart(x, y, drawingView)
     * Tracks the start of the interaction. The default implementation
     * does nothing.
     * @param x the x position where the interaction started
     * @param y the y position where the interaction started
     * @param view the handles container
     */
    public void invokeStart(int x, int y, DrawingView view);

    /**
     * @deprecated As of version 4.1,
     * use invokeStep(x, y, anchorX, anchorY, drawingView)
     *
     * Tracks a step of the interaction.
     * @param dx x delta of this step
     * @param dy y delta of this step
     */
    public void invokeStep (int dx, int dy, Drawing drawing);

    /**
     * Tracks a step of the interaction.
     * @param x the current x position
     * @param y the current y position
     * @param anchorX the x position where the interaction started
     * @param anchorY the y position where the interaction started
     */
    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view);

    /**
     * Tracks the end of the interaction.
     * @param x the current x position
     * @param y the current y position

```

```

    * @param anchorX the x position where the interaction started
    * @param anchorY the y position where the interaction started
    */
    public void invokeEnd(int x, int y, int anchorX, int anchorY,
DrawingView view);

    /**
    * @deprecated As of version 4.1,
    * use invokeEnd(x, y, anchorX, anchorY, drawingView).
    *
    * Tracks the end of the interaction.
    */
    public void invokeEnd (int dx, int dy, Drawing drawing);

    /**
    * Gets the handle's owner.
    */
    public Figure owner();

    /**
    * Gets the display box of the handle.
    */
    public Rectangle displayBox();

    /**
    * Tests if a point is contained in the handle.
    */
    public boolean containsPoint(int x, int y);

    /**
    * Draws this handle.
    */
    public void draw(Graphics g);
}

```

Interface Locator

Los Locators pueden ser utilizados para localizar una posición sobre una figura.

Patrones de diseño

Strategy Locator encapsula la estrategia para localizar una manipulación sobre una figura.

Se encuentra en el fichero Locator.java y su contenido es el siguiente:

```

/*
 * @(#)Locator.java 5.1
 *
 */

package CH.ifa.draw.framework;

```

```
import CH.ifa.draw.util.Storable;
import java.awt.*;
import java.io.Serializable;

/**
 * Locators can be used to locate a position on a figure.<p>
 *
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld034.htm>Strategy</a></b><br>
 * Locator encapsulates the strategy to locate a handle on a figure.
 */

public interface Locator extends Storable, Serializable, Cloneable {

    /**
     * Locates a position on the passed figure.
     * @return a point on the figure.
     */
    public Point locate(Figure owner);
}
```

Interface Painter

Painter define el interface para dibujar un estrato en un DawingView.

Patrones de diseño

Strategy Painter encapsula un algoritmo para dibujar algo en el DrawingView. El DrawingView juega el papel del *Cliente*.

Se encuentra en el fichero Painter.java y su contenido es el siguiente:

```
/*
 * @(#)Painter.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.*;
import java.io.Serializable;

/**
 * Painter defines the interface for drawing a layer
 * into a DrawingView.<p>
 *
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld034.htm>Strategy</a></b><br>
 * Painter encapsulates an algorithm to render something in
```

```

* the DrawingView. The DrawingView plays the role of the
* StrategyContext.
* <hr>
* @see DrawingView
*/

public interface Painter extends Serializable {

    /**
     * Draws into the given DrawingView.
     */
    public void draw(Graphics g, DrawingView view);

}

```

Interface PointConstrainer

Interface para encoger un Point. Puede utilizarse para implementar diferentes tipos de grids (cuadrículas).

Patrones de diseño

Strategy El DrawingView juega el papel del *Cliente*.

Se encuentra en el fichero PointConstrainer.java y su contenido es el siguiente:

```

/*
 * @(#)PointConstrainer.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.*;

/**
 * Interface to constrain a Point. This can be used to implement
 * different kinds of grids.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld034.htm>Strategy</a></b><br>
 * DrawingView is the StrategyContext.<br>
 *
 * @see DrawingView
 */

public interface PointConstrainer {

    /**
     * Constrains the given point.
     * @return constrained point.
     */
}

```

```

public Point constrainPoint(Point p);

/**
 * Gets the x offset to move an object.
 */
public int getStepX();

/**
 * Gets the y offset to move an object.
 */
public int getStepY();
}

```

Interface Tool

Una herramienta define un modo de presentación visual de dibujos. Todos los eventos de entrada sobre la representación visual del dibujo son enviados a su herramienta actual.

Las herramientas informan a su editor cuando se han realizado con una interacción llamando al método `toolDone()` del editor. Las herramientas son creadas una vez y reutilizadas. Son inicializadas y desinicializadas con `activate()` y `deactivate()` respectivamente.

Patrones de diseño

State Las Tools juegan el papel de *Estado* del patrón State. Al encapsular todos los estados especifican el comportamiento. `DrawingView` juega el papel de Contexto.

Se encuentra en el fichero `Tool.java` y su contenido es el siguiente:

```

/*
 * @(#)Tool.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.*;
import java.awt.event.MouseEvent;
import java.awt.event.KeyEvent;

/**
 * A tool defines a mode of the drawing view. All input events
 * targeted to the drawing view are forwarded to its current tool.<p>
 * Tools inform their editor when they are done with an interaction
 * by calling the editor's toolDone() method.
 * The Tools are created once and reused. They
 * are initialized/deinitialized with activate()/deactivate().
 * <hr>
 * <b>Design Patterns</b><P>
 * 

```

```

* <b><a href=../pattlets/sld032.htm>State</a></b><br>
* Tool plays the role of the State. In encapsulates all state
* specific behavior. DrawingView plays the role of the StateContext.
* @see DrawingView
*/

public interface Tool {

    /**
     * Activates the tool for the given view. This method is called
     * whenever the user switches to this tool. Use this method to
     * reinitialize a tool.
     */
    public void activate();

    /**
     * Deactivates the tool. This method is called whenever the user
     * switches to another tool. Use this method to do some clean-up
     * when the tool is switched. Subclassers should always call
     * super.deactivate.
     */
    public void deactivate();

    /**
     * Handles mouse down events in the drawing view.
     */
    public void mouseDown(MouseEvent e, int x, int y);

    /**
     * Handles mouse drag events in the drawing view.
     */
    public void mouseDrag(MouseEvent e, int x, int y);

    /**
     * Handles mouse up in the drawing view.
     */
    public void mouseUp(MouseEvent e, int x, int y);

    /**
     * Handles mouse moves (if the mouse button is up).
     */
    public void mouseMove(MouseEvent evt, int x, int y);

    /**
     * Handles key down events in the drawing view.
     */
    public void keyDown(KeyEvent evt, int key);
}

```

Clase DrawingChangeEvent

El evento pasado a los DrawingChangeListeners.

Se encuentra en el fichero DrawingChangeEvent.java y su contenido es el siguiente:

```

/*
 * @(#)DrawingChangeEvent.java 5.1
 *
 */

package CH.ifa.draw.framework;

import java.awt.Rectangle;
import java.util.EventObject;

/**
 * The event passed to DrawingChangeListeners.
 *
 */
public class DrawingChangeEvent extends EventObject {

    private Rectangle fRectangle;

    /**
     * Constructs a drawing change event.
     */
    public DrawingChangeEvent(Drawing source, Rectangle r) {
        super(source);
        fRectangle = r;
    }

    /**
     * Gets the changed drawing
     */
    public Drawing getDrawing() {
        return (Drawing)getSource();
    }

    /**
     * Gets the changed rectangle
     */
    public Rectangle getInvalidatedRectangle() {
        return fRectangle;
    }
}

```

Clase FigureChangeEvent

Evento FigureChange pasado a los FigureChangeListeners.

Se encuentra en el fichero FigureChangeEvent.java y su contenido es el siguiente:

```

/*
 * @(#)FigureChangeEvent.java 5.1
 *
 */

package CH.ifa.draw.framework;

```



```
import java.awt.Rectangle;
import java.util.EventObject;

/**
 * FigureChange event passed to FigureChangeListeners.
 */
public class FigureChangeEvent extends EventObject {

    private Rectangle fRectangle;
    private static final Rectangle fgEmptyRectangle = new
Rectangle(0, 0, 0, 0);

    /**
     * Constructs an event for the given source Figure. The rectangle
     is the
     * area to be invalidated.
     */
    public FigureChangeEvent(Figure source, Rectangle r) {
        super(source);
        fRectangle = r;
    }

    public FigureChangeEvent(Figure source) {
        super(source);
        fRectangle = fgEmptyRectangle;
    }

    /**
     * Gets the changed figure
     */
    public Figure getFigure() {
        return (Figure)getSource();
    }

    /**
     * Gets the changed rectangle
     */
    public Rectangle getInvalidatedRectangle() {
        return fRectangle;
    }
}
```

Clase FigureSelection

FigureSelection es capaz de transferir las figuras seleccionadas a un portapapeles

Se encuentra en el fichero FigureSelection.java y su contenido es el siguiente:

```
/*
 * @(#)FigureSelection.java 5.1
 */
```

```

package CH.ifa.draw.framework;

import CH.ifa.draw.util.*;
import java.util.*;
import java.io.*;

/**
 * FigureSelection enables to transfer the selected figures
 * to a clipboard.<p>
 * Will soon be converted to the JDK 1.1 Transferable interface.
 *
 * @see Clipboard
 */

public class FigureSelection extends Object {

    private byte[] fData; // flattend figures, ready to be resurrected
    /**
     * The type identifier of the selection.
     */
    public final static String TYPE = "CH.ifa.draw.Figures";

    /**
     * Constructes the Figure selection for the vecotor of figures.
     */
    public FigureSelection(Vector figures) {
        // a FigureSelection is represented as a flattened ByteStream
        // of figures.
        ByteArrayOutputStream output = new ByteArrayOutputStream(200);
        StorableOutput writer = new StorableOutput(output);
        writer.writeInt(figures.size());
        Enumeration selected = figures.elements();
        while (selected.hasMoreElements()) {
            Figure figure = (Figure) selected.nextElement();
            writer.writeStorable(figure);
        }
        writer.close();
        fData = output.toByteArray();
    }

    /**
     * Gets the type of the selection.
     */
    public String getType() {
        return TYPE;
    }

    /**
     * Gets the data of the selection. The result is returned
     * as a Vector of Figures.
     *
     * @return a copy of the figure selection.
     */
    public Object getData(String type) {
        if (type.equals(TYPE)) {
            InputStream input = new ByteArrayInputStream(fData);
            Vector result = new Vector(10);

```

```

        StorableInput reader = new StorableInput(input);
        int numRead = 0;
        try {
            int count = reader.readInt();
            while (numRead < count) {
                Figure newFigure = (Figure) reader.readStorable();
                result.addElement(newFigure);
                numRead++;
            }
        } catch (IOException e) {
            System.out.println(e.toString());
        }
        return result;
    }
    return null;
}
}

```

Error HJDError

Un error HJD.

Se encuentra en el fichero HJDError.java y su contenido es el siguiente:

```

/*
 * @(#)HJDError.java 5.1
 *
 */

package CH.ifa.draw.framework;

/**
 * A HJD Error.
 *
 */
public class HJDError extends Error {

    public HJDError(String msg) {
        super(msg);
    }
}

```


Paquete CH.ifa.draw.standard

El paquete `standard` proporciona implementaciones `standard` de las clases definidas en el paquete `framework`.

Proporciona las clases abstractas que implementan un interface del `framework` y proporcionan las implementaciones por defecto. La convención es añadir el prefijo `Abstract` a tales clases, por ejemplo `AbstractFigure`, `AbstractHandle`.

Las clases que implementan un interface del `framework` y pueden ser utilizadas tal como están empiezan por `Standard`, por ejemplo `StandardDrawing`, `StandardDrawingView`.

Este paquete contiene lo siguiente:

- Interface `TextHolder`.
- Clase `AbstractConnector`.
- Clase `AbstractFigure`.
- Clase `AbstractHandle`.
- Clase `AbstractLocator`.
- Clase `AbstractTool`.
- Clase `ActionTool`.
- Clase `AlignCommand`.
- Clase `BoxHandleKit`.
- Clase `BringToFrontCommand`.
- Clase `BufferedUpdateStrategy`.
- Clase `ChangeAttributeCommand`.
- Clase `ChangeConnectionEndHandle`.
- Clase `ChangeConnectionHandle`.
- Clase `ChangeConnectionStartHandle`.
- Clase `ChopBoxConnector`.
- Clase `CompositeFigure`.
- Clase `ConnectionHandle`.
- Clase `ConnectionTool`.
- Clase `CopyCommand`.
- Clase `CreationTool`.
- Clase `CutCommand`.
- Clase `DecoratorFigure`.
- Clase `DeleteCommand`.
- Clase `DragTracker`.
- Clase `DuplicateCommand`.
- Clase `FigureChangeEventMulticaster`.
- Clase `FigureEnumerator`.
- Clase `FigureTransferCommand`.
- Clase `GridConstrainer`.

- Clase HandleTracker.
- Clase LocatorConnector.
- Clase LocatorHandle.
- Clase NullHandle.
- Clase OffsetLocator.
- Clase PasteCommand.
- Clase RelativeLocator.
- Clase ReverseFigureEnumerator.
- Clase SelectAreaTracker.
- Clase SelectionTool.
- Clase SendToBackCommand.
- Clase SimpleUpdateStrategy.
- Clase StandardDrawing.
- Clase StandardDrawingView.
- Clase ToggleGridCommand.
- Clase ToolButton.

Interface TextHolder

El interface de una figura que tiene algún contenido de texto editable.

Se encuentra en el fichero TextHolder.java y su contenido es el siguiente:

```
/*
 * @(#)TextHolder.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.util.*;
import CH.ifa.draw.framework.*;

/**
 * The interface of a figure that has some editable text contents.
 * @see Figure
 */

public interface TextHolder {

    public Rectangle textDisplayBox();

    /**
     * Gets the text shown by the text figure.
     */
    public String getText();

    /**
     * Sets the text shown by the text figure.
     */
}
```

```

    */
    public void setText(String newText);

    /**
     * Tests whether the figure accepts typing.
     */
    public boolean acceptsTyping();

    /**
     * Gets the number of columns to be overlaid when the figure is
     * edited.
     */
    public int overlayColumns();

    /**
     * Connects a figure to another figure.
     */
    public void connect(Figure connectedFigure);

    /**
     * Gets the font.
     */
    public Font getFont();
}

```

Clase AbstractConnector

AbstractConnector proporciona una implementación por defecto para el interface Connector.

Se encuentra en el fichero AbstractConnector.java y su contenido es el siguiente:

```

/*
 * @(#)AbstractConnector.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.*;

/**
 * AbstractConnector provides default implementation for
 * the Connector interface.
 * @see Connector
 */
public abstract class AbstractConnector implements Connector {
    /**
     * the owner of the connector
     */
    private Figure      fOwner;
}

```

```

/*
 * Serialization support.
 */
private static final long serialVersionUID = -
5170007865562687545L;
private int abstractConnectorSerializedDataVersion = 1;

/**
 * Constructs a connector that has no owner. It is only
 * used internally to resurrect a connectors from a
 * StorableOutput. It should never be called directly.
 */
public AbstractConnector() {
    fOwner = null;
}

/**
 * Constructs a connector with the given owner figure.
 */
public AbstractConnector(Figure owner) {
    fOwner = owner;
}

/**
 * Gets the connector's owner.
 */
public Figure owner() {
    return fOwner;
}

public Point findStart(ConnectionFigure connection) {
    return findPoint(connection);
}

public Point findEnd(ConnectionFigure connection) {
    return findPoint(connection);
}

/**
 * Gets the connection point. Override when the connector
 * does not need to distinguish between the start and end
 * point of a connection.
 */
protected Point findPoint(ConnectionFigure connection) {
    return Geom.center(displayBox());
}

/**
 * Gets the display box of the connector.
 */
public Rectangle displayBox() {
    return owner().displayBox();
}

/**
 * Tests if a point is contained in the connector.
 */

```



```

public boolean containsPoint(int x, int y) {
    return owner().containsPoint(x, y);
}

/**
 * Draws this connector. By default connectors are invisible.
 */
public void draw(Graphics g) {
    // invisible by default
}

/**
 * Stores the connector and its owner to a StorableOutput.
 */
public void write(StorableOutput dw) {
    dw.writeStorable(fOwner);
}

/**
 * Reads the connector and its owner from a StorableInput.
 */
public void read(StorableInput dr) throws IOException {
    fOwner = (Figure)dr.readStorable();
}
}

```

Clase AbstractFigure

AbstractFigure proporciona una implementación por defecto para el interface Figure.

Patrones de diseño

Template Method Template Methods implementan un comportamiento por defecto e invariante para las subclases figuras.

Se encuentra en el fichero AbstractFigure.java y su contenido es el siguiente:

```

/*
 * @(#)AbstractFigure.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

import java.awt.*;
import java.util.*;
import java.io.*;

/**

```

```

* AbstractFigure provides default implementations for
* the Figure interface.
*
* <hr>
* <b>Design Patterns</b><P>
* 
* <b><a href=../pattlets/sld036.htm>Template Method</a></b><br>
* Template Methods implement default and invariant behavior for
* figure subclasses.
* <hr>
*
* @see Figure
* @see Handle
*/

public abstract class AbstractFigure implements Figure {

    /**
     * The listeners for a figure's changes.
     * @see #invalidate
     * @see #changed
     * @see #willChange
     */
    private transient FigureChangeListener fListener;

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = -10857585979273442L;
    private int abstractFigureSerializedDataVersion = 1;

    protected AbstractFigure() { }

    /**
     * Moves the figure by the given offset.
     */
    public void moveBy(int dx, int dy) {
        willChange();
        basicMoveBy(dx, dy);
        changed();
    }

    /**
     * Moves the figure. This is the
     * method that subclassers override. Clients usually
     * call displayBox.
     * @see moveBy
     */
    protected abstract void basicMoveBy(int dx, int dy);

    /**
     * Changes the display box of a figure. Clients usually
     * call this method. It changes the display box
     * and announces the corresponding change.
     * @param origin the new origin
     * @param corner the new corner
     * @see displayBox
     */
}

```

```

public void displayBox(Point origin, Point corner) {
    willChange();
    basicDisplayBox(origin, corner);
    changed();
}

/**
 * Sets the display box of a figure. This is the
 * method that subclasses override. Clients usually
 * call displayBox.
 * @see displayBox
 */
public abstract void basicDisplayBox(Point origin, Point corner);

/**
 * Gets the display box of a figure.
 */
public abstract Rectangle displayBox();

/**
 * Returns the handles of a Figure that can be used
 * to manipulate some of its attributes.
 * @return a Vector of handles
 * @see Handle
 */
public abstract Vector handles();

/**
 * Returns an Enumeration of the figures contained in this figure.
 * @see CompositeFigure
 */
public FigureEnumeration figures() {
    Vector figures = new Vector(1);
    figures.addElement(this);
    return new FigureEnumerator(figures);
}

/**
 * Gets the size of the figure. A convenience method.
 */
public Dimension size() {
    return new Dimension(displayBox().width, displayBox().height);
}

/**
 * Checks if the figure is empty. The default implementation
 * returns true if the width or height of its display box is < 3
 * @see Figure#isEmpty
 */
public boolean isEmpty() {
    return (size().width < 3) || (size().height < 3);
}

/**
 * Returns the figure that contains the given point.
 * In contrast to containsPoint it returns its
 * innermost figure that contains the point.
 */

```

```

    * @see #containsPoint
    */
    public Figure findFigureInside(int x, int y) {
        if (containsPoint(x, y))
            return this;
        return null;
    }

    /**
     * Checks if a point is inside the figure.
     */
    public boolean containsPoint(int x, int y) {
        return displayBox().contains(x, y);
    }

    /**
     * Changes the display box of a figure. This is a
     * convenience method. Implementors should only
     * have to override basicDisplayBox
     * @see displayBox
     */
    public void displayBox(Rectangle r) {
        displayBox(new Point(r.x, r.y), new Point(r.x+r.width,
r.y+r.height));
    }

    /**
     * Checks whether the given figure is contained in this figure.
     */
    public boolean includes(Figure figure) {
        return figure == this;
    }

    /**
     * Decomposes a figure into its parts. It returns a Vector
     * that contains itself.
     * @return an Enumeration for a Vector with itself as the
     * only element.
     */
    public FigureEnumeration decompose() {
        Vector figures = new Vector(1);
        figures.addElement(this);
        return new FigureEnumerator(figures);
    }

    /**
     * Sets the Figure's container and registers the container
     * as a figure change listener. A figure's container can be
     * any kind of FigureChangeListener. A figure is not restricted
     * to have a single container.
     */
    public void addToContainer(FigureChangeListener c) {
        addFigureChangeListener(c);
        invalidate();
    }

    /**
     * Removes a figure from the given container and unregisters

```

```

    * it as a change listener.
    */
    public void removeFromContainer(FigureChangeListener c) {
        invalidate();
        removeFigureChangeListener(c);
        changed();
    }

    /**
     * Adds a listener for this figure.
     */
    public void addFigureChangeListener(FigureChangeListener l) {
        fListener = FigureChangeEventMulticaster.add(fListener, l);
    }

    /**
     * Removes a listener for this figure.
     */
    public void removeFigureChangeListener(FigureChangeListener l) {
        fListener = FigureChangeEventMulticaster.remove(fListener, l);
    }

    /**
     * Gets the figure's listeners.
     */
    public FigureChangeListener listener() {
        return fListener;
    }

    /**
     * A figure is released from the drawing. You never call this
     * method directly. Release notifies its listeners.
     * @see Figure#release
     */
    public void release() {
        if (fListener != null)
            fListener.figureRemoved(new FigureChangeEvent(this));
    }

    /**
     * Invalidates the figure. This method informs the listeners
     * that the figure's current display box is invalid and should be
     * refreshed.
     */
    public void invalidate() {
        if (fListener != null) {
            Rectangle r = displayBox();
            r.grow(Handle.HANDLESIZE, Handle.HANDLESIZE);
            fListener.figureInvalidated(new FigureChangeEvent(this,
r));
        }
    }

    /**
     * Informes that a figure is about to change something that
     * affects the contents of its display box.
     *
     * @see Figure#willChange

```

```

    */
    public void willChange() {
        invalidate();
    }

    /**
     * Informs that a figure changed the area of its display box.
     *
     * @see FigureChangeEvent
     * @see Figure#changed
     */
    public void changed() {
        invalidate();
        if (fListener != null)
            fListener.figureChanged(new FigureChangeEvent(this));
    }

    /**
     * Gets the center of a figure. A convenience
     * method that is rarely overridden.
     */
    public Point center() {
        return Geom.center(displayBox());
    }

    /**
     * Checks if this figure can be connected. By default
     * AbstractFigures can be connected.
     */
    public boolean canConnect() {
        return true;
    }

    /**
     * Returns the connection inset. The connection inset
     * defines the area where the display box of a
     * figure can't be connected. By default the entire
     * display box can be connected.
     */
    public Insets connectionInsets() {
        return new Insets(0, 0, 0, 0);
    }

    /**
     * Returns the Figures connector for the specified location.
     * By default a ChopBoxConnector is returned.
     * @see ChopBoxConnector
     */
    public Connector connectorAt(int x, int y) {
        return new ChopBoxConnector(this);
    }

    /**
     * Sets whether the connectors should be visible.
     * By default they are not visible and
     */

```

```

public void connectorVisibility(boolean isVisible) {
}

/**
 * Returns the locator used to located connected text.
 */
public Locator connectedTextLocator(Figure text) {
    return RelativeLocator.center();
}

/**
 * Returns the named attribute or null if a
 * a figure doesn't have an attribute.
 * By default
 * figures don't have any attributes getAttribute
 * returns null.
 */
public Object getAttribute(String name) {
    return null;
}

/**
 * Sets the named attribute to the new value. By default
 * figures don't have any attributes and the request is ignored.
 */
public void setAttribute(String name, Object value) {
}

/**
 * Clones a figure. Creates a clone by using the storable
 * mechanism to flatten the Figure to stream followed by
 * resurrecting it from the same stream.
 *
 * @see Figure#clone
 */
public Object clone() {
    Object clone = null;
    ByteArrayOutputStream output = new ByteArrayOutputStream(200);
    try {
        ObjectOutputStream writer = new ObjectOutputStream(output);
        writer.writeObject(this);
        writer.close();
    } catch (IOException e) {
        System.out.println("Class not found: " + e);
    }

    InputStream input = new
ByteArrayInputStream(output.toByteArray());
    try {
        ObjectInput reader = new ObjectInputStream(input);
        clone = (Object) reader.readObject();
    } catch (IOException e) {
        System.out.println(e.toString());
    }
    catch (ClassNotFoundException e) {
        System.out.println("Class not found: " + e);
    }
    return clone;
}

```

```

    }

    /**
     * Stores the Figure to a StorableOutput.
     */
    public void write(StorableOutput dw) {
    }

    /**
     * Reads the Figure from a StorableInput.
     */
    public void read(StorableInput dr) throws IOException {
    }
}

```

Clase AbstractHandle

AbstractHandle proporciona una implementación por defecto para el interface Handle.

Se encuentra en el fichero AbstractHandle.java y su contenido es el siguiente:

```

/*
 * @(#)AbstractHandle.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.framework.*;
import java.awt.*;

/**
 * AbstractHandle provides default implementation for the
 * Handle interface.
 *
 * @see Figure
 * @see Handle
 */
public abstract class AbstractHandle implements Handle {

    /**
     * The standard size of a handle.
     */
    public static final int HANDLESIZE = 8;
    private Figure fOwner;

    /**
     * Initializes the owner of the figure.
     */
    public AbstractHandle(Figure owner) {
        fOwner = owner;
    }
}

```



```

/**
 * Locates the handle on the figure. The handle is drawn
 * centered around the returned point.
 */
public abstract Point locate();

/**
 * @ deprecated As of version 4.1,
 * use invokeStart(x, y, drawingView)
 * Tracks the start of the interaction. The default implementation
 * does nothing.
 * @param x the x position where the interaction started
 * @param y the y position where the interaction started
 */
public void invokeStart(int x, int y, Drawing drawing) { }

/**
 * @ deprecated As of version 4.1,
 * use invokeStart(x, y, drawingView)
 * Tracks the start of the interaction. The default implementation
 * does nothing.
 * @param x the x position where the interaction started
 * @param y the y position where the interaction started
 * @param view the handles container
 */
public void invokeStart(int x, int y, DrawingView view) {
    invokeStart(x, y, view.drawing());
}

/**
 * @ deprecated As of version 4.1,
 * use invokeStep(x, y, anchorX, anchorY, drawingView)
 *
 * Tracks a step of the interaction.
 * @param dx x delta of this step
 * @param dy y delta of this step
 */
public void invokeStep (int dx, int dy, Drawing drawing) { }

/**
 * Tracks a step of the interaction.
 * @param x the current x position
 * @param y the current y position
 * @param anchorX the x position where the interaction started
 * @param anchorY the y position where the interaction started
 */
public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
    invokeStep(x-anchorX, y-anchorY, view.drawing());
}

/**
 * Tracks the end of the interaction.
 * @param x the current x position
 * @param y the current y position
 * @param anchorX the x position where the interaction started
 * @param anchorY the y position where the interaction started
 */

```

```

    public void invokeEnd(int x, int y, int anchorX, int anchorY,
DrawingView view) {
        invokeEnd(x-anchorX, y-anchorY, view.drawing());
    }

    /**
     * @deprecated As of version 4.1,
     * use invokeEnd(x, y, anchorX, anchorY, drawingView).
     *
     * Tracks the end of the interaction.
     */
    public void invokeEnd (int dx, int dy, Drawing drawing) { }

    /**
     * Gets the handle's owner.
     */
    public Figure owner() {
        return fOwner;
    }

    /**
     * Gets the display box of the handle.
     */
    public Rectangle displayBox() {
        Point p = locate();
        return new Rectangle(
            p.x - HANDLESIZE / 2,
            p.y - HANDLESIZE / 2,
            HANDLESIZE,
            HANDLESIZE);
    }

    /**
     * Tests if a point is contained in the handle.
     */
    public boolean containsPoint(int x, int y) {
        return displayBox().contains(x, y);
    }

    /**
     * Draws this handle.
     */
    public void draw(Graphics g) {
        Rectangle r = displayBox();

        g.setColor(Color.white);
        g.fillRect(r.x, r.y, r.width, r.height);

        g.setColor(Color.black);
        g.drawRect(r.x, r.y, r.width, r.height);
    }
}

```

Clase AbstractLocator

AbstractLocator proporciona una implementación por defecto para el interface Locator.

Se encuentra en el fichero AbstractLocator.java y su contenido es el siguiente:

```

/*
 * @(#)AbstractLocator.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

import java.awt.*;
import java.util.*;
import java.io.IOException;

/**
 * AbstractLocator provides default implementations for
 * the Locator interface.
 *
 * @see Locator
 * @see Handle
 */
public abstract class AbstractLocator
    implements Locator, Storable, Cloneable {

    protected AbstractLocator() {
    }

    public Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
    }

    /**
     * Stores the arrow tip to a StorableOutput.
     */
    public void write(StorableOutput dw) {
    }

    /**
     * Reads the arrow tip from a StorableInput.
     */
    public void read(StorableInput dr) throws IOException {
    }
}

```

Clase AbstractTool

Implementación por defecto soportada por Tools.

Se encuentra en el fichero AbstractTool.java y su contenido es el siguiente:

```
/*
 * @(#)AbstractTool.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.awt.event.MouseEvent;
import java.awt.event.KeyEvent;
import CH.ifa.draw.framework.*;

/**
 * Default implementation support for Tools.
 *
 * @see DrawingView
 * @see Tool
 */

public class AbstractTool implements Tool {

    protected DrawingView fView;

    /**
     * The position of the initial mouse down.
     */
    protected int fAnchorX, fAnchorY;

    /**
     * Constructs a tool for the given view.
     */
    public AbstractTool(DrawingView itsView) {
        fView = itsView;
    }

    /**
     * Activates the tool for the given view. This method is called
     * whenever the user switches to this tool. Use this method to
     * reinitialize a tool.
     */
    public void activate() {
        fView.clearSelection();
    }

    /**
     * Deactivates the tool. This method is called whenever the user
     * switches to another tool. Use this method to do some clean-up
     * when the tool is switched. Subclassers should always call
     * super.deactivate.
     */
}
```

```

    */
    public void deactivate() {
        fView.setCursor(Cursor.getDefaultCursor());
    }

    /**
     * Handles mouse down events in the drawing view.
     */
    public void mouseDown(MouseEvent e, int x, int y) {
        fAnchorX = x;
        fAnchorY = y;
    }

    /**
     * Handles mouse drag events in the drawing view.
     */
    public void mouseDrag(MouseEvent e, int x, int y) {
    }

    /**
     * Handles mouse up in the drawing view.
     */
    public void mouseUp(MouseEvent e, int x, int y) {
    }

    /**
     * Handles mouse moves (if the mouse button is up).
     */
    public void mouseMove(MouseEvent evt, int x, int y) {
    }

    /**
     * Handles key down events in the drawing view.
     */
    public void keyDown(KeyEvent evt, int key) {
    }

    /**
     * Gets the tool's drawing.
     */
    public Drawing drawing() {
        return fView.drawing();
    }

    /**
     * Gets the tool's editor.
     */
    public DrawingEditor editor() {
        return fView.editor();
    }

    /**
     * Gets the tool's view.
     */
    public DrawingView view() {
        return fView;
    }
}

```

Clase ActionTool

Una herramienta que realiza una acción cuando esta activa y se pulsa el ratón.

Se encuentra en el fichero ActionTool.java y su contenido es el siguiente:

```
/*
 * @(#)ActionTool.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.framework.*;

import java.awt.*;
import java.awt.event.MouseEvent;

/**
 * A tool that performs an action when it is active and
 * the mouse is clicked.
 */
public abstract class ActionTool extends AbstractTool {

    public ActionTool(DrawingView itsView) {
        super(itsView);
    }

    /**
     * Add the touched figure to the selection and invoke action
     * @see #action()
     */
    public void mouseDown(MouseEvent e, int x, int y) {
        Figure target = drawing().findFigure(x, y);
        if (target != null) {
            view().addToSelection(target);
            action(target);
        }
    }

    public void mouseUp(MouseEvent e, int x, int y) {
        editor().toolDone();
    }

    /**
     * Performs an action with the touched figure.
     */
    public abstract void action(Figure figure);
}
```

Clase AlignCommand

Alinea una selección de figuras relativamente unas con otras.

Se encuentra en el fichero AlignCommand.java y su contenido es el siguiente:

```
/*
 * @(#)AlignCommand.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.util.*;
import java.awt.*;

import CH.ifa.draw.util.Command;
import CH.ifa.draw.framework.*;

/**
 * Align a selection of figures relative to each other.
 */
public class AlignCommand extends Command {

    private DrawingView fView;
    private int fOp;

    /**
     * align left sides
     */
    public final static int LEFTS = 0;
    /**
     * align centers (horizontally)
     */
    public final static int CENTERS = 1;
    /**
     * align right sides
     */
    public final static int RIGHTS = 2;
    /**
     * align tops
     */
    public final static int TOPS = 3;
    /**
     * align middles (vertically)
     */
    public final static int MIDDLES = 4;
    /**
     * align bottoms
     */
    public final static int BOTTOMS = 5;

    /**
     * Constructs an alignment command.
     */
}
```

```

* @param name the command name
* @param view the target view
* @param op the alignment operation (LEFTS, CENTERS, RIGHTS, etc.)
*/
public AlignCommand(String name, DrawingView view, int op) {
    super(name);
    fView = view;
    fOp = op;
}

public boolean isExecutable() {
    return fView.selectionCount() > 1;
}

public void execute() {
    FigureEnumeration selection = fView.selectionElements();
    Figure anchorFigure = selection.nextFigure();
    Rectangle r = anchorFigure.displayBox();

    while (selection.hasMoreElements()) {
        Figure f = selection.nextFigure();
        Rectangle rr = f.displayBox();
        switch (fOp) {
            case LEFTS:
                f.moveBy(r.x-rr.x, 0);
                break;
            case CENTERS:
                f.moveBy((r.x+r.width/2) - (rr.x+rr.width/2), 0);
                break;
            case RIGHTS:
                f.moveBy((r.x+r.width) - (rr.x+rr.width), 0);
                break;
            case TOPS:
                f.moveBy(0, r.y-rr.y);
                break;
            case MIDDLES:
                f.moveBy(0, (r.y+r.height/2) - (rr.y+rr.height/2));
                break;
            case BOTTOMS:
                f.moveBy(0, (r.y+r.height) - (rr.y+rr.height));
                break;
        }
    }
    fView.checkDamage();
}
}

```


Clase BoxHandleKit

Un conjunto de métodos de utilidad para crear Handles para las localizaciones comunes sobre una representación visual de una figura.

Se encuentra en el fichero BoxHandleKit.java y su contenido es el siguiente:

```
/*
 * @(#)BoxHandleKit.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.framework.*;
import java.awt.*;
import java.util.Vector;

/**
 * A set of utility methods to create Handles for the common
 * locations on a figure's display box.
 * @see Handle
 */

// TBD: use anonymous inner classes (had some problems with JDK 1.1)

public class BoxHandleKit {

    /**
     * Fills the given Vector with handles at each corner of a
     * figure.
     */
    static public void addCornerHandles(Figure f, Vector handles) {
        handles.addElement(southEast(f));
        handles.addElement(southWest(f));
        handles.addElement(northEast(f));
        handles.addElement(northWest(f));
    }

    /**
     * Fills the given Vector with handles at each corner
     * and the north, south, east, and west of the figure.
     */
    static public void addHandles(Figure f, Vector handles) {
        addCornerHandles(f, handles);
        handles.addElement(south(f));
        handles.addElement(north(f));
        handles.addElement(east(f));
        handles.addElement(west(f));
    }

    static public Handle south(Figure owner) {
        return new SouthHandle(owner);
    }
}
```

```

    }

    static public Handle southEast(Figure owner) {
        return new SouthEastHandle(owner);
    }

    static public Handle southWest(Figure owner) {
        return new SouthWestHandle(owner);
    }

    static public Handle north(Figure owner) {
        return new NorthHandle(owner);
    }

    static public Handle northEast(Figure owner) {
        return new NorthEastHandle(owner);
    }

    static public Handle northWest(Figure owner) {
        return new NorthWestHandle(owner);
    }

    static public Handle east(Figure owner) {
        return new EastHandle(owner);
    }
    static public Handle west(Figure owner) {
        return new WestHandle(owner);
    }
}

class NorthEastHandle extends LocatorHandle {
    NorthEastHandle(Figure owner) {
        super(owner, RelativeLocator.northEast());
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        Rectangle r = owner().displayBox();
        owner().displayBox(
            new Point(r.x, Math.min(r.y + r.height, y)),
            new Point(Math.max(r.x, x), r.y + r.height)
        );
    }
}

class EastHandle extends LocatorHandle {
    EastHandle(Figure owner) {
        super(owner, RelativeLocator.east());
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        Rectangle r = owner().displayBox();
        owner().displayBox(
            new Point(r.x, r.y), new Point(Math.max(r.x, x), r.y +
r.height)
        );
    }
}

```

```

}

class NorthHandle extends LocatorHandle {
    NorthHandle(Figure owner) {
        super(owner, RelativeLocator.north());
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        Rectangle r = owner().displayBox();
        owner().displayBox(
            new Point(r.x, Math.min(r.y + r.height, y)),
            new Point(r.x + r.width, r.y + r.height)
        );
    }
}

class NorthWestHandle extends LocatorHandle {
    NorthWestHandle(Figure owner) {
        super(owner, RelativeLocator.northWest());
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        Rectangle r = owner().displayBox();
        owner().displayBox(
            new Point(Math.min(r.x + r.width, x), Math.min(r.y +
r.height, y)),
            new Point(r.x + r.width, r.y + r.height)
        );
    }
}

class SouthEastHandle extends LocatorHandle {
    SouthEastHandle(Figure owner) {
        super(owner, RelativeLocator.southEast());
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        Rectangle r = owner().displayBox();
        owner().displayBox(
            new Point(r.x, r.y),
            new Point(Math.max(r.x, x), Math.max(r.y, y))
        );
    }
}

class SouthHandle extends LocatorHandle {
    SouthHandle(Figure owner) {
        super(owner, RelativeLocator.south());
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        Rectangle r = owner().displayBox();
        owner().displayBox(
            new Point(r.x, r.y),

```

```

        new Point(r.x + r.width, Math.max(r.y, y))
    );
}
}

class SouthWestHandle extends LocatorHandle {
    SouthWestHandle(Figure owner) {
        super(owner, RelativeLocator.southWest());
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        Rectangle r = owner().displayBox();
        owner().displayBox(
            new Point(Math.min(r.x + r.width, x), r.y),
            new Point(r.x + r.width, Math.max(r.y, y))
        );
    }
}

class WestHandle extends LocatorHandle {
    WestHandle(Figure owner) {
        super(owner, RelativeLocator.west());
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        Rectangle r = owner().displayBox();
        owner().displayBox(
            new Point(Math.min(r.x + r.width, x), r.y),
            new Point(r.x + r.width, r.y + r.height)
        );
    }
}
}

```

Clase BringToFrontCommand

BringToFrontCommand trae las figuras seleccionadas al frente de otras figuras.

Se encuentra en el fichero BringToFrontCommand.java y su contenido es el siguiente:

```

/*
 * @(#)BringToFrontCommand.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.util.*;
import CH.ifa.draw.util.Command;
import CH.ifa.draw.framework.*;

/**
 * BringToFrontCommand brings the selected figures in the front of

```

```

* the other figures.
*
* @see SendToBackCommand
*/
public class BringToFrontCommand
    extends Command {

    private DrawingView fView;

    /**
     * Constructs a bring to front command.
     * @param name the command name
     * @param view the target view
     */
    public BringToFrontCommand(String name, DrawingView view) {
        super(name);
        fView = view;
    }

    public void execute() {
        FigureEnumeration k = new
        FigureEnumerator(fView.selectionZOrdered());
        while (k.hasMoreElements()) {
            fView.drawing().bringToFront(k.nextFigure());
        }
        fView.checkDamage();
    }

    public boolean isExecutable() {
        return fView.selectionCount() > 0;
    }
}

```

Clase BufferedUpdateStrategy

BufferedUpdateStrategy implementa una estrategia de actualización que primero dibuja una vista en un buffer y seguidamente copia el buffer al DrawingView.

Se encuentra en el fichero BufferedUpdateStrategy.java y su contenido es el siguiente:

```

/*
 * @(#)BufferedUpdateStrategy.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.awt.image.*;
import CH.ifa.draw.framework.*;

