UNIVERSIDAD DE OVIEDO

DEPARTAMENTO DE INFORMÁTICA

TESIS DOCTORAL

RDM: Arquitectura Software para el Modelado de Dominios en Sistemas Informáticos

Presentada por

Raúl Izquierdo Castanedo

para la obtención del título de

Doctor por la Universidad de Oviedo

Dirigida por el Dr. Juan Manuel Cueva Lovelle

Oviedo, Junio de 2002

RDM: Arquitectura Software para el Modelado de Dominios en Sistemas Informáticos

UNIVERSIDAD DE OVIEDO

DEPARTAMENTO DE INFORMÁTICA

Autor

Raúl Izquierdo Castanedo

Director

Dr. Juan Manuel Cueva Lovelle

Oviedo, Junio de 2002

A Aquilino, Benjamín, Belén, Paco, Darío, César y el resto de la gente del tercer piso. Por sus revisiones del texto y por el apoyo que me dieron durante su realización.

A Juan Manuel Cueva Lovelle, mi director de tesis, sin el cual ésta no se hubiera realizado (aunque suene a tópico). Sus consejos tanto sobre los contenidos de la tesis como sobre los demás aspectos que la rodean han evitado que la misma hubiera estado en un perpetuo estado de *en construcción*.

A Ricardo. Por darme la oportunidad de colaborar en los proyectos de los cuales surgieron las ideas de esta tesis. Gracias por la paciencia mostrada cada vez que no pude ayudar por que tenía que "dedicarme a la tesis".

A Alejandro, Tomás, Pablo y Javi. Este trabajo es una continuación del que se empezó hace algunos años.

A Fran, Pablo y el resto de la gente de *Comercio Electrónico B2B 2000* por su colaboración en la creación y uso de la arquitectura.

"En el año 2000 expertos de todo el mundo participaron en la elección del hombre más importante de la historia. Aunque nombres de grandes genios como Einstein, Newton, Edison y Leonardo fueron los primeros propuestos finalmente fue un simple orfebre alemán el elegido. Si alguno de los grandes genios no hubiera existido sus ideas se hubieran perdido. Si Guttemberg y su imprenta no hubieran existido se hubieran perdido las ideas de muchos genios". *A mis padres* por ayudarme y haber hecho posible que los demás pudieran ayudarme.

A Paula. Por primera vez desde que nos conocemos no tendremos que negociar si el verano se dedica de nuevo a la tesis. Si todo sale bien el próximo verano por fin iremos a... algún sitio!!!

Resumen

Este documento describe una arquitectura y una implementación de la misma para el modelado de los distintos elementos del dominio de una aplicación. La forma de modelar las entidades, sus relaciones y las operaciones es consecuencia de un conjunto de decisiones las cuales producirán un gran impacto en la productividad obtenido durante el desarrollo.

Los cambios del dominio durante las etapas de desarrollo de las aplicaciones es algo inevitable. Nuevas metodologías de las denominadas *ligeras* (como por ejemplo Extreme Programming) proponen adaptar el proceso de desarrollo para afrontar como parte integral del mismo la actitud hacia el cambio constante. Los requisitos se cambian de una iteración a otra; el diseño se cambia para reflejar exclusivamente los requisitos actuales; el código se refactoriza para implementar dicho diseño de la forma más simple.

Por tanto en procesos de desarrollo con condiciones tan variables (al menos en los primeros ciclos) se acentúa la necesidad de técnicas de diseño e implementación ágiles que permitan seguir el ritmo de cambio constante que se exige al desarrollador.

En este documento se presenta una arquitectura especialmente dirigida a los cambios frecuentes y rápidos de dominio que permita obtener una mayor eficacia a la hora de llevar dichos cambios al diseño y su implementación. Esta arquitectura es a la vez un conjunto de métodos para modelar los distintos elementos del dominio y una implementación base de la que reutilizar las tareas comunes de prácticamente cualquier aplicación.

Palabras Clave

Patrones de diseño, patrones arquitectónicos, Extreme Programming, metodologías ligeras, modelado de dominios orientados a objetos, diseño orientado a objetos, adaptabilidad, encapsulación, ciclo de vida, integridad, especialización, frameworks orientados a objetos, reutilización.

Abstract

This document describes an architecture and one of its possible implementations for computer systems domain modeling. The way to design the computer model of entities, their relationships, the operations and the reactions can make a huge impact in the development phase productivity.

Domain changes during development phases are inevitable. New methodologies known as *lightweight methodologies* (like Extreme Programming) propose to adapt all the phases of the process to continuos change (to *embrace change*). The requirements are changed from one iteration to the next; the design is changed to reflect exactly just the current requirements; code is refactored to implement the design in the simplest way.

So in such a variable way of development its firmly needed new methods of design and implementation that allow the constant change requirements imposed on the development team.

In this document a new architecture is presented especially targeted to constant and rapid domain changes. It offers an improvement in reduced time when translating from requirements to design and its implementation. This architecture is both a methods set to domain modeling and an implementation to reuse in new aplications.

Keywords

Design patterns, architectural patterns, Extreme Programming, lightweight methodologies, object oriented domain modeling, object oriented design, adaptability, encapsulation, development process, integrity, specialization, object oriented frameworks, reusability.

Indice Resumido

Indice

Introducción y Contenidos

1 Introducción

1.1 La Especialización del Objeto

En el dominio de una aplicación, es decir, en la parte del mundo real que debe ser contemplado por la misma, se encuentran distintos elementos que hay que trasladar al modelo que será implementado. La fase de análisis se convierte en una continua tarea de investigación para obtener una relación de dichos elementos: las entidades relevantes, sus relaciones, las operaciones a ejecutar, las reglas que rigen dichas operaciones y las reacciones que producen, las validaciones, los casos excepcionales a considerar, etc.

A la hora de realizar un trabajo basado en las tecnologías orientadas a objetos se suele plantear al objeto como a un elemento *atómico* que tiene un estado y un comportamiento *cualquiera*. El objeto llevará a cabo dicho comportamiento incorporando internamente aquellos elementos que le sean necesarios (datos, operaciones, reacciones, validaciones, etc). Es decir un objeto tradicional encapsulará *a varios* elementos del dominio. Esta encapsulación conjunta, como se expondrá posteriormente, presenta problemas de dependencias en cuanto a que el cambio de uno de esos elementos afecta a los demás encapsulados en el mismo objeto.

Además en la práctica (especialmente en los obtenidos a partir de un dominio) los objetos suelen tener *una clara tendencia*: los hay orientados a representar una información; otros a mantener una estructura; otros a ser aplicados o a controlar a otros objetos, etc. Ignorar esta tendencia hace que los objetos sean *mas complicados de tratar* que si se ajustaran a una determinada funcionalidad más delimitada.

En este documento se afirmará que *toda clase que provenga del dominio no deberá encapsular más de un elemento del mismo y deberá encapsularlo completamente*. Una arquitectura obtenida mediante la traslación de cada tipo de elemento del dominio a un único objeto especializado únicamente en su modelado proporcionará una reducción en la fase de desarrollo y favorecerá la reutilización.

Por tanto se propone que en vez de considerar al objeto como a una entidad que encapsula *cualquier comportamiento* se consideren objetos especializados en *un solo comportamiento o tarea* predefinidos. Esto permitirá su tratamiento de una forma más centrada en sus características.

Un objeto *dejará de ser la unidad indivisible elemental* y se separará en otros objetos *más primitivos* (lo mismo que los átomos se dividen a su vez en electrones, protones y neutrones) que, mediante su adecuada combinación, aportarán la funcionalidad que previamente ofrecía el objeto original, pero que estarán en disposición de ser combinados de otras formas (bajo el dictamen de los cambios en el dominio de la aplicación) minimizando el impacto de dichos cambios sobre las demás partes del objeto.

Las tareas primitivas se obtendrán del dominio de la aplicación. Es decir, aquello que en el dominio se encuentre por separado (relaciones, reglas, reacciones, etc.) se mantendrá por

separado sin necesidad de una fase artificial de agrupamiento en una clase tradicional; mantendrán su identidad a la hora de llevarlo a un modelo software.

Es importante aclarar que cuando se habla de modelar los elementos de forma especializada no se está hablando del mismo tipo de especialización del que se habla en la herencia. En la herencia hay una clase raíz a partir de la cual se crean nuevas clases derivadas que *especializan* el comportamiento de la clase raíz. El verbo *especializar* se está utilizando aquí con otro sentido. De lo que se trata aquí más bien es ¿qué parte del dominio representa esa clase raíz? ¿Debe tener entonces esas responsabilidades?

Lo que se pretende es dar un conjunto de métodos especializados en modelar cada elemento del dominio. Se establecen qué tipos de objetos debe haber y unas directrices sobre lo que tienen o no tienen que hacer cada uno; lo cual es totalmente independiente de que posteriormente cada uno de ellos sea la raíz de una jerarquía de herencia por aspectos de implementación.

Podría plantearse *qué novedad se presenta* frente al diseño clásico orientado a objetos. Si de verdad de esta manera se consigue un mejor diseño el desarrollador experimentado llegaría de todas formas al mismo resultado. La diferencia es:

- El diseño orientado a objeto tradicional promueve encapsular varios elementos en un mismo objeto. Permite modelar objetos independientes; *pero no es* a lo que dirige. Muy al contrario los ejemplos habituales no hacen más que presentar la unión de varios elementos en una clase [Booch94]. Posiblemente el desarrollador experimentado en lo que tenga experiencia sea *en juntar* elementos en clases.

- Aunque el desarrollador tuviera la experiencia de ver que en un diseño concreto sería más flexible la separación de ciertos elementos este proceso debería repetirlo en su próximo diseño. Tendría que llegar a dicha arquitectura *en cada aplicación*.

Varias tecnologías modernas presentan un diseño en el que se puede observar que han llegado a la conveniencia de separar ciertos elementos. Por ejemplo los *Enterprise JavaBeans* extraen ciertas operaciones de los objetos [Roman01]. Otro ejemplo evidente son los *Rule Objects* los cuales proponen extraer las reglas de negocios de los objetos [Arsanjani99].

Lo que aquí se propone es no tener que llegar a la conclusión de la conveniencia de separar *algún* tipo de elemento mediante la iteración sobre el diseño. Se propone que *todos* los elementos se vean de entrada como objetos independientes. Se evitará así tener que llegar a ello mediante un proceso de refinamiento del modelo que requiere tiempo y *experiencia*.

Por tanto en este documento no se propone únicamente utilizar un determinado framework (el que se presentará posteriormente). Se propone fundamentalmente una forma de llevar el dominio al modelo en la cual se vean a los elementos como objetos independientes *por naturaleza*. Un desarrollador que utilice este método de modelado no diría, por ejemplo, que *un objeto tiene* tal relación o tal regla; diría que tal relación y tal regla *son objetos* del modelo.

El framework es en definitiva una forma para implementar un diseño obtenido mediante estos métodos sobre un lenguaje orientado a objetos tradicional. Supone una transición a lo que sería un *Lenguaje Orientado a Elementos* que diera soporte nativo a la metodogía.

El trabajar con objetos especializados en reflejar un único tipo de elemento del dominio permitirá:

- Simplificar las operaciones al poder centrarlas en un solo tipo de comportamiento. Ya no tienen que contemplar los elementos adicionales que antes encapsulaba el objeto. Por ejemplo no es lo mismo estudiar la persistencia en objetos genéricos que en objetos especializados en guardar estado sin operaciones. Lo mismo ocurre con la distribución, concurrencia, etc.

- No hay qué cambiar la clase para insertar, modificar o borrar cualquiera de los elementos que se encuentran mezclados en la misma. Esto supone mayores tiempos de desarrollo.

- Mayor acercamiento del dominio al modelo y de éste al código, potenciando la regla de Traslación Directa (Direct Mapping) [Meyer97]. Esta regla de diseño establece que se debe conservar una correspondencia entre la estructura del problema (descrita por el modelo) y la estructura de la solución (la cual la provee el código). Se obtiene una correspondencia mucho más directa si los elementos del dominio se mantienen por separado en vez de *mezclarlos* en una clase.

- Promover la reutilización ya que los objetos son más flexibles en su composición.

1.2 Organización del Documento

El documentos está organizado en seis secciones:

1) Objetivos (capítulos 2 y 3).

2) Tecnologías Relacionadas (capítulo 4).

3) Solución Propuesta (capítulos 5, 6 y 7).

4) Demostración: Aplicación de RDM (capítulos 8, 9, 10 y 11)

5) Conclusiones y Futuras líneas (capítulos 12 y 13).

6) Apéndices (capítulos 14 y 15)

La primera sección, *Objetivos*, está compuesta por dos capítulos:

- En el capítulo titulado “Problemas en las tecnologías orientadas a objetos” se presentarán los problemas que actualmente se presentan a la hora de modelar los elementos del dominio mediante dichas tecnologías. Dicha relación de problemas servirá como justificación de la necesidad de la arquitectura RDM introducida en esta tesis.

- En el capítulo tres, "Objetivos", a la vista de los problemas observados se establecerán los objetivos buscados para evitar dichos problemas y las restricciones que deberá cumplir la solución planteada.

La segunda sección, *Tecnologías Relacionadas*, está formada por un único capítulo en el cual se mostrará una panorámica de las tecnologías actuales que tienen alguna relación con los objetivos buscados. Se estudiarán sus características y su adecuación a los objetivos y a las restricciones.

Una vez visto que los objetivos no quedan suficientemente cubiertos con las tecnologías actuales se entrará en la tercera sección titulada *Solución Propuesta*. Esta sección se divide en tres capítulos:

- En el capítulo 5, "Arquitectura RDM" se define cómo deberán modelarse los distintos elementos del dominio para que sea posible conseguir los objetivos propuestos. Se justificará los elementos necesarios en la arquitectura. Este capítulo es el que contiene *la tesis* propiamente dicha, mientras que en los dos siguientes se mostrará la viabilidad de su implementación.

- En el capítulo 6 se entra en un nivel de detalle mayor. Una vez que en el capítulo anterior se hayan establecido los elementos de la arquitectura de manera individual en ese capítulo se justificará qué "Mecanismos de Comunicación" deben utilizarse entre ellos de manera que se consiga la máxima independencia pero con total flexibilidad a la hora de su combinación.

- Finalmente en el capítulo 7, "El Framework RDM" se describirá una implementación de la arquitectura. Se verá cómo se han implementado los distintos elementos identificados en el capítulo 5 y cómo se comunican de la manera descrita en el capítulo 6. Se muestran las clases del framework mediante un tutorial del mismo donde se muestra su flexibilidad y sencillez de uso.

Una vez introducida la arquitectura y su implementación se entraría en la cuarta sección *Demostración: Aplicación de RDM*. Esta sección hace la función de demostración de la tesis ya que muestra como se ha utilizado la arquitectura en un proyecto real y cómo se han evitado los problemas inicialmente expuestos en la tesis. Esta sección está dividida a su vez en cuatro capítulos:

- En el capítulo 8 "El proyecto H24 de AENA" se describe el proyecto H24 del Aeropuerto Madrid-Barajas de manera global. Aunque sólo se entrará en detalle en la implementación de uno de sus módulos (el Editor de Procedimientos o EGP) se da una breve visión global para su adecuada situación dentro del mismo.

- Una vez comentada la situación del EGP dentro del proyecto global se presentará en el capítulo 9 "Los Requisitos del Editor Gráfico de Procesos (EGP)". Antes de entrar en los detalles de implementación que aparecerán en el capítulo 10 se hacía necesario un capítulo previo que mostrara al editor desde el punto de vista del usuario.

- Una vez finalizado el capítulo 9 ya se tiene una idea de lo que hay que contruir por lo que en el capítulo 10 se entra en los detalles de implementación que muestran la "Aplicación de RDM al EGP de Barajas" y las ventajas que se obtuvieron.

- El EGP es una de las aplicaciones construidas con la arquitectura RDM. En el capítulo 11 se mostrará una breve enumeración de otros proyectos en los que ha sido utilizada. Se pretende así mostrar que RDM no es una arquitectura teórica sino que tiene asociada una implementación práctica y aplicable (y aplicada) en aplicaciones profesionales.

En la quinta sección, *Conclusiones y Futuras Líneas*, se agrupan las conclusiones obtenidas a lo largo del documento en la aplicación de la tesis propuesta y de la arquitectura con ella construída.

Para acabar el documento se ha incluido una sección sección final en la que se recogen dos apéndices. En el primero, "El Servicio de Contextualización" se explica con más detalle una de las líneas de investigación propuestas. El segundo apéndice es el manual de usuario de la aplicación que ha servido como demostración de la validez de la arquitectura propuesta.

Sección 1. Objetivos

2 Problemas en las Tecnologías Orientadas a Objetos

2.1 Introducción

Las Tecnologías Orientadas a Objetos se fundamentan en la construcción de aplicaciones a base de una estructura de objetos en la que cada uno de ellos tendrá asignadas una serie de responsabilidades. Cumplen sus responsabilidades mediante la implementación de las mismas en métodos y atributos. Es mediante la combinación de las responsabilidades de todos estos objetos como se consigue la funcionalidad final de la aplicación.

A la hora de introducirse en las tecnologías orientadas a objetos la primera gran dificultad es la de cómo determinar *qué clases* hay que modelar en una aplicación. Hay muchos textos que intentan explicar un método para la extracción de clases ([Booch94], [Johnson91]). Esa dificultad rápidamente se junta con la dificultad de averiguar *qué responsabilidades* debe llevar a cabo cada una de ellas y los métodos adecuados para implementarlas [Wirfs90].

La respuesta a qué clases modelar y con qué responsabilidades sale del dominio del problema.

> "El dominio del problema es la parte del mundo real donde el Sistema Informático va a producir los efectos deseados junto con los medios disponibles para producirlos directa o indirectamente" [Kovitz99].

En el mundo real hay infinidad de entidades y relaciones entre ellas. Una aplicación lo que hace es traspasar al ordenador parte de dicha información para que mediante su procesamiento se pueda acceder más eficazmente a la misma.

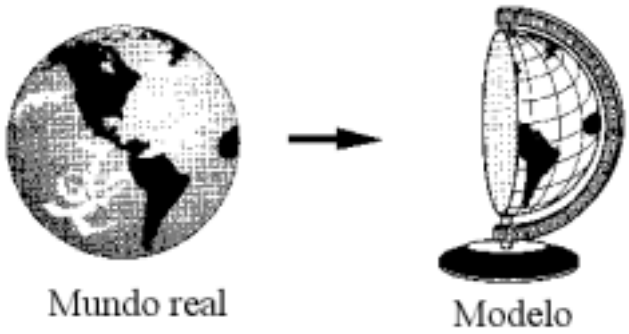

Imagen 2.1-1. El modelo como una representación de una parte del mundo real

Por ejemplo en la vida real las personas pueden tener un automóvil. ¿Quiere decir esto que cada vez que se modele la clase Persona hay que establecer una relación con una clase Automóvil? Esto dependerá de si en el dominio que se ha elegido se incluyen operaciones que para su funcionamiento requieran que dicho aspecto del mundo real esté reflejado en el sistema.

Por tanto será el dominio escogido el que determinará los aspectos a modelar y por tanto el diseño de clases adecuado. Lógicamente esto quiere decir que si se toman dos dominios distintos el modelo de objetos adecuado para cada uno ellos será diferente.

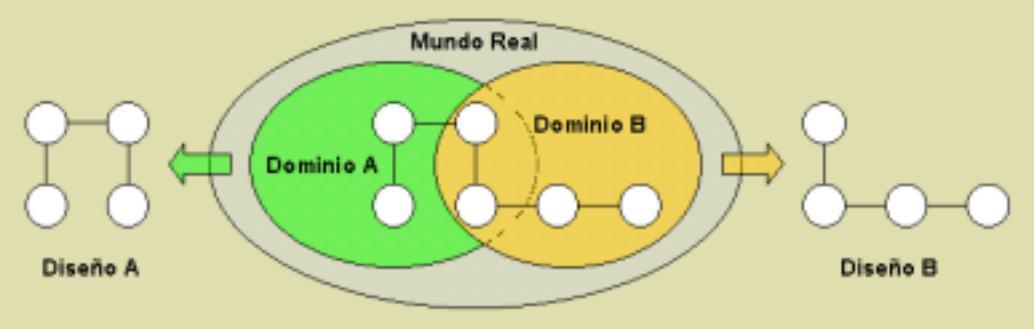

Imagen 2.1-2. Distintos dominios generan distintos diseños.

Entonces se tiene que hay que determinar el dominio para realizar un diseño de clases y que distintos dominios producen distintos diseños, por lo que *los cambios en el dominio* inicial producirán *cambios en las clases* identificadas.

Para evitar tener que estar rehaciendo las clases lo ideal sería no hacer cambios en el dominio, es decir, identificar de forma precisa el dominio de la aplicación desde un principio de manera que así no haya que cambiarlo. En esta idea se basaba el modelo de desarrollo en cascada [Pressman01]; en una fase de análisis tan larga como fuera necesario para definir completamente el dominio. Sin embargo actualmente se acepta que eso en la práctica es imposible [Beck99].

El descubrimiento del dominio es una actividad continua y paralela a todas las demás fases del desarrollo del software. Se descubre *la mayor parte* del dominio en la fase de análisis, pero las fases de diseño e incluso programación son decisivas *para completar* todos sus detalles [Kovitz99]. Por tanto no se trata de evitar los cambios en el dominio; se trata de aceptarlo como algo natural y considerarlo en las distintas fases del desarrollo. Dado que *el cambio no se puede evitar* lo que hay que intentar es que tenga *el menor impacto* posible.

2.2 Tipos de Cambios Comunes

A medida que se vaya avanzando en las fases de desarrollo (análisis, diseño e implementación) se irán produciendo ajustes en el dominio. En la práctica la mayor parte de los cambios que surgen en el dominio son:

- **Cambios en las Entidades**. El dominio puede modificarse para incorporar un nuevo tipo de entidad al modelo o bien para incluir más información sobre una previamente identificada (lo cual se traduce en añadir propiedades a la misma).

- **Cambios en las Relaciones**. Las relaciones son un elemento fundamental en el modelado de objetos. Gran parte de los cambios del dominio suponen reflejar en el modelo alguna relación existente entre dos entidades que inicialmente no se había tenido

en cuenta o bien modificar su cardinalidad. Debido a la frecuencia con la que se presenta este cambio el método de desarrollo deberá permitir la implementación *inmediata* de relaciones entre las distintas clases.

- **Cambios en las Operaciones**. Las operaciones fundamentales del dominio se identifican en la fase de análisis. Sin embargo a medida que dichas operaciones se vayan volviendo una realidad tangible se verán nuevas posibilidades de automatizar nuevas operaciones. Esto es así debido a que generalmente el cliente no tiene una visión completa del producto final que espera. Pero a medida que va viendo algunas operaciones en funcionamiento empieza a darse cuenta de las *posibilidades reales* por lo que comienza a considerar nuevos aspectos de la aplicación.

- **Cambios en las Reglas de Negocio**. Quizás este sea el cambio más frecuente y el que más promueve a los *parches* en el código. No se trata de añadir una operación sino de *concretar la forma en que debe comportarse* una operación *ya existente* ante ciertas circunstancias. Por ejemplo una vez implementada una supuesta operación de facturación puede descubrirse que el usuario no quiere facturar a aquel cliente que tenga ya más de dos facturas impagadas. Estos detalles son los que el cliente suele olvidar mencionar ya que para él son obvios y no cae en la importancia de hacérselos notar a alguien que no conoce el día a día de su empresa[1]. Además dichas reglas pueden afectar a varias operaciones por lo que el impacto puede ser difícil de precisar. En definitiva el proceso de análisis se convierte en un continuo goteo de reglas que hay que ir añadiendo al diseño.

2.3 Impacto de los Cambios en el Diseño y en el Código

En el apartado anterior se mostraron los tipos de cambios más habituales que se suelen presentar en el dominio durante las fases de desarrollo (entidades, operaciones, relaciones y reglas). En este apartado se estudiará *cómo afectan estos cambios* al diseño que se tenga de la aplicación y a su implementación actual.

El dominio determina los requisitos, los cuales a su vez determinan la arquitectura y el diseño que es el que finalmente determina la codificación. Por tanto cada cambio del dominio de los anteriormente expuestos supone cambios en cascada por todas esas fases que acaban, como una gran bola de nieve, en el código que hay que retocar.

2.3.1 Impacto de los Cambios en las Entidades

Si se identifica una nueva entidad habrá que crear una nueva clase con sus propiedades. En el diseño bastará con añadir una nueva clase y posteriormente llevarla a código.

[1] Por ello metodologías como el "Contextual Design" [Beyer02] proponen no basarse en lo que el cliente dice sino en observar lo que hace.

```
class Responsable
{
      String nombre;
}
```

En el caso de añadir nuevas propiedades a una entidad, en lenguajes con control estricto de tipos, habrá que recompilar la clase y todas las que la utilicen (y así recursivamente).

En el caso de añadir una nueva clase no tiene un gran impacto en el código. Sin embargo el verdadero impacto lo supone el añadir el resto de los cambios *que éste trae asociados*: las relaciones con las clases existentes y las nuevas operaciones que incorpora al sistema (ya que una clase nunca se presenta sin que tenga ningún tipo de conexión con otras). El impacto de estos otros cambios es lo que se muestra en los siguientes apartados.

2.3.2 Impacto de los Cambios en las Relaciones

Supóngase ahora que el cambio es añadir una nueva relación entre dos entidades. Llevar dicho cambio al diseño no presenta un gran problema. Basta con dibujar una línea entre las dos entidades a relacionar y detallar la cardinalidad.

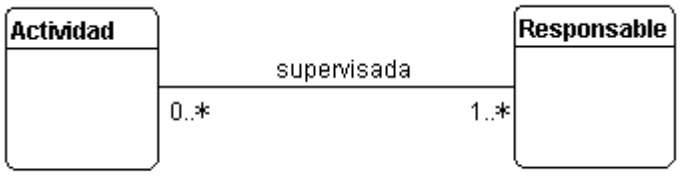

Imagen 2.3-1. Añadir relación en el diseño. Queda llevarlo a código.

Sin embargo, aunque parezca inmediato, el verdadero problema es llevar dicho cambio *del diseño al código*. Un problema recurrente a la hora de llevar a la implementación un diseño orientado a objetos es que por cada dos clases que se quieran relacionar hay que seguir *siempre* la misma secuencia de pasos rutinaria. El primer paso para implementar el cambio es poner un vector en cada una de las clases que almacene las referencias a los objetos relacionados.

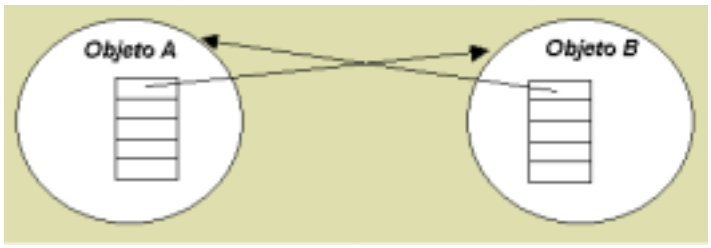

Imagen 2.3-2. Cada objeto debe tener un vector con las referencias a sus asociados

Supóngase que se tiene una clase Actividad y una clase Responsable y se quiere implementar una relación de "supervisión" entre ellas. Para implementar dicha relación en la clase Actividad hay que poner como atributo privado un vector que guarde las referencias a sus responsables.

Para poder mantener este vector hay que añadir al interfaz de dicha clase las distintas operaciones que permitan insertar, consultar y borrar responsables.

```
class Actividad
{
	private Vector responsables = new Vector();

	public void addResponsable(Responsable r)
	{
		responsables.add(r);
	}

	public void removeResponsable(Responsable r) {
		...
	}

	public Responsable getResponsable(String nombre) {
		...
	}

	public Vector getResponsables() {
		...
	}
}
```

Una vez hecho esto hay que repetir el mismo trabajo en la otra clase relacionada: la clase Responsable.

```
class Responsable
{
	private Vector actividades = new Vector();

	public void addActividad(Actividad actividad)
	{
		actividades.add(actividad);
	}

	public void removeActividad(Actividad actividad) {
		...
	}

	public Actividad getActividad(String nombre) {
		...
	}

	public Vector getActividades() {
		...
	}
}
```

Por ahora el único inconveniente que ha aparecido es la repetición del mismo código tedioso en cada una de las clases. Sin embargo quedan más pasos para implementar la relación.

2.3.2.1 Establecimiento de la Conexión

Después de que se hayan definido las operaciones para el mantenimiento de la relación hay que establecer un protocolo de establecimiento y eliminación de la misma. Es decir ¿a cual de los dos objetos hay que comunicar el establecimiento de la relación?

Supóngase que se quiere establecer una relación 1 a 1 entre el responsable "pepe" y la actividad "testing".

```
Responsable pepe = new Responsable();
Actividad testing = new Actividad("Test de Integración");
```

¿Qué sentencia debe escribirse a continuación para relacionar ambos objetos de forma *bidireccional*?

- ¿Se hace mediante la invocación *pepe.addActividad(testing)*?
- ¿O bien se hace mediante *testing.addResponsable(pepe)*?
- ¿O hay que hacer ambas cosas para que la relación no quede en un estado inválido?

En definitiva ¿hay que invocar un método en cada objeto o un sólo objeto se encarga de todo? En éste último caso ¿cuál de los dos objetos es el que se encarga entonces?[2].

No hay una solución generalizada para implementar esto. Distintos frameworks y paquetes de clases toman distintas soluciones. Sin embargo ambos enfoques tienen inconvenientes.

- Si se toma el enfoque de que hay que invocar a un método de cada objeto para formar la relación se corre el peligro de que si se olvida alguna de las dos invocaciones el modelo quedará inestable, lo cual ocurrirá tarde o temprano (al igual que se olvida liberar memoria o cerrar ficheros).

  ```
  pepe.addActividad(testing);
  // Se olvida poner: testing.addResponsable(pepe);

  pepe.getActividades(); // ---> "testing"
  testing.getResponsables(); // --> Ninguno!!!!
  ```

- En las implementaciones basadas en que las relaciones se establecen a través de una sola de las clases (por ejemplo se determina que la relación se establece mediante *Actividad.addResponsable*) dicho método, además de introducir la referencia del Responsable en su vector, deberá acceder al vector privado del responsable para insertarse a si misma como actividad bajo su responsabilidad. Esto supone una violación de la encapsulación y el conocimiento de sus estructuras internas.

[2] Este problema, obviamente, no se presenta en las relaciones unidireccionales. Pero las relaciones por naturaleza son bidireccionales; otra cosa es que por evitar trabajo se implementen en un sólo sentido cuando se tenga la certeza de que ninguna operación requiere ir en sentido contrario (aunque eso ya presupone un conocimiento de todas las operaciones que habrá en un futuro...) Por tanto debe darse solución al caso general de relaciones navegables en ambos sentidos.

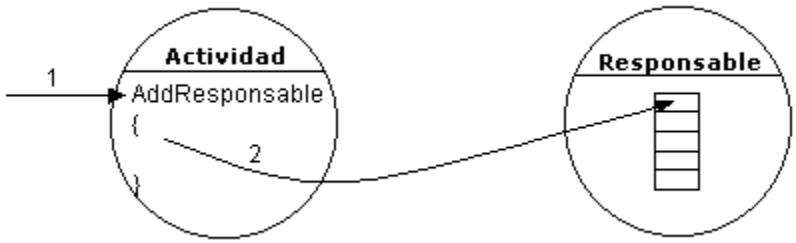

Imagen 2.3-3. La actividad accede al vector privado del responsable para insertarse.

- Otra opción sería que el método *Actividad.addResponsable,* en vez de acceder al vector privado del responsable, usara algún método público del mismo (como por ejemplo *Responsable.addActividad*). Pero eso significaría que un programador podría intentar establecer la relación invocando directamente a éste nuevo método sin pasar por *Actividad.addResponsable*, con lo cual el modelo quedaría corrompido.

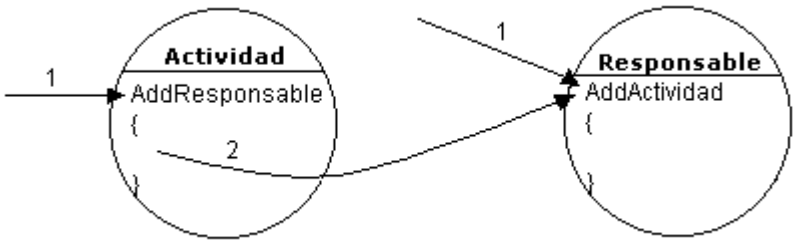

Imagen 2.3-4. La actividad usa un método público para insertarse. Pero éste puede ser llamado por otros métodos.

- Para evitar éste problema podría añadirse código en ambos métodos de tal manera que si se detecta que la llamada no viene del método complementario se invoca a éste para completar la relación (evitando así un bucle infinito).

```
class Actividad
{
	private Vector responsables = new Vector();

	public void addResponsable(Responsable pepe) {
		responsables.add(pepe);

		if (<Si este método no ha sido invocado por “pepe”>)
			pepe.addActividad(this);

	}
...
}
```

```
class Responsable
{
	private Vector actividades = new Vector();

	public void addActividad(Actividad actividad) {
		actividades.add(actividad);

		if (<Si este método no ha sido invocado por "actividad">)
			actividad.addResponsable(this);

	}
...
}
```

Sin embargo en la mayoría de los lenguajes no hay forma de saber quien es el objeto que ha invocado el método (y pasarlo como parámetro no sería una solución práctica).

Una solución que se plantea en algunas ocasiones para averiguar quien es el objeto invocador es suponer que, dado que el método *addResponsable* introduce al responsable en el vector antes de llamar a *pepe.addActividad*, este último método puede saber si ha sido invocado por el responsable si éste ya tiene la actividad en su vector.

```
class Responsable
{
	private Vector actividades = new Vector();

	public void addActividad(Actividad actividad) {
		actividades.add(actividad);

		if (not actividad.getResponsables().includes(this))
			actividad.addResponsable(this);

	}
...
}
```

Esto sin embargo tampoco funciona:

- Por un lado se basa en conocer la implementación de otra clase. Si ésta cambia de implementación todo dejaría de funcionar.

- En el método *addActividad* no se cumpliría la precondición de que un responsable no se puede relacionar con actividades con las que ya esté relacionado. Por tanto se impediría el uso de precondiciones con la falta de protección que ello conllevaría[3].

Por tanto no hay solución general para este problema. Por ello la mayoría de las veces se opta por la opción de establecer la relación a través de una clase predeterminada y que esta viole la encapsulación de la otra accediendo a su vector privado (o mejor dicho, que *debiera* ser privado). Ejemplo de este caso es el paquete Swing de Java [Topley99].

[3] En las aplicaciones que han usado el framework que se presentará en capítulos posteriores se cometía frecuentemente el error de relacionar dos veces dos entidades. Si dicha precondición se hubiera desactivado muchos errores hubieran quedado en el código. Esa opción no es aceptable.

Otros frameworks como JHotDraw [Johnson92] no rompen la encapsulación ya que acceden al objeto opuesto a través de un método público. Pero si por desconocimiento se invoca este último método sin pasar por el que su programador había previsto el modelo queda corrompido.

2.3.2.2 Eliminación de la Relación

Una vez implementada la forma de establecer la relación el trabajo no ha acabado aún. Todavía queda más trabajo para mantener la integridad de la relación. Hay que tener en cuenta que si se elimina la relación debe quedar eliminada en ambos extremos.

Eso suponer *repetir todo el código anterior* (y sus irresolubles problemas) pero ahora para eliminar las referencias de los vectores. Hay que evitar que uno de los dos extremos siga relacionado con el otro.

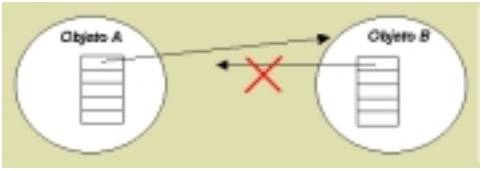

Imagen 2.3-5. El borrado de la relación debe eliminar las referencias en ambos objetos.

```
class Responsable
{
	private Vector actividades = new Vector();

	public void removeActividad(Actividad actividad) {
		actividades.remove(actividad);

		// ¿Se borra aquí el otro extremo o no?

	}
...
}
```

Nótese como a la hora de poner los ejemplos no se detalla todo el código debido a lo tedioso que sería. Sin embargo no hay que olvidar que a la hora de codificarlo realmente habría que soportar ese tedio.

2.3.2.3 Implementación de Eventos

Todavía quedan más pasos para implementar las relaciones. Tanto la clase Actividad como la clase Responsable del ejemplo anterior, viéndolas desde la perspectiva de la arquitectura MVC [Buschman96], serían clases del modelo. La representación de ellas en pantalla (por ejemplo mediante un Pert o un Gantt) sería realizada por otras clases que cumplieran el rol de vistas.

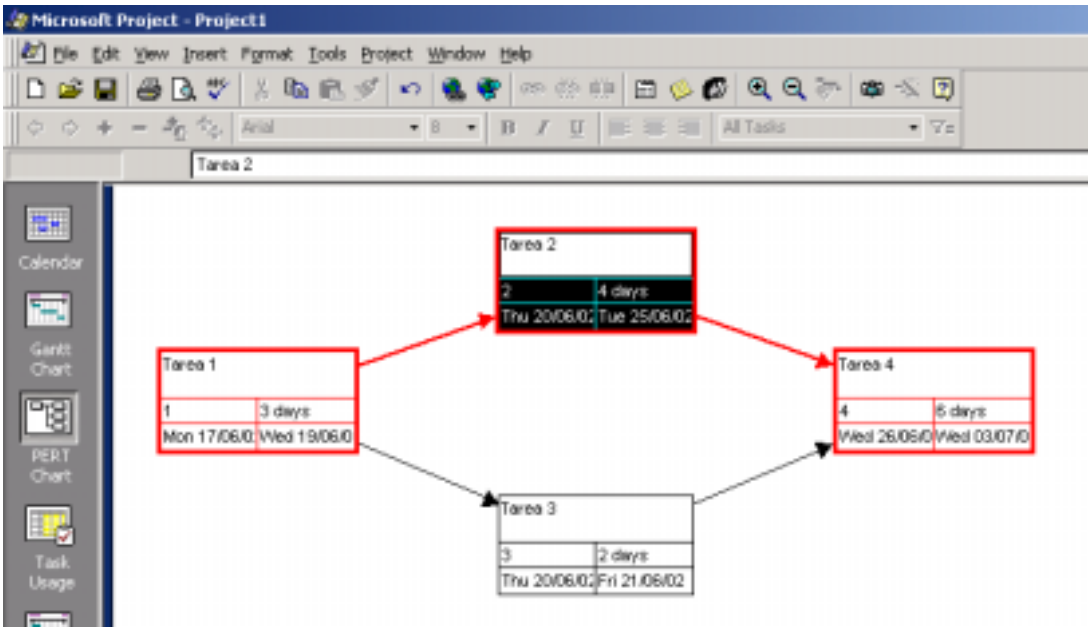

Imagen 2.3-6. Vista del proyecto en modo Pert

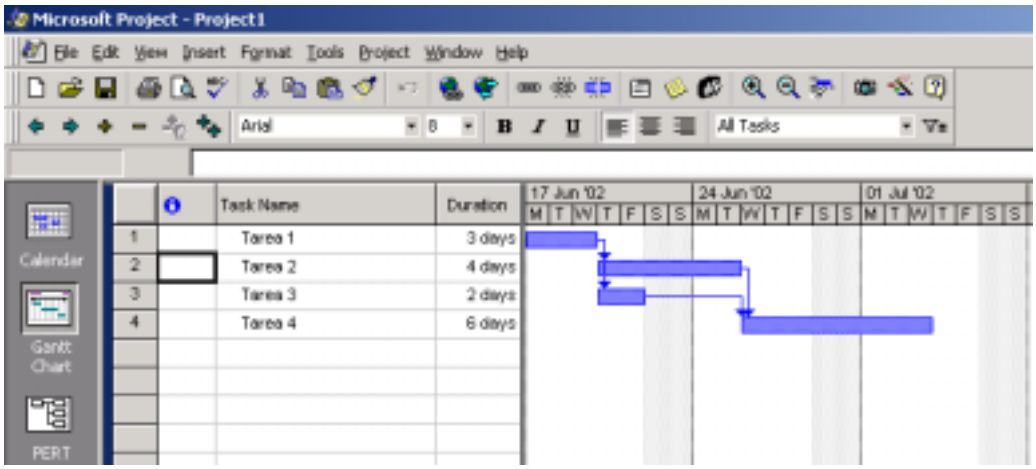

Imagen 2.3-7. Vista del mismo proyecto en modo Gantt

Y como clases del modelo que son entonces deberán lanzar eventos cada vez que cambie su estado para que sus vistas puedan ser notificadas. Es decir, deben seguir el patrón Observer [Gamma94] el cual es la base de la arquitectura MVC.

Por tanto ahora comienza *todo el trabajo* de crear las clases que representan los eventos así como la interfaz (*suscriptor*[4] en el argot Java) que deberán implementar las vistas junto con los métodos del modelo que permitan registrarlos y notificarlos [Sun96]. Lo máximo que se ha

[4] En este texto la palabra inglesa *listener* será traducida como *suscriptor*.

hecho para mitigar esta tediosa tarea es incluir herramientas (llamadas Wizards) en los entornos de desarrollo para automatizarlo en lo posible.

A continuación se muestra el código a implementar para una simple clase *Persona* que pueda relacionarse con *Mascotas* de tal manera que avise de los cambios en su estado mediante eventos. Nótese que faltaría también el código equivalente en la clase Mascota, el cual no se incluye.

```
// Ejemplo generado por JBuilder 3

public class MascotaEvent extends EventObject
{
	public mascotaEvent(Object source)
	{
		super(source);
	}

	<faltan por añadir aquí los atributos del evento>
}


public interface MascotaListener extends EventListener
{
	public void mascotaAdded(mascotaEvent e);
	public void mascotaRemoved(mascotaEvent e);
	public void attributeChanged(mascotaEvent e);
	public void attributeAdded(mascotaEvent e);
	public void attributeRemoved(mascotaEvent e);
}


public class Persona
{
	private transient Vector mascotaListeners;
	private java.util.ArrayList mascotas;

	public Persona() {
	}

	public List getMascotas() {
		return new List(mascotas);
	}

	public void addMascota(Mascota fufi) {
		// <falta comprobar que no está ya y lanzar excepciones>
		mascotas.add(fufi);
		fireMascotaAdded(new mascotaEvent(this));
	}

	public void removeMascota(Mascota fufi) {
		// <falta comprobar que está ya y lanzar excepciones>
		mascotas.remove(fufi);
		fireMascotaRemoved(new mascotaEvent(this));
	}
```

```
	public synchronized void removemascotaListener(mascotaListener l)
{
		if(mascotaListeners != null &&
mascotaListeners.contains(l)) {
			Vector v = (Vector) mascotaListeners.clone();
			v.removeElement(l);
			mascotaListeners = v;
		}
	}


	public synchronized void addmascotaListener(mascotaListener l) {
		Vector v = mascotaListeners == null ? new Vector(2) :
(Vector) mascotaListeners.clone();
		if(!v.contains(l)) {
			v.addElement(l);
			mascotaListeners = v;
		}
	}


	protected void fireMascotaAdded(mascotaEvent e) {
		if(mascotaListeners != null) {
			Vector suscriptores = mascotaListeners;
			int count = suscriptores.size();
			for (int i = 0; i < count; i++) {
				((mascotaListener)
suscriptores.elementAt(i)).mascotaAdded(e);
			}
		}
	}

	protected void fireMascotaRemoved(mascotaEvent e) {
		if(mascotaListeners != null) {
			Vector suscriptores = mascotaListeners;
			int count = suscriptores.size();
			for (int i = 0; i < count; i++) {
				((mascotaListener)
suscriptores.elementAt(i)).mascotaRemoved(e);
			}
		}
	}

	<además faltan aquí los métodos relativos a los eventos de las
propiedades>
}
```

Como puede comprobarse es demasiado código engorroso que oscurece la implementación. Sin embargo es necesario implementarlo si se quieren un diseño mínimamente flexible.

2.3.2.4 Resumen Relaciones

Nótese todo el trabajo que hay que hacer para añadir *una sola relación*. Y posiblemente *haya que borrarlo todo* cuando en el próximo cambio de dominio se decida *eliminar dicha relación* o bien alguna de las clases relacionadas ¿Sería soportable esta forma de codificación en una aplicación real de cientos de clases y relaciones?

Aparte de la cantidad de código otra cosa es la depuración del mismo. La gestión de relaciones entre objetos recuerda enormemente a la gestión de memoria en lenguajes tipo C. En éstos

lenguajes *sólo* había que acordarse de reservar memoria antes de usarla y de liberarla una vez acabada con ella. Esta regla aparentemente simple era el origen de los errores de programación más difíciles de encontrar [Maguire93]. De la misma manera aquí *sólo* hay que acordarse de establecer la relación en el objeto correcto y de eliminarla de la forma adecuada (que como se vió no es estándar). Cualquier olvido producirá modelos de objetos inconsistentes.

Se tiene que las relaciones son algo fundamental del dominio. La inclusión de relaciones *es de los cambios más habituales* del mismo. Y sin embargo las tecnologías orientadas a objetos, además de exigir gran cantidad de código, *no tiene solución* para implementarlas adecuadamente.

2.3.3 Impacto de los Cambios en las Operaciones

En este apartado se está mostrando el impacto que produce en el código cada uno de los cambios habituales del dominio. Hasta el momento se ha visto el impacto que supone el añadir nuevas entidades y añadir nuevas relaciones entre ellas. La decisión de incorporar *nuevas operaciones* al modelo supone otro cambio importante del diseño.

2.3.3.1 Impacto en varias Clases

En las tecnologías orientadas a objetos las operaciones se realizan entre varios objetos que se intercambian mensajes los cuales son despachados por sus métodos. Por tanto la ejecución de una operación consiste en la ejecución de una secuencia de métodos de una o más clases.

Por tanto al añadir una nueva operación habrá que determinar qué clases son las afectadas y habrá que repartir las responsabilidades entre ellas. Es decir, habrá que modificar varias clases *para añadirlas los métodos* que llevan a cabo la operación. Por tanto el código total de la operación quedará repartido por distintos módulos. Los diagramas de interacción de objetos de las distintas metodologías[5] precisamente surgen de la necesidad de documentar de alguna manera esta dispersión.

[5] El nombre aquí utilizado se refieren al Método Unificado [Jacobson99]. Aunque en otras notaciones tienen distintos nombres todas tienen algún diagrama equivalente que muestre la participación de los distintos objetos en la operación.

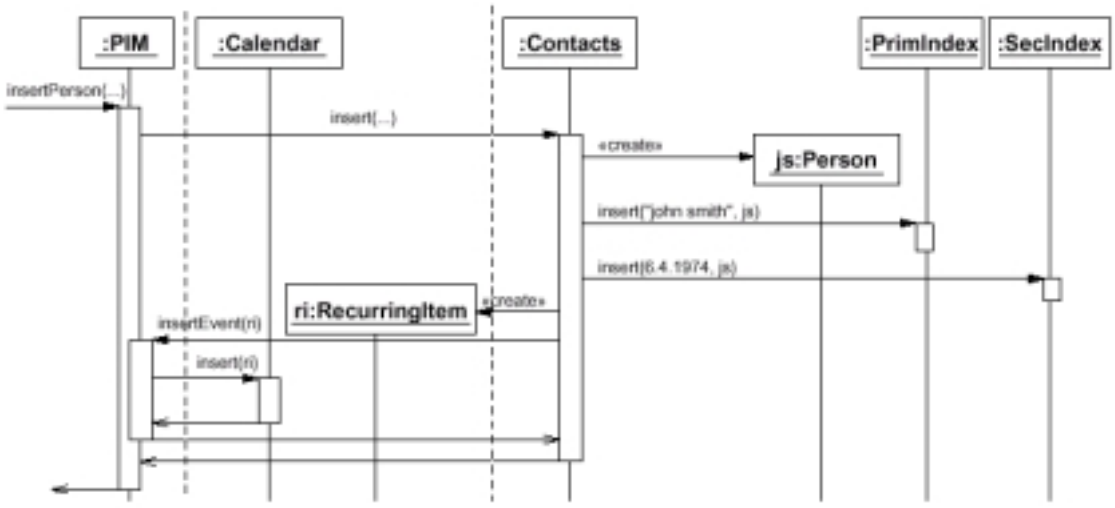

Imagen 2.3-8. Una operación está distribuida en varias clases.

Este problema ya fue señalado en [Lauesen98].

> "La tecnología orientada a objetos no ha conseguido sus expectativas cuando es aplicada a aplicaciones de negocio reales en parte debido al hecho de que no hay un lugar en el que poner las operaciones de alto nivel que afectan a varios objetos.
>
> ... si se implementa en las clases involucradas es imposible obtener una visión del flujo de ejecución"

Una constante durante el desarrollo de una aplicación orientada a objetos es tener que estar modificando las mismas clases a medida que se van incorporando más operaciones del dominio.

Aunque en teoría las clases servían para aislar código de manera que se evitara la propagación de cambios sin embargo en la práctica las clases se están modificando constantemente hasta el último día de desarrollo. Las operaciones son la causa de éste problema. Curiosamente las clases (como módulo) posiblemente se vean *más afectadas por el cambio* de lo que se veían construcciones más primitivas como las struct de C o los Record de Pascal, ya que éstos no veían afectadas sus definiciones por la incorporación posterior de nuevas operaciones que las utilizaran.

Curiosamente la librería STL de C++ [Stepanov95], un ejemplo ya clásico de diseño avanzado orientado a objetos y el mayor exponente de la Programación Genérica [Jazayeri98], tomó como una de sus decisiones fundamentales de diseño no utilizar métodos para evitar este problema [Czarnecki00].

2.3.3.2 Tratamiento de Fallos en las Operaciones

Meyer [Meyer97] define dos propiedades fundamentales de un componente software (de las operaciones como caso particular):

- **Corrección** es la capacidad de realizar su tarea exacta tal y como se definió en las especificaciones.

- **Solidez** (*Robustness*) es la capacidad para reaccionar apropiadamente a las situaciones anormales.

Generalmente se presta atención a la corrección de las operaciones pero su solidez no suele considerarse de manera exahustiva. Sin embargo no basta con suponer que nunca se presentarán casos anormales o que si se presentan no es responsabilidad del desarrollador gestionarlos (ya que no estaban en las especificiones). La responsabilidad del desarrollador es mantener *la integridad* y seguridad de los datos *ante toda situación*.

Además el problema de la solidez se acentúa ahora que las aplicaciones están expuestas a nuevas capas intermedias con las que debe interactuar y que son *fuentes potenciales de multitud de casos anormales* (conexiones a Internet, documentos XML, middlewares de distribución, etc.)

En definitiva a la hora de ejecutar una operación pueden presentarse imprevistos que impidan su ejecución; desde errores en dispositivos hasta errores en la propia codificación. En ambos casos la operación obviamente no puede hacer nada por continuar, pero lo que tampoco debe hacer es dejar el modelo en un *estado inválido*. Lo deseable sería que se indicara al usuario la imposibilidad de ejecutar dicha operación y bien permitirle ejecutar otras operaciones o bien finalizar la aplicación; pero en cualquier caso conservando los datos tal y como estaban antes de comenzar la operación que produjo el fallo.

Sin embargo ésto es muy difícil de obtener en la práctica ya que:

- Aunque lenguajes como Java obligan a atrapar las excepciones, lo que no pueden hacer es determinar lo que hay que corregir debido a la excepción. El tratamiento de excepciones provee de un lugar donde poner el código de recuperación (bloque *catch*), pero no evita la tarea de tener que averiguar cual es *el código adecuado* a colocar en él. Un fallo del programador al no identificar todas las rectificaciones necesarias ante la excepción dejará errores en el modelo. Este problema se acentúa si hay que tener en cuenta las acciones realizadas por otros métodos invocados. Y todo ello habrá que tenerlo en cuenta a la hora de introducir cambios de mantenimiento de la operación.

- En cualquier caso el intentar plantearse *qué habría que rectificar si ocurre cualquier error en cada una de las sentencias del método* es una labor difícil y que, aún en el caso de hacerlo correctamente, oscurecería la lógica de la aplicación y lo haría impracticable.

La solidez debe formar parte de *toda operación*, por lo que lo natural es que la tecnología de desarrollo utilizada dé un *soporte directo* a la misma. Como ya se exigió anteriormente con las relaciones las tareas más comunes de desarrollo deben tener las soluciones más rápidas y eficaces para trasladarse a código. E implementar la solidez de las operaciones *debiera ser* una tarea común. Sin embargo actualmente la solidez debe obtenerse de manera particular en cada método omitiéndose por tanto la mayor parte de las veces.

En definitiva la orientación a objetos *tampoco presenta una solución eficaz* para obtener *operaciones sólidas*.

2.3.4 Impacto de los Cambios en las Reglas de Negocio

Una vez vistos los impactos de los cambios de entidades, de relaciones y de operaciones finalmente se mostrará el impacto de las reglas de negocio en el código. Este último cambio del dominio podría considerarse como un caso particular del cambio de operaciones tratado en el apartado anterior. Al fin y al cabo añadir condiciones a las operaciones es una forma de modificarlas. Sin embargo reciben un tratamiento diferenciado ya que son menos estables que las operaciones. Una misma operación puede tener asociadas distintas reglas de negocio que se van añadiendo durante el desarrollo. El proceso de facturación siempre existe pero inicialmente se puede establecer que no se permita hacer una factura a un determinado tipo de cliente y más tarde se puede decidir no facturar si el importe no llega a un mínimo.

La mayor parte de las modificaciones que sufren las operaciones es por ir descubriendo nuevas reglas a medida que se va concretando el dominio. Además una regla puede afectar a varias operaciones por lo que habrá que determinar cuales son estas y repetir los cambios del apartado anterior en cada una de ellas. Ello no solo supone modificaciones en varias clases sino el peligro de que quede sin modificar una operación que debiera ser controlada por la regla y continúe por tanto ejecutándose *sin respetarla*.

Por ejemplo supóngase que se decide que una persona no puede ser responsable de más de una actividad en el mismo periodo de tiempo:

```
class Actividad
{
      private Vector responsables = new Vector();

      public void addResponsable(Responsable pepe) {

            if (pepe.ocupado(this.periodo()))
                  return;      // Rechazar responsable

            responsables.add(pepe);
            ...
      }
...
}
```

Las reglas se están mezclando con las operaciones de forma estática. Esto supone:

- Tener que modificar las clases cada vez que haya que cambiar las operaciones debido al descubrimiento de nuevas reglas.
- No poder cambiar las reglas en tiempo de ejecución

2.4 Resumen

La Tecnología Orientada a Objetos no soportan adecuadamente las variaciones de dominio. Hasta los cambios *más habituales* (de relaciones, operaciones, reglas, etc.) provocan numerosos y rutinarios cambios en el diseño y en el código.

Todo el trabajo que se ha descrito (y cuyo código se ha omitido para simplificar) ha sido generado para un modelo de *únicamente dos clases*. A medida que el modelo comience a crecer en número de entidades se acabará convirtiendo en un sumidero de código tedioso que hay que teclear y testear.

El hecho de que cambios tan básicos produzcan tanto trabajo supone:

- Una mayor lentitud en el desarrollo al tener que estar rehaciendo continuamente las clases.

- Una mayor cantidad de errores de codificación. Esto es así porque, al contrario de lo que pueda parecer, el mayor número de errores de programación no aparece en las rutinas más complejas. En estas rutinas es donde el programador pone mayor atención y evalúa con mayor detalle todas las situaciones posibles. Sin embargo es en las tareas repetitivas y tediosas donde el programador relaja la atención y comete errores (no inicializar una variable, no reservar memoria, etc) que son curiosamente los que hacen perder la mayor parte del tiempo de depuración para encontrarlos [Maguire93].

Por tanto, dado que el proceso de desarrollo de software es un continuo refinamiento del dominio, la orientación a objetos *no es suficientemente ágil* para dar soporte a las necesidades del desarrollo de software.

La orientación a objetos tiene muchas virtudes y una de las que primero destacó fue la naturalidad de realizar el análisis basándose en los objetos que se iban encontrando en el mundo real. Una vez conseguido un análisis orientado a objetos otra característica importante era que la *transición del análisis al diseño es totalmente gradual*. Los diagramas son muy parecidos y a veces cuesta distinguir a cual de las dos fases pertenece un diagrama. Esto contrastaba con las notaciones tan distintas que se usaban en cada una de las fases en la época del diseño estructurado.