/**
 * The BufferedUpdateStrategy implements an update
 * strategy that first draws a view into a buffer

```

```

* followed by copying the buffer to the DrawingView.
* @see DrawingView
*/

public class BufferedUpdateStrategy
    implements Painter {

    /**
     * The offscreen image
     */
    transient private Image    fOffscreen;
    private int    fImagewidth = -1;
    private int    fImageheight = -1;

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = 6489532222954612824L;
    private int bufferedUpdateSerializedDataVersion = 1;

    /**
     * Draws the view contents.
     */
    public void draw(Graphics g, DrawingView view) {
        // create the buffer if necessary
        Dimension d = view.getSize();
        if ((fOffscreen == null) || (d.width != fImagewidth)
            || (d.height != fImageheight)) {
            fOffscreen = view.createImage(d.width, d.height);
            fImagewidth = d.width;
            fImageheight = d.height;
        }

        // let the view draw on offscreen buffer
        Graphics g2 = fOffscreen.getGraphics();
        view.drawAll(g2);

        g.drawImage(fOffscreen, 0, 0, view);
    }
}

```

Clase ChangeAttributeCommand

Comando que cambia un atributo de una figura.

Se encuentra en el fichero ChangeAttributeCommand.java y su contenido es el siguiente:

```

/*
 * @(#)ChangeAttributeCommand.java 5.1
 *
 */

package CH.ifa.draw.standard;

```

```
import java.awt.Color;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.*;

/**
 * Command to change a named figure attribute.
 */
public class ChangeAttributeCommand
    extends Command {

    private DrawingView fView;
    private String      fAttribute;
    private Object      fValue;

    /**
     * Constructs a change attribute command.
     * @param name the command name
     * @param attributeName the name of the attribute to be changed
     * @param value the new attribute value
     * @param view the target view
     */
    public ChangeAttributeCommand(String name, String attributeName,
                                   Object value, DrawingView view) {
        super(name);
        fAttribute = attributeName;
        fValue = value;
        fView = view;
    }

    public void execute() {
        FigureEnumeration k = fView.selectionElements();
        while (k.hasMoreElements()) {
            Figure f = k.nextFigure();
            f.setAttribute(fAttribute, fValue);
        }
        fView.checkDamage();
    }

    public boolean isExecutable() {
        return fView.selectionCount() > 0;
    }
}
```

Clase ChangeConnectionEndHandle

Un manejador que conecta el punto final de una conexión a otra figura.

Se encuentra en el fichero ChangeConnectionEndHandle.java y su contenido es el siguiente:

```
/*
 * @(#)ChangeConnectionEndHandle.java 5.1
 *
 */
```

```
package CH.ifa.draw.standard;

import java.awt.Point;

import CH.ifa.draw.framework.*;

/**
 * A handle to reconnect the end point of
 * a connection to another figure.
 */

public class ChangeConnectionEndHandle extends ChangeConnectionHandle
{

    /**
     * Constructs the connection handle.
     */
    public ChangeConnectionEndHandle(Figure owner) {
        super(owner);
    }

    /**
     * Gets the end figure of a connection.
     */
    protected Connector target() {
        return fConnection.end();
    }

    /**
     * Disconnects the end figure.
     */
    protected void disconnect() {
        fConnection.disconnectEnd();
    }

    /**
     * Sets the end of the connection.
     */
    protected void connect(Connector c) {
        fConnection.connectEnd(c);
    }

    /**
     * Sets the end point of the connection.
     */
    protected void setPoint(int x, int y) {
        fConnection.endPoint(x, y);
    }

    /**
     * Returns the end point of the connection.
     */
    public Point locate() {
        return fConnection.endPoint();
    }
}
```


Clase ChangeConnectionHandle

ChangeConnectionHandle factoriza el código común para los manejadores que puede ser utilizado para reconectar conexiones.

Se encuentra en el fichero ChangeConnectionHandle.java y su contenido es el siguiente:

```
/*
 * @(#)ChangeConnectionHandle.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;

import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.Geom;

/**
 * ChangeConnectionHandle factors the common code for handles
 * that can be used to reconnect connections.
 *
 * @see ChangeConnectionEndHandle
 * @see ChangeConnectionStartHandle
 */
public abstract class ChangeConnectionHandle extends AbstractHandle {

    protected Connector      fOriginalTarget;
    protected Figure         fTarget;
    protected ConnectionFigure fConnection;
    protected Point          fStart;

    /**
     * Initializes the change connection handle.
     */
    protected ChangeConnectionHandle(Figure owner) {
        super(owner);
        fConnection = (ConnectionFigure) owner();
        fTarget = null;
    }

    /**
     * Returns the target connector of the change.
     */
    protected abstract Connector target();

    /**
     * Disconnects the connection.
     */
    protected abstract void disconnect();

}
```

```

    * Connect the connection with the given figure.
    */
protected abstract void connect(Connector c);

/**
 * Sets the location of the target point.
 */
protected abstract void setPoint(int x, int y);

/**
 * Gets the side of the connection that is unaffected by
 * the change.
 */
protected Connector source() {
    if (target() == fConnection.start())
        return fConnection.end();
    return fConnection.start();
}

/**
 * Disconnects the connection.
 */
public void invokeStart(int x, int y, DrawingView view) {
    fOriginalTarget = target();
    fStart = new Point(x, y);
    disconnect();
}

/**
 * Finds a new target of the connection.
 */
public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
    Point p = new Point(x, y);
    Figure f = findConnectableFigure(x, y, view.drawing());
    // track the figure containing the mouse
    if (f != fTarget) {
        if (fTarget != null)
            fTarget.connectorVisibility(false);
        fTarget = f;
        if (fTarget != null)
            fTarget.connectorVisibility(true);
    }

    Connector target = findConnectionTarget(p.x, p.y,
view.drawing());
    if (target != null)
        p = Geom.center(target.displayBox());
    setPoint(p.x, p.y);
}

/**
 * Connects the figure to the new target. If there is no
 * new target the connection reverts to its original one.
 */
public void invokeEnd(int x, int y, int anchorX, int anchorY,
DrawingView view) {
    Connector target = findConnectionTarget(x, y, view.drawing());

```

```

        if (target == null)
            target = fOriginalTarget;

        setPoint(x, y);
        connect(target);
        fConnection.updateConnection();
        if (fTarget != null) {
            fTarget.connectorVisibility(false);
            fTarget = null;
        }
    }

    private Connector findConnectionTarget(int x, int y, Drawing
drawing) {
        Figure target = findConnectableFigure(x, y, drawing);

        if ((target != null) && target.canConnect()
            && target != fOriginalTarget
            && !target.includes(owner())
            && fConnection.canConnect(source().owner(), target)) {
            return findConnector(x, y, target);
        }
        return null;
    }

    protected Connector findConnector(int x, int y, Figure f) {
        return f.connectorAt(x, y);
    }

    /**
     * Draws this handle.
     */
    public void draw(Graphics g) {
        Rectangle r = displayBox();

        g.setColor(Color.green);
        g.fillRect(r.x, r.y, r.width, r.height);

        g.setColor(Color.black);
        g.drawRect(r.x, r.y, r.width, r.height);
    }

    private Figure findConnectableFigure(int x, int y, Drawing
drawing) {
        FigureEnumeration k = drawing.figuresReverse();
        while (k.hasMoreElements()) {
            Figure figure = k.nextFigure();
            if (!figure.includes(fConnection) && figure.canConnect())
            {
                if (figure.containsPoint(x, y))
                    return figure;
            }
        }
        return null;
    }
}

```

Clase ChangeConnectionStartHandle

Manejador que reconecta el principio de una conexión a otra figura.

Se encuentra en el fichero ChangeConnectionStartHandle.java y su contenido es el siguiente:

```
/*
 * @(#)ChangeConnectionStartHandle.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.Point;

import CH.ifa.draw.framework.*;

/**
 * Handle to reconnect the
 * start of a connection to another figure.
 */

public class ChangeConnectionStartHandle extends
ChangeConnectionHandle {

    /**
     * Constructs the connection handle for the given start figure.
     */
    public ChangeConnectionStartHandle(Figure owner) {
        super(owner);
    }

    /**
     * Gets the start figure of a connection.
     */
    protected Connector target() {
        return fConnection.start();
    }

    /**
     * Disconnects the start figure.
     */
    protected void disconnect() {
        fConnection.disconnectStart();
    }

    /**
     * Sets the start of the connection.
     */
    protected void connect(Connector c) {
        fConnection.connectStart(c);
    }

    /**
     * Sets the start point of the connection.
     */
}
```

```

    */
    protected void setPoint(int x, int y) {
        fConnection.startPoint(x, y);
    }

    /**
     * Returns the start point of the connection.
     */
    public Point locate() {
        return fConnection.startPoint();
    }
}

```

Clase ChopBoxConnector

Un ChopBoxConnector localiza puntos de conexión para realizar la conexión entre los centros de las dos figuras de la zona de representación.

Se encuentra en el fichero ChopBoxConnector.java y su contenido es el siguiente:

```

/*
 * @(#)ChopBoxConnector.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.Geom;

/**
 * A ChopBoxConnector locates connection points by
 * chopping the connection between the centers of the
 * two figures at the display box.
 * @see Connector
 */
public class ChopBoxConnector extends AbstractConnector {

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = -
1461450322712345462L;

    public ChopBoxConnector() { // only used for Storable
implementation
    }

    public ChopBoxConnector(Figure owner) {
        super(owner);
    }

    public Point findStart(ConnectionFigure connection) {

```

```

Figure startFigure = connection.start().owner();
Rectangle r2 = connection.end().displayBox();
Point r2c = null;

if (connection.pointCount() == 2)
    r2c = new Point(r2.x + r2.width/2, r2.y + r2.height/2);
else
    r2c = connection.pointAt(1);

return chop(startFigure, r2c);
}

public Point findEnd(ConnectionFigure connection) {
    Figure endFigure = connection.end().owner();
    Rectangle r1 = connection.start().displayBox();
    Point r1c = null;

    if (connection.pointCount() == 2)
        r1c = new Point(r1.x + r1.width/2, r1.y + r1.height/2);
    else
        r1c = connection.pointAt(connection.pointCount()-2);

    return chop(endFigure, r1c);
}

protected Point chop(Figure target, Point from) {
    Rectangle r = target.displayBox();
    return Geom.angleToPoint(r, (Geom.pointToAngle(r, from)));
}
}

```

Clase CompositeFigure

Una Figure que esta compuesta de varias figuras. Una CompositeFigure no define ningún comportamiento layout. Está esta dispuesto por las subclases que organizan los contenedores de figuras.

Patrones de diseño

Composite CompositeFigure permite tratar una composición de figuras como una simple figura.

Se encuentra en el fichero CompositeFigure.java y su contenido es el siguiente:

```

/*
 * @(#)CompositeFigure.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.util.*;

```

```
import CH.ifa.draw.framework.*;
import java.awt.*;
import java.util.*;
import java.io.*;

/**
 * A Figure that is composed of several figures. A CompositeFigure
 * doesn't define any layout behavior. It is up to subclassers to
 * arrange the contained figures.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld012.htm>Composite</a></b><br>
 * CompositeFigure enables to treat a composition of figures like
 * a single figure.<br>
 * @see Figure
 */

public abstract class CompositeFigure
    extends AbstractFigure
    implements FigureChangeListener {

    /**
     * The figures that this figure is composed of
     * @see #add
     * @see #remove
     */
    protected Vector fFigures;

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = 7408153435700021866L;
    private int compositeFigureSerializedDataVersion = 1;

    protected CompositeFigure() {
        fFigures = new Vector();
    }

    /**
     * Adds a figure to the list of figures. Initializes the
     * the figure's container.
     */
    public Figure add(Figure figure) {
        if (!fFigures.contains(figure)) {
            fFigures.addElement(figure);
            figure.addToContainer(this);
        }
        return figure;
    }

    /**
     * Adds a vector of figures.
     * @see #add
     */
    public void addAll(Vector newFigures) {
        Enumeration k = newFigures.elements();
        while (k.hasMoreElements())
```

```

        add((Figure) k.nextElement());
    }

    /**
     * Removes a figure from the composite.
     * @see #removeAll
     */
    public Figure remove(Figure figure) {
        if (fFigures.contains(figure)) {
            figure.removeFromContainer(this);
            fFigures.removeElement(figure);
        }
        return figure;
    }

    /**
     * Removes a vector of figures.
     * @see #remove
     */
    public void removeAll(Vector figures) {
        Enumeration k = figures.elements();
        while (k.hasMoreElements())
            remove((Figure) k.nextElement());
    }

    /**
     * Removes all children.
     * @see #remove
     */
    public void removeAll() {
        FigureEnumeration k = figures();
        while (k.hasMoreElements()) {
            Figure figure = k.nextFigure();
            figure.removeFromContainer(this);
        }
        fFigures.removeAllElements();
    }

    /**
     * Removes a figure from the figure list, but
     * doesn't release it. Use this method to temporarily
     * manipulate a figure outside of the drawing.
     */
    public synchronized Figure orphan(Figure figure) {
        fFigures.removeElement(figure);
        return figure;
    }

    /**
     * Removes a vector of figures from the figure's list
     * without releasing the figures.
     * @see orphan
     */
    public void orphanAll(Vector newFigures) {
        Enumeration k = newFigures.elements();
        while (k.hasMoreElements())
            orphan((Figure) k.nextElement());
    }

```



```

/**
 * Replaces a figure in the drawing without
 * removing it from the drawing.
 */
public synchronized void replace(Figure figure, Figure
replacement) {
    int index = fFigures.indexOf(figure);
    if (index != -1) {
        replacement.addToContainer(this);    // will invalidate
figure
        figure.changed();
        fFigures.setElementAt(replacement, index);
    }
}

/**
 * Sends a figure to the back of the drawing.
 */
public synchronized void sendToBack(Figure figure) {
    if (fFigures.contains(figure)) {
        fFigures.removeElement(figure);
        fFigures.insertElementAt(figure,0);
        figure.changed();
    }
}

/**
 * Brings a figure to the front.
 */
public synchronized void bringToFront(Figure figure) {
    if (fFigures.contains(figure)) {
        fFigures.removeElement(figure);
        fFigures.addElement(figure);
        figure.changed();
    }
}

/**
 * Draws all the contained figures
 * @see Figure#draw
 */
public void draw(Graphics g) {
    FigureEnumeration k = figures();
    while (k.hasMoreElements())
        k.nextFigure().draw(g);
}

/**
 * Gets a figure at the given index.
 */
public Figure figureAt(int i) {
    return (Figure)fFigures.elementAt(i);
}

/**
 * Returns an Enumeration for accessing the contained figures.
 * The figures are returned in the drawing order.

```

```

    */
    public final FigureEnumeration figures() {
        return new FigureEnumerator(fFigures);
    }

    /**
     * Gets number of child figures.
     */
    public int figureCount() {
        return fFigures.size();
    }

    /**
     * Returns an Enumeration for accessing the contained figures
     * in the reverse drawing order.
     */
    public final FigureEnumeration figuresReverse() {
        return new ReverseFigureEnumerator(fFigures);
    }

    /**
     * Finds a top level Figure. Use this call for hit detection that
     * should not descend into the figure's children.
     */
    public Figure findFigure(int x, int y) {
        FigureEnumeration k = figuresReverse();
        while (k.hasMoreElements()) {
            Figure figure = k.nextFigure();
            if (figure.containsPoint(x, y))
                return figure;
        }
        return null;
    }

    /**
     * Finds a top level Figure that intersects the given rectangle.
     */
    public Figure findFigure(Rectangle r) {
        FigureEnumeration k = figuresReverse();
        while (k.hasMoreElements()) {
            Figure figure = k.nextFigure();
            Rectangle fr = figure.displayBox();
            if (r.intersects(fr))
                return figure;
        }
        return null;
    }

    /**
     * Finds a top level Figure, but supresses the passed
     * in figure. Use this method to ignore a figure
     * that is temporarily inserted into the drawing.
     * @param x the x coordinate
     * @param y the y coordinate
     * @param without the figure to be ignored during
     * the find.
     */
    public Figure findFigureWithout(int x, int y, Figure without) {

```

```

        if (without == null)
            return findFigure(x, y);
        FigureEnumeration k = figuresReverse();
        while (k.hasMoreElements()) {
            Figure figure = k.nextFigure();
            if (figure.containsPoint(x, y) &&
!figure.includes(without))
                return figure;
        }
        return null;
    }

    /**
     * Finds a top level Figure that intersects the given rectangle.
     * It supresses the passed
     * in figure. Use this method to ignore a figure
     * that is temporarily inserted into the drawing.
     */
    public Figure findFigure(Rectangle r, Figure without) {
        if (without == null)
            return findFigure(r);
        FigureEnumeration k = figuresReverse();
        while (k.hasMoreElements()) {
            Figure figure = k.nextFigure();
            Rectangle fr = figure.displayBox();
            if (r.intersects(fr) && !figure.includes(without))
                return figure;
        }
        return null;
    }

    /**
     * Finds a figure but descends into a figure's
     * children. Use this method to implement <i>click-through</i>
     * hit detection, that is, you want to detect the inner most
     * figure containing the given point.
     */
    public Figure findFigureInside(int x, int y) {
        FigureEnumeration k = figuresReverse();
        while (k.hasMoreElements()) {
            Figure figure = k.nextFigure().findFigureInside(x, y);
            if (figure != null)
                return figure;
        }
        return null;
    }

    /**
     * Finds a figure but descends into a figure's
     * children. It supresses the passed
     * in figure. Use this method to ignore a figure
     * that is temporarily inserted into the drawing.
     */
    public Figure findFigureInsideWithout(int x, int y, Figure
without) {
        FigureEnumeration k = figuresReverse();
        while (k.hasMoreElements()) {
            Figure figure = k.nextFigure();

```

```

        if (figure != without) {
            Figure found = figure.findFigureInside(x, y);
            if (found != null)
                return found;
        }
    }
    return null;
}

/**
 * Checks if the composite figure has the argument as one of
 * its children.
 */
public boolean includes(Figure figure) {
    if (super.includes(figure))
        return true;

    FigureEnumeration k = figures();
    while (k.hasMoreElements()) {
        Figure f = k.nextFigure();
        if (f.includes(figure))
            return true;
    }
    return false;
}

/**
 * Moves all the given figures by x and y. Doesn't announce
 * any changes. Subclassers override
 * basicMoveBy. Clients usually call moveBy.
 * @see moveBy
 */
protected void basicMoveBy(int x, int y) {
    FigureEnumeration k = figures();
    while (k.hasMoreElements())
        k.nextFigure().moveBy(x,y);
}

/**
 * Releases the figure and all its children.
 */
public void release() {
    super.release();
    FigureEnumeration k = figures();
    while (k.hasMoreElements()) {
        Figure figure = k.nextFigure();
        figure.release();
    }
}

/**
 * Propagates the figureInvalidated event to my listener.
 * @see FigureChangeListener
 */
public void figureInvalidated(FigureChangeEvent e) {
    if (listener() != null)
        listener().figureInvalidated(e);
}

```

```

/**
 * Propagates the removeFromDrawing request up to the container.
 * @see FigureChangeListener
 */
public void figureRequestRemove(FigureChangeEvent e) {
    if (listener() != null)
        listener().figureRequestRemove(new
FigureChangeEvent(this));
}

/**
 * Propagates the requestUpdate request up to the container.
 * @see FigureChangeListener
 */
public void figureRequestUpdate(FigureChangeEvent e) {
    if (listener() != null)
        listener().figureRequestUpdate(e);
}

public void figureChanged(FigureChangeEvent e) {
}

public void figureRemoved(FigureChangeEvent e) {
}

/**
 * Writes the contained figures to the StorableOutput.
 */
public void write(StorableOutput dw) {
    super.write(dw);
    dw.writeInt(fFigures.size());
    Enumeration k = fFigures.elements();
    while (k.hasMoreElements())
        dw.writeStorable((Storable) k.nextElement());
}

/**
 * Reads the contained figures from StorableInput.
 */
public void read(StorableInput dr) throws IOException {
    super.read(dr);
    int size = dr.readInt();
    fFigures = new Vector(size);
    for (int i=0; i<size; i++)
        add((Figure)dr.readStorable());
}

private void readObject(ObjectInputStream s)
    throws ClassNotFoundException, IOException {
    s.defaultReadObject();

    FigureEnumeration k = figures();
    while (k.hasMoreElements()) {
        Figure figure = k.nextFigure();
        figure.addToContainer(this);
    }
}

```

```
}  
}
```

Clase ConnectionHandle

Un manejador que conecta figuras. El objeto connection que se crea es especificado por un Prototype.

Patrones de diseño

Prototype ConnectionHandle crea la conexión clonando un prototipo.

Se encuentra en el fichero ConnectionHandle.java y su contenido es el siguiente:

```
/*  
 * @(#)ConnectionHandle.java 5.1  
 *  
 */  
  
package CH.ifa.draw.standard;  
  
import java.awt.*;  
  
import CH.ifa.draw.framework.*;  
import CH.ifa.draw.util.Geom;  
  
/**  
 * A handle to connect figures.  
 * The connection object to be created is specified by a prototype.  
 * <hr>  
 * <b>Design Patterns</b><P>  
 *   
 * <b><a href=../pattlets/sld029.htm>Prototype</a></b><br>  
 * ConnectionHandle creates the connection by cloning a prototype.  
 * <hr>  
 *  
 * @see ConnectionFigure  
 * @see Object#clone  
 */  
  
public class ConnectionHandle extends LocatorHandle {  
  
    /**  
     * the currently created connection  
     */  
    private ConnectionFigure fConnection;  
  
    /**  
     * the prototype of the connection to be created  
     */  
    private ConnectionFigure fPrototype;  
  
    /**
```

```

    * the current target
    */
    private Figure fTarget = null;

    /**
     * Constructs a handle with the given owner, locator, and
     * connection prototype
     */
    public ConnectionHandle(Figure owner, Locator l, ConnectionFigure
prototype) {
        super(owner, l);
        fPrototype = prototype;
    }

    /**
     * Creates the connection
     */
    public void invokeStart(int x, int y, DrawingView view) {
        fConnection = createConnection();
        Point p = locate();
        fConnection.startPoint(p.x, p.y);
        fConnection.endPoint(p.x, p.y);
        view.drawing().add(fConnection);
    }

    /**
     * Tracks the connection.
     */
    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        Point p = new Point(x,y);
        Figure f = findConnectableFigure(x, y, view.drawing());
        // track the figure containing the mouse
        if (f != fTarget) {
            if (fTarget != null)
                fTarget.connectorVisibility(false);
            fTarget = f;
            if (fTarget != null)
                fTarget.connectorVisibility(true);
        }

        Connector target = findConnectionTarget(p.x, p.y,
view.drawing());
        if (target != null)
            p = Geom.center(target.displayBox());
        fConnection.endPoint(p.x, p.y);
    }

    /**
     * Connects the figures if the mouse is released over another
     * figure.
     */
    public void invokeEnd(int x, int y, int anchorX, int anchorY,
DrawingView view) {
        Connector target = findConnectionTarget(x, y, view.drawing());
        if (target != null) {
            fConnection.connectStart(startConnector());
            fConnection.connectEnd(target);
        }
    }

```

```

        fConnection.updateConnection();
    } else
        view.drawing().remove(fConnection);
    fConnection = null;
    if (fTarget != null) {
        fTarget.connectorVisibility(false);
        fTarget = null;
    }
}

private Connector startConnector() {
    Point p = locate();
    return owner().connectorAt(p.x, p.y);
}

/**
 * Creates the ConnectionFigure. By default the figure prototype
is
 * cloned.
 */
protected ConnectionFigure createConnection() {
    return (ConnectionFigure)fPrototype.clone();
}

/**
 * Finds a connection end figure.
 */
protected Connector findConnectionTarget(int x, int y, Drawing
drawing) {
    Figure target = findConnectableFigure(x, y, drawing);
    if ((target != null) && target.canConnect()
        && !target.includes(owner())
        && fConnection.canConnect(owner(), target)) {
        return findConnector(x, y, target);
    }
    return null;
}

private Figure findConnectableFigure(int x, int y, Drawing
drawing) {
    FigureEnumeration k = drawing.figuresReverse();
    while (k.hasMoreElements()) {
        Figure figure = k.nextFigure();
        if (!figure.includes(fConnection) && figure.canConnect())
        {
            if (figure.containsPoint(x, y))
                return figure;
        }
    }
    return null;
}

protected Connector findConnector(int x, int y, Figure f) {
    return f.connectorAt(x, y);
}

/**

```



```

    * Draws the connection handle, by default the outline of a
    * blue circle.
    */
    public void draw(Graphics g) {
        Rectangle r = displayBox();
        g.setColor(Color.blue);
        g.drawOval(r.x, r.y, r.width, r.height);
    }
}

```

Clase ConnectionTool

Una herramienta que puede ser utilizada para conectar figuras, para dividir conexiones, y para unir dos segmentos de una conexión. Las ConnectionTools cambian la visibilidad de los Connectors cuando entra una figura. El objeto connection que se crea es especificado por un Prototype.

Patrones de diseño

Prototype ConnectionTools crea la conexión clonando un prototipo.

Se encuentra en el fichero ConnectionTool.java y su contenido es el siguiente:

```

/*
 * @(#)ConnectionTool.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.awt.event.MouseEvent;
import java.util.*;

import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.Geom;

/**
 * A tool that can be used to connect figures, to split
 * connections, and to join two segments of a connection.
 * ConnectionTools turns the visibility of the Connectors
 * on when it enters a figure.
 * The connection object to be created is specified by a prototype.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld029.htm>Prototype</a></b><br>
 * ConnectionTools creates the connection by cloning a prototype.
 * <hr>
 *
 * @see ConnectionFigure

```

```
* @see Object#clone
*/

public class ConnectionTool extends AbstractTool {

    /**
     * the anchor point of the interaction
     */
    private Connector fStartConnector;
    private Connector fEndConnector;
    private Connector fConnectorTarget = null;

    private Figure fTarget = null;

    /**
     * the currently created figure
     */
    private ConnectionFigure fConnection;

    /**
     * the currently manipulated connection point
     */
    private int fSplitPoint;

    /**
     * the currently edited connection
     */
    private ConnectionFigure fEditedConnection = null;

    /**
     * the prototypical figure that is used to create new
     * connections.
     */
    private ConnectionFigure fPrototype;

    public ConnectionTool(DrawingView view, ConnectionFigure
prototype) {
        super(view);
        fPrototype = prototype;
    }

    /**
     * Handles mouse move events in the drawing view.
     */
    public void mouseMove(MouseEvent e, int x, int y) {
        trackConnectors(e, x, y);
    }

    /**
     * Manipulates connections in a context dependent way. If the
     * mouse down hits a figure start a new connection. If the
     * mousedown hits a connection split a segment or join two
     * segments.
     */
    public void mouseDown(MouseEvent e, int x, int y)
    {
        int ex = e.getX();
```

```

        int ey = e.getY();
        fTarget = findConnectionStart(ex, ey, drawing());
        if (fTarget != null) {
            fStartConnector = findConnector(ex, ey, fTarget);
            if (fStartConnector != null) {
                Point p = new Point(ex, ey);
                fConnection = createConnection();
                fConnection.startPoint(p.x, p.y);
                fConnection.endPoint(p.x, p.y);
                view().add(fConnection);
            }
        }
        else {
            ConnectionFigure connection = findConnection(ex, ey,
drawing());
            if (connection != null) {
                if (!connection.joinSegments(ex, ey)) {
                    fSplitPoint = connection.splitSegment(ex, ey);
                    fEditedConnection = connection;
                } else {
                    fEditedConnection = null;
                }
            }
        }
    }

    /**
     * Adjust the created connection or split segment.
     */
    public void mouseDrag(MouseEvent e, int x, int y) {
        Point p = new Point(e.getX(), e.getY());
        if (fConnection != null) {
            trackConnectors(e, x, y);
            if (fConnectorTarget != null)
                p = Geom.center(fConnectorTarget.displayBox());
            fConnection.endPoint(p.x, p.y);
        }
        else if (fEditedConnection != null) {
            Point pp = new Point(x, y);
            fEditedConnection.setPointAt(pp, fSplitPoint);
        }
    }

    /**
     * Connects the figures if the mouse is released over another
     * figure.
     */
    public void mouseUp(MouseEvent e, int x, int y) {
        Figure c = null;
        if (fStartConnector != null)
            c = findTarget(e.getX(), e.getY(), drawing());

        if (c != null) {
            fEndConnector = findConnector(e.getX(), e.getY(), c);
            if (fEndConnector != null) {
                fConnection.connectStart(fStartConnector);
                fConnection.connectEnd(fEndConnector);
                fConnection.updateConnection();
            }
        }
    }

```

```

    }
    } else if (fConnection != null)
        view().remove(fConnection);

    fConnection = null;
    fStartConnector = fEndConnector = null;
    editor().toolDone();
}

public void deactivate() {
    super.deactivate();
    if (fTarget != null)
        fTarget.connectorVisibility(false);
}

/**
 * Creates the ConnectionFigure. By default the figure prototype
is
 * cloned.
 */
protected ConnectionFigure createConnection() {
    return (ConnectionFigure)fPrototype.clone();
}

/**
 * Finds a connectable figure target.
 */
protected Figure findSource(int x, int y, Drawing drawing) {
    return findConnectableFigure(x, y, drawing);
}

/**
 * Finds a connectable figure target.
 */
protected Figure findTarget(int x, int y, Drawing drawing) {
    Figure target = findConnectableFigure(x, y, drawing);
    Figure start = fStartConnector.owner();

    if (target != null
        && fConnection != null
        && target.canConnect()
        && !target.includes(start)
        && fConnection.canConnect(start, target))
        return target;
    return null;
}

/**
 * Finds an existing connection figure.
 */
protected ConnectionFigure findConnection(int x, int y, Drawing
drawing) {
    Enumeration k = drawing.figuresReverse();
    while (k.hasMoreElements()) {
        Figure figure = (Figure) k.nextElement();
        figure = figure.findFigureInside(x, y);
        if (figure != null && (figure instanceof
ConnectionFigure))

```

```

        return (ConnectionFigure)figure;
    }
    return null;
}

/**
 * Gets the currently created figure
 */
protected ConnectionFigure createdFigure() {
    return fConnection;
}

protected void trackConnectors(MouseEvent e, int x, int y) {
    Figure c = null;

    if (fStartConnector == null)
        c = findSource(x, y, drawing());
    else
        c = findTarget(x, y, drawing());

    // track the figure containing the mouse
    if (c != fTarget) {
        if (fTarget != null)
            fTarget.connectorVisibility(false);
        fTarget = c;
        if (fTarget != null)
            fTarget.connectorVisibility(true);
    }

    Connector cc = null;
    if (c != null)
        cc = findConnector(e.getX(), e.getY(), c);
    if (cc != fConnectorTarget)
        fConnectorTarget = cc;

    view().checkDamage();
}

private Connector findConnector(int x, int y, Figure f) {
    return f.connectorAt(x, y);
}

/**
 * Finds a connection start figure.
 */
protected Figure findConnectionStart(int x, int y, Drawing
drawing) {
    Figure target = findConnectableFigure(x, y, drawing);
    if ((target != null) && target.canConnect())
        return target;
    return null;
}

private Figure findConnectableFigure(int x, int y, Drawing
drawing) {
    FigureEnumeration k = drawing.figuresReverse();
    while (k.hasMoreElements()) {
        Figure figure = k.nextFigure();
    }
}

```

```

        if (!figure.includes(fConnection) && figure.canConnect())
        {
            if (figure.containsPoint(x, y))
                return figure;
        }
        return null;
    }

    protected Connector getStartConnector() {
        return fStartConnector;
    }

    protected Connector getEndConnector() {
        return fEndConnector;
    }

    protected Connector getTarget() {
        return fConnectorTarget;
    }
}

```

Clase CopyCommand

Copia la selección al portapapeles.

Se encuentra en el fichero CopyCommand.java y su contenido es el siguiente:

```

/*
 * @(#)CopyCommand.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.util.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

/**
 * Copy the selection to the clipboard.
 * @see Clipboard
 */
public class CopyCommand extends FigureTransferCommand {

    /**
     * Constructs a copy command.
     * @param name the command name
     * @param view the target view
     */
    public CopyCommand(String name, DrawingView view) {
        super(name, view);
    }
}

```

```

    public void execute() {
        copySelection();
    }

    public boolean isExecutable() {
        return fView.selectionCount() > 0;
    }
}

```

Clase CreationTool

Una herramienta que crea nuevas figuras. La figura que se crea es especificada por un Prototype.

Patrones de diseño

Prototype CreationTool crea nuevas figuras clonando un prototipo.

Se encuentra en el fichero CreationTool.java y su contenido es el siguiente:

```

/*
 * @(#)CreationTool.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.awt.event.MouseEvent;

import CH.ifa.draw.framework.*;

/**
 * A tool to create new figures. The figure to be
 * created is specified by a prototype.
 *
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld029.htm>Prototype</a></b><br>
 * CreationTool creates new figures by cloning a prototype.
 * <hr>
 * @see Figure
 * @see Object#clone
 */

public class CreationTool extends AbstractTool {

    /**
     * the anchor point of the interaction

```

```

    */
    private Point    fAnchorPoint;

    /**
     * the currently created figure
     */
    private Figure    fCreatedFigure;

    /**
     * the prototypical figure that is used to create new figures.
     */
    private Figure    fPrototype;

    /**
     * Initializes a CreationTool with the given prototype.
     */
    public CreationTool(DrawingView view, Figure prototype) {
        super(view);
        fPrototype = prototype;
    }

    /**
     * Constructs a CreationTool without a prototype.
     * This is for subclasses overriding createFigure.
     */
    protected CreationTool(DrawingView view) {
        super(view);
        fPrototype = null;
    }

    /**
     * Sets the cross hair cursor.
     */
    public void activate() {
        view().setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
    }

    /**
     * Creates a new figure by cloning the prototype.
     */
    public void mouseDown(MouseEvent e, int x, int y) {
        fAnchorPoint = new Point(x,y);
        fCreatedFigure = createFigure();
        fCreatedFigure.displayBox(fAnchorPoint, fAnchorPoint);
        view().add(fCreatedFigure);
    }

    /**
     * Creates a new figure by cloning the prototype.
     */
    protected Figure createFigure() {
        if (fPrototype == null)
            throw new HJDError("No prototype defined");
        return (Figure) fPrototype.clone();
    }

```



```

/**
 * Adjusts the extent of the created figure
 */
public void mouseDrag(MouseEvent e, int x, int y) {
    fCreatedFigure.displayBox(fAnchorPoint, new Point(x,y));
}

/**
 * Checks if the created figure is empty. If it is, the figure
 * is removed from the drawing.
 * @see Figure#isEmpty
 */
public void mouseUp(MouseEvent e, int x, int y) {
    if (fCreatedFigure.isEmpty())
        drawing().remove(fCreatedFigure);
    fCreatedFigure = null;
    editor().toolDone();
}

/**
 * Gets the currently created figure
 */
protected Figure createdFigure() {
    return fCreatedFigure;
}
}

```

Clase CutCommand

Borra la selección y mueve las figuras seleccionadas al portapapeles.

Se encuentra en el fichero CutCommand.java y su contenido es el siguiente:

```

/*
 * @(#)CutCommand.java 5.1
 */

package CH.ifa.draw.standard;

import java.util.*;
import java.awt.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

/**
 * Delete the selection and move the selected figures to
 * the clipboard.
 * @see Clipboard
 */
public class CutCommand extends FigureTransferCommand {

    /**
     * Constructs a cut command.
     */
}

```

```
    * @param name the command name
    * @param view the target view
    */
    public CutCommand(String name, DrawingView view) {
        super(name, view);
    }

    public void execute() {
        copySelection();
        deleteSelection();
        fView.checkDamage();
    }

    public boolean isExecutable() {
        return fView.selectionCount() > 0;
    }
}
```

Clase DecoratorFigure

DecoratorFigure puede ser utilizado para decorar otras figuras con decoraciones como bordes. Decorator envia todos sus métodos a las figuras que contiene. Las subclases pueden selectivamente sobrescribir estos métodos, extender y filtrar sus comportamientos.

Patrones de diseño

Decorator DecoratorFigure es un *DecoratorAbstracto*.

Se encuentra en el fichero DecoratorFigure.java y su contenido es el siguiente:

```
/*
 * @(#)DecoratorFigure.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

import java.awt.*;
import java.util.*;
import java.io.*;

/**
 * DecoratorFigure can be used to decorate other figures with
 * decorations like borders. Decorator forwards all the
 * methods to their contained figure. Subclasses can selectively
 * override these methods to extend and filter their behavior.
 * <hr>
 * <b>Design Patterns</b><P>
```

```

* 
* <b><a href=../pattlets/sld014.htm>Decorator</a></b><br>
* DecoratorFigure is a decorator.
*
* @see Figure
*/

public abstract class DecoratorFigure
    extends AbstractFigure
    implements FigureChangeListener {

    /**
     * The decorated figure.
     */
    protected Figure fComponent;

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = 8993011151564573288L;
    private int decoratorFigureSerializedDataVersion = 1;

    public DecoratorFigure() { }

    /**
     * Constructs a DecoratorFigure and decorates the passed in
     * figure.
     */
    public DecoratorFigure(Figure figure) {
        decorate(figure);
    }

    /**
     * Forwards the connection insets to its contained figure..
     */
    public Insets connectionInsets() {
        return fComponent.connectionInsets();
    }

    /**
     * Forwards the canConnect to its contained figure..
     */
    public boolean canConnect() {
        return fComponent.canConnect();
    }

    /**
     * Forwards containsPoint to its contained figure.
     */
    public boolean containsPoint(int x, int y) {
        return fComponent.containsPoint(x, y);
    }

    /**
     * Decorates the given figure.
     */
    public void decorate(Figure figure) {
        fComponent = figure;
    }

```

```

        fComponent.addToContainer(this);
    }

    /**
     * Removes the decoration from the contained figure.
     */
    public Figure peelDecoration() {
        fComponent.removeFromContainer(this); //??? set the container
                                           // to the listener()?
        return fComponent;
    }

    /**
     * Forwards displayBox to its contained figure.
     */
    public Rectangle displayBox() {
        return fComponent.displayBox();
    }

    /**
     * Forwards basicDisplayBox to its contained figure.
     */
    public void basicDisplayBox(Point origin, Point corner) {
        fComponent.basicDisplayBox(origin, corner);
    }

    /**
     * Forwards draw to its contained figure.
     */
    public void draw(Graphics g) {
        fComponent.draw(g);
    }

    /**
     * Forwards findFigureInside to its contained figure.
     */
    public Figure findFigureInside(int x, int y) {
        return fComponent.findFigureInside(x, y);
    }

    /**
     * Forwards handles to its contained figure.
     */
    public Vector handles() {
        return fComponent.handles();
    }

    /**
     * Forwards includes to its contained figure.
     */
    public boolean includes(Figure figure) {
        return (super.includes(figure) ||
fComponent.includes(figure));
    }

    /**
     * Forwards moveBy to its contained figure.
     */

```

```

public void moveBy(int x, int y) {
    fComponent.moveBy(x, y);
}

/**
 * Forwards basicMoveBy to its contained figure.
 */
protected void basicMoveBy(int x, int y) {
    // this will never be called
}

/**
 * Releases itself and the contained figure.
 */
public void release() {
    super.release();
    fComponent.removeFromContainer(this);
    fComponent.release();
}

/**
 * Propagates invalidate up the container chain.
 * @see FigureChangeListener
 */
public void figureInvalidated(FigureChangeEvent e) {
    if (listener() != null)
        listener().figureInvalidated(e);
}

public void figureChanged(FigureChangeEvent e) {
}

public void figureRemoved(FigureChangeEvent e) {
}

/**
 * Propagates figureRequestUpdate up the container chain.
 * @see FigureChangeListener
 */
public void figureRequestUpdate(FigureChangeEvent e) {
    if (listener() != null)
        listener().figureRequestUpdate(e);
}

/**
 * Propagates the removeFromDrawing request up to the container.
 * @see FigureChangeListener
 */
public void figureRequestRemove(FigureChangeEvent e) {
    if (listener() != null)
        listener().figureRequestRemove(new
FigureChangeEvent(this));
}

/**
 * Forwards figures to its contained figure.
 */
public FigureEnumeration figures() {

```

```

        return fComponent.figures();
    }

    /**
     * Forwards decompose to its contained figure.
     */
    public FigureEnumeration decompose() {
        return fComponent.decompose();
    }

    /**
     * Forwards setAttribute to its contained figure.
     */
    public void setAttribute(String name, Object value) {
        fComponent.setAttribute(name, value);
    }

    /**
     * Forwards getAttribute to its contained figure.
     */
    public Object getAttribute(String name) {
        return fComponent.getAttribute(name);
    }

    /**
     * Returns the locator used to located connected text.
     */
    public Locator connectedTextLocator(Figure text) {
        return fComponent.connectedTextLocator(text);
    }

    /**
     * Returns the Connector for the given location.
     */
    public Connector connectorAt(int x, int y) {
        return fComponent.connectorAt(x, y);
    }

    /**
     * Forwards the connector visibility request to its component.
     */
    public void connectorVisibility(boolean isVisible) {
        fComponent.connectorVisibility(isVisible);
    }

    /**
     * Writes itself and the contained figure to the StorableOutput.
     */
    public void write(StorableOutput dw) {
        super.write(dw);
        dw.writeStorable(fComponent);
    }

    /**
     * Reads itself and the contained figure from the StorableInput.
     */
    public void read(StorableInput dr) throws IOException {
        super.read(dr);
    }

```

```

        decorate((Figure)dr.readStorable());
    }

    private void readObject(ObjectInputStream s)
        throws ClassNotFoundException, IOException {

        s.defaultReadObject();

        fComponent.addToContainer(this);
    }
}

```

Clase DeleteCommand

Comando para borrar la selección.

Se encuentra en el fichero DeleteCommand.java y su contenido es el siguiente:

```

/*
 * @(#)DeleteCommand.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.util.*;
import CH.ifa.draw.util.Command;
import CH.ifa.draw.framework.*;

/**
 * Command to delete the selection.
 */
public class DeleteCommand extends FigureTransferCommand {

    /**
     * Constructs a delete command.
     * @param name the command name
     * @param view the target view
     */
    public DeleteCommand(String name, DrawingView view) {
        super(name, view);
    }

    public void execute() {
        deleteSelection();
        fView.checkDamage();
    }

    public boolean isExecutable() {
        return fView.selectionCount() > 0;
    }
}

```

Clase DragTracker

DragTracker implementa el arrastre de la figura seleccionada.

Se encuentra en el fichero DragTracker.java y su contenido es el siguiente:

```
/*
 * @(#)DragTracker.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.awt.event.MouseEvent;
import java.util.*;
import CH.ifa.draw.framework.*;

/**
 * DragTracker implements the dragging of the clicked
 * figure.
 *
 * @see SelectionTool
 */
public class DragTracker extends AbstractTool {

    private Figure fAnchorFigure;
    private int fLastX, fLastY; // previous mouse position
    private boolean fMoved = false;

    public DragTracker(DrawingView view, Figure anchor) {
        super(view);
        fAnchorFigure = anchor;
    }

    public void mouseDown(MouseEvent e, int x, int y) {
        super.mouseDown(e, x, y);
        fLastX = x;
        fLastY = y;

        if (e.isShiftDown()) {
            view().toggleSelection(fAnchorFigure);
            fAnchorFigure = null;
        } else if (!view().selection().contains(fAnchorFigure)) {
            view().clearSelection();
            view().addToSelection(fAnchorFigure);
        }
    }

    public void mouseDrag(MouseEvent e, int x, int y) {
        super.mouseDrag(e, x, y);
        fMoved = (Math.abs(x - fAnchorX) > 4) || (Math.abs(y -
fAnchorY) > 4);

        if (fMoved) {
```



```

        FigureEnumeration figures = view().selectionElements();
        while (figures.hasMoreElements())
            figures.nextFigure().moveBy(x - fLastX, y - fLastY);
    }
    fLastX = x;
    fLastY = y;
}
}

```

Clase DuplicateCommand

Duplica la selección y selecciona el duplicado.

Se encuentra en el fichero DuplicateCommand.java y su contenido es el siguiente:

```

/*
 * @(#)DuplicateCommand.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.util.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

/**
 * Duplicate the selection and select the duplicates.
 */
public class DuplicateCommand extends FigureTransferCommand {

    /**
     * Constructs a duplicate command.
     * @param name the command name
     * @param view the target view
     */
    public DuplicateCommand(String name, DrawingView view) {
        super(name, view);
    }

    public void execute() {
        FigureSelection selection = fView.getFigureSelection();

        fView.clearSelection();

        Vector figures =
            (Vector)selection.getData(FigureSelection.TYPE);
        insertFigures(figures, 10, 10);
        fView.checkDamage();
    }

    public boolean isExecutable() {
        return fView.selectionCount() > 0;
    }
}

```

```
}
```

Clase FigureChangeEventMulticaster

Maneja una lista de FigureChangeListener que serán notificados de específicos FigureChangeEvents.

Se encuentra en el fichero FigureChangeEventMulticaster.java y su contenido es el siguiente:

```
/*
 * @(#)FigureChangeEventMulticaster.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;

/**
 * Manages a list of FigureChangeListener to be notified of
 * specific FigureChangeEvents.
 */

public class FigureChangeEventMulticaster extends
    AWTEventMulticaster implements FigureChangeListener {

    public FigureChangeEventMulticaster(EventListener a, EventListener
b) {
        super(a, b);
    }

    public void figureInvalidated(FigureChangeEvent e) {
        ((FigureChangeListener)a).figureInvalidated(e);
        ((FigureChangeListener)b).figureInvalidated(e);
    }

    public void figureRequestRemove(FigureChangeEvent e) {
        ((FigureChangeListener)a).figureRequestRemove(e);
        ((FigureChangeListener)b).figureRequestRemove(e);
    }

    public void figureRequestUpdate(FigureChangeEvent e) {
        ((FigureChangeListener)a).figureRequestUpdate(e);
        ((FigureChangeListener)b).figureRequestUpdate(e);
    }

    public void figureChanged(FigureChangeEvent e) {
```

```

        ((FigureChangeListener)a).figureChanged(e);
        ((FigureChangeListener)b).figureChanged(e);
    }

    public void figureRemoved(FigureChangeEvent e) {
        ((FigureChangeListener)a).figureRemoved(e);
        ((FigureChangeListener)b).figureRemoved(e);
    }

    public static FigureChangeListener add(FigureChangeListener a,
    FigureChangeListener b) {
        return (FigureChangeListener)addInternal(a, b);
    }

    public static FigureChangeListener remove(FigureChangeListener l,
    FigureChangeListener oldl) {
        return (FigureChangeListener) removeInternal(l, oldl);
    }

    protected EventListener remove(EventListener oldl)
    {
        if (oldl == a)
            return b;
        if (oldl == b)
            return a;
        EventListener a2 = removeInternal((FigureChangeListener)a,
oldl);
        EventListener b2 = removeInternal((FigureChangeListener)b,
oldl);
        if (a2 == a && b2 == b)
            return this;
        else
            return addInternal((FigureChangeListener)a2,
(FigureChangeListener)b2);
    }

    protected static EventListener addInternal(FigureChangeListener a,
    FigureChangeListener b) {
        if (a == null) return b;
        if (b == null) return a;
        return new FigureChangeEventMulticaster(a, b);
    }

    protected static EventListener removeInternal(EventListener l,
    EventListener oldl) {
        if (l == oldl || l == null) {
            return null;
        } else if (l instanceof FigureChangeEventMulticaster) {
            return ((FigureChangeEventMulticaster)l).remove(oldl);
        } else {
            return l;        // it's not here
        }
    }
}

```

Clase FigureEnumerator

Una Enumeration para un Vector de Figures.

Se encuentra en el fichero FigureEnumerator.java y su contenido es el siguiente:

```
/*
 * @(#)FigureEnumerator.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.framework.*;
import java.util.*;

/**
 * An Enumeration for a Vector of Figures.
 */
public final class FigureEnumerator implements FigureEnumeration {
    Enumeration fEnumeration;

    public FigureEnumerator(Vector v) {
        fEnumeration = v.elements();
    }

    /**
     * Returns true if the enumeration contains more elements; false
     * if its empty.
     */
    public boolean hasMoreElements() {
        return fEnumeration.hasMoreElements();
    }

    /**
     * Returns the next element of the enumeration. Calls to this
     * method will enumerate successive elements.
     * @exception NoSuchElementException If no more elements exist.
     */
    public Object nextElement() {
        return fEnumeration.nextElement();
    }

    /**
     * Returns the next element of the enumeration. Calls to this
     * method will enumerate successive elements.
     * @exception NoSuchElementException If no more elements exist.
     */
    public Figure nextFigure() {
        return (Figure)fEnumeration.nextElement();
    }
}
```

Clase FigureTransferCommand

La clase base común para los comandos que transfieren figuras entre un drawing y el portapapeles (clipboard).

Se encuentra en el fichero FigureTransferCommand.java y su contenido es el siguiente:

```
/*
 * @(#)FigureTransferCommand.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.util.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

/**
 * Common base class for commands that transfer figures
 * between a drawing and the clipboard.
 */
abstract class FigureTransferCommand extends Command {

    protected DrawingView fView;

    /**
     * Constructs a drawing command.
     * @param name the command name
     * @param view the target view
     */
    protected FigureTransferCommand(String name, DrawingView view) {
        super(name);
        fView = view;
    }

    /**
     * Deletes the selection from the drawing.
     */
    protected void deleteSelection() {
        fView.drawing().removeAll(fView.selection());
        fView.clearSelection();
    }

    /**
     * Copies the selection to the clipboard.
     */
    protected void copySelection() {
        FigureSelection selection = fView.getFigureSelection();
        Clipboard.getClipboard().setContents(selection);
    }

    /**
     * Inserts a vector of figures and translates them by the
     * given offset.
     */
}
```

```

    */
    protected void insertFigures(Vector figures, int dx, int dy) {
        FigureEnumeration e = new FigureEnumerator(figures);
        while (e.hasMoreElements()) {
            Figure figure = e.nextFigure();
            figure.moveBy(dx, dy);
            figure = fView.add(figure);
            fView.addToSelection(figure);
        }
    }
}

```

Clase GridConstrainer

Obliga a un punto de tal forma que caiga sobre una grid.

Se encuentra en el fichero GridConstrainer.java y su contenido es el siguiente:

```

/*
 * @(#)GridConstrainer.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.io.Serializable;

import CH.ifa.draw.framework.PointConstrainer;

/**
 * Constrains a point such that it falls on a grid.
 *
 * @see DrawingView
 */

public class GridConstrainer implements PointConstrainer, Serializable
{
    private int fGridX;
    private int fGridY;

    public GridConstrainer(int x, int y) {
        fGridX = Math.max(1, x);
        fGridY = Math.max(1, y);
    }

    /**
     * Constrains the given point.
     * @return constrained point.
     */
    public Point constrainPoint(Point p) {

```

```

        p.x = ((p.x+fGridX/2) / fGridX) * fGridX;
        p.y = ((p.y+fGridY/2) / fGridY) * fGridY;
        return p;
    }

    /**
     * Gets the x offset to move an object.
     */
    public int getStepX() {
        return fGridX;
    }

    /**
     * Gets the y offset to move an object.
     */
    public int getStepY() {
        return fGridY;
    }
}

```

Clase HandleTracker

HandleTracker implementa interacciones con los manejadores de una figura.

Se encuentra en el fichero HandleTracker.java y su contenido es el siguiente:

```

/*
 * @(#)HandleTracker.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.awt.event.MouseEvent;
import CH.ifa.draw.framework.*;

/**
 * HandleTracker implements interactions with the handles
 * of a Figure.
 *
 * @see SelectionTool
 */
public class HandleTracker extends AbstractTool {

    private Handle fAnchorHandle;

    public HandleTracker(DrawingView view, Handle anchorHandle) {
        super(view);
        fAnchorHandle = anchorHandle;
    }

    public void mouseDown(MouseEvent e, int x, int y) {
        super.mouseDown(e, x, y);
    }
}

```

```

        fAnchorHandle.invokeStart(x, y, view());
    }

    public void mouseDrag(MouseEvent e, int x, int y) {
        super.mouseDrag(e, x, y);
        fAnchorHandle.invokeStep(x, y, fAnchorX, fAnchorY, view());
    }

    public void mouseUp(MouseEvent e, int x, int y) {
        super.mouseDrag(e, x, y);
        fAnchorHandle.invokeEnd(x, y, fAnchorX, fAnchorY, view());
    }
}

```

Clase LocatorConnector

Un LocatorConnector localiza puntos de conexión con la ayuda de un Locator. Soporta la definición de puntos de conexión para localizaciones semánticas.

Se encuentra en el fichero LocatorConnector.java y su contenido es el siguiente:

```

/*
 * @(#)LocatorConnector.java 5.1
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.*;

/**
 * A LocatorConnector locates connection points with
 * the help of a Locator. It supports the definition
 * of connection points to semantic locations.
 * @see Locator
 * @see Connector
 */
public class LocatorConnector extends AbstractConnector {

    /**
     * The standard size of the connector. The display box
     * is centered around the located point.
     */
    public static final int SIZE = 8;

    private Locator fLocator;

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = 5062833203337604181L;
    private int locatorConnectorSerializedDataVersion = 1;

```



```

public LocatorConnector() { // only used for Storable
    fLocator = null;
}

public LocatorConnector(Figure owner, Locator l) {
    super(owner);
    fLocator = l;
}

protected Point locate(ConnectionFigure connection) {
    return fLocator.locate(owner());
}

/**
 * Tests if a point is contained in the connector.
 */
public boolean containsPoint(int x, int y) {
    return displayBox().contains(x, y);
}

/**
 * Gets the display box of the connector.
 */
public Rectangle displayBox() {
    Point p = fLocator.locate(owner());
    return new Rectangle(
        p.x - SIZE / 2,
        p.y - SIZE / 2,
        SIZE,
        SIZE);
}

/**
 * Draws this connector.
 */
public void draw(Graphics g) {
    Rectangle r = displayBox();

    g.setColor(Color.blue);
    g.fillOval(r.x, r.y, r.width, r.height);
    g.setColor(Color.black);
    g.drawOval(r.x, r.y, r.width, r.height);
}

/**
 * Stores the arrow tip to a StorableOutput.
 */
public void write(StorableOutput dw) {
    super.write(dw);
    dw.writeStorable(fLocator);
}

/**
 * Reads the arrow tip from a StorableInput.
 */
public void read(StorableInput dr) throws IOException {
    super.read(dr);
}

```

```

        fLocator = (Locator)dr.readStorable();
    }
}

```

Clase LocatorHandle

Un LocatorHandle implementa un Handle para delegar la localización pedida a un objeto Locator.

Se encuentra en el fichero LocatorHandle.java y su contenido es el siguiente:

```

/*
 * @(#)LocatorHandle.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.Point;
import CH.ifa.draw.framework.*;

/**
 * A LocatorHandle implements a Handle by delegating the location
 * requests to
 * a Locator object.
 *
 * @see Locator
 */
public class LocatorHandle extends AbstractHandle {

    private Locator    fLocator;

    /**
     * Initializes the LocatorHandle with the given Locator.
     */
    public LocatorHandle(Figure owner, Locator l) {
        super(owner);
        fLocator = l;
    }

    /**
     * Locates the handle on the figure by forwarding the request
     * to its figure.
     */
    public Point locate() {
        return fLocator.locate(owner());
    }
}

```

Clase NullHandle

Un manejador que no cambia la figura que lo posee. Su único propósito es mostrar que una figura esta seleccionada.

Patrones de diseño

Null Object Null Object permite tratar los manejadores que no hacen nada del mismo modo que otros manejadores.

Se encuentra en el fichero NullHandle.java y su contenido es el siguiente:

```
/*
 * @(#)NullHandle.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import CH.ifa.draw.framework.*;

/**
 * A handle that doesn't change the owned figure. Its only purpose is
 * to show feedback that a figure is selected.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b>NullObject</b><br>
 * NullObject enables to treat handles that don't do
 * anything in the same way as other handles.
 *
 */
public class NullHandle extends LocatorHandle {

    /**
     * The handle's locator.
     */
    protected Locator fLocator;

    public NullHandle(Figure owner, Locator locator) {
        super(owner, locator);
    }

    /**
     * Draws the NullHandle. NullHandles are drawn as a
     * red framed rectangle.
     */
    public void draw(Graphics g) {
        Rectangle r = displayBox();

        g.setColor(Color.black);
        g.drawRect(r.x, r.y, r.width, r.height);
    }
}
```

```
}
```

Clase OffsetLocator

Un locator para desplazar otro Locator.

Se encuentra en el fichero OffsetLocator.java y su contenido es el siguiente:

```
/*
 * @(#)OffsetLocator.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.*;

/**
 * A locator to offset another Locator.
 * @see Locator
 */
public class OffsetLocator extends AbstractLocator {

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = 2679950024611847621L;
    private int offsetLocatorSerializedDataVersion = 1;

    private Locator fBase;
    private int fOffsetX;
    private int fOffsetY;

    public OffsetLocator() {
        fBase = null;
        fOffsetX = 0;
        fOffsetY = 0;
    }

    public OffsetLocator(Locator base) {
        this();
        fBase = base;
    }

    public OffsetLocator(Locator base, int offsetX, int offsetY) {
        this(base);
        fOffsetX = offsetX;
        fOffsetY = offsetY;
    }

    public Point locate(Figure owner) {
```

```

        Point p = fBase.locate(owner);
        p.x += fOffsetX;
        p.y += fOffsetY;
        return p;
    }

    public void moveBy(int dx, int dy) {
        fOffsetX += dx;
        fOffsetY += dy;
    }

    public void write(StorableOutput dw) {
        super.write(dw);
        dw.writeInt(fOffsetX);
        dw.writeInt(fOffsetY);
        dw.writeStorable(fBase);
    }

    public void read(StorableInput dr) throws IOException {
        super.read(dr);
        fOffsetX = dr.readInt();
        fOffsetY = dr.readInt();
        fBase = (Locator)dr.readStorable();
    }
}

```

Clase PasteCommand

Comando para insertar el contenido del portapapeles en el dibujo.

Se encuentra en el fichero PasteCommand.java y su contenido es el siguiente:

```

/*
 * @(#)PasteCommand.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.util.*;
import java.awt.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

/**
 * Command to insert the clipboard into the drawing.
 * @see Clipboard
 */
public class PasteCommand extends FigureTransferCommand {

    /**
     * Constructs a paste command.
     * @param name the command name
     * @param image the pathname of the image
     */
}

```

```

    * @param view the target view
    */
    public PasteCommand(String name, DrawingView view) {
        super(name, view);
    }

    public void execute() {
        Point lastClick = fView.lastClick();
        FigureSelection selection =
        (FigureSelection)Clipboard.getClipboard().getContents();
        if (selection != null) {
            Vector figures =
            (Vector)selection.getData(FigureSelection.TYPE);
            if (figures.size() == 0)
                return;

            Rectangle r = bounds(figures.elements());
            fView.clearSelection();

            insertFigures(figures, lastClick.x-r.x, lastClick.y-r.y);
            fView.checkDamage();
        }
    }

    Rectangle bounds(Enumeration k) {
        Rectangle r = ((Figure) k.nextElement()).displayBox();
        while (k.hasMoreElements())
            r.add(((Figure) k.nextElement()).displayBox());
        return r;
    }
}

```

Clase RelativeLocator

Un locator que especifica un punto que es relativo a los bordes de una figura.

Se encuentra en el fichero RelativeLocator.java y su contenido es el siguiente:

```

/*
 * @(#)RelativeLocator.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.*;

/**
 * A locator that specifies a point that is relative to the bounds
 * of a figure.
 * @see Locator
 */

```

```

    */
public class RelativeLocator extends AbstractLocator {
    /*
     * Serialization support.
     */
    private static final long serialVersionUID = 2619148876087898602L;
    private int relativeLocatorSerializedDataVersion = 1;

    double fRelativeX;
    double fRelativeY;

    public RelativeLocator() {
        fRelativeX = 0.0;
        fRelativeY = 0.0;
    }

    public RelativeLocator(double relativeX, double relativeY) {
        fRelativeX = relativeX;
        fRelativeY = relativeY;
    }

    public Point locate(Figure owner) {
        Rectangle r = owner.displayBox();
        return new Point(
            r.x + (int)(r.width*fRelativeX),
            r.y + (int)(r.height*fRelativeY)
        );
    }

    public void write(StorableOutput dw) {
        super.write(dw);
        dw.writeDouble(fRelativeX);
        dw.writeDouble(fRelativeY);
    }

    public void read(StorableInput dr) throws IOException {
        super.read(dr);
        fRelativeX = dr.readDouble();
        fRelativeY = dr.readDouble();
    }

    static public Locator east() {
        return new RelativeLocator(1.0, 0.5);
    }

    /**
     * North.
     */
    static public Locator north() {
        return new RelativeLocator(0.5, 0.0);
    }

    /**
     * West.
     */
    static public Locator west() {
        return new RelativeLocator(0.0, 0.5);
    }
}

```

```

/**
 * North east.
 */
static public Locator northEast() {
    return new RelativeLocator(1.0, 0.0);
}

/**
 * North west.
 */
static public Locator northWest() {
    return new RelativeLocator(0.0, 0.0);
}

/**
 * South.
 */
static public Locator south() {
    return new RelativeLocator(0.5, 1.0);
}

/**
 * South east.
 */
static public Locator southEast() {
    return new RelativeLocator(1.0, 1.0);
}

/**
 * South west.
 */
static public Locator southWest() {
    return new RelativeLocator(0.0, 1.0);
}

/**
 * Center.
 */
static public Locator center() {
    return new RelativeLocator(0.5, 0.5);
}
}

```

Clase ReverseFigureEnumerator

Un Enumeration que enumera un vector de figuras desde atrás (size-1) hacia delante (0).

Se encuentra en el fichero ReverseFigureEnumerator.java y su contenido es el siguiente:

```

/*
 * @(#)ReverseFigureEnumerator.java 5.1
 */

```



```

*/

package CH.ifa.draw.standard;

import java.util.*;
import CH.ifa.draw.util.ReverseVectorEnumerator;
import CH.ifa.draw.framework.*;

/**
 * An Enumeration that enumerates a vector of figures back (size-1) to
 * front (0).
 */
public final class ReverseFigureEnumerator implements
FigureEnumeration {
    ReverseVectorEnumerator fEnumeration;

    public ReverseFigureEnumerator(Vector v) {
        fEnumeration = new ReverseVectorEnumerator(v);
    }

    /**
     * Returns true if the enumeration contains more elements; false
     * if its empty.
     */
    public boolean hasMoreElements() {
        return fEnumeration.hasMoreElements();
    }

    /**
     * Returns the next element of the enumeration. Calls to this
     * method will enumerate successive elements.
     * @exception NoSuchElementException If no more elements exist.
     */
    public Object nextElement() {
        return fEnumeration.nextElement();
    }

    /**
     * Returns the next element casted as a figure of the enumeration.
     * Calls to this method will enumerate successive elements.
     * @exception NoSuchElementException If no more elements exist.
     */
    public Figure nextFigure() {
        return (Figure)fEnumeration.nextElement();
    }
}

```

Clase SelectAreaTracker

SelectAreaTracker implementa una selección de un area.

Se encuentra en el fichero SelectAreaTracker.java y su contenido es el siguiente:

```

/*

```

```

* @(#)SelectAreaTracker.java 5.1
*
*/

package CH.ifa.draw.standard;

import java.awt.*;
import java.awt.event.MouseEvent;
import java.util.*;
import CH.ifa.draw.framework.*;

/**
 * SelectAreaTracker implements a rubberband selection of an area.
 */
public class SelectAreaTracker extends AbstractTool {

    private Rectangle fSelectGroup;

    public SelectAreaTracker(DrawingView view) {
        super(view);
    }

    public void mouseDown(MouseEvent e, int x, int y) {
        // use event coordinates to supress any kind of
        // transformations like constraining points to a grid
        super.mouseDown(e, e.getX(), e.getY());
        rubberBand(fAnchorX, fAnchorY, fAnchorX, fAnchorY);
    }

    public void mouseDrag(MouseEvent e, int x, int y) {
        super.mouseDrag(e, x, y);
        eraseRubberBand();
        rubberBand(fAnchorX, fAnchorY, x, y);
    }

    public void mouseUp(MouseEvent e, int x, int y) {
        super.mouseUp(e, x, y);
        eraseRubberBand();
        selectGroup(e.isShiftDown());
    }

    private void rubberBand(int x1, int y1, int x2, int y2)
    {
        fSelectGroup = new Rectangle(new Point(x1, y1));
        fSelectGroup.add(new Point(x2, y2));
        drawXORRect(fSelectGroup);
    }

    private void eraseRubberBand()
    {
        drawXORRect(fSelectGroup);
    }

    private void drawXORRect(Rectangle r)
    {
        Graphics g = view().getGraphics();
        g.setXORMode(view().getBackground());
        g.setColor(Color.black);
    }
}

```

```
        q.drawRect(r.x, r.y, r.width, r.height);
    }

    private void selectGroup(boolean toggle)
    {
        FigureEnumeration k = drawing().figuresReverse();
        while (k.hasMoreElements()) {
            Figure figure = k.nextFigure();
            Rectangle r2 = figure.displayBox();
            if (fSelectGroup.contains(r2.x, r2.y) &&
fSelectGroup.contains(r2.x+r2.width, r2.y+r2.height)) {
                if (toggle)
                    view().toggleSelection(figure);
                else
                    view().addToSelection(figure);
            }
        }
    }
}
```

Clase SelectionTool

Herramienta para seleccionar y manipular figuras. Una herramienta de selección esta en uno de los tres estados, es decir, selección del fondo, selección de una figura, manipulación. Los diferentes estados son manejados por diferentes herramientas hijo.

Patrones de diseño

State SelectionTool es el *Contexto* y el hijo es el *Estado*. SelectionTool delega el comportamiento específico del estado a su herramienta hijo actual.

Se encuentra en el fichero SelectionTool.java y su contenido es el siguiente:

```
/*
 * @(#)SelectionTool.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import java.awt.event.MouseEvent;
import java.util.Vector;
import CH.ifa.draw.framework.*;

/**
 * Tool to select and manipulate figures.
 * A selection tool is in one of three states, e.g., background
 * selection, figure selection, handle manipulation. The different
 * states are handled by different child tools.
 * <hr>
 * <b>Design Patterns</b><P>
 * 
 * <b><a href=../pattlets/sld032.htm>State</a></b><br>
 * SelectionTool is the StateContext and child is the State.
 * The SelectionTool delegates state specific
 * behavior to its current child tool.
 * <hr>
 */

public class SelectionTool extends AbstractTool {

    private Tool fChild = null;

    public SelectionTool(DrawingView view) {
        super(view);
    }

    /**
     * Handles mouse down events and starts the corresponding tracker.
     */
}
```

```

public void mouseDown(MouseEvent e, int x, int y)
{
    // on Windows NT: AWT generates additional mouse down events
    // when the left button is down && right button is clicked.
    // To avoid dead locks we ignore such events
    if (fChild != null)
        return;

    view().freezeView();

    Handle handle = view().findHandle(e.getX(), e.getY());
    if (handle != null) {
        fChild = createHandleTracker(view(), handle);
    }
    else {
        Figure figure = drawing().findFigure(e.getX(), e.getY());
        if (figure != null) {
            fChild = createDragTracker(view(), figure);
        }
        else {
            if (!e.isShiftDown()) {
                view().clearSelection();
            }
            fChild = createAreaTracker(view());
        }
    }
    fChild.mouseDown(e, x, y);
}

/**
 * Handles mouse drag events. The events are forwarded to the
 * current tracker.
 */
public void mouseDrag(MouseEvent e, int x, int y) {
    if (fChild != null) // JDK1.1 doesn't guarantee mouseDown,
                        // mouseDrag, mouseUp
        fChild.mouseDrag(e, x, y);
}

/**
 * Handles mouse up events. The events are forwarded to the
 * current tracker.
 */
public void mouseUp(MouseEvent e, int x, int y) {
    view().unfreezeView();
    if (fChild != null) // JDK1.1 doesn't guarantee mouseDown,
                        // mouseDrag, mouseUp
        fChild.mouseUp(e, x, y);
    fChild = null;
}

/**
 * Factory method to create a Handle tracker. It is used to track
 * a handle.
 */
protected Tool createHandleTracker(DrawingView view, Handle
handle) {
    return new HandleTracker(view, handle);
}

```

```

    }

    /**
     * Factory method to create a Drag tracker. It is used to drag a
     * figure.
     */
    protected Tool createDragTracker(DrawingView view, Figure f) {
        return new DragTracker(view, f);
    }

    /**
     * Factory method to create an area tracker. It is used to select
     * an area.
     */
    protected Tool createAreaTracker(DrawingView view) {
        return new SelectAreaTracker(view);
    }
}

```

Clase SendToBackCommand

Un comando que envía la selección al fondo del drawing.

Se encuentra en el fichero SendToBackCommand.java y su contenido es el siguiente:

```

/*
 * @(#)SendToBackCommand.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.util.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

/**
 * A command to send the selection to the back of the drawing.
 */
public class SendToBackCommand extends Command {

    private DrawingView fView;

    /**
     * Constructs a send to back command.
     * @param name the command name
     * @param view the target view
     */
    public SendToBackCommand(String name, DrawingView view) {
        super(name);
        fView = view;
    }

    public void execute() {

```

```

        FigureEnumeration k = new
ReverseFigureEnumerator(fView.selectionZOrdered());
        while (k.hasMoreElements()) {
            fView.drawing().sendToBack(k.nextFigure());
        }
        fView.checkDamage();
    }

    public boolean isExecutable() {
        return fView.selectionCount() > 0;
    }
}

```

Clase SimpleUpdateStrategy

El SimpleUpdateStrategy implementa una estrategia de actualización que redibuja directamente un DrawingView.

Se encuentra en el fichero SimpleUpdateStrategy.java y su contenido es el siguiente:

```

/*
 * @(#)SimpleUpdateStrategy.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import CH.ifa.draw.framework.*;

/**
 * The SimpleUpdateStrategy implements an update
 * strategy that directly redraws a DrawingView.
 * @see DrawingView
 */
public class SimpleUpdateStrategy implements Painter {

    /**
     * Serialization support. In JavaDraw only the Drawing is
     * serialized.
     * However, for beans support SimpleUpdateStrategy supports
     * serialization
     */
    private static final long serialVersionUID = -
7539925820692134566L;

    /**
     * Draws the view contents.
     */
    public void draw(Graphics g, DrawingView view) {
        view.drawAll(g);
    }
}

```

Clase StandardDrawing

La implementación estándar de un interface Drawing.

Se encuentra en el fichero StandardDrawing.java y su contenido es el siguiente:

```
/*
 * @(#)StandardDrawing.java 5.1
 *
 */

package CH.ifa.draw.standard;

import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;
import java.awt.*;
import java.util.*;
import java.io.*;

/**
 * The standard implementation of the Drawing interface.
 *
 * @see Drawing
 */

public class StandardDrawing extends CompositeFigure implements
Drawing {

    /**
     * the registered listeners
     */
    private transient Vector          fListeners;

    /**
     * boolean that serves as a condition variable
     * to lock the access to the drawing.
     * The lock is recursive and we keep track of the current
     * lock holder.
     */
    private transient Thread          fDrawingLockHolder = null;

    /**
     * Serialization support
     */
    private static final long serialVersionUID = -
2602151437447962046L;
    private int drawingSerializedDataVersion = 1;

    /**
     * Constructs the Drawing.
     */
    public StandardDrawing() {
```



```

        super();
        fListeners = new Vector(2);
    }

    /**
     * Adds a listener for this drawing.
     */
    public void addDrawingChangeListener(DrawingChangeListener
listener) {
        fListeners.addElement(listener);
    }

    /**
     * Removes a listener from this drawing.
     */
    public void removeDrawingChangeListener(DrawingChangeListener
listener) {
        fListeners.removeElement(listener);
    }

    /**
     * Adds a listener for this drawing.
     */
    public Enumeration drawingChangeListeners() {
        return fListeners.elements();
    }

    /**
     * Removes the figure from the drawing and releases it.
     */

    public synchronized Figure remove(Figure figure) {
        // ensure that we remove the top level figure in a drawing
        if (figure.listener() != null) {
            figure.listener().figureRequestRemove(new
FigureChangeEvent(figure, null));
            return figure;
        }
        return null;
    }

    /**
     * Handles a removeFromDrawing request that
     * is passed up the figure container hierarchy.
     * @see FigureChangeListener
     */
    public void figureRequestRemove(FigureChangeEvent e) {
        Figure figure = e.getFigure();
        if (fFigures.contains(figure)) {
            fFigures.removeElement(figure);
            figure.removeFromContainer(this);    // will invalidate
                                                //figure
            figure.release();
        } else
            System.out.println("Attempt to remove non-existing
figure");
    }

```

```

    }

    /**
     * Invalidates a rectangle and merges it with the
     * existing damaged area.
     * @see FigureChangeListener
     */
    public void figureInvalidated(FigureChangeEvent e) {
        if (fListeners != null) {
            for (int i = 0; i < fListeners.size(); i++) {
                DrawingChangeListener l =
                    (DrawingChangeListener)fListeners.elementAt(i);
                l.drawingInvalidated(new DrawingChangeEvent(this,
                    e.getInvalidatedRectangle()));
            }
        }

    }

    /**
     * Forces an update
     */
    public void figureRequestUpdate(FigureChangeEvent e) {
        if (fListeners != null) {
            for (int i = 0; i < fListeners.size(); i++) {
                DrawingChangeListener l =
                    (DrawingChangeListener)fListeners.elementAt(i);
                l.drawingRequestUpdate(new DrawingChangeEvent(this,
                    null));
            }
        }

    }

    /**
     * Return's the figure's handles. This is only used when a drawing
     * is nested inside another drawing.
     */
    public Vector handles() {
        Vector handles = new Vector();
        handles.addElement(new NullHandle(this,
            RelativeLocator.northWest()));
        handles.addElement(new NullHandle(this,
            RelativeLocator.northEast()));
        handles.addElement(new NullHandle(this,
            RelativeLocator.southWest()));
        handles.addElement(new NullHandle(this,
            RelativeLocator.southEast()));
        return handles;
    }

    /**
     * Gets the display box. This is the union of all figures.
     */
    public Rectangle displayBox() {
        if (fFigures.size() > 0) {
            FigureEnumeration k = figures();

            Rectangle r = k.nextFigure().displayBox();
        }
    }

```

```

        while (k.hasMoreElements())
            r.add(k.nextFigure().displayBox());
        return r;
    }
    return new Rectangle(0, 0, 0, 0);
}

public void basicDisplayBox(Point p1, Point p2) {
}

/**
 * Acquires the drawing lock.
 */
public synchronized void lock() {
    // recursive lock
    Thread current = Thread.currentThread();
    if (fDrawingLockHolder == current)
        return;
    while (fDrawingLockHolder != null) {
        try { wait(); } catch (InterruptedException ex) { }
    }
    fDrawingLockHolder = current;
}

/**
 * Releases the drawing lock.
 */
public synchronized void unlock() {
    if (fDrawingLockHolder != null) {
        fDrawingLockHolder = null;
        notifyAll();
    }
}

private void readObject(ObjectInputStream s)
    throws ClassNotFoundException, IOException {

    s.defaultReadObject();

    fListeners = new Vector(2);
}
}

```

Clase StandardDrawingView

La implementación estándar de un interface DrawingView.

Se encuentra en el fichero StandardDrawingView.java y su contenido es el siguiente:

```

/*
 * @(#)StandardDrawingView.java 5.1
 *
 */

```

```

package CH.ifa.draw.standard;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

/**
 * The standard implementation of DrawingView.
 * @see DrawingView
 * @see Painter
 * @see Tool
 */

public class StandardDrawingView
    extends Panel
    implements DrawingView,
               MouseListener,
               MouseMotionListener,
               KeyListener {

    /**
     * The DrawingEditor of the view.
     * @see #tool
     * @see #setStatus
     */
    transient private DrawingEditor fEditor;

    /**
     * The shown drawing.
     */
    private Drawing fDrawing;

    /**
     * the accumulated damaged area
     */
    private transient Rectangle fDamage = null;

    /**
     * The list of currently selected figures.
     */
    transient private Vector fSelection;

    /**
     * The shown selection handles.
     */
    transient private Vector fSelectionHandles;

    /**
     * The preferred size of the view
     */
    private Dimension fViewSize;

    /**
     * The position of the last mouse click
     * inside the view.

```

```

    */
    private Point fLastClick;

    /**
     * A vector of optional backgrounds. The vector maintains
     * a list a view painters that are drawn before the contents,
     * that is in the background.
     */
    private Vector fBackgrounds = null;

    /**
     * A vector of optional foregrounds. The vector maintains
     * a list a view painters that are drawn after the contents,
     * that is in the foreground.
     */
    private Vector fForegrounds = null;

    /**
     * The update strategy used to repair the view.
     */
    private Painter fUpdateStrategy;

    /**
     * The grid used to constrain points for snap to
     * grid functionality.
     */
    private PointConstrainer fConstrainer;

    /*
     * Serialization support. In JavaDraw only the Drawing is
     * serialized. However, for beans support StandardDrawingView
     * supports serialization
     */
    private static final long serialVersionUID = -
3878153366174603336L;
    private int drawingViewSerializedDataVersion = 1;

    /**
     * Constructs the view.
     */
    public StandardDrawingView(DrawingEditor editor, int width, int
height) {
        fEditor = editor;
        fViewSize = new Dimension(width,height);
        fLastClick = new Point(0, 0);
        fConstrainer = null;
        fSelection = new Vector();
        setDisplayUpdate(new BufferedUpdateStrategy());
        setBackground(Color.lightGray);

        addMouseListener(this);
        addMouseMotionListener(this);
        addKeyListener(this);
    }

    /**
     * Sets the view's editor.
     */

```

```

public void setEditor(DrawingEditor editor) {
    fEditor = editor;
}

/**
 * Gets the current tool.
 */
public Tool tool() {
    return fEditor.tool();
}

/**
 * Gets the drawing.
 */
public Drawing drawing() {
    return fDrawing;
}

/**
 * Sets and installs another drawing in the view.
 */
public void setDrawing(Drawing d) {
    clearSelection();

    if (fDrawing != null)
        fDrawing.removeDrawingChangeListener(this);

    fDrawing = d;
    if (fDrawing != null)
        fDrawing.addDrawingChangeListener(this);
    checkMinimumSize();
    repaint();
}

/**
 * Gets the editor.
 */
public DrawingEditor editor() {
    return fEditor;
}

/**
 * Adds a figure to the drawing.
 * @return the added figure.
 */
public Figure add(Figure figure) {
    return drawing().add(figure);
}

/**
 * Removes a figure from the drawing.
 * @return the removed figure
 */
public Figure remove(Figure figure) {
    return drawing().remove(figure);
}

/**

```

```

    * Adds a vector of figures to the drawing.
    */
    public void addAll(Vector figures) {
        FigureEnumeration k = new FigureEnumerator(figures);
        while (k.hasMoreElements())
            add(k.nextFigure());
    }

    /**
     * Gets the minimum dimension of the drawing.
     */
    public Dimension getMinimumSize() {
        return fViewSize;
    }

    /**
     * Gets the preferred dimension of the drawing..
     */
    public Dimension getPreferredSize() {
        return getMinimumSize();
    }

    /**
     * Sets the current display update strategy.
     * @see UpdateStrategy
     */
    public void setDisplayUpdate(Painter updateStrategy) {
        fUpdateStrategy = updateStrategy;
    }

    /**
     * Gets the currently selected figures.
     * @return a vector with the selected figures. The vector
     * is a copy of the current selection.
     */
    public Vector selection() {
        // protect the vector with the current selection
        return (Vector)fSelection.clone();
    }

    /**
     * Gets an enumeration over the currently selected figures.
     */
    public FigureEnumeration selectionElements() {
        return new FigureEnumerator(fSelection);
    }

    /**
     * Gets the currently selected figures in Z order.
     * @see #selection
     * @return a vector with the selected figures. The vector
     * is a copy of the current selection.
     */
    public Vector selectionZOrdered() {
        Vector result = new Vector(fSelection.size());
        FigureEnumeration figures = drawing().figures();

        while (figures.hasMoreElements()) {

```

```

        Figure f= figures.nextFigure();
        if (fSelection.contains(f)) {
            result.addElement(f);
        }
    }
    return result;
}

/**
 * Gets the number of selected figures.
 */
public int selectionCount() {
    return fSelection.size();
}

/**
 * Adds a figure to the current selection.
 */
public void addToSelection(Figure figure) {
    if (!fSelection.contains(figure)) {
        fSelection.addElement(figure);
        fSelectionHandles = null;
        figure.invalidate();
        selectionChanged();
    }
}