Sin embargo aunque la orientación a objetos ha conseguido suavizar la transición entre las primeras fases del desarrollo sigue teniendo *un salto muy grande a la hora de pasar del diseño a la implementación*.

3 Objetivo de la Tesis

3.1 Modelado del Dominio en un Sistema Informático

El objetivo de esta tesis es disminuir el tiempo requerido para la representación del dominio y de sus posteriores cambios durante el desarrollo en las aplicaciones basadas en la tecnología orientada a objetos.

Es decir, se busca un método que una vez identificados los distintos elementos del dominio indique cual es la forma adecuada de llevarlos al diseño y posteriormente a código de tal manera que los cambios posteriores de dichos elementos tengan el menor impacto posible en el tiempo de desarrollo.

Este método deberá suponer por tanto la unión de un conjunto de *patrones de diseño* y *de codificación* cuya adecuada combinación dará lugar a un *Patrón Arquitectónico* para el modelado de dominios de tal manera que se eviten los problemas de las Tecnologías Orientadas a Objetos descritos anteriormente.

3.2 Traslación entre Dominios

Un objetivo derivado del anterior es promover la reutilización de los distintos elementos del dominio en distintas aplicaciones (datos, operaciones, reglas, etc.)

Es decir por un lado se pretende conseguir disminuir los efectos de los cambios en el dominio sobre las clases. Pero cuando se pretende reutilizar una clase en otra aplicación lo que se está haciendo realmente es *cambiarla de dominio* (cada aplicación tiene su dominio). Entonces esa traslación de la clase entre dominios, en lo que a ella respecta, *se podría considerar como un cambio* del primer dominio hacia el segundo. Por tanto no solo se simplificaría adaptar las clases a los cambios sino que se simplificaría el hecho de reutilizar clases en otras aplicaciones con otros dominios.

No se propone aquí una reutilización de clases encapsuladas. La reutilización no debe centrarse únicamente a intentar producir clases que puedan ser reutilizadas sin cambios en distintos dominios. Por definición distintos dominios dan lugar a distintos diseños con distintas clases y responsabilidades. Por tanto en la práctica para poder reutilizar una clase tradicional:

- Tendría que ser en otro dominio igual (con los mismos atributos, relaciones, operaciones y reglas) – lo cual es muy difícil que ocurra.

- O bien tienen que ser clases que sean totalmente independientes de cualquier dominio (lo cual sólo ocurre en las clases de utilidad).

La prueba de ello es ¿Cuantas veces se ha implementado la clase Persona? ¿Cuantas veces se ha reutilizado? Dado que es imposible que dicha clase se haya implementado con las mismas responsabilidades en distintas aplicaciones se acaba antes reimplementándola que reutilizándola.

Por lo tanto aquí se intenta como subobjetivo un grado de reutilización intermedio que no sea todo o nada. Para que un objeto pueda reutilizarse no tiene que ser exactamente lo que se necesita en la nueva aplicación sino que mediante sus oportunas combinaciones con otros elementos primitivos se adapte a lo requerido en dicho dominio.

3.3 Restricciones

Para alcanzar los objetivos anteriormente expuestos se han planteado una serie de restricciones que deberá respetar la solución propuesta.

Estas restricciones vienen derivadas de la intención de obtener ventajas *inmediatas* con las herramientas y lenguajes comerciales actuales en uso, puesto que la experiencia demuestra que otras estrategias generalmente no acaban trasladándose al mundo comercial y profesional *que es lo que se pretende*.

- La solución no debe basarse en tener que utilizar un nuevo lenguaje de programación con nuevas características. La solución debe poder aplicarse con los lenguajes orientados a objetos existentes para poder aprovechar los entornos de desarrollo y las potentes librerías de éstos. Como condición imprescindible debe poder ser aplicada en lenguajes como Java [Gosling96], C++ [Stroustrup98], Smalltalk [Goldberg89], C# [Microsoft00b] o Eiffel [Meyer97], algunos de los cuales tienen control estricto de tipos. Estos son los lenguajes más usados en el desarrollo de aplicaciones comerciales que son las que sufren, al fin y al cabo, los problemas anteriormente expuestos. Proponer una solución que excluya a éstos lenguajes producirá *una exclusión automática de la solución*.

- Tampoco la solución deberá exigir hacer variantes a dichos lenguajes modificándolos con nuevas instrucciones o compiladores adaptados. Deben poder seguir usándose los entornos de desarrollo profesionales de dichos lenguajes. La solución debe ser aplicable con cualquier entorno de desarrollo.

- La solución tampoco puede ser un framework que necesite dirigir el hilo de ejecución de la aplicación y colisione por tanto con otras librerías o frameworks como Servlets [Allamaraju00], Applets [Geary97], JHotDraw [Johnson92], etc.

Sección 2. Tecnologías Relacionadas

4 Panorámica Actual

En este capítulo se presentan distintas tecnologías, arquitecturas o metodologías que pretenden alcanzar objetivos similares a los planteados en esta tesis. De esta manera se podrá observar de qué forma se han abordado los problemas aquí expuestos y de ésta manera comparar su adecuación al caso concreto aquí estudiado.

4.1 Extreme Programming

Extreme Programming [Beck99] es una metodología de las denominadas *ligeras* dirigida a equipos de desarrollo de tamaño medio o pequeño que desarrollan software en un entorno de cambio rápido de requisitos. Se distingue de otras metodologías en:

- Sus tempranos, concretos y continuos feedback debido a los cortos ciclos.
- Su planificación incremental que rápidamente produce un plan que se espera que evolucione a lo largo de la vida del proyecto.
- La flexibilidad a la hora de secuenciar la implementación de la funcionalidad requerida en función de las necesidades variables del negocio.
- Su soporte sobre tests automatizados escritos por programadores y clientes para monitorizar el progreso del desarrollo, permitir la evolución del proyecto y detectar las variaciones de manera temprana.
- Su apoyo en la comunicación oral y código fuente como forma de comunicación de la estructura del sistema y su intención.
- Su apoyo en un diseño revolucionario que dura tanto como dure el sistema.
- Su utilización de prácticas que combinan los instintos a corto plazo de los desarrolladores con los intereses a largo plazo del proyecto.

Ninguna de las ideas de Extreme Programming son nuevas si se las examina por separado. Muchas vienen de los primeros tiempos de la programación. Pero la innovación de esta metodología es:

- Juntar todas estas prácticas de forma coherente.
- Asegurarse de que se practican de forma concienzuda.
- Asegurarse que cada una da soporte a las demás de tal manera que cubran los inconvenientes que planteaban cuando se las usaba individualmente.

4.1.1 El Coste del Cambio

Un axioma de la ingeniería del software es que el coste de cambiar un programa crece exponencialmente con el tiempo. Un problema que puede costar X si se detecta en el análisis podría costar miles de veces más si se detecta cuando el software está en producción.

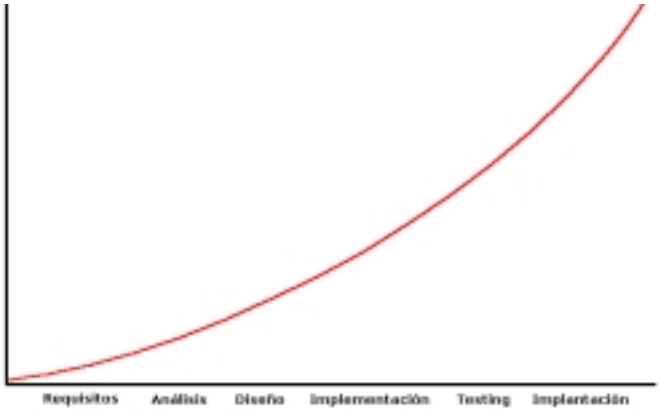

Imagen 4.1-1. El coste del cambio en las distintas fases

Por tanto se intenta tomar todas las decisiones en la primera fase para que no surjan cambios en las demás. Esta primera fase se convierte en un continuo: ¿hará falta hacer esto? ¿se presentará esta situación?

Sin embargo Extreme Programming plantea lo siguiente: ¿Y si el coste del cambio no fuera así? ¿Y si tuviera esta otra forma?

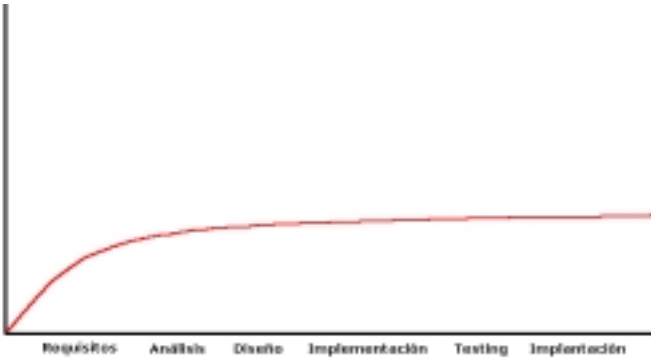

Imagen 4.1-2. ¿Y si el coste fuera así?

Si la curva tuviera esa forma ¿no cambiaría la forma de hacer software? ¿Tendría sentido plantearse de manera anticipada lo que va a hacer falta en la aplicación? ¿No sería más práctico ir añadiendo solo lo que *seguro* se necesita actualmente y luego ir cambiando cosas a medida que se presenten?

Beck afirma que ha visto proyectos pasar semanas o meses intentando decidir qué hacer en vez de hacer lo que necesitan. En cualquier caso cuando la implementación avanzó tuvieron que hacer cosas que realmente no habían previsto y no utilizaron gran parte de las que si lo habían sido.

Extreme Programming, en vez de centrarse en tomar grandes decisiones al principio, se basa en una aproximación a la codificación que se basa en decisiones rápidas y seguras y lo codifica de manera que se asuma que habrá que modificarlo a medida que se descubran nuevas cosas. Hay que cambiar la actitud de que el código haya que hacerlo suficientemente flexible y bien diseñado para que no haya que cambiarlo si cambian los requisitos. El código hará únicamente lo que haya que hacer actualmente y habrá que cambiarlo seguro. Pero el hecho de que solo se contemple lo que hay que hacer dará lugar a un diseño extremadamente simple. Eso supone una mayor eficacia en la programación al centrarse en una actividad más simple y una mayor sencillez a la hora de introducir los esperados cambios.

Pero para que esto sea posible se deben cumplir tres condiciones:

- Un diseño simple con código evidente (patrones, convenios de codificación, etc).
- Tests automatizados.
- Actitud ante los cambios constantes.

Es mejor hacer una cosa sencilla hoy y mejorarla si hace falta reutilizarla que hacer un megacomponente reutilizable que puede que nunca se use. Extremme Programming no propone módulos reutilizables para un futuro. Propone crear un módulo reutilizable una vez que se haya utilizado en varios sitios mediante su modificación.

Beck afirma que desarrollar proyectos software es como conducir un coche. No se puede determinar cual es la dirección correcta para ir a otra ciudad y después de alinear el coche en dicha dirección cerrar los ojos y acelerar. Se trata de mantener los ojos en la carretera e ir girando el volante en función de lo que vaya apareciendo en el camino (cruces, desvíos por obras, atascos, atajos, etc.). Por tanto no se trata de prever todos los atascos e incidencias que vayan a ocurrir. Se trata de dirigir el coche en cada momento en la dirección adecuada.

4.1.2 Métodos de Extreme Programming

Se puede pensar que Extreme Programming está manifestando el sueño de todo estudiante de informática: a programar sin analizar ni diseñar!!. Sin embargo esto no es así en absoluto. Simplemente acota estas fases para que consideren únicamente el momento actual y no el futuro. Con esta limitación no solo no impide el diseño sino que enfatiza su importancia en cuanto a sencillez y elegancia (fundamentado en patrones y refactoring [Opdyke93]).

Algunos de los métodos de Extreme Programming para conseguir sus objetivos son:

- *El Juego de la Planificación*. Determinar rápidamente el alcance de la siguiente versión en función las prioridades del negocio y las estimaciones técnicas. Tan pronto como la realidad sobrepase el plan rehacerlo.
- *Versiones pequeñas*. Implementar completamente aquella funcionalidad que más valor supone para el cliente y hacerla funcionar. Da igual si el proyecto se cancela antes o después: lo que se haya hasta el momento será funcional. No hay que hacer proyectos con varias funcionalidades que si se decide anular por los tiempos o los costes no se tenga nada acabado y funcional.

- *Diseño simple*. El sistema deberá tener el diseño más simple que solucione el problema actual. Las partes para futuras ampliaciones que dan complejidad se extraen inmediatamente. Beck afirma que el diseño adecuado es aquel que realice la funcionalidad con las menos clases y métodos posibles.

- *Testing*. Antes de implementar cada módulo se construyen sus tests. El módulo estará acabado cuando todos los tests se ejecuten correctamente. Esto tiene una doble utilidad. Por un lado obliga a pensar cual debe ser el comportamiento desde el exterior antes de empezar a codificarlo. Por otro lado permite ejecutar dichos tests cada vez que se cambie el diseño de tal manera que lo que antes funcionaba deberá seguir funcionando.

- *Refactoring.* El refactoring son un conjunto de patrones de codificación para modificar el código ante nuevos requisitos. Permiten reestructurar el código para remover duplicaciones, mejorarlo o simplificarlo.

  Supóngase que se parte de un dominio A y se llega a B. ¿Se obtendrá el mismo diseño que si se parte de B directamente? Se debería. Sin embargo no es así en la práctica. El diseño que proviene de A suele estar cargado de parches y no muestra un diseño tan limpio y obvio como el que proviene directamente de B.

  El refactoring, ante un cambio del dominio, propone hacer el mejor diseño posible para la nueva situación actual. Se consigue que independientemente de por cuantos cambios haya pasado la aplicación éstos no hayan dejado huellas y siempre se tenga el diseño más simple posible. Esto facilita las modificaciones y el mantenimiento del programa drásticamente. En caso contrario lo que se va obteniendo es la típica aplicación monstruo que se va convirtiendo en un sumidero de parches y que se convierte en una pesadilla de modificar y depurar.

- *Propiedad Colectiva del Código*. Cualquiera puede cambiar el código de cualquier módulo. No hay un programador asignado a cada sitio. Esto fomenta un estilo de codificación más transparente y que ningún módulo se convierta en el coto privado de los parches de un programador.

- *Estándares de Codificación*. Los programadores siguen unas normas de codificación que enfatizan la comunicación a través del código. La documentación de la aplicación, al menos en estas fases de tanto cambio, debe ser el propio código. Los lenguajes orientados a objetos modernos permiten un código claro y basado en patrones que se auto-explique sin necesidad de estar generando documentación que rápidamente haya que cambiar.

4.1.3 Adaptación a los Objetivos de la Tesis

Esta metodología se basa en la completa naturalidad, incluso el fomento, de los cambios en el código a medida que vaya cambiando el dominio. No hay nada intocable en el diseño.

Pero esto tiene un problema. Extreme Programming funciona si los cambios en el dominio se trasladan rápidamente al diseño y a la codificación. Por tanto los problemas inicialmente expuestos toman *un mayor protagonismo*.

La Arquitectura RDM que posteriormente se presentará está especialmente pensada como complemento de Extreme Programming. Ésta metodología no aborda los problemas expuestos al inicio de la tesis ya que supone que el dominio se llevará al sistema informático siguiendo el modelado tradicional de la orientación a objetos. Pero una vez observados los inconvenientes de ésta forma de modelado esta suposición no encaja con la filosofía del resto de la metodología y supone un lastre para mantener el ritmo de desarrollo propuesto basado en ciclos ultra cortos.

4.2 Arquitectura Modelo-Vista-Controlador

La arquitectura Modelo-Vista-Controlador fue propuesta por Trygsve Reenskaug durante su estancia en Xerox Palo Alto en los años 70 [Reenskaug95].

Posiblemente sea la arquitectura más aplicada en los diseños orientados a objetos y una de las que más variantes ha tenido para adaptala a distintos contextos (por ejemplo a Swing [Sundsted98] o JSP [Duffey01]).

4.2.1 Descripción de la Arquitectura

La arquitectura Modelo-Vista-Controlador también surge de la necesidad de separar elementos para conseguir mayor independencia entre ellos. Está formada por tres roles[6]:

- Modelo será aquel objeto que guarde los datos de la aplicación.
- Vista será aquel que sepa como representar el modelo en distintos formatos o dispositivos y requiera estar actualizado en tiempo real a medida que se produzcan cambios en el modelo.
- Controlador será aquel objeto que modifique el modelo.

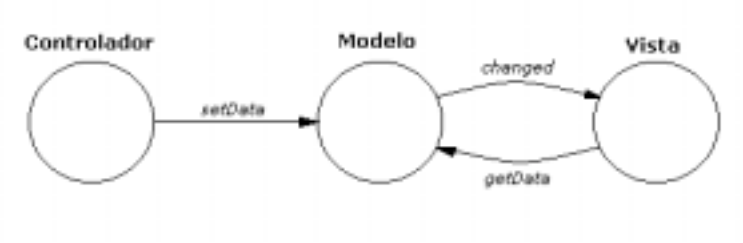

Imagen 4.2-1. Los tres roles de la arquitectura MVC y los mensajes que se intercambian.

[6] Es importante remarcar que son *roles* y no clases, ya que es habitual que haya clases que cumplan a la vez el rol de *vista* y de *controlador* (por ejemplo una ventana que permita la edición del modelo).

La comunicación entre controlador y modelo es directa mediante la invocación de los métodos de éste último. Sin embargo la comunicación entre el modelo y las vistas no puede ser así: se pretende que el modelo no sepa cuantas vistas tiene de tal manera que éstas puedan cambiar dinámicamente. El añadir un *nuevo tipo de vista no debe suponer un cambio del modelo*.

Imagen 4.2-2. El modelo no se debe ver afectado por el número y tipo de vistas que tenga

Por tanto se utiliza el patrón Observer [Gamma94][7]. Este patrón se basa en que el modelo permita la suscripción de una serie de objetos que cumplan un determinado interfaz de tal manera que envíe un mensaje a todos ellos cada vez que cambie su estado.

4.2.2 Adecuación a los Objetivos

La arquitectura Modelo-Vista-Controlador tiene algunos inconvenientes a la hora de aplicarla al modelado rápido de los elementos del dominio. Esencialmente no permite la reutilización de los distintos elementos de la arquitectura de forma independiente.

4.2.2.1 Cambios del Modelo

Tanto los controladores como las vistas conocen a las clases del modelo que manipulan. Por tanto cualquier cambio en dichas clases supondrá una modificación de los demás elementos de la arquitectura.

En este trabajo se pretende que cualquier cambio del modelo que no afecte a la funcionalidad básica de otro elemento tampoco afecte a su implementación. Aquí sin embargo el hecho de añadir una propiedad al modelo afecta al controlador aunque no utilice dicha propiedad. Lo mismo ocurre cada vez que haya un cambio en el interfaz de notificación del modelo. Las vistas se verán afectadas aunque no necesiten dicha notificación.

[7] En realidad no es que la arquitectura utilice el patrón Observer sino más bien que el patrón Observer se ha creado de abstraer la forma en la que se comunican el modelo y la vista en esta arquitectura.

4.2.2.2 Reutilización de Vistas

Un problema que se presenta habitualmente en ésta arquitectura es cuando se quiere reutilizar una clase vista con varios modelos.

Supóngase que se tiene una vista denominada *VistaIcono* y un modelo *Actividad*.

```
class Actividad
{
	void addVistaActividad(VistaActividad vt) {
		...
	}
}


interfaz VistaActividad
{
	void actividadChanged(Actividad t);
}


class VistaIcono implements VistaActividad
{
	VistaIcono(Actividad vt, Icono c) { ... }

	void actividadChanged(Actividad t) {
		<dibujar icono y debajo nombre de la actividad>
	}
}
```

El resultado de la VistaIcono puede verse a la derecha de la siguiente imagen.

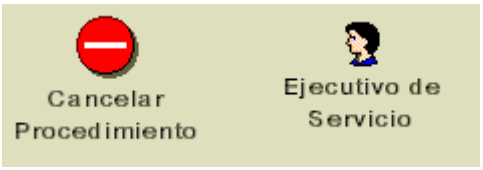

Imagen 4.2-1. Dos instancias de *VistaIcono* sobre distintos modelos.

Supóngase que otro elemento del modelo es la clase *Responsable*. La vista de un responsable se pretende que sea tal y como aparece a la izquierda de la imagen anterior: un icono y debajo el nombre del responsable. Por tanto se trata de reutilizar la clase VistaIcono pero aplicándola a otro objeto del modelo.

Como primer problema se plantea que hay que modificar la clase *VistaIcono* añadiendo la implementación del interfaz *VistaResponsable* para que pueda suscribirse a los eventos de un responsable (por ejemplo el cambio de nombre). Eso ya implica disponer del código fuente.

Sin embargo queda otro problema ¿Qué parámetro se pone en el constructor de la clase *VistaIcono*?

- Si se mantiene la clase Actividad habrá que hacer que la clase Responsable derive de ella de tal manera que pueda ser utilizada por la vista sin necesidad de cambios. Sin embargo no se puede degenerar el modelo por un simple aspecto de representación en pantalla. Un Responsable *no es* una *Actividad*. No todos los métodos de la clase

Actividad se aplican a un Responsable ni se quiere que un responsable pueda ser pasado como parámetro allá donde se requiera una Actividad. Soluciones de diseño como ésta desembocarían rápidamente en un modelo inconsistente guiado más por la representación en pantalla que por los criterios del Diseño Orientado a Objetos[8].

- Otra solución sería crear un interfaz que sirva para aislar a la vista de ambas clases. La vista trabajaría con objetos que implementaran el interfaz *ObjetoConNombre* el cual simplemente tendrá un método para obtener el nombre del objeto. Si se hace que tanto la clase *Responsable* como *Actividad* implementen éste interfaz la vista se podría combinar con ambos objetos.

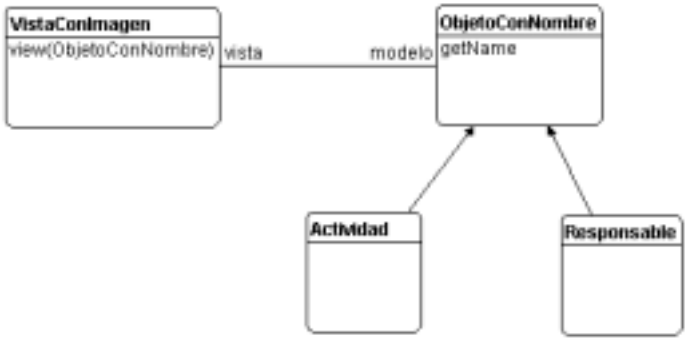

Imagen 4.2-2. La vista trata a Actividad y Responsable a través del interfaz ObjetoConNombre

Sin embargo esto tampoco es una solución. Las clases del modelo han tenido que modificarse para implementar el interfaz ObjetoConNombre. Eso indicaría que ante nuevas vistas posiblemente se avecinen nuevos cambios al modelo (si a actividad quiere ser mostrada en otra vista habrá que modificarla para que implemente el interfaz que la nueva vista requiere).

En general, ¿deben las clases del modelo modificarse ante los cambios en sus vistas? Esto es del todo contrario a la arquitectura MVC. Ésta propone la separación entre modelo y vista porque cuando la funcionalidad que realiza la vista se realiza dentro del propio modelo (por ejemplo el dibujarse) ante cualquier cambio en el interfaz de usuario se provoca un cambio en el modelo. Al sacar esa funcionalidad a las vistas el modelo no debe conocer ni el número ni las características de éstas y se independiza del interfaz de usuario. Pero no se está usando MVC porque el modelo y las vistas *sean clases distintas*; se aplica MVC cuando los *cambios en las vistas* efectivamente *no afectan a las clases del modelo*. Y como caso concreto el hecho de que se quiera reutilizar una clase vista no debe afectar al modelo. Es un detalle de implementación pero que, si se utilizan interfaces, se está permitiendo que afecte al diseño.

- Una tercera solución sería utilizar adaptadores de tal manera que no se modifiquen las clases del modelo para poder conectarse a la vista. Se crea una clase adaptadora que implemente el interfaz *ObjetoConNombre* por cada clase del modelo.

[8] Otra variante de este problema es hacer derivar de una clase común a clases jerárquicamente independientes por el hecho de querer guardarlos en el mismo contenedor. Esto solo consigue polucionar el diseño de clases irrelevantes.

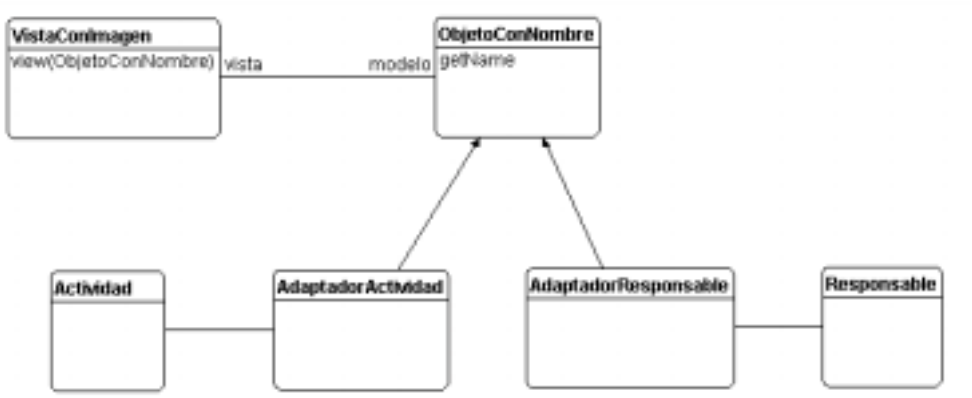

Imagen 4.2-3. Conexión de la vista sin modificar las clases del modelo.

Sin embargo nótese que lo que inicialmente eran tres clases (Actividad, Responsable y VistaIcono) ya se han duplicado pasando a ser seis (se añaden *ObjetoConNombre*, *AdaptadorActividad* y *AdaptadorResponsable*). Esto recarga excesivamente el diseño con clases irrelevantes surgidas por aspectos menores de representación en pantalla.

En definitiva en este trabajo se busca una forma de conectar las vistas y el modelo de tal forma que:

- No requiera el cambio del modelo ante cambios en las vistas (la reutilización de vistas como caso concreto).
- No requiera clases adaptadores como mecanismo general de conexión entre vistas y modelos.

4.2.2.3 Situación de las Validaciones

Otro inconveniente en la arquitectura Modelo-Vista-Controlador es que las reglas de negocio no tienen un sitio claramente establecido. Hay dos lugares candidatos a albergar las reglas:

- Los controladores
- El propio modelo

4.2.2.3.1 Validaciones en los Controladores

Si las validaciones se incluyen dentro de los *controladores* serán éstos los responsables de garantizar que los datos sean válidos antes de añadirlos al modelo. Pero entonces habrá que repetir las mismas comprobaciones en todos los demás controladores.

Supóngase que una aplicación tiene dos vistas de los clientes. Una de ellas es la clásica ficha donde se muestran todos los datos del cliente (a la derecha en la figura). Otra es una lista donde en cada línea se muestra los datos de un cliente (a la izquierda). Si desde ambas ventanas se pueden modificar los datos de los clientes habrá que repetir exactamente las mismas

validaciones en ambas clases, ya que no tiene sentido que solo una de ellas compruebe, por ejemplo, el DNI y la otra no.

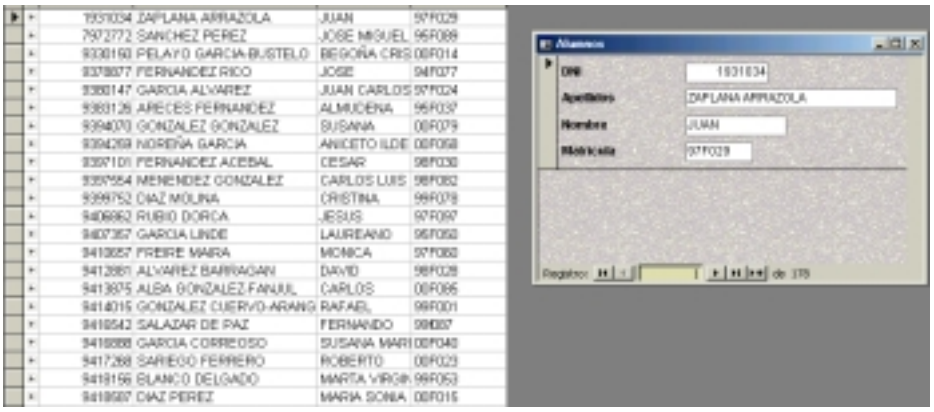

Imagen 4.2-4. Dos vistas de los mismos datos.

Nótese que dado que no es el modelo el que valida los datos sino aquel que los añada, también cualquier operación que añada datos al modelo tendrá que repetir el mismo código de validación. Y cada vez que se añada una nueva regla habrá que modificar todas las operaciones. Si alguna se olvida se estará creando un agujero desde el cual se podrá formar un modelo inválido.

4.2.2.3.2 Validaciones en el Modelo

La otra alternativa que queda es que las validaciones se codifiquen en el modelo. Sin embargo esta solución tampoco es satisfactoria. Cada clase del modelo tiene que conocer a las mínimas clases posibles; debe tener una visión local del modelo. Por tanto no puede realizar validaciones del sistema, es decir, solo puede decir si sus datos son correctos pero no si forman una combinación adecuada con los de otro objeto. Un modelo puede estar en un estado inválido aunque todos sus objetos estén en un estado válido individualmente.

Podría ponerse entonces un objeto *modificador* del modelo que centralizara las modificaciones del modelo y aplicara todas las reglas. Sin embargo este objeto tendría que conocer toda la estructura del modelo y todas sus reglas, por lo que sería afectado por todos los cambios del modelo. En capítulos posteriores se presentará la arquitectura RDM la cual muestra el problema con un ejemplo y cómo lo evita de forma natural.

4.2.2.4 No hay Mecanismos de Recuperación

La arquitectura Modelo-Vista-Controlador no contempla las situaciones de error en el modelo. No se define qué ocurre si una de las vistas produce una excepción y cómo reacciona el modelo.

La forma de tratar los errores y la forma adecuada de dejar el modelo en un estado inválido se deja en manos de la codificación, cuando debería ser una decisión de la arquitectura en global.

4.2.2.5 No hay Direct Mapping

Utilizando un diseño basado en ésta arquitectura la mayor parte de los elementos del dominio no son contempladas. Un modelo no solo comprende a un objeto sino al conjunto que forman los datos y sus relaciones junto con las operaciones, reglas, etc.

A la hora de pasar el diseño a código se presentan los problemas planteados en ésta tesis respecto a la hora de implementar las relaciones y la gestión de su integridad que esta arquitectura no plantea.

4.3 Modelos de Objetos Adaptativos

Muchos sistemas de información orientados a objetos comparten un estilo de arquitectura que enfatiza la flexibilidad y adaptabilidad en tiempo de ejecución. Las reglas de negocio se almacenan de manera externa al programa en una base de datos o en ficheros XML [W3C98] en vez de en el código. El modelo de objetos que al usuario le afecta es parte de la base de datos y el modelo de objetos del código es un intérprete de modelos de objetos de usuario. A estos sistemas se les denomina "Modelos de Objetos Adaptativos" (Adaptive Object-Models) ya que el modelo de objetos del usuario es interpretado en tiempo de ejecución y puede ser cambiado con efectos inmediatos en el sistema que lo interpreta.

La característica de los modelos de objetos adaptativos es que tienen una definición del modelo del dominio y reglas para su integridad y pueden ser configurados por expertos del dominio de manera externa a la ejecución del programa.

4.3.1 Introducción

Los usuarios a menudo quieren cambiar las reglas de su negocio sin necesidad de rescribir el código. Los clientes requieren que los sistemas se adapten más fácilmente a las cambiantes necesidades de su negocio [Rouvellou00].

Muchos sistemas de información de hoy en día requieren ser dinámicos y configurables de tal manera que puedan adaptarse. Esto usualmente se hace moviendo ciertos aspectos del sistema, tales como las reglas de negocio, a una base de datos de tal manera que puedan ser fácilmente cambiadas. El modelo resultante permite al sistema adaptarse rápidamente a las necesidades de cambio simplemente cambiando valores en la base de datos en vez de en el código.

También promueve el desarrollo de herramientas que permitan a los directivos y administradores introducir nuevos productos sin programar y cambiar sus modelos de negocio en tiempo de ejecución. Esto puede reducir el tiempo de salida al mercado de nuevas ideas de meses a semanas o días. Por tanto la posibilidad de adaptar el sistema se sitúa en las manos de aquellos que tienen el conocimiento del negocio para hacerlo de manera efectiva.

Las arquitecturas que pueden dinámicamente adaptarse en tiempo de ejecución a los requisitos del usuario en ocasiones son denominadas "arquitecturas reflectivas" o "meta-arquitecturas". Los "Modelos de Objetos Adaptables" son un tipo de estas arquitecturas que también se las conoce por otros muchos nombres. Se las ha llamado "Type Instance Pattern" en [Gamma95]. También, dado que generalmente se utilizan para modelar sistemas que gestionan productos, se la denomina "User Defined Product Arquitecture" [Johnson98]. También es conocida como "Active Object Models" [Foote98] y como "Dinamic Object Models" [Riehle00].

Un Modelo de Objetos Adaptativo es un sistema que representa clases, atributos y relaciones como metadatos. El sistema se modela basándose en instancias en vez de en clases. Los usuarios cambian los metadatos para reflejar los cambios del dominio. Estos cambios modifican el comportamiento del sistema. En otras palabras, almacena el modelo de objetos en la *base de datos y lo interpreta*. Por tanto el modelo de objetos es activo, cuando éste se cambia el sistema cambia inmediatamente. Esta arquitectura ha sido usada para implementar pólizas de seguros [Johnson98], para gestionar la facturación de una empresa telefónica [Martine98]; para modelar modelos de procesos [Manolescu00] y [Tilman99]; etc.

4.3.2 Arquitectura de los Modelos de Objetos Adaptativos

Los modelos de objetos adaptativos proporcionan una alternativa a los diseños orientados a objeto tradicionales. El diseño tradicional asigna clases a los diferentes tipos de entidades de negocio y las asocia atributos y métodos. Las clases modelan el negocio así que un cambio en el negocio ocasiona un cambio en el código que produce una nueva versión de la aplicación.

Un modelo de objetos adaptativo no modelas las entidades como clases. En vez de eso modelan las descripciones de las clases *que son interpretadas* en tiempo de ejecución. Por tanto cuando un cambio del negocio es requerido estas descripciones son cambiadas e inmediatamente reflejadas en la aplicación en ejecución.

Esta arquitectura normalmente se compone de varios patrones de diseño. El patrón Type Object [Johnson98b] separa las entidades de su tipo. Las entidades tienen atributos que son implementados con el patrón "Property" y de nuevo es vuelto a aplicar el patrón Type Object para separa a éstos de su tipo. Finalmente el patrón Strategy se ocupa del modelado de las operaciones.

4.3.2.1 Patrón Type Object

Los lenguajes orientados a objetos modelan los programas como un conjunto de clases. Una clase define la estructura y el comportamiento de sus instancias. Generalmente se usa una clase para cada tipo de entidad así que introducir un nuevo tipo de objeto supone programar una nueva clase. Sin embargo los desarrolladores de grandes sistemas usualmente se enfrentan al problema de tener clases de las cuales deben crear un conjunto indeterminado de subclases [Johnson98b]. Cada subclase es una abstracción de un elemento del dominio variable. Las diferencias entre las subclases son pequeñas y pueden ser parametrizadas estableciendo valores o objetos que representen algoritmos. Type Object hace de las subclases desconocidas simples instancias de una clase genérica. Nuevas clases pueden ser creadas dinámicamente en tiempo de ejecución instanciando dicha clase genérica. Los objetos creados de la jerarquía tradicional son creados pero haciendo explícita la relación entre ellos y su tipo.

4.3.2.2 Patrón Property

Los atributos de un objeto usualmente se implementan en sus variables de instancia. Dichas variables de definen mediante su clase. Si objetos de distintos tipos son todos creados a partir de la misma clase genérica se presenta el problema de cómo pueden tener distintos atributos. La solución es aplicar el patrón Property el cual modela cada uno de los atributos de una entidad como un objeto independiente. Por tanto todas las entidades tienen como variable de instancia lo mismo: un contenedor de propiedades. El objeto propiedad contienen tres elementos: el nombre del atributos, su valor y su tipo.

La mayor parte de las arquitecturas basadas en modelos de objetos adaptativos utilizan en patrón Type Object y el Property. El Type Object divide el sistema en entidades y tipos de entidades. Las entidades tienen atributos y cada atributo tiene una Entity Type resultado de volver a aplicar el mismo patrón a las propiedades.

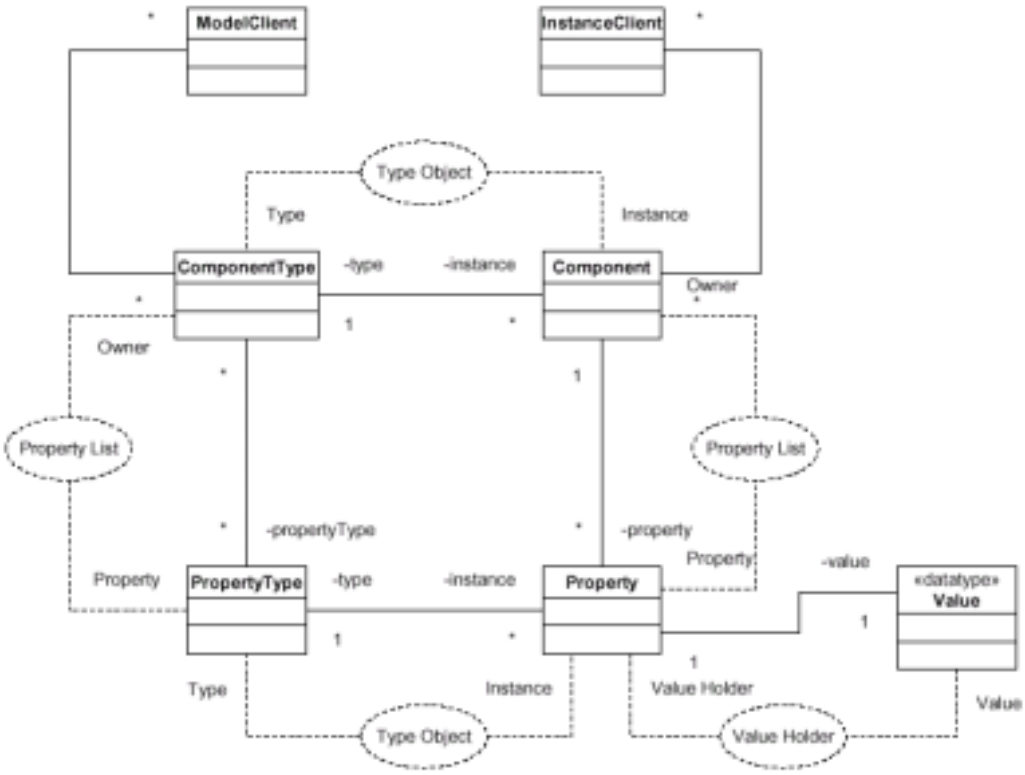

Imagen 4.3-1. Modelo de clases de AOM y los patrones por los que se generan.

4.3.2.3 Patrón Strategy

Una estrategia es un objeto que encapsula a un algoritmo [Gamma94]. El patrón estrategia define un interfaz estándar para una familia de algoritmos para que los clientes puedan trabajar con cualquiera de ellos de la misma manera. Si el comportamiento de un objeto se define en función de una o más estrategias entonces su comportamiento puede ser modificado dinámicamente.

Cada aplicación del patrón Strategy produce una interfaz diferente y por lo tanto una jerarquía de clases distinta. Las estrategias pueden evolucionar para llegar a ser complicadas reglas de negocio que son construidas o interpretadas en tiempo de ejecución. Estas pueden ser reglas primitivas o combinación de reglas mediante la aplicación del patrón Composite [Gamma94].

El modelado de reglas complicadas junto con los interfaces de usuario dinámico usualmente es la parte más difícil de los modelos de objetos adaptativos y es la razón por la que *no son implementables en un framework* genérico [Yoder01].

4.3.2.4 Relaciones

Los atributos son propiedades que usualmente se refieren a tipos primitivos. Estas asociaciones son usualmente unidireccionales. Las relaciones son propiedades que se refieren a otras entidades y son generalmente bidireccionales. Esto generalmente se modela mediante dos subclases de propiedades, una para los atributos y otra para las relaciones.

Estas dos subclases se suelen modelar de tres formas.

- La primera es volviendo a aplicar otra vez el patrón Property para las relaciones al igual que se aplicó para los atributos.

- La segunda manera es utilizar la misma construcción que se hizo para los atributos pero poniendo como clase padre a la clase Property y subdividiendo mediante herencia el Property y en Association. Una Association conocería su cardinalidad.

- La tercera forma sería discriminando por el valor de la propiedad. Si dicho valor es otra entidad se supone que es una relación.

4.3.2.5 Interfaces de Usuario para la Definición de Tipos

Una de las razones principales para diseñar modelos de objetos adaptativos es permitir a los usuarios y expertos de dominio cambiar el comportamiento del sistema definiendo nuevas entidades, relaciones y reglas. Algunas veces el objetivo es permitir a los usuarios extender el sistema sin programadores. Pero incluso cuando solo los desarrolladores definan nuevas entidades y reglas es habitual construir un interfaz de usuario especializado para definirlos.

Los tipos son guardados en una base de datos centralizada. Esto significa que cuando alguien define nuevos tipos las aplicaciones pueden usarlos sin necesidad de ser recompiladas. A menudo las aplicación son capaces de usar los nuevos tipos inmediatamente mientras otras veces guardan en una caché la información de tipos y deben ser actualizadas antes de que usen los nuevos tipos.

4.3.3 Crítica a los Modelos de Objetos Adaptativos

4.3.3.1 Lenguaje de Reglas

El usuario debe poder introducir de alguna manera las reglas de negocio en el sistema. ¿En qué lenguaje lo hace? Este es uno de los problemas principales de los Modelos de Objetos Adaptativos.

> "MetaData is just saying that if something is going to vary in a predictable way, store the description of the variation in a database so that it is easy to change. In other words, if something is going to change a lot, make it easy to change. The problem is that it can be hard to figure out what changes, and even if you know what changes then it can be hard to figure out how to describe the change in your database. Code is powerful, and it can be hard to make your data as powerful as your code without making it as complicated as your code.". [Johnson98]

Es decir ¿Hará falta un nuevo lenguaje de programación para describirlo? ¿Qué ventaja tiene sobre el lenguaje de programación al que sustituye el metamodelo?

Podría iniciarse con un lenguaje sencillo de script pero ¿cómo acabaría dicho lenguaje?

> "The patterns in our emerging pattern language begin to chronicle how domain specific languages emerge as programs evolve. A program may begin simply, performing but a single task. Later, programmers may broaden its utility by adding options and parameters. When more configuration information is needed, separate configuration files may emerge. As these become more complex, entries in these files may be connected to entities in the program using properties, dynamic variables, and dialogs. Simple values may not suffice. Once properties and dynamic values are present, simple parsers and expression analyzers often are added to the system. This, in turn creates a temptation to add control structures, assignment, and looping facilities to the functional vocabulary provided by the expression analyzer. These can flower into full-blown scripting facilities". [Foote98].

Por tanto se plantea un problema entre:

- Facilidad de uso del lenguaje.
- Limitaciones sobre las reglas a modelar.

Si el lenguaje es demasiado simple harán falta programadores para introducir *el resto* de las reglas. Si el lenguaje es potente pero complejo harán falta programadores *para todas* las reglas.

En esta tesis no se busca un nuevo lenguaje de reglas ni se pretende que las edite el usuario. Parte con una orientación clara hacia los problemas de los desarrolladores y éstos son capaces de utilizar lenguajes de programación profesionales en los cuales modelar las reglas por lo que no es práctico plantear un nuevo lenguaje.

4.3.3.2 Desarrollo de Herramientas Específicas

Una vez definido la forma de manipular el modelo ¿cómo se introduce en el sistema?

> "Power never comes without a price. When you confer the power to program on users, you give them the power to make mistakes. Just as certainly as ants follow picnics, where programs go, bugs shall surely follow. It is for this reason that the construction of ACTIVE-OBJECT MODELS should not be undertaken without a solid infrastructure of editing, programming, and support tools." [Foote98].

> "New development tools. Programmers can no longer rely on their familiar development tools, such as browsers to edit and view types of components. Other traditional tools break down because they are not effective anymore. Debuggers and inspectors, for instance, still work, but they are much harder to use: type objects appear as any other field in an inspector, whereas we should be able to view components as instances of their component type. You need to provide new tools that replace or enhance the existing tools." [Riehle00].

> "As stated before, you are in a sense building a domain-specific language. And although it is often easier for users to understand than a general-purpose language, you are still developing a language. This means that you inherit all of the problems associated with developing any language such as needing debuggers, version control, and documentation tools. Other problems are those involved with training. There are ways around these problems but they are problems non-the less and should be faced." [Yoder01].

En definitiva para un proyecto adaptativo hace falta crearse un entorno de desarrollo completo, lo cual es demasiado costoso y no cumple el requisito de poder reutilizar las herramientas actuales para poder permitir desarrollos rápidos.

4.3.3.3 Implementación de Vistas

La AOM se caracteriza por permitir a los expertos de dominio cambiar las reglas y el modelo. Sin embargo ese modelo debe de ser representado en distintas vistas ¿tiene que haber también un lenguaje de construcción de vistas o herramientas? ¿Con qué complejidad?

Las aplicaciones adaptativas generalmente generan las vistas basadas en formularios a partir de la información de los metadatos [Anderson98]. Sin embargo esto supone generalmente:

- Un problema de usabilidad [Nielsen94]. Los generadores automáticos se limitan a asignar componentes visuales a cada una de las propiedades en función de su tipo. Sin embargo [Cooper95] afirma que no basta usar componentes para que un interfaz sea usable.

- Cuando el interfaz de usuario no está basado en formularios es necesaria la intervención de programadores igualmente.

Por tanto aquí no se busca que las interfaces de usuario se generen automáticamente (todavía no es posible) ni que sea el usuario el que lo haga. Por tanto se prefiere que sigan siendo los desarrolladores los que hagan las interfaces de usuario pero facilitándoles la conexión con los objetos del modelo e independizándolos lo máximo posible ante los cambios de éstos.

4.3.3.4 No tiene una Implementación Reutilizable

El diseño de modelos de objetos adaptativos incluye tres actividades principales:

- Definir las entidades y las reglas
- Desarrollar un motor de instanciación y manipulación de las entidades dirigido por las reglas
- Desarrollo de herramientas que permitan las dos tareas anteriores

Pero estas son actividades. Cada aplicación se tiene que construir su propio framework con sus propias clases pero no hay framework genérico para construirlos [Yoder01c].

4.3.3.5 Aplicable en Sistemas muy Variables

Por todo lo visto hasta el momento se llega a la conclusión de que los Modelos de Objetos Adaptativos solo compensan en grandes entornos de gran variabilidad debido a la gran inversión de recursos que supone.

> "Active Object-Models are certainly harder to build than a conventional systems. They usually evolve out of frameworks. [...] If your system is only going to change a couple of times over its lifetime, then the effort entailed in constructing a framework may not be worth the cost and bother.". [Foote98]
>
> "Adaptive Object-Models are not the solution to every problem. They require sustained commitment and investment, and their flexibility and power can make them harder to understand and maintain. They should only be undertaken when either the products or rules associated with products are rapidly changing, or, when users demand to have highly configurable systems." [Yoder2001].
>
> "Because of potential complexity and performance issues, use of dynamic object models should be limited to areas of the system wich are likely to experience considerable change". [Yoder98].

Sin embargo en esta tesis no se busca una técnica compleja que solo sea adecuada para modelos de objetos muy dinámicos. Se pretende una arquitectura para modelar, precisamente, los modelos de objetos *más habituales*.

4.3.4 Adecuación a los Objetivos

Lo que en esta tesis se pretende es una arquitectura:

- Centrada en el desarrollador y no en el usuario.
- Que no esté centrada en el uso de bases de datos.
- Que no requiera un intérprete ni un entorno de desarrollo específico.

Aunque los Modelos de Objetos Adaptativos es una tecnología prometedora todavía necesita un mayor desarrollo para que pueda ser aplicado sin tantos inconvenientes. Hasta que se pueda conseguir que el usuario pueda definir el modelo y las reglas de forma sencilla hay que recorrer todavía un gran camino. Y no solo en el aspecto tecnológico sino también en el humano[9].

Pero aunque no se sabe si se llegará a conseguir que los usuarios puedan adaptar su software lo que sí se puede afirmar es que es necesario que los programadores *puedan hacerlo ya*. Por eso la arquitectura RDM aquí propuesta se centra en los desarrolladores como *paso intermedio* antes de llegar a los modelos adaptativos.

Los modelos adaptativos son adecuados para proyectos que vayan a sufrir muchos cambios en fase de explotación. Sin embargo *todo* proyecto sufre cambios en fase de desarrollo. Aquí se busca una solución para esas fases. Por tanto RDM supondrá una solución menos ambiciosa que los AOM pero de aplicación inmediata con cualquier lenguaje y herramienta.

4.4 Aspect Oriented Programming

Existen situaciones en las que los lenguajes orientados a objetos no permiten modelar de forma suficientemente clara las decisiones de diseño tomadas previamente a la implementación. El sistema final se codifica entremezclando el código propio de la especificación funcional del diseño, con llamadas a rutinas de diversas librerías encargadas de obtener una funcionalidad adicional (por ejemplo, distribución, persistencia o multitarea). El resultado es un código fuente excesivamente difícil de desarrollar, entender, y por lo tanto mantener [Kiczales97].

Cuando desarrollamos una aplicación, podemos hacerlo de dos forma bien distintas: la primera, siguiendo una estructura computacional clara y organizada; la segunda, primando la eficiencia de la aplicación, generalmente con un código fuente menos legible. La programación orientada al aspecto permite separar la funcionalidad de la aplicación de la gestión de memoria y eficiencia de los algoritmos utilizados, y compone el sistema final a partir del conjunto de aspectos identificados por el programador. En la programación orientada a aspectos se definen dos términos distintos [Kiczales97]:

- Un componente es aquel módulo software que puede ser encapsulado en un procedimiento (un objeto, método, procedimiento o API). Los componentes serán unidades funcionales en las que se descompone el sistema.

- Un aspecto es aquel módulo software que no puede ser encapsulado en un procedimiento. No son unidades funcionales en las que se pueda dividir un sistema, sino propiedades que afectan la ejecución o semántica de los componentes. Ejemplos de aspectos son la gestión de la memoria o la sincronización de hilos.

Una aplicación desarrollada mediante programación orientada al aspecto está compuesta por la programación de sus componentes funcionales en un determinado lenguaje, y por la

[9] El cliente no siempre está dispuesto a modificar el software y le es más cómodo delegarlo en programadores. Esto es una razón más para intentar hacer más eficaz el trabajo de éstos.

programación de un conjunto de aspectos que especifican características adicionales del sistema. El "tejedor de aspectos" (aspect weaver) acepta la programación de aspectos y componentes, y genera la aplicación final en un determinado lenguaje de programación.

La generación de la aplicación final se produce en tres fases [Ortín01]:

1. En la primera fase, el tejedor construye un grafo del flujo de ejecución del programa de componentes.

2. El segundo paso consiste en ejecutar la programación de cada uno de los aspectos. Éstos actúan sobre el grafo que define la funcionalidad del sistema (punto anterior) realizando las modificaciones o añadiduras oportunas.

3. La última fase consiste en recorrer el grafo modificado y generar el código final de la aplicación; ésta cumple con la funcionalidad descrita por sus componentes y ofrece los aspectos definidos por el programador.

Se han implementado distintos sistemas de programación orientada al aspecto, siendo uno de los más avanzados el denominado AspectJ [Kiczales2001]; una extensión de la plataforma de Java. Una aplicación en AspectJ consta de:

- Puntos de unión (join points): Describen puntos de ejecución de una aplicación. AspectJ proporciona un elevado grupo de puntos de unión, siendo un ejemplo de éstos la invocación de métodos.

- Designación de puntos de ruptura (pointcut designators): Identifican colecciones de puntos de unión en el flujo de ejecución de un programa, para posteriormente poder modificar su semántica.

- Consejo (advice): Acciones que se ejecutan previamente, en el momento, o posteriormente a que se alcance la ejecución de una designación de puntos de ruptura.

- Aspecto (aspect): Módulo de implementación, definido como una clase, que agrupa acciones en puntos de unión.

Utilizando este entorno de programación se han desarrollado sistemas de trazabilidad de software, auditoría de invocación a métodos (logging), adición de precondiciones y poscondiciones, así como utilización de programación por contratos [Meyer97] en fase de desarrollo y su posterior eliminación sin necesidad de modificar el código fuente. También se ha conseguido desarrollar aspectos reutilizables como por ejemplo determinadas políticas de sincronización de hilos [Kiczales2001].

4.4.1 Conclusiones sobre la Programación Basada en Aspectos

La utilidad buscada de la programación orientada a aspectos debe ser la siguiente:

- Se teclea una funcionalidad básica.

- Si luego se necesita añadir una funcionalidad adicional (concurrencia, por ejemplo) que se permita añadirlo de manera externa sin necesidad de afectar al código ya construido; de manera transparente para éste el cual no debe ser consciente de esta ampliación.

- De esta manera se le van añadiendo distintos aspectos sin que la funcionalidad original se vea afectada por estos temas *circunstanciales*.

Una aplicación basada en aspectos no se trata simplemente de que se pueda separar en varias clases. Se trata de que cada una de esas clases sea independiente. Si la clase interceptada se ha escrito de manera que tenga que prever los distintos aspectos de alguna manera no se ha conseguido nada. Por tanto para conseguir en la práctica es espíritu de la programación basada en aspectos se tienen que dar dos circunstancias obligatorias:

- Cualquier clase debe poder ser interceptada sin ninguna preparación por parte de ella.

- Pero también que la clase interceptadora no sepa nada de la implementación de la interceptada. Un ejemplo que se suele poner es añadir eventos a un objeto cada vez que cambie su estado. Por tanto hay que saber qué métodos cambian el modelo y así interceptarlos. Pero supóngase que un método realiza una serie de comprobaciones para decidir si cambia el estado. Entonces el interceptador deberá incluir dichas comprobaciones para lanzar los eventos solo en ese caso: por tanto deberá conocer dichas comprobaciones – la implementación del método. No solo estarán duplicadas en dos sitios sino que además son dos sitios a mantener en caso de cambiar dichas condiciones. Por tanto el único que puede avisar de los cambios en su estado es el propio modelo. Si este no lo hace la solución no es añadir aspectos; la solución es implementarlo correctamente ya que el problema en el mal diseño interno de esa clase que no se puede corregir con condimentos externos.

En caso de que no se cumpla la primera condición puede hacerse con patrones; por ejemplo que el objeto base se haya implementado delegando en un patrón Strategy o State (si hay que preverlos es lo mismo prever un aspecto que prever una estrategia). Se consigue lo mismo pero sin necesidad de nuevas herramientas de programación, por lo que es aplicable en cualquier lenguaje orientado a objetos.

En caso de que no se cumpla la segunda condición se estarán requiriendo los códigos fuentes, por lo cual no hay ninguna ventaja práctica frente a modificar éstos. Hasta ahora los ejemplos mostrados en distintos artículos fallaban de una cosa o de otra [Javaworld2002].

Si al hacer la funcionalidad principal ya se saben los aspectos necesarios no se consigue mucho. De lo que se trata es de que luego se puedan añadir los que inicialmente no se han previsto sin modificar la funcionalidad inicial. Pero los que ya se han previsto, dado su acoplamiento a la funcionalidad inicial, no hay ventaja frente a encapsularlo todo junto con dicha funcionalidad. Se podría argumentar que el separar las cosas facilita su aprendizaje. Pero esta afirmación ya es más subjetiva ya que sería discutible que el hecho de que una funcionalidad esté repartida por varios sitios es más fácil de comprender que si está todo junto y con una secuenciación explícita.