/**
 * Adds a vector of figures to the current selection.
 */
public void addToSelectionAll(Vector figures) {
    FigureEnumeration k = new FigureEnumerator(figures);
    while (k.hasMoreElements())
        addToSelection(k.nextFigure());
}

/**
 * Removes a figure from the selection.
 */
public void removeFromSelection(Figure figure) {
    if (fSelection.contains(figure)) {
        fSelection.removeElement(figure);
        fSelectionHandles = null;
        figure.invalidate();
        selectionChanged();
    }
}

/**
 * If a figure isn't selected it is added to the selection.
 * Otherwise it is removed from the selection.
 */
public void toggleSelection(Figure figure) {
    if (fSelection.contains(figure))
        removeFromSelection(figure);
    else
        addToSelection(figure);
    selectionChanged();
}

```



```

    }

    /**
     * Clears the current selection.
     */
    public void clearSelection() {
        Figure figure;

        FigureEnumeration k = selectionElements();

        while (k.hasMoreElements())
            k.nextFigure().invalidate();
        fSelection = new Vector();
        fSelectionHandles = null;
        selectionChanged();
    }

    /**
     * Gets an enumeration of the currently active handles.
     */
    private Enumeration selectionHandles() {
        if (fSelectionHandles == null) {
            fSelectionHandles = new Vector();
            FigureEnumeration k = selectionElements();
            while (k.hasMoreElements()) {
                Figure figure = k.nextFigure();
                Enumeration kk = figure.handles().elements();
                while (kk.hasMoreElements())
                    fSelectionHandles.addElement(kk.nextElement());
            }
        }
        return fSelectionHandles.elements();
    }

    /**
     * Gets the current selection as a FigureSelection. A
     * FigureSelection can be cut, copied, pasted.
     */
    public FigureSelection getFigureSelection() {
        return new FigureSelection(selectionZOrdered());
    }

    /**
     * Finds a handle at the given coordinates.
     * @return the hit handle, null if no handle is found.
     */
    public Handle findHandle(int x, int y) {
        Handle handle;

        Enumeration k = selectionHandles();
        while (k.hasMoreElements()) {
            handle = (Handle) k.nextElement();
            if (handle.containsPoint(x, y))
                return handle;
        }
        return null;
    }

```

```

/**
 * Informs that the current selection changed.
 * By default this event is forwarded to the
 * drawing editor.
 */
protected void selectionChanged() {
    fEditor.selectionChanged(this);
}

/**
 * Gets the position of the last click inside the view.
 */
public Point lastClick() {
    return fLastClick;
}

/**
 * Sets the grid spacing that is used to constrain points.
 */
public void setConstrainer(PointConstrainer c) {
    fConstrainer = c;
}

/**
 * Gets the current constrainer.
 */
public PointConstrainer getConstrainer() {
    return fConstrainer;
}

/**
 * Constrains a point to the current grid.
 */
protected Point constrainPoint(Point p) {
    // constrain to view size
    Dimension size = getSize();
    //p.x = Math.min(size.width, Math.max(1, p.x));
    //p.y = Math.min(size.height, Math.max(1, p.y));
    p.x = Geom.range(1, size.width, p.x);
    p.y = Geom.range(1, size.height, p.y);

    if (fConstrainer != null )
        return fConstrainer.constrainPoint(p);
    return p;
}

/**
 * Handles mouse down events. The event is delegated to the
 * currently active tool.
 * @return whether the event was handled.
 */
public void mousePressed(MouseEvent e) {
    requestFocus(); // JDK1.1
    Point p = constrainPoint(new Point(e.getX(), e.getY()));
    fLastClick = new Point(e.getX(), e.getY());
    tool().mouseDown(e, p.x, p.y);
    checkDamage();
}

```

```

/**
 * Handles mouse drag events. The event is delegated to the
 * currently active tool.
 * @return whether the event was handled.
 */
public void mouseDragged(MouseEvent e) {
    Point p = constrainPoint(new Point(e.getX(), e.getY()));
    tool().mouseDrag(e, p.x, p.y);
    checkDamage();
}

/**
 * Handles mouse move events. The event is delegated to the
 * currently active tool.
 * @return whether the event was handled.
 */
public void mouseMoved(MouseEvent e) {
    tool().mouseMove(e, e.getX(), e.getY());
}

/**
 * Handles mouse up events. The event is delegated to the
 * currently active tool.
 * @return whether the event was handled.
 */
public void mouseReleased(MouseEvent e) {
    Point p = constrainPoint(new Point(e.getX(), e.getY()));
    tool().mouseUp(e, p.x, p.y);
    checkDamage();
}

/**
 * Handles key down events. Cursor keys are handled
 * by the view the other key events are delegated to the
 * currently active tool.
 * @return whether the event was handled.
 */
public void keyPressed(KeyEvent e) {
    int code = e.getKeyCode();
    if ((code == KeyEvent.VK_BACK_SPACE) || (code ==
KeyEvent.VK_DELETE)) {
        Command cmd = new DeleteCommand("Delete", this);
        cmd.execute();
    } else if (code == KeyEvent.VK_DOWN || code == KeyEvent.VK_UP
||
        code == KeyEvent.VK_RIGHT || code == KeyEvent.VK_LEFT) {
        handleCursorKey(code);
    } else {
        tool().keyDown(e, code);
    }
    checkDamage();
}

/**
 * Handles cursor keys by moving all the selected figures
 * one grid point in the cursor direction.
 */

```

```

protected void handleCursorKey(int key) {
    int dx = 0, dy = 0;
    int stepX = 1, stepY = 1;
    // should consider Null Object.
    if (fConstrainer != null) {
        stepX = fConstrainer.getStepX();
        stepY = fConstrainer.getStepY();
    }

    switch (key) {
        case KeyEvent.VK_DOWN:
            dy = stepY;
            break;
        case KeyEvent.VK_UP:
            dy = -stepY;
            break;
        case KeyEvent.VK_RIGHT:
            dx = stepX;
            break;
        case KeyEvent.VK_LEFT:
            dx = -stepX;
            break;
    }
    moveSelection(dx, dy);
}

private void moveSelection(int dx, int dy) {
    FigureEnumeration figures = selectionElements();
    while (figures.hasMoreElements())
        figures.nextFigure().moveBy(dx, dy);
    checkDamage();
}

/**
 * Refreshes the drawing if there is some accumulated damage
 */
public synchronized void checkDamage() {
    Enumeration each = drawing().drawingChangeListeners();
    while (each.hasMoreElements()) {
        Object l = each.nextElement();
        if (l instanceof DrawingView) {
            ((DrawingView)l).repairDamage();
        }
    }
}

public void repairDamage() {
    if (fDamage != null) {
        repaint(fDamage.x, fDamage.y, fDamage.width,
fDamage.height);
        fDamage = null;
    }
}

public void drawingInvalidated(DrawingChangeEvent e) {
    Rectangle r = e.getInvalidatedRectangle();
    if (fDamage == null)

```

```

        fDamage = r;
    else
        fDamage.add(r);
    }

    public void drawingRequestUpdate(DrawingChangeEvent e) {
        repairDamage();
    }

    /**
     * Updates the drawing view.
     */
    public void update(Graphics g) {
        paint(g);
    }

    /**
     * Paints the drawing view. The actual drawing is delegated to
     * the current update strategy.
     * @see Painter
     */
    public void paint(Graphics g) {
        fUpdateStrategy.draw(g, this);
    }

    /**
     * Draws the contents of the drawing view.
     * The view has three layers: background, drawing, handles.
     * The layers are drawn in back to front order.
     */
    public void drawAll(Graphics g) {
        boolean isPrinting = g instanceof PrintGraphics;
        drawBackground(g);
        if (fBackgrounds != null && !isPrinting)
            drawPainters(g, fBackgrounds);
        drawDrawing(g);
        if (fForegrounds != null && !isPrinting)
            drawPainters(g, fForegrounds);
        if (!isPrinting)
            drawHandles(g);
    }

    /**
     * Draws the currently active handles.
     */
    public void drawHandles(Graphics g) {
        Enumeration k = selectionHandles();
        while (k.hasMoreElements())
            ((Handle) k.nextElement()).draw(g);
    }

    /**
     * Draws the drawing.
     */
    public void drawDrawing(Graphics g) {
        fDrawing.draw(g);
    }

```

```

/**
 * Draws the background. If a background pattern is set it
 * is used to fill the background. Otherwise the background
 * is filled in the background color.
 */
public void drawBackground(Graphics g) {
    g.setColor(getBackground());
    g.fillRect(0, 0, getBounds().width, getBounds().height);
}

private void drawPainters(Graphics g, Vector v) {
    for (int i = 0; i < v.size(); i++)
        ((Painter)v.elementAt(i)).draw(g, this);
}

/**
 * Adds a background.
 */
public void addBackground(Painter painter) {
    if (fBackgrounds == null)
        fBackgrounds = new Vector(3);
    fBackgrounds.addElement(painter);
    repaint();
}

/**
 * Removes a background.
 */
public void removeBackground(Painter painter) {
    if (fBackgrounds != null)
        fBackgrounds.removeElement(painter);
    repaint();
}

/**
 * Removes a foreground.
 */
public void removeForeground(Painter painter) {
    if (fForegrounds != null)
        fForegrounds.removeElement(painter);
    repaint();
}

/**
 * Adds a foreground.
 */
public void addForeground(Painter painter) {
    if (fForegrounds == null)
        fForegrounds = new Vector(3);
    fForegrounds.addElement(painter);
    repaint();
}

/**
 * Freezes the view by acquiring the drawing lock.
 * @see Drawing#lock
 */
public void freezeView() {

```

```

        drawing().lock();
    }

    /**
     * Unfreezes the view by releasing the drawing lock.
     * @see Drawing#unlock
     */
    public void unfreezeView() {
        drawing().unlock();
    }

    private void readObject(ObjectInputStream s)
        throws ClassNotFoundException, IOException {

        s.defaultReadObject();

        fSelection = new Vector(); // could use lazy initialization
                                   //instead

        if (fDrawing != null)
            fDrawing.addDrawingChangeListener(this);
    }

    private void checkMinimumSize() {
        FigureEnumeration k = drawing().figures();
        Dimension d = new Dimension(0, 0);
        while (k.hasMoreElements()) {
            Rectangle r = k.nextFigure().displayBox();
            d.width = Math.max(d.width, r.x+r.width);
            d.height = Math.max(d.height, r.y+r.height);
        }
        if (fViewSize.height < d.height || fViewSize.width < d.width)
        {
            fViewSize.height = d.height+10;
            fViewSize.width = d.width+10;
            setSize(fViewSize);
        }
    }

    public boolean isFocusTraversable() {
        return true;
    }

    // listener methods we are not interested in
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mouseClicked(MouseEvent e) {}
    public void keyPressed(KeyEvent e) {}
    public void keyReleased(KeyEvent e) {}
}

```

Clase ToggleGridCommand

Un comando para mantener la instantánea de un comportamiento de un grid.

Se encuentra en el fichero ToggleGridCommand.java y su contenido es el siguiente:

```

/*
 * @(#)ToggleGridCommand.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.util.*;
import java.awt.Point;
import CH.ifa.draw.util.Command;
import CH.ifa.draw.framework.*;

/**
 * A command to toggle the snap to grid behavior.
 */
public class ToggleGridCommand extends Command {

    private DrawingView fView;
    private Point fGrid;

    /**
     * Constructs a toggle grid command.
     * @param name the command name
     * @param image the pathname of the image
     * @param grid the grid size. A grid size of 1,1 turns grid
     * snapping off.
     */
    public ToggleGridCommand(String name, DrawingView view, Point
grid) {
        super(name);
        fView = view;
        fGrid = new Point(grid.x, grid.y);
    }

    public void execute() {
        PointConstrainer grid = fView.getConstrainer();
        if (grid != null) {
            fView.setConstrainer(null);
        }
        else {
            fView.setConstrainer(new GridConstrainer(fGrid.x,
fGrid.y));
        }
    }
}

```

Clase ToolButton

Una PaletteButton que esta asociada con una herramienta.

Se encuentra en el fichero ToolButton.java y su contenido es el siguiente:


```

/*
 * @(#)ToolButton.java 5.1
 *
 */

package CH.ifa.draw.standard;

import java.awt.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

/**
 * A PaletteButton that is associated with a tool.
 * @see Tool
 */
public class ToolButton extends PaletteButton {

    private String      fName;
    private Tool        fTool;
    private PaletteIcon fIcon;

    public ToolButton(PaletteListener listener, String iconName,
String name, Tool tool) {
        super(listener);
        // use a Mediatracker to ensure that all the images are
        // initially loaded
        Iconkit kit = Iconkit.instance();
        if (kit == null)
            throw new HJDError("Iconkit instance isn't set");

        Image im[] = new Image[3];
        im[0] = kit.loadImageResource(iconName+"1.gif");
        im[1] = kit.loadImageResource(iconName+"2.gif");
        im[2] = kit.loadImageResource(iconName+"3.gif");

        MediaTracker tracker = new MediaTracker(this);
        for (int i = 0; i < 3; i++) {
            tracker.addImage(im[i], i);
        }
        try {
            tracker.waitForAll();
        } catch (Exception e) { }

        fIcon = new PaletteIcon(new Dimension(24,24), im[0], im[1],
im[2]);
        fTool = tool;
        fName = name;
    }

    public Tool tool() {
        return fTool;
    }

    public String name() {
        return fName;
    }
}

```

```
public Object attributeValue() {
    return tool();
}

public Dimension getMinimumSize() {
    return new Dimension(fIcon.getWidth(), fIcon.getHeight());
}

public Dimension getPreferredSize() {
    return new Dimension(fIcon.getWidth(), fIcon.getHeight());
}

public void paintBackground(Graphics g) { }

public void paintNormal(Graphics g) {
    if (fIcon.normal() != null)
        g.drawImage(fIcon.normal(), 0, 0, this);
}

public void paintPressed(Graphics g) {
    if (fIcon.pressed() != null)
        g.drawImage(fIcon.pressed(), 0, 0, this);
}

public void paintSelected(Graphics g) {
    if (fIcon.selected() != null)
        g.drawImage(fIcon.selected(), 0, 0, this);
}
}
```

Paquete CH.ifa.draw.figures

Un conjunto de figuras junto con sus clases de apoyo asociadas (tools, handles).

Este paquete contiene lo siguiente:

- Interface LineDecoration
- Clase ArrowTip.
- Clase AttributeFigure.
- Clase BorderDecorator.
- Clase BorderTool.
- Clase ChopEllipseConnector.
- Clase ConnectedTextTool.
- Clase ElbowConnection.
- ClaseElbowHandle.
- Clase EllipseFigure.
- Clase FigureAttributes.
- Clase FontSizeHandle.
- Clase GroupCommand.
- Clase GroupFigure.
- Clase ImageFigure.
- Clase InsertImageCommand.
- Clase LineConnection.
- Clase LineFigure.
- Clase NumberTextFigure.
- Clase PolyLineConnector.
- Clase PolyLineFigure.
- Clase PolyLineHandle.
- Clase PolyLineLocator.
- Clase RadiusHandle.
- Clase RectangleFigure.
- Clase RoundRectangleFigure.
- Clase ScribbleTool.
- Clase ShortestDistanceConnector.
- Clase TextFigure.
- Clase TextTool.
- Clase UngroupCommand.

Interface LineDecoration

Decora el punto del principio y del final de una line o poly line figure. LineDecoration es la clase base para las diferentes decoraciones de la linea.

Se encuentra en el fichero LineDecoration.java y su contenido es el siguiente:

```

/*
 * @(#)LineDecoration.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.io.Serializable;

import CH.ifa.draw.util.Storable;

/**
 * Decorate the start or end point of a line or poly line figure.
 * LineDecoration is the base class for the different line
 * decorations.
 * @see PolyLineFigure
 */
public interface LineDecoration
    extends Storable, Cloneable, Serializable {

    /**
     * Draws the decoration in the direction specified by the two
     * points.
     */
    public abstract void draw(Graphics g, int x1, int y1, int x2, int
y2);
}

```

Clase ArrowTip

Una punta de la flecha de la decoración de una linea.

Se encuentra en el fichero ArrowTip.java y su contenido es el siguiente:

```

/*
 * @(#)ArrowTip.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.io.*;
import java.awt.*;

import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.*;

/**

```

```

* An arrow tip line decoration.
* @see PolyLineFigure
*/
public class ArrowTip implements LineDecoration {

    private double fAngle;          // pointiness of arrow
    private double fOuterRadius;
    private double fInnerRadius;

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = -
3459171428373823638L;
    private int arrowTipSerializedDataVersion = 1;

    public ArrowTip() {
        fAngle = 0.40;//0.35;
        fOuterRadius = 8;//15;
        fInnerRadius = 8;//12;
    }

    /**
     * Constructs an arrow tip with the given angle and radius.
     */
    public ArrowTip(double angle, double outerRadius, double
innerRadius) {
        fAngle = angle;
        fOuterRadius = outerRadius;
        fInnerRadius = innerRadius;
    }

    /**
     * Draws the arrow tip in the direction specified by the given two
     * points..
     */
    public void draw(Graphics g, int x1, int y1, int x2, int y2) {
        // TBD: reuse the Polygon object
        Polygon p = outline(x1, y1, x2, y2);
        g.fillPolygon(p.xpoints, p.ypoints, p.npoints);
    }

    /**
     * Calculates the outline of an arrow tip.
     */
    public Polygon outline(int x1, int y1, int x2, int y2) {
        double dir = Math.PI/2 - Math.atan2(x2-x1, y1-y2);
        return outline(x1, y1, dir);
    }

    private Polygon outline(int x, int y, double direction) {
        Polygon shape = new Polygon();

        shape.addPoint(x, y);
        addPointRelative(shape, x, y, fOuterRadius, direction -
fAngle);
        addPointRelative(shape, x, y, fInnerRadius, direction);
    }

```

```

        addPointRelative(shape, x, y, fOuterRadius, direction +
fAngle);
        shape.addPoint(x,y); // Closing the polygon (TEG 97-04-23)
        return shape;
    }

    private void addPointRelative(Polygon shape, int x, int y, double
radius, double angle) {
        shape.addPoint(
            x + (int) (radius * Math.cos(angle)),
            y - (int) (radius * Math.sin(angle)));
    }

    /**
     * Stores the arrow tip to a StorableOutput.
     */
    public void write(StorableOutput dw) {
    }

    /**
     * Reads the arrow tip from a StorableInput.
     */
    public void read(StorableInput dr) throws IOException {
    }
}

```

Clase AttributeFigure

Una figura puede tener un conjunto no limitado de atributos. Los atributos son almacenados en un diccionario implementado por FigureAttributes.

Se encuentra en el fichero AttributeFigure.java y su contenido es el siguiente:

```

/*
 * @(#)AttributeFigure.java 5.1
 *
 */

package CH.ifa.draw.figures;

import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;

import java.awt.*;
import java.util.*;
import java.io.*;

/**
 * A figure that can keep track of an open ended set of attributes.
 * The attributes are stored in a dictionary implemented by
 * FigureAttributes.
 */

```

```

* @see Figure
* @see Handle
* @see FigureAttributes
*/
public abstract class AttributeFigure extends AbstractFigure {

    /**
     * The attributes of a figure. Each figure can have
     * an open ended set of attributes. Attributes are
     * identified by name.
     * @see #getAttribute
     * @see #setAttribute
     */
    private FigureAttributes fAttributes;

    /**
     * The default attributes associated with a figure.
     * If a figure doesn't have an attribute set, a default
     * value from this shared attribute set is returned.
     * @see #getAttribute
     * @see #setAttribute
     */
    private static FigureAttributes fgDefaultAttributes = null;

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = -10857585979273442L;
    private int attributeFigureSerializedDataVersion = 1;

    protected AttributeFigure() { }

    /**
     * Draws the figure in the given graphics. Draw is a template
     * method calling drawBackground followed by drawFrame.
     */
    public void draw(Graphics g) {
        Color fill = getFillColor();
        if (!ColorMap.isTransparent(fill)) {
            g.setColor(fill);
            drawBackground(g);
        }
        Color frame = getFrameColor();
        if (!ColorMap.isTransparent(frame)) {
            g.setColor(frame);
            drawFrame(g);
        }
    }

    /**
     * Draws the background of the figure.
     * @see #draw
     */
    protected void drawBackground(Graphics g) {
    }

    /**
     * Draws the frame of the figure.

```

```

    * @see #draw
    */
protected void drawFrame(Graphics g) {
}

/**
 * Gets the fill color of a figure. This is a convenience
 * method.
 * @see getAttribute
 */
public Color getFillColor() {
    return (Color) getAttribute("FillColor");
}

/**
 * Gets the frame color of a figure. This is a convenience
 * method.
 * @see getAttribute
 */
public Color getFrameColor() {
    return (Color) getAttribute("FrameColor");
}

//---- figure attributes -----

private static void initializeAttributes() {
    fgDefaultAttributes = new FigureAttributes();
    fgDefaultAttributes.set("FrameColor", Color.black);
    fgDefaultAttributes.set("FillColor", new Color(0x70DB93));
    fgDefaultAttributes.set("TextColor", Color.black);
    fgDefaultAttributes.set("ArrowMode", new Integer(0));
    fgDefaultAttributes.set("FontName", "Helvetica");
    fgDefaultAttributes.set("FontSize", new Integer(12));
    fgDefaultAttributes.set("FontStyle", new
Integer(Font.PLAIN));
}

/**
 * Gets a the default value for a named attribute
 * @see getAttribute
 */
public static Object getDefaultAttribute(String name) {
    if (fgDefaultAttributes == null)
        initializeAttributes();
    return fgDefaultAttributes.get(name);
}

/**
 * Returns the named attribute or null if a
 * a figure doesn't have an attribute.
 * All figures support the attribute names
 * FillColor and FrameColor
 */
public Object getAttribute(String name) {
    if (fAttributes != null) {
        if (fAttributes.hasDefined(name))
            return fAttributes.get(name);
    }
}

```



```

        return getDefaultAttribute(name);
    }

    /**
     * Sets the named attribute to the new value
     */
    public void setAttribute(String name, Object value) {
        if (fAttributes == null)
            fAttributes = new FigureAttributes();
        fAttributes.set(name, value);
        changed();
    }

    /**
     * Stores the Figure to a StorableOutput.
     */
    public void write(StorableOutput dw) {
        super.write(dw);
        if (fAttributes == null)
            dw.writeString("no_attributes");
        else {
            dw.writeString("attributes");
            fAttributes.write(dw);
        }
    }

    /**
     * Reads the Figure from a StorableInput.
     */
    public void read(StorableInput dr) throws IOException {
        super.read(dr);
        String s = dr.readString();
        if (s.toLowerCase().equals("attributes")) {
            fAttributes = new FigureAttributes();
            fAttributes.read(dr);
        }
    }
}

```

Clase BorderDecorator

BorderDecorator decora una Figure arbitraria con un borde.

Se encuentra en el fichero BorderDecorator.java y su contenido es el siguiente:

```

/*
 * @(#)BorderDecorator.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.util.*;

```

```
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;

/**
 * BorderDecorator decorates an arbitrary Figure with
 * a border.
 */
public class BorderDecorator extends DecoratorFigure {

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = 1205601808259084917L;
    private int borderDecoratorSerializedDataVersion = 1;

    public BorderDecorator() { }
    public BorderDecorator(Figure figure) {
        super(figure);
    }

    private Point border() {
        return new Point(3,3);
    }

    /**
     * Draws a the figure and decorates it with a border.
     */
    public void draw(Graphics g) {
        Rectangle r = displayBox();
        super.draw(g);
        g.setColor(Color.white);
        g.drawLine(r.x, r.y, r.x, r.y + r.height);
        g.drawLine(r.x, r.y, r.x + r.width, r.y);
        g.setColor(Color.gray);
        g.drawLine(r.x + r.width, r.y, r.x + r.width, r.y + r.height);
        g.drawLine(r.x , r.y + r.height, r.x + r.width, r.y +
r.height);
    }

    /**
     * Gets the displaybox including the border.
     */
    public Rectangle displayBox() {
        Rectangle r = fComponent.displayBox();
        r.grow(border().x, border().y);
        return r;
    }

    /**
     * Invalidates the figure extended by its border.
     */
    public void figureInvalidated(FigureChangeEvent e) {
        Rectangle rect = e.getInvalidatedRectangle();
        rect.grow(border().x, border().y);
        super.figureInvalidated(new FigureChangeEvent(e.getFigure(),
rect));
    }
}
```

```

    public Insets connectionInsets() {
        Insets i = super.connectionInsets();
        i.top -= 3;
        i.bottom -= 3;
        i.left -= 3;
        i.right -= 3;
        return i;
    }
}

```

Clase BorderTool

BorderTool decora la figura seleccionada con un BorderDecorator.

Se encuentra en el fichero BorderTool.java y su contenido es el siguiente:

```

/*
 * @(#)BorderTool.java 5.1
 *
 */

package CH.ifa.draw.figures;

import CH.ifa.draw.standard.*;
import CH.ifa.draw.framework.*;

/**
 * BorderTool decorates the clicked figure with a BorderDecorator.
 *
 * @see BorderDecorator
 */
public class BorderTool extends ActionTool {

    public BorderTool(DrawingView view) {
        super(view);
    }

    /**
     * Decorates the clicked figure with a border.
     */
    public void action(Figure figure) {
        drawing().replace(figure, new BorderDecorator(figure));
    }
}

```

Clase ChopEllipseConnector

Un ChopEllipseConnector localiza un punto de conexión para la conexión de una elipse definida por la caja de representación de la figura.

Se encuentra en el fichero ChopEllipseConnector.java y su contenido es el siguiente:

```

/*
 * @(#)ChopEllipseConnector.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.Geom;

/**
 * A ChopEllipseConnector locates a connection point by
 * chopping the connection at the ellipse defined by the
 * figure's display box.
 */
public class ChopEllipseConnector extends ChopBoxConnector {

    /*
     * Serialization support.
     */
    private static final long serialVersionUID = -
3165091511154766610L;

    public ChopEllipseConnector() {
    }

    public ChopEllipseConnector(Figure owner) {
        super(owner);
    }

    protected Point chop(Figure target, Point from) {
        Rectangle r = target.displayBox();
        double angle = Geom.pointToAngle(r, from);
        return Geom.ovalAngleToPoint(r, angle);
    }
}

```

Clase ConnectedTextTool

Herramienta para crear una nueva o editar las figuras de texto existentes. Una nueva figura de texto se conecta con la figura seleccionada.

Se encuentra en el fichero ConnectedTextTool.java y su contenido es el siguiente:

```

/*
 * @(#)ConnectedTextTool.java 5.1
 *
 */

```

```

package CH.ifa.draw.figures;

import java.awt.*;
import java.awt.event.MouseEvent;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;

/**
 * Tool to create new or edit existing text figures.
 * A new text figure is connected with the clicked figure.
 *
 * @see TextHolder
 */
public class ConnectedTextTool extends TextTool {

    boolean    fConnected = false;

    public ConnectedTextTool(DrawingView view, Figure prototype) {
        super(view, prototype);
    }

    /**
     * If the pressed figure is a TextHolder it can be edited
     * otherwise a new text figure is created.
     */
    public void mouseDown(MouseEvent e, int x, int y) {
        super.mouseDown(e, x, y);

        Figure pressedFigure = drawing().findFigureInside(x, y);

        TextHolder textHolder = (TextHolder)createdFigure();
        if (!fConnected && pressedFigure != null &&
            textHolder != null && pressedFigure !=
textHolder) {
            textHolder.connect(pressedFigure);
            fConnected = true;
        }
    }

    /**
     * If the pressed figure is a TextHolder it can be edited
     * otherwise a new text figure is created.
     */
    public void activate() {
        fConnected = false;
    }
}

```

Clase ElbowConnection

Una LineConnection que contiene una conexión de líneas ortogonales.

Se encuentra en el fichero ElbowConnection.java y su contenido es el siguiente:

```

/*
 * @(#)ElbowConnection.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.util.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.*;

/**
 * A LineConnection that constrains a connection to
 * orthogonal lines.
 */
public class ElbowConnection extends LineConnection {

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = 2193968743082078559L;
    private int elbowConnectionSerializedDataVersion = 1;

    public ElbowConnection() {
        super();
    }

    public void updateConnection() {
        super.updateConnection();
        updatePoints();
    }

    public void layoutConnection() {
    }

    /**
     * Gets the handles of the figure.
     */
    public Vector handles() {
        Vector handles = new Vector(fPoints.size()*2);
        handles.addElement(new ChangeConnectionStartHandle(this));
        for (int i = 1; i < fPoints.size()-1; i++)
            handles.addElement(new NullHandle(this, locator(i)));
        handles.addElement(new ChangeConnectionEndHandle(this));
        for (int i = 0; i < fPoints.size()-1; i++)
            handles.addElement(new ElbowHandle(this, i));
        return handles;
    }

    public Locator connectedTextLocator(Figure f) {
        return new ElbowTextLocator();
    }

    protected void updatePoints() {
        willChange();
    }

```

```

        Point start = startPoint();
        Point end = endPoint();
        fPoints.removeAllElements();
        fPoints.addElement(start);

        if (start.x == end.x || start.y == end.y) {
            fPoints.addElement(end);
        }
        else {
            Rectangle r1 = start().owner().displayBox();
            Rectangle r2 = end().owner().displayBox();

            int x1, y1, x2, y2;
            int dir = Geom.direction(r1.x + r1.width/2, r1.y +
r1.height/2,
                                r2.x + r2.width/2, r2.y + r2.height/2);
            if (dir == Geom.NORTH || dir == Geom.SOUTH) {
                fPoints.addElement(new Point(start.x, (start.y +
end.y)/2));
                fPoints.addElement(new Point(end.x, (start.y +
end.y)/2));
            }
            else {
                fPoints.addElement(new Point((start.x + end.x)/2,
start.y));
                fPoints.addElement(new Point((start.x + end.x)/2,
end.y));
            }
            fPoints.addElement(end);
        }
        changed();
    }
}

class ElbowTextLocator extends AbstractLocator {
    public Point locate(Figure owner) {
        Point p = owner.center();
        Rectangle r = owner.displayBox();
        return new Point(p.x, p.y-10); // hack
    }
}

```

Clase ElbowHandle

Un Handle para mover una ElbowConnection a la izquierda y derecha o arriba y abajo.

Se encuentra en el fichero ElbowHandle.java y su contenido es el siguiente:

```

/*
 * @(#)ElbowHandle.java 5.1
 *
 */

package CH.ifa.draw.figures;

```

```

import java.awt.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.Geom;

/**
 * A Handle to move an ElbowConnection left/right or up/down.
 */
public class ElbowHandle extends AbstractHandle {

    private int fSegment;
    private int fLastX, fLastY;      // previous mouse position

    public ElbowHandle(LineConnection owner, int segment) {
        super(owner);
        fSegment = segment;
    }

    public void invokeStart(int x, int y, DrawingView view) {
        fLastX = x;
        fLastY = y;
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        LineConnection line = ownerConnection();
        Point p1 = line.pointAt(fSegment);
        Point p2 = line.pointAt(fSegment+1);
        int ddx = x - fLastX;
        int ddy = y - fLastY;

        Point np1;
        Point np2;
        if (isVertical(p1, p2)) {
            int cx = constrainX(p1.x + ddx);
            np1 = new Point(cx, p1.y);
            np2 = new Point(cx, p2.y);
        } else {
            int cy = constrainY(p1.y + ddy);
            np1 = new Point(p1.x, cy);
            np2 = new Point(p2.x, cy);
        }
        line.setPointAt(np1, fSegment);
        line.setPointAt(np2, fSegment+1);
        fLastX = x;
        fLastY = y;
    }

    private boolean isVertical(Point p1, Point p2) {
        return p1.x == p2.x;
    }

    public Point locate() {
        LineConnection line = ownerConnection();
        int segment = Math.min(fSegment, line.pointCount()-2);
        Point p1 = line.pointAt(segment);
        Point p2 = line.pointAt(segment+1);
    }
}

```



```

        return new Point((p1.x + p2.x)/2, (p1.y + p2.y)/2);
    }

    public void draw(Graphics g) {
        Rectangle r = displayBox();

        g.setColor(Color.yellow);
        g.fillOval(r.x, r.y, r.width, r.height);

        g.setColor(Color.black);
        g.drawOval(r.x, r.y, r.width, r.height);
    }

    private int constrainX(int x) {
        LineConnection line = ownerConnection();
        Figure startFigure = line.start().owner();
        Figure endFigure = line.end().owner();
        Rectangle start = startFigure.displayBox();
        Rectangle end = endFigure.displayBox();
        Insets i1 = startFigure.connectionInsets();
        Insets i2 = endFigure.connectionInsets();

        int r1x, r1width, r2x, r2width;
        r1x = start.x + i1.left;
        r1width = start.width - i1.left - i1.right-1;

        r2x = end.x + i2.left;
        r2width = end.width - i2.left - i2.right-1;

        if (fSegment == 0)
            x = Geom.range(r1x, r1x + r1width, x);
        if (fSegment == line.pointCount()-2)
            x = Geom.range(r2x, r2x + r2width, x);
        return x;
    }

    private int constrainY(int y) {
        LineConnection line = ownerConnection();
        Figure startFigure = line.start().owner();
        Figure endFigure = line.end().owner();
        Rectangle start = startFigure.displayBox();
        Rectangle end = endFigure.displayBox();
        Insets i1 = startFigure.connectionInsets();
        Insets i2 = endFigure.connectionInsets();

        int r1y, r1height, r2y, r2height;
        r1y = start.y + i1.top;
        r1height = start.height - i1.top - i1.bottom-1;
        r2y = end.y + i2.top;
        r2height = end.height - i2.top - i2.bottom-1;

        if (fSegment == 0)
            y = Geom.range(r1y, r1y + r1height, y);
        if (fSegment == line.pointCount()-2)
            y = Geom.range(r2y, r2y + r2height, y);
        return y;
    }

```

```

        private LineConnection ownerConnection() {
            return (LineConnection)owner();
        }
    }
}

```

Clase EllipseFigure

Una figura ellipse.

Se encuentra en el fichero EllipseFigure.java y su contenido es el siguiente:

```

/*
 * @(#)EllipseFigure.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.io.IOException;
import java.util.Vector;
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;

/**
 * An ellipse figure.
 */
public class EllipseFigure extends AttributeFigure {

    private Rectangle    fDisplayBox;

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = -
6856203289355118951L;
    private int ellipseFigureSerializedDataVersion = 1;

    public EllipseFigure() {
        this(new Point(0,0), new Point(0,0));
    }

    public EllipseFigure(Point origin, Point corner) {
        basicDisplayBox(origin,corner);
    }

    public Vector handles() {
        Vector handles = new Vector();
        BoxHandleKit.addHandles(this, handles);
        return handles;
    }
}

```

```

public void basicDisplayBox(Point origin, Point corner) {
    fDisplayBox = new Rectangle(origin);
    fDisplayBox.add(corner);
}

public Rectangle displayBox() {
    return new Rectangle(
        fDisplayBox.x,
        fDisplayBox.y,
        fDisplayBox.width,
        fDisplayBox.height);
}

protected void basicMoveBy(int x, int y) {
    fDisplayBox.translate(x,y);
}

public void drawBackground(Graphics g) {
    Rectangle r = displayBox();
    g.fillOval(r.x, r.y, r.width, r.height);
}

public void drawFrame(Graphics g) {
    Rectangle r = displayBox();
    g.drawOval(r.x, r.y, r.width-1, r.height-1);
}

public Insets connectionInsets() {
    Rectangle r = fDisplayBox;
    int cx = r.width/2;
    int cy = r.height/2;
    return new Insets(cy, cx, cy, cx);
}

public Connector connectorAt(int x, int y) {
    return new ChopEllipseConnector(this);
}

public void write(StorableOutput dw) {
    super.write(dw);
    dw.writeInt(fDisplayBox.x);
    dw.writeInt(fDisplayBox.y);
    dw.writeInt(fDisplayBox.width);
    dw.writeInt(fDisplayBox.height);
}

public void read(StorableInput dr) throws IOException {
    super.read(dr);
    fDisplayBox = new Rectangle(
        dr.readInt(),
        dr.readInt(),
        dr.readInt(),
        dr.readInt());
}
}

```

Clase FigureAttributes

Un contenedor para los atributos de una figura. Los atributos son guardados como parejas clave/valor.

Se encuentra en el fichero FigureAttributes.java y su contenido es el siguiente:

```
/*
 * @(#)FigureAttributes.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.util.*;
import java.awt.Color;
import java.io.IOException;
import java.io.Serializable;

import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;

/**
 * A container for a figure's attributes. The attributes are stored
 * as key/value pairs.
 *
 * @see Figure
 */
public class FigureAttributes
    extends Object
    implements Cloneable, Serializable {

    private Hashtable fMap;

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = -
6886355144423666716L;
    private int figureAttributesSerializedDataVersion = 1;

    /**
     * Constructs the FigureAttributes.
     */
    public FigureAttributes() {
        fMap = new Hashtable();
    }

    /**
     * Gets the attribute with the given name.
     * @returns attribute or null if the key is not defined
     */
    public Object get(String name) {
        return fMap.get(name);
    }
}
```

```

    }

    /**
     * Sets the attribute with the given name and
     * overwrites its previous value.
     */
    public void set(String name, Object value) {
        fMap.put(name, value);
    }

    /**
     * Tests if an attribute is defined.
     */
    public boolean hasDefined(String name) {
        return fMap.containsKey(name);
    }

    /**
     * Clones the attributes.
     */
    public Object clone() {
        try {
            FigureAttributes a = (FigureAttributes) super.clone();
            a.fMap = (Hashtable) fMap.clone();
            return a;
        } catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
    }

    /**
     * Reads the attributes from a StorableInput.
     * FigureAttributes store the following types directly:
     * Color, Boolean, String, Int. Other attribute types
     * have to implement the Storable interface or they
     * have to be wrapped by an object that implements Storable.
     * @see Storable
     * @see #write
     */
    public void read(StorableInput dr) throws IOException {
        String s = dr.readString();
        if (!s.toLowerCase().equals("attributes"))
            throw new IOException("Attributes expected");

        fMap = new Hashtable();
        int size = dr.readInt();
        for (int i=0; i<size; i++) {
            String key = dr.readString();
            String valtype = dr.readString();
            Object val = null;
            if (valtype.equals("Color"))
                val = new Color(dr.readInt(), dr.readInt(),
dr.readInt());
            else if (valtype.equals("Boolean"))
                val = new Boolean(dr.readString());
            else if (valtype.equals("String"))
                val = dr.readString();
            else if (valtype.equals("Int"))

```

```

        val = new Integer(dr.readInt());
    else if (valtype.equals("Storable"))
        val = dr.readStorable();
    else if (valtype.equals("UNKNOWN"))
        continue;

    fMap.put(key, val);
}
}

/**
 * Writes the attributes to a StorableInput.
 * FigureAttributes store the following types directly:
 * Color, Boolean, String, Int. Other attribute types
 * have to implement the Storable interface or they
 * have to be wrapped by an object that implements Storable.
 * @see Storable
 * @see #write
 */
public void write(StorableOutput dw) {
    dw.writeString("attributes");

    dw.writeInt(fMap.size()); // number of attributes
    Enumeration k = fMap.keys();
    while (k.hasMoreElements()) {
        String s = (String) k.nextElement();
        dw.writeString(s);
        Object v = fMap.get(s);
        if (v instanceof String) {
            dw.writeString("String");
            dw.writeString((String) v);
        } else if (v instanceof Color) {
            dw.writeString("Color");
            dw.writeInt(((Color)v).getRed());
            dw.writeInt(((Color)v).getGreen());
            dw.writeInt(((Color)v).getBlue());
        } else if (v instanceof Boolean) {
            dw.writeString("Boolean");
            if (((Boolean)v).booleanValue())
                dw.writeString("TRUE");
            else
                dw.writeString("FALSE");
        } else if (v instanceof Integer) {
            dw.writeString("Int");
            dw.writeInt(((Integer)v).intValue());
        } else if (v instanceof Storable) {
            dw.writeString("Storable");
            dw.writeStorable((Storable)v);
        } else {
            System.out.println(v);
            dw.writeString("UNKNOWN");
        }
    }
}
}
}

```

Clase FontSizeHandle

Un Handle para cambiar el tamaño de la fuente por manipulación directa.

Se encuentra en el fichero FontSizeHandle.java y su contenido es el siguiente:

```
/*
 * @(#)FontSizeHandle.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;