4.4.2 Adecuación a los Objetivos Buscados

La Programación Basada en Aspectos, en cuanto al problema aquí tratado, no es exactamente la solución buscada. Se puede plantear que los aspectos son un mecanismo de diseño, una forma de distribuir las responsabilidades en el código, pero que en el dominio no existen como tal; son una forma de modelar las operaciones en el ordenador. La esencia de AOP es la operación y de ésta extrae aspectos de la misma. La esencia de RDM es el dominio y de éste extrae, entre otros elementos, las operaciones. Por tanto podrían considerarse técnicas complementarias a partir del *elemento* operación.

En cualquier caso la arquitectura RDM podría verse como el tratamiento especializado para aquellos *aspectos* del dominio más habituales ofreciendo un conjunto de métodos para modelarlos de una forma más eficaz que con las tecnologías orientadas a objetos tradicionales. Pero para ello no se basa en interceptar la ejecución de la aplicación sino en organizarla a base de patrones y sin necesidad de lenguajes adicionales ni preprocesadores.

4.5 Separación Multidimensional de Incumbencias

La separación de incumbencias [Parnas72] es parte fundamental de la ingeniería del software y todo desarrollador lo realiza de una manera o de otra. En su forma más general se refiere a la habilidad de identificar, encapsular y manipular solo aquellas partes del software que son relevantes para un concepto particular, objetivo o propósito. Las incumbencias son la razón principal para organizar y descomponer el software en partes manejables y comprensibles. Muchas clases diferentes de incumbencias pueden ser relevantes para distintos desarrolladores en distintos roles o bien en diferentes etapas del ciclo de desarrollo. Por ejemplo el tipo fundamental de incumbencia en la programación orientada a objetos es la clase; cada incumbencia de éste tipo es un tipo de datos definido y encapsulado en una case. Operaciones [Turner98] como el imprimir, persistencia y representación visual habitualmente son también incumbencias lo mismo que lo son aspectos [Kiczales97] como la concurrencia y la distribución. Se suele denominar a un tipo de incumbencia como la clase o la operación como una dimensión de una incumbencia. La separación de incumbencias implica la descomposición del software de acuerdo a una o más dimensiones de incumbencias.

Lograr una clara separación de incumbencias ayuda a:

- Reducir la complejidad del software y mejorar su comprensión
- Mejorar el seguimiento por los distintos artefactos del ciclo de desarrollo
- Limitar el impacto del cambio.
- Facilitar la reutilización
- Simplificar la integración de componentes.

Estos objetivos, aunque importantes, no se han conseguido plenamente en la práctica [IBM00e]. Esto es principalmente debido a que el conjunto de incumbencias relevantes varía a lo largo del tiempo y se ve afectado por el contexto. Diferentes fases del desarrollo, desarrolladores y roles a menudo afectan a incumbencias de tipos radicalmente distintos y por tanto a múltiples dimensiones. La separación a lo largo de una dimensión puede producir unos objetivos a la vez que impide otros. Por tanto cualquier criterio de descomposición e integración será apropiado para algunos contextos y requisitos pero no para todos. Por ejemplo la descomposición basada en clases de las tecnologías orientadas a objetos facilita en gran medida la evolución de los detalles de las estructuras de datos ya que son encapsuladas en una clase. Sin embargo impide la edición o evolución ya que están repartidas por varias clases. Además varias dimensiones pueden ser relevantes simultáneamente y pueden sobreponerse e interactuar lo mismo que lo hacen las clases y las operaciones. Por tanto la modularización de distintas dimensiones de incumbencias es necesaria para distintos objetivos: unas veces mediante clases, otras mediante operaciones, otras mediante aspectos, otras mediante roles, etc.

Se usa el término separación multi-dimensional de incumbencias a aquella separación de incumbencias que permite [Ortín01]:

- Dimensiones múltiples de incumbencias
- Separación a lo largo de estas dimensiones simultáneamente. No debe haber una dimensión dominante sobre las demás.
- La habilidad de manejar nuevos incumbencias y nuevas dimensiones de incumbencias dinámicamente a medida que surjan del ciclo de desarrollo.
- Superposición e interacción de incumbencias. Es más sencillo pensar en varios incumbencias independientes u ortogonales pero raramente lo son en la práctica.

La separación multi-dimensional de incumbencias representa un conjunto de objetivos muy ambiciosos. El soporte para la separación multi-dimensional de incumbencias abre la puerta para la remodularización bajo demanda, permitiendo al desarrollador elegir en cada momento la mejor forma de modularizar para la actividad de desarrollo en curso.

4.5.1 Adecuación a los Objetivos Buscados

La separación multi-dimensional de incumbencias es un área de investigación de gran actualidad y que actualmente ya tiene implementaciones utilizables para el desarrollo de aplicaciones [IBM00f].

Sin embargo, en cuanto a esta tesis se refiere, no se centra en el modelado de los elementos del dominio o, mejor dicho, al igual que ocurría en la programación basada en aspectos se centra fundamentalmente en la organización *de las operaciones*. Pero no afecta al resto de los elementos aquí tratados (como por ejemplo las relaciones). Por tanto esta tecnología podría combinarse con la arquitectura RDM sin que supongan opciones excluyentes.

4.6 Adaptative Programming

La programación adaptable es una extensión de la programación orientada a objetos, en la que se flexibiliza las relaciones entre la computación y los datos; el software se puede adaptar para manejar cambios en los requerimientos del sistema. La programación adaptable permite expresar la intención general de un programa sin necesidad de especificar todos los detalles de las estructuras de los objetos [Lieberherr96].

El software adaptable toma el paradigma de la orientación a objetos y lo amplía para que éste soporte un desarrollo evolutivo de aplicaciones. La naturaleza progresiva de la mayor parte de los sistemas informáticos, hace que éstos sufran un número elevado de cambios a lo largo de su ciclo de vida. La adaptabilidad de un sistema guía la creación de éste a una técnica que minimice el impacto producido por los cambios. Software adaptable es aquél que se amolda automáticamente a los cambios contextuales (por ejemplo, la implementación de un método, la estructura de una clase, la sincronización de procesos o la migración de objetos).

Un caso práctico de programación adaptable es el método Demeter [Lieberherr96]. En este método se identifican inicialmente clases y su comportamiento básico sin necesidad de definir su estructura. Las relaciones entre éstas y sus restricciones se establecen mediante patrones de propagación (propagation patterns). En este nivel de programación se especifica la intención general del programa sin llegar a realizar una especificación formal.

Cada patrón de propagación de divide en las siguientes partes:

- Signatura de la operación o prototipo de la computación a implementar.
- Especificación de un camino; indicando la dirección en el grafo de relaciones entre clases que va a seguir el cálculo de la operación.
- Fragmento de código; especificando algún tipo de restricción, o la realización de la operación desde un elevado nivel de abstracción.

Una vez especificado el grafo de clases y los patrones de propagación, tenemos una aplicación de mayor nivel de abstracción que una programada convencionalmente sobre un lenguaje orientado a objetos. Esta aplicación podrá ser formalizada en distintas aplicaciones finales mediante distintas personalizaciones (customizations). En cada personalización se especifica la estructura y comportamiento exacto de cada clase en un lenguaje de programación –en el caso de Demeter, en C++ [Stroustrup98].

Finalmente, una vez especificado el programa adaptable y su personalización, el compilador Demeter genera la aplicación final en C++. La adaptabilidad del sistema final se centra en la modificación de las personalizaciones a lo largo del ciclo de vida de la aplicación; distintas personalizaciones de un mismo programa adaptable dan lugar a diferentes aplicaciones finales, sin variar la semántica del más elevado nivel de abstracción.

4.6.1 Adecuación a los Objetivos

Demeter es una técnica que se combinaría con la arquitectura RDM para reforzar más la independencia entre operaciones y entidades especialmente en la operación de navegación por el modelo.

Sin embargo Demeter exige para su implementación:

- Un lenguaje con introspección.
- O bien un lenguaje externo (como en la versión en C++).

Esto supone que la implementación en lenguajes como Java y C++ sea completamente distinta. En el primero se utiliza el patrón Visitor y el segundo utiliza un generador de código.

Además, aunque permite no tener que cambiar las operaciones al cambiar la estructura del modelo, no se encarga de otros elementos del dominio como:

- Las relaciones y su integridad.
- Las reglas de negocio.
- El tratamiento de la interrupción de operaciones.

Por tanto Demeter podría considerarse como un complemento de esta arquitectura. Precisamente una de las líneas de investigación propuestas en esta tesis es implementar una versión de Demeter que aproveche las características de los modelos construidos con RDM. Se evitarían así los problemas que tiene la versión en Java de Demeter-J con las colecciones sin tipo en cuanto a la expansión incontrolada de rutas [Demeter].

4.7 Generative Programming

La programación generativa [Czarnecki00] ofrece una infraestructura para la transformación de descripciones del sistema en código. Trata con un gran abanico de posibilidades incluyendo Aspect Oriented Programming. Aunque la programación generativa no excluye a la los modelos de objetos adaptativos sin embargo la mayor parte de las técnicas tratan con la generación de código a partir de descripciones. Las descripciones se hacen mediante una estructura de primitivas o elementos y pueden llegar a ser un lenguaje visual de dominio.

El concepto central de la programación generativa es el modelo generativo del dominio. El modelo debe cubrir a una familia de problemas (no solamente una solución o aplicación). Además debe haber alguna representación de alto nivel específica del dominio que permita especificar o definir una instancia concreta de la familia. Finalmente hay alguna configuración que transforma de la descripción del problema al espacio de la solución.

Hay distintas tecnologías de trasformación para un modelo generativo del dominio. La especificación específica del dominio puede ser implementada mediante un nuevo lenguaje textual o gráfico o puede ser implementa directamente en un lenguaje de programación o incluso puede ser llevado a cabo por herramientas especializadas (Wizards) o formularios. La transformación entre la especificación y la solución puede ser obtenida mediante generación de código, reflectividad dinámica o simplemente utilizando el Patrón Factory.

4.7.1 Adaptación a los Objetivos Planteados

En esta tesis se plantea el modelado de los distintos elementos del dominio. La programación generativa, en su amplia definición, podría incluir la arquitectura aquí propuesta como una de las tecnologías posibles *a las cuales trasformar* la especificación del dominio.

Sin embargo en este trabajo se busca una arquitectura no requiera ni de un nuevo lenguaje que controle la transformación ni de herramientas externas.

4.8 Adaptable Systems y Adaptive Systems

Hasta ahora se ha hablado de Modelos de Objetos Adaptativos y Programación Adaptable. Sin embargo, para aumentar la confusión, también existen otras dos definiciones que incluyen la palabra *adaptativo* o similar [Borland98].

- Un *Sistema Adaptable* es aquel en el que el usuario puede ajustar controles para adaptar partes del sistema a un estilo individual de trabajo. Por ejemplo en un procesador de texto el usuario podría seleccionar desactivar ciertas opciones avanzadas de los menús.

- Un *Sistema Adaptativo* es aquel que ajusta sus controles y su representación basándose en las acciones que el usuario realiza mientras trabaja. Por ejemplo el sistema podría tener en cuenta cuantas veces el usuario escoge un determinado comando y ponerlo en una posición de acceso más cómodo.

Éstas dos últimas técnicas no están relacionadas con el objetivo de la tesis y solo se referencian para evitar la confusión que produce la similitud de sus nombres. Esta tesis no contempla ni tiene relación con ninguna de estas técnicas.

4.9 Table-Driven Systems

Los sistemas dirigidos mediante tablas llevan utilizándose desde los primeros tiempos de las bases de datos. Este método se utiliza cuando las diferencias en las reglas de negocio *pueden ser parametrizadas* [Perkins00]. Almacenando los valores de éstos parámetros en una base de datos

el sistema puede consultar dichos valores en tiempo de ejecución y reaccionar así de la forma adecuada.

Esta tecnología quizás sea uno de los primeros intentos de extraer las reglas de negocio del código. Sin embargo solo sirve para reglas simples y que sus parámetros son conocidos y estables, pero no suponen un mecanismo *genérico* de modelado de reglas.

Sección 3. Solución Propuesta

Una vez visto que los objetivos no quedan suficientemente cubiertos con las tecnologías actuales se entrará en la presentación de la tesis. Esta sección se divide en tres capítulos:

- En el capítulo 5, "Arquitectura RDM" se define cómo deberán modelarse los distintos elementos del dominio para que sea posible conseguir los objetivos propuestos. Se justificará los elementos que tiene que haber en la arquitectura y se introducirán los principios que dirigirán su construcción. En este capítulo no se entrará en detalles de implementación sino más bien se establece lo que va a tener que proporcionar ésta (independientemente de cómo lo haga). Este capítulo es el que contiene *la tesis* propiamente dicha.

- En el capítulo 6 se entra en un nivel de detalle mayor. Una vez que en el capítulo anterior se hayan establecido los elementos de la arquitectura de manera individual en ese capítulo se justificará qué "Mecanismos de Comunicación" deben utilizarse entre ellos de manera que se consiga la máxima independencia pero con total flexibilidad a la hora de su combinación.

- Finalmente en el capítulo 7, "El Framework RDM" se describirá una implementación de la arquitectura. Se verá cómo se han implementado los distintos elementos identificados en el capítulo 5 y cómo se comunican de la manera descrita en el capítulo 6. Se muestran las clases del framework mediante un tutorial del mismo donde se muestra su flexibilidad y sencillez de uso.

Una vez introducida la arquitectura y su implementación en la siguiente sección titulada "Demostración: Aplicación de RDM" se mostrará cómo se ha utilizado la arquitectura en un proyecto real y cómo se han evitado los problemas inicialmente expuestos en la tesis.

5 La Arquitectura RDM

En el segundo capítulo se mostró el problema que plantea la tecnología orientada a objetos a la hora de modelar los distintos elementos de un dominio (relaciones, operaciones, etc.). Para evitar este problema en el capítulo 3 se planteó como objetivo encontrar alguna forma de *disminuir el tiempo requerido para la representación del dominio y de sus posteriores cambios durante el desarrollo en las aplicaciones basadas en las tecnologías orientadas a objetos*.

Para cumplir dicho objetivo se defenderá la tesis de que *una arquitectura que esté basada en la representación especializada e independiente de los distintos elementos del dominio proporciona una reducción en los tiempos de desarrollo y favorece la reutilización*.

En este capítulo se darán las guías de cómo deberán modelarse los distintos elementos del dominio siguiendo esta tesis para que así sea posible conseguir los objetivos propuestos. Se justificará los elementos que tienen que conformar la arquitectura y se introducirán los principios que dirigirán su construcción.

En este capítulo no se entrará en detalles de implementación sino más bien se establece lo que tendrá ésta que proporcionar (independientemente de cómo lo haga, lo cual se concretará en el capítulo 7).

5.1 Principio de Especialización

La tecnología orientada a objetos tradicional no es suficientemente ágil para dar respuesta a un proceso de desarrollo rápido. Hay una gran falta de eficacia a la hora de llevar el diseño a código. La razón de estos problemas radica en la encapsulación *conjunta de todos* los elementos del dominio en un mismo módulo: la clase.

A la hora de hacer un diseño orientado a objetos se debe seguir el *Principio de Especialización*. Este principio establece que *toda clase que provenga del dominio no deberá encapsular más de un elemento del mismo y deberá encapsularlo completamente*.

Algunos ejemplos en los que la clase tradicional no está cumpliendo el Principio de Especialización:

- Las clases encapsulan las relaciones con las operaciones que las manipulan. Se está encapsulando *más de un elemento* del dominio (relaciones y operaciones).

- Las clases encapsulan las operaciones pero no en su totalidad; solo una parte de la misma (aquella que entra dentro de su responsabilidad). Por tanto el cambio de una operación afecta a varias clases. No se ha encapsulado dicho elemento – la operación – *de manera completa* en una clase.

- Los cambios en las reglas afectan a las operaciones (y por tanto a varias clases). En una misma clase se están encapsulando dos elementos: las operaciones y sus reglas.

Siguiendo el Principio de Especialización se propondrá a continuación otra forma de trasladar los distintos elementos del dominio a código. Se creará una arquitectura denominada RDM (Rapid Domain Modeling) derivada de la aplicación de dicho principio en distintas situaciones. Se verá como particionar las responsabilidades de las clases para cumplir los objetivos buscados.

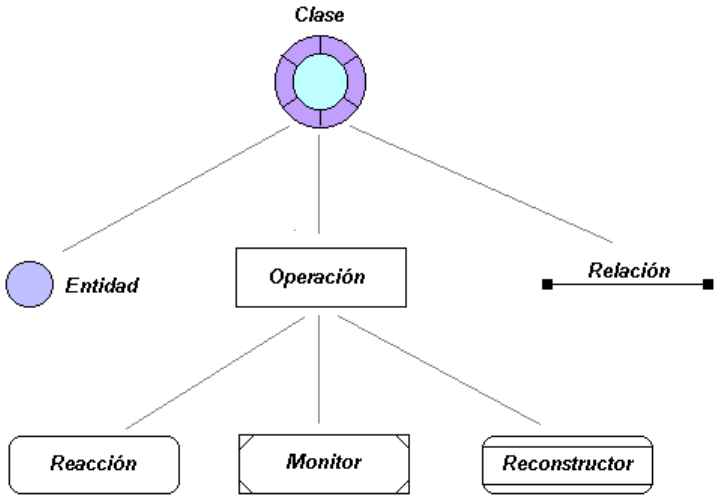

Imagen 5.1-1. La clase se especializa en elementos de más bajo nivel.

Este capítulo mostrará para cada uno de los elementos las ventajas de modelarlo independientemente. Sin embargo no se tratará su conexión por ahora. En el siguiente capítulo se verá además que para que los distintos elementos de la arquitectura puedan interactuar será necesario que la arquitectura cumpla otros principios complementarios: principio de Externalización (5.2.3.1), principio de Inconexión (5.2.1), principio de Conformidad y principio de Adyacencia (5.2.2).

Estos cuatro criterios junto con el de especialización serán los que dicten la forma de la arquitectura para conseguir los objetivos buscados.

5.2 Elementos de la Arquitectura RDM

En este apartado se irá tomando cada uno de los tipos de elementos del dominio y se razonará la forma conveniente de llevarlo al modelo para lograr los objetivos propuestos.

5.2.1 Representación de Entidades y Relaciones

Los datos se representarán como objetos especializados independientes. Solo estarán encargados de modelar la información propia de una entidad del dominio mediante propiedades. No deben tener ninguna operación[10]. De esta manera se les pueden aplicar nuevas operaciones sin que ello les afecte.

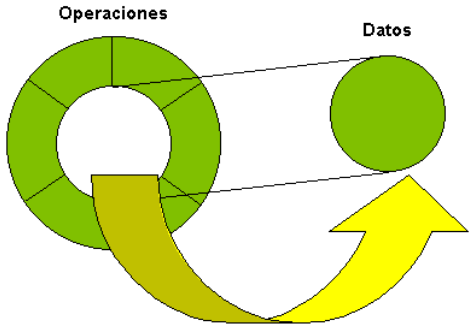

Imagen 5.2-1. Separación de métodos y atributos.

De la misma manera los datos tienen que ser a su vez independientes de otros datos. Es decir, no deben incluir relaciones. De esta manera no les afectará tampoco los cambios en otras entidades.

Como afirma la Ley de Demeter:

> "Each unit should only use a limited set of other units: only units "closely" related to the current unit" [Lieberherr96].

Relativo a esto mismo Rumbaugh afirma:

> "Avoid traversing multiple links or methods. A method should have limited knowledge of an object model. A method must be able to traverse links to obtain its neighbors and must be able to call operations on them, but it should not traverse a second link from the neighbor to a third class." [Rumbaugh91]

Rumbaugh afirma que para obtener independencia entre objetos deben limitarse a acceder a sus vecinos inmediatos. Pero entonces le afectan los cambios en ese primer nivel con lo que ella también afectará recursivamente a otras clases. Por ello la única forma de que las entidades sean verdaderamente independientes es no conocer *ni sus propias relaciones*. Debe seguirse el *Principio de Inconexión* el cual afirma que una entidad no debe conocer a ninguna otra entidad del dominio. E incluir las relaciones en la entidad es una forma de conocer a la entidad del otro extremo.

Sin embargo aunque una entidad no conozca sus relaciones no se debe renunciar a la siguiente funcionalidad:

- Se la debe poder relacionar con otras sin modificar su código fuente.

[10] En realidad ninguna *macrooperación*, como se verá posteriormente.

- Pero además deberá estar implícito el mantenimiento de la integridad de las entidades a la hora de establecer o eliminar una relación.

- Y, por supuesto, tiene que haber alguna forma de preguntarle *a una entidad* con qué otras entidades está relacionada con una determinada relación ya que si no las relaciones no tendrían sentido.

Y todo esto debe ser algo inmediato de representar en código ya que el añadir y eliminar relaciones es algo frecuente en el dominio.

5.2.2 Representación de Operaciones

En orientación a objetos la implementación de una operación queda repartida entre las distintas clases a las que afecta. Eso implica el *conocimiento* entre dichas clases por lo que los cambios sobre una de ellas afectan a las demás.

Por tanto las operaciones deben ser objetos independientes del modelo los cuales podrán aplicarse sobre distintas entidades del mismo pero sin afectar a las clases de éstas. Es decir, las entidades *no deberán verse afectadas* porque se añadan, modifiquen o eliminen operaciones sobre ellas.

La implementación de las operaciones debe hacerse de tal manera que:

- Las operaciones puedan ser aplicadas sobre otras entidades compatibles. Supóngase que se tiene una operación que calcula el total de los importes de un conjunto de entidades. Se debe poder aplicar dicha operación en otra aplicación que requiera calcular el importe total aunque las clases de las entidades sean otras. La única condición será que tengan algún valor que se quiera sumar.

- Las operaciones deben tener la mínima vulnerabilidad a los cambios en el dominio. Es decir que una operación no haya que modificarla porque al modelo se le añadan nuevas entidades, relaciones u otras operaciones. Mientras no se eliminen los datos que la operación manipula no debe verse afectada por cualquier cosa que pase en el resto del dominio. En el capítulo "Aplicación de la Arquitectura" se mostrará un ejemplo de esto.

Pero separar operaciones y datos no es sencillamente implementarlos en clases distintas. Supóngase la siguiente clase que junta datos y operaciones.

```
class DataWithOperations
{
       dataA()

       operationA()
}
```

Separar datos y operaciones no quiere decir solamente que se haga una clase para cada cosa.

```
class Data
{
      dataA()
}

class Operations
{
      setData(Data d)

      operationA()
}
```

Con esto no se consigue nada ya que sigue habiendo una dependencia entre ambas clases. La clase Operations sigue conociendo el nombre de la clase en la que se encapsulan los datos por lo que le siguen afectando los cambios de ésta. Además las operaciones siguen sin poder aplicarse sobre otros datos de otra clase.

La forma de conseguir los dos objetivos anteriores es siguiendo el *Principio de Conformidad* y el *Principio de Adyacencia*.

- El *Principio de Conformidad* establece que un objeto (una operación como caso particular) no debe requerir de otro que se le pase como parámetro *ninguna propiedad* que no sea esencial para su ejecución. Si no se cumple este principio se está disminuyendo innecesariamente el conjunto de entidades compatibles con la operación con el impacto que ello supone en las posibilidades de reutilización de la operación

- El *Principio de Adyacencia* establece que la demostración por parte de un parámetro de su compatibilidad con los requisitos impuestos por el objeto receptor debe poder hacerse en el mismo momento de la invocación. Este principio evita, como caso particular, que una operación exija a una entidad que tenga las propiedades requeridas una modificación de su tipo para poder acceder a las mismas.

Estos dos principios se obtienen directamente en los lenguajes sin control de tipos. Sin embargo la arquitectura pretende poder ser implementada en cualquier lenguaje orientada a objetos. Por tanto debe establecerse un mecanismo de conexión que aunque esté basado en tipos permita conectar los objetos de forma flexible sin que se conozcan entre sí.

El siguiente capítulo estará centrado exclusivamente en seleccionar la forma de comunicación de los distintos elementos de la arquitectura cumpliendo estos principios.

5.2.2.1 Microoperaciones y Macrooperaciones

Si las operaciones se implementan como objetos ¿Quiere esto decir que las entidades no deben tener ningún método? Pueden siempre que éstos se limiten a acceder únicamente a sus atributos o a agregados. A este tipo de operaciones se las denominará *operaciones de entidad* o *microoperaciones*. Si un método de la entidad accede a otras clases estará añadiendo un *conocimiento* de la existencia de las mismas que es lo que se está intentando eliminar. Recuérdese que incluso se propone que las entidades no conozcan ni sus relaciones, por lo cual las microoperaciones no deberán hacer uso de éstas.

Las operaciones que fundamentalmente se encuentran en el dominio son operaciones que afectan a conjuntos de objetos y que no se debe pretender asignar a una clase determinada. Por ejemplo

una facturación, una representación gráfica de un dibujo, una transformación, etc. son operaciones que se realizan sobre todo o parte del modelo. A este tipo de operaciones se las denominará *operaciones del dominio* o *macrooperaciones*.

La tecnología orientada a objetos *da soporte a las microoperaciones* y para modelar dominios *es necesario soporte para las macrooperaciones*. Lo que permite es añadir *métodos* a un *objeto* y se necesita añadir *operaciones* a un *dominio*.

5.2.3 Representación de Reglas de Negocio

Una vez establecido como deberán ser modeladas las entidades, las relaciones y las operaciones en la nueva arquitectura para alcanzar el objetivo propuesto toca a continuación contemplar las reglas de negocio.

Las reglas de negocio se dividirán en tres elementos especializados que se modelarán por separado:

- Monitores
- Reacciones
- Reconstructores

5.2.3.1 Monitores

Se tienen entidades a las que añadir atributos y relaciones. Se tienen operaciones que se pueden aplicar sobre distintos modelos siempre y cuando tengan las propiedades adecuadas. Pero una cosa es que la arquitectura sea suficientemente flexible para permitir estas combinaciones y otra cosa es que en un determinado dominio *no se pueda restringir* ciertos cambios en el modelo. Las validaciones son aquellas reglas que indican bajo qué condiciones se pueden aplicar las operaciones sobre las entidades. Un ejemplo de validación sería aquella que impidiera que se relacionara una entidad Cliente con una entidad Factura si dicho cliente tiene dos facturas anteriores sin pagar. En ese caso no podrá establecerse una nueva relación.

Uno de los principios de la orientación a objetos es que para garantizar la integridad de un objeto debe garantizarse que todo método de la clase debe iniciar su ejecución estando en un estado válido y dejarlo también en un estado válido. De eso se encargan las precondiciones, postcondiciones e invariantes [Meyer97].

Pero las precondiciones, postcondiciones e invariantes solo sirven para garantizar la integridad de *un solo objeto*. Lo mismo que se ha visto que las operaciones a modelar son a nivel de dominio y no a nivel de clase de la misma manera hay que mantener la integridad también a nivel *de todo el modelo*. De eso se encargan los *monitores* que son, por así decirlo, el equivalente a las precondiciones e invariantes *del modelo* (macroprecondiciones y macroinvariantes).

Las reglas deben ser objetos independientes. Pero para que se puedan conectar con el resto de los elementos se necesita que el modelo cumpla el *Principio de Externalización*. Este principio dice que *cualquier tipo de cambio* en el estado del modelo debe producir una notificación. A dichos

eventos debe poder suscribirse cualquier objeto. De esta manera el modelo permite que se le valide de manera externa *sin necesidad de que él tenga conciencia* de dichas validaciones. Así las validaciones no se implementan dentro de la entidad sino en unos objetos que monitorizan los cambios de estas actuando si detectan un estado inválido.

Pero no habrá un gran objeto monitor que escuche todos los eventos y monitorice todos los cambios. Dicho objeto debería conocer entonces todo modelo y sería sensible a sus modificaciones. Cada validación debe ser un objeto independiente que se limite a realizar una comprobación básica. Un ejemplo de validación podría ser que no se elimine una relación si uno de los objetos no cumple unas determinadas condiciones. A cada una de estas validaciones se la suscribe en la entidad o relación a controlar. De esta manera ante un cambio del modelo puede que no se active ningún monitor o que se activen varios.

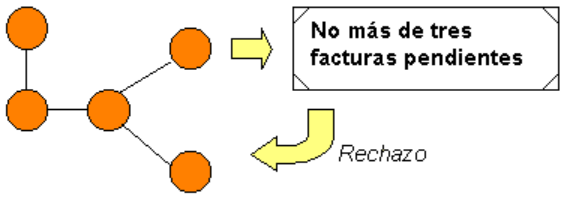

Imagen 5.2-2. Si el cambio del modelo no cumple una regla se rechaza la operación.

Ahora las operaciones simplemente modifican o añaden lo que quieran al modelo. Si alguno de los cambios no cumple alguna regla el monitor lo detectará y producirá una excepción que será recibida por la operación. De esta manera sabrá que las modificaciones no eran válidas sin necesidad de incluir las validaciones en su código.

Al extraer las validaciones como un elemento independiente se conseguirá:

- No tener que repetirlas en varias operaciones.
- No tener que modificar las operaciones si cambian las reglas.
- No correr el riesgo de olvidar añadir una regla a una operación que también debiera comprobarla.
- Poder añadir validaciones al modelo dinámicamente.
- Mayor facilidad para implementarlas ya que suelen estar asociadas únicamente a una relación o entidad y no necesitan un conocimiento global del modelo. Se limitan a monitorizar una pequeña parte del modelo. La monitorización del modelo en global se conseguirá mediante la *colaboración automática* de todas validaciones desconocidas entre sí (lo cual se verá posteriormente).

5.2.3.2 Reacciones

Cuando se dice que cada elemento del dominio debe ser implementado de manera completa e independiente hay que especificar qué se entiende por *completa* cuando el elemento al que se refiere es una operación.

Las operaciones hay que implementarlas de tal manera que estén centradas en una única operación autónoma del dominio. En el apartado anterior ya se ha visto que las validaciones no forman parte de una operación *completa*. Pero tampoco forman parte de ella las operaciones que se produzcan *como reacción* de la ejecución de ésta.

Por ejemplo supóngase que se implementa la operación "Venta de un artículo". La descripción que se obtiene del cliente explica que cada vez que vende un artículo decrementa el stock y si se ha llegado a cierto nivel se emite un pedido. Esto habitualmente se implementaría de la siguiente manera:

```
void vender(Artículo articulo, int cantidad)
{
      articulo.quitarExistencias(cantidad);

      if (articulo.existencias() < articulo.nivelMinimo())
            pedidos.add(new Pedido(articulo));
}
```

Sin embargo ahí se han implementado dos operaciones juntas. Una cosa es vender un artículo y otra hacer un pedido. Que se realicen de forma secuencial no quiere decir que haya que implementarlo junto. Para entender mejor el por qué no debe implementarse así supóngase que el usuario cambia de idea y propone alguna de las siguientes reglas:

- No quiere hacer el pedido si tiene suficiente cantidad de otro artículo que puede sustituirle.

- No quiere hacer pedidos para ciertos artículos aunque bajen su nivel de stock. Está evaluando la posibilidad de retirarlos.

- Decide que en vez de emitir un pedido después de cada venta prefiere esperar a final de mes y hacer un pedido conjunto para todos los artículos bajos de existencias.

Todos estos cambios de las reglas del dominio modifican la implementación de la operación *vender* cuando ella realmente no ha sido modificada; se sigue vendiendo de la misma manera. Pero al encapsular juntas dos operaciones los cambios en una afectan a la otra.

Sin embargo, volviendo al primer caso en que cada venta produce un pedido si es necesario, es obvio que existe una relación entre las operaciones que hay que modelar de alguna manera. Esta relación es de causa-efecto y hay que implementarla pero no en ninguna de las dos operaciones. El hacer un pedido es una *reacción* producida por la operación de venta. La forma de implementar esto debiera ser mediante la *reacción* como un elemento independiente y desconocido para las operaciones.

Las reacciones son otros objetos especializados también en modelar las reglas de negocio. Y al igual que los monitores también se basan en el *Principio de Externalización*. Ambos responden ante cambios del modelo pero aunque ambos se activan de la misma manera la diferencia fundamental entre ambos es:

- Las validaciones (implementadas en los monitores) son reglas de dominio que *cancelan* las macrooperaciones que no cumplen unos determinados criterios.

- Las reacciones son reglas del dominio que *extienden* las macrooperaciones a otras partes del modelo que éstas no deben conocer.

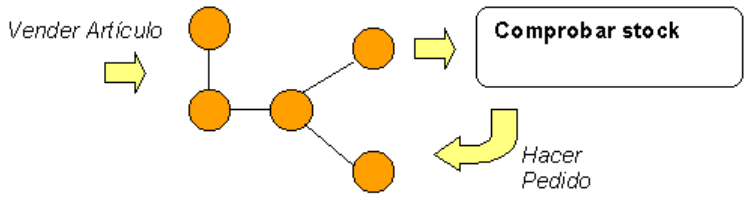

Imagen 5.2-3. La operación *vender* activa la operación *pedido*.

Cuando la operación *vender* modifica el stock de un artículo la reacción detecta el cambio del modelo y activa automáticamente la operación *pedido* si es necesario.

Al tener las reacciones separadas se puede:

- Se puede sustituir la regla por otra que refleje las necesidades cambiantes del cliente. Por ejemplo se puede cambiar la reacción por otra que comprueba que si es un determinado artículo a retirar entonces no se active la reacción.

- Eliminar la reacción y por tanto tener dos operaciones que se pueden invocar independientemente. Por ejemplo en el caso de que se quiera hacer un pedido a final de mes se podría invocar la operación *hacerPedido* directamente, no como reacción de la operación *vender*.

- Y finalmente se pueden activar varias operaciones mediante el disparo de reacciones en cascada sin que éstas se conozcan entre sí.

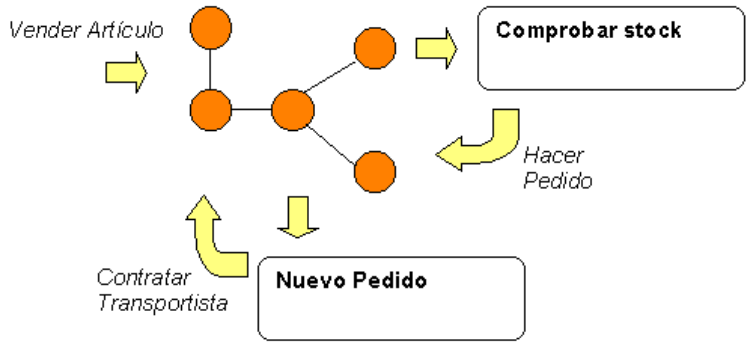

Imagen 5.2-4. Encadenamiento de operaciones mediante reacciones.

En resumen las reacciones cumplen dos objetivos:

- Permite separar dos operaciones conceptualmente distintas impidiendo la propagación de cambios y permitiendo su uso independiente.

- Hacer que una macrooperación alcance a entidades a las que debe afectar y sin embargo *no conoce*.

Por tanto las macrooperaciones ya no están implementadas en un objeto solamente. Una macrooperación es llevada a cabo por un objeto *operación* y todas aquellas *reacciones* que

desencadene. Por tanto las operaciones del dominio se están implementando de forma *descentralizada*. No hay un lugar donde estén escritas todas las acciones que forman parte de una operación global. Los pasos de una operación están repartidos por varios objetos cada uno de los cuales hace una operación básica y que se combinan en función de las reglas de negocio.

Nótese que aquí aparentemente hay un conflicto con el *Principio de Especialización* el cual dice que un objeto debe encapsular a un elemento del dominio de forma completa. Las macrooperaciones *se han repartido en varias clases* que es al fin y al cabo lo que hacen las tecnologías orientadas a objetos y que se intentaba evitar.

Sin embargo hay que hacer notar dos cosas:

- Cuando en programación orientada a objetos una operación se divide en varios métodos hay un conocimiento estático entre las clases. Aquí son *las reacciones* y no las propias operaciones las que se encargan de la secuenciación.

- Las operaciones realmente están encapsulando completamente a un elemento del dominio. Cada uno de los objetos operación que interviene en una macrooperación realiza una operación básica completa. La prueba de ello es que pueden ejecutarse de forma independiente.

5.2.3.3 Reconstructores

Hasta ahora se han ido cogiendo cada uno de los elementos del dominio y se han mostrados las ideas preliminares de como deberán ser modelados dentro de la arquitectura. Los reconstructores son los únicos elementos que no provienen del dominio. Sin embargo son los que hacen posible que la arquitectura aquí propuesta *sea implementable*.

Todos los demás elementos se han implementado de manera independiente y sin que se conozcan para que los cambios de unos no afecten a los otros y para que se puedan reutilizar de manera independiente. Se realizan operaciones sobre entidades lo cual activa monitores y reacciones. Sin embargo cada uno de ellos no se conoce entre sí. La validación no sabe a qué operación está validando ni ésta última sabe que la están validando ni que va a producir reacciones.

Sin embargo hay una situación en que ese desconocimiento mutuo plantea un problema.

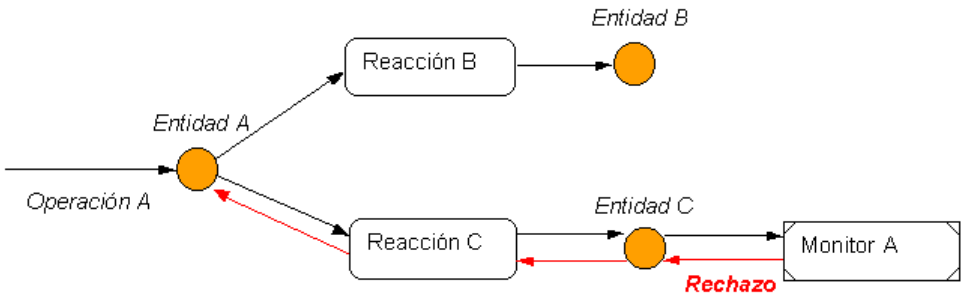

Imagen 5.2-5. Reacciones y rechazo de operaciones.

Supóngase que una operación denominada *operaciónA* modifica a una entidad *entidadA*. Al modificar dicha entidad se produce una reacciónB en entidadB. Resulta que la modificación de entidadA produce una segunda reacciónC que modifica entidadC. Pero al intentar modificar entidadC resulta que se viola una regla de negocio, por lo que se produce una excepción que *operaciónA* atrapa. Por tanto sabe que no cumple las reglas y no puede realizarse.

Sin embargo ¿cómo ha quedado el modelo? Debería haber quedado tal y como estaba antes de intentar la operación. Sin embargo la entidadB ha sido modificada. El modelo ha quedado en un estado inválido. Nótese cómo el modelo puede estar en un estado inválido aunque individualmente cada uno de los objetos esté en un estado válido.

¿Cómo arreglarlo? El único que sabe que se ha modificado la entidadB es la reacciónB. Sin embargo reacciónB no ha podido enterarse de que la operación ha quedado anulada ya que el camino de retorno que toma la excepción no pasa a través de ella. El que sabe que la operación no puede realizarse es operaciónA pero ésta tampoco puede avisar a la reacciónB para que corrija su acción porque ni siquiera sabe de su existencia.

Este problema no se presentaba con la programación orientada a objetos ya que lo habitual era que una operación fuera implementada mediante un método o una secuencia de métodos estática y conocida. Dentro de este método se describían secuencialmente las operaciones a realizar y las validaciones antes de realizar cada una. Si surgía algún problema y había que deshacer lo hecho se sabía todo lo que se había hecho previamente.

```
Procedure opera()
{
       Hacer operaciónA
       Hacer operaciónB
       Si (operaciónB falla)
              deshacer operaciónA
}
// Esto ya no se puede hacer así
```

Sin embargo ahora las operaciones se configuran dinámicamente en tiempo de ejecución en función de las reacciones y validaciones determinadas por la lógica de negocio. Cada uno de estos elementos atómicos no conoce de los demás para que puedan combinarse libremente. Sin embargo ese desconocimiento plantea aquí un problema de descoordinación a la hora de que unas reacciones deban saber de ciertas validaciones que impidan su ejecución.

La solución no pasa por hacer que las reacciones conozcan a los monitores. La solución que se ha adoptado es introducir un nuevo elemento en la arquitectura el cual estará *especializado* en evitar este problema de descoordinación manteniendo así la independencia entre los demás elementos: los reconstructures.

Si el *constructor* de una clase es un método que deja a *un objeto* en un estado inicial un *reconstructor* es un objeto que se encarga de dejar *al modelo en el último estado válido* que haya tenido. Es decir, el nombre pretende indicar que vuelven a construir (en el sentido de establecer un estado válido) objetos ya construidos.

Los reconstructores se activarán de forma automática cuando el modelo haya quedado a medio modificar sin necesidad de que las operaciones, entidades, monitores o reacciones tengan que saber todo lo que se ha modificado hasta el momento.

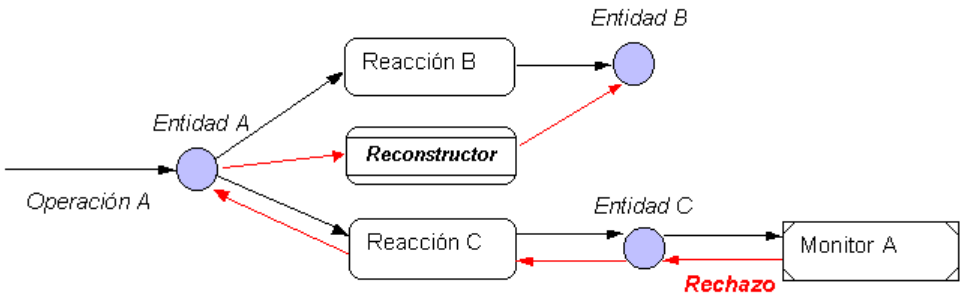

Imagen 5.2-6. Reconstrucción de todo el modelo ante un rechazo de la operación.

Si los monitores equivalían a macroprecondiciones y macroinvariantes los reconstructores equivalen a una forma mejorada de *macropostcondiciones*. Una postcondición comprueba que cuando el método finalice éste deje al objeto en un estado válido [Meyer98]. Si no queda en un estado válido simplemente produce una excepción. Los reconstructores aseguran que cuando finalice una macrooperación se *va a dejar al modelo de objetos* en un estado válido. Es decir o se ha realizado la operación con éxito o bien la operación se ha anulado por un monitor y el modelo se encontrará *exactamente igual* que en el momento anterior a invocar la macrooperación (gracias a la labor de fondo de los reconstructores).

Los reconstructores son similares al concepto de transacción en los sistemas de bases de datos. Sin embargo si el modelo no está en una base de datos no se tiene ningún mecanismo de recuperación; solo se ofrece la implementación del concepto de transacción en los SGBD. No se debe volcar un modelo a una base de datos sólo por obtener transacciones cuando sus características no se prestan a dicho tipo de almacenamiento. Se debe tener esa funcionalidad de forma nativa; *debe ser proporcionado* por el entorno de ejecución de forma automática para poder construir aplicaciones verdaderamente sólidas[11].

5.3 Resumen de la Arquitectura

Se ha afirmado que para cumplir el objetivo es necesaria *una arquitectura que esté basada en la representación especializada e independiente de los distintos elementos del dominio proporciona una reducción en los tiempos de desarrollo y favorece la reutilización*.

Lo que se ha hecho en este capítulo es ver cómo se debe modelar cada uno de los elementos del dominio siguiendo dicha afirmación. La arquitectura estará formada por:

- Un conjunto de objetos que modelarán (exclusivamente) las entidades y sus propiedades.

[11] En un capítulo posterior se mostrará la implementación de los reconstructores para obtener automáticamente la recuperación de modelos en memoria. La implementación de las transacciones que se obtiene en los SGBD comerciales podría considerarse otra implementación posible de los reconstructores para el caso de que el modelo se encuentre en memoria secundaria.

- Sobre dichas entidades se aplicarán una serie de relaciones extraídas del dominio de tal manera que no haya que modificar las clases ni implementar en cada una de ellas todo el código que garantice la integridad de la relación.

- Sobre la estructura de entidades y relaciones se aplican objetos especializados en macrooperaciones (operaciones de dominio).

- Los objetos monitores impedirán la aplicación incorrecta de las macrooperaciones sobre el modelo garantizando la integridad del mismo.

- Las reacciones se encargarán de extender las macrooperaciones para que, siguiendo la lógica de negocio, actúen sobre nuevas entidades u otras operaciones que desconocen.

- Los reconstructores asegurarán la integridad del modelo recuperándolo de estados inválidos. Permite así la independencia entre los demás elementos al no tener que saber cada uno de ellos lo que el otro ha cambiado.

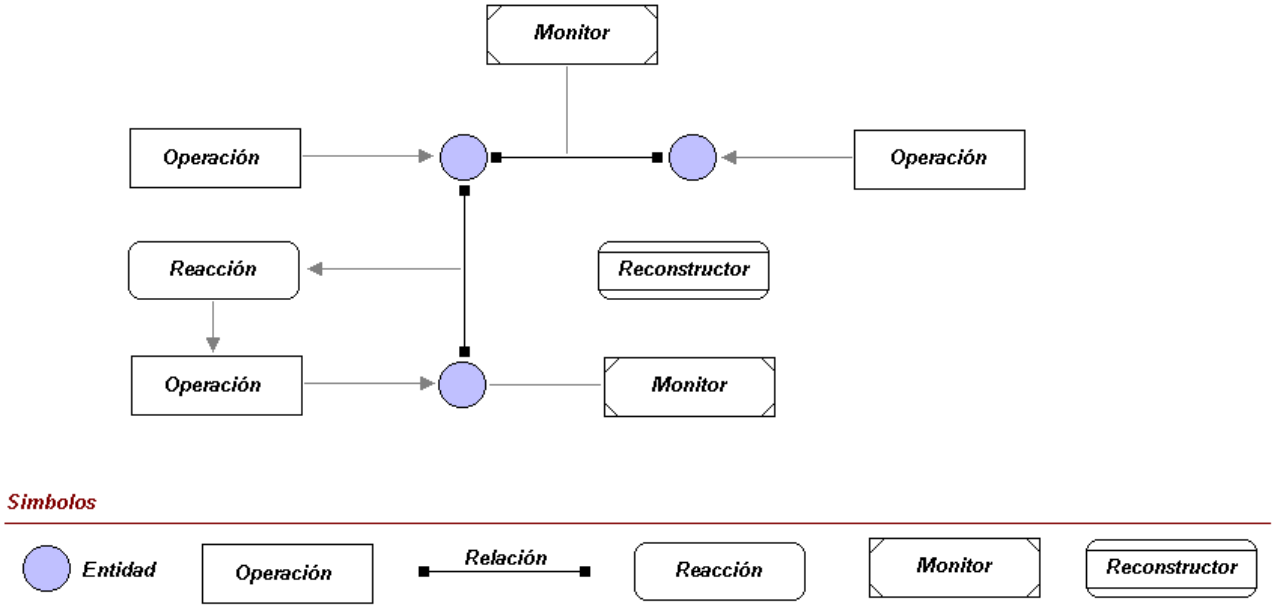

Imagen 5.3-1. Elementos de la Arquitectura RDM

En la figura anterior se muestran los símbolos utilizados en este documento para representar cada uno de los elementos de la arquitectura. Se ve cómo las entidades (círculos) son unidas mediante relaciones y sobre ambos se aplican operaciones. La actuación de las operaciones estará controlada por los monitores y podrá ser extendida por las reacciones. Y pendientes de todos ellos están los reconstructores para asegurar la consistencia del modelo.

Por tanto en el diseño en vez de hablar de objetos genéricos que tengan cualquier comportamiento se hablará de objetos más primitivos. Ya no se tomará al objeto y se le tratará como si pudiera tener combinados, por ejemplo, cualquier cantidad de información y cualquier cantidad de lógica. Un objeto representará información o bien representará lógica (o reglas, o reacciones, etc.) Y sabiendo esto se le tratará de una forma más sencilla y enfocada.

Se pasa de modelar de la forma tradicional, representada en la siguiente imagen:

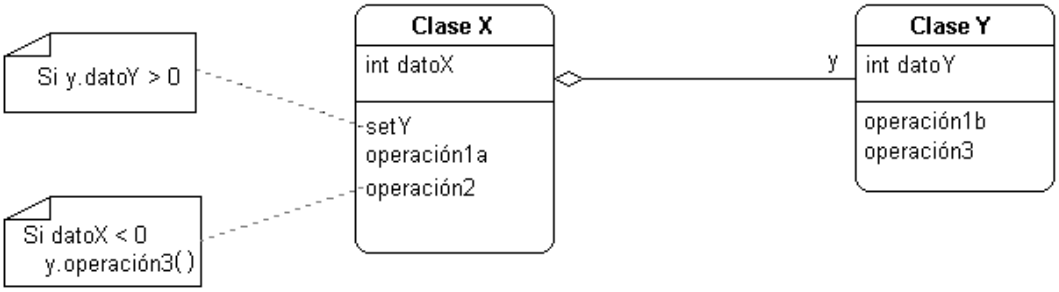

Imagen 5.3-2. Modelado tradicional.

A una representación en la que los elementos permanecen independientes pudiendo combinarse más fácilmente.

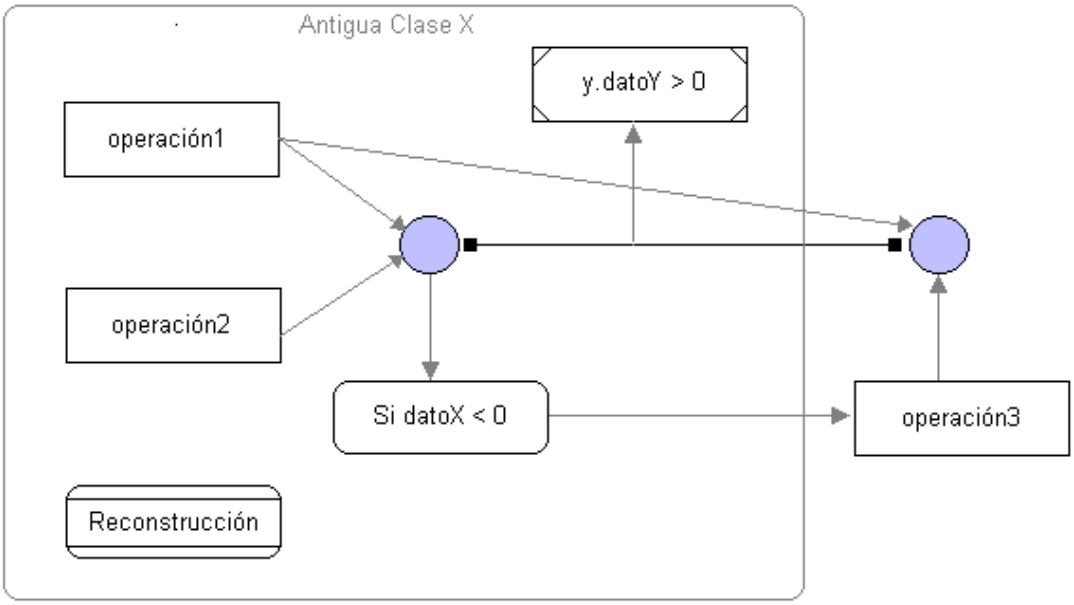

Imagen 5.3-3. Modelado RDM. La clase se descompone en elementos independientes.

Una vez establecido los elementos que tiene que formar parte de la arquitectura y la misión de cada uno en el siguiente capítulo se concretará la forma de comunicarlos. Detrás de éste, en el capítulo 7, se verá ya finalmente la implementación del framework obtenido mediante la aplicación de la tesis aquí presentada.

5.4 Especialización: ¿Vuelta a la Programación Estructurada?

Como puede haberse observado una de las consecuencias del Principio de Especialización es que *los datos y las operaciones* (entidades y operaciones) deben estar separados. Sin embargo esto sería volver otra vez a la programación estructurada con los problemas que esta suponía [Meyer97]. Precisamente para solucionar dichos problemas surgió la orientación a objetos.

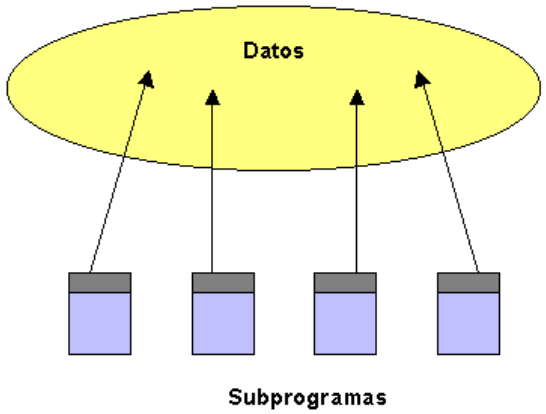

Imagen 5.4-1. Los datos separados de las operaciones.

Bertrand Meyer [Meyer97] afirma que los problemas fundamentales que la separación de los datos y las operaciones plantea son:

- Propagación de cambios. El cambiar la estructura de datos afecta a procedimientos y viceversa.

- Propagación de errores. Si un módulo erróneo deja el modelo en un estado inválido el error se propaga por otros módulos.

Es cierto que se tienen dichos problemas y es cierto que la orientación a objetos los soluciona. Sin embargo el precio a pagar ha sido encontrarse *otros nuevos problemas* debidos a la encapsulación (los cuales se han visto en el capítulo 2). Se están planteando otras dependencias entre elementos:

- Como cada clase del modelo tiene la *responsabilidad de mantener sus relaciones* y operaciones se tiene que modificar con cada variación de éstos.

- Como las operaciones *conocen las clases* del modelo (y por tanto sus atributos y relaciones) se verán afectadas cada vez que cambie el interfaz de éstas.

- Como las operaciones *conocen las clases* del modelo no puede hacerse que esas operaciones se apliquen a otras del mismo o de otro modelo *aunque cumplan* los requisitos de éstas. Las operaciones están atadas a unas clase del modelo, no a una funcionalidad de las mismas.

Por tanto se trata de encontrar una solución intermedia que evite los problemas de ambos enfoques y ofrezca sus beneficios. Y esa solución pasa por impedir los inconvenientes de la separación de datos y operaciones *pero sin recurrir* a juntarlos.

Para ello hay que observar que el inconveniente de dicha separación no estaba en el hecho en sí de que estuvieran separados sino en el hecho de que *se conocieran demasiado*. Había una gran dependencia entre ellos de tal manera que era imposible reutilizar una operación sin sus estructuras de datos. Todo formaba una red entrelazada de tal manera que no había piezas autónomas que se pudieran extraer limpiamente.

Si se elimina este conocimiento mutuo se podrá volver a tener datos y acciones separados pero reutilizables. En el capítulo siguiente se mostrará como conectar los elementos para obtener la independencia que la programación estructurada no tenía. Los datos podrán ser usados por nuevas operaciones (sin modificar la clase) y las operaciones podrán ser aplicadas sobre otros datos que no conocían.

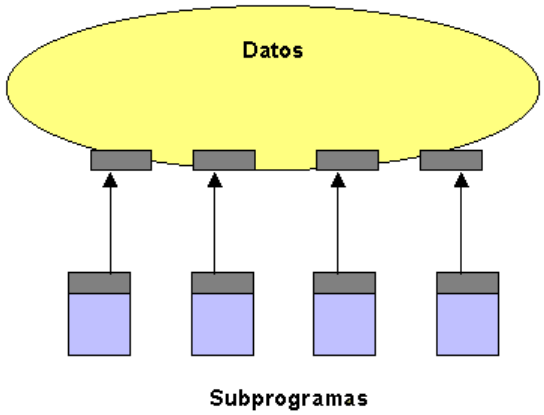

Imagen 5.4-2. Separación pero manteniendo independencia.

De esta manera se evitará el primero de los problemas planteados por Meyer (propagación de cambios). El segundo problema (propagación de errores) se evitará con dos elementos que ya han sido presentados: los monitores y los reconstructores. Éstos no solo evitarán que los errores que una operación haya podido introducir en el modelo se propaguen a otras operaciones sino que *ni siquiera permitirán* que a su finalización dicha operación pueda dejar errores en el modelo.

Pero en cualquier caso no hay que olvidar que cada uno de los elementos de la arquitectura se implementa como un objeto. Por tanto *no se está proponiendo eliminar las clases sino* hacer *clases más especializadas* en una misión.

6 Mecanismos de Comunicación entre Objetos

6.1 Introducción

En el capítulo anterior se proponía construir la arquitectura de las aplicaciones mediante la aplicación del *Principio de Especialización* a la hora de trasladar el dominio al diseño. Se mostró como modelar los principales elementos del dominio y las ventajas de hacerlo de dicha manera.

Pero para poder entrar en más detalle queda por determinar un aspecto fundamental: cómo será la comunicación entre los distintos elementos del dominio (especialmente entre las operaciones y las entidades). Para cumplir los objetivos buscados resultará que la arquitectura (y por tanto la implementación elegida) deberá cumplir además los principios adicionales introducidos en el capítulo anterior:

- Principio de Conformidad (5.2.2).
- Principio de Adyacencia (5.2.2).
- Principio de Inconexión (5.2.1).
- Principio de Externalización (5.2.3.1).

Una vez que se decida la forma de comunicación de los distintos elementos en el capítulo siguiente ya se podrá concretar la implementación de la arquitectura.

6.2 Mecanismos de Comunicación

Se tienen varios elementos del dominio implementados de forma separada para poder combinarlos y modificarlos de forma independiente. Pero ¿cómo comunicar dichos elementos para que puedan interactuar de manera flexible sin conocerse?

La razón fundamental de la falta de reutilización de las operaciones es el *incorporar demasiado conocimiento* sobre las entidades que manipula. Todo lo que sepa una operación sobre una entidad se transforma en lazos de unión los cuales serán impedimento para su separación.

Por ejemplo supóngase que en lenguaje estructurado un procedimiento *f()* llama a otro procedimiento *nombre()*.

```
void f()
{
      valor = nombre();   /* Imposible reutilizar f() de manera
independiente de "nombre()" */
}
```

El procedimiento *f()* ya no puede reutilizarse en ninguna otra aplicación si no se lleva también el procedimento *nombre()*. No puede hacerse que llame a otro procedimiento que realice la misma función que *nombre()* sin que haya que modificar el código fuente.

Los mecanismos de conexión deben permitir que *f() pueda* utilizar *nombre()* o cualquier otra operación equivalente. De esta manera *no dependerá* de ella para su reutilización.

6.2.1 Clases

En las tecnologías orientadas a objetos las clases consiguen una mayor independencia que en el caso anterior basado en funciones. Supóngase que la clase A pueda parametrizarse con un objeto de tipo B (por ejemplo mediante el constructor).

```
class A
{
      private B objeto;

      A(B objeto)
      {
            this.objeto = objeto;
      }

      void f() {
            valor = objeto.nombre();
      }
}
```

Ahora el método *f()* en vez de comunicarse directamente con una entidad fija se puede comunicar con cualquiera de los objetos de la clase B o de sus derivadas. Gracias al polimorfismo el método *f()* no sabe quien es el receptor del mensaje ni cual será la implementación que se ejecute, por lo que a la clase A se la podrá conectar con aquel objeto compatible con el rol B que realice la funcionalidad *nombre()* de la manera más adecuada.

Sin embargo las clases tienen el problema de que se sigue sabiendo demasiado del objeto con el cual comunicarse. Se conoce al tipo padre del objeto, es decir, la clase B. Si se quisiera que un objeto A se comunicara con un nuevo objeto X (sabiendo que X cumple la funcionalidad requerida – en este caso tener un *nombre*) no se podría conectar porque no es derivado de la clase B.

El problema de las clases, y por lo que no son adecuadas es esta arquitectura, es que no cumplen el *Principio de Conformidad*. Recuérdese que este principio establece que un objeto (una operación como caso particular) no debe requerir de otro que se le pase como parámetro *ninguna propiedad* que no sea esencial para su ejecución. Como se comentó anteriormente si no se cumple este principio entonces se está disminuyendo innecesariamente el conjunto de entidades compatibles con la operación con el impacto que ello supone en las posibilidades de reutilización de la operación.

Las clases, como mecanismo de conexión, no cumplen el principio de conformidad ya que utilizan como test de compatibilidad el que los datos se encuentren en una determinada clase, es decir, no solo se trata de tener una serie de propiedades sino además formar parte de una determinada *estructura jerárquica de tipos*, lo cual no es realmente necesario para la operación.

Usar clases como mecanismo de conexión sería como si para evaluar a un candidato a un puesto de trabajo se exigiera que el candidato fuera hijo de una determinada persona. Se rechazarían directamente cualquier otro candidato aunque tuviera la preparación adecuada.

6.2.2 Interfaces

No importa quién sea su tipo padre, lo que importa es ¿se quiere que esta entidad sea manipulada por esta operación? ¿Tiene las propiedades necesarias para ello? Cuando una operación recibe una entidad lo que quiere es que le proporcione o permita almacenar determinada información. No hay que exigir más de lo que se necesita. Se necesita un mecanismo que permita identificar la funcionalidad sin incluir relaciones jerárquicas irrelevantes.

Ese papel lo cumplen los *interfaces*. Cuando se declara un interfaz como parámetro lo que se está pidiendo es *cualquier* objeto que realice las operaciones que el mismo declara.

```
interface IB {
	String nombre();
}

class A
{
	private IB objeto;

	A(IB objeto)
	{
		this.objeto = objeto;
	}

	void f() {
		valor = objeto.nombre();
	}
}

class UnCandidato extends PadreA implements IB {
	String nombre() { ... }
}


class OtroCandidato extends PadreB implements IB {
	String nombre() { ... }
}
```

Ahora se ha ampliado el conjunto de objetos compatibles permitiendo conectar cualquiera que realice la funcionalidad *nombre()* sin importar su clase ni la clase padre de la que deriven.

Por tanto la forma de aplicar los interfaces sería que cada operación declarara mediante un interfaz aquellas propiedades que requiera para su aplicación. Aquellas entidades que deseen ser manipuladas por dicha operación deberán implementarlo.

Los interfaces, y en general los tipos, son como títulos acreditativos que los objetos candidatos deben poseer para poder ser conectados. Al igual que en el mundo real no se acepta a un candidato a una plaza que no tenga el título requerido tampoco en el software se acepta a un objeto que no tenga el tipo adecuado. Y esto se hará aunque en ambos casos el candidato tenga la capacidad de realizar la labor adecuadamente.

Sin embargo hay una razón fundamental por la que ese sistema es aplicable en el mundo real y sin embargo desde el punto de vista de la reutilización no es aceptable en el software: en el mundo real no hay mayor inconveniente para la empresa si se descarta un candidato válido por no tener título ya que un candidato titulado será contratado; el puesto generalmente quedará cubierto. Sin embargo si una entidad no se puede conectar a una operación esta última no se podrá reutilizar. Solo hay un candidato que si se rechaza injustamente la plaza quedará vacante.

¿Y qué probabilidad hay de que un objeto candidato implemente el interfaz adecuado? Prácticamente ninguna. En el mundo real los títulos universitarios son conocidos. Están tipificados por algún organismo y tanto el examinador como el candidato tienen la misma referencia. Sin embargo esto no es así en el software. No existe un conjunto de interfaces catalogados y conocidos por todos. En cada aplicación surgen interfaces específicos a los requisitos del dominio de la aplicación. ¿Es posible que haya alguna entidad de otro dominio que implemente dichos interfaces y pueda ser conectado?

Por tanto no puede usarse un mecanismo de conexión directa basada en tipos porque rara vez se encontrarán entidades compatibles.

En definitiva con los interfaces:

- Si no se tiene acceso a los fuentes es prácticamente imposible hacer que el objeto pueda conectarse a la operación, ya que no implementará el interfaz requerido.

- Si se tiene acceso a los fuentes podrá añadirse la declaración del interfaz, pero cada conexión del objeto a un nuevo modelo no debe suponer una modificación de su tipo porque se volverían a tener los problemas iniciales.

Los problemas de los interfaces se deben a que no cumplen el *Principio de Adyacencia* el cual establece que la demostración por parte de un objeto de su compatibilidad con los requisitos requeridos por una operación debe poder hacerse *en el mismo momento* de la invocación. Con los interfaces se valida la compatibilidad comprobando las declaraciones que se hicieron *a la hora de definir la clase del objeto*. Por tanto para añadir nuevas compatibilidades se requiere modificar el tipo.

En cambio que se cumpla el principio de adyacencia significa que la demostración pueda hacerse en el momento de la invocación, es decir, que sea el programador que intente conectar los elementos el que, sabiendo la adecuación de la entidad a la operación, *tenga alguna posibilidad de demostrarlo* aunque no fuera previsto por los programadores de los elementos a conectar (la operación y la entidad).

Utilizar interfaces como forma de compatibilización de conexiones sería como si a un candidato a un puesto de programador se le preguntara si en el momento de su nacimiento (de la creación de su tipo) el padre declaró en el registro que su hijo iba a ser informático. Por mucho que haya aprendido, como no lo declaró en su momento, es rechazado (aunque se le da la opción de modificar su tipo y volver a nacer/instanciarse con la nueva declaración). El principio de

adyacencia dice que el futuro de un objeto no quede condicionado por las intenciones de su creador, sino por la capacidad que pueda demostrar en el contexto que se plantee.

6.2.3 Introspección

Dado que no puede haber un mecanismo de conexión directa basada en tipos (ya que, como se vio anteriormente, los tipos son locales a cada aplicación) otra alternativa es utilizar un mecanismo no basado en tipos: la Introspección.

La Introspección es la capacidad de un sistema para poder inspeccionar u observar, pero no modificar, los objetos de un sistema [Foote92]. Con este mecanismo se facilita al programador acceder al estado del sistema en tiempo de ejecución: el sistema ofrece la capacidad de conocerse a sí mismo.

Esta característica se ha utilizado de forma práctica en numerosos sistemas. A modo de ejemplo la plataforma Java [Kramer96] ofrece introspección mediante su paquete `java.reflect` [Sun97d].

Otro ejemplo de introspección, en este caso para código nativo compilado, es la adición de información en tiempo de ejecución al estándar ISO/ANSI C++ (RTTI, *Run-Time Type Information*) [Kalev98]. Este mecanismo introspectivo permite conocer la clase de la que es instancia un objeto, para poder ahormar éste de forma segura respecto al tipo [Cardelli97].

En definitiva la Introspección permite interrogar a un objeto sobre los miembros que tiene de manera que puede averiguarse la funcionalidad que puede realizar y así invocarla.

```
void f(Object entidad)
{
       List methods = entidad.getMethods();
       Method methodNombre = methods.get("nombre");
       if (methodNombre != null)
              nombre = methodNombre.invoke();
}
```

La Introspección aparentemente cumple tanto el principio de conformidad como el de adyacencia. No importa el tipo del objeto (ya que el parámetro es de tipo *Object*) y además se le puede pedir exclusivamente las propiedades que se necesiten (sin importar que tenga otras adicionales).

Sin embargo la realidad es que no cumple ninguno de los dos principios. Supóngase que la operación acepta entidades que tengan un atributo llamado *nombre*. Sin embargo el candidato que se quiere conectar tiene un atributo llamado *identificador* que cumple la misma misión: identificar al objeto. La operación, al no encontrar ningún atributo *nombre*, rechazará la entidad.

No hay nada que se pueda hacer en el momento de la conexión para hacer compatible el objeto (habría que modificar el fuente para cambiar el nombre al atributo). Por tanto la Introspección, al igual que los interfaces, tampoco cumple el *Principio de Adyacencia*.

El problema es que lo que realmente necesita la operación de la entidad es que ésta tenga un miembro cuya semántica sea la de identificar al objeto. Eso es lo que realmente necesita. Sin

embargo impone también que ese miembro tenga que llamarse *nombre*. Está pidiendo más de lo que necesita por lo que tampoco está cumpliendo el *Principio de Conformidad*. Y el no cumplir este principio supone, como así ha sido en este caso, descartar entidades compatibles y la falta de reutilización que eso conlleva.

Por tanto en la conexión de una entidad con una operación debe comprobarse que la entidad *cumpla con la semántica* de cada una de las propiedades que se le requieren *sin importar si coincide el léxico* que hayan utilizado cada uno de los elementos para denominarlas. Esto es una consecuencia del *Principio de Conformidad* ya que el hecho de que la operación exija que los miembros se llamen de una determinada manera no es esencial (o no debería serlo) máxime cuando es prácticamente imposible si los objetos no están hechos por el mismo programador.

Otro inconveniente de la Introspección es que está basado en una característica del lenguaje concreto de implementación. No todos los lenguajes tienen Introspección. Se quiere una solución aplicable en cualquier lenguaje orientado a objetos.

6.2.4 Conexión Indirecta: Patrón Adapter

Dado que no puede haber comunicación *directa* basada en tipos otra opción es realizar una comunicación *indirecta* a través de un objeto intermediario. El patrón Adapter [Gamma94] se utiliza cuando un objeto quiere comunicarse con otro pero este último no cumple el tipo que espera el emisor. Por tanto lo que se hace es crear una nueva clase auxiliar intermedia que cumpla el interfaz requerido por el emisor y reenvíe las peticiones al receptor adecuado.

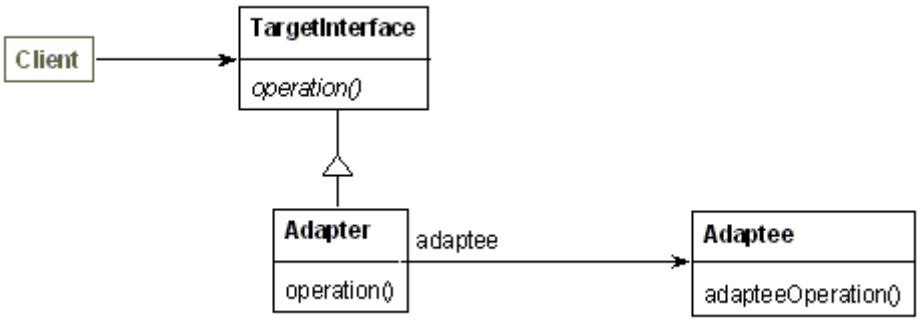

Imagen 6.2-1. Patrón Adapter

Supóngase que se tiene una operación que acepta como entidad a aquella que cumpla el interfaz *Rol*, es decir, aquella que tenga un método que devuelva un nombre.

```
interface Rol {
	String nombre();
}

class Operation
{
	void f(Rol candidato) {
		valor = candidato.nombre();
	}
}
```

Se quiere conectar la operación con una entidad de la clase *Candidato*. El problema es que dicha entidad no implementa el interfaz *Rol* aunque sin embargo tiene la propiedad que la operación requiere a través del método *identificador*.

```
class Candidato {
	String identificador() { ... }
}
```

Por tanto hay que conseguir que cuando la operación invoque al método *nombre* la petición pase a ser despachada por el método *identificador* de la entidad. Para ello lo que hay que hacer en el momento de la conexión es *adaptar* la entidad al *Rol* utilizando una clase *Adaptador*. Esta clase implementa el interfaz *Rol*, por lo que es aceptada por la operación. Pero cuando pida al adaptador su nombre lo que éste hace es devolver el identificador de la entidad.

```
class Adaptador implements Rol
{
	private Candidato objeto;

	Adaptador(Candidato objeto)
	{
		this.objeto = objeto;
	}

	String nombre() { return objeto.identificador(); }
}
```

Siguiendo con la analogía entre el software y el hardware es como cuando se quiere conectar un ratón con un puerto serie de 9 pines en un ordenador con un conector de 25. Se utiliza un *adaptador* con un lado de 9 pines y otro de 25. Lo mismo ocurre con las operaciones y las entidades. Por un lado del adaptador software se conecta la operación y por el otro la entidad realizándose la traducción necesaria para pasar de un formato a otro.

Los adaptadores son clases sencillas que generalmente siempre tienen la misma estructura:

- Un constructor en el que se pasa el objeto a adaptar.
- Cada uno de los métodos del interfaz en cuya implementación suele limitarse a invocar alguno de los métodos del objeto pasado en el constructor.

La forma de conectar el objeto a la operación ahora sería:

```
Operación operation = new ExampleOperation();
Candidato entidad = new Candidato();

Adaptador adaptador = new Adaptador(entidad);
operation.f(adaptador);
```

Nótese que ahora se cumplen ambos principios:

- El principio de *Conformidad* se cumple ya que se pide exclusivamente lo que se necesita: un objeto que tenga una forma de identificarse independientemente de cual sea su clase e independientemente de cómo se llame el método para obtenerlo.

- El principio de *Adyacencia* se cumple porque en el mismo lugar de la conexión el programador, mediante un adaptador, ha podido demostrar la adecuación del objeto al rol. No ha tenido que modificar ni el objeto ni la operación.

Aunque los adaptadores son una solución aceptable como mecanismo de conexión entre operaciones y objetos sin embargo presenta el problema de identidad el cual, aunque no suele presentarse frecuentemente, hay que tenerlo en cuenta. Este problema puede aparecer cuando el mismo objeto está siendo utiliza a través de distintos adaptadores simultáneamente

```
Object persona = new Persona();
AdaptadorA adaptadorA = new AdaptadorA(persona);
AdaptadorA adaptadorB = new AdaptadorB(persona);

Set set = new Set();      // Conjunto: no admite elemento
repetidos
set.add(adaptadorA);
set.add(adaptadorB);      // ¿Debería aceptarlo?
```

En el caso anterior se crea una persona a la que se la añaden dos adaptadores. A continuación se crea un conjunto (estructura de datos caracterizada por no admitir elementos repetidos) y se intenta guardar dos veces la misma persona. El conjunto lógicamente acepta a los dos adaptadores, ya que para él son objetos distintos, pero sin embargo es realmente la misma persona. Dicho de otra manera, cada adaptador tiene su propia identidad que es distinta a la del propio objeto persona. Sin embargo los adaptadores son dos formas de acceder a la misma persona y a la hora de compararlos todos deberían formar una misma entidad. Si ahora se aplicara una operación sobre todos los elementos del conjunto se obtendría que misteriosamente se aplica dos veces sobre la misma persona.

Este problema de identidad algunos lenguajes lo solucionan estableciendo un mensaje común que permita obtener la identidad del objeto [Box99] (si es un adaptador devuelve la del objeto al que adapta) y otros mediante alguna característica del lenguaje que permita fusionar objetos manteniendo una misma funcionalidad [Microsoft95]. Sin embargo estas son soluciones específicas del lenguaje.

Otro inconveniente es el hecho de que aumenta el número de clases auxiliares por lo que se recarga el diseño (especialmente los diagramas de clases). Esta recarga será notable si lo que hay que adaptar es *un grafo completo de objetos* relacionados en vez de un simple objeto como se ha hecho en los ejemplos.

6.2.5 Variable State Pattern

El patrón Variable State Pattern (VSP) [Beck96] se utiliza para acceder de forma uniforme a los miembros de un objeto. Se basa en que los objeto definan dos métodos:

```
void setMember(String member, Object Value);
Object getMember(String member);
```

De esta manera en vez de tener que conocer y utilizar un método distinto para acceder a cada atributo basta con conocer a uno predefinido que sirve para acceder a todos ellos.

```
// Tradicionalmente
persona.getName();
persona.getParent();


// Con VSP
persona.getMember("Name");
persona.getMember("Parent");
```

Este patrón se suele utilizar también para poder añadir a los objetos nuevos atributos en tiempo de ejecución, aunque hay que tener en cuenta que esa es otra característica derivada, siendo el hecho de no tener que conocer un método distinto para cada atributo lo que realmente es relevante para el tema aquí tratado.

Por tanto la forma de usar el Variable State Pattern en las operaciones en principio sería muy similar a la Introspección:

```
void f(Entity entidad)
{
	valor = entidad.getMember("nombre");
	if (valor == null) {
		// Entidad no compatible
	}
}
```

En principio el Variable State Pattern presenta los mismos problemas que la Introspección, es decir, que no funciona cuando la entidad no ha denominado al miembro exactamente igual que la operación.

Sin embargo hay una diferencia fundamental entre la Introspección y el Variable State Pattern. Cuando se examina a un objeto mediante la Introspección el objeto examinado es un espectador pasivo; es el lenguaje el que realiza dicho examen sin la intervención del objeto. Sin embargo con el VSP es el propio objeto al que se le pide la información. Por tanto puede resolver cualquier problema léxico adecuadamente.

Para ello se propone hacer una variación del VSP que permita *separar la semántica* de los miembros *de los nombres* por los cuales se quiera hacer referencia a los mismos. Es decir, un objeto puede tener un miembro cuya semántica sea la de identificar al objeto y a ese miembro se debe poder acceder por distintas etiquetas (nombre, name, id, identificador, etc.) Por tanto en esta nueva versión del VSP se añade un tercer método que permita añadir sinónimos a los miembros del objeto.

```
void setMember(String member, Object Value);
Object getMember(String member);
void addSynonym(String name, String anotherName);
```

Por tanto ahora para conectar a una operación que espera un miembro llamado *nombre* una entidad que denomina a ese miembro *identificador* bastaría con hacer la declaración correspondiente antes de conectar el objeto.

```
Operation operation = new ExampleOperation();
Candidato entidad = new Candidato();

entidad.addSynonym("identificador", "nombre");
operation.f(entidad);
```

Y por tanto la operación se encontrará con que efectivamente existe un miembro *nombre* aunque realmente está accediendo al miembro *identificador*.

Esa es la diferencia fundamental con la Introspección. Usar la Introspección es como entrar a una farmacia buscando una marca de medicamento concreta y, al no verlo en los estantes, determinar que esa farmacia no es adecuada. En cambio el VSP sería como ir al dependiente – al propio objeto – a decirle las características del medicamento buscado y que sea el dependiente el que ofrezca el más adecuado sin importar el nombre.

En resumen el patrón Variable State también cumple los dos principios:

- El principio de *Conformidad* se cumple ya que se pide únicamente los miembros que se necesitan sin importar la forma de denominarlos que utilice el objeto.

- El principio de *Adyacencia* se cumple porque en el mismo lugar de la conexión el programador, declarando los sinónimos adecuados, puede demostrar la adecuación del objeto al rol.

El Variable State Pattern, comparado con los adaptadores, tiene las siguientes ventajas:

- No tiene el problema de identidad. Si el objeto se inserta en varias operaciones todas acceden a él a través del mismo identificador.

- No hace falta crear nuevas clases auxiliares. Es el propio objeto el que busca el miembro adecuado.

Pero también tiene los inconvenientes

- Solo es adecuado cuando no es más que una mera traducción léxica. No es adecuado si la traducción requiere transformaciones de parámetros o el desglose de la llamada en varios métodos del objeto.

- Aunque en la práctica es poco probable que se presente, también puede ocurrir el caso de que el objeto se conecte a dos operaciones y las dos se refieran con el mismo nombre a distintos miembros (homónimos).

6.2.6 Conclusiones

El Variable State Pattern no suele incluirse dentro de los mecanismos de comunicación entre objetos ya que están más orientados al acceso a datos y no a la invocación de métodos. En ese sentido suelen utilizarse los interfaces exclusivamente[12].

Pero sin embargo en nuestra arquitectura los *elementos están especializados y las entidades no tienen más misión que esa*. Por tanto la idea sería que las operaciones accedieran a las entidades a través del VSP ya que es el mecanismo de comunicación con mayores ventajas:

- Se accede de la misma manera a las propiedades de cualquier entidad.
- Mantiene la identidad de las entidades.
- No necesita de clases adicionales adaptadoras que recarguen el modelo.

Y el mecanismo de conexión entre operaciones (validaciones y reacciones) será el otro mecanismo que cumple los principios de Conformidad y Adyacencia: el patrón Adapter.

- Está orientado a la invocación de métodos.
- Permite transformaciones más elaboradas que la mera traducción léxica (por ejemplo de parámetros)

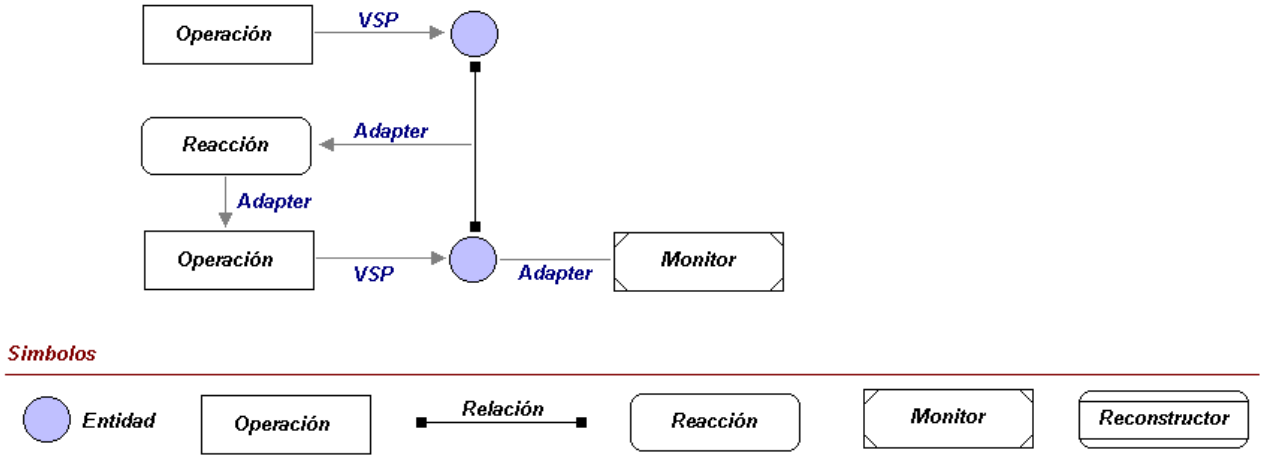

Imagen 6.2-2. Mecanismos de Comunicación entre los elementos de RDM.

[12] En ellos se basan otras arquitecturas que pretenden comunicar objetos independientemente de su implementación como por ejemplo CORBA [Brose01], DCOM [Brown98], RMI [Grosso01] o los Enterprise Javabeans [Roman01], etc.

Una vez establecida cómo se realizará los elementos de la arquitectura en el siguiente capítulo ya se puede entrar a detallar la implementación de la arquitectura y la forma de utilizarla para la construcción de aplicaciones.

7 El Framework RDM

En capítulos anteriores se han definido los elementos de la arquitectura y se ha determinado la forma de comunicarlos. En este capítulo finalmente se va a juntar todo ello para dar una implementación de la Arquitectura. Se presenta aquí un framework del mismo nombre que la arquitectura el cual aporta un lugar donde modelar los elementos del dominio sobre lenguajes orientados a objetos.

7.1 Implementación de los Elementos de RDM

7.1.1 Implementación de Entidades

El framework sigue la estructura para la creación de frameworks propuesta en [Gamma95] con el JHotDraw en la cual se separan los interfaces (sobre los que actuarán el resto de los elementos del framework) de las posibles implementaciones que pudieran tener. Lo que un framework tiene que ofrecer al menos es una implementación standard de cada uno de los interfaces para que de entrada éste sea funcional.

El primer tipo del framework es el interfaz *Entity*. Básicamente una entidad es la implementación de una variante del patrón *Variable State Pattern*. La implementación por defecto de dicho interfaz es la clase StandardEntity (la cual guarda sus atributos en una tabla hash). Sin embargo podrían incorporarse otras implementaciones de las entidades sin afectar al resto del framework (podrían implementarse versiones mas compactas, versiones remotas, etc).

En el siguiente ejemplo se crea una entidad denominada *persona* y sobre ella se muestra como añadir, consultar y borrar atributos:

```
Entity persona = new StandardEntity();

persona.add(“DNI”, 987654321);
System.out.println(persona.get(“DNI”));// se escribe 987654321

persona.remove(“DNI”);
System.out.println((persona.get(“DNI”));      // se escribe null

persona.add(“Nombre”, “Aquilino”);
System.out.println(persona.get(“Nombre”)); // se escribe Aquilino
```

Como se ha mostrado en el fragmento anterior se pueden añadir atributos a instancias individuales de manera dinámica en tiempo de ejecución. Sin embargo el resto de los objetos del mismo tipo no tienen por qué tener ese atributo ni tampoco hubo que modificar la clase del objeto *persona* para codificar los métodos de asignación y consulta del nuevo atributo.

Un caso en el que se presenta muy útil el hecho de poder añadir atributos dinámicamente es cuando se quiere representar en memoria un documento semiestructurado, como por ejemplo los codificados en XML, en los cuales no todas las entidades tienen las mismas etiquetas ni éstas se conocen con antelación.

Esta primera funcionalidad del interfaz *Entity* es la que presentan los lenguajes basados en prototipos ([Borning86], [Evins94], [Lieberman86]) en los cuales se pueden añadir atributos dinámicamente a cualquier objeto. La única diferencia con éstos y con el Patrón Variable State Pattern original es que el método para añadir atributos se ha separado en dos:

- El método *add* sirve para añadir un atributo nuevo. Si el atributo ya existe se produce una excepción.

- El método *set* sirve para modificar un atributo ya existente. Si no existe se produce una excepción.

El problema que aparentemente puede presentar esto es ¿qué ocurre si se quiere poner valor a un atributo y no se sabe si existe? Aunque hay un método *contains* que permite averiguarlo (e incluso un tercer método *put* que equivaldría a la combinación de ambos métodos) en la práctica se descubrió que esto no representaba un problema. Es decir, las operaciones saben cuando quieren añadir un atributo a una entidad y saben cuando quieren modificar alguno existente. No tienen problema a la hora de determinar qué método necesitan.

Sin embargo cuando todo se hacía a través de un único método *set* un problema real y que castigó frecuente la implementación era creer estar añadiendo un atributo cuando ya existía o creer estar modificando un atributo que en realidad no existía.

- Lo primero indicaba que las operaciones no se estaban ejecutando en el orden en el que se suponían (alguien ya había puesto ese atributo, lo cual solía ser un error).

- Lo segundo indicaba seguramente que se había tecleado mal el atributo, por lo que el atributo requerido no solo no era modificado sino que se creaba otro nuevo.

En cuanto se separaron las operaciones de *añadir* y de *modificar* en dos métodos aparecieron en las aplicaciones que usaban este framework *decenas* de errores en los que el nombre del atributo se había escrito incorrectamente y había pasado desapercibido en tiempo de ejecución. Actualmente todavía es habitual, al añadir alguna nueva operación, que el método *set* produzca una excepción por usar un atributo no existente al haber escrito mal el nombre, con lo cual el error se arregla en el momento.

7.1.2 Implementación de Relaciones

Más que añadir atributos dinámicamente la verdadera ventaja de utilizar una *Entity* es la capacidad para añadir relaciones. Se puede hacer que una entidad pueda asociarse en tiempo de ejecución con otros objetos a los que no conoce.

Supóngase como ejemplo el siguiente diagrama de objetos.

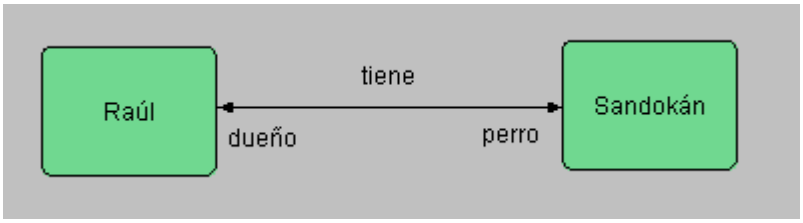

Imagen 7.1-1. Relación entre dos entidades con sus roles.

Esta relación se definiría de la siguiente forma:

```
Relation tiene = new StandardRelation("dueño", "tiene", "perro",
Cardinality.ONETOMANY);
```

Una vez creada el tipo de relación éste puede ser añadido a cualquier entidad. Por ejemplo supóngase el objeto *persona* y el objeto *sandokán* de la clase perro. A la persona se le añade una relación en la que tomará el rol de *dueño*. De la misma manera a sandokán se le dice que va a participar en una relación en la que tomará en rol de *perro*.

```
StandardEntity persona = new StandardEntity();
StandardEntity sandokán = new StandardEntity();

persona.add(tiene, “dueño”);
sandokán.add(tiene, “perro”);
```

Añadir una relación equivale a añadir como atributo el rol opuesto al que cumple el objeto en la relación. Por ejemplo después de ejecutar el fragmento siguiente lo que se obtiene es una entidad *persona* con dos miembros: *DNI* y *perro*.

```
Relation tiene = new StandardRelation("dueño", "tiene", "perro",
Cardinality.ONETOMANY);

persona.add(“DNI”, 987654321);
persona.add(tiene, “dueño”);
// Ahora la persona tiene dos miembros: DNI y perro
```

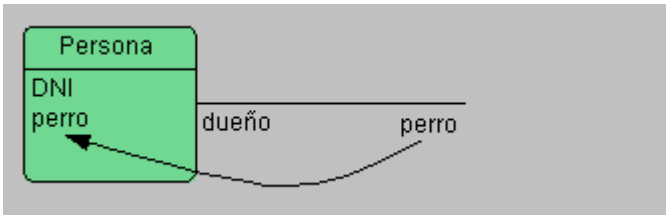

Imagen 7.1-2. Los roles opuestos de una relación se consideran atributos de la entidad.

Pero una cosa es añadir un tipo de relación a una entidad y otra es *relacionarla con otras entidades* mediante dicha relación. En el caso anterior lo que se ha dado al objeto persona es *la posibilidad* de tener perros. Pero en ese momento *todavía no tiene ninguno* (razón por lo que en la figura anterior no hay ninguna entidad al final de la línea; pero ya hay una línea en la que *pegar* entidades).

7.1.2.1 Establecimiento de Relaciones y su Sincronización

En el apartado anterior se mostró como añadir una relación a las entidades. Sin embargo todavía no tienen ninguna instancia de dicha relación, es decir, la entidad no ha sido relacionada con ninguna otra entidad (puede tener perros, pero todavía no tiene ninguno)[13].

En el primer capítulo se planteaba la cuestión de cuál debería ser el protocolo para establecer la relación entre dos objetos ¿Se asigna el perro al dueño o el dueño al perro? En esta arquitectura ambos *son equivalentes*.

Supóngase el siguiente caso:

```
Relation tiene = new StandardRelation("dueño", "tiene", "perro",
Cardinality.ONETOONE);

raul.add(tiene, “dueño”); // En esta relación raul será el “dueño”
sandokan.add(tiene, “perro”);    // En esta relación será el “perro”

raul.set(“perro”, sandokán);

raul.get(“perro”);  → sandokán
sandokán.get(“dueño”);    → raul
```

En la primera línea se crea la relación. En las dos siguientes se les añade dicha relación a dos entidades mediante el método *add*. A partir de ese momento dichas entidades podrán ser relacionadas con otras entidades mediante la relación “tiene”.

En la cuarta línea finalmente se relacionan las dos entidades. Da igual a cual de los dos extremos se le comunique la relación. Ambos se dan por enterados. Por tanto cuando en la última línea se le pregunta a sandokán quien es su dueño responde adecuadamente aunque a este no se le hubiera notificado nada anteriormente.

Pero la integridad debe mantenerse también al eliminar la relación:

```
raul.set(“perro”, sandokán); // equivale: sandokán.set(“dueño”, raul);

raul.get(“perro”);  → sandokán

sandokan.remove(“dueño”);
raul.get(“perro”);  → null
```

Cuando en la tercera línea a sandokán se le retira el dueño debe cumplirse que si en la última línea al dueño se le pide su perro responda que no tiene ninguno.

También hay que tener en cuenta los cambios en las relaciones de terceros que afecten a las entidades. Por ejemplo supóngase que se tiene la siguiente relación.

[13] Hay que diferenciar entre la *relación* (o también *tipo de relación*) y la *ocurrencia o instancia de la relación*. En el apartado anterior se creó la relación (por ejemplo los “dueños tienen perros”). En este apartado se crearán instancias de dicha relación (Pepe con Tintín, Juan con Milú, etc). Sin embargo en la práctica se denomina indistintamente *relación* a ambos conceptos y se distinguen por el contexto.

```
raul.set("perro", sandokán);
```

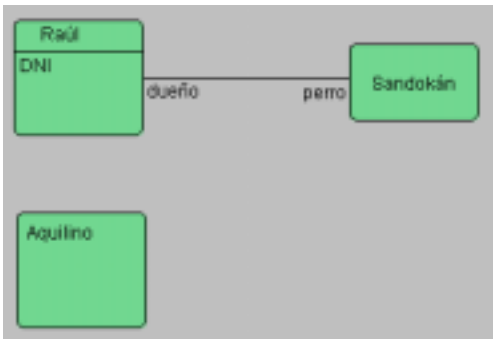

Imagen 7.1-3

Y ahora se hace

```
aquilino.set("perro", sandokán);
```

La situación obtenida sería:

```
raul.get("perro")   → null
aquilino.get("perro")     → sandokan
sandokán.get("dueño")     → aquilino
```

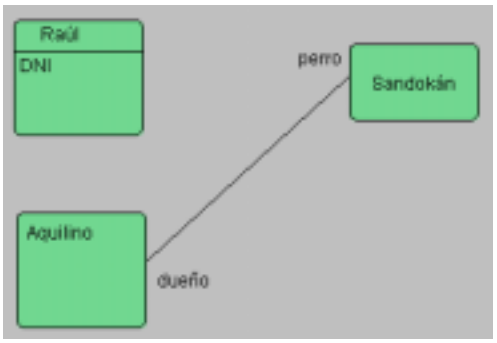

Imagen 7.1-4

No sólo sandokán ha quedado relacionado con Aquilino sino que además Raúl también se ha enterado y ha eliminado su relación con su mascota ya que ésta solo puede tener un amo[14].

Por tanto si se diera la siguiente situación inicial:

[14] Se podrá limitar que una relación pueda ser elimina por otra usando otro elemento del framework que se mostrará posteriormente: los monitores.

```
raul.set(“perro”, sandokán);
aquilino.set(“perro”, lassie);
```

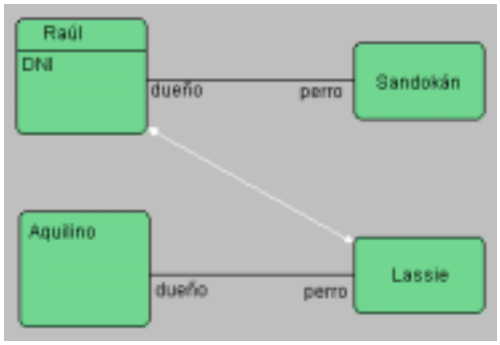

Imagen 7.1-5

Y a continuación se hace:

```
raul.set(“perro”, lassie);
```

El diagrama de objetos quedaría, por ser las relaciones de *uno a uno*, de la siguiente forma:

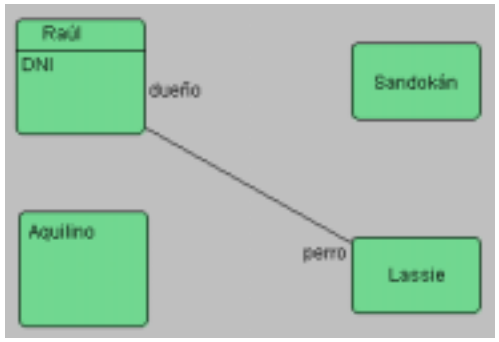

Imagen 7.1-6

```
raul.get(“perro”)   → lassie
lassie.get(“dueño”) → raul
aquilino.get(“perro”)     → null
sandokán.get(“dueño”)     → null
```

Si la relación hubiera sido de *uno a muchos* la situación hubiera quedado así:

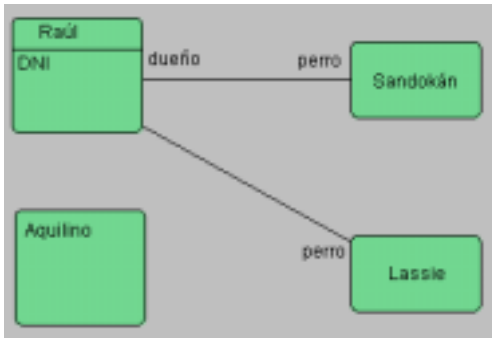

Imagen 7.1-7

```
raul.get("perro")   → {Sandokán, lassie}
lassie.get("dueño") → raul
aquilino.get("perro")     → {}
sandokán.get("dueño")     → raul
```

Y si hubiera sido de *muchos a uno*:

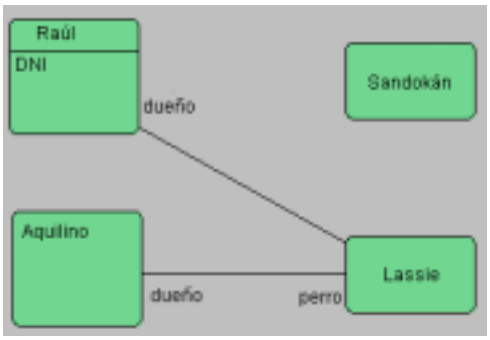

Imagen 7.1-8

```
raul.get("perro")   → lassie
lassie.get("dueño") → {raul, Aquilino}
aquilino.get("perro")     → null
sandokán.get("dueño")     → {}
```

Y finalmente si hubiera sido de *muchos a muchos*:

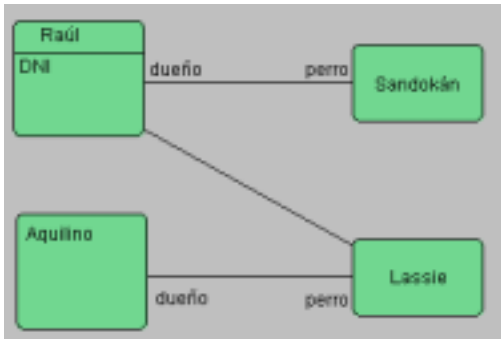

Imagen 7.1-9

```
raul.get("perro")   → {Sandokán, lassie}
lassie.get("dueño") → {raul, Aquilino}
aquilino.get("perro")     → {lassie}
sandokán.get("dueño")     → {raul}
```

En definitiva son las propias relaciones y no los objetos los que mantienen la integridad respecto a la cardinalidad. Esto es lógico ya que los objetos *no conocen* a las relaciones en las que se encuentran participando.

7.1.2.2 Uniformidad de Acceso

Se ha visto inicialmente como añadir atributos y posteriormente como añadir relaciones a una entidad. Sin embargo en RDM no se debe diferenciar entre atributos y relaciones a la hora de acceder a ellos, por lo que se referirá a ambos como *miembros* de la entidad. Se accede a ellos de

igual manera siguiendo el Principio de Acceso Uniforme [Meyer97]. Se considerará entonces que las entidades tienen un conjunto de miembros que se dividen en:

- **Simples**. Son aquellos cuyo nombre está asociado a un único valor. Los miembros simples son tanto *los atributos como las relaciones* en las que el rol del otro extremo tiene una cardinalidad de *uno* (es decir, de *uno a uno* o de *muchos a uno*). Dado que el nombre del miembro (el rol) está asociado a un único objeto no se diferencia de los atributos. Un *atributo puede pasar a ser una relación de uno a uno y viceversa* sin tener que modificar la forma de acceder al miembro.

- **Compuestos**. Son aquellos cuyo nombre está asociado a un conjunto de valores. Serán los atributos cuyo valor sea una colección de objetos y las relaciones en las que el rol del otro extremo tiene una cardinalidad de *muchos* (es decir, de *uno a muchos* o de *muchos a muchos*)

La multiplicidad de un miembro se tiene en cuenta tanto en las asignaciones como en las consultas. A la hora de asignar valor a un miembro:

- Si es simple se asigna el nuevo valor sobrescribiendo el anterior.

- Si es compuesto el nuevo valor se añade al conjunto.

A la hora de consultar el valor de un miembro:

- Si es simple se devuelve el objeto directamente (recuérdese que también en el caso de las relaciones con cardinalidad *uno* en el otro extremo).

- Si es compuesto se devuelve una colección con los valores que tiene dicho miembro.

Como ejemplo de lo anterior se comenzará por formar el modelo de objetos de la figura mediante las siguientes sentencias:

```
Relation relationA = new StandardRelation("dueño", "tiene", "perro",
Cardinality.ONETOMANY);
Relation relationB = new StandardRelation("marido", "tiene", "esposa",
Cardinality.ONETOONE);

pepe.add(relation, “dueño”);
pepe.add(relation, “marido”);
pepe.add(“DNI”, 987654321);

pepe.set(“perro”, sandokán);
pepe.set(“perro”, pulgoso);
pepe.set(“esposa”, pepa);
```

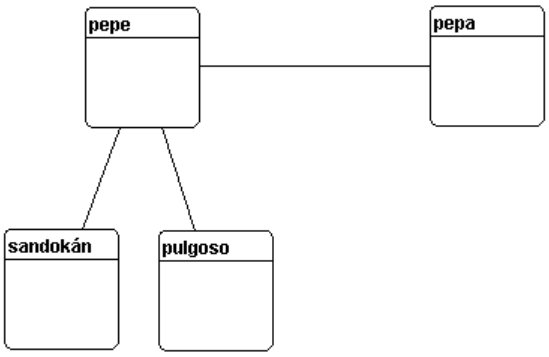

Imagen 7.1-10. Diagrama de objetos con las relaciones iniciales.

Estando el modelo en la situación de la figura anterior a continuación supóngase que se ejecutan las siguientes sentencias adicionales:

```
pepe.get("DNI"); → 987654321
pepe.get("perro"); → {sandokan, pulgoso}
pepe.get("esposa"); → pepa // Un objeto directamente, no una colección
con un objeto

pepe.remove("perro", sandokán);  → Solo sandokán
pepe.get("perro"); → {sandokan} // Una colección con un objeto
```

En la primera se pide el valor del atributo DNI y, como miembro simple que es, se devuelve un solo valor. En la siguiente sentencia se pide el valor de un miembro compuesto (la relación con el rol perro) por lo que se devuelve un conjunto. En la tercera sentencia, aunque corresponde a una relación, sin embargo se devuelve un objeto en vez de un conjunto formado por un solo objeto. Esto es así porque, como se comentó anteriormente, se trata a las relaciones con una cardinalidad de *uno* como a un atributo.

Hay dos versiones sobrecargadas del método de borrado (*remove*). En la primera se indica solamente el nombre del atributo a borrar y por tanto se borran *todos los valores* que tenga (sea simple o compuesto). En la segunda versión de este método se indica además del atributo el valor que se quiere eliminar.

- En el caso de los miembros compuestos se eliminará ese valor del conjunto quedando el resto de los valores. Si el miembro no tiene ese valor se producirá una excepción.

- En el caso de los miembros simples, dado que solo tienen un valor, el valor a borrar debe coincidir con el valor actual del miembro. En caso contrario se produce una excepción.

```
persona.remove("DNI");
persona.remove("perro", sandokán);      → Solo sandokán (si tiene ese
valor en el conjunto)
persona.remove("perro"); → Todos los perros
persona.remove("DNI", 123456789);       → lo borra si tiene ese valor.
```

Este es un ejemplo de cómo se tratan todos los miembros de manera homogénea independientemente de que vengan de un atributo o una relación. Esto es fundamental ya que en

las aplicaciones que han utilizado este framework un *cambio habitual del dominio* era que un atributo pasara a ser una relación porque se descubría que debía ser compartido por más entidades. Esto no suponía ningún cambio en las operaciones ni en el resto del modelo.

Nótese que a cualquier objeto se le pueden añadir relaciones *dinámicamente* sin tener que recompilar la clase (codificar métodos de acceso a la nueva relación). Las relaciones se asocian a objetos, no a clases. Asociar una relación a una clase se conseguiría asociando a todo objeto de dicha clase la relación en el momento de su creación (mediante el constructor o bien mediante el patrón Abstract Factory [Gamma94]).

7.1.2.3 Cardinalidad y Roles en las Relaciones

El framework RDM permite cualquier tipo de relación en cuanto a roles, entidades relacionadas y cardinalidades.

Permite autorelaciones (relaciones de un objeto consigo mismo):

```
Relation tiene = new StandardRelation("cliente", "tiene", "abogado",
Cardinality.ONETOMANY);
raul.add(tiene, “abogado”);
raul.add(tiene, “cliente”);

raul.set(“cliente”, raul);
```

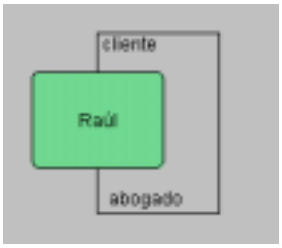

Imagen 7.1-11. Autorelación

En este caso hubiera sido lo mismo haber establecido la relación con esta otra sentencia:

```
raul.set("abogado", raul);
```

Eso si, solo se debe poner una de las dos ya que si no se produciría una excepción de relación duplicada (es la misma relación).

También se permiten relaciones con el mismo rol en ambos extremos:

```
Relation tiene = new StandardRelation("amigo", "tiene", "amigo",
Cardinality.ONETOMANY);
pepe.add(tiene, “amigo”);
juan.add(tiene, “amigo”);

pepe.set(“amigo”, juan);
```

Imagen 7.1-12. Relación con mismo rol en ambos extremos.

Y por último también se permiten autorelaciones (mismo objeto) y además con mismos roles:

```
Relation tiene = new StandardRelation("amigo", "tiene", "amigo",
Cardinality.ONETOMANY);
raul.add(tiene, “amigo”);

raul.set("amigo", raul);
```

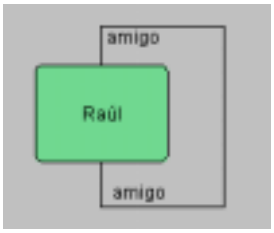

Imagen 7.1-13. Autorelación con mismos roles en ambos extremos.

Y por supuesto todos estos casos siempre manteniendo la integridad de las relaciones en ambos extremos.

7.1.2.4 Metadatos en las Entidades

El interfaz Entity permite interrogar por los atributos y por las *relaciones* en las que participa la entidad en tiempo de ejecución.

Esto se utilizará generalmente para permitir la introspección de las entidades y así poder hacer operaciones genéricas de lectura y escritura de modelos. Por ejemplo con el framework RDM viene un conversor de modelos RDM a XML y viceversa utilizando dicha posibilidad.

```
interface Entity
{
	...

	public List getMembers();
	public boolean contains(String member);
	public boolean isSingleValue(String member);
	public boolean isAttribute(String member);

	public boolean hasRelation(Relation r, String role);
	public List getRelatedRoles();
	public List getRoles();
	public Relation getRelationTo(String relatedRole);
}
```

7.1.2.5 Navegación por el Modelo

Hasta el momento se tiene una forma de implementar los objetos del modelo en la cual se pueden añadir atributos, se pueden añadir relaciones y se puede consultar y modificar ambos a través de los mismos métodos sin necesidad de saber su naturaleza. Esto quiere decir que se puede navegar a través de distintas clases *sin necesidad de conocerlas*. Supóngase el siguiente diagrama.

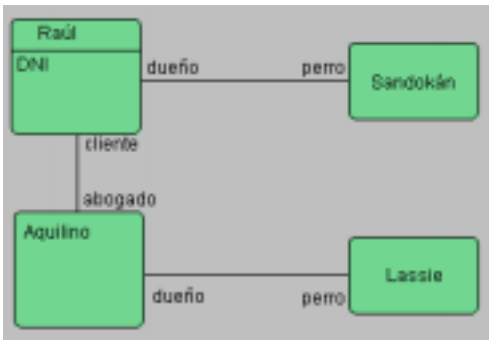

Imagen 7.1-14

Supóngase que ahora sandokán quiere saber como se llama el perro del abogado de su dueño. La forma habitual de hacerlo sería tener que mirar cada una de las clases para saber con qué métodos ir obteniendo cada uno de los objetos intermedios del recorrido

```
lassie = sandokan.getDueño().getAbogado().getPerro();
```

Sin embargo con RDM no se necesita conocer las clases para que dado sandokán se pueda acceder a los objetos buscados. Basta con la información que está en el diagrama. Sería:

```
sandokán.getPath("dueño.abogado.perro"); // {Lassie}
```

La sentencia anterior devolvería una lista con un solo elemento. Si hubiera habido varios perros (o varios abogados) se hubiera devuelto un conjunto de todos los perros que se encuentren en cualquier ruta con ese nombre partiendo de sandokán.

Se tiene por tanto un lenguaje de navegación que permite acceder a los objetos independientemente de su clase concreta y de sus métodos de acceso específicos. Esta implementación ha permitido ahorrar varios cientos de líneas de código y no tener que modificar

rutas aunque cambiara la implementación de las clases que recorrían. Además al ser las rutas de tipo String pueden pasarse como parámetro o sustituirse dinámicamente.

7.1.3 Implementación de Operaciones

Una vez visto cómo se implementan en el framework las entidades y sus relaciones en este apartado se mostrará como son manipuladas por las operaciones.

7.1.3.1 Conexión de Operaciones y Entidades

La forma de implementar ahora las operaciones es directa. Una operación será una clase normal y corriente que tendrá un método en el que estará implementada la operación el cual admitirá como parámetro objetos de tipo *Entidad*.

```
class OperaciónEjemplo
{
	void ejecutar(Entidad entidad)
	{
		entidad.get(...);
		entidad.set(...);

		otraEntidadRelacionada = entidad.get(“dueño”);
		otraEntidadRelacionada.set(...);
	}
}
```

La operación recibe una entidad y se limita a manipular los miembros que necesite mediante *get* y *set* sin importar la clase de la entidad; si son atributos o relaciones y mucho menos si la entidad tiene más miembros que ella no necesita.

Como consecuencia dicha operación se podrá aplicar con cualquier modelo que tenga los miembros que ésta requiera. En el siguiente capítulos se mostrarán ejemplos de la aplicación de la misma operación a varios modelos distintos a los cuales no conoce. Por tanto se consigue así la independencia y la reutilización de operaciones y entidades.

El interfaz Entity, para las clases cuya tarea es almacenar información, sería el equivalente a la clase Object de Java. Al igual que Object se puede utilizar Entity para crear nuevas clases; pero, a diferencia de Object, Entity ya tiene suficientes métodos como para que sea suficientemente manipulable (gestión de atributos, relaciones y metadatos) sin que las operaciones necesiten saber más del tipo concreto del objeto.

Sin embargo no se está proponiendo utilizar únicamente el tipo *Entity* a la hora de declarar parámetros ya que eso sería algo parecido a trabajar en un lenguaje sin tipos. Se pueden seguir utilizando tipos concretos a la hora de pasar parámetros. Lo que se ofrece es la posibilidad de elegir entre mayor control de tipos o mayor flexibilidad. Se tienen ambas posibilidades de tal manera que se pueda elegir *dentro del mismo lenguaje* la opción más apropiada *para cada operación*.

7.1.3.2 Resolución de la Desadaptación de Nombres

RDM permite *sinónimos* a la hora de referirse a los miembros. De esta manera distintas operaciones pueden referirse al mismo miembro utilizando lexemas distintos. La entidad sabrá que todos ellos, con distinto nombre, se están refiriendo a lo mismo.

```
objeto.addAlias("identificador", "nombre");
```

Por ejemplo supóngase que se quiere aplicar la operación *calcula* sobre el objeto *dato*. La operación requiere que el objeto tenga un atributo *total*. Sin embargo el objeto tiene dicho valor asociado al nombre *cantidad*. Bastaría un código como el siguiente:

```
void aplicarCalcula(Ob)
{
      dato.addAlias("cantidad", "total");
      calcula(dato);
      dato.removeAlias("cantidad", "total");        // Esto es opcional
}
```

Pero ¿qué pasa si ese nombre ya está asignado a otro miembro con una semántica distinta? Supóngase que en el caso anterior el miembro *total* ya existe para otro miembro. No se podría añadir *total* como sinónimo de *cantidad*. Entonces lo que habría que hacer es usar el Patrón Adapter de tal manera que las peticiones de la operación pasen por él y realice la traducción. Cada vez que se le pida el *total* al adapter éste devolverá el valor del miembro *cantidad*.

```
void aplicarOp(Ob)
{
      adapter = new TotalAdapter(Ob);
      Op(adapter); // Se aplica la operación sobre Ob usando total
}
```

El patrón *Adapter* siempre puede aplicarse, pero en la inmensa mayoría de los casos este problema no se presenta. Es muy difícil que ocurra una coincidencia como la del caso anterior. Usar el adaptador es una solución más engorrosa ya que hay que crear una nueva clase específica para este detalle de implementación. Pero si *no se tiene* ese problema no tiene sentido adoptar dicha solución para el resto de los casos que tienen otra más sencilla.

Por tanto en general se usará VSP con sinónimos. Pero sabiendo que si se presenta una situación extrema como la anterior siempre hay un último recurso que sería usar un adaptador.

7.1.4 Implementación de Reglas de Negocio

Una vez vista la forma de implementar las entidades, las relaciones y las operaciones quedan todavía por mostrar las reglas de negocio, las cuales se dividen en Monitores y Reacciones.

7.1.4.1 Notificaciones de Cambios en el Modelo

Debido a la aplicación del *Principio de Externalización* los objetos del modelo tienen la responsabilidad de lanzar eventos cada vez que se cambie su estado para permitir su interacción con otros elementos del dominio.

La codificación en los lenguajes orientados a objetos de estos eventos suele hacerse en función del objeto que lo lanza y del tipo de modificación que haya sufrido su estado. Esto da lugar a código reiterativo de creación de interfaces y de gestión de suscriptores (registro, borrado y notificación) como se vio en el capítulo 2 "Problemas de las Tecnologías Orientadas a Objetos".