/**
 * A Handle to change the font size by direct manipulation.
 */
public class FontSizeHandle extends LocatorHandle {

    private Font    fFont;
    private int     fSize;

    public FontSizeHandle(Figure owner, Locator l) {
        super(owner, l);
    }

    public void invokeStart(int x, int y, DrawingView view) {
        TextFigure textOwner = (TextFigure) owner();
        fFont = textOwner.getFont();
        fSize = fFont.getSize();
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        TextFigure textOwner = (TextFigure) owner();
        int newSize = fSize + y-anchorY;
        textOwner.setFont(new Font(fFont.getName(), fFont.getStyle(),
newSize) );
    }

    public void draw(Graphics g) {
        Rectangle r = displayBox();

        g.setColor(Color.yellow);
        g.fillOval(r.x, r.y, r.width, r.height);

        g.setColor(Color.black);
        g.drawOval(r.x, r.y, r.width, r.height);
    }
}
```

Clase GroupCommand

Un comando para agrupar la selección en un GroupFigure.

Se encuentra en el fichero GroupCommand.java y su contenido es el siguiente:

```
/*
 * @(#)GroupCommand.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.util.*;
import CH.ifa.draw.util.Command;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;

/**
 * Command to group the selection into a GroupFigure.
 *
 * @see GroupFigure
 */
public class GroupCommand extends Command {

    private DrawingView fView;

    /**
     * Constructs a group command.
     * @param name the command name
     * @param view the target view
     */
    public GroupCommand(String name, DrawingView view) {
        super(name);
        fView = view;
    }

    public void execute() {
        Vector selected = fView.selectionZOrdered();
        Drawing drawing = fView.drawing();
        if (selected.size() > 0) {
            fView.clearSelection();
            drawing.orphanAll(selected);

            GroupFigure group = new GroupFigure();
            group.addAll(selected);
            fView.addToSelection(drawing.add(group));
        }
        fView.checkDamage();
    }

    public boolean isExecutable() {
        return fView.selectionCount() > 0;
    }
}
```



```
}
```

Clase GroupFigure

Una Figure que agrupa una colección de figuras.

Se encuentra en el fichero GroupFigure.java y su contenido es el siguiente:

```
/*
 * @(#)GroupFigure.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.util.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;

/**
 * A Figure that groups a collection of figures.
 */
public class GroupFigure extends CompositeFigure {

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = 8311226373023297933L;
    private int groupFigureSerializedDataVersion = 1;

    /**
     * GroupFigures cannot be connected
     */
    public boolean canConnect() {
        return false;
    }

    /**
     * Gets the display box. The display box is defined as the union
     * of the contained figures.
     */
    public Rectangle displayBox() {
        FigureEnumeration k = figures();
        Rectangle r = k.nextFigure().displayBox();

        while (k.hasMoreElements())
            r.add(k.nextFigure().displayBox());
        return r;
    }

    public void basicDisplayBox(Point origin, Point corner) {
        // do nothing
        // we could transform all components proportionally
    }
}
```

```

    }

    public FigureEnumeration decompose() {
        return new FigureEnumerator(fFigures);
    }

    /**
     * Gets the handles for the GroupFigure.
     */
    public Vector handles() {
        Vector handles = new Vector();
        handles.addElement(new GroupHandle(this,
RelativeLocator.northWest()));
        handles.addElement(new GroupHandle(this,
RelativeLocator.northEast()));
        handles.addElement(new GroupHandle(this,
RelativeLocator.southWest()));
        handles.addElement(new GroupHandle(this,
RelativeLocator.southEast()));
        return handles;
    }

    /**
     * Sets the attribute of all the contained figures.
     */
    public void setAttribute(String name, Object value) {
        super.setAttribute(name, value);
        FigureEnumeration k = figures();
        while (k.hasMoreElements())
            k.nextFigure().setAttribute(name, value);
    }
}

```

Clase GroupHandle

Un NullHandle para GroupFigure.

Se encuentra en el fichero GroupHandle.java y su contenido es el siguiente:

```

/*
 * @(#)GroupHandle.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.NullHandle;

/**
 * A Handle for a GroupFigure.
 */
final class GroupHandle extends NullHandle {

```

```

public GroupHandle(Figure owner, Locator locator) {
    super(owner, locator);
}

/**
 * Draws the Group handle.
 */
public void draw(Graphics g) {
    Rectangle r = displayBox();

    g.setColor(Color.black);
    g.drawRect(r.x, r.y, r.width, r.height);
    r.grow(-1,-1);
    g.setColor(Color.white);
    g.drawRect(r.x, r.y, r.width, r.height);
}
}

```

Clase ImageFigure

Una Figure que muestra una Image. Las Images muestran una figura imagen son compartidas utilizando el Iconkit.

Se encuentra en el fichero ImageFigure.java y su contenido es el siguiente:

```

/*
 * @(#)ImageFigure.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.io.*;
import java.util.Vector;
import java.awt.image.ImageObserver;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.*;

/**
 * A Figure that shows an Image.
 * Images shown by an image figure are shared by using the Iconkit.
 * @see Iconkit
 */
public class ImageFigure
    extends AttributeFigure implements ImageObserver {

    private String fFileName;
    private transient Image fImage;
    private Rectangle fDisplayBox;

    /**
     * Serialization support.

```

```

    */
    private static final long serialVersionUID = 148012030121282439L;
    private int imageFigureSerializedDataVersion = 1;

    public ImageFigure() {
        fFileName = null;
        fImage = null;
        fDisplayBox = null;
    }

    public ImageFigure(Image image, String fileName, Point origin) {
        fFileName = fileName;
        fImage = image;
        fDisplayBox = new Rectangle(origin.x, origin.y, 0, 0);
        fDisplayBox.width = fImage.getWidth(this);
        fDisplayBox.height = fImage.getHeight(this);
    }

    public void basicDisplayBox(Point origin, Point corner) {
        fDisplayBox = new Rectangle(origin);
        fDisplayBox.add(corner);
    }

    public Vector handles() {
        Vector handles = new Vector();
        BoxHandleKit.addHandles(this, handles);
        return handles;
    }

    public Rectangle displayBox() {
        return new Rectangle(
            fDisplayBox.x,
            fDisplayBox.y,
            fDisplayBox.width,
            fDisplayBox.height);
    }

    protected void basicMoveBy(int x, int y) {
        fDisplayBox.translate(x,y);
    }

    public void draw(Graphics g) {
        if (fImage == null)
            fImage = Iconkit.instance().getImage(fFileName);
        if (fImage != null)
            g.drawImage(fImage, fDisplayBox.x, fDisplayBox.y,
fDisplayBox.width, fDisplayBox.height, this);
        else
            drawGhost(g);
    }

    private void drawGhost(Graphics g) {
        g.setColor(Color.gray);
        g.fillRect(fDisplayBox.x, fDisplayBox.y, fDisplayBox.width,
fDisplayBox.height);
    }

    /**

```

```

    * Handles asynchronous image updates.
    */
    public boolean imageUpdate(Image img, int flags, int x, int y, int
w, int h) {
        if ((flags & (FRAMEBITS|ALLBITS)) != 0) {
            invalidate();
            if (listener() != null)
                listener().figureRequestUpdate(new
FigureChangeEvent(this));
        }
        return (flags & (ALLBITS|ABORT)) == 0;
    }

    /**
    * Writes the ImageFigure to a StorableOutput. Only a reference to
    * the image, that is its pathname is saved.
    */
    public void write(StorableOutput dw) {
        super.write(dw);
        dw.writeInt(fDisplayBox.x);
        dw.writeInt(fDisplayBox.y);
        dw.writeInt(fDisplayBox.width);
        dw.writeInt(fDisplayBox.height);
        dw.writeString(fFileName);
    }

    /**
    * Reads the ImageFigure from a StorableInput. It registers the
    * referenced figure to be loaded from the Iconkit.
    * @see Iconkit#registerImage
    */
    public void read(StorableInput dr) throws IOException {
        super.read(dr);
        fDisplayBox = new Rectangle(
            dr.readInt(),
            dr.readInt(),
            dr.readInt(),
            dr.readInt());
        fFileName = dr.readString();
        Iconkit.instance().registerImage(fFileName);
    }

    private void readObject(ObjectInputStream s)
        throws ClassNotFoundException, IOException {

        s.defaultReadObject();
        Iconkit.instance().registerImage(fFileName);
        fImage = null;
    }
}

```

Clase InsertImageCommand

Comando para insertar una imagen.

Se encuentra en el fichero InsertImageCommand.java y su contenido es el siguiente:

```

/*
 * @(#)InsertImageCommand.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.util.*;
import java.awt.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;

/**
 * Command to insert a named image.
 */
public class InsertImageCommand extends Command {

    private DrawingView fView;
    private String fImage;

    /**
     * Constructs an insert image command.
     * @param name the command name
     * @param image the pathname of the image
     * @param view the target view
     */
    public InsertImageCommand(String name, String image, DrawingView
view) {
        super(name);
        fImage = image;
        fView = view;
    }

    public void execute() {
        // ugly cast to component, but AWT wants and Component instead
        //of an ImageObserver...
        Image image =
Iconkit.instance().registerAndLoadImage((Component)fView, fImage);
        ImageFigure figure = new ImageFigure(image, fImage,
fView.lastClick());
        fView.add(figure);
        fView.clearSelection();
        fView.addToSelection(figure);
        fView.checkDamage();
    }
}

```

Clase LineConnection

Una LineConnection es una implementación estándar del interface ConnectionFigure. El interface es implementado por PolyLineFigure.

Se encuentra en el fichero LineConnection.java y su contenido es el siguiente:

```

/*
 * @(#)LineConnection.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.util.*;
import java.io.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.*;

/**
 * A LineConnection is a standard implementation of the
 * ConnectionFigure interface. The interface is implemented with
 * PolyLineFigure.
 * @see ConnectionFigure
 */
public class LineConnection extends PolyLineFigure implements
ConnectionFigure {

    protected Connector    fStart = null;
    protected Connector    fEnd = null;

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = 6883731614578414801L;
    private int lineConnectionSerializedDataVersion = 1;

    /**
     * Constructs a LineConnection. A connection figure has
     * an arrow decoration at the start and end.
     */
    public LineConnection() {
        super(4);
        setStartDecoration(new ArrowTip());
        setEndDecoration(new ArrowTip());
    }

    /**
     * Tests whether a figure can be a connection target.
     * ConnectionFigures cannot be connected and return false.
     */
    public boolean canConnect() {
        return false;
    }

    /**
     * Ensures that a connection is updated if the connection
     * was moved.
     */
}

```

```

protected void basicMoveBy(int dx, int dy) {
    // don't move the start and end point since they are connected
    for (int i = 1; i < fPoints.size()-1; i++)
        ((Point) fPoints.elementAt(i)).translate(dx, dy);

    updateConnection(); // make sure that we are still connected
}

/**
 * Sets the start figure of the connection.
 */
public void connectStart(Connector start) {
    fStart = start;
    startFigure().addFigureChangeListener(this);
}

/**
 * Sets the end figure of the connection.
 */
public void connectEnd(Connector end) {
    fEnd = end;
    endFigure().addFigureChangeListener(this);
    handleConnect(startFigure(), endFigure());
}

/**
 * Disconnects the start figure.
 */
public void disconnectStart() {
    startFigure().removeFigureChangeListener(this);
    fStart = null;
}

/**
 * Disconnects the end figure.
 */
public void disconnectEnd() {
    handleDisconnect(startFigure(), endFigure());
    endFigure().removeFigureChangeListener(this);
    fEnd = null;
}

/**
 * Tests whether a connection connects the same figures
 * as another ConnectionFigure.
 */
public boolean connectsSame(ConnectionFigure other) {
    return other.start() == start() && other.end() == end();
}

/**
 * Handles the disconnection of a connection.
 * Override this method to handle this event.
 */
protected void handleDisconnect(Figure start, Figure end) {}

/**
 * Handles the connection of a connection.

```



```

    * Override this method to handle this event.
    */
    protected void handleConnect(Figure start, Figure end) {}

    /**
     * Gets the start figure of the connection.
     */
    public Figure startFigure() {
        if (start() != null)
            return start().owner();
        return null;
    }

    /**
     * Gets the end figure of the connection.
     */
    public Figure endFigure() {
        if (end() != null)
            return end().owner();
        return null;
    }

    /**
     * Gets the start figure of the connection.
     */
    public Connector start() {
        return fStart;
    }

    /**
     * Gets the end figure of the connection.
     */
    public Connector end() {
        return fEnd;
    }

    /**
     * Tests whether two figures can be connected.
     */
    public boolean canConnect(Figure start, Figure end) {
        return true;
    }

    /**
     * Sets the start point.
     */
    public void startPoint(int x, int y) {
        willChange();
        if (fPoints.size() == 0)
            fPoints.addElement(new Point(x, y));
        else
            fPoints.setElementAt(new Point(x, y), 0);
        changed();
    }

    /**
     * Sets the end point.
     */

```

```

public void endPoint(int x, int y) {
    willChange();
    if (fPoints.size() < 2)
        fPoints.addElement(new Point(x, y));
    else
        fPoints.setElementAt(new Point(x, y), fPoints.size()-1);
    changed();
}

/**
 * Gets the start point.
 */
public Point startPoint(){
    Point p = (Point)fPoints.firstElement();
    return new Point(p.x, p.y);
}

/**
 * Gets the end point.
 */
public Point endPoint() {
    Point p = (Point)fPoints.lastElement();
    return new Point(p.x, p.y);
}

/**
 * Gets the handles of the figure. It returns the normal
 * PolyLineHandles but adds ChangeConnectionHandles at the
 * start and end.
 */
public Vector handles() {
    Vector handles = new Vector(fPoints.size());
    handles.addElement(new ChangeConnectionStartHandle(this));
    for (int i = 1; i < fPoints.size()-1; i++)
        handles.addElement(new PolyLineHandle(this, locator(i),
i));
    handles.addElement(new ChangeConnectionEndHandle(this));
    return handles;
}

/**
 * Sets the point and updates the connection.
 */
public void setPointAt(Point p, int i) {
    super.setPointAt(p, i);
    layoutConnection();
}

/**
 * Inserts the point and updates the connection.
 */
public void insertPointAt(Point p, int i) {
    super.insertPointAt(p, i);
    layoutConnection();
}

/**
 * Removes the point and updates the connection.

```

```

    */
    public void removePointAt(int i) {
        super.removePointAt(i);
        layoutConnection();
    }

    /**
     * Updates the connection.
     */
    public void updateConnection() {
        if (fStart != null) {
            Point start = fStart.findStart(this);
            startPoint(start.x, start.y);
        }
        if (fEnd != null) {
            Point end = fEnd.findEnd(this);
            endPoint(end.x, end.y);
        }
    }

    /**
     * Lays out the connection. This is called when the connection
     * itself changes. By default the connection is recalculated
     */
    public void layoutConnection() {
        updateConnection();
    }

    public void figureChanged(FigureChangeEvent e) {
        updateConnection();
    }

    public void figureRemoved(FigureChangeEvent e) {
        if (listener() != null)
            listener().figureRequestRemove(new
FigureChangeEvent(this));
    }

    public void figureRequestRemove(FigureChangeEvent e) {}
    public void figureInvalidated(FigureChangeEvent e) {}
    public void figureRequestUpdate(FigureChangeEvent e) {}

    public void release() {
        super.release();
        handleDisconnect(startFigure(), endFigure());
        if (fStart != null)
startFigure().removeFigureChangeListener(this);
        if (fEnd != null)
endFigure().removeFigureChangeListener(this);
    }

    public void write(StorableOutput dw) {
        super.write(dw);
        dw.writeStorable(fStart);
        dw.writeStorable(fEnd);
    }

    public void read(StorableInput dr) throws IOException {

```

```

        super.read(dr);
        Connector start = (Connector)dr.readStorable();
        if (start != null)
            connectStart(start);
        Connector end = (Connector)dr.readStorable();
        if (end != null)
            connectEnd(end);
        if (start != null && end != null)
            updateConnection();
    }

    private void readObject(ObjectInputStream s)
        throws ClassNotFoundException, IOException {

        s.defaultReadObject();

        if (fStart != null)
            connectStart(fStart);
        if (fEnd != null)
            connectEnd(fEnd);
    }
}

```

Clase LineFigure

Una figura linea.

Se encuentra en el fichero LineFigure.java y su contenido es el siguiente:

```

/*
 * @(#)LineFigure.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.util.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.*;

/**
 * A line figure.
 */
public class LineFigure extends PolyLineFigure {

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = 511503575249212371L;
    private int lineFigureSerializedDataVersion = 1;

    /**

```

```
    * Constructs a LineFigure with both start and end set to
    * Point(0,0).
    */
    public LineFigure() {
        addPoint(0, 0);
        addPoint(0, 0);
    }

    /**
     * Gets a copy of the start point.
     */
    public Point startPoint() {
        return pointAt(0);
    }

    /**
     * Gets a copy of the end point.
     */
    public Point endPoint() {
        return pointAt(1);
    }

    /**
     * Sets the start point.
     */
    public void startPoint(int x, int y) {
        setPointAt(new Point(x,y), 0);
    }

    /**
     * Sets the end point.
     */
    public void endPoint(int x, int y) {
        setPointAt(new Point(x,y), 1);
    }

    /**
     * Sets the start and end point.
     */
    public void setPoints(Point start, Point end) {
        setPointAt(start, 0);
        setPointAt(end, 1);
    }

    public void basicDisplayBox(Point origin, Point corner) {
        setPoints(origin, corner);
    }
}
```

Clase NumberTextFigure

Un TextFigure especializado para editar números.

Se encuentra en el fichero NumberTextFigure.java y su contenido es el siguiente:

```

/*
 * @(#)NumberTextFigure.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.util.*;
import java.awt.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;

/**
 * A TextFigure specialized to edit numbers.
 */
public class NumberTextFigure extends TextFigure {

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = -
4056859232918336475L;
    private int numberTextFigureSerializedDataVersion = 1;

    /**
     * Gets the number of columns to be used by the text overlay.
     * @see FloatingTextField
     */
    public int overlayColumns() {
        return Math.max(4, getText().length());
    }

    /**
     * Gets the numerical value of the contained text.
     * return the value or 0 in the case of an illegal number format.
     */
    public int getValue() {
        int value = 0;
        try {
            value = Integer.parseInt(getText());
        } catch (NumberFormatException e) {
            value = 0;
        }
        return value;
    }

    /**
     * Sets the numerical value of the contained text.
     */
    public void setValue(int value) {
        setText(Integer.toString(value));
    }
}

```

Clase PolyLineConnector

PolyLineConnector encuentra puntos de conexión sobre una PolyLineFigure.

Se encuentra en el fichero PolyLineConnector.java y su contenido es el siguiente:

```
/*
 * @(#)PolyLineConnector.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.*;

/**
 * PolyLineConnector finds connection points on a
 * PolyLineFigure.
 *
 * @see PolyLineFigure
 */
public class PolyLineConnector extends ChopBoxConnector {

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = 6018435940519102865L;

    public PolyLineConnector() {
        super();
    }

    /**
     * Constructs a connector with the given owner figure.
     */
    public PolyLineConnector(Figure owner) {
        super(owner);
    }

    protected Point chop(Figure target, Point from) {
        PolyLineFigure p = (PolyLineFigure)owner();
        // *** based on PolygonFigure's heuristic
        Point ctr = p.center();
        int cx = -1;
        int cy = -1;
        long len = Long.MAX_VALUE;

        // Try for points along edge
```

```

        for (int i = 0; i < p.pointCount()-1; i++) {
            Point p1 = p.pointAt(i);
            Point p2 = p.pointAt(i+1);
            Point chop = Geom.intersect(p1.x,
                                        p1.y,
                                        p2.x,
                                        p2.y,
                                        from.x,
                                        from.y,
                                        ctr.x,
                                        ctr.y);

            if (chop != null) {
                long cl = Geom.length2(chop.x, chop.y, from.x,
from.y);

                if (cl < len) {
                    len = cl;
                    cx = chop.x;
                    cy = chop.y;
                }
            }
        }
        // if none found, pick closest vertex
        //if (len == Long.MAX_VALUE) {
        { // try anyway
            for (int i = 0; i < p.pointCount(); i++) {
                Point pp = p.pointAt(i);
                long l = Geom.length2(pp.x, pp.y, from.x, from.y);
                if (l < len) {
                    len = l;
                    cx = pp.x;
                    cy = pp.y;
                }
            }
        }
        return new Point(cx, cy);
    }
}

```

Clase PolyLineFigure

Una figura poly line consta de una lista de puntos. Tiene una decoración opcional de la línea al principio y al final.

Se encuentra en el fichero PolyLineFigure.java y su contenido es el siguiente:

```

/*
 * @(#)PolyLineFigure.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.util.*;

```



```
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.*;

/**
 * A poly line figure consists of a list of points.
 * It has an optional line decoration at the start and end.
 *
 * @see LineDecoration
 */
public class PolyLineFigure extends AbstractFigure {

    public final static int ARROW_TIP_NONE = 0;
    public final static int ARROW_TIP_START = 1;
    public final static int ARROW_TIP_END = 2;
    public final static int ARROW_TIP_BOTH = 3;

    protected Vector          fPoints;
    protected LineDecoration  fStartDecoration = null;
    protected LineDecoration  fEndDecoration = null;
    protected Color           fFrameColor = Color.black;

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = -
7951352179906577773L;
    private int polyLineFigureSerializedDataVersion = 1;

    public PolyLineFigure() {
        fPoints = new Vector(4);
    }

    public PolyLineFigure(int size) {
        fPoints = new Vector(size);
    }

    public PolyLineFigure(int x, int y) {
        fPoints = new Vector();
        fPoints.addElement(new Point(x, y));
    }

    public Rectangle displayBox() {
        Enumeration k = points();
        Rectangle r = new Rectangle((Point) k.nextElement());

        while (k.hasMoreElements())
            r.add((Point) k.nextElement());

        return r;
    }

    public boolean isEmpty() {
        return (size().width < 3) && (size().height < 3);
    }

    public Vector handles() {
```

```

        Vector handles = new Vector(fPoints.size());
        for (int i = 0; i < fPoints.size(); i++)
            handles.addElement(new PolyLineHandle(this, locator(i),
i));
        return handles;
    }

    public void basicDisplayBox(Point origin, Point corner) {
    }

    /**
     * Adds a node to the list of points.
     */
    public void addPoint(int x, int y) {
        fPoints.addElement(new Point(x, y));
        changed();
    }

    public Enumeration points() {
        return fPoints.elements();
    }

    public int pointCount() {
        return fPoints.size();
    }

    protected void basicMoveBy(int dx, int dy) {
        Enumeration k = fPoints.elements();
        while (k.hasMoreElements())
            ((Point) k.nextElement()).translate(dx, dy);
    }

    /**
     * Changes the position of a node.
     */
    public void setPointAt(Point p, int i) {
        willChange();
        fPoints.setElementAt(p, i);
        changed();
    }

    /**
     * Insert a node at the given point.
     */
    public void insertPointAt(Point p, int i) {
        fPoints.insertElementAt(p, i);
        changed();
    }

    public void removePointAt(int i) {
        willChange();
        fPoints.removeElementAt(i);
        changed();
    }

    /**
     * Splits the segment at the given point if a segment was hit.
     * @return the index of the segment or -1 if no segment was hit.

```

```

    */
    public int splitSegment(int x, int y) {
        int i = findSegment(x, y);
        if (i != -1)
            insertPointAt(new Point(x, y), i+1);
        return i+1;
    }

    public Point pointAt(int i) {
        return (Point)fPoints.elementAt(i);
    }

    /**
     * Joins to segments into one if the given point hits a node
     * of the polyline.
     * @return true if the two segments were joined.
     */
    public boolean joinSegments(int x, int y) {
        for (int i= 1; i < fPoints.size()-1; i++) {
            Point p = pointAt(i);
            if (Geom.length(x, y, p.x, p.y) < 3) {
                removePointAt(i);
                return true;
            }
        }
        return false;
    }

    public Connector connectorAt(int x, int y) {
        return new PolyLineConnector(this);
    }

    /**
     * Sets the start decoration.
     */
    public void setStartDecoration(LineDecoration l) {
        fStartDecoration = l;
    }

    /**
     * Sets the end decoration.
     */
    public void setEndDecoration(LineDecoration l) {
        fEndDecoration = l;
    }

    public void draw(Graphics g) {
        g.setColor(getFrameColor());
        Point p1, p2;
        for (int i = 0; i < fPoints.size()-1; i++) {
            p1 = (Point) fPoints.elementAt(i);
            p2 = (Point) fPoints.elementAt(i+1);
            g.drawLine(p1.x, p1.y, p2.x, p2.y);
        }
        decorate(g);
    }

    public boolean containsPoint(int x, int y) {

```

```

        Rectangle bounds = displayBox();
        bounds.grow(4,4);
        if (!bounds.contains(x, y))
            return false;

        Point p1, p2;
        for (int i = 0; i < fPoints.size()-1; i++) {
            p1 = (Point) fPoints.elementAt(i);
            p2 = (Point) fPoints.elementAt(i+1);
            if (Geom.lineContainsPoint(p1.x, p1.y, p2.x, p2.y, x, y))
                return true;
        }
        return false;
    }

    /**
     * Gets the segment of the polyline that is hit by
     * the given point.
     * @return the index of the segment or -1 if no segment was hit.
     */
    public int findSegment(int x, int y) {
        Point p1, p2;
        for (int i = 0; i < fPoints.size()-1; i++) {
            p1 = (Point) fPoints.elementAt(i);
            p2 = (Point) fPoints.elementAt(i+1);
            if (Geom.lineContainsPoint(p1.x, p1.y, p2.x, p2.y, x, y))
                return i;
        }
        return -1;
    }

    private void decorate(Graphics g) {
        if (fStartDecoration != null) {
            Point p1 = (Point)fPoints.elementAt(0);
            Point p2 = (Point)fPoints.elementAt(1);
            fStartDecoration.draw(g, p1.x, p1.y, p2.x, p2.y);
        }
        if (fEndDecoration != null) {
            Point p3 = (Point)fPoints.elementAt(fPoints.size()-2);
            Point p4 = (Point)fPoints.elementAt(fPoints.size()-1);
            fEndDecoration.draw(g, p4.x, p4.y, p3.x, p3.y);
        }
    }

    /**
     * Gets the attribute with the given name.
     * PolyLineFigure maps "ArrowMode" to a
     * line decoration.
     */
    public Object getAttribute(String name) {
        if (name.equals("FrameColor")) {
            return getFrameColor();
        }
        else if (name.equals("ArrowMode")) {
            int value = 0;
            if (fStartDecoration != null)
                value |= ARROW_TIP_START;
            if (fEndDecoration != null)

```

```

        value |= ARROW_TIP_END;
        return new Integer(value);
    }
    return super.getAttribute(name);
}

/**
 * Sets the attribute with the given name.
 * PolyLineFigure interprets "ArrowMode" to set
 * the line decoration.
 */
public void setAttribute(String name, Object value) {
    if (name.equals("FrameColor")) {
        setFrameColor((Color) value);
        changed();
    }
    else if (name.equals("ArrowMode")) {
        Integer intObj = (Integer) value;
        if (intObj != null) {
            int decoration = intObj.intValue();
            if ((decoration & ARROW_TIP_START) != 0)
                fStartDecoration = new ArrowTip();
            else
                fStartDecoration = null;
            if ((decoration & ARROW_TIP_END) != 0)
                fEndDecoration = new ArrowTip();
            else
                fEndDecoration = null;
        }
        changed();
    }
    else
        super.setAttribute(name, value);
}

public void write(StorableOutput dw) {
    super.write(dw);
    dw.writeInt(fPoints.size());
    Enumeration k = fPoints.elements();
    while (k.hasMoreElements()) {
        Point p = (Point) k.nextElement();
        dw.writeInt(p.x);
        dw.writeInt(p.y);
    }
    dw.writeStorable(fStartDecoration);
    dw.writeStorable(fEndDecoration);
    dw.writeColor(fFrameColor);
}

public void read(StorableInput dr) throws IOException {
    super.read(dr);
    int size = dr.readInt();
    fPoints = new Vector(size);
    for (int i=0; i<size; i++) {
        int x = dr.readInt();
        int y = dr.readInt();
        fPoints.addElement(new Point(x,y));
    }
}

```

```

        fStartDecoration = (LineDecoration)dr.readStorable();
        fEndDecoration = (LineDecoration)dr.readStorable();
        fFrameColor = dr.readColor();
    }

    /**
     * Creates a locator for the point with the given index.
     */
    public static Locator locator(int pointIndex) {
        return new PolyLineLocator(pointIndex);
    }

    protected Color getFrameColor() {
        return fFrameColor;
    }

    protected void setFrameColor(Color c) {
        fFrameColor = c;
    }
}

```

Clase PolyLineHandle

Un manejador para un nodo de la poly line.

Se encuentra en el fichero PolyLineHandle.java y su contenido es el siguiente:

```

/**
 * @(#)PolyLineHandle.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.LocatorHandle;

/**
 * A handle for a node on the polyline.
 */
public class PolyLineHandle extends LocatorHandle {

    private int fIndex;
    private Point fAnchor;

    /**
     * Constructs a poly line handle.
     * @param owner the owning polyline figure.
     * @param l the locator
     * @param index the index of the node associated with this handle.
     */
    public PolyLineHandle(PolyLineFigure owner, Locator l, int index)
    {

```

```

        super(owner, 1);
        fIndex = index;
    }

    public void invokeStart(int x, int y, DrawingView view) {
        fAnchor = new Point(x, y);
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        myOwner().setPointAt(new Point(x, y), fIndex);
    }

    private PolyLineFigure myOwner() {
        return (PolyLineFigure)owner();
    }
}

```

Clase PolyLineLocator

Una figura poly line consta de una lista de puntos. Tiene una decoración opcional de la línea al principio y al final.

Se encuentra en el fichero PolyLineLocator.java y su contenido es el siguiente:

```

/*
 * @(#)PolyLineLocator.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.*;

/**
 * A poly line figure consists of a list of points.
 * It has an optional line decoration at the start and end.
 *
 * @see LineDecoration
 */
class PolyLineLocator extends AbstractLocator {
    int fIndex;

    public PolyLineLocator(int index) {
        fIndex = index;
    }

    public Point locate(Figure owner) {
        PolyLineFigure plf = (PolyLineFigure)owner;
        // guard against changing PolyLineFigures -> temporary hack
    }
}

```

```

        if (fIndex < plf.pointCount())
            return ((PolyLineFigure)owner).pointAt(fIndex);
        return new Point(0, 0);
    }
}

```

Clase RadiusHandle

Un Handle que manipula el radio de una esquina redonda de un rectángulo.

Se encuentra en el fichero RadiusHandle.java y su contenido es el siguiente:

```

/*
 * @(#)RadiusHandle.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.Geom;

/**
 * A Handle to manipulate the radius of a round corner rectangle.
 */
class RadiusHandle extends AbstractHandle {

    private Point fRadius;
    private RoundRectangleFigure fOwner;
    private static final int OFFSET = 4;

    public RadiusHandle(RoundRectangleFigure owner) {
        super(owner);
        fOwner = owner;
    }

    public void invokeStart(int x, int y, DrawingView view) {
        fRadius = fOwner.getArc();
        fRadius.x = fRadius.x/2;
        fRadius.y = fRadius.y/2;
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        int dx = x-anchorX;
        int dy = y-anchorY;
        Rectangle r = fOwner.displayBox();
        int rx = Geom.range(0, r.width, 2*(fRadius.x + dx));
        int ry = Geom.range(0, r.height, 2*(fRadius.y + dy));
        fOwner.setArc(rx, ry);
    }
}

```



```

    public Point locate() {
        Point radius = fOwner.getArc();
        Rectangle r = fOwner.displayBox();
        return new Point(r.x+radius.x/2+OFFSET,
r.y+radius.y/2+OFFSET);
    }

    public void draw(Graphics g) {
        Rectangle r = displayBox();

        g.setColor(Color.yellow);
        g.fillOval(r.x, r.y, r.width, r.height);

        g.setColor(Color.black);
        g.drawOval(r.x, r.y, r.width, r.height);
    }
}

```

Clase RectangleFigure

Una figura rectángulo.

Se encuentra en el fichero RectangleFigure.java y su contenido es el siguiente:

```

/*
 * @(#)RectangleFigure.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.io.IOException;
import java.util.Vector;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.*;

/**
 * A rectangle figure.
 */
public class RectangleFigure extends AttributeFigure {

    private Rectangle    fDisplayBox;

    /*
     * Serialization support.
     */
    private static final long serialVersionUID = 184722075881789163L;
    private int rectangleFigureSerializedDataVersion = 1;

    public RectangleFigure() {
        this(new Point(0,0), new Point(0,0));
    }
}

```

```

    }

    public RectangleFigure(Point origin, Point corner) {
        basicDisplayBox(origin, corner);
    }

    public void basicDisplayBox(Point origin, Point corner) {
        fDisplayBox = new Rectangle(origin);
        fDisplayBox.add(corner);
    }

    public Vector handles() {
        Vector handles = new Vector();
        BoxHandleKit.addHandles(this, handles);
        return handles;
    }

    public Rectangle displayBox() {
        return new Rectangle(
            fDisplayBox.x,
            fDisplayBox.y,
            fDisplayBox.width,
            fDisplayBox.height);
    }

    protected void basicMoveBy(int x, int y) {
        fDisplayBox.translate(x, y);
    }

    public void drawBackground(Graphics g) {
        Rectangle r = displayBox();
        g.fillRect(r.x, r.y, r.width, r.height);
    }

    public void drawFrame(Graphics g) {
        Rectangle r = displayBox();
        g.drawRect(r.x, r.y, r.width-1, r.height-1);
    }

    //-- store / load -----

    public void write(StorableOutput dw) {
        super.write(dw);
        dw.writeInt(fDisplayBox.x);
        dw.writeInt(fDisplayBox.y);
        dw.writeInt(fDisplayBox.width);
        dw.writeInt(fDisplayBox.height);
    }

    public void read(StorableInput dr) throws IOException {
        super.read(dr);
        fDisplayBox = new Rectangle(
            dr.readInt(),
            dr.readInt(),
            dr.readInt(),
            dr.readInt());
    }

```

```
}
```

Clase RoundedRectangleFigure

Una figura rectángulo redondeado.

Se encuentra en el fichero RoundedRectangleFigure.java y su contenido es el siguiente:

```
/*
 * @(#)RoundedRectangleFigure.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.io.IOException;
import java.util.Vector;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.*;

/**
 * A round rectangle figure.
 * @see RadiusHandle
 */
public class RoundedRectangleFigure extends AttributeFigure {

    private Rectangle    fDisplayBox;
    private int          fArcWidth;
    private int          fArcHeight;
    private static final int DEFAULT_ARC = 8;

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = 7907900248924036885L;
    private int roundRectangleSerializedDataVersion = 1;

    public RoundedRectangleFigure() {
        this(new Point(0,0), new Point(0,0));
        fArcWidth = fArcHeight = DEFAULT_ARC;
    }

    public RoundedRectangleFigure(Point origin, Point corner) {
        basicDisplayBox(origin,corner);
        fArcWidth = fArcHeight = DEFAULT_ARC;
    }

    public void basicDisplayBox(Point origin, Point corner) {
        fDisplayBox = new Rectangle(origin);
        fDisplayBox.add(corner);
    }
}
```

```

/**
 * Sets the arc's width and height.
 */
public void setArc(int width, int height) {
    willChange();
    fArcWidth = width;
    fArcHeight = height;
    changed();
}

/**
 * Gets the arc's width and height.
 */
public Point getArc() {
    return new Point(fArcWidth, fArcHeight);
}

public Vector handles() {
    Vector handles = new Vector();
    BoxHandleKit.addHandles(this, handles);

    handles.addElement(new RadiusHandle(this));

    return handles;
}

public Rectangle displayBox() {
    return new Rectangle(
        fDisplayBox.x,
        fDisplayBox.y,
        fDisplayBox.width,
        fDisplayBox.height);
}

protected void basicMoveBy(int x, int y) {
    fDisplayBox.translate(x,y);
}

public void drawBackground(Graphics g) {
    Rectangle r = displayBox();
    g.fillRoundRect(r.x, r.y, r.width, r.height, fArcWidth,
fArcHeight);
}

public void drawFrame(Graphics g) {
    Rectangle r = displayBox();
    g.drawRoundRect(r.x, r.y, r.width-1, r.height-1, fArcWidth,
fArcHeight);
}

public Insets connectionInsets() {
    return new Insets(fArcHeight/2, fArcWidth/2, fArcHeight/2,
fArcWidth/2);
}

public Connector connectorAt(int x, int y) {
    return new ShortestDistanceConnector(this); // just for demo

```

```

                                                                    //purposes
    }

    public void write(StorableOutput dw) {
        super.write(dw);
        dw.writeInt(fDisplayBox.x);
        dw.writeInt(fDisplayBox.y);
        dw.writeInt(fDisplayBox.width);
        dw.writeInt(fDisplayBox.height);
        dw.writeInt(fArcWidth);
        dw.writeInt(fArcHeight);
    }

    public void read(StorableInput dr) throws IOException {
        super.read(dr);
        fDisplayBox = new Rectangle(
            dr.readInt(),
            dr.readInt(),
            dr.readInt(),
            dr.readInt());
        fArcWidth = dr.readInt();
        fArcHeight = dr.readInt();
    }
}

```

Clase ScribbleTool

Herramienta para garabatear una PolyLineFigure.

Se encuentra en el fichero ScribbleTool.java y su contenido es el siguiente:

```

/*
 * @(#)ScribbleTool.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.awt.event.MouseEvent;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.AbstractTool;

/**
 * Tool to scribble a PolyLineFigure
 * @see PolyLineFigure
 */
public class ScribbleTool extends AbstractTool {

    private PolyLineFigure fScribble;
    private int fLastX, fLastY;
}

```

```

public ScribbleTool(DrawingView view) {
    super(view);
}

public void activate() {
    super.activate();
    fScribble = null;
}

public void deactivate() {
    super.deactivate();
    if (fScribble != null) {
        if (fScribble.size().width < 4 || fScribble.size().height
< 4)
            drawing().remove(fScribble);
    }
}

private void point(int x, int y) {
    if (fScribble == null) {
        fScribble = new PolyLineFigure(x, y);
        view().add(fScribble);
    } else if (fLastX != x || fLastY != y)
        fScribble.addPoint(x, y);

    fLastX = x;
    fLastY = y;
}

public void mouseDown(MouseEvent e, int x, int y) {
    if (e.getClickCount() >= 2) {
        fScribble = null;
        editor().toolDone();
    }
    else {
        // use original event coordinates to avoid
        // supress that the scribble is constrained to
        // the grid
        point(e.getX(), e.getY());
    }
}

public void mouseDrag(MouseEvent e, int x, int y) {
    if (fScribble != null)
        point(e.getX(), e.getY());
}
}

```

Clase ShortestDistanceConnector

Un ShortestDistance localiza los puntos de conexión para encontrar la menor distancia entre el principio y el final de una conexión.

Se encuentra en el fichero ShortestDistanceConnector.java y su contenido es el siguiente:

```

/*
 * @(#)ShortestDistanceConnector.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.Geom;

/**
 * A ShortestDistance locates connection points by
 * finding the shortest distance between the start and
 * end of the connection.
 * It doesn't connect to the arcs defined by Figure.connectionInsets()
 * @see Figure#connectionInsets
 * @see Connector
 */
public class ShortestDistanceConnector extends AbstractConnector {

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = -
2273446020593433887L;

    public ShortestDistanceConnector() { // only used for Storable
        //implementation
        super();
    }

    public ShortestDistanceConnector(Figure owner) {
        super(owner);
    }

    public Point findStart(ConnectionFigure connection) {
        return findPoint(connection, true);
    }

    public Point findEnd(ConnectionFigure connection) {
        return findPoint(connection, false);
    }

    protected Point findPoint(ConnectionFigure connection, boolean
getStart) {
        Figure startFigure = connection.start().owner();
        Figure endFigure = connection.end().owner();

        Rectangle r1 = startFigure.displayBox();
        Rectangle r2 = endFigure.displayBox();

        Insets i1 = startFigure.connectionInsets();
        Insets i2 = endFigure.connectionInsets();
    }
}

```

```

Point p1, p2;
Point start = null, end = null, s = null, e = null;
long len2 = Long.MAX_VALUE, l2;
int x1, x2, y1, y2; // connection points
int xmin, xmax, ymin, ymax;

// X-dimension
// constrain width connection insets
int r1x, r1width, r2x, r2width, r1y, r1height, r2y, r2height;
r1x = r1.x + i1.left;
r1width = r1.width - i1.left - i1.right-1;
r2x = r2.x + i2.left;
r2width = r2.width - i2.left - i2.right-1;