Con RDM cada vez que se modifica cualquier miembro (por tanto atributo o relación) se lanza automáticamente un evento a los *EntityListeners* subscritos sin necesidad de implementar nada de lo anterior. Concretamente se lanza un evento antes de la modificación y otro después de que ya se haya realizado. Por ejemplo cuando se modifica un atributo con el método *set* este lanza un evento *memberSetRequest* (solicitud de modificación de atributo) y después de que haya sido modificado se lanza un evento *memberSet* (el miembro *ha sido* modificado).

Supóngase la siguiente situación en la que las personas y los perros tienen una relación de uno a uno:

```
class Tracer implements EntityListener
{
	public void memberSet(EntityEvent e)   // Da igual que sea
atributo o relación
	{
		System.out.println("Entidad: " + e.getEntity());
		System.out.println("Se ha añadido el miembro: " +
e.getMember());
	}

	public void memberRemoved(EntityEvent e);
	{
		System.out.println("Entidad: " + e.getEntity());
		System.out.println("Se ha borrado el miembro: " +
e.getMember());
	}

	public void memberSetRequest(EntityEvent e)   {} // Estos eventos
se lanzan antes de los cambios
	public void memberRemoveRequest(EntityEvent e) {} // No se
necesitan en este ejemplo
}

...

Tracer tracer = new Tracer();
dueño.addEntityListener(tracer);
sandokan.addEntityListener(tracer);
lassie.addEntityListener(tracer);

dueño.set("DNI", 123456789); → evento con origen en dueño (dni changed)
dueño.set("perro", sandokán); → evento con origen en dueño y otro en
sandokan
dueño.set("perro", lassie);      → evento en dueño, en lassie y en
sandokan (se ha quedado sin dueño)
```

Los eventos emitidos son los siguientes:

- En el primer *set* realizado sobre el objeto *dueño* todos sus suscriptores recibirán la notificación de que se ha modificado su DNI al valor 123456789.

- En la siguiente sentencia se emite la notificación de que el miembro *perro* ha sido modificado en la entidad *dueño* y los suscriptores de sandokán también recibirán un evento notificando que éste tiene ahora dueño.

- Finalmente en el tercer caso, además de los suscriptores de *dueño* y *lassie*, también reciben un evento los suscriptores de *sandokán*. Aunque dicha entidad no aparece mencionada en la sentencia debe lanzar el evento de que su *dueño* ha cambiado. Al ser la relación de uno a uno ya no está asociado a la entidad *dueño* y por tanto dicho miembro le ha sido eliminado.

Los suscriptores pueden subscribirse tanto a las entidades *como a las relaciones*, es decir, un suscriptor puede ser de dos tipos

- Si quiere ser notificado de los cambios de estado de una entidad concreta deberá implementar *EntityListener*.

- Si quiere ser informado cuando se produzca cualquier cambio en un tipo de relación independientemente de los objetos afectados deberá implementar *RelationListener*.

A continuación se muestra un ejemplo de cómo controlar los cambios en una relación.

```
class CounterRelationListener implements RelationListener
{
	int added;
	int removed;

	public void relationAdded(RelationEvent e) {
		added++;
	}

	public void relationRemoved(RelationEvent e) {
		removed++;
	}
}

...

StandardRelation tiene = new StandardRelation("dueño", "tiene", "perro",
Cardinality.ONETOONE);
tiene.addRelationListener(new CounterRelationListener());

dueño.set(“perro”, sandokan);          // Evento
```

En el fragmento anterior el suscriptor *CounterRelationListener* será notificado cada vez que a cualquier dueño se le asigne cualquier perro.

Como se dijo anteriormente las notificaciones de cambio de estado se utilizan fundamentalmente para permitir la implementación de dos elementos del dominio:

- Monitores: cuando el modelo necesite limitar ciertas operaciones en función de las reglas del dominio.

- Extensión de operaciones: una operación podrá extenderse mediante *reacciones* para que ésta actúe adecuadamente sobre nuevas entidades que no conocía originalmente.

7.1.4.2 Implementación de Monitores

Supóngase que solo se quisiera permitir que un dueño tuviera tres perros como máximo. La forma habitual de codificar esta restricción sin usar RDM sería modificando el método *addPerro(Perro)* de la clase *Dueño* para que se controle si ya se tiene tres perros.

```
public class Dueño
{
  private Vector perros = new Vector();

  public void addPerro(Perro p) {
       if (perros.size() < 3)
              perros.add(p);
  }
       ...
}
```

Pero al hacerlo de esta manera esta clase no podría reutilizarse en otra aplicación que no tuviera dicha restricción. Y de la misma manera también habría que modificarla si se quisieran añadir nuevas restricciones.

Con RDM cuando un suscriptor recibe la notificación de que un miembro de una entidad ha sido modificado tiene la opción de rechazar dicha operación mediante el lanzamiento de una *ValidationException*. En ese caso la entidad recibe la excepción y se deshace el cambio volviendo el miembro a su valor anterior.

Por tanto la forma de incluir la regla con RDM sería registrándola como suscriptor de la relación. Cada vez que se asigne un perro a un dueño la validación tendrá la oportunidad de comprobar cuantos perros tiene éste y si ya son tres rechazará la operación.

```
public class Max3DogsValidation implements RelationListener
{

  public void relationAdded(RelationEvent e) {
       Entity dueño = e.getEntity("dueño");
       if (dueño.getList("perro").size() > 3)
              throw new ValidationException("Solo tres perros por
dueño");
  }

  public void relationRemoved(RelationEvent e) {
  }
       ...
}
```

Y esta sería la manera de añadir los invariantes que debe cumplir el modelo del dominio. Se implementan como suscriptores que monitorizan la parte del modelo que tengan encomendada.

```
StandardRelation tiene = new StandardRelation("dueño", "tiene", "perro",
Cardinality.ONETOONE);
tiene.addRelationListener(new Max3DogsValidation());

dueño.set("perro", sandokan);    // Rechazado si ya tienen tres perros
```

De todas formas en el framework vienen ya implementados los monitores más comunes independientes del dominio, los cuales se mostrarán brévemente a continuación. De esta manera

pueden ser reutilizados directamente de manera que el programador se pueda centrar en las reglas específicas de su dominio.

7.1.4.2.1 AutoRelationValidator

El framework permite relaciones de un objeto consigo mismo. Por ejemplo una persona puede ser su propio abogado. Sin embargo en otras relaciones no tiene sentido: una persona no puede ser su padre. En este tipo de relaciones, en vez de poner un parámetro en la relación que indique si se deben permitir autorelaciones[15], lo coherente es permitirlas siempre y que se añada un *AutoRelationValidator* en aquellas relaciones en las que se quiera impedir.

```
public class AutoRelationValidator implements RelationRequestListener
{
  public void addRequest(RelationEvent e) {
       if (e.getEntityA() == e.getEntityB())
             throw new ValidationException("Auto relations not
allowed");
  }
}
```

7.1.4.2.2 SubRelationValidator

El *SubRelationValidator* es un monitor que exige que los objetos con los que se relacione una entidad mediante una relación R1 deben ser un subconjunto de los que se relacionan con esa misma entidad mediante otra relación R2.

Por ejemplo supóngase que un procedimiento tiene una relación "actividades" que comprende a todas las actividades de dicho procedimiento (esta sería la relación R2). Por otro lado ese mismo procedimiento tiene una relación llamada "primera" que indica por cual de esas actividades debe comenzar la ejecución (lo que sería la relación R1). Está claro que dicha actividad debe de ser una de las que pertenecen al procedimiento mediante la relación "actividades". Si se intentara asignar al procedimiento una actividad inicial que no fuera una de las suyas la operación debería ser rechazada.

Este monitor se añade en la relación R1 y tiene como parámetro el nombre de la relación R2. Cada vez que se añada una relación en R1 consulta R2 para comprobar el cumplimento de la regla.

Imagen 7.1-15. Relación "tiene" y la subrelación "empieza por"

[15] En la primera versión de RDM el hecho de que se permitieran autorelaciones era un parámetro del constructor de la relación. Pero por esa regla de tres tendría que conocer otras limitaciones de la relación. Lo lógico es extraerlo a una validación como cualquier otra.

7.1.4.2.3 ClassValidator

El framework RDM permite relacionar entidades de cualquier clase. Sin embargo en algunos casos, por razones de depuración fundamentalmente, quiere controlarse que los objetos que cumplan cada rol sean de una determinada clase. Este monitor rechaza cualquier relación que no sea entre objetos de las clases adecuadas.

```
public class ClassValidator implements RelationRequestListener
{
  private Class classOfRoleA, classOfRoleB;

  public ClassValidator(Class classOfRoleA, Class classOfRoleB) {
    this.classOfRoleA = classOfRoleA;
    this.classOfRoleB = classOfRoleB;
  }

  public void addRequest(RelationEvent e) {
    if (!classOfRoleA.isInstance(e.getEntity(0)))
      throw new ValidationException("Entity should be an instance of
class "+classOfRoleA.getName());

    if (!classOfRoleB.isInstance(e.getEntity(1)))
      throw new ValidationException("Entity entity should be an instance
of class "+classOfRoleB.getName());
  }

  public void removeRequest(RelationEvent e) {
  }

}
```

7.1.4.2.4 Otros Monitores Estándar

A una entidad se la puede tanto añadir miembros como quitárselos. El hecho de añadir nuevos miembros no suele tener peligro (sencillamente se ignoran en las operaciones existentes y ya serán usados por alguna nueva operación). Sin embargo puede que se desee impedir que algún miembro fundamental para el estado del objeto sea eliminado. El *FixedMemberValidator* impide que se elimine de la entidad el miembro que se le pase como parámetro en el constructor.

```
public class FixedMemberValidator implements EntityRequestListener
{
	private List members;

	public FixedMember(Collection members) {
		this.members = new ArrayList(members);
	}

	public void removeRequest(EntityEvent e) {
		if (members.contains(e.getMember()))
			throw new ValidationException("El miembro no puede
ser eliminado");
	}

}
```

Muy parecido a éste último es el *ReadOnlyValidator*. Este validator no solo no permite la eliminación sino tampoco la modificación del atributo. De esta manera se consiguen los

habituales miembros de solo lectura sin necesidad de que la clase *Entidad haya tenido que contemplar* explícitamente esta característica.

Finalmente el último monitor estándar es el *CardinalityValidator*. Permite establecer el límite superior de la cardinalidad de una relación de tipo *muchos* ya que por defecto esta es ilimitada. Cada vez que se intenta establecer una relación comprueba si los objetos ya tienen el número de relaciones máximo (el cual se le especifica en su constructor) y en ese caso rechaza la operación. El ejemplo *Max3DogsValidation* del apartado 7.1.4.2 en realidad no hubiera sido necesario implementarlo ya que la clase *CardinalityValidator* hubiera realizado la misma función si en el constructor se la hubiera pasado un tres como límite de relaciones permitidas.

7.1.4.2.5 Resumen Monitores

En definitiva el extraer las validaciones de los objetos produce dos consecuencias inmediatas:

- La reutilización. Ya no habrá que teclear código para implementar una y otra vez estas validaciones tan comunes. El diseñador solo tendrá que encargarse de averiguar aquellas que sean específicas de su dominio. Una vez hecho esto la forma de codificarlas y de insertarlas en el modelo es inmediata al igual que el eliminarlas.

- La reutilización de las clases del modelo en otros dominios con otras reglas. Así una clase no tendrá que ser recodificada en otra aplicación porque incluya una regla no aplicable en el nuevo dominio o bien no se le puedan aplicar nuevas reglas.

Este es un paso más para maximizar las posibilidades de reutilización de un modelo. Antes no se podía utilizar un objeto en un nuevo dominio porque hacían falta o sobraban relaciones o reglas. Ahora se pueden añadir o quitar ambas.

7.1.4.3 Descentralización de Operaciones mediante Reacciones

Anteriormente se comentó que las notificaciones de cambio de estado se utilizaban fundamentalmente para dos labores. La primera de ellas, la validación mediante monitores, ha sido descrita en el apartado anterior por lo que ahora se mostrará el uso de las notificaciones para la activación de reacciones.

Un suscriptor, ante la recepción de una notificación de cambio de una entidad, puede a su vez modificar el estado de otras entidades las cuales a su vez emitirán nuevos eventos produciéndose así una sucesión de eventos y *reacciones* encadenadas. Esto puede ser usado para *enganchar* nueva funcionalidad a las operaciones existentes y poder así extenderlas para que realicen nuevas acciones sobre el modelo sin tener que modificarlas.

```
// Esta reacción hace que cuando se borre la relación sobre la que esté
// aplicada la reacción (a la cual esté suscrita) encadene otra
// operación adicional si se cumple una determinada condición

public class ReactionExample implements RelationRequestListener
{

  public void addRequest(RelationEvent e) { // Aquí no hacer nada
  }

  public void removeRequest(RelationEvent e) {
       if (<condición>)
             ejecutarOperaciónB();
  }
}
```

Como ejemplo del uso de reacciones se utilizará la habitual operación de *borrado* de objetos. Supóngase el caso de un editor de procedimientos en el que se ha construido un procedimiento con tres actividades (A1, A2 y A3). Cada actividad puede tener una o más salidas. En este caso la actividad A1 tiene dos salidas y las demás actividades solamente una. El otro elemento que aparece en color rojo es un terminador el cual equivale a las salidas del procedimiento.

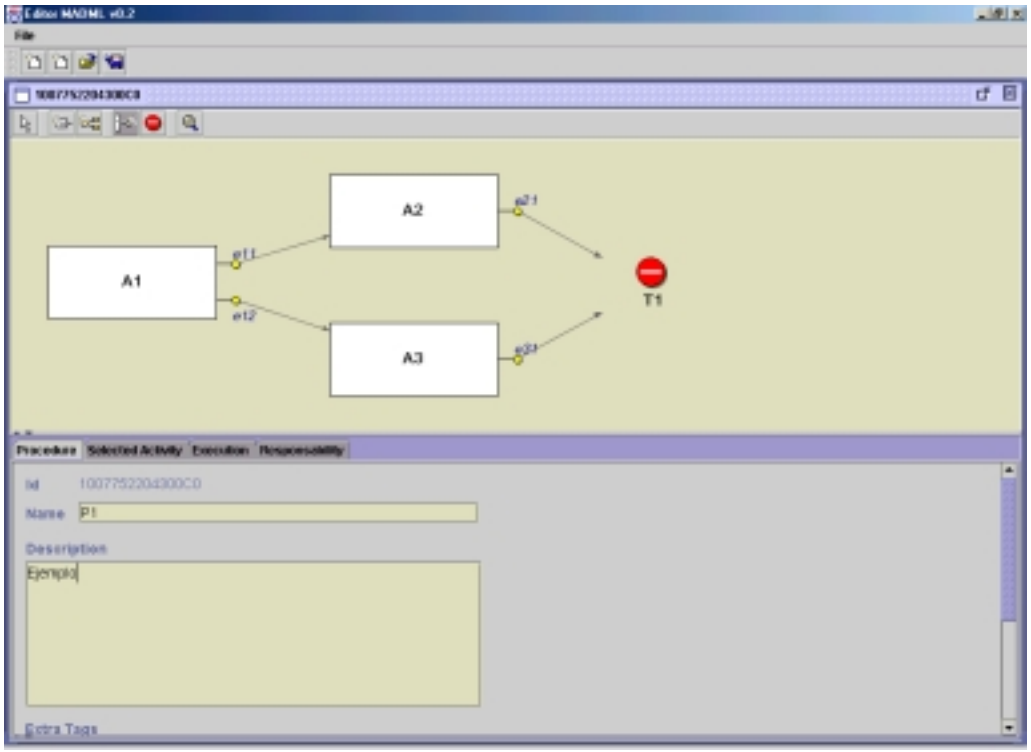

Imagen 7.1-16. Vista gráfica del modelo.

Ese procedimiento internamente se representa con el siguiente modelo de objetos:

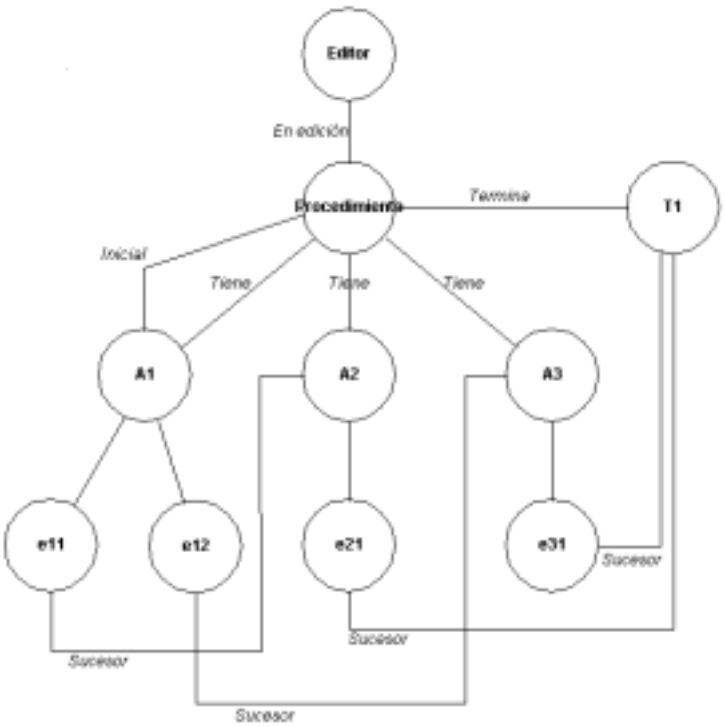

Imagen 7.1-17. Diagrama de objetos del modelo correspondiente a la vista anterior.

A la hora de implementar la operación de borrado para una *salida* (por ejemplo e11) habrá que seguir la siguiente secuencia de pasos:

- Eliminar las relaciones con sus sucesores (en este caso la que tiene con A2)
- Eliminar la relación con su actividad (la que tiene con A1)

En cambio si lo que se quiere borrar es una *actividad* (por ejemplo A1) habrá que hacer lo siguiente:

- Eliminar las dos relaciones que tiene con su procedimiento P1 (*Tiene* e *Inicial*)
- Eliminar las dos relaciones con sus salidas (e11 y e12).
- Debe borrar los propios objetos e11 y e12. Las salidas no tienen sentido sin su actividad por lo que al borrar una entidad de tipo actividad, además de sus relaciones, hay que borrar sus salidas. Por tanto recursivamente también hay que aplicar los pasos enumerados anteriormente para el caso de borrar una salida. Sin embargo nótese que no hay que borrar el procedimiento. Es decir, no se trata de que cuando se borre un objeto haya que borrar todos los demás relacionados. *Depende* del tipo de objeto.

Y para borrar el *procedimiento* la secuencia de pasos sería:

- Eliminar las relaciones que tiene con el editor, con su terminador T1 y con sus tres actividades.

- Debe borrar también los objetos T1, A1, A2 y A3 y recursivamente aplicar los pasos enumerados anteriormente para cada tipo de objeto.

El problema ya no es solo que haya que hacer una operación de borrado por cada entidad. El problema es que *por cada nueva relación o entidad* que se añada al modelo *habrá que modificar* todas *las operaciones de borrado* para que actúen de manera adecuada ante el nuevo modelo. Por ejemplo al añadir la clase de los *terminadores* hay que modificar el borrado del procedimiento para que borre también sus terminadores. No basta con que se elimine la relación; un terminador no tiene sentido sin estar asociado a un procedimiento y *alguien tendrá que borrarlo* expresamente.

Nótese que todo este trabajo es única y exclusivamente para una de las operaciones más simples y habituales de una aplicación: *la eliminación* de objetos del modelo.

La raíz del problema, una vez más, es el excesivo conocimiento del modelo por parte de las operaciones. Las distintas implementaciones de la operación *borrado* (una para cada tipo de objeto) tienen que conocer:

- Qué objetos forman parte del modelo.

- Qué tipos de relaciones hay entre ellos y la semántica de cada una, de tal manera que tiene que saber si la eliminación de una relación tiene algún efecto sobre el objeto relacionado ya que unas veces habrá que borrarlo y otras no (o realizar alguna otra operación).

En RDM esto se soluciona con reacciones. Para ello se introduce la relación de *agregación*[16]. Una relación de agregación es una relación que se caracteriza por tener asociada una reación que borra las demás relaciones que tenga el objeto agregado. Para saber las relaciones que éste tiene utiliza los metadatos de la entidad.

```
Relation tiene = new Aggregation("activity", "r2", "exit",
Cardinality.ONETOMANY);
Activity actividad = new Activity();
Exit salida = new Exit();

actividad.add(tiene, “exit”);
salida.add(tiene, “activity”);

activity.set(“exit”, salida);
```

Excepto a la hora de la creación de la relación puede observarse que su uso es exactamente igual que el de cualquier otra relación. Se obtendría el mismo resultado creando una *StandardRelation* y añadiendo manualmente la reacción a dicha relación. La clase *Aggregation* es más cómoda ya que realiza los dos pasos en uno.

Ahora el diagrama de objetos quedaría de la siguiente manera: tendría la misma estructura pero con las reacciones en los lugares oportunos.

[16] Se entiende la agregación tal y como está definida en [Cook94]. Tal definición diferencia la agregación de las demás relaciones en el hecho de que un objeto agregado no podrá cambiar de agregador a lo largo de su vida (forma parte de éste) y no tendrá sentido de manera independiente (es decir, deberá borrarse junto con su agregador).

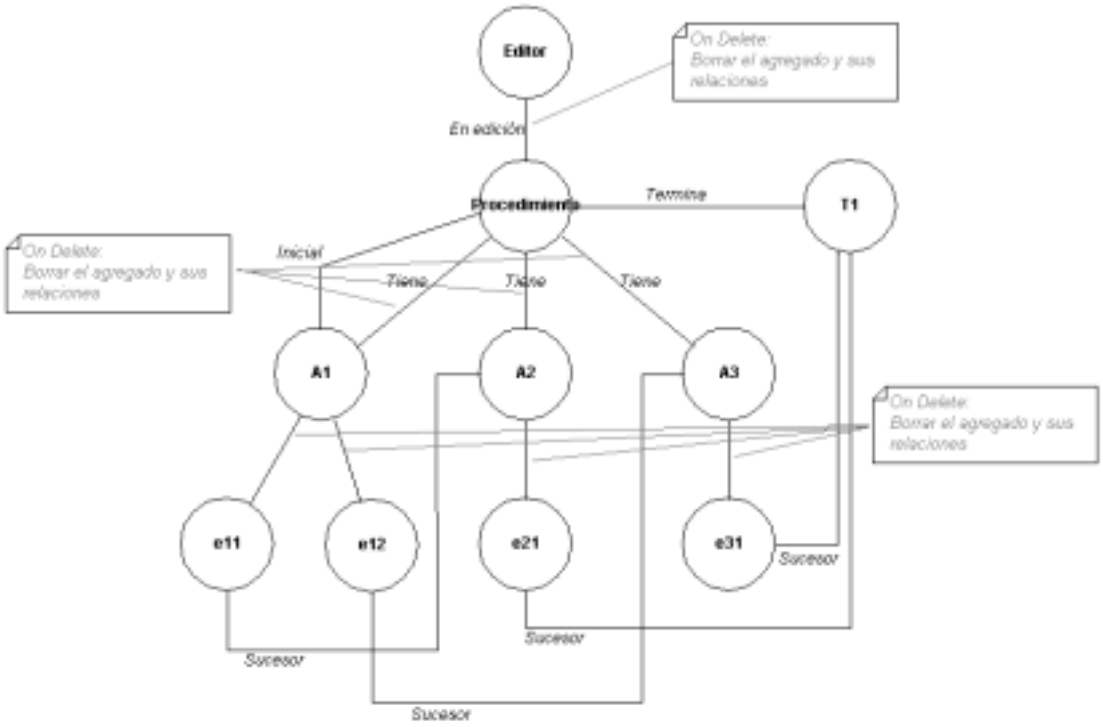

Imagen 7.1-18. Diagrama de objetos con sus reacciones.

Supóngase que se vuelve a intentar el borrado de la relación entre la actividad A1 y su salida e11. La reacción asociada a dicha relación eliminará todas las demás relaciones del objeto agregado e11 y luego lo borrará[17]. En este caso no se borrarían más objetos (la relación entre e11 y A2 no es de agregación, por lo que aunque esta se elimina no se elimina A2) y la operación finaliza correctamente.

Supóngase que ahora se elimina la relación "tiene" entre el procedimiento y la actividad A1. La reacción borra automáticamente todas las demás relaciones del objeto A1. Por tanto, al borrar las relaciones con las actividades, automáticamente se disparan también las reacciones que eliminan las salidas y sus relaciones con los sucesores.

[17] Esto último no será necesario en lenguajes con recolección de memoria.

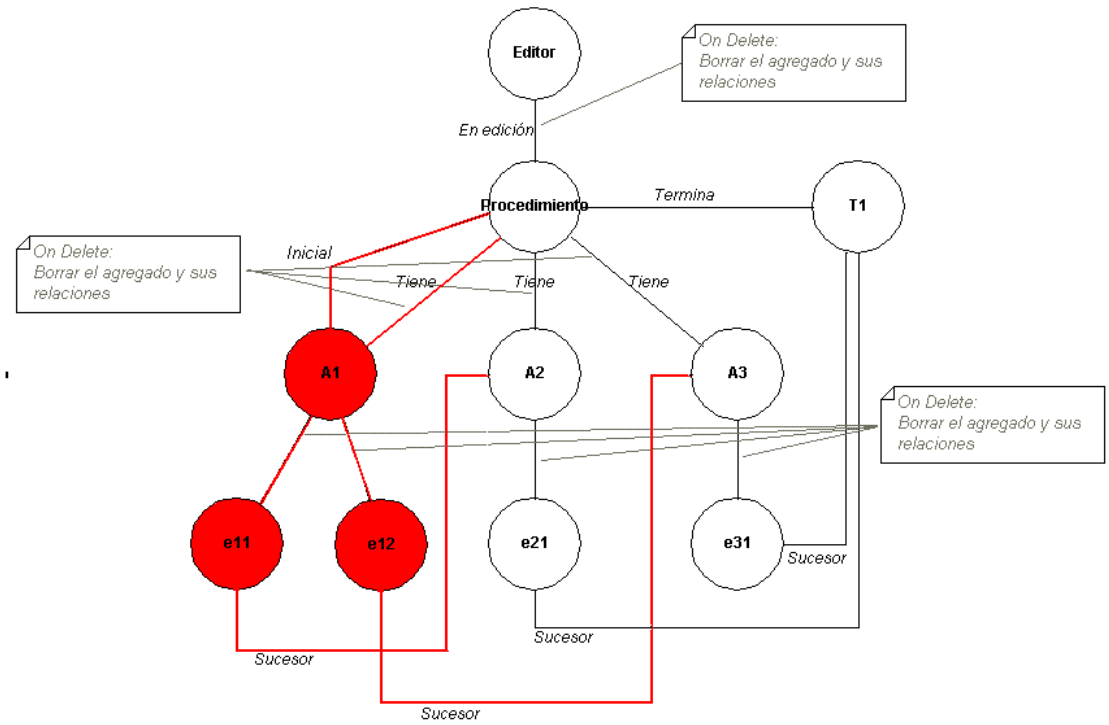

Imagen 7.1-19. Eliminación de la relación entre el procedimiento y A1.

Finalmente puede comprobarse como al eliminar la relación entre el editor y el procedimiento (por ejemplo al cerrar la ventana en la que se está mostrando éste) automáticamente se eliminan todos los objetos del modelo a base de reacciones. Pero el editor, cuando borró la relación con el procedimiento, no tenía que conocer todos los objetos y relaciones del modelo para borrarlas[18].

En definitiva la operación de borrado se ha convertido la simple eliminación de una relación: la que une al objeto con su agregador; su cordón umbilical con el modelo. Ahora toda entidad se borra igual; ya no hay que hacer una operación de borrado para cada entidad[19]. La operación que elimine la relación *ni siquiera tiene por qué saber que está borrando entidades*. Es la reacción la que decide que hay que borrar el objeto agregado si se elimina dicha relación (se le quita esa responsabilidad a la operación).

Y todo seguirá funcionando igual cuando se añadan mas tipos al modelo. Por ejemplo al añadir la nueva entidad *Responsable* asociada a las actividades también se borrarán éstos igualmente sin modificar la operación de borrado de actividades ni de procedimientos.

---

[18] En este caso lo que se obtiene es el equivalente a lo que en bases de datos se denomina *borrado en cascada*. Pero las reacciones no son unicamente para implementar dicho borrado. El borrado en cascada *es una* de las aplicaciones prácticas del uso de reacciones (el cual se ha usado aquí como ejemplo por su sencillez).

[19] Podría argumentarse que no todo objeto tiene una relación con un objeto contenedor. Sin embargo en la práctica suele tenerlo (por ejemplo los procedimientos son agregados del editor). En cualquier caso en la clase Entity se ha añadido una operación *delete* (la cual borra todas sus relaciones) para aquellos objetos no agregados (la cual, en la práctica, rara vez es necesaria).

Nótese que, por ejemplo, al borrar un procedimiento se acaban haciendo los mismos pasos que antes había que codificar de manera secuencial y conjunta (borrar procedimiento, sus actividades, sus salidas, etc). Pero ahora esos pasos *se organizan dinámicamente* en función de la estructura de objetos. Y se rehacen al aparecer nuevas entidades. El código que se ejecuta es *el mismo*; pero no está escrito en un único método. Esto permite cambiar las piezas individualmente y que automáticamente se ensamblen *sin instrucciones expresas*. La operación *se adapta a la estructura del modelo* y actúa adecuadamente ante cada objeto y relación *sin saberlo*.

7.1.4.4 Efecto de los Monitores Sobre las Operaciones Descentralizadas

Cuando se modifica un miembro de una entidad no sólo hay que comprobar las validaciones de dicha entidad; la notificación de la modificación puede producir reacciones en cascada en otras entidades que a su vez tendrán sus propias validaciones que habrá que respetar. Si éstas no se cumplen *habrá que rechazar la modificación que se realizó sobre la entidada inicial* (aunque ésta *no conoce* a las otras entidades a las que ha afectado ni sus validaciones específicas).

Supóngase por ejemplo que se tiene un objeto *pepe* asociado a una serie de objetos *perro* mediante una relación *mascota*. También existe una relación *matrimonio* que asocia a dos personas con la condición de que no se podrán unir cuando alguna de las dos ya esté casada.

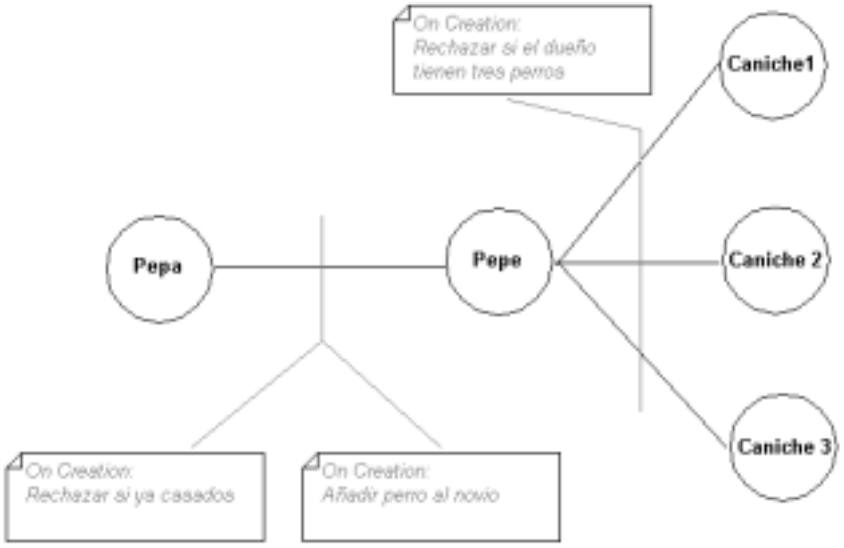

Imagen 7.1-20. Diagrama de objetos con sus monitores y reacciones.

Supóngase que además, por motivos culturales, la tradición dicta que cuando dos personas se casan el padre de la novia tiene que regalar como dote un perro a su yerno. Por tanto se añade una nueva reacción a la relación matrimonio de tal manera que al producirse un enlace se añada un perro al marido.

Supóngase que posteriormente surge una supuesta ley que impida que una persona pueda tener más de tres perros. Se añadiría una validación a la relación *mascota* que impida añadir un cuarto perro.

Sin embargo, a la hora de realizar la operación *matrimonio* ¿qué pasa si el marido tiene ya tres perros? El marido no podría realizar la boda hasta que regale alguno de sus perros. Es decir, al añadir una relación *matrimonio* se produce la siguiente secuencia de operaciones:

- Se activa la validación de que los novios no estén ya casados. Esta validación se cumple.
- Se añade la relación *matrimonio* entre *pepe* y *pepa*.
- Al notificar a los suscriptores de que hay un nuevo matrimonio se activa la reacción *dote* que añade una relación entre el novio y el nuevo perro regalado por su suegro.
- Al añadirse la relación entre el novio y el perro se comprueban a su vez los monitores de esta relación. Al tener más de tres perros se rechaza la relación produciéndose una excepción.
- La excepción le llega a la reacción *dote* que, al ver que el regalo no puede hacerse, rechaza a su vez la relación matrimonio.

Por tanto la operación *matrimonio* es rechazada por razones que su programador no tuvo porqué tener en cuenta, pero que el sistema en global sí que tiene que contemplar.

Supóngase que ahora la norma de posesión de perros se relaja y la ley dicta que los caniches cuentan como medio perro. Se retira la validación antigua y se sustituye por otra que tenga en cuenta este aspecto. A partir de ese momento la operación *matrimonio* vuelve a afectar a pepe ya que sus tres perros eran caniches.

Nótese ahora cómo se hubiera hecho *sin usar* RDM. Es decir, si en la función *matrimonio* hubiera que haber sabido todas las precondiciones para poder ejecutar el matrimonio. En la primera implementación de dicha operación (antes de que surgiera la ley que impedía tener más de tres perros) se hubiera incrustado el código que comprobaba si los dos novios estaban solteros:

```
public void matrimonio(Marido el, Mujer ella) {
      if (el.soltero() && ella.soltera())    // precondición para el
matrimonio
            casarlos();
}
```

Ese era el único impedimento que se veía para anular la operación.

Posteriormente hubiera aparecido la norma de que una persona no pueda tener más de tres perros. Entonces hubiera habido que modificar la operación la cual, en principio, no tenía ni que conocer *que había una relación* entre personas y perros.

```
public void matrimonio(Marido el, Mujer ella) {
      if (el.soltero() && ella.soltera() && el.numeroPerros() < 3)
      // precondición para el matrimonio
            casarlos;
}
```

Sin embargo las precondiciones de dicha operación efectivamente han cambiado. La nueva regla para permitir el matrimonio ahora debe ser "que los novios estén solteros y que el marido no tenga ya tres perros". Es decir, la operación, además de conocer la única relación necesaria para

realizar su función – relación de matrimonio – tenía que saber todas las reacciones qué ésta iba a producir en todas las demás entidades para saber *además qué validaciones se deberían realizar* para evitar corromper un modelo *que no debiera conocer*. ¿Cómo iba a suponer el codificador de la operación *matrimonio* que tendría que tener en cuenta el número de perros que tuviera el novio?

El problema no solo es que haya que modificar la operación *matrimonio con cada cambio* de reglas de entidades desconocidas para ella. Otro problema más importante aún es el *no darse cuenta* de que hay que modificarla. De repente aparecerán personas con más de tres perros ¿por dónde se están introduciendo todos esos errores que vulneran la regla? Por el matrimonio.

Volviendo a la arquitectura RDM lo que se ha conseguido con ésta es que basándose en validaciones simples se vayan formando dinámicamente validaciones complejas que se reconstruyan automáticamente cada vez que se añaden o quitan éstas. Es decir, a la hora de ejecutar la operación *matrimonio* hay que comprobar "que los novios estén solteros y que el marido no tenga ya tres perros". Y efectivamente eso *se está comprobando* aunque en ningún sitio aparezca todo junto. De la misma manera cuando se cambió la regla para contemplar la excepción de los caniches la *precondición* de la operación matrimonio *automáticamente se rescribió* como "que los novios estén solteros y que el marido no tenga ya tres perros contando a los caniches como medio perro". Y todo eso se obtuvo sin tocar el código de la operación *matrimonio*.

7.1.4.5 Los Monitores en el Aumento de la Reutilización

El hecho de poder afectar al funcionamiento de una operación de manera externa mediante monitores (como se ha visto en el apartado anterior) tiene un gran impacto en las posibilidades de reutilización de las operaciones.

Supóngase que en el ejemplo del apartado anterior, en la que una persona no podía poseer más de tres perros, se pretende añadir un interfaz de usuario visual en incluir en él las operaciones de selección y copiado a través del portapapeles (operaciones de cortar y pegar). De esta manera se podrían seleccionar perros y moverlos de un dueño a otro.

¿Cómo se implementaría la operación *paste* (pegado desde el portapapeles) si no hubiera monitores?. Supóngase que se seleccionan tres perros y se llevan al portapapeles. Posteriormente se intentan pegar sobre otro dueño que tiene ya tres perros. La operación de pegado no debería permitirlo porque el dueño acabaría con 6 perros, lo cual viola la regla.

¿Será necesario entonces añadir un caso especial dentro del código de la operación *paste* que evite el caso concreto de pegar más de tres perros?

- Si dentro de la operación paste se implementa la regla de no pegar si se juntan más de tres perros la operación no podrá ser reutilizada en otra aplicaciones. Difícilmente tendrá la clase perro en su modelo ni la restricción de tres perros como máximo.

- Si no se incluye dicha regla la operación paste entonces directamente ya no vale para ésta aplicación.

¿Entonces solo podrán reutilizarse operaciones sobre modelos que no tengan sus reglas particulares (como la del límite de perros)? En ese caso nunca podrán reutilizarse ya que todo

modelo tendrá sus particularidades. Esa es la razón por la que no se reutilizan las operaciones: siempre hay que estar modificándolas para que se adapten a las reglas del dominio, a sus particularidades, y por tanto *se acaba antes* haciéndolas de nuevo.

Con RDM las validaciones descentralizadas permiten utilizar una única versión del paste para cualquier modelo que se limite a copiar lo que halla en el portapapeles y añadirlo al modelo sin importarle lo que és. Pero de tal manera que, si dicha operación no es aplicable en el contexto actual, se rechazará dejando el modelo sin cambios. En el caso del paste, sin que ella lo sepa, la inserción de los perros será rechazada. Recibirá una excepción indicando la causa del rechazo; explicación que podrá ser mostrada al usuario a través del interfaz de usuario.

Se pueden reutilizar las operaciones pero además sin renunciar a las particularidades que haya que contemplar sobre el nuevo modelo en el que se aplique.

7.1.5 Tratamiento de Errores: Reconstructores

Anteriormente se ha visto que cuando un monitor lanza una excepción la entidad deshace la modificación del miembro por lo que la entidad permanece estable.

Supóngase que la clase persona tiene un monitor que limita la edad de una persona al intervalo entre 0 y 200 años.

```
persona.getInt("edad");    --> Devuelve 5
persona.setInt("edad", -1);      --> Devuelve excepción suponiendo un
monitor adecuado conectado
persona.getInt("edad");    --> Devuelve 5
```

Sin embargo se vio también que al añadir las reacciones esa estabilidad, aunque se seguía manteniendo para objetos individuales, ya no se mantenía para el modelo en global. En el capítulo anterior se explicó como en una situación como la de la figura unas entidades del modelo eran modificadas y otras no.

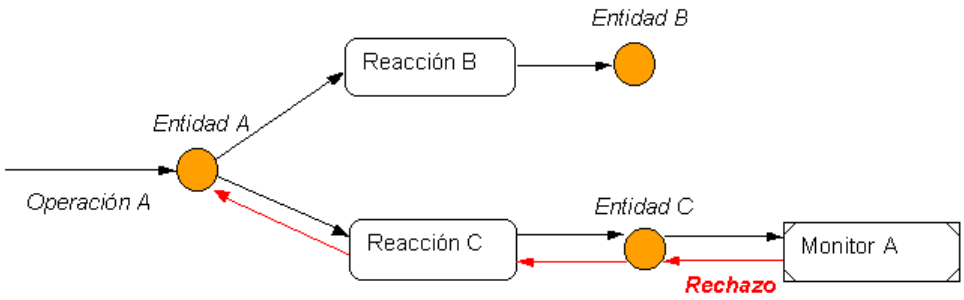

Imagen 7.1-21. Reacciones y el rechazo de operaciones por monitores.

Por tanto al implementar las operaciones de manera descentralizada se plantea un problema de integridad en el modelo. La primera opción sería guardar una copia del modelo y si la operación falla restaurarlo completamente. Sin embargo esta opción obviamente supone una gran carga de tiempo y memoria por lo que se hace impracticable.

7.1.5.1 Patrón Command

La siguiente opción sería intentar guardar sólo las entidades que se han modificado e incluso sólo los cambios que se han realizado sobre ellas. Para ello un candidato sería el Patrón Command con un mecanismo parecido al que ya presenta en el esquema Undo/Redo. Cada vez que una entidad se modifique genera un Command de tal manera que si algo sale mal el Command permita deshacer los cambios.

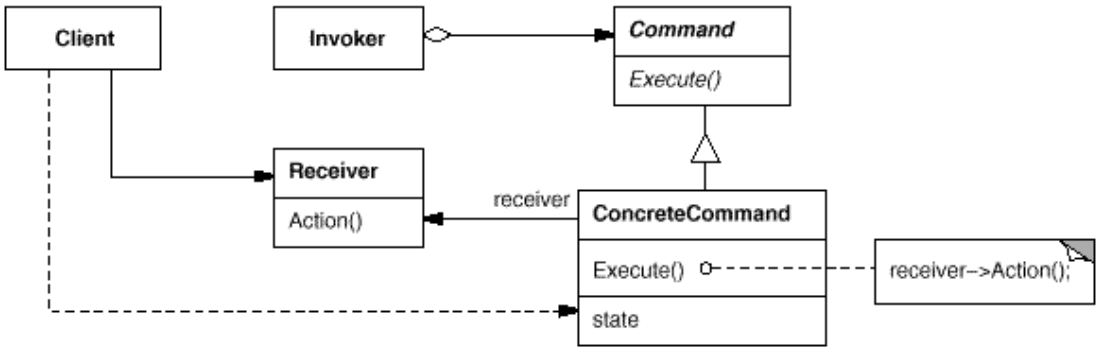

Imagen 7.1-22. Estructura del Patrón Command

Pero el Patrón Command no es una solución completa en el problema que aquí se presenta. Un Command es un objeto externo que intenta *modificar* una entidad para establecerla a un valor anterior. Sin embargo:

- No se puede usar el interfaz público de la entidad. El cambio del miembro, como cualquier otro cambio de la entidad, producirá nuevas validaciones y reacciones en cadena en otras entidades. Lo que se está pretendiendo es restaurar un modelo debido a que una validación ha rechazado la operación. No tiene sentido aplicar más validaciones sobre un modelo *que se sabe inestable* (con lo cual se producirán *más excepciones en cascada*) y mucho menos provocar más reacciones *que cambien otras entidades* cuando la operación se quiere anular. Esto ya daría lugar a un modelo imposible de restaurar.

- Podría pensarse en alguna forma de "desactivar" temporalmente los eventos y relaciones. Además de que esto sería realmente engorroso y una gran fuente de errores debido al olvido de volver a conectarlo tampoco sería solución. Cuando un Command se utiliza en un esquema de Undo hay veces que solo puede hacer "la operación contraria" a la operación que tiene que deshacer. Sin embargo no es lo mismo realizar "la operación contraria" de una operación que "deshacer" dicha operación (que es lo que realmente tiene que hacer).

Para aclarar este último punto supóngase que se utiliza la siguiente clase para saber cuantas relaciones se han establecido y cuantas se han eliminado del modelo:

```
class Counter implements RelationListener
{
	private int creadas = 0;
	private int eliminadas = 0;

	void incrementa()
	{
		creadas++;
	}

	void decrementa()
	{
		eliminadas++;
	}
}
```

Supóngase que se quiere implementar el objeto Command que deshaga la operación *incrementa*. ¿Debería el Command invocar entonces a la operación contraria *decrementa*? Puede verse en el código que eso no funcionaría ya que eso no deshace la operación (que debiera ser decrementar la variable *creadas*). El Command no puede deshacer la operación.

Podría proponerse entonces añadir los métodos para deshacer cada una de ellas (*undoIncrementa* y *undoDecrementa*).

```
class Counter
{
	private int creadas = 0;
	private int eliminadas = 0;

	void incrementa()
	{
		creadas++;
	}

	void undoIncrementa()
	{
		creadas--;
	}

	void decrementa()
	{
		eliminadas++;
	}

	void undoDecrementa()
	{
		eliminadas--;
	}
}
```

Sin embargo estos métodos serían públicos con lo cual se está dando acceso al estado interno del objeto. Además no tiene sentido que por cada método de una clase tenga que haber otro que lo deshaga, ya que eso duplicaría el interfaz de las clases.

En definitiva el Patrón Command no es solución para implementar los Reconstructores ya que al modificar las entidades para restaurarlas pueden obtenerse efectos laterales debido a las reacciones.

7.1.5.2 Patrón Memento

Como ya se ha visto con el Patrón Command no se puede reconstruir un objeto a través del interfaz público. Por tanto otro candidato mejor es el Patrón Memento. Este patrón se basa en que a un objeto se le pueda pedir que entregue otro denominado Memento que encapsulará su estado interno actual. La forma de representar dicho estado es privada al objeto y nadie más podrá acceder a ella. Lo único que se puede hacer con ese objeto es guardarlo. Si posteriormente se quiere que dicho objeto recupere su estado se le pasará de nuevo el Memento y este sabrá como interpretarlo adecuadamente.

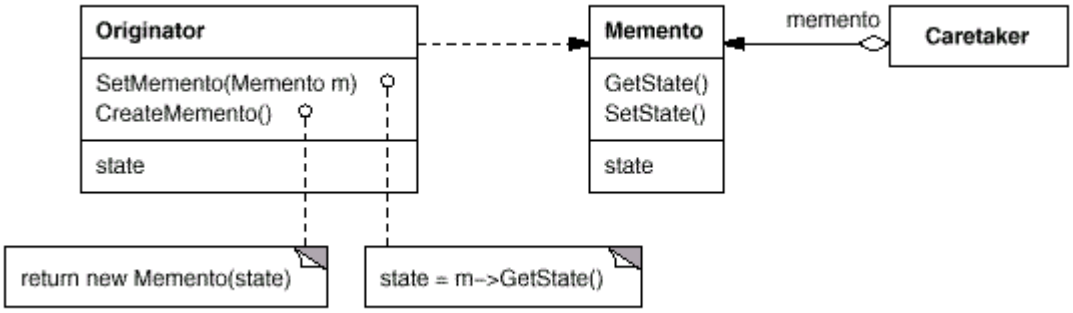

Imagen 7.1-23. Patrón Memento

Nótese que aquí, dado que otros objetos no conocen su estado interno y no pueden modificarlo, es el propio objeto el que se restaura cuando se le solicita.

Sin embargo el Memento tiene también algún inconveniente que hay que evitar:

- Se basa en que algún objeto le pida a la entidad su estado actual. Sin embargo no se sabe qué entidades van a ser afectadas por las reacciones por lo que no se sabe a quienes pedir. Se podría usar una reacción (en cada una de las entidades) que guardara el estado de la misma si esta notifica que va a ser modificada. Sin embargo el orden de invocación de las reacciones no está establecido y no se garantiza que vaya a llamarse a dicha reacción antes que a otra[20]. Y, por supuesto, las entidades no deben *conocer* a ese tipo de *reacción especial* para darle un tratamiento preferente.

- Cuando posteriormente se quiera restaurar el estado hay que saber a qué objeto corresponde cada Memento para darle a cada uno el adecuado. El Patrón Command no necesitaba recordar quién había generado cada comando, lo cual es mucho más práctico.

- El Memento guarda todo su estado, cuando sólo una parte del mismo ha sido modificada.

[20] Y esto es necesario ya que si una reacción es invocada antes de guardar el estado de la entidad puede que se produzca una excepción y el estado nunca sea guardado.

7.1.5.3 Diseño de los Reconstructores

Los reconstructores se han implementado cogiendo una parte de cada uno de los patrones anteriormente descritos.

- Un Reconstructor es un Memento que guarda únicamente la parte del estado interno que ha sido modificado en una entidad.

- Sin embargo es autónomo como un Command de tal manera que es él mismo (y no la entidad) el que restaura el estado interno de la misma. Para ello deberá tener algún tipo de privilegio sobre ella (por ejemplo en C++ sería *friend*). Por tanto no produce reacciones ya que no va a través del interfaz público.

El diseño de las clases relativas a los reconstructores es:

```
interface Reconstructor
{
	void reconstruct();
	void discard();
}


class ReconstructionManager
{
	add(Reconstructor r) { .. }
	reconstruct();
	discard();

	openContext() {... }
	closeContext() {...} // Se vuelve al contexto anterior añadiéndole
los reconstructores del actual

}
```

Un reconstructor es el encargado de restablecer un elemento del modelo (una propiedad o una relación). Cada vez que una parte del modelo es modificada ésta genera un objeto reconstructor que pueda deshacer el cambio si algo va mal. Dicho reconstructor se añade al *ReconstructionManager*. A medida que las reacciones van modificando otras entidades estas van añadiendo igualmente su reconstructor.

```
class StandardEntity implements Entity
{
	void set(String member, Object value)
	{
		ReconstructionManager.add(new MemberReconstructor(this,
member, get(member)));

		<modificar atributo>
		<notificar a reacciones y monitores>
	}

	...<resto métodos Entity>
}

//
// Clase ejemplo que reconstruye un cambio de un atributo
//
class MemberReconstructor implements Reconstructor
{
	MemberReconstructor(Entity entity, String member, Object oldValue)
	{
		<guardar datos necesarios para la posible reconstrucción>
	}

	void reconstruct() {
		<volver a poner oldValue sobre el miembro de la entidad>
	}

	void discard() {
	}
}
```

Al acabar la operación si todo ha ido bien se descartan los reconstructores mediante el método *ReconstructionManager.discard*. En cambio si se ha incumplido alguna validación se reconstruye el modelo con el método *ReconstructionManager.reconstruct*.

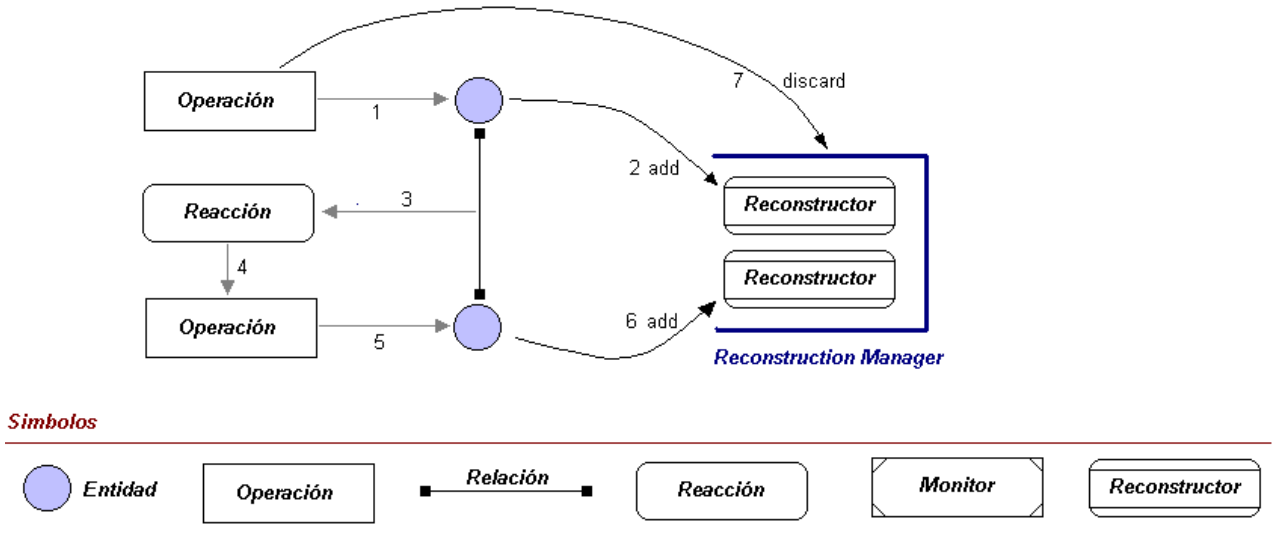

Imagen 7.1-24. Uso del Reconstruction Manager.

En la figura anterior se realiza la siguiente secuencia de pasos:

- Todo comienza cuando una operación modifica la entidad superior. Esta genera un reconstructor y lo guarda en el *ReconstructionManager* (pasos 1 y 2).
- A continuación notifica a todos sus suscriptores que ha cambiado de estado (paso 3).
- Dicha notificación activa una reacción que ejecuta una nueva operación que modifica una segunda entidad (pasos 4 y 5).
- La entidad modificada, al igual que lo hizo la primera, guarda su reconstructor por si hay que anular la operación (paso 6).
- Finalmente la ejecución retorna a la operación inicial y, dado que no se ha producido ninguna excepción, le dice al *ReconstructionManager* que descarte todos los reconstructores almacenados ya que no van a ser necesarios (paso 7).

En la figura anterior puede verse como la operación se realiza sin problemas y los reconstructores que se han ido generando a medida que se modificaba el modelo son descartados.

En la figura siguiente se muestra el escenario opuesto.

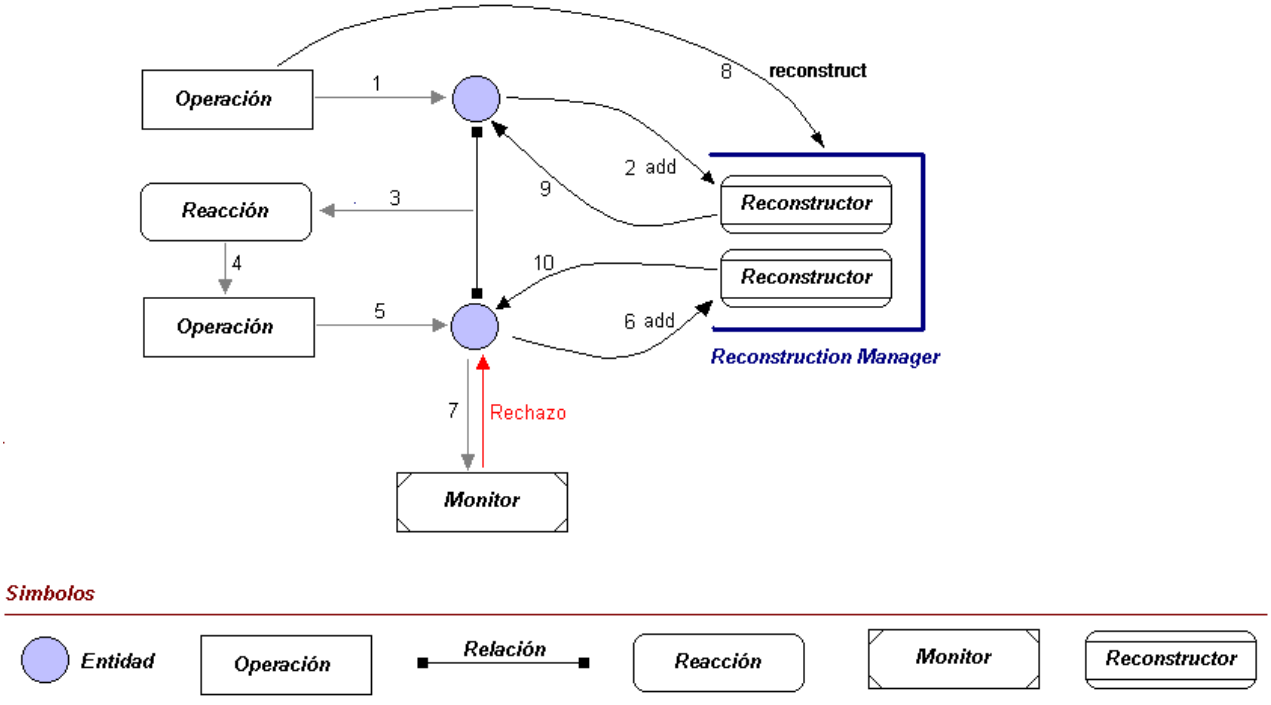

Imagen 7.1-25. Reconstrucción de un modelo.