// find x connection point
if (r1x + r1width < r2x) {
    x1 = r1x + r1width;
    x2 = r2x;
} else if (r1x > r2x + r2width) {
    x1 = r1x;
    x2 = r2x + r2width;
} else {
    xmax = Math.max(r1x, r2x);
    xmin = Math.min(r1x+r1width, r2x+r2width);
    x1 = x2 = (xmax + xmin) /2;
}

// Y-Dimension
// constrain with connection insets
r1y = r1.y + i1.top;
r1height = r1.height - i1.top - i1.bottom-1;
r2y = r2.y + i2.top;
r2height = r2.height - i2.top - i2.bottom-1;

// y connection point
if (r1y + r1height < r2y) {
    y1 = r1y + r1height;
    y2 = r2y;
} else if (r1y > r2y + r2height) {
    y1 = r1y;
    y2 = r2y + r2height;
} else {
    ymax = Math.max(r1y, r2y);
    ymin = Math.min(r1y+r1height, r2y+r2height);
    y1 = y2 = (ymax + ymin) /2;
}

// find shortest connection
for (int i = 0; i < 4; i++) {
    switch(i) {
        case 0:
            // EAST-WEST
            p1 = Geom.east(r1);
            p2 = Geom.west(r2);
            s = new Point(p1.x, y1);
            e = new Point(p2.x, y2);
            break;
        case 1:
            // WEST-EAST

```



```

        p1 = Geom.west(r1);
        p2 = Geom.east(r2);
        s = new Point(p1.x, y1);
        e = new Point(p2.x, y2);
        break;
    case 2:
        // NORTH-SOUTH
        p1 = Geom.north(r1);
        p2 = Geom.south(r2);
        s = new Point(x1, p1.y);
        e = new Point(x2, p2.y);
        break;
    case 3:
        // SOUTH-NORTH
        p1 = Geom.south(r1);
        p2 = Geom.north(r2);
        s = new Point(x1, p1.y);
        e = new Point(x2, p2.y);
        break;
    }
    l2 = Geom.length2(s.x, s.y, e.x, e.y);
    if (l2 < len2) {
        start = s;
        end = e;
        len2 = l2;
    }
}
if (getStart())
    return start;
return end;
}
}

```

Clase TextFigure

Una figura texto.

Se encuentra en el fichero TextFigure.java y su contenido es el siguiente:

```

/*
 * @(#)TextFigure.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.util.*;
import java.awt.*;
import java.io.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.*;

```

```

/**
 * A text figure.
 *
 * @see TextTool
 */
public class TextFigure
    extends AttributeFigure
    implements FigureChangeListener, TextHolder {

    private int          fOriginX;
    private int          fOriginY;

    // cache of the TextFigure's size
    transient private boolean fSizeIsDirty = true;
    transient private int    fWidth;
    transient private int    fHeight;

    private String  fText;
    private Font    fFont;
    private boolean fIsReadOnly;

    private Figure fObservedFigure = null;
    private OffsetLocator fLocator = null;

    private static String fgCurrentFontName  = "Helvetica";
    private static int    fgCurrentFontSize  = 12;
    private static int    fgCurrentFontStyle = Font.PLAIN;

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = 4599820785949456124L;
    private int textFigureSerializedDataVersion = 1;

    public TextFigure() {
        fOriginX = 0;
        fOriginY = 0;
        fFont = createCurrentFont();
        setAttribute("FillColor", ColorMap.color("None"));
        fText = new String("");
        fSizeIsDirty = true;
    }

    public void moveBy(int x, int y) {
        willChange();
        basicMoveBy(x, y);
        if (fLocator != null)
            fLocator.moveBy(x, y);
        changed();
    }

    protected void basicMoveBy(int x, int y) {
        fOriginX += x;
        fOriginY += y;
    }

    public void basicDisplayBox(Point newOrigin, Point newCorner) {
        fOriginX = newOrigin.x;

```

```

        fOriginY = newOrigin.y;
    }

    public Rectangle displayBox() {
        Dimension extent = textExtent();
        return new Rectangle(fOriginX, fOriginY, extent.width,
extent.height);
    }

    public Rectangle textDisplayBox() {
        return displayBox();
    }

    /**
     * Tests whether this figure is read only.
     */
    public boolean readOnly() {
        return fIsReadOnly;
    }

    /**
     * Sets the read only status of the text figure.
     */
    public void setReadOnly(boolean isReadOnly) {
        fIsReadOnly = isReadOnly;
    }

    /**
     * Gets the font.
     */
    public Font getFont() {
        return fFont;
    }

    /**
     * Sets the font.
     */
    public void setFont(Font newFont) {
        willChange();
        fFont = newFont;
        markDirty();
        changed();
    }

    /**
     * Updates the location whenever the figure changes itself.
     */
    public void changed() {
        super.changed();
        updateLocation();
    }

    /**
     * A text figure understands the "FontSize", "FontStyle", and
     * "FontName" attributes.
     */
    public Object getAttribute(String name) {
        Font font = getFont();

```

```

        if (name.equals("FontSize"))
            return new Integer(font.getSize());
        if (name.equals("FontStyle"))
            return new Integer(font.getStyle());
        if (name.equals("FontName"))
            return font.getName();
        return super.getAttribute(name);
    }

    /**
     * A text figure understands the "FontSize", "FontStyle", and
     * "FontName" attributes.
     */
    public void setAttribute(String name, Object value) {
        Font font = getFont();
        if (name.equals("FontSize")) {
            Integer s = (Integer)value;
            setFont(new Font(font.getName(), font.getStyle(),
s.intValue() ));
        }
        else if (name.equals("FontStyle")) {
            Integer s = (Integer)value;
            int style = font.getStyle();
            if (s.intValue() == Font.PLAIN)
                style = font.PLAIN;
            else
                style = style ^ s.intValue();
            setFont(new Font(font.getName(), style, font.getSize() ));
        }
        else if (name.equals("FontName")) {
            String n = (String)value;
            setFont(new Font(n, font.getStyle(), font.getSize() ));
        }
        else
            super.setAttribute(name, value);
    }

    /**
     * Gets the text shown by the text figure.
     */
    public String getText() {
        return fText;
    }

    /**
     * Sets the text shown by the text figure.
     */
    public void setText(String newText) {
        if (!newText.equals(fText)) {
            willChange();
            fText = new String(newText);
            markDirty();
            changed();
        }
    }

    /**
     * Tests whether the figure accepts typing.

```

```

    */
    public boolean acceptsTyping() {
        return !fIsReadOnly;
    }

    public void drawBackground(Graphics g) {
        Rectangle r = displayBox();
        g.fillRect(r.x, r.y, r.width, r.height);
    }

    public void drawFrame(Graphics g) {
        g.setFont(fFont);
        g.setColor((Color) getAttribute("TextColor"));
        FontMetrics metrics = g.getFontMetrics(fFont);
        g.drawString(fText, fOriginX, fOriginY + metrics.getAscent());
    }

    private Dimension textExtent() {
        if (!fSizeIsDirty)
            return new Dimension(fWidth, fHeight);
        FontMetrics metrics =
Toolkit.getDefaultToolkit().getFontMetrics(fFont);
        fWidth = metrics.stringWidth(fText);
        fHeight = metrics.getHeight();
        fSizeIsDirty = false;
        return new Dimension(metrics.stringWidth(fText),
metrics.getHeight());
    }

    private void markDirty() {
        fSizeIsDirty = true;
    }

    /**
     * Gets the number of columns to be overlaid when the figure is
     * edited.
     */
    public int overlayColumns() {
        int length = getText().length();
        int columns = 20;
        if (length != 0)
            columns = getText().length() + 3;
        return columns;
    }

    public Vector handles() {
        Vector handles = new Vector();
        handles.addElement(new NullHandle(this,
RelativeLocator.northWest()));
        handles.addElement(new NullHandle(this,
RelativeLocator.northEast()));
        handles.addElement(new NullHandle(this,
RelativeLocator.southEast()));
        handles.addElement(new FontSizeHandle(this,
RelativeLocator.southWest()));
        return handles;
    }

```

```

public void write(StorableOutput dw) {
    super.write(dw);
    dw.writeInt(fOriginX);
    dw.writeInt(fOriginY);
    dw.writeString(fText);
    dw.writeString(fFont.getName());
    dw.writeInt(fFont.getStyle());
    dw.writeInt(fFont.getSize());
    dw.writeBoolean(fIsReadOnly);
    dw.writeStorable(fObservedFigure);
    dw.writeStorable(fLocator);
}

public void read(StorableInput dr) throws IOException {
    super.read(dr);
    markDirty();
    fOriginX = dr.readInt();
    fOriginY = dr.readInt();
    fText = dr.readString();
    fFont = new Font(dr.readString(), dr.readInt(), dr.readInt());
    fIsReadOnly = dr.readBoolean();

    fObservedFigure = (Figure)dr.readStorable();
    if (fObservedFigure != null)
        fObservedFigure.addFigureChangeListener(this);
    fLocator = (OffsetLocator)dr.readStorable();
}

private void readObject(ObjectInputStream s)
    throws ClassNotFoundException, IOException {
    s.defaultReadObject();

    if (fObservedFigure != null)
        fObservedFigure.addFigureChangeListener(this);
    markDirty();
}

public void connect(Figure figure) {
    if (fObservedFigure != null)
        fObservedFigure.removeFigureChangeListener(this);

    fObservedFigure = figure;
    fLocator = new
OffsetLocator(figure.connectedTextLocator(this));
    fObservedFigure.addFigureChangeListener(this);
    updateLocation();
}

public void figureChanged(FigureChangeEvent e) {
    updateLocation();
}

public void figureRemoved(FigureChangeEvent e) {
    if (listener() != null)
        listener().figureRequestRemove(new
FigureChangeEvent(this));
}

```

```

public void figureRequestRemove(FigureChangeEvent e) {}
public void figureInvalidated(FigureChangeEvent e) {}
public void figureRequestUpdate(FigureChangeEvent e) {}

/**
 * Updates the location relative to the connected figure.
 * The TextFigure is centered around the located point.
 */
protected void updateLocation() {
    if (fLocator != null) {
        Point p = fLocator.locate(fObservedFigure);
        p.x -= size().width/2 + fOriginX;
        p.y -= size().height/2 + fOriginY;

        if (p.x != 0 || p.y != 0) {
            willChange();
            basicMoveBy(p.x, p.y);
            changed();
        }
    }
}

public void release() {
    super.release();
    if (fObservedFigure != null)
        fObservedFigure.removeFigureChangeListener(this);
}

/**
 * Disconnects the text figure.
 */
public void disconnect() {
    fObservedFigure.removeFigureChangeListener(this);
    fObservedFigure = null;
    fLocator = null;
}

/**
 * Creates the current font to be used for new text figures.
 */
static public Font createCurrentFont() {
    return new Font(fgCurrentFontName, fgCurrentFontStyle,
fgCurrentFontSize);
}

/**
 * Sets the current font name
 */
static public void setCurrentFontName(String name) {
    fgCurrentFontName = name;
}

/**
 * Sets the current font size.
 */
static public void setCurrentFontSize(int size) {

```

```

        fgCurrentFontSize = size;
    }

    /**
     * Sets the current font style.
     */
    static public void setCurrentFontStyle(int style) {
        fgCurrentFontStyle = style;
    }
}

```

Clase TextTool

Herramienta para crear o editar figuras existentes de texto. El comportamiento de edición es implementado sobrescribiendo la Figure proporcionando el texto con un FloatingTextField.

Se encuentra en el fichero TextTool.java y su contenido es el siguiente:

```

/*
 * @(#)TextTool.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.awt.event.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.FloatingTextField;

/**
 * Tool to create new or edit existing text figures.
 * The editing behavior is implemented by overlaying the
 * Figure providing the text with a FloatingTextField.<p>
 * A tool interaction is done once a Figure that is not
 * a TextHolder is clicked.
 *
 * @see TextHolder
 * @see FloatingTextField
 */
public class TextTool extends CreationTool {

    private FloatingTextField fTextField;
    private TextHolder fTypingTarget;

    public TextTool(DrawingView view, Figure prototype) {
        super(view, prototype);
    }

    /**
     * If the pressed figure is a TextHolder it can be edited
     * otherwise a new text figure is created.
     */
}

```



```

public void mouseDown(MouseEvent e, int x, int y)
{
    Figure pressedFigure;
    TextHolder textHolder = null;

    pressedFigure = drawing().findFigureInside(x, y);
    if (pressedFigure instanceof TextHolder) {
        textHolder = (TextHolder) pressedFigure;
        if (!textHolder.acceptsTyping())
            textHolder = null;
    }
    if (textHolder != null) {
        beginEdit(textHolder);
        return;
    }
    if (fTypingTarget != null) {
        editor().toolDone();
        endEdit();
    } else {
        super.mouseDown(e, x, y);
        textHolder = (TextHolder) createdFigure();
        beginEdit(textHolder);
    }
}

public void mouseDrag(MouseEvent e, int x, int y) {
}

public void mouseUp(MouseEvent e, int x, int y) {
}

/**
 * Terminates the editing of a text figure.
 */
public void deactivate() {
    super.deactivate();
    endEdit();
}

/**
 * Sets the text cursor.
 */
public void activate() {
    super.activate();
    view().clearSelection();
    // JDK1.1 TEXT_CURSOR has an incorrect hot spot
    //view.setCursor(Cursor.getPredefinedCursor(Cursor.TEXT_CURSOR));
}

protected void beginEdit(TextHolder figure) {
    if (fTextField == null)
        fTextField = new FloatingTextField();

    if (figure != fTypingTarget && fTypingTarget != null)
        endEdit();

    fTextField.createOverlay((Container) view(), figure.getFont());
}

```

```

        fTextField.setBounds(fieldBounds(figure), figure.getText());
        fTypingTarget = figure;
    }

    protected void endEdit() {
        if (fTypingTarget != null) {
            if (fTextField.getText().length() > 0)
                fTypingTarget.setText(fTextField.getText());
            else
                drawing().remove((Figure)fTypingTarget);
            fTypingTarget = null;
            fTextField.endOverlay();
            view().checkDamage();
        }
    }

    private Rectangle fieldBounds(TextHolder figure) {
        Rectangle box = figure.textDisplayBox();
        int nChars = figure.overlayColumns();
        Dimension d = fTextField.getPreferredSize(nChars);
        return new Rectangle(box.x, box.y, d.width, d.height);
    }
}

```

Clase UngroupCommand

Comando para desagrupar las figuras seleccionadas.

Se encuentra en el fichero UngroupCommand.java y su contenido es el siguiente:

```

/*
 * @(#)UngroupCommand.java 5.1
 *
 */

package CH.ifa.draw.figures;

import java.awt.*;
import java.util.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.Command;

/**
 * Command to ungroup the selected figures.
 * @see GroupCommand
 */
public class UngroupCommand extends Command {

    private DrawingView fView;

    /**
     * Constructs a group command.
     * @param name the command name
     * @param view the target view
     */
}

```

```

    */
    public UngroupCommand(String name, DrawingView view) {
        super(name);
        fView = view;
    }

    public void execute() {
        FigureEnumeration selection = fView.selectionElements();
        fView.clearSelection();

        Vector parts = new Vector();
        while (selection.hasMoreElements()) {
            Figure selected = selection.nextFigure();
            Figure group = fView.drawing().orphan(selected);
            FigureEnumeration k = group.decompose();
            while (k.hasMoreElements())
                fView.addToSelection(fView.add(k.nextFigure()));
        }
        fView.checkDamage();
    }

    public boolean isExecutable() {
        return fView.selectionCount() > 0;
    }
}

```


Paquete CH.ifa.draw.contrib

Las clases donde han colaborado otros. Es un paquete con aportaciones al JHotDraw.

Este paquete contiene lo siguiente:

- Clase ChopPolygonConnector.
- Clase DiamondFigure.
- Clase PolygonFigure.
- Clase PolygonHandle.
- Clase PolygonScaleHandle.
- Clase PolygonTool.
- Clase TriangleFigure.
- Clase TriangleRotationHandle

Clase ChopPolygonConnector

Un ChopPolygonConnector localiza un punto de conexión mediante partición de la conexión en la frontera del polígono.

Se encuentra en el fichero ChopPolygonConnector.java y su contenido es el siguiente:

```
/*
 * Copyright (c) 1996, 1997 Erich Gamma
 * All Rights Reserved
 */

package CH.ifa.draw.contrib;

import java.awt.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.Geom;

/**
 * A ChopPolygonConnector locates a connection point by
 * chopping the connection at the polygon boundary.
 */
public class ChopPolygonConnector extends ChopBoxConnector {

    /**
     * Serialization support.
     */
    private static final long serialVersionUID = -156024908227796826L;

    public ChopPolygonConnector() {
    }
}
```

```
public ChopPolygonConnector(Figure owner) {
    super(owner);
}

protected Point chop(Figure target, Point from) {
    return ((PolygonFigure)target).chop(from);
}
}
```

Clase DiamondFigure

Un diamante con vértices en la mediatriz del rectángulo que lo encierra.

Se encuentra en el fichero DiamondFigure.java y su contenido es el siguiente:

```
/*
 * Hacked together by Doug Lea
 * Tue Feb 25 17:39:44 1997  Doug Lea  (dl at gee)
 *
 */

package CH.ifa.draw.contrib;

import java.awt.*;
import java.util.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;

/**
 * A diamond with vertices at the midpoints of its enclosing rectangle
 */
public class DiamondFigure extends RectangleFigure {

    public DiamondFigure() {
        super(new Point(0,0), new Point(0,0));
    }

    public DiamondFigure(Point origin, Point corner) {
        super(origin, corner);
    }

    /** Return the polygon describing the diamond */
    protected Polygon polygon() {
        Rectangle r = displayBox();
        Polygon p = new Polygon();
        p.addPoint(r.x, r.y+r.height/2);
        p.addPoint(r.x+r.width/2, r.y);
        p.addPoint(r.x+r.width, r.y+r.height/2);
        p.addPoint(r.x+r.width/2, r.y+r.height);
        return p;
    }
}
```

```

    }

    public void draw(Graphics g) {
        Polygon p = polygon();
        g.setColor(getFillColor());
        g.fillPolygon(p);
        g.setColor(getFrameColor());
        g.drawPolygon(p);
    }

    public Insets connectionInsets() {
        Rectangle r = displayBox();
        return new Insets(r.height/2, r.width/2, r.height/2, r.width/2);
    }

    public boolean containsPoint(int x, int y) {
        return polygon().contains(x, y);
    }

    /*public Point chop(Point p) {
        return PolygonFigure.chop(polygon(), p);
    }*/
}

```

Clase PolygonFigure

Un polígono escalable, rotatable con un número arbitrario de puntos.

Se encuentra en el fichero PolygonFigure.java y su contenido es el siguiente:

```

/*
 * Fri Feb 28 07:47:05 1997  Doug Lea  (dl at gee)
 * Based on PolyLineFigure
 */

package CH.ifa.draw.contrib;

import java.awt.*;
import java.util.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;

/**
 * A scalable, rotatable polygon with an arbitrary number of points
 */
public class PolygonFigure extends AttributeFigure {

    /**
     * Distance threshold for smoothing away or locating points
     */
}

```

```

static final int TOO_CLOSE = 2;

/*
 * Serialization support.
 */
private static final long serialVersionUID = 6254089689239215026L;
private int polygonFigureSerializedDataVersion = 1;

protected Polygon fPoly = new Polygon();

public PolygonFigure() {
    super();
}

public PolygonFigure(int x, int y) {
    fPoly.addPoint(x, y);
}

public PolygonFigure(Polygon p) {
    fPoly = new Polygon(p.xpoints, p.ypoints, p.npoints);
}

public Rectangle displayBox() {
    return bounds(fPoly);
}

public boolean isEmpty() {
    return (fPoly.npoints < 3 ||
            (size().width < TOO_CLOSE) && (size().height <
TOO_CLOSE));
}

public Vector handles() {
    Vector handles = new Vector(fPoly.npoints);
    for (int i = 0; i < fPoly.npoints; i++)
        handles.addElement(new PolygonHandle(this, locator(i), i));
    handles.addElement(new PolygonScaleHandle(this));
    //handles.addElement(new PolygonPointAddHandle(this));
    return handles;
}

public void basicDisplayBox(Point origin, Point corner) {
    Rectangle r = displayBox();
    int dx = origin.x - r.x;
    int dy = origin.y - r.y;
    fPoly.translate(dx, dy);
    r = displayBox();
    Point oldCorner = new Point(r.x + r.width, r.y + r.height);
    Polygon p = getPolygon();
    scaleRotate(oldCorner, p, corner);
}

/**
 * return a copy of the raw polygon
 */
public Polygon getPolygon() {

```



```

    return new Polygon(fPoly.xpoints, fPoly.ypoints, fPoly.npoints);
}

public Point center() {
    return center(fPoly);
}

public Enumeration points() {
    Vector pts = new Vector(fPoly.npoints);
    for (int i = 0; i < fPoly.npoints; ++i)
        pts.addElement(new Point(fPoly.xpoints[i], fPoly.ypoints[i]));
    return pts.elements();
}

public int pointCount() {
    return fPoly.npoints;
}

public void basicMoveBy(int dx, int dy) {
    fPoly.translate(dx, dy);
}

public void drawBackground(Graphics g) {
    g.fillPolygon(fPoly);
}

public void drawFrame(Graphics g) {
    g.drawPolygon(fPoly);
}

public boolean containsPoint(int x, int y) {
    return fPoly.contains(x, y);
}

public Connector connectorAt(int x, int y) {
    return new ChopPolygonConnector(this);
}

/**
 * Adds a node to the list of points.
 */
public void addPoint(int x, int y) {
    fPoly.addPoint(x, y);
    changed();
}

/**
 * Changes the position of a node.
 */
public void setPointAt(Point p, int i) {
    willChange();
    fPoly.xpoints[i] = p.x;
    fPoly.ypoints[i] = p.y;
    changed();
}

/**

```

```

    * Insert a node at the given point.
    */
    public void insertPointAt(Point p, int i) {
        willChange();
        int n = fPoly.npoints + 1;
        int[] xs = new int[n];
        int[] ys = new int[n];
        for (int j = 0; j < i; ++j) {
            xs[j] = fPoly.xpoints[j];
            ys[j] = fPoly.ypoints[j];
        }
        xs[i] = p.x;
        ys[i] = p.y;
        for (int j = i; j < fPoly.npoints; ++j) {
            xs[j+1] = fPoly.xpoints[j];
            ys[j+1] = fPoly.ypoints[j];
        }

        fPoly = new Polygon(xs, ys, n);
        changed();
    }

    public void removePointAt(int i) {
        willChange();
        int n = fPoly.npoints - 1;
        int[] xs = new int[n];
        int[] ys = new int[n];
        for (int j = 0; j < i; ++j) {
            xs[j] = fPoly.xpoints[j];
            ys[j] = fPoly.ypoints[j];
        }
        for (int j = i; j < n; ++j) {
            xs[j] = fPoly.xpoints[j+1];
            ys[j] = fPoly.ypoints[j+1];
        }
        fPoly = new Polygon(xs, ys, n);
        changed();
    }

    /**
     * Scale and rotate relative to anchor
     */
    public void scaleRotate(Point anchor, Polygon originalPolygon,
        Point p) {
        willChange();

        // use center to determine relative angles and lengths
        Point ctr = center(originalPolygon);
        double anchorLen = Geom.length(ctr.x, ctr.y, anchor.x, anchor.y);

        if (anchorLen > 0.0) {
            double newLen = Geom.length(ctr.x, ctr.y, p.x, p.y);
            double ratio = newLen / anchorLen;

            double anchorAngle = Math.atan2(anchor.y - ctr.y, anchor.x -
ctr.x);
            double newAngle = Math.atan2(p.y - ctr.y, p.x - ctr.x);
            double rotation = newAngle - anchorAngle;

```

```

        int n = originalPolygon.npoints;
        int[] xs = new int[n];
        int[] ys = new int[n];

        for (int i = 0; i < n; ++i) {
            int x = originalPolygon.xpoints[i];
            int y = originalPolygon.ypoints[i];
            double l = Geom.length(ctr.x, ctr.y, x, y) * ratio;
            double a = Math.atan2(y - ctr.y, x - ctr.x) + rotation;
            xs[i] = (int)(ctr.x + l * Math.cos(a) + 0.5);
            ys[i] = (int)(ctr.y + l * Math.sin(a) + 0.5);
        }
        fPoly = new Polygon(xs, ys, n);
    }
    changed();
}

/**
 * Remove points that are nearly colinear with others
 */
public void smoothPoints() {
    willChange();
    boolean removed = false;
    int n = fPoly.npoints;
    do {
        removed = false;
        int i = 0;
        while (i < n && n >= 3) {
            int nxt = (i + 1) % n;
            int prv = (i - 1 + n) % n;

            if ((distanceFromLine(fPoly.xpoints[prv], fPoly.ypoints[prv],
                                fPoly.xpoints[nxt], fPoly.ypoints[nxt],
                                fPoly.xpoints[i], fPoly.ypoints[i]) <
TOO_CLOSE)) {
                removed = true;
                --n;
                for (int j = i; j < n; ++j) {
                    fPoly.xpoints[j] = fPoly.xpoints[j+1];
                    fPoly.ypoints[j] = fPoly.ypoints[j+1];
                }
            }
            else
                ++i;
        }
    } while(removed);
    if (n != fPoly.npoints)
        fPoly = new Polygon(fPoly.xpoints, fPoly.ypoints, n);
    changed();
}

/**
 * Splits the segment at the given point if a segment was hit.
 * @return the index of the segment or -1 if no segment was hit.
 */
public int splitSegment(int x, int y) {

```

```

int i = findSegment(x, y);
if (i != -1) {
    insertPointAt(new Point(x, y), i+1);
    return i + 1;
}
else
    return -1;
}

public Point pointAt(int i) {
    return new Point(fPoly.xpoints[i], fPoly.ypoints[i]);
}

/**
 * Return the point on the polygon that is furthest from the center
 */
public Point outermostPoint() {
    Point ctr = center();
    int outer = 0;
    long dist = 0;

    for (int i = 0; i < fPoly.npoints; ++i) {
        long d = Geom.length2(ctr.x, ctr.y, fPoly.xpoints[i],
fPoly.ypoints[i]);
        if (d > dist) {
            dist = d;
            outer = i;
        }
    }

    return new Point(fPoly.xpoints[outer], fPoly.ypoints[outer]);
}

/**
 * Gets the segment that is hit by the given point.
 * @return the index of the segment or -1 if no segment was hit.
 */
public int findSegment(int x, int y) {
    double dist = TOO_CLOSE;
    int best = -1;

    for (int i = 0; i < fPoly.npoints; i++) {
        int n = (i + 1) % fPoly.npoints;
        double d = distanceFromLine(fPoly.xpoints[i], fPoly.ypoints[i],
fPoly.xpoints[n], fPoly.ypoints[n],
x, y);

        if (d < dist) {
            dist = d;
            best = i;
        }
    }
    return best;
}

public Point chop(Point p) {
    return chop(fPoly, p);
}

```



```
//          (YA-YC)(YA-YB)-(XA-XC)(XB-XA)
//          r = -----
//                      L**2
//
//          (YA-YC)(XB-XA)-(XA-XC)(YB-YA)
//          s = -----
//                      L**2
//
//      Let I be the point of perpendicular projection of C onto
//      AB, the
//
//          XI=XA+r(XB-XA)
//          YI=YA+r(YB-YA)
//
//      Distance from A to I = r*L
//      Distance from C to I = s*L
//
//      If r < 0 I is on backward extension of AB
//      If r>1 I is on ahead extension of AB
//      If 0<=r<=1 I is on AB
//
//      If s < 0 C is left of AB (you can just check the
//                                numerator)
//      If s>0 C is right of AB
//      If s=0 C is on AB

int xdiff = xb - xa;
int ydiff = yb - ya;
long l2 = xdiff*xdiff + ydiff*ydiff;

if (l2 == 0) return Geom.length(xa, ya, xc, yc);

double rnum = (ya-yc) * (ya-yb) - (xa-xc) * (xb-xa);
double r = rnum / l2;

if (r < 0.0 || r > 1.0) return Double.MAX_VALUE;

double xi = xa + r * xdiff;
double yi = ya + r * ydiff;
double xd = xc - xi;
double yd = yc - yi;
return Math.sqrt(xd * xd + yd * yd);

/*
   for directional version, instead use
   double snum = (ya-yc) * (xb-xa) - (xa-xc) * (yb-ya);
   double s = snum / l2;

   double l = Math.sqrt((double)l2);
   return = s * l;
*/
}

/**
 * replacement for builtin Polygon.getBounds that doesn't always
 * update?
 */
```

```

public static Rectangle bounds(Polygon p) {
    int minx = Integer.MAX_VALUE;
    int miny = Integer.MAX_VALUE;
    int maxx = Integer.MIN_VALUE;
    int maxy = Integer.MIN_VALUE;
    int n = p.npoints;
    for (int i = 0; i < n; i++) {
        int x = p.xpoints[i];
        int y = p.ypoints[i];
        if (x > maxx) maxx = x;
        if (x < minx) minx = x;
        if (y > maxy) maxy = y;
        if (y < miny) miny = y;
    }

    return new Rectangle(minx, miny, maxx-minx, maxy-miny);
}

public static Point center(Polygon p) {
    long sx = 0;
    long sy = 0;
    int n = p.npoints;
    for (int i = 0; i < n; i++) {
        sx += p.xpoints[i];
        sy += p.ypoints[i];
    }

    return new Point((int)(sx/n), (int)(sy/n));
}

public static Point chop(Polygon poly, Point p) {
    Point ctr = center(poly);
    int cx = -1;
    int cy = -1;
    long len = Long.MAX_VALUE;

    // Try for points along edge

    for (int i = 0; i < poly.npoints; ++i) {
        int nxt = (i + 1) % poly.npoints;
        Point chop = Geom.intersect(poly.xpoints[i],
                                    poly.ypoints[i],
                                    poly.xpoints[nxt],
                                    poly.ypoints[nxt],
                                    p.x,
                                    p.y,
                                    ctr.x,
                                    ctr.y);

        if (chop != null) {
            long cl = Geom.length2(chop.x, chop.y, p.x, p.y);
            if (cl < len) {
                len = cl;
                cx = chop.x;
                cy = chop.y;
            }
        }
    }

    // if none found, pick closest vertex

```

```

        //if (len == Long.MAX VALUE) {
        { // try anyway
            for (int i = 0; i < poly.npoints; ++i) {
                long l = Geom.length2(poly.xpoints[i], poly.ypoints[i], p.x,
p.y);
                if (l < len) {
                    len = l;
                    cx = poly.xpoints[i];
                    cy = poly.ypoints[i];
                }
            }
        }
        return new Point(cx, cy);
    }
}

```

Clase PolygonHandle

Un manejador para un nodo sobre un polígono.

Se encuentra en el fichero PolygonHandle.java y su contenido es el siguiente:

```

/*
 * Fri Feb 28 07:47:13 1997  Doug Lea  (dl at gee)
 * Based on PolyLineHandle
 */

package CH.ifa.draw.contrib;

import java.awt.*;
import java.util.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.standard.*;

/**
 * A handle for a node on the polygon.
 */
public class PolygonHandle extends AbstractHandle {

    private int fIndex;
    private Locator fLocator;

    /**
     * Constructs a polygon handle.
     * @param owner the owning polygon figure.
     * @param l the locator
     * @param index the index of the node associated with this handle.
     */
    public PolygonHandle(PolygonFigure owner, Locator l, int index) {
        super(owner);
    }
}

```



```

        fLocator = 1;
        fIndex = index;
    }

    public void invokeStep (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        myOwner().setPointAt(new Point(x, y), fIndex);
    }

    public void invokeEnd (int x, int y, int anchorX, int anchorY,
DrawingView view) {
        myOwner().smoothPoints();
    }

    public Point locate() {
        return fLocator.locate(owner());
    }

    private PolygonFigure myOwner() {
        return (PolygonFigure)owner();
    }
}

```

Clase PolygonScaleHandle

Un Handle para cambiar el tamaño y rotar un PolygonFigure.

Se encuentra en el fichero PolygonScaleHandle.java y su contenido es el siguiente:

```

/*
 * Sat Mar 1 09:06:09 1997 Doug Lea (dl at gee)
 * Based on RadiusHandle
 */

package CH.ifa.draw.contrib;

import java.awt.*;
import java.util.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.standard.*;

/**
 * A Handle to scale and rotate a PolygonFigure
 */
class PolygonScaleHandle extends AbstractHandle {

    private Point fOrigin = null;
    private Point fCurrent = null;
    private Polygon fOrigPoly = null;

    public PolygonScaleHandle(PolygonFigure owner) {

```

```

    super(owner);
}

public void invokeStart(int x, int y, Drawing drawing) {
    fOrigPoly = ((PolygonFigure)(owner())).getPolygon();
    fOrigin = getOrigin();
    fCurrent = new Point(fOrigin.x, fOrigin.y);
}

public void invokeStep (int dx, int dy, Drawing drawing) {
    fCurrent = new Point(fOrigin.x + dx, fOrigin.y + dy);
    ((PolygonFigure)(owner())).scaleRotate(fOrigin, fOrigPoly,
fCurrent);
}

public void invokeEnd (int dx, int dy, Drawing drawing) {
    fOrigPoly = null;
    fOrigin = null;
    fCurrent = null;
}

public Point locate() {
    if (fCurrent != null)
        return fCurrent;
    else
        return getOrigin();
}

Point getOrigin() { // find a nice place to put handle
    // Need to pick a place that will not overlap with point handle
    // and is internal to polygon

    // Try for one HANDLESIZE step away from outermost toward center

    Point outer = ((PolygonFigure)(owner())).outermostPoint();
    Point ctr = ((PolygonFigure)(owner())).center();
    double len = Geom.length(outer.x, outer.y, ctr.x, ctr.y);
    if (len == 0) // best we can do?
        return new Point(outer.x - HANDLESIZE/2, outer.y +
HANDLESIZE/2);

    double u = HANDLESIZE / len;
    if (u > 1.0) // best we can do?
        return new Point((outer.x * 3 + ctr.x)/4, (outer.y * 3 +
ctr.y)/4);
    else
        return new Point((int)(outer.x * (1.0 - u) + ctr.x * u),
            (int)(outer.y * (1.0 - u) + ctr.y * u));
}

public void draw(Graphics g) {
    Rectangle r = displayBox();

    g.setColor(Color.yellow);
    g.fillOval(r.x, r.y, r.width, r.height);

    g.setColor(Color.black);
    g.drawOval(r.x, r.y, r.width, r.height);
}

```

```

    /*
     * for debugging ...
     Point ctr = ((PolygonFigure)(owner())).center();
     g.setColor(Color.blue);
     g.fillOval(ctr.x, ctr.y, r.width, r.height);

     g.setColor(Color.black);
     g.drawOval(ctr.x, ctr.y, r.width, r.height);

     */
}
}

```

Clase PolygonTool

Se encuentra en el fichero PolygonTool.java y su contenido es el siguiente:

```

/*
 * Fri Feb 28 07:47:05 1997  Doug Lea  (dl at gee)
 * Based on ScribbleTool
 */

package CH.ifa.draw.contrib;

import java.awt.*;
import java.awt.event.MouseEvent;
import java.util.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.standard.*;

/**
 */
public class PolygonTool extends AbstractTool {

    private PolygonFigure fPolygon;
    private int           fLastX, fLastY;

    public PolygonTool(DrawingView view) {
        super(view);
    }

    public void activate() {
        super.activate();
        fPolygon = null;
    }

    public void deactivate() {
        super.deactivate();
        if (fPolygon != null) {
            fPolygon.smoothPoints();
        }
    }
}

```

```

        if (fPolygon.pointCount() < 3 ||
            fPolygon.size().width < 4 || fPolygon.size().height < 4)
            drawing().remove(fPolygon);
    }
    fPolygon = null;
}

private void addPoint(int x, int y) {
    if (fPolygon == null) {
        fPolygon = new PolygonFigure(x, y);
        view().add(fPolygon);
        fPolygon.addPoint(x, y);
    } else if (fLastX != x || fLastY != y)
        fPolygon.addPoint(x, y);

    fLastX = x;
    fLastY = y;
}

public void mouseDown(MouseEvent e, int x, int y) {
    // replace pts by actual event pts
    x = e.getX();
    y = e.getY();

    if (e.getClickCount() >= 2) {
        if (fPolygon != null) {
            fPolygon.smoothPoints();
            editor().toolDone();
        }
        fPolygon = null;
    } else {
        // use original event coordinates to avoid
        // supress that the scribble is constrained to
        // the grid
        addPoint(e.getX(), e.getY());
    }
}

public void mouseMove(MouseEvent e, int x, int y) {
    if (fPolygon != null) {
        if (fPolygon.pointCount() > 1) {
            fPolygon.setPointAt(new Point(x, y), fPolygon.pointCount()-
1);
            view().checkDamage();
        }
    }
}

public void mouseDrag(MouseEvent e, int x, int y) {
    // replace pts by actual event pts
    x = e.getX();
    y = e.getY();
    addPoint(x, y);
}

public void mouseUp(MouseEvent e, int x, int y) {

```

```
}  
  
}
```

Clase TriangleFigure

Un triángulo con las mismas dimensiones que el rectángulo que lo encierra y con el pico en alguno de los 8 lugares.

Se encuentra en el fichero TriangleFigure.java y su contenido es el siguiente:

```
/*  
 * Hacked together by Doug lea  
 * Tue Feb 25 17:30:58 1997  Doug Lea  (dl at gee)  
 */  
  
package CH.ifa.draw.contrib;  
  
import java.awt.*;  
import java.util.*;  
import java.io.IOException;  
import CH.ifa.draw.framework.*;  
import CH.ifa.draw.util.*;  
import CH.ifa.draw.standard.*;  
import CH.ifa.draw.figures.*;  
  
/**  
 * A triangle with same dimensions as its enclosing rectangle,  
 * and apex at any of 8 places  
 */  
public class TriangleFigure extends RectangleFigure {  
  
    static double[] rotations = {  
        -Math.PI/2, -Math.PI/4,  
        0.0, Math.PI/4,  
        Math.PI/2, Math.PI * 3/4,  
        Math.PI, -Math.PI * 3/4  
    };  
  
    protected int fRotation = 0;  
  
    public TriangleFigure() {  
        super(new Point(0,0), new Point(0,0));  
    }  
  
    public TriangleFigure(Point origin, Point corner) {  
        super(origin, corner);  
    }  
  
    public Vector handles() {  
        Vector h = super.handles();  
        h.addElement(new TriangleRotationHandle(this));  
        return h;  
    }  
}
```

```

    }

    public void rotate(double angle) {
        willChange();
        //System.out.println("a:"+angle);
        double dist = Double.MAX_VALUE;
        int best = 0;
        for (int i = 0; i < rotations.length; ++i) {
            double d = Math.abs(angle - rotations[i]);
            if (d < dist) {
                dist = d;
                best = i;
            }
        }
        fRotation = best;
        changed();
    }

    /** Return the polygon describing the triangle */
    public Polygon polygon() {
        Rectangle r = displayBox();
        Polygon p = new Polygon();
        switch (fRotation) {
            case 0:
                p.addPoint(r.x+r.width/2, r.y);
                p.addPoint(r.x+r.width, r.y+r.height);
                p.addPoint(r.x, r.y+r.height);
                break;
            case 1:
                p.addPoint(r.x + r.width, r.y);
                p.addPoint(r.x+r.width, r.y+r.height);
                p.addPoint(r.x, r.y);
                break;
            case 2:
                p.addPoint(r.x + r.width, r.y+r.height/2);
                p.addPoint(r.x, r.y+r.height);
                p.addPoint(r.x, r.y);
                break;
            case 3:
                p.addPoint(r.x+r.width, r.y+r.height);
                p.addPoint(r.x, r.y+r.height);
                p.addPoint(r.x + r.width, r.y);
                break;
            case 4:
                p.addPoint(r.x+r.width/2, r.y+r.height);
                p.addPoint(r.x, r.y);
                p.addPoint(r.x + r.width, r.y);
                break;
            case 5:
                p.addPoint(r.x, r.y+r.height);
                p.addPoint(r.x, r.y);
                p.addPoint(r.x+r.width, r.y+r.height);
                break;
            case 6:
                p.addPoint(r.x, r.y+r.height/2);
                p.addPoint(r.x + r.width, r.y);
                p.addPoint(r.x+r.width, r.y+r.height);
                break;
        }
    }

```