En este escenario los cuatro primeros pasos son iguales al caso anterior (pasos 1 a 6). Sin embargo, después de que la segunda entidad haya guardado su reconstructor, ocurre lo siguiente:

- La entidad notifica que se ha modificado su estado (paso 7).

- Dicha notificación le llega a un monitor que, por alguna determinada razón, rechaza el cambio de estado generando una excepción.

- La excepción se propaga siguiendo la secuencia de llamadas en orden inverso hasta llegar de nuevo hasta la operación inicial la cual, al atrapar la excepción, indica al ReconstructionManager que active los reconstructores para dejar el modelo tal y como estaba cuando ésta comenzó su ejecución (pasos 8, 9 y 10).

Por lo explicado anteriormente puede comprobarse entonces que la estructura general de una operación será la siguiente:

```
class OperationPattern
{
	void run(Entity persona)
	{
		<sentencias que modifiquen el modelo>

		if (excepción de rechazo) {
			ReconstructionManager.reconstruct();
			<informar al usuario >
		}
		else
			ReconstructionManager.discard();
	}
}
```

Un Reconstructor solo tiene sentido mientras se está ejecutando una operación por lo que al acabar ésta deben descartarse todos. En caso de que durante la ejecución haya habido alguna excepción éste se habrá usado pero si todo ha ido bien se descartarán sin llegar a ser usado. Esta es una diferencia importante con el Command el cual sobrevive a la operación que lo creó.

En definitiva después de invocar a un método de una entidad solo hay dos posibles respuestas:

- O la operación se ha llevado a cabo con éxito y se han añadido los Reconstructores correspondientes en el ReconstructorManager.

- O la operación devuelve una excepción indicando la razón del rechazo. En ese caso tanto la entidad invocada como las entidades afectadas indirectamente estarán en el estado en el que estaban antes de realizar la invocación.

7.1.5.4 Reconstrucciones Parciales

Cuando una operación o reacción intenta modificar una entidad puede encontrarse una excepción de un Monitor. Como ya se ha comentado la entidad de la cual se recibe la excepción ha sido restaurada. Sin embargo dicha operación pudo haber modificado previamente otras entidades. Por tanto la operación tiene la posibilidad de anular todos esos cambios previos o bien dejarlos sin deshacer y seguir con otras acciones.

Para ello la clase ReconstructionManager incorpora *Contextos*. Los reconstructores se añaden automáticamente en el último contexto que se haya abierto. Cuando al manager se le dice que reconstruya el modelo solo invoca los reconstructores que pertenezcan al último contexto creado. Por defecto en el manager solo hay un contexto con lo cual se desharía todo. Sin

embargo si se abren contextos en puntos determinados se puede controlar qué acciones deshacer y cuales mantener.

Por ejemplo supóngase el siguiente modelo:

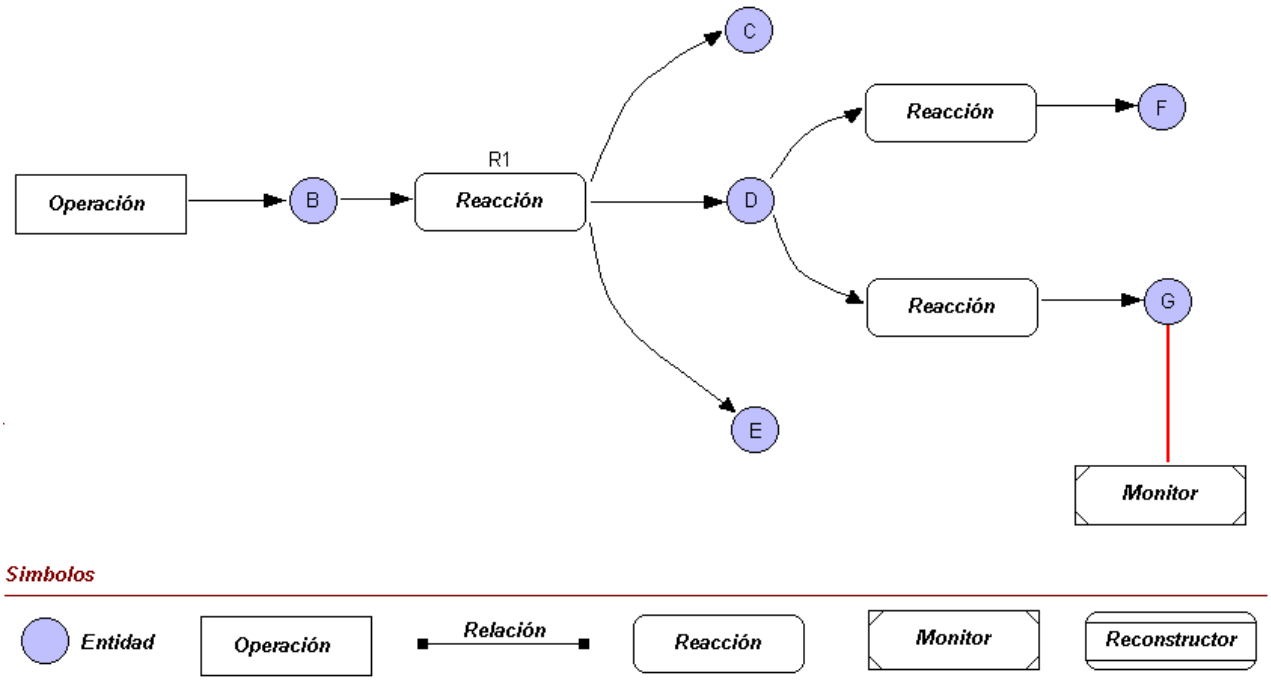

Imagen 7.1-26. Reconstrucción parcial de modelos.

La operación A modifica la entidad B y como consecuencia de ello se activa la reacción R1. Esta reacción pretende modificar tres entidades más (C, D y E).

- La modificación de C se realiza con éxito. Por tanto al volver a R1 en el ReconstructionManager estarán los reconstructores de B y de C.

- Después se modifica D lo cual produce otra nueva reacción sobre F la cual se modifica sin problemas. Al volver de nuevo a D se tendrán los reconstructores de B, C, D y F.

- Pero D también produce otra reacción sobre G. Sin embargo dicha reacción es rechazada por lo que el control vuelve a D con una excepción. Esta entidad debe salir dejando todo tal y como estaba cuando la invocaron por lo que se ejecutan los reconstructores de F y D.

- La excepción propagada por D llega a R1. Sabe que D está estable. Ahora puede decidir si deshacer también C o bien continuar con E. Si decide abortar toda la operación entonces basta con que llame a *reconstruct* para ejecutar el reconstructor de éste nivel (el de C) y relance la excepción que ha recibido. Esta será atrapada por la entidad B y se ejecutarán los reconstructores que queden (el de B). En cambio puede decidir continuar y modificar E.

7.1.5.5 Reconstructores como Eventos Especializados

Todos los elementos de la arquitectura surgen como consecuencia de aplicar el *Principio de Especialización* a los distintos elementos del dominio. Los reconstructores también nacen por la aplicación de este principio; pero en vez de haberlo hecho sobre el dominio se ha aplicado sobre el modelo de eventos. Los Reconstructores *son la especialización de un tipo de eventos*. Para entender mejor esto se planteará un ejemplo sin los reconstructores para ver cómo se estaban implementando tradicionalmente.

Supóngase que un objeto lanza un evento a sus suscriptores debido a la modificación de un atributo. Cada uno de estos suscriptores realizará las acciones que considere oportunas ante dicha modificación. Supóngase que posteriormente se quiere deshacer dicha modificación. Habrá que avisar de nuevo a los suscriptores porque tendrán que corregir las acciones que tomaron. Por tanto habrá que notificarles con otro evento.

En el ejemplo siguiente[21] se muestra la clase *DepartamentoDePersonal* de una empresa. Esta está interesada en dos notificaciones del director: que se contrate y que se despedida a un empleado. En este último caso, debido a una supuesta ley, la empresa tendrá que indemnizar al trabajador con un millón de pesetas.

```
class DepartamentoDePersonal implements DirectorListener
{
	Vector empleados = new Vector();

	void contrataEmpleado (Empleado empleado)
	{
		empleados.add(empleado);
		<otras acciones>
	}

	void despideEmpleado (Empleado empleado)
	{
		empleados.remove(empleado);

		// ¿despido o anulación de un contrato abortado?
		empleados.indemnizar(1000000);
	}
}
```

Supóngase que como parte de una operación se contrata a un empleado. Sin embargo, y después de que ya se haya informado a este departamento, se anula la operación (y por tanto la contratación). Entonces habrá que avisar al departamento con otro evento *despideEmpleado* para que actualice su información. Pero ¿es el mismo código para despedir a un empleado que para descartar una contratación? En el segundo caso no hay que dar indemnización ¿Cómo saber entonces si es una anulación o un despido real? El método *despideEmpleado* tiene estas dos responsabilidades y va a tener que decidir entre ellas[22].

---

[21] Los eventos que aquí recibe la clase Empresa son ficticios y no se lanzan en el framework RDM. En RDM realmente se lanzarían los eventos genéricos de que se "ha añadido una relación" y el nombre de la relación sería el parámetro. Sin embargo en el ejemplo se han concretado los nombre de los métodos para su mejor comprensión.

[22] Se recuerda que la opción de tener otro método distinto para deshacer la contratación supondría tener todos los métodos duplicados como ya se vio con el Patrón Command.

Por tanto los eventos se están utilizando para dos cosas distintas:

- Para indicar *modificaciones* del estado.
- Para indicar *rectificaciones en las modificaciones* del estado.

Los Reconstructores serán entonces *la especialización de los eventos que señalan que hay que rectificar un cambio* en una entidad. Dichos eventos se extraen del modelo de eventos de tal manera que ambas cosas se puedan implementar de manera más sencilla por separado:

- Se invocan objetos suscriptores para indicar modificaciones debido a una operación.
- Se ejecutan reconstructores para indicar que se ha anulado dicha operación.

A continuación se muestra la nueva implementación con RDM. Ahora se separan claramente los conceptos de "deshacer una operación" y "hacer la operación opuesta". Cada uno tiene su sitio en la arquitectura.

- En el método *contrataEmpleado* se realiza la contratación pero, *por si esto se anula*, deja además un reconstructor que es el encargado de deshacer dicha operación. Pero ambas cosas se implementan en sitios distintos lo cual simplifica su implementación.
- La operación *despideEmpleado* solo tiene que preocuparse de despedir a un empleado. Nunca se la llamará para indicar que tiene que deshacer *lo que ha hecho otro método*. De esta manera *se centra en su funcionalidad* quedando una implementación más simple.

```
class Empresa
{
	Vector empleados = new Vector();

	void contrataEmpleado () {
		empleados.add(empleado);
		añadir Recontructor(<eliminar empleado del vector>); // Por
si se rechaza
	}

	void despideEmpleado () {
		// centrarse en un despido normal (con indemnización)
		empleados.remove(empleado);
		empleados.indemnizar(1000000);
	}
}
```

Es decir, una entidad implementa el caso de que todo vaya bien. Pero *por si algo va mal* entrega un reconstructor para volver a un estado válido. Podría considerarse que fuera como si las recuperaciones ante fallos tuvieran *su propio hilo de ejecución* – que solo se activa en caso de fallo – y que si alguien quiere incorporar código a dicho hilo solo tiene que añadir su reconstructor[23] al ReconstructionManager.

[23] Esto es en sentido figurado ya que RDM no requiere de hilos.

Cuando anteriormente se dijo que los reconstructores modifican a las entidades sin que éstas lancen eventos pudo haber quedado la duda de si realmente no habrá ningún otro objeto que necesite enterarse de la anulación de la operación y por tanto debiera habérsele avisado con un evento. Pero dado que ahora se ha establecido que ninguna parte del modelo cuenta con los eventos para enterarse de los rechazos (para eso están los reconstructores) se ve que no hay ningún peligro en no lanzar eventos[24]. Si hay algún objeto interesado en enterarse de la anulación de la operación simplemente tiene que añadir su reconstructor y olvidarse de la misma. Ya se invocará éste si ocurre algo. Los *reconstructures son las suscripciones* a las anulaciones.

7.1.5.6 Los Reconstructores en la Implementación

Aunque pueda haber quedado la sensación de que para hacer cualquier operación hay que implementar demasiado código debido al tema de los reconstructores en la práctica no es así. Los reconstructores no dependen del dominio. Todo ese código ya está encapsulado en las clases del framework. En el caso de las entidades y las relaciones éstas ya incluyen el código para añadir reconstructores cuando las modifique una operación. Por tanto cuando se realiza un diseño no hay que plantearse la implementación de los reconstructures. Estos se están reutilizando tal cual de manera transparente y en la práctica no hace falta saber de su existencia.

En el caso de la implementación de *operaciones* tampoco es necesario tratar con los reconstructores ni con el ReconstructionManager. Basta con hacer una implementación base que, mediante el patrón *Template Method*, permita que las nuevas operaciones se limiten a *redefinir y añadir el código específico* de dicha operación obteniendo así la gestión de la reconstrucción sin saberlo.

```
class OperationPattern
{
	void run(Entity entity)
	{
		templateMethod(entity);

		if (excepción de rechazo) {
			ReconstructionManager.reconstruct();
			<informar al usuario del rechazo>
		}
		else
			ReconstructionManager.discard();
	}

	abstract void templateMethod(Entity entity) {}
	// Redefinir este método en cada nueva operación
}
```

[24] Las primeras propuestas para modelar los reconstructores tomaron como axioma que no se debía cambiar el estado interno de un objeto desde fuera. Este es al fin y al cabo uno de los principios fundamentales de la orientación a objetos. ¿Cómo iba a estar bien el modificar directamente un objeto sin notificar además a sus suscriptores? Sin embargo esto no funcionaba debido a las reacciones que se producían al confundir operación con reconstrucción. Finalmente el replantearse dicho axioma fue la solución adecuada.

```
class NuevaOperacion extends OperationPattern {
      void templateMethod(Entity entity) {
            //Poner aquí las acciones de esta operación
      }
}
```

El programador de la operación conseguirá así una operación que o bien se ejecuta correctamente o bien deja el modelo como estaba.

Al igual que con las operaciones descentralizadas (por ejemplo la del borrado 7.1.4.3, que no había que hacer una versión distinta para cada tipo de entidad) en las restauraciones del modelo se forma dinámicamente un macro-reconstructor bajo demanda. Es decir, no hay que implementar una operación de reconstrucción por cada operación y por cada sentencia en la que pueda fallar (y deshacer así más o menos cosas). Éste se crea automáticamente en función de a qué parte del modelo haya accedido mediante la combinación dinámica de los reconstructores independientes. Por tanto se ejecutan las *mínimas operaciones necesarias* para restaurar el modelo.

De esta manera el programador no se tiene que preocupar de las excepciones (llenando el código de try/catch) ni de arreglar el modelo si algo falla. Sin embargo tiene la tranquilidad de que todo eso se está contemplando internamente y tratando adecuadamente. Lo que ocurre es que se hace de una vez y para todas las aplicaciones en vez de tenerlo que repetir en cada una (o mejor dicho, *no implementándolo* en ninguna suponiendo que no van a ocurrir excepciones ni se va a dejar el modelo a medio modificar).

7.2 Métodos para el Modelado de Elementos

En este apartado, una vez presentados en detalle todos los elementos de la arquitectura, se expondrán los criterios de modelado a seguir en cada uno de los elementos.

Antes de ello se incluye un diagrama de clases del framework que fije los elementos fundamentales del framework. No pretende ser un diagrama de clases exhaustivo (ya que faltan tanto clases como relaciones) sino mostrar sus elementos más importantes los cuales han sido comentados en este capítulo por separado.

En el diagrama aparece una línea horizontal que pretende separar dos áreas del diagrama:

- En la parte superior se encuentran las interfaces que forman la verdadera esencia del framework. Con estos tipos es con los que se trabaja sin importar sus implementaciones concretas.

- En la parte inferior se encuentran las implementaciones por defecto de dichas interfaces de tal manera que el framework sea operativo. Sin embargo estas implementaciones se podrán cambiar sin afectar al resto del framework.

Como puede observarse el diseño es muy simple (intencionadamente simple) ya que basta entender cuatro tipos básicos (la parte superior) para entender la estructura del framework y su forma de uso.

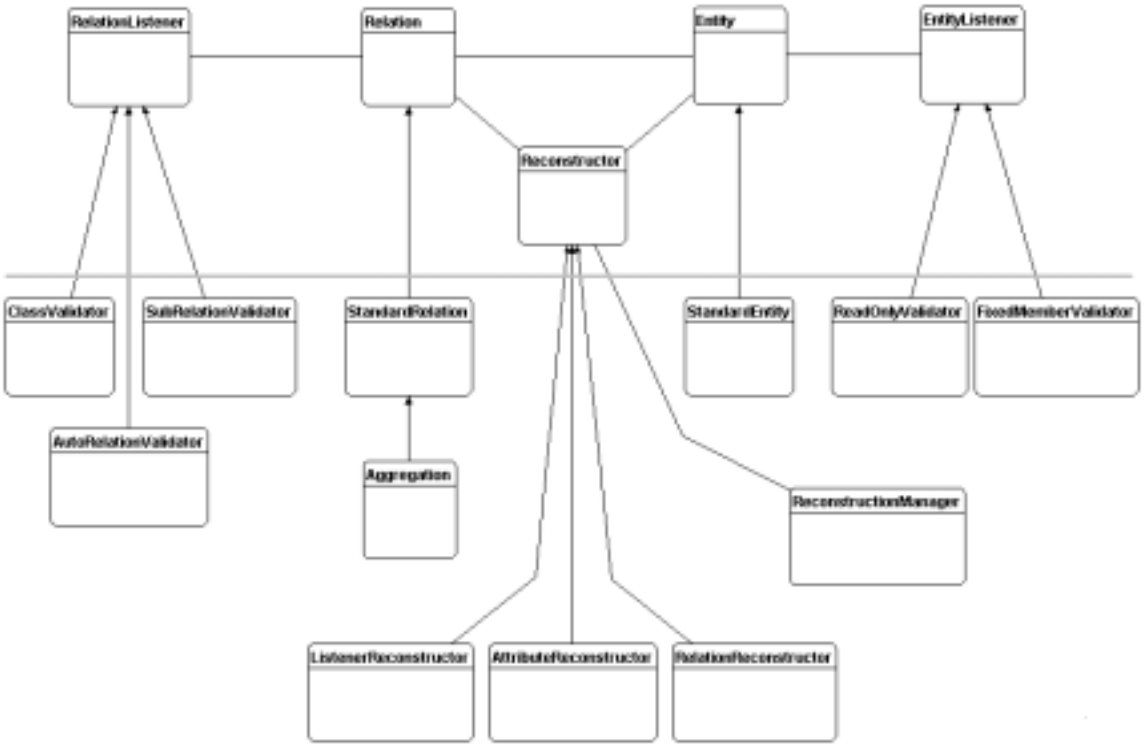

Imagen 7.2-1. Diagrama de las principales clases del Framework RDM

Una vez establecido el *mapa* de los elementos se describirán los métodos para su adecuado modelado siguiendo la tesis presentada.

7.2.1 Método de Modelado de Entidades

Para modelar una entidad hay que centrarse *en las propiedades que la caracterizan*. Por tanto una entidad **no debe incluir**:

- Relaciones con otras entidades.
- Las comprobaciones de la validez de las modificaciones de su estado. Éstas pueden variar y además el que una entidad esté en un estado válido no quiere decir que lo esté todo el modelo.
- Las notificaciones de los cambios de estado no deben ser gestionados por las entidades. Cada vez que se añada una nueva entidad no habrá que plantearse qué eventos debe emitir. La arquitectura deberá avisar de todo cambio de estado de cualquier miembro de cualquier entidad (atributos y relaciones) y de forma automática.
- La navegación por los objetos del modelo no debe ser responsabilidad de las entidades. Ésta *no debe incluir métodos de enlace* con otras entidades.

```
lassie = sandokan.getDueño().getAbogado().getPerro();      // No
```

La arquitectura debe proveer de un mecanismo de navegación externo a las entidades.

7.2.1.1 Uniformidad de Acceso

Las entidades deben ofrecer uniformidad de acceso a la hora de acceder a los miembros, es decir, no se debe necesitar saber si un valor se almacena mediante una propiedad o mediante una relación con cardinalidad simple.

```
persona.get("coche");     // ¿atributo o relación "de x a uno"?
```

Durante el uso de RDM habitualmente surge la duda de cuando utilizar un atributo y cuando una relación *uno a uno*. Es más, se podría plantear entonces eliminar las propiedades y diseñar todo mediante relaciones de éste tipo. Por ejemplo una persona y su nombre se podrían implementar mediante dos entidades relacionadas de uno a uno.

El criterio es usar atributos cuando no es necesario mantener la bidireccionalidad de la relación entre dos elementos (a un atributo no se va a necesitar preguntarle con qué entidad está relacionado) y además el atributo *no va a compartirse* con otras entidades. Por tanto se obtiene una implementación más compacta y eficiente al no instanciar objetos *relación*.

Pero dado que el usar atributos, como acaba de verse, es una decisión dirigida por la eficiencia en implementación por eso es tan importante el principio de uniformidad de acceso. La operación debe poder acceder a un miembro sin necesidad *de saber* su implementación: no debe depender de si su creador decidió usar una propiedad o una relación uno a uno.

7.2.2 Método de Modelado de Relaciones

Las relaciones se modelarán como objetos independientes. Ellas mismas se encargarán de la gestión de su integridad (el mantenimiento de la bidireccionalidad y de la cardinalidad) así como del protocolo de establecimiento de relaciones. Se libera así a las entidades de dichas tareas.

7.2.2.1 Asignación de las relaciones a las Entidades

A la hora de asignar relaciones a las entidades se pueden tomar dos enfoques.

- Se puede utilizar el constructor de la entidad para añadir las relaciones que deba tener toda instancia de dicho tipo.

```
public class Activity extends StandardEntity {

	public Activity () {
		add(NodesRelations.r1, "activity");
		add(NodesRelations.r2, "activity");
		add(NodesRelations.r3, "successor");

		add("id", 0);
		add("name", "zname");
		add("description", "zdesc");
	}
}
```

- Si no se desea modificar el constructor de la clase cada vez que se quiera añadir una relación se puede utilizar el patrón Abstract Factory o bien el Prototype. De esta manera se crearía la instancia y se añadirían externamente las relaciones deseadas.

```
class ModelFactory {
	public Entity createActivity()
	{
		Entity activity = new StandardEntity();

		activity.add(relation1, "activity");
		activity.add(relation2, "activity");
		activity.add(relation3, "successor");

		return activity;
	}
}
```

7.2.2.2 Asociación frente a Agregación

En el framework se encuentra en interfaz *Relation* de la cual se ofrecen dos implementaciones: *StandardRelation* y *Aggregation*.

Se utilizará una relación de agregación cuando uno de los dos objetos forme parte del estado del otro. Por tanto una relación será de agregación si se cumplen todas las condiciones siguientes:

- El agregado no debe seguir existiendo si el agregador se elimina.
- Al clonar el agregador el agregado debe ser también clonado, ya que forma su estado. Se clonarán también las relaciones que éste tenga con otros agregados del mismo agregador.
- Un objeto no puede tener varias relaciones de agregación en la que él tenga el rol de agregado.
- El grafo del modelo formado exclusivamente por las relaciones de agregación debe formar un árbol (no puede haber ciclos ni unión de ramas).

7.2.3 Método de Modelado de Operaciones

Las operaciones deberán implementarse siguiendo los siguientes criterios:

- Si afectan a más de una entidad se deberán implementar como macrooperaciones independientes en vez de repartir su código por las clases de todas ellas.

- Si se desea maximizar su reutilización a la hora de manipular las entidades deberán basarse únicamente en los miembros que necesitan sin poner ninguna restricción en función de su tipo.

```
class Operación
{
	void aplica(Entidad entidad)
	{
		entidad.get(...);
		entidad.set(...);

		otraEntidadRelacionada = entidad.get("dueño");
		otraEntidadRelacionada.set(...);
	}
}
```

- No deben incluir reglas estáticas que impliquen saber los estados válidos del modelo. Deberán sacarse a los monitores.

```
class Facturar
{
	void factura(Entidad cliente)
	{
		if (cliente.getList("facturaImpagada").size() < 3) // No
			entidad.set(...);

	}
}
```

- No deben cumplir realizar más de una actividad concreta en una parte del modelo mínima que conozcan. Si dichas acciones sobre el modelo deben tener consecuencias en otras partes nuevas del mismo deberá ampliarse la operación mediante reacciones.

```
void vender(Artículo articulo, int cantidad)
{
	articulo.quitarExistencias(cantidad);

	if (articulo.existencias() < articulo.nivelMinimo())
		hacerPedido();	// Esto es otra operación independiente
}
```

- No necesitarán ocuparse de reconstruir el modelo si algo falla durante la ejecución de la operación. Bastará con delegar en los reconstructores y en el ReconstructionManager.

7.2.4 Método de Modelado de Reglas

Las reglas del dominio se modelarán mediante dos tipos de objetos:

- Reacciones si se corresponden a reglas que extienden las acciones de las operaciones.
- Monitores si se corresponden a reglas que limitan las acciones de las operaciones.

A la hora de añadir una nueva entidad al modelo hay que decidir:

- Cómo *le afectan las operaciones existentes*, es decir, qué *reacciones* deben incorporarse y en qué sitios para que la nueva entidad sea considerada por las operaciones existentes. Por ejemplo indicar que se la borre cuando se borre tal otra entidad o relación.
- Cómo *afecta la nueva entidad a las operaciones existentes*, es decir, *qué monitores* deben incorporarse. La nueva entidad puede requerir que alguna operación existente deba considerar su estado para su ejecución. Por ejemplo se puede añadir sobre una relación ya existente alguna regla que impida su borrado si la nueva entidad no cumple ciertos requisitos.

La respuesta a estas dos últimas cuestiones dará lugar a las reacciones y monitores que hacen que el cambio en los tipos del modelo no afecte a los elementos existentes y sin embargo la nueva entidad *afecte y pueda ser afectada* por ellos. Las mismas operaciones de antes seguirán ejecutándose de la misma manera pero ahora estarán manipulando además a la nueva entidad o bien *estarán siendo validadas por ésta sin saberlo*.

7.2.5 Método para la Recuperación del Modelo por Incumplimiento de Reglas

La recuperación ante errores de ejecución no debe incluirse en las operaciones:

- Difícil de contemplar todas las posibles acciones a rehacer.
- Es demasiado código engorroso.
- Habrá que modificarlo a medida que se modifique el modelo.

Esta labor la realizarán los reconstructores de manera transparente dejando el modelo tal y como estaba antes de la operación fallida.

La razón que hace posible todo este mecanismo de recuperación automático y transparente es el Principio de Especialización. Al centrar la responsabilidad de mantener los datos del modelo en un tipo concreto de objetos *especializado en tal actividad* (las entidades) se puede gestionar la recuperación ocupándose únicamente de dichos objetos. Con el diseño tradicional orientado a objetos, al estar todo mezclado en distintas clases, *cualquier* clase podía tener datos que habría que reponer y cada una de ellas *con un interfaz distinto* para acceder a los mismos. No podía hacerse un mecanismo de reconstrucción *independiente* de las clases que formaran el modelo.

Esa es la razón por la que en la práctica generalmente ésta se obvia y no toda aplicación orientada a objetos está preparada para recibir una excepción en cualquier momento y poder continuar sin haber quedado corrompida (tolerancia a fallos). Y las que lo hacen es a costa de recargar el código con molestos bloques de gestión de excepciones y código de recuperación.

7.2.5.1 Creación de Reconstructores para clases externas a la Arquitectura

Si alguna operación se aplica sobre alguna entidad *que no haya sido implementada con las clases del framework RDM* (ya que éstas se reconstruyen automáticamente) deberá diseñar el reconstructor de dicha clase. Dicho reconstructor deberá ser añadido al hilo de reconstrucción (el cual mantiene el Reconstruction Manager) y la clase se podrá olvidar de cualquier problema posterior que pueda ocurrir durante la ejecución de la operación.

Por ejemplo la siguiente clase muestra una reacción muy simple que actualiza en tiempo real todos los cambios del modelo en una base de datos. Esta reacción necesita saber si la operación se anula para deshacer la transacción en progreso. Para ello se limita a escribir la entidad en la base de datos y añade un reconstructor que deshaga la transacción si algo falla sin que ella se entere. De esta manera tiene asegurado que la base de datos y el modelo en memoria siempre estarán sincronizados.

```
class GestorBD implements EntityListener
{
	public void memberSet(EntityEvent e)
	{
		<Guardar entidad modificada en la BD>
		ReconstructionManager.add(new UpdateBDReconstructor());
	// Éste reconstructor anula la transacción sobre la BD
	}
}
```

Pero es importante recordar que sólo habrá que crear nuevos reconstructores para recuperar repositorios de datos especiales que no hayan sido creados con el framework. Los modelos que se creen mediante el framework ya incluyen todos los reconstructores necesarios y no habrá que crearlos en cada aplicación.

7.3 Rendimiento de RDM

Como conclusión de RDM se analizará desde el punto de vista de su rendimiento tanto en ocupación de memoria como en velocidad de ejecución comparándolo con la implementación tradicional de una clase. Para realizar esta comparativa se ha utilizado el lenguaje Java.

7.3.1 Ocupación en Memoria

7.3.1.1 Almacenamiento de Atributos

Cada entidad de RDM tiene una tabla hash donde almacena sus miembros. Esta tabla inicialmente se crea con capacidad de cinco elementos y posteriormente irá creciendo dinámicamente a medida que se la vayan añadiendo más miembros. De la misma manera decrecerá si está sobredimensionada.

Por tanto donde en Java habría una clase con dos variables de instancia en RDM habrá una entidad con una sola variables de instancia: una tabla hash con dos elementos. Por tanto por cada entidad habrá una tabla hash adicional que anteriormente no había.

```
// Clase tradicional
class Persona
{
	String nombre;
	String dni;
}


// Equivalente con RDM
class StandardEntity implements Entity
{
	HashTable members = new HashTable(5); // guarda nombre y dni
	...
}
```

Esto en la práctica (al menos en las aplicaciones en las que ha sido utilizada) no ha supuesto un problema de excesiva utilización de memoria.

En cualquier caso si éste se presentara el framework se puede configurar para ajustarse a la memoria mínima. El framework tiene dos capas: los interfaces y la implementación. Por un lado se tienen los interfaces *Entity* y *Relation* y por otro lado se provee de unas implementaciones por defecto *StandardEntity* y *StandardRelation*. Sin embargo todas las clases se basan en los interfaces para comunicarse con los demás elementos del framework. Por tanto se pueden añadir otras implementaciones de dichos interfaces que tengan otra política de gestión de memoria más ajustada a las necesidades de la aplicación.

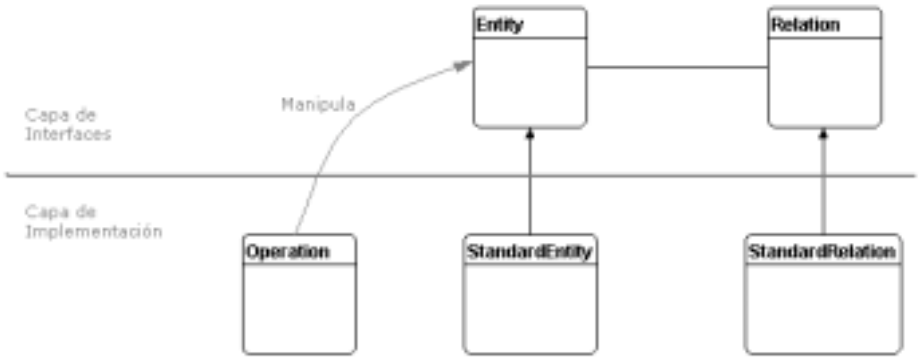

Imagen 7.3-1. Separación de interfaces e implementación en RDM.

Por ejemplo si se prefiere ajustar más la ocupación de memoria para una determinada clase podría hacerse de la siguiente manera:

```
class Actividad implements Entity
{
	String nombre;

	void set(String member, Object value)
	{
		if (member.equals(“nombre”))
			nombre = value;
	}

	...
}
```

Con lo cual tanto la versión con el framework como la versión tradicional ocuparían la misma memoria.

7.3.1.2 Creación de Reconstructores

Otra actividad que hay que estudiar respecto a la ocupación de memoria es la reconstrucción del modelo.

Cada modificación de una entidad supone un nuevo objeto reconstructor. Sin embargo esto no tiene que ser obligatoriamente así. Esta es la opción más sencilla y la que realiza la implementación por defecto de la entidad. Sin embargo sería sencillo cambiar esa política de tal manera que los objetos recontructores se reutilizaran en memoria. Formas habituales de hacer esto serían:

- Que cada entidad, en vez de crear un nuevo objeto cada vez, reutilizara el mismo reconstructor configurándolo cada vez para que reestablezca un miembro distinto. Por tanto en vez de crear un reconstructor en cada modificación se crearía uno por cada entidad que se encargaría de todas sus modificaciones.

- Sin embargo podría hacerse que simplemente hubiera *un solo reconstructor de cada tipo* de modificación que se utilizara para toda entidad. Por ejemplo habría un solo reconstructor que se encargaría de reponer los atributos. Por tanto toda entidad que se vea modificada añadiría su atributo a la lista que mantenga este reconstructor. Si finalmente éste se ejecuta repondrá todos los atributos de todas las entidades que se lo hayan solicitado.

Por tanto la gestión de las reconstrucciones se podría hacer con solo tres instancias de reconstructores:

- El que recupera atributos.

- El que recupera relaciones.

- Y el que recupera los suscriptores que se pudieran haber dado de baja.

Y nótese que esto tres objetos serían siempre los mismos para toda aplicación y *repondrían cualquier modelo sin conocerlo* liberando así a cada una de las operaciones.

7.3.1.3 Almacenamiento de las Relaciones

Mediante una implementación tradicional cada clase tendría su propio vector con sus elementos relacionados.

```
// Clase tradicional
class Persona
{
	// Un vector por cada relación
	Vector mascotas = new Vector();
	Vector amigos = new Vector();
	Vector coches = new Vector();
}
```

Por tanto con la implementación tradicional cada entidad estaría formada por dos objetos (uno sería la propia instancia y el otro la instancia agregada del vector).

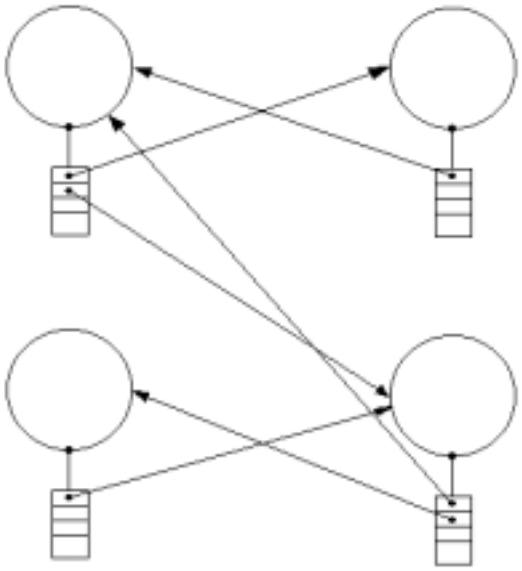

Imagen 7.3-2. Implementación tradicional de relaciones.

En RDM un solo objeto contenedor (el objeto *relación*) mantiene todas las ocurrencias de esa relación entre cualquier pareja de entidades. Lo que tiene cada instancia es una referencia a dicho objeto *relación* compartido por todas aquellas entidades a las cuales se les ha asignado dinámicamente dicha relación.

```
// Equivalente con RDM
class StandardEntity implements Entity
{
	HashTable relations; // Apunta a los objetos “Relation“
compartidos
	...
}
```

La implementación actual de RDM mantiene dentro de los objetos *relación* un contenedor para cada objeto (en el que guarda sus objetos relacionados). Por tanto se utilizan el mismo número de objetos que la implementación tradicional.

Sin embargo podría hacerse una implementación más optimizada de la clase StandardRelation que utilizara un sólo contenedor para guardar todas las parejas relacionadas. De esta manera el número de objetos se reduciría a la mitad al eliminar las instancias de la clase Vector[25]

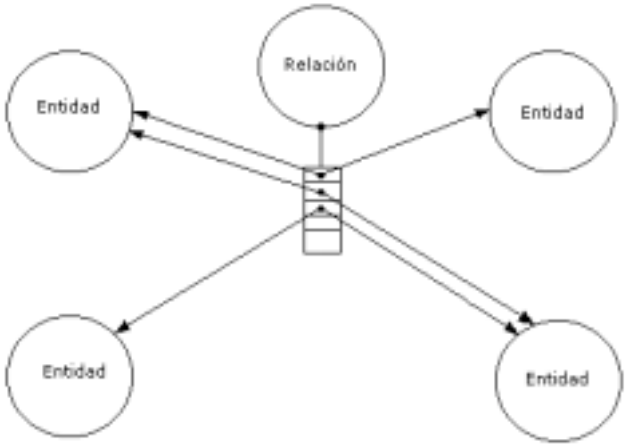

Imagen 7.3-3. Implementación optimizada de RDM

7.3.2 Velocidad de Ejecución

7.3.2.1 Acceso a miembros

A continuación se incluye una tabla comparativa de las operaciones de modificación y consulta de miembros utilizando RDM frente a la implementación tradicional mediante métodos estáticos. En ambos casos no se ha incluido ningún suscriptor ante los cambios de los objetos ya que éste factor sería igual en ambos.

Operación	**Acceso estático (ms)**	**Acceso en RDM (ms)**	**Razón**
Consultar miembro (get)	2640	90790	*34*
Modificar miembro (set)	13730	272980	*20*

Estos tiempos son meramente indicativos ya que influye tanto la velocidad del procesador como la versión del intérprete de Java utilizado[26]. Además la implementación actual del framework no

[25] La razón de no haberlo hecho así inicialmente ha sido para simplificar el código de la clase StandardRelation. Pero se podría cambiar esta implementación sin afectar al framework.

ha pasado por ninguna fase de optimización ya que ha primado más la sencillez de implementación y comprensión que la eficiencia.

Éstas diferencias, en la práctica, no han supuesto ningún inconveniente a la hora de utilizar el framework ya que, aunque lógicamente el acceso dinámico es más lento, es *suficientemente* rápido para que no sea un factor destacable comparado con aspectos como el acceso a bases de datos, dibujado en pantalla o comunicaciones que son los que representan el grueso del tiempo de ejecución.

Sin embargo si en algún caso concreto RDM fuera el cuello de botella de nuevo podría volver a aplicarse una implementación específica del interfaz Entity que permitiera acceso estático a ciertos atributos. En el siguiente ejemplo se permite en acceso dinámico pero además se da una segunda posibilidad más eficiente para que pueda ser utilizada por aquellas operaciones en las que prefiera sacrificarse la flexibilidad a costa de eficiencia.

```
class Actividad extends StandardEntity
{
	String nombre;

	// Acceso dinámico
	void set(String member, Object value)
	{
		if (member.equals("nombre")) {
			setName((String) value);
			return;
		}
		super(member, value);
	}

	// Acceso estático adicional para atributos "críticos"
	void setName(String value)
	{
		nombre = value;
	}
}
```

En este caso no habría diferencia en cuanto a la velocidad de acceso obtenido utilizando RDM o utilizando la programación tradicional. Sin embargo se seguirían obteniendo todas *las demás ventajas* de RDM.

Cuando RDM pasó a utilizarse en aplicaciones reales[27] se planificó hacer la versión optimizada cambiando las estructuras de datos y los algoritmos internos del framework. Sin embargo finalmente no se hizo ya que al sustituir modelos tradicionales por modelos RDM no se notó diferencia en el rendimiento. Por tanto no merecía la pena dedicar recursos humanos a mejorar algo que no iba a suponer una mejora apreciable en el producto final.

[26] Estos tiempos corresponden a ejecutar 10 millones de operaciones sobre el JDK 1.2.0 sobre un Pentium III 600 Mhz Mobile

[27] Una relación de dichas aplicaciones se incluye en el siguiente capítulo.

7.3.2.2 Reconstrucción

En el caso anterior se mostraba el comportamiento de RDM en el caso normal de ejecución de las operaciones. Pero cuando se produce el fallo de una operación hay que activar un proceso alternativo de recuperación que es la reconstrucción.

Aunque en la primera propuesta para la recuperación del modelo se hacía una copia completa del mismo para el caso de que hubiera que reponerlo, en la versión final basada en reconstructores se ejecutarán *las acciones mínimas necesarias* para dejar el modelo en el estado anterior ya que no se accede a *ninguna entidad ni ningún miembro* que no sea necesario reponer.

Por tanto la operación de reconstrucción no supone ninguna carga de ejecución *que no sea necesaria* y que no hubiera habido que hacer en una implementación tradicional.

7.3.3 Conclusiones sobre el rendimiento

En definitiva el framework no supone ninguna carga en cuanto a ocupación de memoria. Incluso en el caso de las relaciones el número de objetos en memoria se puede reducir a la mitad.

En cuanto a la velocidad de ejecución, aunque RDM supone un mayor tiempo de acceso a los miembros, en la práctica esto no supone ningún inconveniente como así ha demostrado su uso real. En cualquier caso se podría lograr fácilmente una velocidad equivalente a la que se obtiene mediante la implementación tradicional.

Sin embargo en lo que sí se obtiene una diferencia considerable a la hora de comparar ambas maneras de diseñar es en la *productividad* obtenida; la cual hubiera compensado las posibles desventajas que RDM hubiera tenido en tamaño o velocidad.

Sección 4. Demostración: Aplicación de RDM

Una vez que en la sección anterior ha quedado introducida la arquitectura y su implementación en esta sección hará la función de demostración de la tesis ya que muestra como se ha utilizado la arquitectura en un proyecto real y cómo se han evitado los problemas inicialmente expuestos en la tesis. Esta sección está dividida a su vez en cuatro capítulos:

- En el primer capítulo "El proyecto H24 de AENA" se describe el proyecto H24 de manera global. Aunque en capítulos posteriores sólo se entrará en detalle en la implementación de uno de sus módulos (el Editor de Procedimientos o EGP) en éste capítulo se da una breve visión global para poder situar el módulo en su contexto adecuado.

- Una vez comentada la situación del EGP dentro del proyecto global se presentará en el capítulo 9 "Los Requisitos del Editor Gráfico de Procesos (EGP)". Antes de entrar en los detalles de implementación que aparecerán en el capítulo 10 se hacía necesario un capítulo previo que mostrara al editor desde el punto de vista del usuario.

- Una vez finalizado el capítulo 9 ya se tiene una idea de lo que hay que contruir por lo que en el capítulo 10 se entre en los detalles de implementación que muestran la "Aplicación de RDM al EGP de Barajas" y las ventajas que se obtuvieron.

- El EGP es una de las aplicaciones construidas con la arquitectura RDM. En el capítulo 11, último capítulo de la sección dedicada a la aplicación de la arquitectura, se mostrará una breve enumeración de otros proyectos en los que ha sido utilizada. Se pretende así mostrar que RDM no es una Arquitectura teórica sino que tiene asociada una implementación práctica y aplicable (y aplicada) en aplicaciones profesionales.

8 Proyecto H24 de AENA

8.1 Introducción

Después de haber visto la Arquitectura RDM y su implementación en un framework, en este capítulo comienza la sección dedicada a la *Aplicación de la Arquitectura.* Se mostrará cómo se ha aplicado RDM en un proyecto real y por tanto correspondería a la *demostración* de la tesis ya que muestra cómo la arquitectura RDM cumple los objetivos propuestos y cómo se han evitado los problemas inicialmente expuestos.

En el capítulo actual se describirá el proyecto H24 del aeropuerto Madrid-Barajas para dar una perspectiva global del mismo. Aunque en esta tesis sólo se va a entrar a detallar la implementación de uno de sus módulos (el Editor de Procedimientos o EGP) se necesita previamente una visión global del proyecto en el que está enmarcado.

8.2 Visión General del proyecto

Este capítulo describe la aplicación de la arquitectura RDM en el Proyecto H24, expediente Nº: 1893/01 titulado "Desarrollo de una Herramienta de Ejecución y Seguimiento de Procesos en Tiempo Real" desarrollado por Comercio Electrónico B2B 2000 S.A. para AENA – Madrid Barajas.

Uno de los mayores problemas del aeropuerto es la definición exacta y el control de todos sus procedimientos.

- La documentación sobre los procedimientos no estaba en un formato unificado e incluso ciertos pasos se tomaban más por la experiencia del responsable que por el hecho de que estuvieran documentados.

- La documentación de tales procedimientos solía estar separada de su ejecución, por lo que, salvo excepciones, no resulta adecuada para el día-a-día.

Se pretendía documentar de manera homogénea todos los procedimientos a seguir ante situaciones de contingencia. Por tanto se creó un lenguaje XML de descripción de procesos y procedimientos aeroportuarios especialmente ajustado a la casuística del Aeropuerto de Madrid-Barajas. A dicho lenguaje se le denominó MADML (MAD[28] Markup Language).

La razón de la adopción del lenguaje XML fue debido a sus ventajas:

[28] MAD es el mnemotécnico asignado a dicho aeropuerto dentro de AENA.

- A la hora de que los procedimientos fueran manipulados por distintas aplicaciones

- A la sencillez en la transformación de los procedimientos a otros formatos (HTML, PDF, etc.)

Un documento codificado usando MADML constituye, en sí mismo, la más completa documentación del procedimiento aeroportuario que describe. MADML es un lenguaje ejecutable. La "descripción de un procedimiento" es a la vez *legible* por humanos y *procesable* por un ordenador.

Por tanto se consigue que todo cambio en un procedimiento implique:

- La modificación automática de la documentación

- Su actualización automática en ejecución.

Las funciones principales del MADML son:

- Definir un conjunto de actividades organizadas en secuencia y/o en paralelo.

- Asignación de responsables a dichas actividades.

- Registro de suscriptores a la evolución de las actividades (tanto personas como otros procedimientos).

- Coordinación entre procedimientos de distintos áreas del aeropuerto.

A continuación se muestra como ejemplo un fragmento del procedimiento de "Baja visibilidad en plataforma" codificado en MADML:

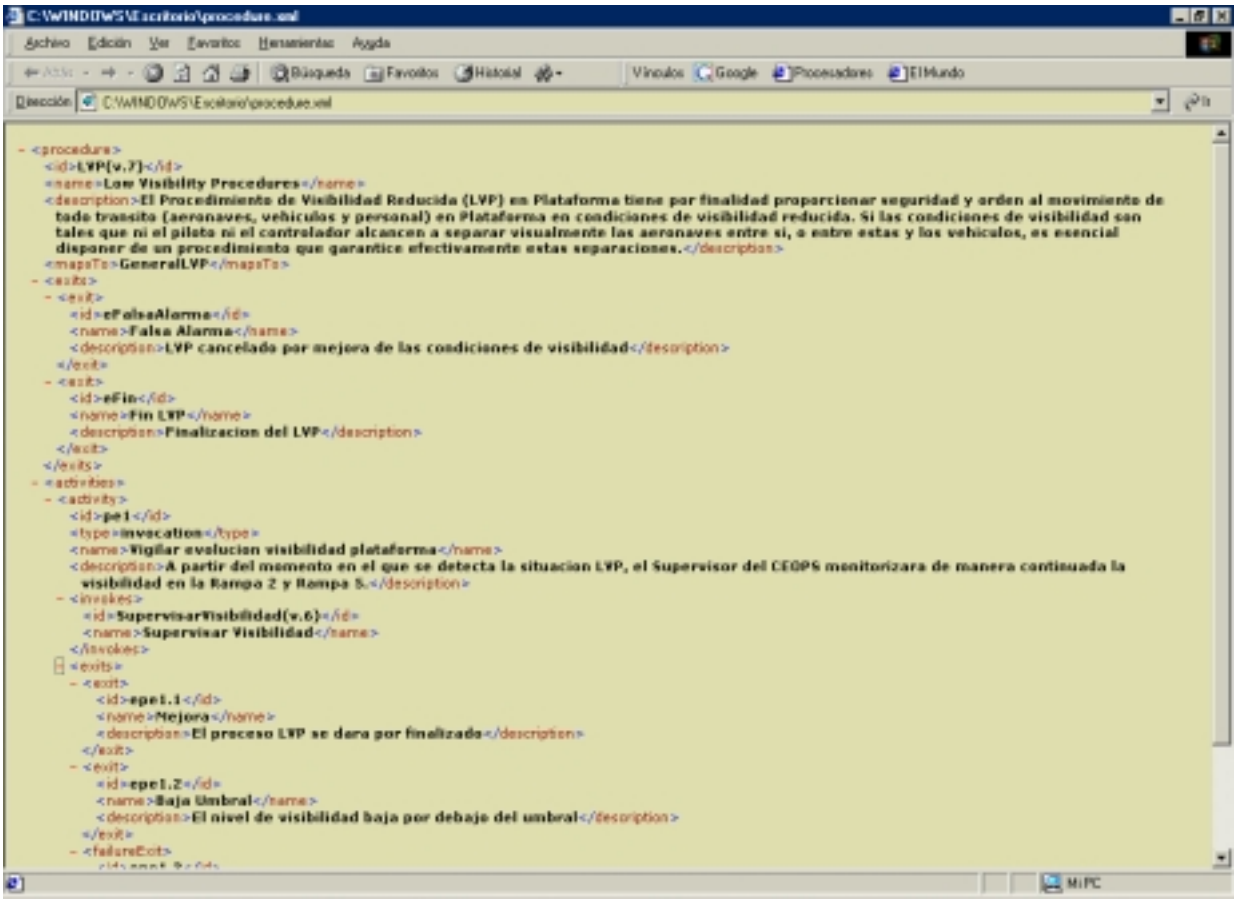

Imagen 8.2-1. Ejemplo de procedimiento en MADML

El diseño de este lenguaje rápidamente requirió de nuevos módulos en el proyecto:

- El Lenguaje MADML, aunque flexible y sencillo para las labores que desempeña, no es trivial para confeccionarlo a mano. Los documentos son tediosos de elaborar y caben en ellos errores de trascripción. Por tanto se requería un editor gráfico de procedimientos (EGP).

- El lenguaje MADML no es directamente ejecutable (no es un lenguaje de programación). Por tanto se requiere un módulo que entienda dicho lenguaje y realice los procedimientos: el Motor de Procedimientos.

- La ejecución de un procedimientos no puede realizarse en una caja negra que no permita su monitorización. Se necesita una forma de hacer un seguimiento de los procedimientos ejecutados y en ejecución. Por tanto hacía falta un Visor de Instancias de Procedimientos.

En los siguientes apartados se da una breve descripción de cada uno de estos tres elementos.

8.2.1.1 El Editor Gráfico de Procedimientos (EGP)

El Editor Gráfico de Procedimientos es una herramienta visual que permite representar los distintos elementos de un procedimiento de manera fácil e intuitiva y con distintos niveles de detalle:

- Permite insertar actividades, condiciones de finalización de actividades y la edición de los atributos de ambos.
- La ejecución de las actividades podrá ser secuencial o concurrente.
- Permite invocaciones a otros procedimientos
- Se deben poder determinar las acciones ante situaciones de error en tiempo de ejecución. Un procedimiento puede decidir finalizar o bien recuperarse y continuar la ejecución por otra rama.
- En necesaria la edición simultánea de varios procedimientos y las operaciones de copiado de fragmentos entre ellos.

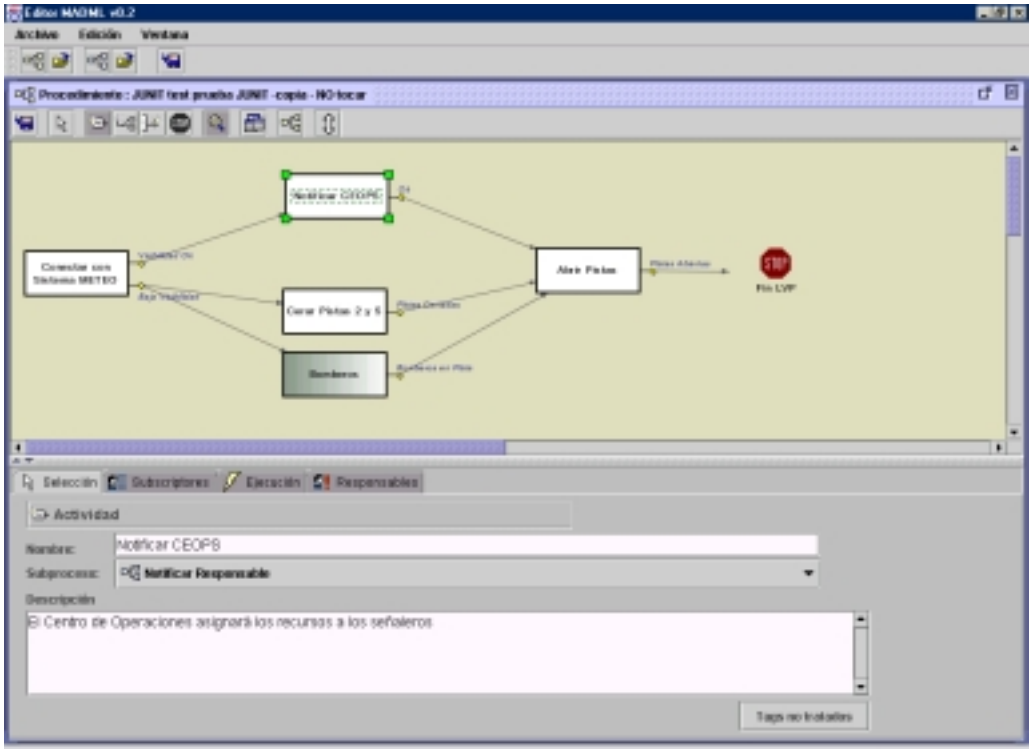

Imagen 8.2-2. Elementos gráficos de un procedimiento.

En la imagen anterior puede verse un procedimiento formado por cinco actividades y un terminador. La primera actividad tiene dos formas de finalización. Si acaba de la primera forma pasará a ejecutarse la actividad “Notificar CEOPS”. En caso de acabar por la segunda (“Baja Visibilidad”) entonces se ejecutarán en paralelo dos actividades, una de las cuales se corresponde con la invocación del procedimiento de los Bomberos ante baja visibilidad.

Finalmente la actividad “Abrir Pistas” se ejecutará cuando acabe la actividad “Notificar CEOS” o bien acaben las otras dos actividades (ambas).

Una vez dibujado el procedimiento el editor generará automáticamente su descripción en MADML.

Este será el módulo elegido en el capítulo siguiente para mostrar el uso del framework en su construcción.

8.2.1.2 El Motor de Ejecución

El Motor de Ejecución se encarga de la ejecución de los procedimientos. Éste puede recibir de distintas fuentes órdenes de ejecución de procesos y automáticamente pasará a ejecutarlos. El motor permite la ejecución concurrente de cualquier número de procedimientos o de varias instancias del mismo procedimiento.

Además genera información sobre la ejecución de cada una de las actividades en tiempo real. Se puede conocer así la hora de comienzo y finalización del procedimiento y cada una de sus actividades así como sus detalles e incidentes durante la ejecución de los mismos.

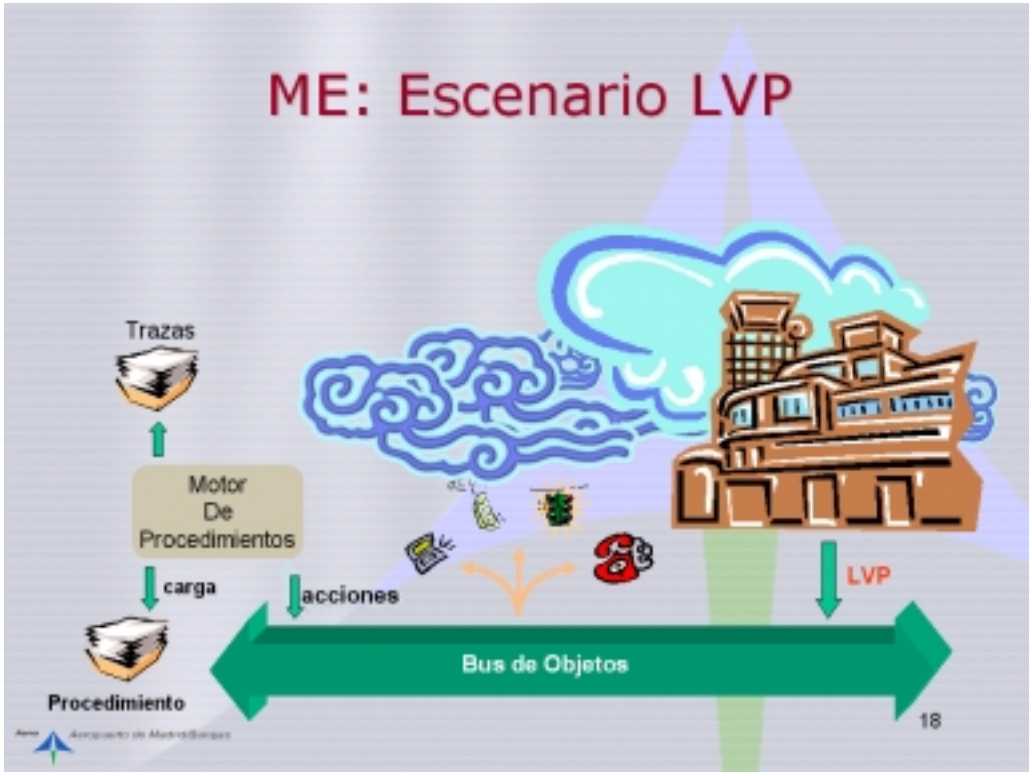

Imagen 8.2-3. El Bus de Objetos como mecanismos de comunicación de sistemas de información.

En la figura anterior aparece un elemento del proyecto denominado el Bus de Objetos. El Bus es un broker de mensajes que permite la comunicación entre todos los sistemas del aeropuerto.

El escenario de la figura se realiza con la siguiente secuencia de pasos:

- La torre de control avisa de que hay baja visibilidad por niebla (LVP). Deposita dicha notificación en el bus.

- El Motor de Procedimientos carga del repositorio el procedimiento a ejecutar ante dicha circunstancia (el cual fue creado con el editor) y comienza la ejecución de una instancia del mismo.

- La ejecución de las distintas actividades del procedimiento producirán órdenes que se transmitirán a través del bus al destinatario adecuado (sea persona u otro sistema).

- Durante la ejecución el motor además almacenará en un repositorio las distintas incidencias de la ejecución de cada instancia.

8.2.1.3 El Visor de Procedimientos

El Motor genera la traza de un procedimiento en otro lenguaje XML. El Visor de Procedimientos (que más correctamente debía haberse denominado Visor de Trazas de Ejecución de Instancias de Procedimientos) permite la visualización gráfica en tiempo real de dicho lenguaje.

Fundamentalmente mostrará:

- A qué hora comenzarón y finalizaron el procedimiento y cada una de sus actividades.

- Qué camino se tomó de las distintas ramas del procedimiento y por qué.

- En caso de error de ejecución saber los detalles del mismo y como se manejó.

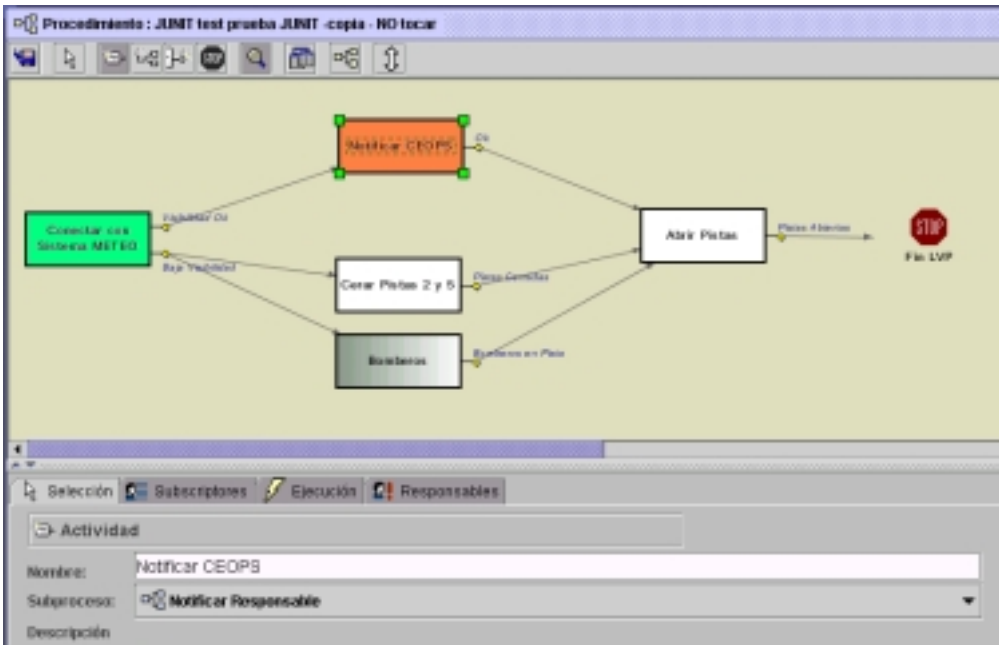

Imagen 8.2-4. Traza de ejecución de un procedimiento.

En la imagen anterior se puede ver un ejemplo de una ejecución de un procedimiento. El significado de los colores de las actividades es:

- Si está en *verde* es que la actividad ha acabado correctamente. El visor mostrará como propiedades de la actividad la hora de inicio, de finalización y la salida que ha tomado la actividad.

- Si está en *naranja* es que la actividad se está ejecutando en estos momentos. En la traza solo estará disponible la hora de inicio (además de las demás propiedades asignadas con el editor como el responsable, descripción, etc.)

- Si la actividad está en *rojo* es que hubo un error en su ejecución y se explicará la causa del error. Si se estableció una vía de recuperación se podrá comprobar en el visor que el flujo ha continuado por el camino establecido.

8.2.2 Conclusiones

El proyecto H24 tiene como objetivo la documentación y ejecución automatizada de los procesos y procedimientos del aeropuerto de Madrid Barajas.

El elemento central del proyecto es el lenguaje MADML en el cual se describen los procedimientos. En torno a este lenguaje surgen módulos adicionales para su tratamiento automatizado como son el Editor Gráfico de Procedimientos, el Motor de Ejecución y el Visor de Procedimientos. Todos ellos se comunican entre sí y con otros sistemas del aeropuerto a través del Bus de Objetos, el cual es un middleware con arquitectura de broker de mensajes adaptado a las características del aeropuerto.

En este capítulo simplemente se han explicado brevemente algunos de los módulos del proyecto H24 ya que el proyecto global incluye además temas prolijos como son el Control de Calidad, Reingeniería de Procesos, Gestión de Notificaciones, etc. que se han omitido ya que el objetivo no era dar una visión exahustiva del proyecto que alejara del tema de éste documento.

9 Requisitos del Editor Gráfico de Procesos (EGP)

En el apartado anterior se mostró una visión global del proyecto H24. A partir de ahora se entrará en más detalle en el Editor Gráfico de Procesos (EGP) para mostrar por qué y cómo se aplicó el framework RDM. Los demás módulos del proyecto también utilizaron dicho framework pero en este documento solo se hablará del EGP ya que la descripción del diseño de todos los módulos del proyecto sería demasiado extensa para ser incluida en esta tesis.

Pero antes de entrar en la implementación del mismo (lo cual se hará en el próximo capítulo) se mostrarán brevemente los requisitos del editor para tener una referencia más clara del módulo cuya implementación se describe en el siguiente capítulo.

9.1 Suscripciones

El EGP debe permitir subscripciones a actividades de tal manera que una persona pueda ser notificada al inicio o final de una actividad. Esto se utiliza cuando haya personas que, aunque no estén implicadas directamente en la ejecución del procedimiento, necesiten saber de la ejecución de ciertas actividades. Un ejemplo sería un directivo interesado en llevar una estadística de con qué frecuencia se realizan ciertas actividades para en función de eso optimizar o ajustar los recursos de los procedimientos.

En definitiva se debe permitir asociar actividades y personas mediante una relación de *notificación*.

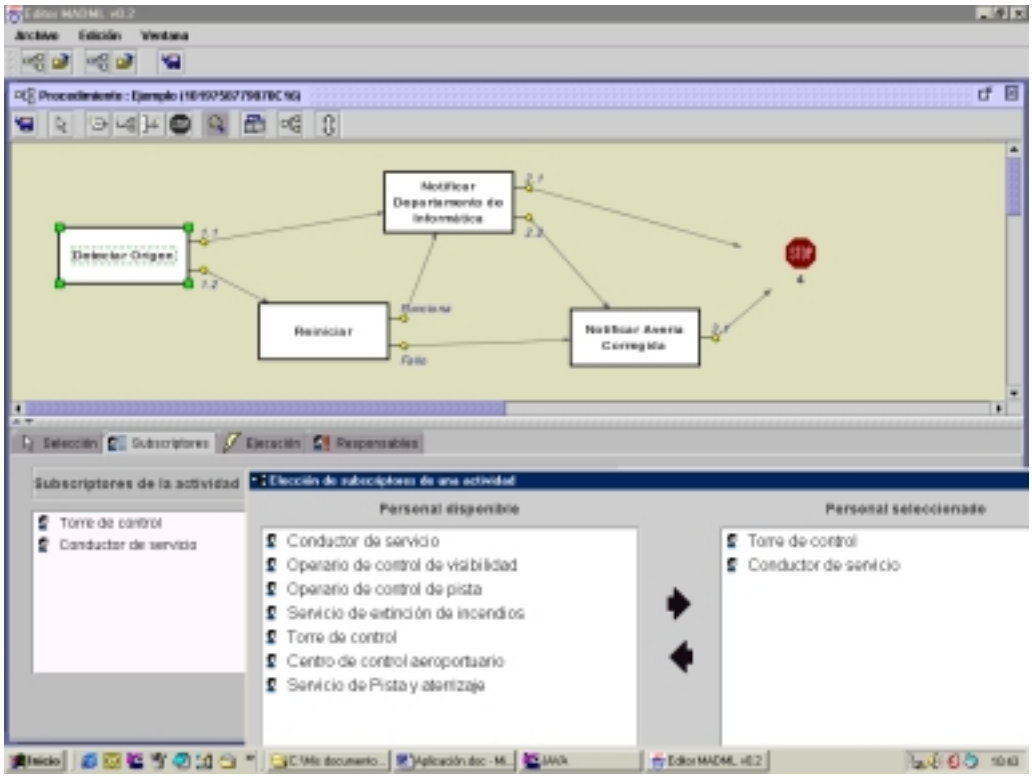

Imagen 9.1-1. Asignación de suscriptores a una actividad.

En el ejemplo anterior puede verse que cuando se ejecute la actividad “Detectar Origen” se notificará a la “Torre de Control” y al “Conductor de Servicio”.

9.2 Variación de las Reglas de Ejecución

Los procedimientos, obviamente, no son una serie de acciones fijas que se puedan codificar en Java y no vayan a cambiar. Precisamente la labor del EGP es permitir la edición de procedimientos. Sin embargo el EGP trata los procedimientos con dos objetivos.

- Por un lado como una documentación de los pasos que se llevan a cabo ante una situación de contingencia.

- Opcionalmente como la secuencia de pasos a ejecutar por un ordenador cuando ocurre dicha contingencia.

Todo procedimiento debe estar documentado. Pero no todos son *ejecutables*. Por ejemplo el procedimiento de conducción de pasajeros puede describirse (facturar, embarcar, etc) pero no puede *instanciarse* ni pueden aplicarse acciones sobre las entidades que en él participan (en este caso pasajeros) ya que éstas no se van a modelar individualmente dentro del ordenador.

Sin embargo hay otros procedimientos (como por ejemplo el Procedimiento de Baja Visibilidad – LVP) que sí son ejecutables por un ordenador. En este tipo de procedimientos debe permitirse al usuario que introduzca las acciones a realizar (pedir la visibilidad al sistema meteorológico, avisar a los bomberos, etc).

Por tanto en el EGP se puede describir el procedimiento (el *qué* hacer) y opcionalmente, en aquellos procedimientos automatizables, asignar a cada actividad el *cómo* hacerlo mediante un script. El lenguaje de script es también un lenguaje visual que se ha intentando hacer lo más parecido posible al del procedimiento para simplificar su uso.

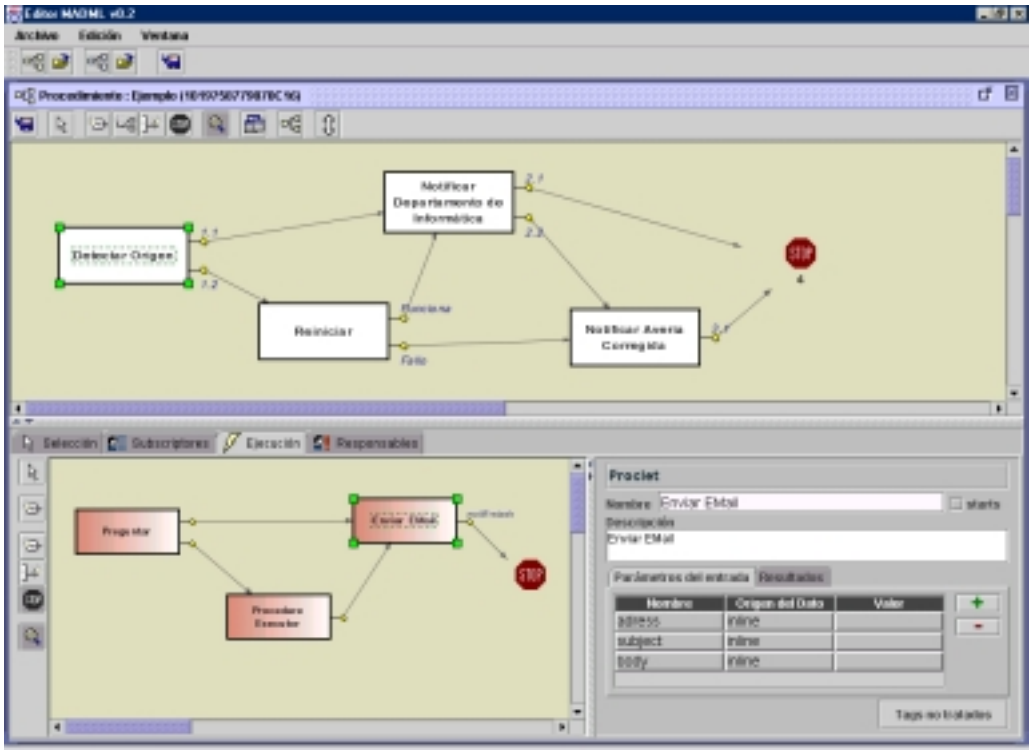

Imagen 9.2-1. Actividad y su script asociado.

En la figura anterior puede verse el procedimiento en la parte superior. Al seleccionar la actividad "Detectar Origen" aparecen en la mitad inferior sus distintas vistas de entre las cuales está seleccionada la *ejecución* de esta actividad, es decir, cual es su script, por lo que aparecerá un editor con sus propias herramientas.

El editor incorpora un modelo de componentes derivados de los JavaBeans denominados *proclets*. El script está formado entonces por un conjunto de componentes (o *proclets*) que se insertan en el mismo para formar el flujo de ejecución. En este caso se encuentra seleccionado el proclet de enviar un mail y a su derecha se muestran las propiedades de dicho componente.

Un aspecto importante del lenguaje de script es que también es un lenguaje XML. Se incrusta en el MADML como una etiqueta más de cada actividad. Más que un lenguaje de programación parece un nivel mayor de detalle de lo que hace una actividad. Es a la vez un *qué* hacer y un *cómo* hacerlo.

En definitiva con el EGP se pueden modelar tres tipos de procedimientos:

- Documentar un procedimiento que es llevado a cabo fuera del dominio informático. En este caso el procedimiento se utilizará para su lectura por las personas que deban llevarlas a cabo por lo que ninguna actividad tendrá un script.

- Documentar las actividades y además, mediante un script, describir las acciones concretas a realizar en cada actividad. Dichas acciones (tomar datos de otros sistemas, transformarlos, tomar decisiones, etc.) se llevarán a cabo por el motor de forma automatizada.

- Procedimientos mixtos. Estos procedimientos tienen actividades que pueden ser automatizadas y otras que deben ser realizadas por personas. El primer tipo de actividades serán aquellas que tengan un script. En este caso el motor actuará como un coordinador que mande realizar actividades a ciertas personas y tomará sus resultados mientras que él mismo realizará aquellas que tengan un script.

9.3 Responsables

Cada actividad debe tener un responsable. Sin embargo, a diferencia de los suscriptores, la asignación de responsables a una actividad no es añadir una mera relación. Aunque una actividad solo puede tener un responsable puede tener varios candidatos ya que la asignación de responsables es dinámica. Por ejemplo la responsabilidad de una actividad puede ser asignada en función de la hora en la que se ejecute la actividad.

Por tanto la actividad puede estar asociada a un responsable o a un algoritmo que devuelva un responsable. Este algoritmo debe poder ser leído pero debe poder también poder ser ejecutado, ya que en tiempo de ejecución puede que sea necesario saber quién es el responsable por el script de la actividad (por ejemplo para poder enviarle una notificación).

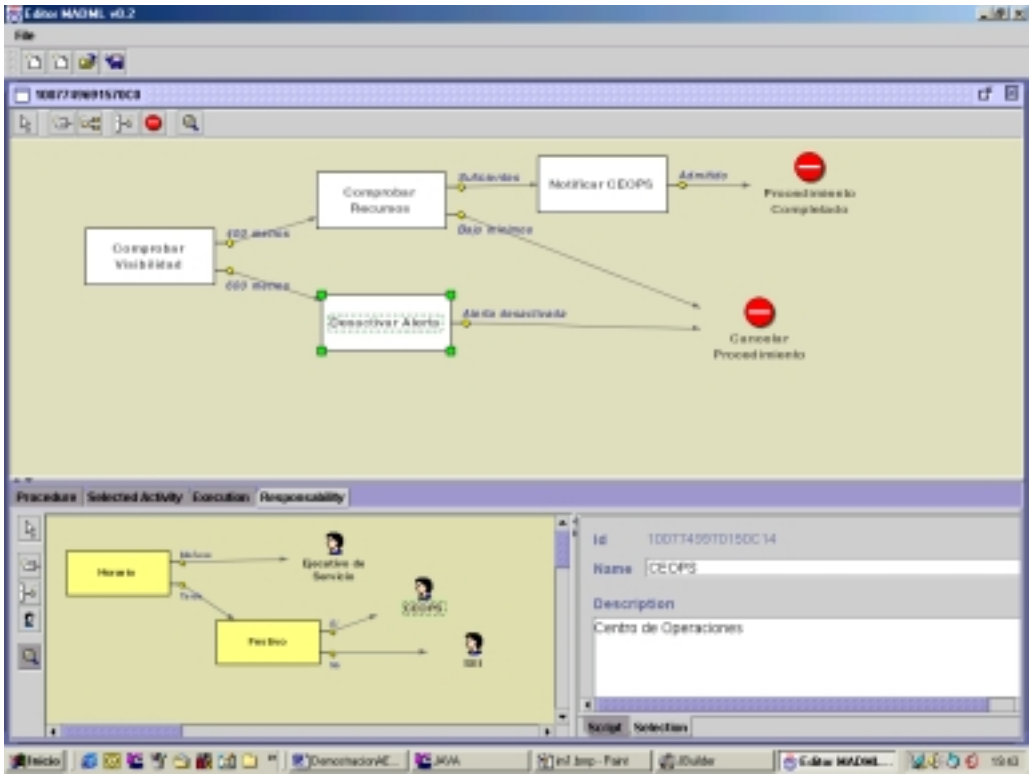

Imagen 9.3-1. Asignación de responsables a una actividad.

9.4 Adición Dinámica de Entidades y Propiedades

El lenguaje MADML tuvo un problema importante en su diseño: descubrir toda la información que sería necesaria incluir sobre todo procedimiento para conseguir que el lenguaje fuera estable. Esa estabilidad era necesaria para no tener que alterar los distintos módulos del proyecto (EGP, motor, visor y otros no descritos aquí) *cada vez que se cambiara el mismo*. Pero bastó implementar un núcleo básico del MADML para ver que dicha estabilidad era imposible.

La razón era que la nueva codificación de los procedimientos debía ser tal que aunque homogeneizara la descripción de los mismos pudieran incluirse además atributos específicos a la naturaleza de cada procedimiento. Esto debía ser así ya que no había un conjunto de atributos comunes a todas las actividades. La información que había que incorporar en el lenguaje dependía de cada actividad (ni siquiera las de un mismo tipo tenían que tener todas la misma información), dependían del procedimiento, de su contexto e incluso, en el caso del lenguaje de trazas, del estado de otros elementos del aeropuerto en el momento de la ejecución.

¿Por qué es tan importante permitir añadir cualquier tipo de información? Porque no se puede permitir que un procedimiento no se pueda documentar porque no se ha pensado un campo para poner cierta característica del mismo. O, aún peor, que un empleado no pueda anotar en una traza que ocurrieron determinados incidentes durante la ejecución de una actividad porque no se hubiera pensado unos campos para describir dichos incidentes (y en un aeropuerto son

muchísimos los incidentes que pueden ocurrir). Son tantas las entidades y las situaciones que no se puede limitar la información a una serie de campos predefinidos. Era necesario que cada uno pudiera añadir la información necesaria para describir de la forma más exacta y útil la realidad. Cuando la cantidad y tipo de información es tan grande y variable se requiere un sistema *flexible* de información.

¿Y poner un campo genérico llamado *observaciones* para cada entidad? Esa es la solución adoptada habitualmente. El problema es que entonces los datos no serían procesables. Es decir, no se podrían hacer búsquedas del tipo "lista todos las entidades con tal atributo...". El nuevo dato debe estar localizable como otro dato cualquiera. Es decir, los datos deben poder ser accedidos de manera uniforme sin depender de si el dato fue identificado inicialmente en el análisis o no. Si los atributos que fueron encontrados en el análisis tienen su propio campo y los demás están mezclados en *observaciones* con el tiempo dicho campo será un sumidero de datos de segunda categoría. Además las operaciones tendrían que saber si dicho atributo se identificó en el análisis o no para saber donde buscar el dato. No debería ser así. Hay que poder acceder a todos los datos de manera uniforme. Por ejemplo el Gestor de Calidad[29] apunta en cada actividad el criterio de calidad a seguir en la ejecución de cada actividad. Esta información se debe añadir a la misma como cualquier otro atributo para que dicho módulo pueda localizarlo (y cualquier otro servicio que lo quiera utilizar).

En definitiva cualquier persona o servicio debe poder añadir su propia información para poder ser procesada por ellos mismos o por otros servicios. Podría considerarse como análisis *bajo demanda*, de tal manera que no hay que conjeturar que atributos van a hacer falta antes de hacer las herramientas sino que se añadan los que *ya* hagan falta a medida que se necesiten y de manera homogénea a los que ya estén.

Por tanto se necesita:

- Que el editor permita la introducción de nuevas entidades, atributos y relaciones en tiempo de ejecución para que no haya que modificar las clases del modelo.

- Que dichos cambios no afecten a los módulos construidos (motor, visor, etc).

El Bus de Objetos permite el acceso a las entidades del aeropuerto. El tipo de entidades accesibles y las propiedades de cada una de ellas depende de los sistemas que los publiquen al bus. El editor debe permitir relacionar entidades accesibles a través del bus sin conocerlas y permitir establecer entre ellas nuevos tipos de relaciones.

[29] Otro módulo no descrito del proyecto H24 encargado, como su propio nombre indica, de estimar la calidad con la que se están ejecutando los procedimientos en el aeropuerto.

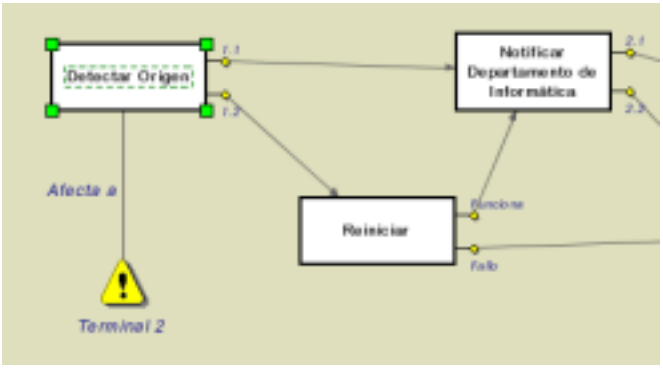

Imagen 9.4-1. Relación entre entidades aeroportuarias del bus.

En la imagen anterior puede verse el resultado de la siguiente secuencia de pasos[30]:

- El usuario indica al editor que quiere incluir en el procedimiento una entidad del aeropuerto. Concretamente la Terminal 2.

- El editor accede a dicha entidad a través del bus y recibe sus propiedades en XML. Basándose en esta información crea dinámicamente un objeto entidad que lo represente y lo inserta en el modelo (además de crear una vista para dicho objeto).

- A continuación el usuario indica que quiere establecer una relación entre la primera actividad y la Terminal. Esta relación no tiene por qué haber sido prevista inicialmente y no tiene por qué estar en más actividades. Puede que sea como *mera documentación* o que un nuevo proceso que se vaya a implementar *vaya a utilizar* dicha relación.

Por tanto el modelo debe de ser suficientemente flexible para soportar todo esto sin necesidad de recompilar las clases del modelo.

9.5 Creación de Procesos: la Clase del Procedimiento

A la hora de modelar los procedimientos se comprobó como algunos de ellos compartían características similares. Tenían la misma secuencia de pasos y las mismas propiedades. Por ejemplo todos los procedimientos de resolución de averías tenían las misma secuencia de pasos: informar del fallo, detectar origen, notificar responsable, evaluar impacto, etc. Sin embargo todos estos procedimientos *refinaban* estos pasos de tal manera que en el caso de una avería informática la actividad de notificar responsable se *redefinía* mediante una actividad consistente en localizar al desarrollador de dicha aplicación.

[30] La figura mostrada es un fotomontaje ya que esta característica todavía no estaba desarrollada en el momento de hacer este documento.

Por tanto hacía falta un mecanismo que permitiera:

- Establecer la secuencia de pasos general para una familia de procedimientos. Por ejemplo para crear un nuevo procedimiento de avería se ofrece así la mayor parte del trabajo hecho y, sobre todo, se evita el olvidar alguno de los pasos al modelar el procedimiento.

- Poder tener identificados todos los procedimientos de un tipo de manera que si cambia el proceso se sepa a qué procedimientos afecta.

Por tanto hacía falta el concepto de *tipo* de procedimiento, al cual se denomina *proceso*. Es decir, el *procedimiento* de "Solución de averías informáticas" es una instancia del *proceso* "Solución de averías". Por tanto el editor debía permitir

- Crear nuevos tipos que incluyeran las características comunes a los procedimientos de dicho tipo (actividades, scripts, responsables, propiedades específicas del tipo, etc.). Además de que el proceso sea el tipo del procedimiento las actividades de un proceso son *el tipo de las actividades* de sus procedimientos.

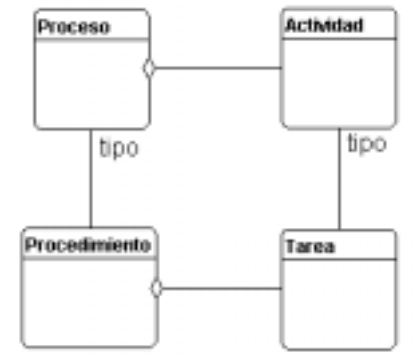

Imagen 9.5-1. Si un proceso define un tipo para los procedimientos, las actividades del primero son un tipo para las de los segundos.

- Se debe poder *instanciar* procedimientos a partir de un tipo. De esta manera aparecen ya las principales actividades de dicho tipo pero el procedimiento podrá redefinirlas o añadir otras actividades adicionales.

- Puede haber procedimientos que no tengan tipo; no es obligatorio para crearlos. Sin embargo si a medida que se van haciendo procedimientos se detecta la aparición de un nuevo tipo podrá crearse un proceso y *permitir la asignación dinámica de tipos* a los procedimientos ya existentes (y a cada una de sus actividades individualmente).

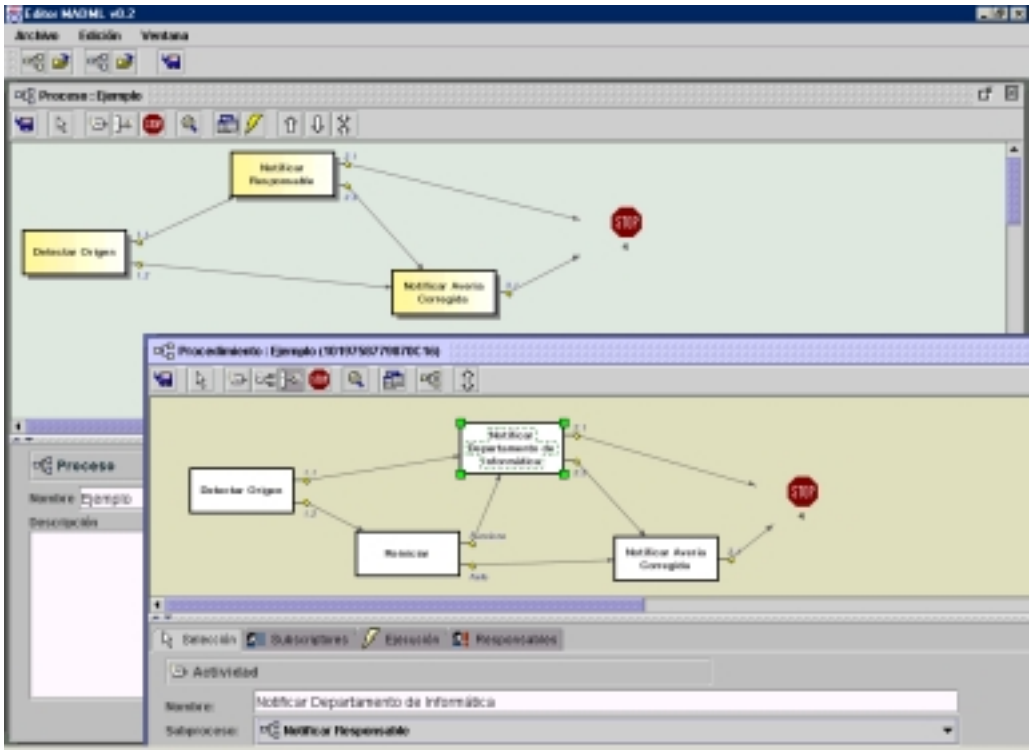

Imagen 9.5-2. Un proceso y un procedimiento de dicho tipo

En la imagen anterior puede observarse en la parte superior el proceso general para solucionar averías. En la parte inferior puede verse el procedimiento concreto para solucionar averías en sistemas informáticos[31].

El procedimiento tiene como tipo al proceso y puede verse también en la parte inferior que la actividad seleccionada "Notificar Departamento de Informática" es de tipo "Notificar Responsable" (actividad que aparece en el proceso).

Puede verse también que el procedimiento puede añadir nuevas actividades específicas del procedimiento (actividad "Reiniciar") que no tienen por qué estar contempladas en su tipo. Es decir, un tipo debe proporcionar una guía para una serie de pasos comunes, pero no debe limitar las actividades a realizar de un procedimiento.

9.6 Conclusiones. La Arquitectura RDM en el Proyecto H24

Por lo que se ha visto hasta el momento el editor comparte las características generales de los modelos de objetos adaptables:

[31] Este proceso y este procedimiento no son los reales ya que éstos serían demasiado extensos.

- Creación dinámica de tipos. Se crean tipos (procesos) e instancias de dichos tipos (procedimientos).

- Adición dinámica de propiedades a las entidades. Tanto a las actividades, a las exits, etc, se les puede añadir nuevos atributos

- Adición dinámica relaciones con otras entidades del aeropuerto.

- Modificación de las reglas de ejecución de los procedimientos.

El autor Joe Yoder (en [Yoder01]) afirma que los Adaptable Object-Models se caracterizan por:

1. Ser adecuados cuando haya muchos cambios en fase de explotación.

2. Tienen altos costes de introducción.

3. Requiere recursos humanos con altos conocimientos.

4. Demanda herramientas específicas para el desarrollo.

Sin embargo en el proyecto H24:

1. Aunque se preveían muchos cambios durante el desarrollo (el lenguaje pasó por distintas fases de refinamiento) una vez establecido los módulos no iban a requerir cambios. En cualquier caso cualquier cambio que hubiera que hacer en el editor o en el motor el usuario no iba a ser capaz de introducirlo. Era más práctico que lo hiciera un programador.

2. Se tenían tres meses de plazo.

3. El equipo de desarrollo no era conocedor de las técnicas adaptativas y apenas tenía experiencia en patrones.

4. Se debían utilizar herramientas estándar ya que no había tiempo para implementar un lenguaje y su entorno (intérprete, depurador, etc.)

Además tampoco se requería:

- Un modelo basado en Bases de Datos. Los procedimientos se editan en memoria y a la hora de transferirlos se utiliza un middleware con XML.

- No se requiere generación de interfaz de usuario basada en formularios ni generada automáticamente a partir de metadatos. La usabilidad era un requisito fundamental de este proyecto y había que crear los interfaces con herramientas de desarrollo adecuadas.

Por todo ello lo que se requería era una arquitectura que permitiera implementar lo más rápidamente posible las tres aplicaciones y que permitiera que los cambios en el lenguaje (es decir, el dominio) se implementarán lo más rápidamente posible. RDM ofrecía:

- Modelar inmediatamente nuevos elementos del lenguaje y sus relaciones.

- Los cambios en el modelo no afectan a las operaciones existentes ni las nuevas operaciones existentes afectan al modelo.

- Reutilizar operaciones y vistas con nuevos elementos del modelo.

Por tanto la arquitectura final del editor se basó en dos frameworks:

- RDM para representar el modelo y las operaciones.

- JHotDraw [Johnson92] para implementar las vistas.

Estos dos frameworks encajaban perfectamente ya que ahora implementar cualquier elemento era cuestión de minutos.

Este capítulo ha supuesto una introducción a la funcionalidad del EGP. En el siguiente capítulo se mostrará algunos ejemplos de cómo se ha utilizado RDM en su implementación. Además en el Apéndice B se muestra el manual de usuario del mismo.

10 Aplicación de RDM

Una vez vistos la arquitectura RDM y los requisitos a cumplir por el Editor Gráfico de Procesos (EGP) en éste capítulo se mostrarán algunos ejemplos de cómo se usó el framework para la construcción del editor. Esto será una demostración de su uso en una aplicación real y un ejemplo de todas las ventajas que ofrece.

Una vez mostrado como se ha aplicado RDM en el proyecto se concluirá con que éste supone una demostración de la tesis presentada.

10.1 RDM en el EGP del Proyecto H24

A continuación se describe como se usó RDM en el EGP:

- Para crear el modelo del Editor de Procedimientos.
- Reutilizar un modelo que ya se tenía en otra aplicación.
- Cómo se reutilizaron las vistas para distintos modelos.
- Cómo permite reutilizar operaciones existentes y además crearlas de forma que se maximicen sus posibilidades de reutilización en otros proyectos.

10.1.1 Creación del Modelo

En la siguiente figura se muestra un extracto del diseño del EGP.

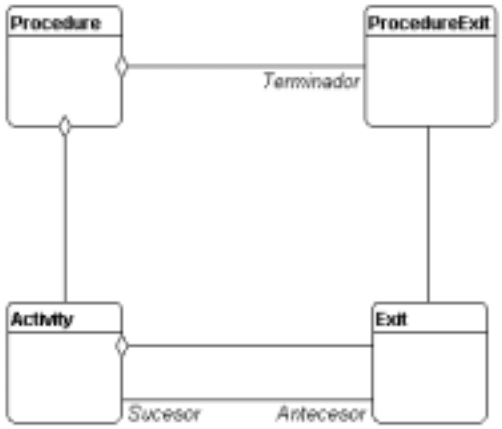

Imagen 10.1-1. Modelo de clases simplificado del editor.

Una vez hecho el diseño lo que se necesita es una forma rápida de pasarlo a código. Usando RDM todo el código del modelo se reduce a las siguientes líneas:

```
public class NodesRelations {
      static Relation r1 = new Aggregation("procedure", "r1",
"activity", Cardinality.ONETOMANY);

      static Relation r2 = new Aggregation("activity ", "r2", "exit",
Cardinality.ONETOMANY);

      static Relation r3 = new StandardRelation("predecessor", "r3",
"successor", Cardinality.MANYTOMANY);

      static Relation r4 = new Aggregation("procedure", "r4",
"terminador", Cardinality.ONETOMANY);

      static Relation r5 = new StandardRelation("terminador", "r5",
"exit", Cardinality.ONETOMANY);
}



public class Procedure extends StandardEntity {
      public Procedure () {
            add(NodesRelations.r1, "procedure");
            add(NodesRelations.r4, "procedure");

            add("id", ID);
            add("name", ID);
            add("description", "zdesc");
      }
}
```

```
public class Activity extends StandardEntity {

	public Activity () {
		add(NodesRelations.r1, "activity");
		add(NodesRelations.r2, "activity");
		add(NodesRelations.r3, "successor");

		add("id", 0);
		add("name", "zname");
		add("description", "zdesc");
	}
}


public class Exit extends StandardEntity
{
	public Exit() {
		add(NodesRelations.r2, "exit");
		add(NodesRelations.r3, "predecessor");
		add(NodesRelations.r5, "exit");

		add("id", 0);
		add("name", “”);
		add("description", "");
	}
}


public class ProcedureExit extends StandardEntity
{
	public ProcedureExit() {
		add(NodesRelations.r4, "terminador");
		add(NodesRelations.r5, "terminador");

		add("id", ID);
		add("name", "zname");
		add("description", "zdesc");
	}
}
```

Nótese que aunque no aparece en el código automáticamente ya se obtiene la gestión automática de eventos, integridad en las relaciones y, lo que no es menos importante, una implementación fiable[32].

A continuación se muestra un ejemplo de como se formaría manualmente el modelo de instancias de la figura siguiente:

[32] La fiabilidad de las clases que iban a formar el modelo de distintas aplicaciones fue uno de los aspectos a la que se dio especial importancia. Por ello durante la codificación del RDM se sometió al código a más de 2.600 líneas de tests (los cuales están disponibles para su ejecución automatizada por si fuera necesario probar la fiabilidad de algún cambio en el código). Hasta la fecha, y después de haberla usado intensivamente en varios proyectos reales, no se ha producido ningún error debido al mismo.

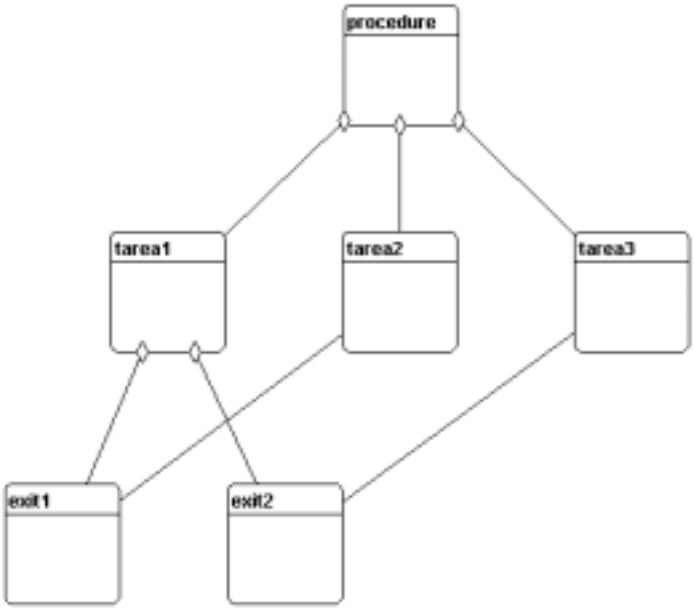

Imagen 10.1-2. Diagrama de objetos del procedimiento

```
Procedure procedure = new Procedure();
Activity tarea1 = new Activity();
Activity tarea2 = new Activity();
Activity tarea3 = new Activity();

procedure.add(“activity”, tarea1);
procedure.add(“activity”, tarea2);
procedure.add(“activity”, tarea3);

Exit exit1 = new Exit();
Exit exit2 = new Exit();
tarea1.add(“exit”, exit1);
tarea1.add(“exit”, exit2);

exit1.set(“successor”, tarea2);
exit2.set(“successor”, tarea3);
```

Estas sentencias son las que el editor precisamente evita tener que escribir ya que el modelo realmente se forma de manera visual.

10.1.2 Modificación del Modelo

Supóngase que ahora, como realmente ocurrió, hay que modificar el modelo. Se descubre que las actividades deben tener un responsable. Basta con las siguientes líneas de código para añadir la entidad *Responsible* y una relación que asocie a una *Activity*[33].

```
public class NodesRelations {
	...
	public static Relation r7 = new Aggregation("activity", "r7",
"responsible", Cardinality.ONETOONE);
}

public class Responsible extends StandardEntity
{
	public ProcedureExit() {
		add(NodesRelations.r7, "responsible");

		add("id", ID);
		add("name", "zname");
		add("description", "zdesc");
	}
}

public class Activity extends StandardEntity {

	public Activity () {
		...
		add(NodesRelations.r7, "activity");
	}
}
```

Durante el desarrollo del editor se realizaron tres prototipos. El primero se hizo de la forma tradicional sin usar RDM (codificando las clases y sus relaciones desde cero). A la vista de los problemas observados (los cuales se presentan en el apartado *Problemas* de esta sección) se pasó a utilizar RDM.

En la primera versión del prototipo se tardaron varios días en obtener la funcionalidad equivalente y depurarla (además del tiempo que supuso cada cambió del modelo). Sin embargo en los dos siguientes prototipos la implementación del modelo fue cuestión de minutos. Y esto no ocurrió porque se aprovecharan las clases de los editores anteriores. Los modelos de ambos editores eran radicalmente distintos (se pasó de modelar los procedimientos de forma jerárquica a encadenarlos al mismo nivel). Cada prototipo tenía su propio de diseño de clases con distintas relaciones.

RDM permitió centrarse *en los problemas reales* del dominio (operaciones y reglas) y *reutilizar los aspectos de codificación secundarios* y comunes a toda aplicación.

[33] La razón por la que se modifica la clase Activity es por la comodidad que supone disponer del código fuente. Sin embargo la relación se podría añadir a las instancias usando el patrón Abstract Factory sin necesidad de modificar la clase. En un apartado posterior se mostrará un ejemplo de cómo añadir relaciones a objetos en los cuales no se puede modificar el código fuente.

10.1.3 Reutilización del Modelo

En el apartado anterior se ha visto cómo crear nuevas entidades y relaciones. Sin embargo el hecho de disponer del código fuente hace que en ese ejemplo no se muestren todas las posibilidades de la asignación dinámica de relaciones. Un ejemplo de esto último se presentó cuando al editor hubo que añadirle los scripts.

Como se comentó en el capítulo anterior se quería dar a los procedimientos la posibilidad de ser ejecutados automáticamente (9.2). Por tanto hacía falta añadir a cada actividad la descripción de la secuencia de instrucciones a realizar; su script.

Sin embargo, en vez de desarrollar un nuevo lenguaje, se decidió reutilizar uno creado para un proyecto anterior; un proyecto de Business to Business (B2B) de intercambio de documentos XML. Uno de los módulos de dicho proyecto era el despachador de documentos el cual, en función del tipo de documento, lo trataba de la manera adecuada siguiendo un script. El script se modelaba en XML e incluía transiciones, bucles y condiciones aplicadas sobre modelo de componentes denominados *proclets*. Dichos componentes implementaban las acciones habituales (enviar un mail, acceder a una base de datos, obtener información de un WebService, etc.) además de disponer de una API en Java para la programación de nuevos proclets.

```
<script>
	<id>noname</id>
	<actions>
		<action>
			<name>sendMail</name>
			<params>...</params>
			<successor>...<succesor>
		</action>
		<action>
			...
		</action>
		...
	</actions>
<script>
```

Por tanto se tenía un lenguaje de documentación de procedimientos que requería descripciones ejecutables (el MADML) y por otro lado se tenía un lenguaje XML ejecutable (el ScriptML) por lo que la solución obvia fue insertar el script *como una más* de las etiquetas de las actividades.

```
<activity>
	<description>zdesc</description>
	<responsible>...<responsible>

	<script>
		<id>noname</id>
		<actions>
			<action>
				<name>sendMail</name>
				<params>...</params>
				<successor>...<succesor>
			</action>
			<action>
				...
			</action>
			...
		</actions>
	<script>

	...
</activity>
```

Quedaba implementar las clases para representar en memoria las etiquetas del nuevo lenguaje MADML que ahora incluyen scripts. Sin embargo al igual que se había reutilizado la sintaxis XML del lenguaje de script se pretendía reutilizar de la misma manera las clases Java que implementaban el script en el proyecto anterior (clase *script*, *proclet*, etc. junto con todas sus relaciones y reglas).

Sin embargo había algo que, en principio, impedía la reutilización de dichas clases. Una actividad tiene una serie de salidas y el script a su vez tiene sus propias salidas. Al acabar el script debe saberse por cuales de las salidas de su actividad continúa la ejecución. Es decir, el editor debe tener alguna forma de relacionar cada salida del script con la salida de la actividad a la que corresponde. Esto es similar a lo que ocurría en los DFD [Metrica21] en los que se podían expandir distintos niveles pero las salidas de un nivel se correspondían con las del padre.

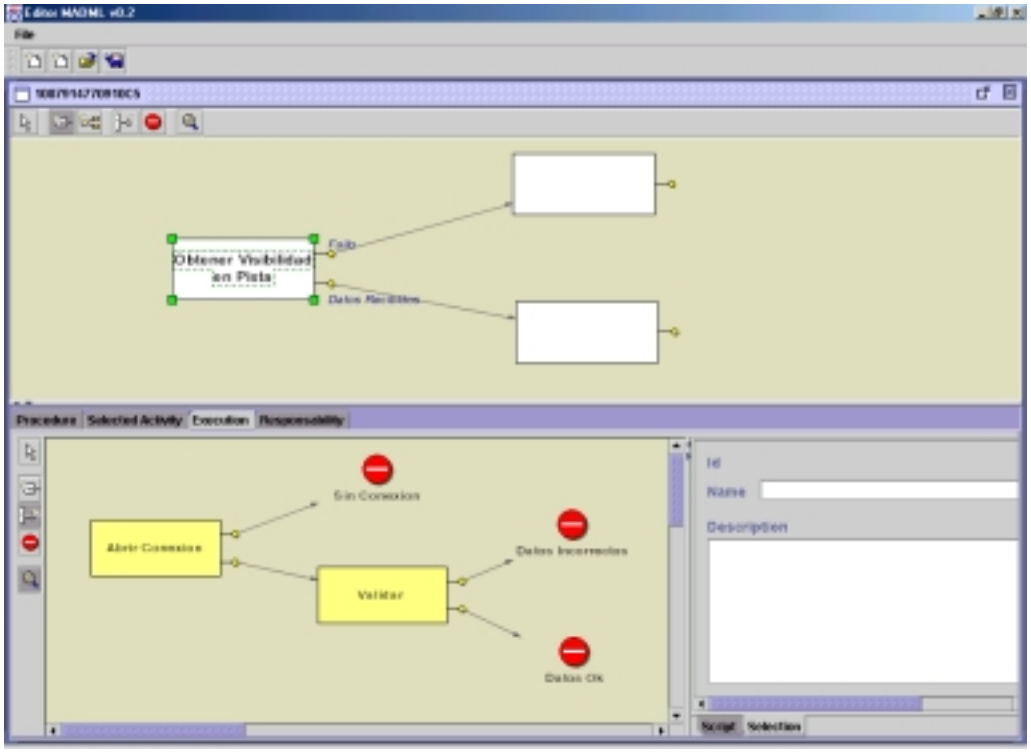

Imagen 10.1-3. Hay que relacionar las salidas de una actividad con las salidas de su script.

Por ejemplo la imagen anterior muestra en la mitad superior la actividad *Obtener Visibilidad en Pista* y en la mitad inferior se detalla cómo se realizará dicha actividad. En este caso se quiere indicar que los dos terminadores del script denominados “Sin Conexión” y “Datos Incorrectos” corresponden a la salida “Fallo” de la actividad y que el terminado denominado “Datos Ok” corresponden a la salida “Datos Recibidos”.

Esto en el modelo debe representarse con una relación entre la actividad y su script y entre los terminadores del script y las salidas de la actividad.

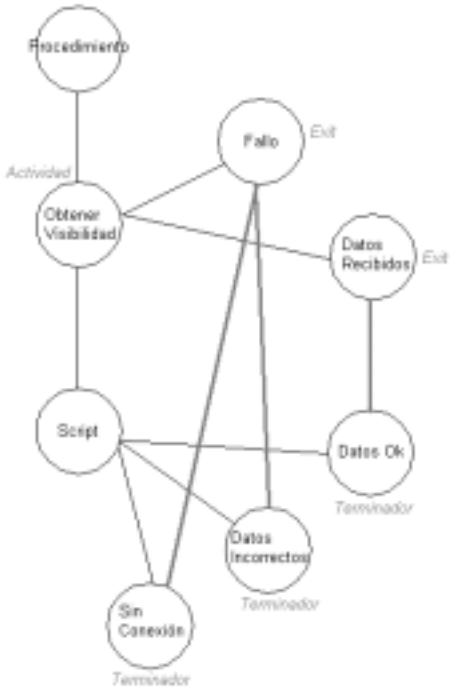

Imagen 10.1-4. Diagrama de objetos y las relaciones que hay que añadir (en negrita).

Se pretendían reutilizar las clases del modelo que utilizó en el ejecutor de scripts. Sin embargo la clase Script no tiene ninguna relación ni conoce a ninguna clase Activity al igual que sus terminadores no conocen a los del procedimiento. Se tienen una serie de objetos (el modelo del script) que deben utilizarse en un nuevo contexto. Deben colaborar con nuevas clases (las del modelo del procedimiento) sin conocerlas.

En el siguiente fragmento se muestra en pseudocódigo cómo hizo el EGP para transformar el XML de una actividad en un modelo.

```
/* Definido en otro lugar:
 Relation execution = new Aggregation("activity", "execution",
"executionScript", Cardinality.ONETOONE);

 Relation maspTo = new StandardRelation("scriptExit", "mapsTo",
"activityExit", Cardinality.MANYTOONE);
*/

// 1. Convertir actividad XML en objetos Java
activity = <leer la actividad en XML>

// 2. Leer script
<scriptXML = extraer de la actividad la etiqueta XML donde se describe
su script >

// 3. Convertir texto XML en objetos Java
Entity script = scriptXMLSerializer.readScript(scriptXML);

// 4. Añadir relación entre una actividad y su script
script.add(execution, "executionScript");

// 5. Añadir relaciones entre terminadores de script y salidas actividad
List exits = script.getList(“exit”);
for (int i = 0; i < exits.size(); i++) {
      Exit exit = (Exit) exits.get(i);
      exit.add(maspTo, “scriptExit”);
}

// 6. Añadir script al modelo (a su actividad)
activity.set(“executionScript”, script);
```

La actividad extrae del XML toda la información necesaria excepto el script (pasos 1 y 2). Para leer el script reutiliza la librería de clases de manipulación de scripts creadas para el despachador del otro proyecto. Una de las clases de esa librería es el *scriptXMLSerializer* el cual se encarga de transformar un XML en un modelo de objetos y viceversa. Este objeto serializador, lógicamente, no sabe nada de actividades ni procedimientos. Se limita a retornar un objeto script del que cuelgan sus componentes, salidas y transiciones (paso 3).

¿Qué le falta al script leído para que pueda juntarse con el modelo del procedimiento? Que pueda relacionarse con una actividad y que sus salidas (los terminadores) se puedan relacionar con las salidas de la actividad (pasos 4 y 5). Una vez hecho esto se añade al modelo del procedimiento sin que uno y otro necesiten conocerse (paso 6).

Ahora se puede navegar de manera uniforme sin importar de donde se sacaron y de que tipo sean los objetos del modelo. Por ejemplo la siguiente línea daría todos los tipos de acciones usadas en un procedimiento.

```
List actions = procedure.getPath(“activity.script.action.name”);
```

En definitiva no se debe dejar de reutilizar un objeto porque en el nuevo contexto se requiera un nuevo atributo o relación. Es seguro que va a necesitar algo distinto (porque sino sería el mismo modelo).

10.1.4 Reutilización de Vistas

Las vistas del EGP se hicieron con JHotDraw y con Swing. El protocolo para crear vistas era:

- Las vistas tienen como parámetro una Entity a la cual se suscriben como suscriptores.
- Cada vez que la entidad es modificada avisa a la vista y ésta obtiene las propiedades que necesite. Gracias al VSP esta vista se podrá combinar con cualquier entidad que tenga dichas propiedades sin importar su clase.

Así se reutilizan vistas para distintas entidades. Un ejemplo de esto es la vista *ImageFigure*. Esta clase era la vista de los Terminadores y mostraba el nombre del mismo (en la parte derecha de la siguiente imagen):

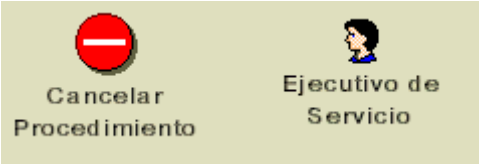

Imagen 10.1-5. Dos instancias de ImageFigure con distintos modelos

```
class ImageFigure extends AbstractFigure
{
	void setModel(Entity t) {
	...
	}
}
```

En el apartado de *Creación de vistas* se mostró como introducir un nuevo elemento *Responsible* al modelo. Pero quedaba por hacer su vista. Sin embargo la clase ImageFigure requería como modelo un objeto que tuviera un atributo "nombre". Por tanto se reutilizo la vista ImageFigure sobre una clase (el Responsable) que no conocía.

Lo mismo se hizo con la clase BoxFigure. Esta vista permitía mostrar el nombre de cualquier entidad en un rectángulo móvil. De esta manera cualquier entidad que tenga un nombre podrá mostrarse con cualquiera de las dos vistas (ImageFigure o BoxFigure).

Imagen 10.1-6. Tres vistas BoxFigure con distintos objetos del modelo

Un último ejemplo de reutilización de vistas es la clase *EntityView*. Esta clase implementa un diálogo que permite editar los atributos de cualquier entidad sin conocer su tipo. Aunque debido a razones de usabilidad suele ser más adecuado utilizar vistas específicas para cada tipo de entidad, la EntityView es muy útil para poder crear rápidamente prototipos de la aplicación sin necesidad de tanto trabajo inicial (especialmente teniendo en cuenta la volatibilidad de los atributos de las entidades en esas fases tempranas).

Posteriormente, a medida que las entidades se vayan asentando, se puede ir sustituyendo progresivamente este vista por otras más adaptadas a cada tipo de entidad. Concretamente en el caso del EGP, aunque en las imágenes de este texto ya aparecen diálogos especializados, inicialmente se utilizó esta vista para todos los elementos visuales seleccionables (procedimientos, actividades, salidas, etc.)

10.1.4.1 Las Relaciones como Requisito de Conexión

El retirar las clases y los interfaces y usar el VSP como forma de cumplimiento de contrato a la hora de pasar parámetros todavía fue más útil con un tercer tipo de vista: la ExitFigure. La ExitFigure es la vista de una salida de una actividad.

Imagen 10.1-7. Una ExitFigure es una vista que se *pega* a otras vistas

Al añadir una ExitFigure en la ventana de edición se añade una Exit a una actividad del modelo y de la misma manera al borrar la ExitFigure también se borra la Exit del modelo.

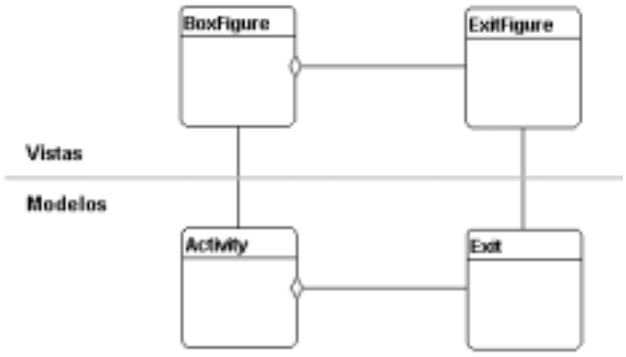

Imagen 10.1-8. Correspondencia entre las vistas y sus modelos

En el primer prototipo del editor la ExitFigure requería ser añadida a una BoxFigure. A ésta le pedía su modelo (una actividad) para añadirle la exit correspondiente.

```
class ExitFigure implements Vista
{
	void pegarseA(BoxFigure figura) {
	// Aquí se le pasa la vista a cuya derecha tiene que dibujarse
	}
}
```

Pero luego resultó que se querían añadir salidas a distintos tipos de vistas además de a la BoxFigure (a las vistas de los procesos y de los proclets). Sin embargo todas estas vistas tenían algo en común: todas eran *vistas* de un *modelo* que *tenía una relación* con exits. Eso era lo que realmente necesitaba la ExitFigure; un modelo al que añadir una exit. Por tanto la ExitFigure, en vez de requerir una BoxFigure, admitía cualquier vista *cuyo modelo* tuviera dicha relación.