```

        case 7:
            p.addPoint(r.x, r.y);
            p.addPoint(r.x + r.width, r.y);
            p.addPoint(r.x, r.y+r.height);
            break;
        }
        return p;
    }

    public void draw(Graphics g) {
        Polygon p = polygon();
        g.setColor(getFillColor());
        g.fillPolygon(p);
        g.setColor(getFrameColor());
        g.drawPolygon(p);
    }

    public Insets connectionInsets() {
        Rectangle r = displayBox();
        switch(fRotation) {
            case 0:
                return new Insets(r.height, r.width/2, 0, r.width/2);
            case 1:
                return new Insets(0, r.width, r.height, 0);
            case 2:
                return new Insets(r.height/2, 0, r.height/2, r.width);
            case 3:
                return new Insets(r.height, r.width, 0, 0);
            case 4:
                return new Insets(0, r.width/2, r.height, r.width/2);
            case 5:
                return new Insets(r.height, 0, 0, r.width);
            case 6:
                return new Insets(r.height/2, r.width, r.height/2, 0);
            case 7:
                return new Insets(0, 0, r.height, r.width);
            default:
                return null;
        }
    }

    public boolean containsPoint(int x, int y) {
        return polygon().contains(x, y);
    }

    public Point center() {
        return PolygonFigure.center(polygon());
    }

    public Point chop(Point p) {
        return PolygonFigure.chop(polygon(), p);
    }

    public Object clone() {
        TriangleFigure figure = (TriangleFigure) super.clone();
        figure.fRotation = fRotation;
    }

```

```

    return figure;
}

    //-- store / load -----

    public void write(StorableOutput dw) {
        super.write(dw);
        dw.writeInt(fRotation);
    }

    public void read(StorableInput dr) throws IOException {
        super.read(dr);
        fRotation = dr.readInt();
    }
}

```

Clase TriangleRotationHandle

Un Handle para rotar una TriangleFigure.

Se encuentra en el fichero TriangleRotationHandle.java y su contenido es el siguiente:

```

/*
 * Sun Mar  2 19:15:28 1997  Doug Lea  (dl at gee)
 * Based on RadiusHandle
 */

package CH.ifa.draw.contrib;

import java.awt.*;
import java.util.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.standard.*;

/**
 * A Handle to rotate a TriangleFigure
 */
class TriangleRotationHandle extends AbstractHandle {

    private Point fOrigin = null;
    private Point fCenter = null;

    public TriangleRotationHandle(TriangleFigure owner) {
        super(owner);
    }

    public void invokeStart(int x, int y, Drawing drawing) {
        fCenter = owner().center();
        fOrigin = getOrigin();
    }
}

```



```

public void invokeStep (int dx, int dy, Drawing drawing) {
    double angle = Math.atan2(fOrigin.y + dy - fCenter.y,
                              fOrigin.x + dx - fCenter.x);
    ((TriangleFigure)(owner())).rotate(angle);
}

public void invokeEnd (int dx, int dy, Drawing drawing) {
    fOrigin = null;
    fCenter = null;
}

public Point locate() {
    return getOrigin();
}

Point getOrigin() { // find a nice place to put handle
    // almost same code as PolygonScaleHandle
    Polygon p = ((TriangleFigure)(owner())).polygon();
    Point first = new Point(p.xpoints[0], p.ypoints[0]);
    Point ctr = owner().center();
    double len = Geom.length(first.x, first.y, ctr.x, ctr.y);
    if (len == 0) // best we can do?
        return new Point(first.x - HANDLESIZE/2, first.y +
HANDLESIZE/2);

    double u = HANDLESIZE / len;
    if (u > 1.0) // best we can do?
        return new Point((first.x * 3 + ctr.x)/4, (first.y * 3 +
ctr.y)/4);
    else
        return new Point((int)(first.x * (1.0 - u) + ctr.x * u),
                          (int)(first.y * (1.0 - u) + ctr.y * u));
}

public void draw(Graphics g) {
    Rectangle r = displayBox();

    g.setColor(Color.yellow);
    g.fillOval(r.x, r.y, r.width, r.height);

    g.setColor(Color.black);
    g.drawOval(r.x, r.y, r.width, r.height);
}
}

```


Paquete CH.ifa.draw.applet

El paquete applet define un applet por defecto con interface de usuario para un applet JHotDraw. Los applets JHotDraw pueden ser desarrollados sin utilizar este paquete y pueden tener un interface de usuario diferente.

Este paquete contiene lo siguiente:

- Clase DrawApplet.

Clase DrawApplet

DrawApplication define una representación estándar para un editor de dibujo que se ejecuta como applet. La representación se realiza a medida en las subclases.

Soporta parámetros en el applet:

DRAWINGS: Una lista separada con espacios en blanco de los nombres de los dibujos que son mostrados en el choice del drawing.

Se encuentra en el fichero DrawApplet.java y su contenido es el siguiente:

```
/*
 * @(#)DrawApplet.java 5.1
 *
 */

package CH.ifa.draw.applet;

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.net.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.util.*;

/**
 * DraaApplication defines a standard presentation for
 * a drawing editor that is run as an applet. The presentation is
 * customized in subclasses.<p>
 * Supported applet parameters: <br>
 * <i>DRAWINGS</i>: a blank separated list of drawing names that is
 * shown in the drawings choice.
 */
```

```

public class DrawApplet
    extends Applet
    implements DrawingEditor, PaletteListener {

    transient private Drawing      fDrawing;
    transient private Tool         fTool;

    transient private StandardDrawingView fView;
    transient private ToolButton    fDefaultToolButton;
    transient private ToolButton    fSelectedToolButton;

    transient private boolean       fSimpleUpdate;
    transient private Button        fUpdateButton;

    transient private Choice        fFrameColor;
    transient private Choice        fFillColor;
    //transient private Choice      fTextColor;
    transient private Choice        fArrowChoice;
    transient private Choice        fFontChoice;

    transient private Thread        fSleeper;
    private IconKit                 fIconkit;

    static String                   fgUntitled = "untitled";

    private static final String     fgDrawPath = "/CH/ifa/draw/";
    public static final String      IMAGES = fgDrawPath+"images/";

    /**
     * Initializes the applet and creates its contents.
     */
    public void init() {
        fIconkit = new IconKit(this);

        setLayout(new BorderLayout());

        fView = createDrawingView();

        Panel attributes = createAttributesPanel();
        createAttributeChoices(attributes);
        add("North", attributes);

        Panel toolPanel = createToolPalette();
        createTools(toolPanel);
        add("West", toolPanel);

        add("Center", fView);
        Panel buttonPalette = createButtonPanel();
        createButtons(buttonPalette);
        add("South", buttonPalette);

        initDrawing();
        setBufferedDisplayUpdate();
        setupAttributes();
    }

    /*

```

```

    * Gets the iconkit to be used in the applet.
    */

    /**
     * **** not sure whether this will still be needed on 1.1 enabled
     browsers
     */
    protected Iconkit iconkit() {
        if (fIconkit == null) {

            startSleeper();
            loadAllImages(this); // blocks until images loaded
            stopSleeper();
        }
        return fIconkit;
    }

    /**
     * Creates the attributes panel.
     */
    protected Panel createAttributesPanel() {
        Panel panel = new Panel();
        panel.setLayout(new PaletteLayout(2, new Point(2,2), false));
        return panel;
    }

    /**
     * Creates the attribute choices. Override to add additional
     choices.
     */
    protected void createAttributeChoices(Panel panel) {
        panel.add(new Label("Fill"));
        fFillColor = createColorChoice("FillColor");
        panel.add(fFillColor);

        //panel.add(new Label("Text"));
        //fTextColor = createColorChoice("TextColor");
        //panel.add(fTextColor);

        panel.add(new Label("Pen"));
        fFrameColor = createColorChoice("FrameColor");
        panel.add(fFrameColor);

        panel.add(new Label("Arrow"));
        CommandChoice choice = new CommandChoice();
        fArrowChoice = choice;
        choice.addItem(new ChangeAttributeCommand("none",
"ArrowMode", new Integer(PolyLineFigure.ARROW_TIP_NONE), fView));
        choice.addItem(new ChangeAttributeCommand("at Start",
"ArrowMode", new Integer(PolyLineFigure.ARROW_TIP_START), fView));
        choice.addItem(new ChangeAttributeCommand("at End",
"ArrowMode", new Integer(PolyLineFigure.ARROW_TIP_END), fView));
        choice.addItem(new ChangeAttributeCommand("at Both",
"ArrowMode", new Integer(PolyLineFigure.ARROW_TIP_BOTH), fView));
        panel.add(fArrowChoice);

        panel.add(new Label("Font"));
        fFontChoice = createFontChoice();
        panel.add(fFontChoice);
    }

```

```

/**
 * Creates the color choice for the given attribute.
 */
protected Choice createColorChoice(String attribute) {
    CommandChoice choice = new CommandChoice();
    for (int i=0; i<ColorMap.size(); i++)
        choice.addItem(
            new ChangeAttributeCommand(
                ColorMap.name(i),
                attribute,
                ColorMap.color(i),
                fView
            )
        );
    return choice;
}

/**
 * Creates the font choice. The choice is filled with
 * all the fonts supported by the toolkit.
 */
protected Choice createFontChoice() {
    CommandChoice choice = new CommandChoice();
    String fonts[] = Toolkit.getDefaultToolkit().getFontList();
    for (int i = 0; i < fonts.length; i++)
        choice.addItem(new ChangeAttributeCommand(fonts[i],
"FontName", fonts[i], fView));
    return choice;
}

/**
 * Creates the buttons panel.
 */
protected Panel createButtonPanel() {
    Panel panel = new Panel();
    panel.setLayout(new PaletteLayout(2, new Point(2,2), false));
    return panel;
}

/**
 * Creates the buttons shown in the buttons panel. Override to
 * add additional buttons.
 * @param panel the buttons panel.
 */
protected void createButtons(Panel panel) {
    panel.add(new Filler(24,20));

    Choice drawingChoice = new Choice();
    drawingChoice.addItem(fgUntitled);

    String param = getParameter("DRAWINGS");
    if (param == null)
        param = "";
    StringTokenizer st = new StringTokenizer(param);
    while (st.hasMoreTokens())
        drawingChoice.addItem(st.nextToken());
    // offer choice only if more than one

```

```

        if (drawingChoice.getItemCount() > 1)
            panel.add(drawingChoice);
        else
            panel.add(new Label(fgUntitled));

        drawingChoice.addItemListener(
            new ItemListener() {
                public void itemStateChanged(ItemEvent e) {
                    if (e.getStateChange() == ItemEvent.SELECTED) {
                        loadDrawing((String)e.getItem());
                    }
                }
            }
        );

        panel.add(new Filler(6,20));

        Button button;
        button = new CommandButton(new DeleteCommand("Delete",
fView));
        panel.add(button);

        button = new CommandButton(new DuplicateCommand("Duplicate",
fView));
        panel.add(button);

        button = new CommandButton(new GroupCommand("Group", fView));
        panel.add(button);

        button = new CommandButton(new UngroupCommand("Ungroup",
fView));
        panel.add(button);

        button = new Button("Help");
        button.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    showHelp();
                }
            }
        );
        panel.add(button);

        fUpdateButton = new Button("Simple Update");
        fUpdateButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    if (fSimpleUpdate)
                        setBufferedDisplayUpdate();
                    else
                        setSimpleDisplayUpdate();
                }
            }
        );

        // panel.add(fUpdateButton); // not shown currently
    }

```

```

/**
 * Creates the tools palette.
 */
protected Panel createToolPalette() {
    Panel palette = new Panel();
    palette.setLayout(new PaletteLayout(2,new Point(2,2)));
    return palette;
}

/**
 * Creates the tools. By default only the selection tool is added.
 * Override this method to add additional tools.
 * Call the inherited method to include the selection tool.
 * @param palette the palette where the tools are added.
 */
protected void createTools(Panel palette) {
    Tool tool = createSelectionTool();

    fDefaultToolButton = createToolButton(IMAGES+"SEL", "Selection
Tool", tool);
    palette.add(fDefaultToolButton);
}

/**
 * Creates the selection tool used in this editor. Override to use
 * a custom selection tool.
 */
protected Tool createSelectionTool() {
    return new SelectionTool(view());
}

/**
 * Creates a tool button with the given image, tool, and text
 */
protected ToolButton createToolButton(String iconName, String
toolName, Tool tool) {
    return new ToolButton(this, iconName, toolName, tool);
}

/**
 * Creates the drawing used in this application.
 * You need to override this method to use a Drawing
 * subclass in your application. By default a standard
 * Drawing is returned.
 */
protected Drawing createDrawing() {
    return new StandardDrawing();
}

/**
 * Creates the drawing view used in this application.
 * You need to override this method to use a DrawingView
 * subclass in your application. By default a standard
 * DrawingView is returned.
 */
protected StandardDrawingView createDrawingView() {
    return new StandardDrawingView(this, 410, 370);
}

```



```
/**
 * Handles a user selection in the palette.
 * @see PaletteListener
 */
public void paletteUserSelected(PaletteButton button) {
    ToolButton toolButton = (ToolButton) button;
    setTool(toolButton.tool(), toolButton.name());
    setSelected(toolButton);
}

/**
 * Handles when the mouse enters or leaves a palette button.
 * @see PaletteListener
 */
public void paletteUserOver(PaletteButton button, boolean inside)
{
    if (inside)
        showStatus(((ToolButton) button).name());
    else
        showStatus(fSelectedToolButton.name());
}

/**
 * Gets the current drawing.
 * @see DrawingEditor
 */
public Drawing drawing() {
    return fDrawing;
}

/**
 * Gets the current tool.
 * @see DrawingEditor
 */
public Tool tool() {
    return fTool;
}

/**
 * Gets the current drawing view.
 * @see DrawingEditor
 */
public DrawingView view() {
    return fView;
}

/**
 * Sets the default tool of the editor.
 * @see DrawingEditor
 */
public void toolDone() {
    setTool(fDefaultToolButton.tool(), fDefaultToolButton.name());
    setSelected(fDefaultToolButton);
}

/**
 * Handles a change of the current selection. Updates all
```

```

    * menu items that are selection sensitive.
    * @see DrawingEditor
    */
    public void selectionChanged(DrawingView view) {
        setupAttributes();
    }

    private void initDrawing() {
        fDrawing = createDrawing();
        fView.setDrawing(fDrawing);
        toolDone();
    }

    private void setTool(Tool t, String name) {
        if (fTool != null)
            fTool.deactivate();
        fTool = t;
        if (fTool != null) {
            showStatus(name);
            fTool.activate();
        }
    }

    private void setSelected(ToolButton button) {
        if (fSelectedToolButton != null)
            fSelectedToolButton.reset();
        fSelectedToolButton = button;
        if (fSelectedToolButton != null)
            fSelectedToolButton.select();
    }

    protected void loadDrawing(String param) {
        if (param == fgUntitled) {
            fDrawing.release();
            initDrawing();
            return;
        }

        String filename = getParameter(param);
        if (filename != null)
            readDrawing(filename);
    }

    private void readDrawing(String filename) {
        toolDone();
        String type = guessType(filename);
        if (type.equals("storable"))
            readFromStorableInput(filename);
        else if (type.equals("serialized"))
            readFromObjectInput(filename);
        else
            showStatus("Unknown file type");
    }

    private void readFromStorableInput(String filename) {
        try {
            URL url = new URL(getCodeBase(), filename);
            InputStream stream = url.openStream();

```

```

        StorableInput input = new StorableInput(stream);
        fDrawing.release();

        fDrawing = (Drawing)input.readStorable();
        fView.setDrawing(fDrawing);
    } catch (IOException e) {
        initDrawing();
        showStatus("Error:" + e);
    }
}

private void readFromObjectInput(String filename) {
    try {
        URL url = new URL(getCodeBase(), filename);
        InputStream stream = url.openStream();
        ObjectInput input = new ObjectInputStream(stream);
        fDrawing.release();
        fDrawing = (Drawing)input.readObject();
        fView.setDrawing(fDrawing);
    } catch (IOException e) {
        initDrawing();
        showStatus("Error: " + e);
    } catch (ClassNotFoundException e) {
        initDrawing();
        showStatus("Class not found: " + e);
    }
}

private String guessType(String file) {
    if (file.endsWith(".draw"))
        return "storable";
    if (file.endsWith(".ser"))
        return "serialized";
    return "unknown";
}

private void setupAttributes() {
    Color frameColor = (Color)
AttributeFigure.getDefaultAttribute("FrameColor");
    Color fillColor = (Color)
AttributeFigure.getDefaultAttribute("FillColor");
    Color textColor = (Color)
AttributeFigure.getDefaultAttribute("TextColor");
    Integer arrowMode = (Integer)
AttributeFigure.getDefaultAttribute("ArrowMode");
    String fontName = (String)
AttributeFigure.getDefaultAttribute("FontName");

    FigureEnumeration k = fView.selectionElements();
    while (k.hasMoreElements()) {
        Figure f = k.nextFigure();
        frameColor = (Color) f.getAttribute("FrameColor");
        fillColor = (Color) f.getAttribute("FillColor");
        textColor = (Color) f.getAttribute("TextColor");
        arrowMode = (Integer) f.getAttribute("ArrowMode");
        fontName = (String) f.getAttribute("FontName");
    }
}

```

```

        fFrameColor.select(ColorMap.colorIndex(frameColor));
        fFillColor.select(ColorMap.colorIndex(fillColor));
        //fTextColor.select(ColorMap.colorIndex(textColor));
        if (arrowMode != null)
            fArrowChoice.select(arrowMode.intValue());
        if (fontName != null)
            fFontChoice.select(fontName);
    }

    protected void setSimpleDisplayUpdate() {
        fView.setDisplayUpdate(new SimpleUpdateStrategy());
        fUpdateButton.setLabel("Simple Update");
        fSimpleUpdate = true;
    }

    protected void setBufferedDisplayUpdate() {
        fView.setDisplayUpdate(new BufferedUpdateStrategy());
        fUpdateButton.setLabel("Buffered Update");
        fSimpleUpdate = false;
    }

    /**
     * Shows a help page for the applet. The URL of the help
     * page is derived as follows: codeBase + appletClassname +
     * + "Help.html"
     */
    protected void showHelp() {
        try {
            String appletPath = getClass().getName().replace('.',
'/'');
            URL url = new URL(getCodeBase(), appletPath+"Help.html");
            getAppletContext().showDocument(url, "Help");
        } catch (IOException e) {
            showStatus("Help file not found");
        }
    }

    /**
     * *** netscape browser work around ***
     */
    private void startSleeper() {
        if (fSleeper == null)
            fSleeper = new SleeperThread(this);
        fSleeper.start();
    }

    private void stopSleeper() {
        if (fSleeper != null)
            fSleeper.stop();
    }
}

class SleeperThread extends Thread {

    Applet fApplet;

```

```
SleeperThread(Applet applet) {
    fApplet = applet;
}

public void run() {
    try {
        for (;;) {
            fApplet.showStatus("loading icons...");
            sleep(50);
        }
    } catch (InterruptedException e) {
        return;
    }
}
}
```


Paquete CH.ifa.draw.application

El paquete application define un interface de usuario por defecto para aplicaciones JHotDraw. Esto es solamente una presentación por defecto.

Este paquete contiene lo siguiente:

- Clase DrawApplication.

Clase DrawApplication

DrawApplication define una representación estándar para un editor de dibujo. La representación se realiza en las subclases.

La aplicación se empieza como sigue:

```
public static void main (String[] args)
{
    MyDrawApp window= new MyDrawApp();
    window.open();
}
```

Se encuentra en el fichero DrawApplication.java y su contenido es el siguiente:

```
/*
 * @(#)DrawApplication.java 5.1
 *
 */

package CH.ifa.draw.application;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.util.*;

/**
 * DrawApplication defines a standard presentation for
 * standalone drawing editors. The presentation is
 * customized in subclasses.
 * The application is started as follows:
 * <pre>
 * public static void main(String[] args) {
 *     MayDrawApp window = new MyDrawApp();
 */
```

```

*      window.open();
*  }
* </pre>
*/

public class DrawApplication
    extends Frame
    implements DrawingEditor, PaletteListener {

    private Drawing          fDrawing;
    private Tool             fTool;
    private Iconkit          fIconkit;

    private TextField        fStatusLine;
    private StandardDrawingView fView;
    private ToolButton       fDefaultToolButton;
    private ToolButton       fSelectedToolButton;

    private String           fDrawingFilename;
    static String            fgUntitled = "untitled";

    // the image resource path
    private static final String fgDrawPath = "/CH/ifa/draw/";
    public static final String IMAGES = fgDrawPath+"images/";

    /**
     * The index of the file menu in the menu bar.
     */
    public static final int    FILE_MENU = 0;
    /**
     * The index of the edit menu in the menu bar.
     */
    public static final int    EDIT_MENU = 1;
    /**
     * The index of the alignment menu in the menu bar.
     */
    public static final int    ALIGNMENT_MENU = 2;
    /**
     * The index of the attributes menu in the menu bar.
     */
    public static final int    ATTRIBUTES_MENU = 3;

    /**
     * Constructs a drawing window with a default title.
     */
    public DrawApplication() {
        super("JHotDraw");
    }

    /**
     * Constructs a drawing window with the given title.
     */
    public DrawApplication(String title) {
        super(title);
    }

    /**

```



```

    * Opens the window and initializes its contents.
    * Clients usually only call but don't override it.
    */

    public void open() {
        fIconkit = new Iconkit(this);
        setLayout(new BorderLayout());

        fView = createDrawingView();
        Component contents = createContents(fView);
        add("Center", contents);
        //add("Center", fView);

        Panel tools = createToolPalette();
        createTools(tools);
        add("West", tools);

        fStatusLine = createStatusLine();
        add("South", fStatusLine);

        MenuBar mb = new MenuBar();
        createMenus(mb);
        setMenuBar(mb);

        initDrawing();
        Dimension d = defaultSize();
        setSize(d.width, d.height);

        addListeners();

        show();
    }

    /**
     * Registers the listeners for this window
     */
    protected void addListeners() {
        addWindowListener(
            new WindowAdapter() {
                public void windowClosing(WindowEvent event) {
                    exit();
                }
            }
        );
    }

    private void initDrawing() {
        fDrawing = createDrawing();
        fDrawingFilename = fgUntitled;
        fView.setDrawing(fDrawing);
        toolDone();
    }

    /**
     * Creates the standard menus. Clients override this
     * method to add additional menus.
     */
    protected void createMenus(MenuBar mb) {

```

```

        mb.add(createFileMenu());
        mb.add(createEditMenu());
        mb.add(createAlignmentMenu());
        mb.add(createAttributesMenu());
        mb.add(createDebugMenu());
    }

    /**
     * Creates the file menu. Clients override this
     * method to add additional menu items.
     */
    protected Menu createFileMenu() {
        Menu menu = new Menu("File");
        MenuItem mi = new MenuItem("New", new MenuShortcut('n'));
        mi.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    promptNew();
                }
            }
        );
        menu.add(mi);

        mi = new MenuItem("Open...", new MenuShortcut('o'));
        mi.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    promptOpen();
                }
            }
        );
        menu.add(mi);

        mi = new MenuItem("Save As...", new MenuShortcut('s'));
        mi.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    promptSaveAs();
                }
            }
        );
        menu.add(mi);

        mi = new MenuItem("Save As Serialized...");
        mi.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    promptSaveAsSerialized();
                }
            }
        );
        menu.add(mi);
        menu.addSeparator();
        mi = new MenuItem("Print...", new MenuShortcut('p'));
        mi.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    print();
                }
            }
        );
    }

```

```

        }
    }
    );
    menu.add(mi);
    menu.addSeparator();
    mi = new MenuItem("Exit");
    mi.addActionListener(
        new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                exit();
            }
        }
    );
    menu.add(mi);
    return menu;
}

/**
 * Creates the edit menu. Clients override this
 * method to add additional menu items.
 */
protected Menu createEditMenu() {
    CommandMenu menu = new CommandMenu("Edit");
    menu.add(new CutCommand("Cut", fView), new MenuShortcut('x'));
    menu.add(new CopyCommand("Copy", fView), new MenuShortcut('c'));
    menu.add(new PasteCommand("Paste", fView), new
MenuShortcut('v'));
    menu.addSeparator();
    menu.add(new DuplicateCommand("Duplicate", fView), new
MenuShortcut('d'));
    menu.add(new DeleteCommand("Delete", fView));
    menu.addSeparator();
    menu.add(new GroupCommand("Group", fView));
    menu.add(new UngroupCommand("Ungroup", fView));
    menu.addSeparator();
    menu.add(new SendToBackCommand("Send to Back", fView));
    menu.add(new BringToFrontCommand("Bring to Front", fView));
    return menu;
}

/**
 * Creates the alignment menu. Clients override this
 * method to add additional menu items.
 */
protected Menu createAlignmentMenu() {
    CommandMenu menu = new CommandMenu("Align");
    menu.add(new ToggleGridCommand("Toggle Snap to Grid", fView, new
Point(4,4)));
    menu.addSeparator();
    menu.add(new AlignCommand("Lefts", fView, AlignCommand.LEFTS));
    menu.add(new AlignCommand("Centers", fView,
AlignCommand.CENTERS));
    menu.add(new AlignCommand("Rights", fView,
AlignCommand.RIGHTS));
    menu.addSeparator();
    menu.add(new AlignCommand("Tops", fView, AlignCommand.TOPS));
    menu.add(new AlignCommand("Middles", fView,
AlignCommand.MIDDLES));
}

```

```

        menu.add(new AlignCommand("Bottoms", fView,
AlignCommand.BOTTOMS));
        return menu;
    }

    /**
     * Creates the debug menu. Clients override this
     * method to add additional menu items.
     */
    protected Menu createDebugMenu() {
        Menu menu = new Menu("Debug");

        MenuItem mi = new MenuItem("Simple Update");
        mi.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    fView.setDisplayUpdate(new SimpleUpdateStrategy());
                }
            }
        );
        menu.add(mi);

        mi = new MenuItem("Buffered Update");
        mi.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    fView.setDisplayUpdate(new
BufferedUpdateStrategy());
                }
            }
        );
        menu.add(mi);
        return menu;
    }

    /**
     * Creates the attributes menu and its submenus. Clients override
     * this method to add additional menu items.
     */
    protected Menu createAttributesMenu() {
        Menu menu = new Menu("Attributes");
        menu.add(createColorMenu("Fill Color", "FillColor"));
        menu.add(createColorMenu("Pen Color", "FrameColor"));
        menu.add(createArrowMenu());
        menu.addSeparator();
        menu.add(createFontMenu());
        menu.add(createFontSizeMenu());
        menu.add(createFontStyleMenu());
        menu.add(createColorMenu("Text Color", "TextColor"));
        return menu;
    }

    /**
     * Creates the color menu.
     */
    protected Menu createColorMenu(String title, String attribute) {
        CommandMenu menu = new CommandMenu(title);
        for (int i=0; i<ColorMap.size(); i++)

```

```

        menu.add(
            new ChangeAttributeCommand(
                ColorMap.name(i),
                attribute,
                ColorMap.color(i),
                fView
            )
        );
    }
    return menu;
}

/**
 * Creates the arrows menu.
 */
protected Menu createArrowMenu() {
    CommandMenu menu = new CommandMenu("Arrow");
    menu.add(new ChangeAttributeCommand("none", "ArrowMode",
new Integer(PolyLineFigure.ARROW_TIP_NONE), fView));
    menu.add(new ChangeAttributeCommand("at Start", "ArrowMode",
new Integer(PolyLineFigure.ARROW_TIP_START), fView));
    menu.add(new ChangeAttributeCommand("at End", "ArrowMode",
new Integer(PolyLineFigure.ARROW_TIP_END), fView));
    menu.add(new ChangeAttributeCommand("at Both", "ArrowMode",
new Integer(PolyLineFigure.ARROW_TIP_BOTH), fView));
    return menu;
}

/**
 * Creates the fonts menus. It installs all available fonts
 * supported by the toolkit implementation.
 */
protected Menu createFontMenu() {
    CommandMenu menu = new CommandMenu("Font");
    String fonts[] = Toolkit.getDefaultToolkit().getFontList();
    for (int i = 0; i < fonts.length; i++)
        menu.add(new ChangeAttributeCommand(fonts[i], "FontName",
fonts[i], fView));
    return menu;
}

/**
 * Creates the font style menu with entries (Plain, Italic, Bold).
 */
protected Menu createFontStyleMenu() {
    CommandMenu menu = new CommandMenu("Font Style");
    menu.add(new ChangeAttributeCommand("Plain", "FontStyle", new
Integer(Font.PLAIN), fView));
    menu.add(new ChangeAttributeCommand("Italic", "FontStyle", new
Integer(Font.ITALIC), fView));
    menu.add(new ChangeAttributeCommand("Bold", "FontStyle", new
Integer(Font.BOLD), fView));
    return menu;
}

/**
 * Creates the font size menu.
 */
protected Menu createFontSizeMenu() {

```

```

        CommandMenu menu = new CommandMenu("Font Size");
        int sizes[] = { 9, 10, 12, 14, 18, 24, 36, 48, 72 };
        for (int i = 0; i < sizes.length; i++) {
            menu.add(
                new ChangeAttributeCommand(
                    Integer.toString(sizes[i]),
                    "FontSize",
                    new Integer(sizes[i]), fView)
            );
        }
        return menu;
    }

    /**
     * Creates the tool palette.
     */
    protected Panel createToolPalette() {
        Panel palette = new Panel();
        palette.setBackground(Color.lightGray);
        palette.setLayout(new PaletteLayout(2, new Point(2, 2)));
        return palette;
    }

    /**
     * Creates the tools. By default only the selection tool is added.
     * Override this method to add additional tools.
     * Call the inherited method to include the selection tool.
     * @param palette the palette where the tools are added.
     */
    protected void createTools(Panel palette) {
        Tool tool = createSelectionTool();

        fDefaultToolButton = createToolButton(IMAGES+"SEL", "Selection
Tool", tool);
        palette.add(fDefaultToolButton);
    }

    /**
     * Creates the selection tool used in this editor. Override to use
     * a custom selection tool.
     */
    protected Tool createSelectionTool() {
        return new SelectionTool(view());
    }

    /**
     * Creates a tool button with the given image, tool, and text
     */
    protected ToolButton createToolButton(String iconName, String
toolName, Tool tool) {
        return new ToolButton(this, iconName, toolName, tool);
    }

    /**
     * Creates the drawing view used in this application.
     * You need to override this method to use a DrawingView
     * subclass in your application. By default a standard
     * DrawingView is returned.

```

```

    */
    protected StandardDrawingView createDrawingView() {
        Dimension d = getDrawingViewSize();
        return new StandardDrawingView(this, d.width, d.height);
    }

    /**
     * Override to define the dimensions of the drawing view.
     */
    protected Dimension getDrawingViewSize() {
        return new Dimension(400, 600);
    }

    /**
     * Creates the drawing used in this application.
     * You need to override this method to use a Drawing
     * subclass in your application. By default a standard
     * Drawing is returned.
     */
    protected Drawing createDrawing() {
        return new StandardDrawing();
    }

    /**
     * Creates the contents component of the application
     * frame. By default the DrawingView is returned in
     * a ScrollPane.
     */
    protected Component createContents(StandardDrawingView view) {
        ScrollPane sp = new ScrollPane();
        Adjustable vadjust = sp.getVAdjustable();
        Adjustable hadjust = sp.getHAdjustable();
        hadjust.setUnitIncrement(16);
        vadjust.setUnitIncrement(16);

        sp.add(view);
        return sp;
    }

    /**
     * Sets the drawing to be edited.
     */
    public void setDrawing(Drawing drawing) {
        fView.setDrawing(drawing);
        fDrawing = drawing;
    }

    /**
     * Gets the default size of the window.
     */
    protected Dimension defaultSize() {
        return new Dimension(430, 406);
    }

    /**
     * Creates the status line.
     */
    protected TextField createStatusLine() {

```

```

        TextField field = new TextField("No Tool", 40);
        field.setEditable(false);
        return field;
    }

    /**
     * Handles a user selection in the palette.
     * @see PaletteListener
     */
    public void paletteUserSelected(PaletteButton button) {
        ToolButton toolButton = (ToolButton) button;
        setTool(toolButton.tool(), toolButton.name());
        setSelected(toolButton);
    }

    /**
     * Handles when the mouse enters or leaves a palette button.
     * @see PaletteListener
     */
    public void paletteUserOver(PaletteButton button, boolean inside)
    {
        ToolButton toolButton = (ToolButton) button;
        if (inside)
            showStatus(toolButton.name());
        else
            showStatus(fSelectedToolButton.name());
    }

    /**
     * Gets the current drawing.
     * @see DrawingEditor
     */
    public Drawing drawing() {
        return fDrawing;
    }

    /**
     * Gets the current tool.
     * @see DrawingEditor
     */
    public Tool tool() {
        return fTool;
    }

    /**
     * Gets the current drawing view.
     * @see DrawingEditor
     */
    public DrawingView view() {
        return fView;
    }

    /**
     * Sets the default tool of the editor.
     * @see DrawingEditor
     */
    public void toolDone() {
        if (fDefaultToolButton != null) {

```



```

        setTool(fDefaultToolButton.tool(),
fDefaultToolButton.name());
        setSelected(fDefaultToolButton);
    }
}

/**
 * Handles a change of the current selection. Updates all
 * menu items that are selection sensitive.
 * @see DrawingEditor
 */
public void selectionChanged(DrawingView view) {
    MenuBar mb = getMenuBar();
    CommandMenu editMenu = (CommandMenu)mb.getMenu(EDIT_MENU);
    editMenu.setEnabled();
    CommandMenu alignmentMenu =
(CommandMenu)mb.getMenu(ALIGNMENT_MENU);
    alignmentMenu.setEnabled();
}

/**
 * Shows a status message.
 * @see DrawingEditor
 */
public void showStatus(String string) {
    fStatusLine.setText(string);
}

private void setTool(Tool t, String name) {
    if (fTool != null)
        fTool.deactivate();
    fTool = t;
    if (fTool != null) {
        fStatusLine.setText(name);
        fTool.activate();
    }
}

private void setSelected(ToolButton button) {
    if (fSelectedToolButton != null)
        fSelectedToolButton.reset();
    fSelectedToolButton = button;
    if (fSelectedToolButton != null)
        fSelectedToolButton.select();
}

/**
 * Exits the application. You should never override this method
 */
public void exit() {
    destroy();
    setVisible(false); // hide the Frame
    dispose(); // tell windowing system to free resources
    System.exit(0);
}

/**
 * Handles additional clean up operations. Override to destroy

```

```

        * or release drawing editor resources.
        */
protected void destroy() {
}

/**
 * Resets the drawing to a new empty drawing.
 */
public void promptNew() {
    initDrawing();
}

/**
 * Shows a file dialog and opens a drawing.
 */
public void promptOpen() {
    FileDialog dialog = new FileDialog(this, "Open File...",
FileDialog.LOAD);
    dialog.show();
    String filename = dialog.getFile();
    if (filename != null) {
        filename = stripTrailingAsterisks(filename);
        String dirname = dialog.getDirectory();
        loadDrawing(dirname + filename);
    }
    dialog.dispose();
}

/**
 * Shows a file dialog and saves drawing.
 */
public void promptSaveAs() {
    toolDone();
    String path = getSavePath("Save File...");
    if (path != null) {
        if (!path.endsWith(".draw"))
            path += ".draw";
        saveAsStorableOutput(path);
    }
}

/**
 * Shows a file dialog and saves drawing.
 */
public void promptSaveAsSerialized() {
    toolDone();
    String path = getSavePath("Save File...");
    if (path != null) {
        if (!path.endsWith(".ser"))
            path += ".ser";
        saveAsObjectOutput(path);
    }
}

/**
 * Prints the drawing.
 */
public void print() {

```

```

        fTool.deactivate();
        PrintJob printJob = getToolkit().getPrintJob(this, "Print
Drawing", null);

        if (printJob != null) {
            Graphics pg = printJob.getGraphics();

            if (pg != null) {
                fView.printAll(pg);
                pg.dispose(); // flush page
            }
            printJob.end();
        }
        fTool.activate();
    }

    private String getSavePath(String title) {
        String path = null;
        FileDialog dialog = new FileDialog(this, title,
FileDialog.SAVE);
        dialog.show();
        String filename = dialog.getFile();
        if (filename != null) {
            filename = stripTrailingAsterisks(filename);
            String dirname = dialog.getDirectory();
            path = dirname + filename;
        }
        dialog.dispose();
        return path;
    }

    private String stripTrailingAsterisks(String filename) {
        // workaround for bug on NT
        if (filename.endsWith("*. *"))
            return filename.substring(0, filename.length() - 4);
        else
            return filename;
    }

    private void saveAsStorableOutput(String file) {
        // TBD: should write a MIME header
        try {
            FileOutputStream stream = new FileOutputStream(file);
            StorableOutput output = new StorableOutput(stream);
            output.writeStorable(fDrawing);
            output.close();
        } catch (IOException e) {
            showStatus(e.toString());
        }
    }

    private void saveAsObjectOutput(String file) {
        // TBD: should write a MIME header
        try {
            FileOutputStream stream = new FileOutputStream(file);
            ObjectOutputStream output = new ObjectOutputStream(stream);
            output.writeObject(fDrawing);
            output.close();
        }
    }

```

```

    } catch (IOException e) {
        showStatus(e.toString());
    }
}

private void loadDrawing(String file) {
    toolDone();
    String type = guessType(file);
    if (type.equals("storable"))
        readFromStorableInput(file);
    else if (type.equals("serialized"))
        readFromObjectInput(file);
    else
        showStatus("Unknown file type");
}

private void readFromStorableInput(String file) {
    try {
        FileInputStream stream = new FileInputStream(file);
        StorableInput input = new StorableInput(stream);
        fDrawing.release();
        fDrawing = (Drawing)input.readStorable();
        fView.setDrawing(fDrawing);
    } catch (IOException e) {
        initDrawing();
        showStatus("Error: " + e);
    }
}

private void readFromObjectInput(String file) {
    try {
        FileInputStream stream = new FileInputStream(file);
        ObjectInput input = new ObjectInputStream(stream);
        fDrawing.release();
        fDrawing = (Drawing)input.readObject();
        fView.setDrawing(fDrawing);
    } catch (IOException e) {
        initDrawing();
        showStatus("Error: " + e);
    } catch (ClassNotFoundException e) {
        initDrawing();
        showStatus("Class not found: " + e);
    }
}

private String guessType(String file) {
    if (file.endsWith(".draw"))
        return "storable";
    if (file.endsWith(".ser"))
        return "serialized";
    return "unknown";
}
}

```

Paquete CH.ifa.draw.samples.javadraw

El paquete para las aplicaciones ejemplo javadraw. Este incluye el applet javadraw, la aplicación, y un applet viewer.

Este paquete contiene lo siguiente:

- Clase AnimationDecorator.
- Clase Animator.
- Clase BouncingDrawing.
- Clase FollowURLTool.
- Clase JavaDrawApp.
- Clase JavaDrawApplet.
- Clase JavaDrawViewer.
- Clase MySelectionTool.
- Clase PatternPainter.
- Clase URLTool.

Clase AnimationDecorator

Se encuentra en el fichero AnimationDecorator.java y su contenido es el siguiente:

```
/*
 * @(#)AnimationDecorator.java 5.1
 *
 */

package CH.ifa.draw.samples.javadraw;

import java.awt.*;
import java.util.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.*;

public class AnimationDecorator extends DecoratorFigure {

    private int fxVelocity;
    private int fyVelocity;

    /*
     * Serialization support.
     */
    private static final long serialVersionUID = 7894632974364110685L;
    private int animationDecoratorSerializedDataVersion = 1;
```

```

public AnimationDecorator() { }

public AnimationDecorator(Figure figure) {
    super(figure);
    fXVelocity = 4;
    fYVelocity = 4;
}

public void velocity(int xVelocity, int yVelocity) {
    fXVelocity = xVelocity;
    fYVelocity = yVelocity;
}

public Point velocity() {
    return new Point(fXVelocity, fYVelocity);
}

public void animationStep() {
    int xSpeed = fXVelocity;
    int ySpeed = fYVelocity;
    Rectangle box = displayBox();

    if ((box.x + box.width > 300) && (xSpeed > 0))
        xSpeed = -xSpeed;

    if ((box.y + box.height > 300) && (ySpeed > 0))
        ySpeed = -ySpeed;

    if ((box.x < 0) && (xSpeed < 0))
        xSpeed = -xSpeed;

    if ((box.y < 0) && (ySpeed < 0))
        ySpeed = -ySpeed;

    velocity(xSpeed, ySpeed);
    moveBy(xSpeed, ySpeed);
}

// guard concurrent access to display box

public synchronized void basicMoveBy(int x, int y) {
    super.basicMoveBy(x, y);
}

public synchronized void basicDisplayBox(Point origin, Point
corner) {
    super.basicDisplayBox(origin, corner);
}

public synchronized Rectangle displayBox() {
    return super.displayBox();
}

//-- store / load -----

public void write(StorableOutput dw) {
    super.write(dw);
}

```

```

        dw.writeInt(fXVelocity);
        dw.writeInt(fYVelocity);
    }

    public void read(StorableInput dr) throws IOException {
        super.read(dr);
        fXVelocity = dr.readInt();
        fYVelocity = dr.readInt();
    }
}

```

Clase Animator

Se encuentra en el fichero Animator.java y su contenido es el siguiente:

```

/*
 * @(#)Animator.java 5.1
 *
 */

package CH.ifa.draw.samples.javadraw;

import java.awt.*;
import java.util.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.Animatable;

public class Animator extends Thread {

    private DrawingView    fView;
    private Animatable     fAnimatable;

    private boolean        fIsRunning;
    private static final int    DELAY = 1000 / 16;

    public Animator(Animatable animatable, DrawingView view) {
        super("Animator");
        fView = view;
        fAnimatable = animatable;
    }

    public void start() {
        super.start();
        fIsRunning = true;
    }

    public void end() {
        fIsRunning = false;
    }

    public void run() {
        while (fIsRunning) {

```

```

        long tm = System.currentTimeMillis();
        fView.freezeView();
        fAnimatable.animationStep();
        fView.checkDamage();
        fView.unfreezeView();

        // Delay for a while
        try {
            tm += DELAY;
            Thread.sleep(Math.max(0, tm -
System.currentTimeMillis()));
        } catch (InterruptedException e) {
            break;
        }
    }
}
}

```

Clase BouncingDrawing

Se encuentra en el fichero BouncingDrawing.java y su contenido es el siguiente:

```

/*
 * @(#)BouncingDrawing.java 5.1
 *
 */

package CH.ifa.draw.samples.javadraw;

import java.awt.*;
import java.util.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.Animatable;

public class BouncingDrawing extends StandardDrawing implements
Animatable {
    /*
     * Serialization support.
     */
    private static final long serialVersionUID = -
8566272817418441758L;
    private int bouncingDrawingSerializedDataVersion = 1;

    public synchronized Figure add(Figure figure) {
        if (!(figure instanceof AnimationDecorator))
            figure = new AnimationDecorator(figure);
        return super.add(figure);
    }

    public synchronized Figure remove(Figure figure) {
        Figure f = super.remove(figure);
        if (f instanceof AnimationDecorator)

```



```

        return ((AnimationDecorator) f).peelDecoration();
    }
    return f;
}

public synchronized void replace(Figure figure, Figure
replacement) {
    if (!(replacement instanceof AnimationDecorator))
        replacement = new AnimationDecorator(replacement);
    super.replace(figure, replacement);
}

public void animationStep() {
    Enumeration k = figures();
    while (k.hasMoreElements())
        ((AnimationDecorator) k.nextElement()).animationStep();
}
}

```

Clase FollowURLTool

Se encuentra en el fichero FollowURLTool.java y su contenido es el siguiente:

```

/*
 * @(#)FollowURLTool.java 5.1
 *
 */

package CH.ifa.draw.samples.javadraw;

import java.awt.*;
import java.applet.*;
import java.awt.event.*;
import java.net.*;
import java.util.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.AbstractTool;

```

```

        fApplet.showStatus(urlstring);
    else
        fApplet.showStatus("");
    }

    /**
     * Handles mouse up in the drawing view.
     */
    public void mouseUp(MouseEvent e, int x, int y) {
        Figure figure = drawing().findFigureInside(x, y);
        if (figure == null)
            return;
        String urlstring = (String) figure.getAttribute("URL");
        if (urlstring == null)
            return;

        try {
            URL url = new URL(fApplet.getDocumentBase(), urlstring);
            fApplet.getAppletContext().showDocument(url);
        } catch (MalformedURLException exception) {
            fApplet.showStatus(exception.toString());
        }
    }
}

```

Clase JavaDrawApp

Se encuentra en el fichero JavaDrawApp.java y su contenido es el siguiente:

```

/*
 * @(#)JavaDrawApp.java 5.1
 *
 */

package CH.ifa.draw.samples.javadraw;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.application.*;
import CH.ifa.draw.contrib.*;

public class JavaDrawApp extends DrawApplication {

    private Animator          fAnimator;
    private static String      fgSampleImagesPath =
"CH/ifa/draw/samples/javadraw/sampleimages/";
    private static String      fgSampleImagesResourcePath =
"/"+fgSampleImagesPath;

```

```

JavaDrawApp() {
    super("JHotDraw");
}

public void open() {
    super.open();
}

//-- application life cycle -----

public void destroy() {
    super.destroy();
    endAnimation();
}

//-- DrawApplication overrides -----

protected void createTools(Palette palette) {
    super.createTools(palette);

    Tool tool = new TextTool(view(), new TextFigure());
    palette.add(createToolButton(IMAGES+"TEXT", "Text Tool",
tool));

    tool = new ConnectedTextTool(view(), new TextFigure());
    palette.add(createToolButton(IMAGES+"ATEXT", "Connected Text
Tool", tool));

    tool = new URLTool(view());
    palette.add(createToolButton(IMAGES+"URL", "URL Tool", tool));

    tool = new CreationTool(view(), new RectangleFigure());
    palette.add(createToolButton(IMAGES+"RECT", "Rectangle Tool",
tool));

    tool = new CreationTool(view(), new RoundRectangleFigure());
    palette.add(createToolButton(IMAGES+"RRECT", "Round Rectangle
Tool", tool));

    tool = new CreationTool(view(), new EllipseFigure());
    palette.add(createToolButton(IMAGES+"ELLIPSE", "Ellipse Tool",
tool));

    tool = new CreationTool(view(), new LineFigure());
    palette.add(createToolButton(IMAGES+"LINE", "Line Tool",
tool));

    tool = new ConnectionTool(view(), new LineConnection());
    palette.add(createToolButton(IMAGES+"CONN", "Connection Tool",
tool));

    tool = new ConnectionTool(view(), new ElbowConnection());
    palette.add(createToolButton(IMAGES+"OCONN", "Elbow Connection
Tool", tool));

    tool = new ScribbleTool(view());

```

```

        palette.add(createToolButton(IMAGES+"SCRIBBL", "Scribble
Tool", tool));

        tool = new PolygonTool(view());
        palette.add(createToolButton(IMAGES+"POLYGON", "Polygon Tool",
tool));

        tool = new BorderTool(view());
        palette.add(createToolButton(IMAGES+"BORDDEC", "Border Tool",
tool));
    }

    protected Tool createSelectionTool() {
        return new MySelectionTool(view());
    }

    protected void createMenus(MenuBar mb) {
        super.createMenus(mb);
        mb.add(createAnimationMenu());
        mb.add(createImagesMenu());
        mb.add(createWindowMenu());
    }

    protected Menu createAnimationMenu() {
        Menu menu = new Menu("Animation");
        MenuItem mi = new MenuItem("Start Animation");
        mi.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    startAnimation();
                }
            }
        );
        menu.add(mi);

        mi = new MenuItem("Stop Animation");
        mi.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    endAnimation();
                }
            }
        );
        menu.add(mi);
        return menu;
    }

    protected Menu createWindowMenu() {
        Menu menu = new Menu("Window");
        MenuItem mi = new MenuItem("New Window");
        mi.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    openView();
                }
            }
        );
        menu.add(mi);
    }

```

```

        return menu;
    }

    protected Menu createImagesMenu() {
        CommandMenu menu = new CommandMenu("Images");
        File imagesDirectory = new File(fgSampleImagesPath);
        try {
            String[] list = imagesDirectory.list();
            for (int i = 0; i < list.length; i++) {
                String name = list[i];
                String path = fgSampleImagesResourcePath+name;
                menu.add(new InsertImageCommand(name, path, view()));
            }
        } catch (Exception e) {}
        return menu;
    }

    protected Drawing createDrawing() {
        return new BouncingDrawing();
        //return new StandardDrawing();
    }

    //---- animation support -----

    public void startAnimation() {
        if (drawing() instanceof Animatable && fAnimator == null) {
            fAnimator = new Animator((Animatable)drawing(), view());
            fAnimator.start();
        }
    }

    public void endAnimation() {
        if (fAnimator != null) {
            fAnimator.end();
            fAnimator = null;
        }
    }

    public void openView() {
        JavaDrawApp window = new JavaDrawApp();
        window.open();
        window.setDrawing(drawing());
        window.setTitle("JHotDraw (View)");
    }

    //-- main -----

    public static void main(String[] args) {
        JavaDrawApp window = new JavaDrawApp();
        window.open();
    }
}

```

Clase JavaDrawApplet

Se encuentra en el fichero JavaDrawApplet.java y su contenido es el siguiente:

```

/*
 * @(#)JavaDrawApplet.java 5.1
 *
 */

package CH.ifa.draw.samples.javadraw;

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.net.*;

import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.applet.*;
import CH.ifa.draw.contrib.*;

public class JavaDrawApplet extends DrawApplet {

    transient private Button      fAnimationButton;
    transient private Animator    fAnimator;

    //-- applet life cycle -----

    public void destroy() {
        super.destroy();
        endAnimation();
    }

    //-- DrawApplet overrides -----

    protected void createTools(Palette palette) {
        super.createTools(palette);

        Tool tool = new TextTool(view(), new TextFigure());
        palette.add(createToolButton(IMGES+"TEXT", "Text Tool",
tool));

        tool = new ConnectedTextTool(view(), new TextFigure());
        palette.add(createToolButton(IMGES+"ATEXT", "Connected Text
Tool", tool));

        tool = new URLTool(view());
        palette.add(createToolButton(IMGES+"URL", "URL Tool", tool));

        tool = new CreationTool(view(), new RectangleFigure());
    }

```

```

        palette.add(createToolButton(IMAGES+"RECT", "Rectangle Tool",
tool));

        tool = new CreationTool(view(), new RoundRectangleFigure());
        palette.add(createToolButton(IMAGES+"RRECT", "Round Rectangle
Tool", tool));

        tool = new CreationTool(view(), new EllipseFigure());
        palette.add(createToolButton(IMAGES+"ELLIPSE", "Ellipse Tool",
tool));

        tool = new CreationTool(view(), new LineFigure());
        palette.add(createToolButton(IMAGES+"LINE", "Line Tool",
tool));

        tool = new ConnectionTool(view(), new LineConnection());
        palette.add(createToolButton(IMAGES+"CONN", "Connection Tool",
tool));

        tool = new ConnectionTool(view(), new ElbowConnection());
        palette.add(createToolButton(IMAGES+"OCONN", "Elbow Connection
Tool", tool));

        tool = new ScribbleTool(view());
        palette.add(createToolButton(IMAGES+"SCRIBBL", "Scribble
Tool", tool));

        tool = new PolygonTool(view());
        palette.add(createToolButton(IMAGES+"POLYGON", "Polygon Tool",
tool));

        tool = new BorderTool(view());
        palette.add(createToolButton(IMAGES+"BORDDEC", "Border Tool",
tool));
    }

    protected void createButtons(Panel panel) {
        super.createButtons(panel);
        fAnimationButton = new Button("Start Animation");
        fAnimationButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    toggleAnimation();
                }
            }
        );
        panel.add(fAnimationButton);
    }

    protected Drawing createDrawing() {
        return new BouncingDrawing();
    }

    //-- animation support -----

    public void startAnimation() {
        if (drawing() instanceof Animatable && fAnimator == null) {
            fAnimator = new Animator((Animatable) drawing(), view());
        }
    }

```

```

        fAnimator.start();
        fAnimationButton.setLabel("End Animation");
    }
}

public void endAnimation() {
    if (fAnimator != null) {
        fAnimator.end();
        fAnimator = null;
        fAnimationButton.setLabel("Start Animation");
    }
}

public void toggleAnimation() {
    if (fAnimator != null)
        endAnimation();
    else
        startAnimation();
}
}

```

Clase JavaDrawViewer

Se encuentra en el fichero JavaDrawViewer.java y su contenido es el siguiente:

```

/*
 * @(#)JavaDrawViewer.java 5.1
 *
 */

package CH.ifa.draw.samples.javadraw;

import java.applet.Applet;
import java.awt.*;
import java.awt.event.MouseEvent;
import java.util.*;
import java.io.*;
import java.net.*;

import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.*;

public class JavaDrawViewer extends Applet implements DrawingEditor {

    private Drawing          fDrawing;
    private Tool             fTool;
    private StandardDrawingView fView;
    private Iconkit          fIconkit;

    public void init() {
        setLayout(new BorderLayout());
    }
}

```



```

fView = new StandardDrawingView(this, 400, 370);
add("Center", fView);
fTool = new FollowURLTool(view(), this);

fIconkit = new Iconkit(this);

String filename = getParameter("Drawing");
if (filename != null) {
    loadDrawing(filename);
    fView.setDrawing(fDrawing);
} else
    showStatus("Unable to load drawing");
}

private void loadDrawing(String filename) {
    try {
        URL url = new URL(getCodeBase(), filename);
        InputStream stream = url.openStream();
        StorableInput reader = new StorableInput(stream);
        fDrawing = (Drawing)reader.readStorable();
    } catch (IOException e) {
        fDrawing = new StandardDrawing();
        System.out.println("Error when Loading: " + e);
        showStatus("Error when Loading: " + e);
    }
}

/**
 * Gets the editor's drawing view.
 */
public DrawingView view() {
    return fView;
}

/**
 * Gets the editor's drawing.
 */
public Drawing drawing() {
    return fDrawing;
}

/**
 * Gets the current the tool (there is only one):
 */
public Tool tool() {
    return fTool;
}

/**
 * Sets the editor's default tool. Do nothing since we only have
 * one tool.
 */
public void toolDone() {}

/**
 * Ignore selection changes, we don't show any selection
 */
public void selectionChanged(DrawingView view) {}

```

```
}
```

Clase MySelectionTool

Una SelectionTool que interpreta doble clicks para chequear la figura seleccionada.

Se encuentra en el fichero MySelectionTool.java y su contenido es el siguiente:

```
/*
 * @(#)MySelectionTool.java 5.1
 *
 */

package CH.ifa.draw.samples.javadraw;

import java.awt.*;
import java.awt.event.MouseEvent;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;

/**
 * A SelectionTool that interprets double clicks to inspect the
 * clicked figure
 */

public class MySelectionTool extends SelectionTool {

    public MySelectionTool(DrawingView view) {
        super(view);
    }

    /**
     * Handles mouse down events and starts the corresponding tracker.
     */
    public void mouseDown(MouseEvent e, int x, int y) {
        if (e.getClickCount() == 2) {
            Figure figure = drawing().findFigure(e.getX(), e.getY());
            if (figure != null) {
                inspectFigure(figure);
                return;
            }
        }
        super.mouseDown(e, x, y);
    }

    protected void inspectFigure(Figure f) {
        System.out.println("inspect figure"+f);
    }
}
```

Clase PatternPainter

PatternPainter un fondo que se puede añadir a un drawing.

Se encuentra en el fichero PatternPainter.java y su contenido es el siguiente:

```
/*
 * @(#)PatternPainter.java 5.1
 *
 */

package CH.ifa.draw.samples.javadraw;

import java.awt.*;
import CH.ifa.draw.framework.*;

/**
 * PatternPainter a background that can be added to a
 * drawing.
 * @see DrawingView
 * @see Painter
 */

public class PatternPainter
    implements Painter {

    private Image    fImage;

    public PatternPainter(Image image) {
        fImage = image;
    }

    public void draw(Graphics g, DrawingView view) {
        drawPattern(g, fImage, view);
    }

    /**
     * Draws a pattern background pattern by replicating an image.
     */
    private void drawPattern(Graphics g, Image image, DrawingView
view) {
        int iwidth = image.getWidth(view);
        int iheight = image.getHeight(view);
        Dimension d = view.getSize();
        int x = 0; int y = 0;

        while (y < d.height) {
            while (x < d.width) {
                g.drawImage(image, x, y, view);
                x += iwidth;
            }
            y += iheight;
            x = 0;
        }
    }
}
```

```
}
```

Clase URLTool

Una herramienta que adjunta URLs a figuras. Las URLs son guardadas en el atributo “URL” de la figure. El texto URL se anota con un FloatingTextField.

Se encuentra en el fichero URLTool.java y su contenido es el siguiente:

```
/*
 * @(#)URLTool.java 5.1
 *
 */

package CH.ifa.draw.samples.javadraw;

import java.awt.*;
import java.awt.event.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.util.*;

/**
 * A tool to attach URLs to figures.
 * The URLs are stored in the figure's "URL" attribute.
 * The URL text is entered with a FloatingTextField.
 * @see FloatingTextField
 */
public class URLTool extends AbstractTool {

    private FloatingTextField fTextField;
    private Figure fURLETarget;

    public URLTool(DrawingView view) {
        super(view);
    }

    public void mouseDown(MouseEvent e, int x, int y)
    {
        Figure pressedFigure;

        pressedFigure = drawing().findFigureInside(x, y);
        if (pressedFigure != null) {
            beginEdit(pressedFigure);
            return;
        }
        endEdit();
    }

    public void mouseUp(MouseEvent e, int x, int y) {
    }

    public void deactivate(DrawingView view) {
```

```

        super.deactivate();
        endEdit();
    }

    public void endAction(ActionEvent e) {
        endEdit();
    }

    private void beginEdit(Figure figure) {
        if (fTextField == null) {
            fTextField = new FloatingTextField();
            fTextField.addActionListener(
                new ActionListener() {
                    public void actionPerformed(ActionEvent event) {
                        endAction(event);
                    }
                }
            );
        }

        if (figure != fURLTarget && fURLTarget != null)
            endEdit();
        if (figure != fURLTarget) {
            fTextField.createOverlay((Container)view());
            fTextField.setBounds(fieldBounds(figure), getURL(figure));
            fURLTarget = figure;
        }
    }

    private void endEdit() {
        if (fURLTarget != null) {
            setURL(fURLTarget, fTextField.getText());
            fURLTarget = null;
            fTextField.endOverlay();
        }
    }

    private Rectangle fieldBounds(Figure figure) {
        Rectangle box = figure.displayBox();
        int nChars = Math.max(20, getURL(figure).length());
        Dimension d = fTextField.getPreferredSize(nChars);
        box.x = Math.max(0, box.x + (box.width - d.width)/2);
        box.y = Math.max(0, box.y + (box.height - d.height)/2);
        return new Rectangle(box.x, box.y, d.width, d.height);
    }

    private String getURL(Figure figure) {
        String url = (String) figure.getAttribute("URL");
        if (url == null)
            url = "";
        return url;
    }

    private void setURL(Figure figure, String url) {
        figure.setAttribute("URL", url);
    }
}

```


Paquete CH.ifa.draw.samples.pert

El paquete para el applet y la aplicación del ejemplo pert.

Este paquete contiene lo siguiente:

- Clase PertApplet.
- Clase PertApplication.
- Clase PertDependency.
- Clase PertFigure.
- Clase PertFigureCreationTool

Clase PertApplet

Se encuentra en el fichero PertApplet.java y su contenido es el siguiente:

```
/*
 * @(#)PertApplet.java 5.1
 *
 */

package CH.ifa.draw.samples.pert;

import java.applet.Applet;
import java.awt.*;
import java.util.*;
import java.io.*;
import java.net.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.applet.*;

public class PertApplet extends DrawApplet {

    private final static String PERTIMAGES =
"/CH/ifa/draw/samples/pert/images/";

    protected void createTools(Panel palette) {
        super.createTools(palette);

        Tool tool;
        tool = new TextTool(view(), new TextFigure());
        palette.add(createToolButton(IMAGES+"TEXT", "Text Tool",
tool));

        tool = new PertFigureCreationTool(view());
```

```

        palette.add(createToolButton(PERTIMAGES+"PERT", "Task Tool",
tool));

        tool = new ConnectionTool(view(), new PertDependency());
        palette.add(createToolButton(IMAGES+"CONN", "Dependency Tool",
tool));

        tool = new CreationTool(view(), new LineFigure());
        palette.add(createToolButton(IMAGES+"Line", "Line Tool",
tool));
    }
}

```

Clase PertApplication

Se encuentra en el fichero PertApplication.java y su contenido es el siguiente:

```

/*
 * @(#)PertApplication.java 5.1
 *
 */

package CH.ifa.draw.samples.pert;

import java.awt.*;
import java.util.*;
import java.io.*;
import java.net.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.application.*;

public class PertApplication extends DrawApplication {

    static private final String PERTIMAGES =
"/CH/ifa/draw/samples/pert/images/";

    PertApplication() {
        super("PERT Editor");
    }

    protected void createTools(Panel palette) {
        super.createTools(palette);

        Tool tool;
        tool = new TextTool(view(), new TextFigure());
        palette.add(createToolButton(IMAGES+"TEXT", "Text Tool",
tool));

        // the generic but slower version
        //tool = new CreationTool(new PertFigure());
        //palette.add(createToolButton(PERTIMAGES+"PERT", "Task Tool",

```



```

        //tool));

        tool = new PertFigureCreationTool(view());
        palette.add(createToolButton(PERTIMAGES+"PERT", "Task Tool",
tool));

        tool = new ConnectionTool(view(), new PertDependency());
        palette.add(createToolButton(IMAGES+"CONN", "Dependency Tool",
tool));

        tool = new CreationTool(view(), new LineFigure());
        palette.add(createToolButton(IMAGES+"Line", "Line Tool",
tool));
    }

    //-- main -----

    public static void main(String[] args) {
        PertApplication pert = new PertApplication();
        pert.open();
    }
}

```

Clase PertDependency

Se encuentra en el fichero PertDependency.java y su contenido es el siguiente:

```

/*
 * @(#)PertDependency.java 5.1
 *
 */

package CH.ifa.draw.samples.pert;

import java.awt.*;
import java.util.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.standard.*;

public class PertDependency extends LineConnection {
    /*
     * Serialization support.
     */
    private static final long serialVersionUID = -
7959500008698525009L;
    private int pertDependencySerializedDataVersion = 1;

    public PertDependency() {
        setEndDecoration(new ArrowTip());
        setStartDecoration(null);
    }
}

```

```

    public void handleConnect(Figure start, Figure end) {
        PertFigure source = (PertFigure)start;
        PertFigure target = (PertFigure)end;
        if (source.hasCycle(target)) {
            setAttribute("FrameColor", Color.red);
        } else {
            target.addPreTask(source);
            source.addPostTask(target);
            source.notifyPostTasks();
        }
    }

    public void handleDisconnect(Figure start, Figure end) {
        PertFigure source = (PertFigure)start;
        PertFigure target = (PertFigure)end;
        if (target != null) {
            target.removePreTask(source);
            target.updateDurations();
        }
        if (source != null)
            source.removePostTask(target);
    }

    public boolean canConnect(Figure start, Figure end) {
        return (start instanceof PertFigure && end instanceof
PertFigure);
    }

    public Vector handles() {
        Vector handles = super.handles();
        // don't allow to reconnect the starting figure
        handles.setElementAt(
            new NullHandle(this, PolyLineFigure.locator(0)), 0);
        return handles;
    }
}

```

Clase PertFigure

Se encuentra en el fichero PertFigure.java y su contenido es el siguiente:

```

/*
 * @(#)PertFigure.java 5.1
 *
 */

package CH.ifa.draw.samples.pert;

import java.awt.*;
import java.util.*;
import java.io.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;

```

```
import CH.ifa.draw.figures.*;
import CH.ifa.draw.util.*;

public class PertFigure extends CompositeFigure {
    private static final int BORDER = 3;
    private Rectangle fDisplayBox;
    private Vector fPreTasks;
    private Vector fPostTasks;

    /*
     * Serialization support.
     */
    private static final long serialVersionUID = -
7877776240236946511L;
    private int pertFigureSerializedDataVersion = 1;

    public PertFigure() {
        initialize();
    }

    public int start() {
        int start = 0;
        Enumeration i = fPreTasks.elements();
        while (i.hasMoreElements()) {
            PertFigure f = (PertFigure) i.nextElement();
            start = Math.max(start, f.end());
        }
        return start;
    }

    public int end() {
        return asInt(2);
    }

    public int duration() {
        return asInt(1);
    }

    public void setEnd(int value) {
        setInt(2, value);
    }

    public void addPreTask(PertFigure figure) {
        if (!fPreTasks.contains(figure)) {
            fPreTasks.addElement(figure);
        }
    }

    public void addPostTask(PertFigure figure) {
        if (!fPostTasks.contains(figure)) {
            fPostTasks.addElement(figure);
        }
    }

    public void removePreTask(PertFigure figure) {
        fPreTasks.removeElement(figure);
    }
}
```

```

public void removePostTask(PertFigure figure) {
    fPostTasks.removeElement(figure);
}

private int asInt(int figureIndex) {
    NumberTextFigure t = (NumberTextFigure)figureAt(figureIndex);
    return t.getValue();
}

private String taskName() {
    TextFigure t = (TextFigure)figureAt(0);
    return t.getText();
}

private void setInt(int figureIndex, int value) {
    NumberTextFigure t = (NumberTextFigure)figureAt(figureIndex);
    t.setValue(value);
}

protected void basicMoveBy(int x, int y) {
    fDisplayBox.translate(x, y);
    super.basicMoveBy(x, y);
}

public Rectangle displayBox() {
    return new Rectangle(
        fDisplayBox.x,
        fDisplayBox.y,
        fDisplayBox.width,
        fDisplayBox.height);
}

public void basicDisplayBox(Point origin, Point corner) {
    fDisplayBox = new Rectangle(origin);
    fDisplayBox.add(corner);
    layout();
}

private void drawBorder(Graphics g) {
    super.draw(g);

    Rectangle r = displayBox();

    Figure f = figureAt(0);
    Rectangle rf = f.displayBox();
    g.setColor(Color.gray);
    g.drawLine(r.x, r.y+rf.height+2, r.x+r.width, r.y +
rf.height+2);
    g.setColor(Color.white);
    g.drawLine(r.x, r.y+rf.height+3, r.x+r.width, r.y +
rf.height+3);

    g.setColor(Color.white);
    g.drawLine(r.x, r.y, r.x, r.y + r.height);
    g.drawLine(r.x, r.y, r.x + r.width, r.y);
    g.setColor(Color.gray);
    g.drawLine(r.x + r.width, r.y, r.x + r.width, r.y + r.height);
}

```

```

        g.drawLine(r.x , r.y + r.height, r.x + r.width, r.y +
r.height);
    }

    public void draw(Graphics g) {
        drawBorder(g);
        super.draw(g);
    }

    public Vector handles() {
        Vector handles = new Vector();
        handles.addElement(new NullHandle(this,
RelativeLocator.northWest()));
        handles.addElement(new NullHandle(this,
RelativeLocator.northEast()));
        handles.addElement(new NullHandle(this,
RelativeLocator.southWest()));
        handles.addElement(new NullHandle(this,
RelativeLocator.southEast()));
        handles.addElement(new ConnectionHandle(this,
RelativeLocator.east(),
                                new PertDependency())
                                );
        return handles;
    }

    private void initialize() {
        fPostTasks = new Vector();
        fPreTasks = new Vector();
        fDisplayBox = new Rectangle(0, 0, 0, 0);

        Font f = new Font("Helvetica", Font.PLAIN, 12);
        Font fb = new Font("Helvetica", Font.BOLD, 12);

        TextFigure name = new TextFigure();
        name.setFont(fb);
        name.setText("Task");
        //name.setAttribute("TextColor",Color.white);
        add(name);

        NumberTextFigure duration = new NumberTextFigure();
        duration.setValue(0);
        duration.setFont(fb);
        add(duration);

        NumberTextFigure end = new NumberTextFigure();
        end.setValue(0);
        end.setFont(f);
        end.setReadOnly(true);
        add(end);
    }

    private void layout() {
        Point partOrigin = new Point(fDisplayBox.x, fDisplayBox.y);
        partOrigin.translate(BORDER, BORDER);
        Dimension extent = new Dimension(0, 0);

        FigureEnumeration k = figures();
    }

```

```

        while (k.hasMoreElements()) {
            Figure f = k.nextFigure();

            Dimension partExtent = f.size();
            Point corner = new Point(
                partOrigin.x+partExtent.width,
                partOrigin.y+partExtent.height);
            f.basicDisplayBox(partOrigin, corner);

            extent.width = Math.max(extent.width, partExtent.width);
            extent.height += partExtent.height;
            partOrigin.y += partExtent.height;
        }
        fDisplayBox.width = extent.width + 2*BORDER;
        fDisplayBox.height = extent.height + 2*BORDER;
    }

    private boolean needsLayout() {
        Dimension extent = new Dimension(0, 0);

        FigureEnumeration k = figures();
        while (k.hasMoreElements()) {
            Figure f = k.nextFigure();
            extent.width = Math.max(extent.width, f.size().width);
        }
        int newExtent = extent.width + 2*BORDER;
        return newExtent != fDisplayBox.width;
    }

    public void update(FigureChangeEvent e) {
        if (e.getFigure() == figureAt(1)) // duration has changed
            updateDurations();
        if (needsLayout()) {
            layout();
            changed();
        }
    }

    public void figureChanged(FigureChangeEvent e) {
        update(e);
    }

    public void figureRemoved(FigureChangeEvent e) {
        update(e);
    }

    public void notifyPostTasks() {
        Enumeration i = fPostTasks.elements();
        while (i.hasMoreElements())
            ((PertFigure) i.nextElement()).updateDurations();
    }

    public void updateDurations() {
        int newEnd = start()+duration();
        if (newEnd != end()) {
            setEnd(newEnd);
            notifyPostTasks();
        }
    }

```

```

    }
}

public boolean hasCycle(Figure start) {
    if (start == this)
        return true;
    Enumeration i = fPreTasks.elements();
    while (i.hasMoreElements()) {
        if (((PertFigure) i.nextElement()).hasCycle(start))
            return true;
    }
    return false;
}

//-- store / load -----

public void write(StorableOutput dw) {
    super.write(dw);
    dw.writeInt(fDisplayBox.x);
    dw.writeInt(fDisplayBox.y);
    dw.writeInt(fDisplayBox.width);
    dw.writeInt(fDisplayBox.height);

    writeTasks(dw, fPreTasks);
    writeTasks(dw, fPostTasks);
}

public void writeTasks(StorableOutput dw, Vector v) {
    dw.writeInt(v.size());
    Enumeration i = v.elements();
    while (i.hasMoreElements())
        dw.writeStorable((Storable) i.nextElement());
}

public void read(StorableInput dr) throws IOException {
    super.read(dr);
    fDisplayBox = new Rectangle(
        dr.readInt(),
        dr.readInt(),
        dr.readInt(),
        dr.readInt());
    layout();
    fPreTasks = readTasks(dr);
    fPostTasks = readTasks(dr);
}

public Insets connectionInsets() {
    Rectangle r = fDisplayBox;
    int cx = r.width/2;
    int cy = r.height/2;
    return new Insets(cy, cx, cy, cx);
}

public Vector readTasks(StorableInput dr) throws IOException {
    int size = dr.readInt();
    Vector v = new Vector(size);
    for (int i=0; i<size; i++)
        v.addElement((Figure)dr.readStorable());
}

```

```

        }
        return v;
    }
}

```

Clase PertFigureCreationTool

Una versión más eficiente de la herramienta de creación Pert que no esta basada en la clonación.

Se encuentra en el fichero PertFigureCreationTool.java y su contenido es el siguiente:

```

/*
 * @(#)PertFigureCreationTool.java 5.1
 *
 */

package CH.ifa.draw.samples.pert;

import java.awt.*;

import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;

/**
 * A more efficient version of the generic Pert creation
 * tool that is not based on cloning.
 */

public class PertFigureCreationTool extends CreationTool {

    public PertFigureCreationTool(DrawingView view) {
        super(view);
    }

    /**
     * Creates a new PertFigure.
     */
    protected Figure createFigure() {
        return new PertFigure();
    }
}

```


Paquete CH.ifa.draw.samples.nothing

El paquete del applet y la aplicación para el ejemplo nothing.

Este paquete contiene lo siguiente:

- Clase NothingApp.
- Clase NothingApplet.

Clase NothingApp

Se encuentra en el fichero NothingApp.java y su contenido es el siguiente:

```
/*
 * @(#)NothingApp.java 5.1
 *
 */

package CH.ifa.draw.samples.nothing;

import java.awt.*;
import java.util.*;
import java.io.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.contrib.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.application.*;

public class NothingApp extends DrawApplication {

    NothingApp() {
        super("Nothing");
    }

    protected void createTools(Palette palette) {
        super.createTools(palette);

        Tool tool = new TextTool(view(), new TextFigure());
        palette.add(createToolButton(Images.TEXT, "Text Tool",
tool));

        tool = new CreationTool(view(), new RectangleFigure());
        palette.add(createToolButton(Images.RECT, "Rectangle Tool",
tool));

        tool = new CreationTool(view(), new RoundedRectangleFigure());
        palette.add(createToolButton(Images.RRECT, "Round Rectangle
Tool", tool));
    }
}
```

```

        tool = new CreationTool(view(), new EllipseFigure());
        palette.add(createToolButton(IMAGES+"ELLIPSE", "Ellipse Tool",
tool));

        tool = new CreationTool(view(), new LineFigure());
        palette.add(createToolButton(IMAGES+"LINE", "Line Tool",
tool));

        tool = new PolygonTool(view());
        palette.add(createToolButton(IMAGES+"POLYGON", "Polygon Tool",
tool));

        tool = new ConnectionTool(view(), new LineConnection());
        palette.add(createToolButton(IMAGES+"CONN", "Connection Tool",
tool));

        tool = new ConnectionTool(view(), new ElbowConnection());
        palette.add(createToolButton(IMAGES+"OCONN", "Elbow Connection
Tool", tool));
    }

    //-- main -----

    public static void main(String[] args) {
        DrawApplication window = new NothingApp();
        window.open();
    }
}

```

Clase NothingApplet

Se encuentra en el fichero NothingApplet.java y su contenido es el siguiente:

```

/*
 * @(#)NothingApplet.java 5.1
 *
 */

package CH.ifa.draw.samples.nothing;

import java.applet.Applet;
import java.awt.*;
import java.util.*;
import java.io.*;
import java.net.*;

import CH.ifa.draw.util.Iconkit;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.applet.*;
import CH.ifa.draw.contrib.*;

```

```

public class NothingApplet extends DrawApplet {

    //-- DrawApplet overrides -----

    protected void createTools(Palette palette) {
        super.createTools(palette);

        Tool tool = new TextTool(view(), new TextFigure());
        palette.add(createToolButton(IMAGES+"TEXT", "Text Tool",
tool));

        tool = new CreationTool(view(), new RectangleFigure());
        palette.add(createToolButton(IMAGES+"RECT", "Rectangle Tool",
tool));

        tool = new CreationTool(view(), new RoundRectangleFigure());
        palette.add(createToolButton(IMAGES+"RRECT", "Round Rectangle
Tool", tool));

        tool = new CreationTool(view(), new EllipseFigure());
        palette.add(createToolButton(IMAGES+"ELLIPSE", "Ellipse Tool",
tool));

        tool = new CreationTool(view(), new LineFigure());
        palette.add(createToolButton(IMAGES+"LINE", "Line Tool",
tool));

        tool = new PolygonTool(view());
        palette.add(createToolButton(IMAGES+"POLYGON", "Polygon Tool",
tool));

        tool = new ConnectionTool(view(), new LineConnection());
        palette.add(createToolButton(IMAGES+"CONN", "Connection Tool",
tool));

        tool = new ConnectionTool(view(), new ElbowConnection());
        palette.add(createToolButton(IMAGES+"OCONN", "Elbow Connection
Tool", tool));
    }
}

```


Paquete CH.ifa.draw.samples.net

El paquete para la aplicación del ejemplo net.

Este paquete contiene lo siguiente:

- Clase NetApp.
- Clase NodeFigure.

Clase NetApp

Se encuentra en el fichero NetApp.java y su contenido es el siguiente:

```
/*
 * @(#)NetApp.java 5.1
 *
 */

package CH.ifa.draw.samples.net;

import java.awt.*;
import java.util.*;
import java.io.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.application.*;

public class NetApp extends DrawApplication {

    NetApp() {
        super("Net");
    }

    protected void createTools(Palette palette) {
        super.createTools(palette);

        Tool tool = new TextTool(view(), new NodeFigure());
        palette.add(createToolButton(Images.TEXT, "Text Tool",
tool));

        tool = new CreationTool(view(), new NodeFigure());
        palette.add(createToolButton(Images.RECT, "Create Org Unit",
tool));

        tool = new ConnectionTool(view(), new LineConnection());
        palette.add(createToolButton(Images.CONN, "Connection Tool",
tool));
    }
}
```

```

    //-- main -----

    public static void main(String[] args) {
        DrawApplication window = new NetApp();
        window.open();
    }
}

```

Clase NodeFigure

Se encuentra en el fichero NodeFigure.java y su contenido es el siguiente:

```

/*
 * @(#)NodeFigure.java 5.1
 *
 */

package CH.ifa.draw.samples.net;

import java.awt.*;
import java.util.*;
import java.io.IOException;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;
import CH.ifa.draw.figures.*;
import CH.ifa.draw.util.*;

public class NodeFigure extends TextFigure {
    private static final int BORDER = 6;
    private Vector          fConnectors;
    private boolean         fConnectorsVisible;

    public NodeFigure() {
        initialize();
        fConnectors = null;
    }

    public Rectangle displayBox() {
        Rectangle box = super.displayBox();
        int d = BORDER;
        box.grow(d, d);
        return box;
    }

    public boolean containsPoint(int x, int y) {
        // add slop for connectors
        if (fConnectorsVisible) {
            Rectangle r = displayBox();
            int d = LocatorConnector.SIZE/2;
            r.grow(d, d);
            return r.contains(x, y);
        }
        return super.containsPoint(x, y);
    }
}

```

```

    }

    private void drawBorder(Graphics g) {
        Rectangle r = displayBox();
        g.setColor(getFrameColor());
        g.drawRect(r.x, r.y, r.width-1, r.height-1);
    }

    public void draw(Graphics g) {
        super.draw(g);
        drawBorder(g);
        drawConnectors(g);
    }

    public Vector handles() {
        ConnectionFigure prototype = new LineConnection();
        Vector handles = new Vector();
        handles.addElement(new ConnectionHandle(this,
RelativeLocator.east(), prototype));
        handles.addElement(new ConnectionHandle(this,
RelativeLocator.west(), prototype));
        handles.addElement(new ConnectionHandle(this,
RelativeLocator.south(), prototype));
        handles.addElement(new ConnectionHandle(this,
RelativeLocator.north(), prototype));

        handles.addElement(new NullHandle(this,
RelativeLocator.southEast()));
        handles.addElement(new NullHandle(this,
RelativeLocator.southWest()));
        handles.addElement(new NullHandle(this,
RelativeLocator.northEast()));
        handles.addElement(new NullHandle(this,
RelativeLocator.northWest()));
        return handles;
    }

    private void drawConnectors(Graphics g) {
        if (fConnectorsVisible) {
            Enumeration e = connectors().elements();
            while (e.hasMoreElements())
                ((Connector) e.nextElement()).draw(g);
        }
    }

    /**
     */
    public void connectorVisibility(boolean isVisible) {
        fConnectorsVisible = isVisible;
        invalidate();
    }

    /**
     */
    public Connector connectorAt(int x, int y) {
        return findConnector(x, y);
    }

```

```

/**
 */
private Vector connectors() {
    if (fConnectors == null)
        createConnectors();
    return fConnectors;
}

private void createConnectors() {
    fConnectors = new Vector(4);
    fConnectors.addElement(new LocatorConnector(this,
RelativeLocator.north()) );
    fConnectors.addElement(new LocatorConnector(this,
RelativeLocator.south()) );
    fConnectors.addElement(new LocatorConnector(this,
RelativeLocator.west()) );
    fConnectors.addElement(new LocatorConnector(this,
RelativeLocator.east()) );
}

private Connector findConnector(int x, int y) {
    // return closest connector
    long min = Long.MAX_VALUE;
    Connector closest = null;
    Enumeration e = connectors().elements();
    while (e.hasMoreElements()) {
        Connector c = (Connector)e.nextElement();
        Point p2 = Geom.center(c.displayBox());
        long d = Geom.length2(x, y, p2.x, p2.y);
        if (d < min) {
            min = d;
            closest = c;
        }
    }
    return closest;
}

private void initialize() {
    setText("node");
    Font fb = new Font("Helvetica", Font.BOLD, 12);
    setFont(fb);
    createConnectors();
}
}

```