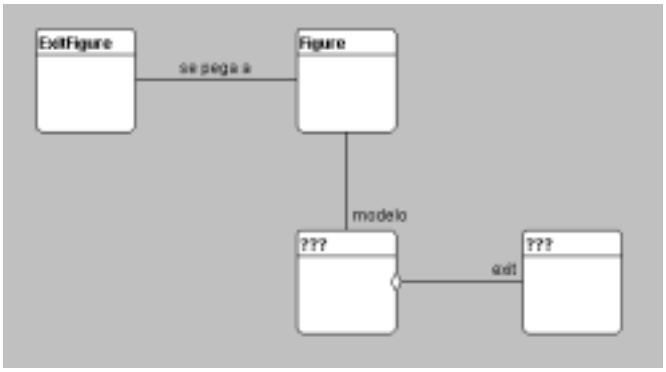

Imagen 10.1-9. Requisitos de una ExitFigure: una figura (vista) cuyo modelo admita *exits*

```
class ExitFigure implements Vista
{
	boolean pegarseA(Vista figura) {
	// Aquí se le pasa la vista a cuya derecha tiene que dibujarse
	Si a figura.getModel() se le pueden añadir salidas
		<dibujarse>
		return true;
	}
	<rechazar>
	return false;
}
```

Por tanto nótese que en vez de requerir, como es habitual, que el parámetro sea de un determinado tipo (por ejemplo una BoxFigure) se está requiriendo que tenga unas determinadas propiedades: en este caso *tener una relación* con un objeto adecuado. Por eso se le pueden añadir salidas a la BoxFigure; porque su modelo – el objeto *Activity* – tiene una relación con objetos de la clase *Exit*.

Un ejemplo representativo de esto es el caso de la combinación de la ExitFigure sobre una ImageFigure. Si la ImageFigure está asociada a un modelo que permita exits (por ejemplo una Actividad) se le pueden añadir ExitFigures.

Imagen 10.1-10. Una ExitFigure sobre una ImageFigure.

Sin embargo si está representado a un objeto del modelo sin esa capacidad (por ejemplo a un responsable) al pinchar salidas sobre él no ocurre nada. Es decir, no es la figura la que tiene la capacidad de que le añadan salidas; es el modelo al que esté visualizando en ese momento.

Supóngase cómo hubiera sido la solución habitual con interfaces. Supóngase que la ExitFigure pudiera haber sido añadida a cualquier figura que hubiera implementado un interfaz *FigureWithExits*.

```
class ImageFigure implements Vista
{
       void pegarseA(FigureWithExits figura) {
       // Aquí se le pasa la vista a cuya derecha tiene que dibujarse
       }
}
```

Cada vista que quiera que se le puedan dibujar salidas a su derecha (BoxFigure por ejemplo) tendría que implementar dicho interfaz para poder ser usado con el método *pegarseA*.

- Pero entonces la BoxFigure siempre hubiera permitido que la añadieran salidas, aunque con el modelo al que representara no tuviera razón de ser. Eso sería una inconsistencia del diseño. ¿Si la BoxFigure está mostrando un responsable qué sentido tiene añadirle una salida?

- Además limitaría la reutilización de la clase BoxFigure. Conocería a las Exit (ya que implementaría el interfaz *FigureWithExits*). ¿Y si se quiere poner una BoxFigure en otra aplicación que no tenga nada que ver con editores de procedimientos? ¿Qué es eso del *FigureWithExits*? El problema se presenta de nuevo. La BoxFigure sabe demasiado de su entorno. Sabe que está en un editor que contempla salidas. No se la podrá por tanto llevar a otro contexto.

Esto no se puede hacer con clases ni interfaces. La capacidad de aceptación de la ExitFigure no se aplica a toda una categoría de objetos; se aplica a objetos individuales. Es decir, el hecho de que una ExitFigure se pueda asociar a una BoxFigure no se puede evaluar para todas las BoxFigures; solo para aquellas que *actualmente estén asociadas* a un modelo adecuado (es decir, que tengal una relación con Exits).

10.1.5 Reutilización de Operaciones

En los apartados anteriores se ha visto como construir y adaptar modelos existentes y se ha visto como conectar estos con distintas vistas sin que se conozcan. Ahora se mostrará un ejemplo de como se utilizó RDM en el EGP para implementar las operaciones.

El EGP debía permitir la edición simultánea de varios procedimientos y el copiado de fragmentos entre ellos. Por tanto había que implementar la operación de clonado de objetos. La operación de clonado es algo que se repite una y otra vez cada vez que se implementa un modelo. Aunque realmente lo que se hace es dejar la implementación por defecto con el peligro que supone que no quede con la semántica adecuada.

Dado que hay que hacer dicha operación ¿Se puede hacer de una vez por todas para toda aplicación? La pregunta es si al clonar un objeto hay que clonar todos los objetos que estén relacionados con él.

La respuesta es que depende del objeto sobre el que se aplica la clonación. Volviendo a la figura del modelo del procedimiento puede verse que si la clonación se aplica al procedimiento hay que

clonar todos los demás objetos pero si ésta se aplica a un terminal no hay que clonar ningún objeto más.

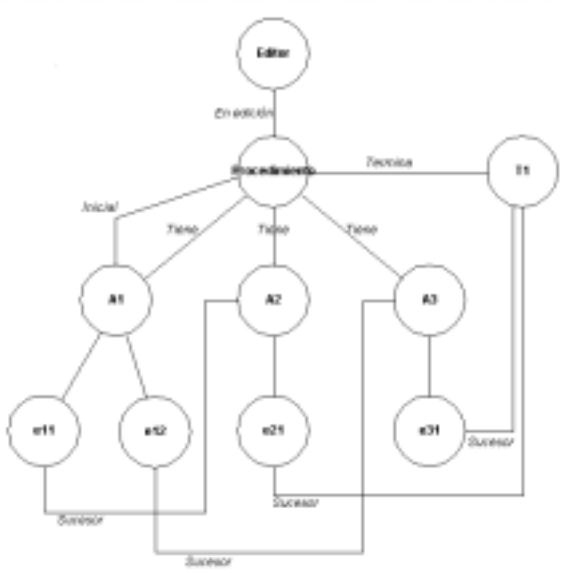

Ilustración 10.1-1. La operación de clonar es distinta dependiendo del objeto al que se aplique

Sin embargo la solución que se adopta generalmente es que sea el propio objeto, mediante la redefinición de un método especial denominado *clone* o equivalente, el que decida qué objetos hay que clonar con él [Campione99]. Es decir, al igual que ocurría con el borrado, para cada objeto del modelo hay que codificar una operación de clonación, operaciones que habrá que actualizar cada vez que entre un nuevo tipo en el modelo.

En el EGP se ha implementado la operación de clonado de tal manera que pueda ser reutilizada en distintos modelos. La operación de clonado, al aplicarla sobre un objeto, además de clonar el propio objeto produce lo siguiente:

- Automáticamente todo objeto que esté relacionado con él mediante una agregación es a su vez clonado y relacionado con el clon de su agregador. Esta es precisamente la semántica de una relación de agregación en el AOO/DOO [DSouza98]. Cuando un objeto tiene una relación de agregación sus agregados forman parte de su estado. Las partes no tienen sentido sin su agregador; y copiar el agregador implica copiar también sus partes. Copiar el procedimiento y no copiar las actividades no es copiar el procedimiento. Nótese que las relaciones de agregación no tienen bidireccionalidad en cuanto a la clonación. Clonar una salida no significa clonar su actividad aunque las una la misma relación de agregación.

- En cuanto al resto de las relaciones no hay que copiar los objetos que estén al otro lado del clonado. Al copiar una actividad no hay que copiar las actividades sucesoras. Sin embargo no basta tampoco con copiar solamente los objetos unidos mediante agregación y dichas relaciones. Hay que copiar también toda relación, aunque no sea de agregación, *que una a dos objetos clonados*. Supóngase el caso de copiar un procedimiento. Siguiendo el árbol de agregaciones se clonarían también sus actividades y sus salidas. Pero deben clonarse también las relaciones entre las salidas de una actividad y sus sucesores. En caso contrario el usuario se encontrará que al visualizar el nuevo

procedimiento en una ventana habrán desaparecido todas las flechas entre las salidas y los sucesores. Las transiciones entre actividades, aunque no sean relaciones de agregación, forman parte del estado del procedimiento y deben ser copiadas con éste.

Por ejemplo supóngase que en el editor se tienen seleccionadas las actividades A1 y A2 del procedimiento y se quiere hacer una copia para pegarla en otro procedimiento similar.

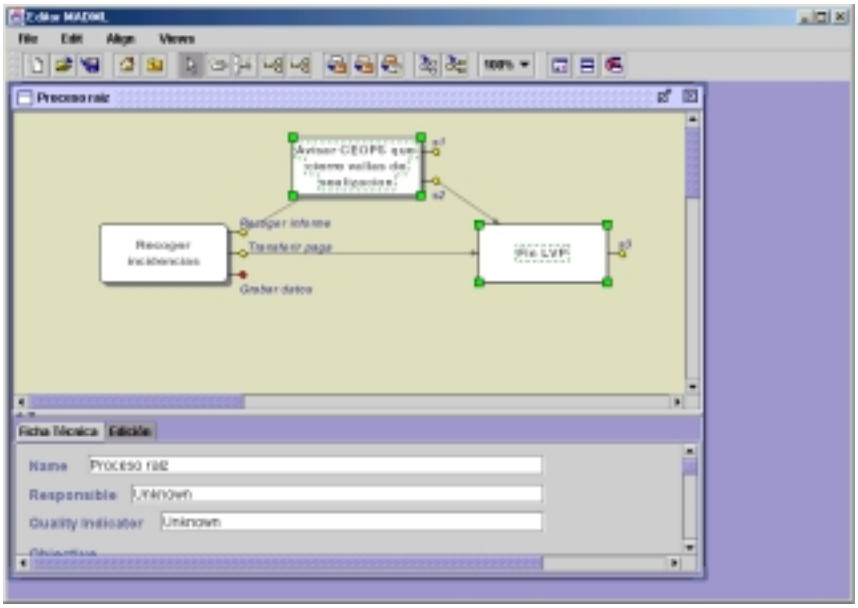

Imagen 10.1-11. Procedimiento con dos actividades seleccionadas.

Los objetos del modelo y las relaciones copiadas son las que aparecen resaltadas en negrita.

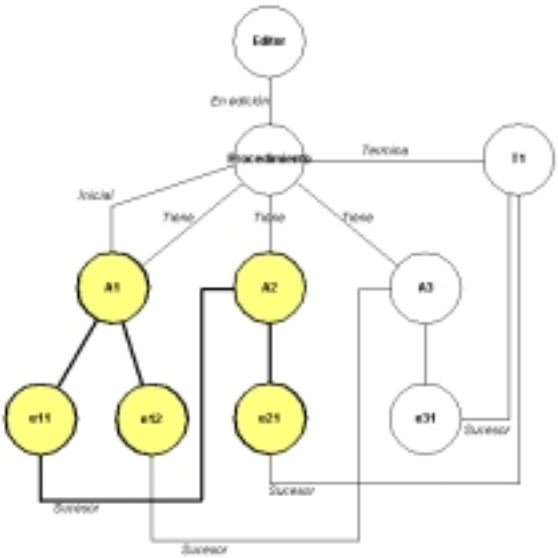

Imagen 10.1-12. Objetos y relaciones a clonar (en negrita)

Se copian los dos elementos seleccionados junto con sus salidas. Se copia también la relación de sucesión entre las dos actividades pero no las que haya con otras actividades que no estarán en el nuevo contexto.

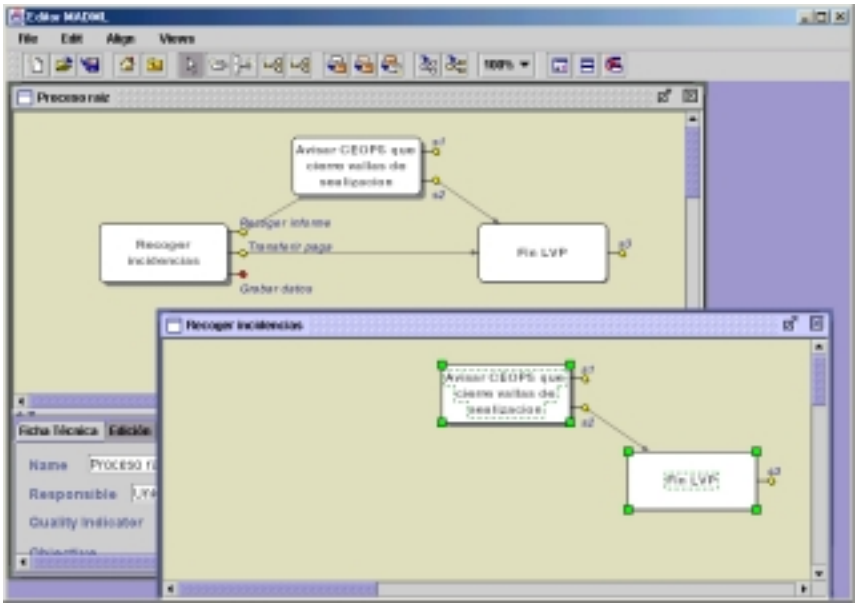

Imagen 10.1-13. Resultado de la clonación.

En definitiva las relaciones de agregación determinan qué objetos forman parte de la copia. Una vez determinado este conjunto se copia toda relación (sin importar ya el tipo) que haya entre toda pareja de objetos de dicho conjunto.

Lo que se pretende mostrar con el proceso anterior es que ahora se puede clonar cualquier modelo independientemente de las entidades que lo formen. La operación de clonado se podrá utilizar ahora sobre cualquier modelo de cualquier aplicación sin tener que volver a codificarlo.

Otros ejemplos de operaciones que pueden ser reutilizadas con cualquier modelo son las habituales opciones de Cut, Copy y Paste que se encuentran en cualquier editor.

- La operación Copy se apoya en la operación de clonado para crear una copia de la parte seleccionada del modelo y guardarla en el portapapeles.

- La operación Cut se apoya en la anterior y en la de borrado para eliminar el original.

- Finalmente la operación Paste se limita a realizar una copia de las entidades contenidas en el portapapeles y a asociarla con la entidad designada como destino. Y, como se vio anteriormente, al modificar el modelo *cumplirá sus reglas* sin conocerlas gracias a los monitores.

Puede observarse que su independencia del modelo se obtiene mediante su desconocimiento del mismo, ya que todo acceso al mismo lo hacen a través de las operaciones de borrado y de clonado (las cuales, como se mostró anteriormente, se pueden usar con cualquier modelo).

Finalmente un último ejemplo de reutilización de operaciones es la operación de serialización de modelos RDM en XML y viceversa. Un mismo serializador puede utilizarse para guardar en XML cualquier modelo RDM sin saber su objetivo.

10.1.6 Unificación de Operaciones

Poder aplicar una misma operación sobre varios modelos también hace que se evite tener que hacer varias operaciones ya que éstas se pueden sustituir por una sola. Un ejemplo de esto es el caso de las herramientas de creación de entidades y las de creación de relaciones.

Imagen 10.1-14. Operaciones del editor

Sin RDM la forma de implementar las herramientas hubiera sido:

- La herramienta que añade actividades tendría que crear un objeto de la clase *Activity* y usar el método *addActivity* de la clase *Procedure* para añadir la nueva actividad.

- La herramienta que añade Responsables tendría que crear un objeto de la clase *Responsible* y usar el método *addResponsible* de la clase *Activity* para añadir la nueva actividad.

- La herramienta que añade terminadores tendría que crear un objeto de la clase *Exit* y usar el método *addExit* de la clase *Procedure* para añadir la nueva actividad. Y así con todas las herramientas de creación de objetos del modelo.

Lo mismo ocurría a la hora de establecer relaciones:

- La herramienta que añade un sucesor tendría que relacionar los dos objetos seleccionados (salida y actividad) mediante el método *setSuccessor* de la clase *Exit*.

- La herramienta que añade un responsable tendría que relacionar los dos objetos seleccionados (actividad y responsable) mediante el método *setResponsible* de la clase *Activity*, además de tener en cuenta la acciones para mantener la integridad (quitar el anterior responsable, por ejemplo).

- Y así con el resto de las relaciones (con recursos, con otros procedimientos, etc).

Sin embargo ahora utilizando RDM:

- Solo hay una *herramienta de creación* de entidades que se reutiliza para las distintas operaciones. Las distintas herramientas son instancias de esta clase a cada una de las cuales se les pasa como parámetro el objeto a crear y el nombre del rol con la que se añaden al procedimiento.

- Solo hay una *herramienta de establecimiento* de relaciones entre entidades. Las distintas herramientas son objetos de esta clase a cada una de las cuales se les pasa como parámetro el rol por el cual relacionar los dos objetos seleccionados.

10.2 Conclusiones sobre el uso de RDM. Demostración de la Tesis

La tesis presentada en éste documento afirmaba que *una arquitectura que esté basada en la representación especializada e independiente de los distintos elementos del dominio proporciona una reducción en los tiempos de desarrollo y favorece la reutilización*.

El capítulo 7, "El Framework RDM", demostró la viabilidad de implementar una arquitectura basándose en el Principio de Especialización. A continuación se demostrará que efectivamente esta arquitectura supone una *reducción en los tiempos de desarrollo* y que *favorece la reutilización*.

10.2.1 Reducción en Tiempo de Desarrollo

Mediante la programación tradicional el número de líneas de código en Java de cualquier clase del modelo del editor incluía *al menos*:

- Tres atributos.
- Dos relaciones.
- Notificación de eventos ante los cambios de todos ellos.

La implementación de una clase de éstas características mínimas estaría formada aproximadamente por unas 40 líneas[34]. Pero esto sería *sin haber implementado* todavía el mantenimiento de la integridad de las relaciones, la recuperación ante fallos, ni las líneas de test para verificar que no hay errores de codificación en todo lo anterior.

Una clase equivalente en RDM tiene cinco líneas (como se ha visto en 10.1.1) e incorporaría automáticamente todo lo que le falta a la clase anterior (y que sería *requisito obligatorio* en una aplicación profesional). Por tanto incluso ignorando este hecho ya se obtiene una reducción de las líneas de código de un 87% (ya solamente en lo que refiere a las clases que forman parte del modelo).

El código del resto de las clases también se ha visto reducido debido a:

- Reutilización de vistas y de operaciones independientes del modelo (borrado, Copy, Paste, transformación a XML, etc.).
- La reutilización los monitores y reconstructores del framework.

[34] Generado automáticamente con Borland JBuilder 3 y sin contar líneas en blanco ni ninguna otra línea que no tuviera alguna sentencia en Java (como por ejemplo las líneas con paréntesis).

Esto produce una reducción de código adicional, pero basta considerar únicamente el que se ha obtenido en la implementación de las clases del modelo para llegar a una reducción que *demuestra el cumplimiento de la tesis presentada*.

10.2.2 Aumento de Reutilización

Como subobjetivo de la tesis se planteaba también la reutilización de los distintos elementos del dominio. En las Tecnologías Orientadas a Objetos una clase del modelo realizaba demasiadas funcionalidades:

- Conocer con qué otras clases estaba relacionada.
- Conocer sus operaciones y las reacciones que producían éstas sobre terceros.
- Conocer sus reglas de validación y las de todo aquel a los que afectaba tanto directa como indirectamente en sus operaciones

Cuando una clase se llevaba a otra aplicación *bastaba que uno* de estos elementos fuera distinto (faltara o sobrara) para que las clases del modelo no pudieran ser reutilizadas. Por ejemplo basta que una de las reglas no tenga sentido en el nuevo dominio o que sean necesarias nuevas relaciones.

Y la probabilidad de que en la nueva aplicación hagan falta *los mismos atributos* con *las mismas relaciones* y con *las mismas reglas* de negocio es extremadamente baja ya que entonces sería la misma aplicación. Por eso rara vez se reutilizan las clases del modelo. Se acaba antes reimplementándolas.

Sin embargo con RDM el grado de reutilización es mucho mayor ya que se puede:

- Reutilización de los *modelos* ya existentes con la posibilidad de adaptarlos al nuevo dominio (añadiendo las reglas o relaciones necesarias). Ver 10.1.3.
- Reutilizar *vistas* con nuevos modelos. Ver 10.1.4.
- Reutilizar *monitores*. Ver 7.1.4.2.
- Reutilizar *operaciones* pero *bajo el control* de las reglas específicas del dominio. Ya no hay operaciones *centralizadas* que deban conocer todo el modelo – sus objetos, relaciones y restricciones – y que por tanto haya que recodificarlas al cambiar cualquiera de éstos. Ver 7.1.4.4 y 7.1.4.5.
- Reutilizar *el propio framework* RDM a la hora de crear modelos para nuevas aplicaciones. La gestión de relaciones y la reconstrucción de modelos no tiene sentido repetirla en cada aplicación. RDM se presenta como una librería stándard para modelar esos aspectos de la aplicación. Ver 10.1.1.

11 Otras Aplicaciones Basadas en RDM

Se ha elegido el Editor Gráfico de Procedimientos como caso concreto para demostrar las ventajas de la Arquitectura RDM y por tanto la tesis aquí presentada. Pero RDM tambien ha sido utilizada en otros proyectos reales tanto nacionales como internacionales que podrían también considerarse como demostración de lo expuesto en éste documento.

A continuación se da una relación de los proyectos en los que se ha aplicado la Arquitectura RDM agrupados por la empresa para la que se desarrolló.

11.1 Saunier Duval España (Bilbao)

Proyecto	COMNET
Descripción	Herramienta de Gestión Comercial para los Agentes de Saunier Duval.

Proyecto	PORTAL DE NOTICIAS
Descripción	Portal de negocio construido a partir de una capa de abstracción sobre el ERP de Saunier Duval (AMIS)

11.2 Saunier Duval Italia (Milán)

Proyecto	SCM ITALIA
Descripción	Proyecto de Supply Chain Management para la gestión de stocks de la planta de Saunier Duval en Italia

11.3 CalorYFrio.com (Bilbao)

Proyecto	PORTAL – GESTIÓN DE NOTICIAS
Descripción	Portal web del grupo que comprende a los fabricantes pertenecientes al sector de calderas y aire acondicionado de España.

Proyecto	CATÁLOGO DE PROFESIONALES
Descripción	Módulo adscrito al portal anterior

11.4 Grupo Hepworth (Paris-Praga)

Proyecto	SUPPLY CHAIN MANAGEMENT
Descripción	Integración multinacional para varios ERPs (MOVEX, JDE, AMIS).

11.5 AENA (Madrid - Barajas)

Proyecto	CUADRO DE MANDOS
Descripción	Módulo de notificación de incidencias y cuadro de mandos para los ejecutivos de servicio del aeropuerto.

Proyecto	INTEGRACIÓN CISIA
Descripción	Módulo de notificación-recepción de incidencias para el departamento de mantenimiento del aeropuerto

Proyecto	H24
Descripción	Editor, motor de ejecución y visor de la traza de ejecución de procesos aeroportuarios

11.6 Grupo Farmasierra (Madrid)

Proyecto	MIDDLEWARE DE COMUNICACIÓN
Descripción	Desarrollo de middleware de interconexión y creación de los servicios del grupo farmacéutico

Proyecto	DISTRIBUCIÓN
Descripción	Desarrollo para la empresa Farmasierra Distribución que resuelve la gestión y tracking de pedidos, así como, la facturación asociada.

Proyecto	COMPRAS
Descripción	Desarrollo para Farmasierra S.A. y Madariaga S.A.. Módulo de gestión de aprovisionamiento para proveedores del grupo.

Proyecto	PRODUCCIÓN
Descripción	Desarrollo para Farmasierra S.A. y Farmasierra Natural, para la asignación de estados y control del flujo productivo.

11.7 B2B2K (Bilbao)

Proyecto	INTRANET Corporativa
Descripción	Intranet corporativa de Comercio Electrónico B2B 2000 S.A.

Sección 5. Conclusiones y Futuras Líneas

12 Conclusiones

12.1 Resumen del Documento

La utilización actual de las tecnologías orientadas a objetos en el ámbito profesional presenta unas deficiencias que disminuyen la productividad de los equipos de desarrollo. Se necesita implementar de manera engorrosa, repetitiva y proclive a errores funcionalidad no relacionada directamente con el objetivo de la aplicación. Además este código es difícil de mantener y pequeñas modificaciones del dominio implican largas y tediosas modificaciones en el sistema. Además esta funcionalidad no es reutilizable directamente en otros sistemas.

RDM es una arquitectura que surge para resolver los problemas *recurrentes* del modelado de dominios. Lo mismo que los procesadores RISC propusieron aumentar el rendimiento del procesador haciendo más rápidas las instrucciones que se usaban *con mayor frecuencia* aquí se propone disminuir el tiempo de desarrollo haciendo más rápida la implementación de los elementos que suponen la mayor parte de los cambios del dominio.

El objetivo es poder centrarse en el diseño simplificando su posterior traslación al código lo máximo posible. Si los patrones de diseño [Gamma94] suponen la solución a los casos *habituales de diseño* este framework supone un patrón arquitectónico para modelar los *elementos habituales* del dominio (entidades, relaciones, reglas, etc).

La arquitectura ha surgido de la aplicación sucesiva del Principio de Especialización. Éste no solo ha separado datos y operaciones sino que además a estas últimas se las ha extraído a su vez tres responsabilidades: comprobar si se puede hacer la operación (monitor), saber todas las entidades que deben ser afectadas (reacciones) y recuperar el modelo si algo de lo anterior falla (reconstructores). Por tanto las operaciones se quedan en la verdadera esencia de la tarea a realizar lo cual facilita su implementación, comprensión y reutilización.

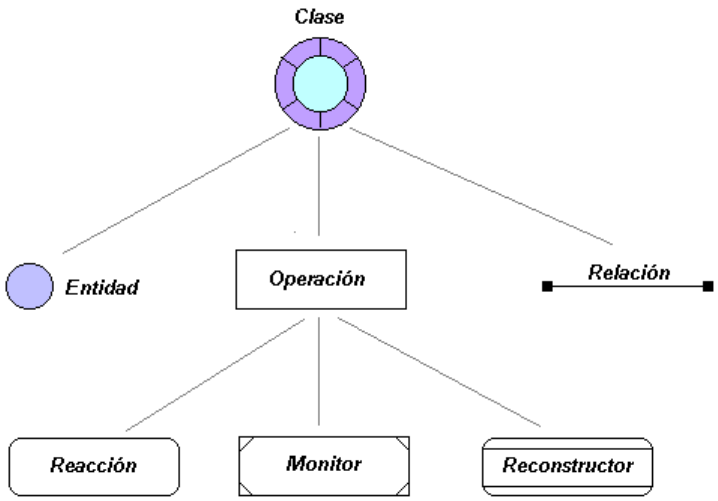

Ilustración 12.1-1. Elementos obtenidos mediante la aplicación del Principio de Especialización a la Clase.

La conexión adecuada de éstos elementos mediante el cumplimiento los cuatro principios adicionales de Conformidad, Adyacencia, Inconexión y Externalización (apartado 6.2) da lugar a la Arquitectura RDM.

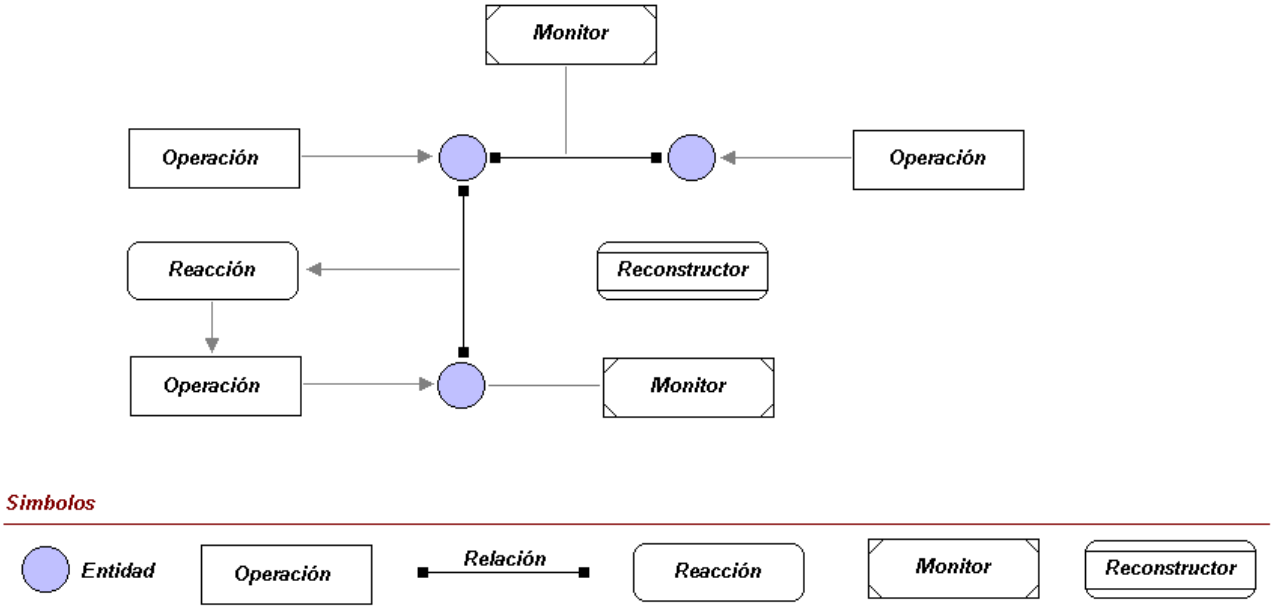

Ilustración 12.1-2. Elementos de la Arquitectura RDM

La arquitectura se valida por la construcción de un framework implementado en Java el cual se ha aplicado con éxito en diversos sistemas comerciales constatándose la ganancia de productividad.

12.2 Elementos frente a Objetos

Un objeto tradicional encapsulaba varios de éstos elementos. El inconveniente de esta encapsulación es:

- Hay qué cambiar la clase para insertar, modificar o borrar cualquiera de los elementos que se encuentran mezclados en la misma. Esto supone mayores tiempos de desarrollo.
- Se pierde correspondencia entre el dominio y la solución (Direct Mapping) ya que es más difícil trazar el camino de cada elemento al diseño al estar mezclado con otros elementos o bien disperso en varias clases.
- Se dificulta la reutilización ya que los objetos son más rígidos en su composición.

Se propone mantener los elementos por separado de tal manera que se consiga la funcionalidad del objeto tradicional pero permitiendo nuevas combinaciones a medida que cambie el dominio.

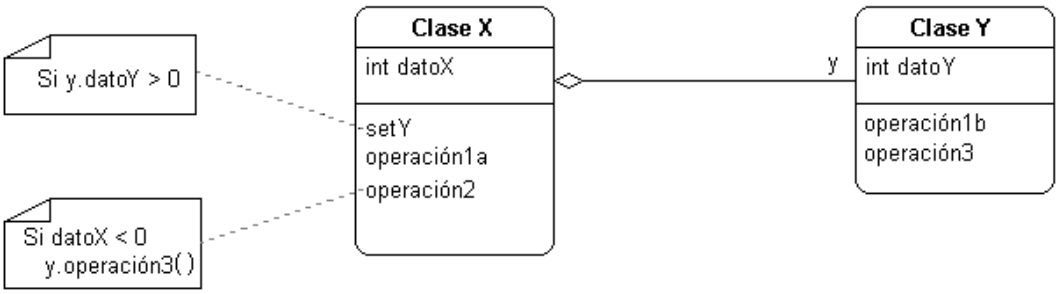

Imagen 12.2-1. Modelado tradicional.

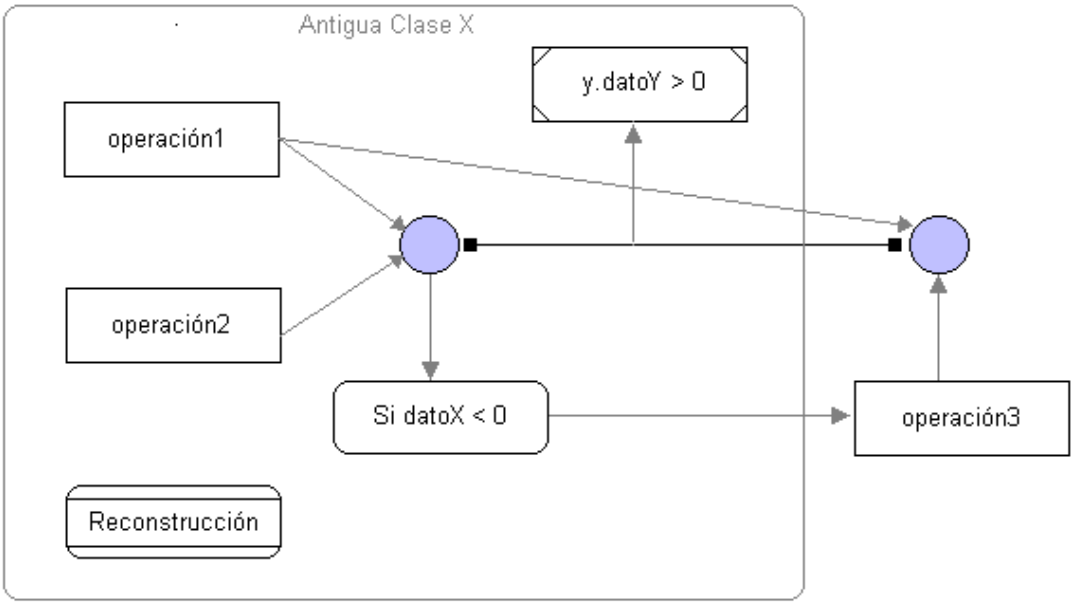

Imagen 12.2-2. Modelado RDM. La clase se descompone en elementos independientes.

12.3 Elementos frente a la Programación Estructurada

Aunque se vuelven a separar los datos y las operaciones no se tienen los problemas de la programación estructurada identificados por Meyer [Meyer97]:

- La propagación de cambios se evita porque las operaciones conocen lo mínimo de los datos gracias al VSP (Principios de Conformidad y Adyacencia) y de la misma manera las clases de los datos no necesitan conocer a las operaciones, a las reglas ni a sus relaciones (debido al Principio de Externalización y al de Inconexión).

- Los errores no se propagan entre los módulos ya que los monitores cuidan de la integridad del modelo y los reconstructores impiden los estados inválidos.

Sin embargo es innegable que la clase Pila, el clásico ejemplo de implementación orientada a objetos, obtiene un mejor diseño mediante la encapsulación de datos y operaciones. ¿Es entonces mejor ocultar los datos dentro de las operaciones o separarlos? Hay que combinar ambas estrategias adecuadamente:

- La separación de datos y operaciones es mejor cuando los objetos *se definen en función de un dominio*, es decir, cuando su definición se realiza en función de los aspectos del mundo real que se quieran modelar. Cuantos más elementos se encapsulen mayor probabilidad de tener que romper la cápsula para sacar alguno de ellos que deba ser modificado.

- La encapsulación es adecuada cuando la responsabilidad del objeto *no depende del dominio* de la aplicación. Ejemplos de éste caso son las habituales clases Pila, Ventana, Botón, etc. cuyo comportamiento no proviene del dominio. Son elementos autosuficientes.

12.4 Aplicación de RDM

RDM es una solución cómoda y práctica para cualquier aplicación que necesite manipular un modelo de objetos.

- Editores y herramientas de composición en general.
- Herramientas de Simulación.
- Compiladores e Intérpretes (la tabla de símbolos es un candidato perfecto a ser modelado en RDM, especialmente en intérpretes que permitan la adición dinámica de tipos, relaciones entre ellos y nuevas operaciones en fase de ejecución).
- Prototipos rápidos en la fase de análisis.
- En el caso de aplicaciones de tres niveles (3 Tier) la capa intermedia – como por ejemplo los Servlets – pueden guardar cómodamente información sobre el contexto (por ejemplo para modelar el carrito de la compra) y en general para representar y manipular en una capa cliente o intermedia el resultado de accesos a bases de datos.
- Para el modelado de documentos XML, en especial aquellos que incluyan relaciones entre los elementos. Normalmente estos documentos se representan en memoria de forma jerárquica (por ejemplo como obligan paquetes como DOM [W3C98b] o JDOM [Hunter]). Este modelo es muy engorroso para navegar por las entidades ya que *la única relación* que contemplan es la de pertenencia a un elemento padre.

Hasta hace pocos años era habitual impartir la docencia de introducción a la programación usando lenguajes sin librerías estándar de estructuras de datos (listas, tablas hash, etc.). El lenguaje más habitual sin duda era el Pascal. La mayor parte del trabajo del alumno se empleaba en reimplementar dichas estructuras de datos (la clase Nodo, las operaciones de manipulación de punteros, etc.). Y no solo en cada aplicación sino a veces incluso *para cada tipo de elemento a guardar* en una misma aplicación. Eso distraía del objetivo fundamental de la aplicación a la vez que era la mayor fuente de errores de ejecución.

Al pasar a utilizar nuevos lenguajes con contenedores reutilizables (C++ con STL [Kalev98] o Java con su librería estándar *Collections*) el alumno se puede centrar en decidir qué guardar en

cada estructura de datos tratándola como una caja negra[35]. Esto supone un gran aumento de eficacia en el desarrollo y *ya no se acepta volver* a la etapa anterior. Esta es la situación actual.

RDM supone la siguiente etapa lógica. Aunque para implementar la clase Cliente se esté reutilizando la clase Vector para guardar las relaciones con sus Facturas no se puede tener que estar reimplementando los métodos que permiten altas, bajas, modificaciones y consultas sobre el mismo; no se puede tener que estar implementando la gestión de la integridad de las relaciones a ambos extremos; la gestión de reglas; comunicación con las vistas; las recuperaciones, etc.

Y este problema no se limita a la enseñanza y a la implementación de prácticas. La situación en la empresa privada con programadores profesionales es idéntica. Pierden el mismo tiempo implementando las mismas tareas de bajo nivel. Por ello cuando la arquitectura se ha utilizado en un proyecto no se plantea volver a la etapa anterior lo mismo que no se plantea volver a reimplementar la clase Vector o tabla hash.

[35] Otro tema independiente sería la necesidad de conocer cómo implementar estructuras de datos como bagage necesario para un titulado en informática. Sin embargo este sería el objetivo de otra asignatura de tal manera que no distraiga del objetivo fundamental de un temario dedicado al modelado orientado a objetos.

13 Futuras Líneas de Investigación

La Arquitectura RDM establece las líneas básicas para el modelado de los distintos elementos del dominio. Pero quedan abiertas nuevas líneas de investigación para completar o mejorar lo existente.

13.1 Arquitectura RDM distribuida

Hay que diferenciar la Arquitectura RDM del framework del mismo nombre que supone una de sus posibles implementación. Otra implementación posible sería una versión distribuida de la misma que permitiera:

- Relacionar objetos remotos sin que éstos se conozcan.
- Acceder a ellos de forma uniforme a través de un VSP remoto.
- Suscribirse a notificaciones de cambio de estado de objetos remotos.
- Navegar por dichos objetos independientemente de su localización mediante el mismo lenguaje.
- Añadir operaciones a objetos remotos sin tener que modificarlos. Se aprovecharía así que las operaciones no tienen que estar dentro de los objetos.

La siguiente fase del proyecto H24 de AENA afrontará este problema. En el *Apéndice A* se incluye un documento que establece la necesidad de dicha implementación.

13.2 Extensiones de la Arquitectura

13.2.1 Evolución del Sistema de Eventos

El modelo de eventos actual de RDM solo permite registrarse a los cambios de estado de una entidad o de una relación. Sin embargo hacen falta modelos de suscripción más avanzados:

- Sistema de Notificación Integral
- Sistema de Suscripción Basado en Rutas

13.2.1.1 Sistema de Notificación Integral

Cuando un objeto tiene una relación de *agregación* con otro objeto este último pasa a formar parte de su estado [Cook94]. Por el Principio de Externalización todo objeto deberá notificar de los cambios que sufra en su estado. Por tanto si un objeto *agregado* es modificado ¿deberá el objeto *agregador* notificar de dicho cambio como si lo hubiera sufrido él?

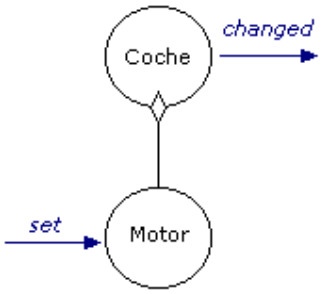

Imagen 13.2-1

En el EGP se presentó el siguiente problema. En la ficha de procedimiento hace falta una *vista textual* que describa lo que el usuario ha dibujado graficamente. En dicha ficha se muestran los datos del procedimiento así como las actividades que la forma y las transiciones entre sus salidas y otras actividades.

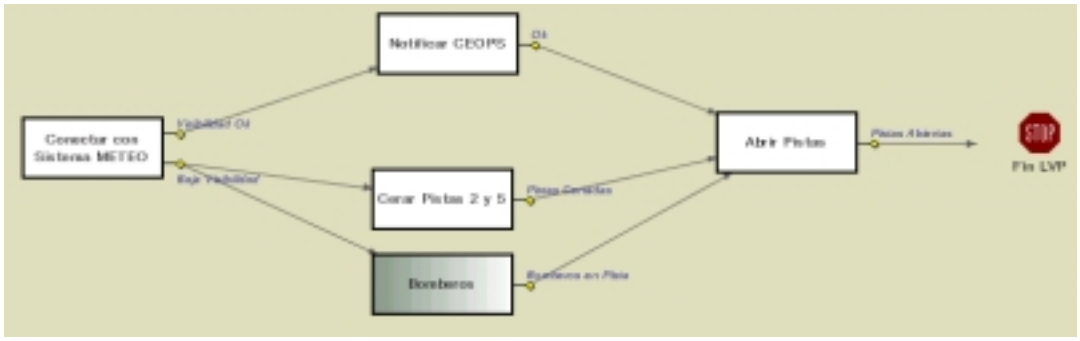

Imagen 13.2-2. Vista gráfica de un procedimiento

Ficha

Ficha

Nombre: Baja Visibilidad

Descripción: Medidas a tomar cuando la visibilidad en torre baja del nivel umbral

Acciones

Acción	Salida	Sucesor
Conectar con Sistema METEO	Visibilidad Ok	Notificar CEOPS
Conectar con Sistema METEO	Baja Visibilidad	Cerar Pistas 2 y 5
Conectar con Sistema METEO	Baja Visibilidad	Bomberos
Notificar CEOPS	Ok	Abrir Pistas
Cerar Pistas 2 y 5	Pistas Cerradas	Abrir Pistas
Bomberos	Bomberos en Pista	Abrir Pistas
Abrir Pistas	Pistas Abiertas	Fin LVP

Imagen 13.2-3. Vista texto del mismo procedimiento

La ficha de procedimiento, como vista que era, estaba registrada como suscriptor del procedimiento. De esta manera a medida que se iba modificando el dibujo del procedimiento se obtenía en tiempo real su documentación textual.

Sin embargo había un cambio del modelo que no se reflejaba en la ficha: cada vez que se añadía una nueva salida a una actividad dicho cambio no era reflejado hasta que la vista no se refrescaba completamente. La razón era que el evento de añadir una salida no le llegaba a las vistas del procedimiento; solo a los de las actividades.

Por tanto una solución era que a medida que se fueran añadiendo actividades al procedimiento registrar la ficha de procedimiento como suscriptora para que ésta se enterara de cuando se añadían salidas a dichas actividades. Pero esto no siempre era así: si la ficha no estaba abierta no era necesario este proceso. Esto era tremendamente engorroso ya que se preveía que sería fácil cometer errores de codificación de tal manera que quedaran actividades sin registrar o suscripciones a actividades ya borradas.

Además resulta que esto tampoco era suficiente. ¿Qué pasaría cuando cambiara el nombre de una salida? De nuevo la ficha tampoco se enteraría ya que aunque estaba ahora suscrita al procedimiento y sus actividades no recibía las notificaciones de las salidas. Por tanto había que repetir el proceso para cada una de las salidas de las actividades. Cada vez que se añadiera una actividad al procedimiento habría que suscribirse a sus salidas y darse de baja de las mismas al darse de baja su actividad. En definitiva el código de suscripción de la ficha se complicaba terriblemente.

Es obvio que cuando se le añade una actividad al procedimiento éste ha cambiado. Sin embargo por esta misma razón:

- Si cambia el nombre de una actividad cambia el procedimiento.
- Si una actividad tiene nuevas formas de finalización (salidas) el procedimiento ha cambiado.
- Si cambia el nombre de una salida el procedimiento ha cambiado.

Por tanto el sistema de notificaciones tiene que tener en cuenta la semántica de la relación de agregación para que todas estas notificaciones sean recibidas de manera transparente por los suscriptores de los objetos agregadores.

Sin embargo cuando se produce un evento una información que viaja con él es el nombre del miembro de la entidad que ha sido modificado. ¿Qué miembro *del procedimiento* se ha modificado cuando se modifica el nombre de una salida? ¿Cómo se diferencia de otro atributo que pudiera tener el agregador con el mismo nombre?

Las notificaciones de eventos deberán incluir *el nombre absoluto* del miembro. Se define el nombre *absoluto* de un miembro como el nombre de dicho miembro precedido de todos los roles por los que haya que navegar desde la entidad emisora del evento hasta llegar a la entidad que lo contiene. Por ejemplo supóngase que el objeto *procedimiento* tiene un suscriptor. Dicho suscriptor recibirá los siguientes nombres de miembro:

- Si cambia el nombre del procedimiento recibirá la notificación de que se ha modificado el miembro “nombre”.
- Si cambia el nombre de una actividad recibirá la notificación de que se ha modificado “actividad.nombre”. Es decir, se indica que se ha cambiado *el nombre de una de sus actividades*.

- Si cambia el nombre de una salida recibirá la notificación de que se ha modificado el miembro “actividad.exit.nombre”.

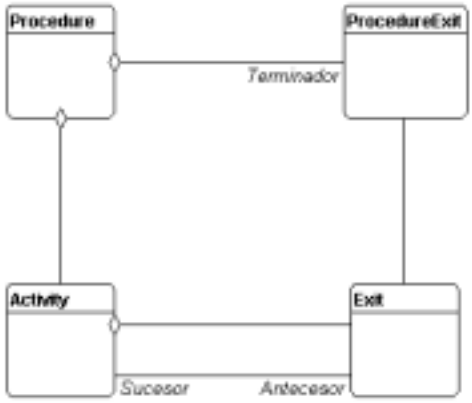

Imagen 13.2-4. Relaciones de Agregación del modelo

De esta manera no solo se evita el código de gestión de suscripciones sino que además también se es coherente con la semántica que se le había dado en otros sitios del framework a la relación de agregación y que no se había aplicado igualmente al sistema de eventos.

13.2.1.2 Suscripciones Basadas en Rutas

La aplicación de esta propuesta debería estudiarse quizás como un caso general de la anterior en vez de como un complemento. La Suscripciones Basadas en Rutas permite la recepción de notificaciones no solo cuando cambie el estado de una entidad sino también cuando cambie el estado de alguna entidad relacionada mediante una ruta establecida.

Supóngase como ejemplo la siguiente vista que muestra el script de una actividad. A la hora de seleccionar un terminador (llamado “Fin Script” en la imagen) aparecen en una lista desplegable las salidas que tiene la actividad a la que corresponde dicho script para que indique a cual de ellas corresponde.

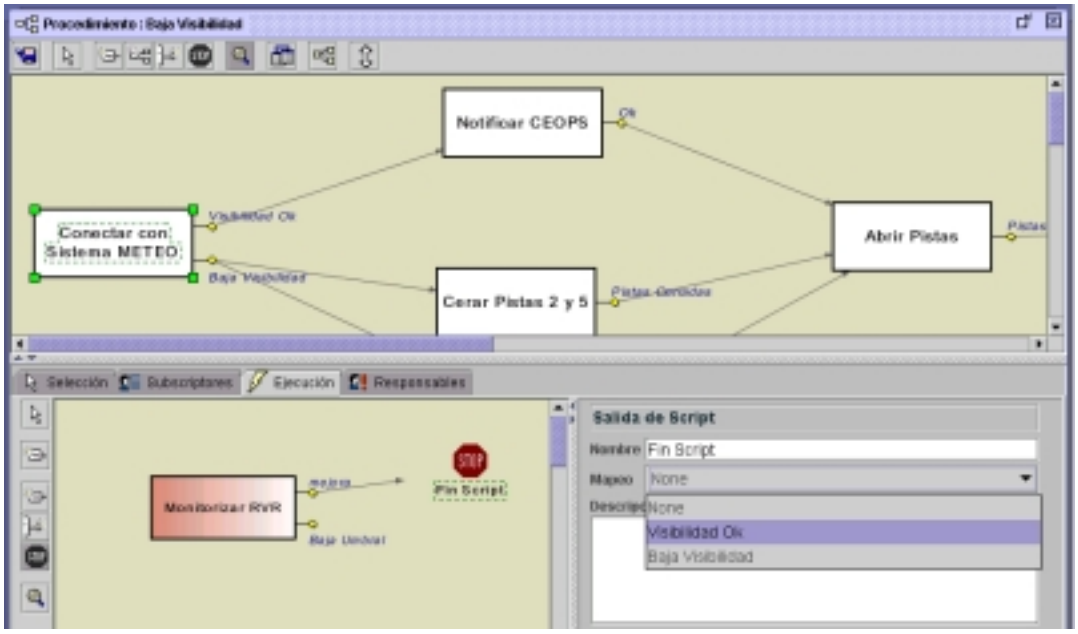

Imagen 13.2-5. Las salidas de un script se relacionan con la salida de la actividad a la que pertenecen

La vista está suscrita al script. Por tanto cada vez que se cambia el nombre de alguna de las salidas de su actividad la lista desplegable no se actualiza.

Lo que se necesita es una forma de poder suscribirse a *las salidas de la actividad del script*. Por tanto además de poder suscribirse a una entidad se necesita un mecanismo que permita suscribirse a todo objeto que esté dentro de una determinada ruta a partir del objeto en el que se ha realizado la suscripción.

```
// Se suscribe a las modificaciones del objeto script
script.addListener(vista);

// Se suscribe a las salidas de su actividad
script.addListener("activity.exit", vista);
```

Por supuesto esta suscripción es *dinámica* y no se limita a los objetos que estén relacionados con esa ruta *en el momento de la suscripción*. A medida que las relaciones cambien se actualizará el conjunto de objetos afectados. Cuando se añada una salida a la actividad automáticamente la vista del script recibirá notificaciones suyas sin que explícitamente se haya registrado en ella. De la misma manera dejará de recibir sus notificaciones cuando dicha salida deje de estar relacionada con la actividad a la que pertenece el script.

Nótese que si la ruta se forma mediante relaciones de agregación se estará obteniendo como caso particular el Sistema de Notificación Integral. Sin embargo quizás sea adecuado obtener este último automáticamente sin necesidad de una suscripción explícita a las rutas de agregación.

13.2.2 Demeter sobre RDM

La implementación actual de RDM incorpora un lenguaje de navegación muy simple consistente en enumerar los roles por los que navegar a partir de una entidad origen.

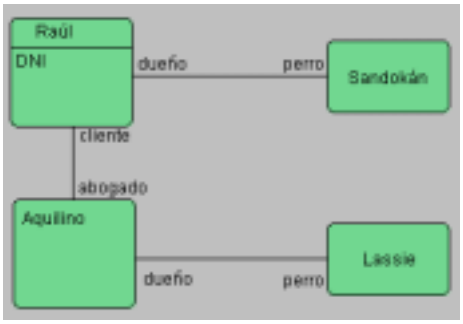

Imagen 13.2-6

Supóngase que sandokán quiere saber como se llama el perro del abogado de su dueño. La forma habitual de hacerlo sería:

```
sandokán.getPath("dueño.abogado.perro"); // {Lassie}
```

Sin embargo la implementación actual no permite:

- Indicar condiciones para seleccionar solo ciertas entidades del camino
- Permitir búsquedas sin tener que partir de una entidad conocida (por ejemplo buscar todos los clientes cuyo saldo sea deudor).

Por tanto como línea de investigación podría plantearse la adaptación de los lenguajes de consulta de objetos [Loomis94], o quizás sean más adecuados los de consulta de datos semiestructurados XML [Abiteboul99], XPATH [Kay01] o bien una adaptación de Demeter [Lieberherr96] sobre RDM.

13.3 Crear un Lenguaje RDM

Basándose en los principios de la Arquitectura RDM se ha implementado un framework en Java. Otra línea de investigación futura sería estudiar la conveniencia de llevar la arquitectura RDM a un lenguaje directamente, el cual estaría basado en lo que se denominará *Programación Orientada a Elementos* (EOP). Habría que estudiar la conveniencia de que fuera un traductor, un intérprete o incluso apoyarla sobre máquina abstracta de menor nivel que ofrezca las primitivas adecuadas.

En el campo de las máquinas abstractas, como parte de la investigación del grupo Oviedo3 [Álvarez96] y [Álvarez97], se implementó el prototipo de una máquina abstracta denominada Carbayonia [Izquierdo96]. Dicha máquina ya incluía en los objetos las relaciones de agregación y asociación. Posteriormente en [Ortín97]] se añadió reflectividad a dicha máquina lo cual permitió cambiar sus miembros dinámicamente. La arquitectura RDM pretende hacer el equivalente pero sobre lenguajes comerciales y basándose en patrones en vez de en una máquina abstracta.

También en la parte de traductores se utilizan actualmente numerosos patrones [Basanta00] [Acebal02]. Sin embargo se aplican de manera aislada y sin llegar a formar una arquitectura que les permita, por ejemplo, simplificar la implementación de la tabla de símbolos [Aho90].

En definitiva el nuevo lenguaje que pretenda dar soporte nativo a la arquitectura RDM debería incorporar:

- El Principio de Externalización. Que se avise de cualquier cambio en el estado de cualquier entidad sin necesidad de código explícito por parte de ésta. Deberá incluir también el Modelo de Notificación Integral y el Basado en Rutas.

- El Principio de Inconexión. Deberá permitir añadir relaciones dinámicamente y el mantenimiento automático de su integridad sin cambiar el código de las entidades.

- Soporte al acceso dinámico de miembros mediante VSP manteniendo la uniformidad entre atributos y relaciones (principios de Conformidad y Adyacencia).

- La incorporación de reglas de forma externa y la posibilidad de que rechacen la operación en curso.

- La reconstrucción automática de los objetos del modelo si se produce algún error o rechazo durante la ejecución de una operación. Las alternativas del programador no deben ser atrapar una excepción o que aborte la aplicación. Las alternativas en este nuevo lenguaje deben ser que *se ejecute correctamente la operación o se restaure el modelo*.

- Un lenguaje de consulta y navegación del modelo que aproveche la especialización de los objetos.

13.4 RDM como extensión de Metodologías Orientadas a Objetos

13.4.1 Introducción

En [Booch94] se define *metodología* como "una colección de métodos aplicados a lo largo del ciclo de vida del desarrollo del software y unificados por alguna aproximación general o filosófica". A su vez se define *método* como "un proceso disciplinado para generar un conjunto de modelos que describen varios aspecto de un sistema de software en desarrollo".

En el capítulo de Conclusiones (apartado 0) se dan una serie de métodos para para modelar los elementos del dominio. Estos, desarrollándolos de una forma mas sistemática y combinándolos con otros métodos existentes podrían completar una metodología de desarrollo, especialmente una metodología de las denominadas *ligeras* (como lo son Extreme Programming, Crystal, SCRUM, etc.)

13.4.2 RDM como complemento de Extreme Programming

Extreme Programming [Beck99] define la codificación como la actividad principal del proceso de desarrollo. Las demás tareas son los pasos necesarios hasta poder llegar a concretar en un lenguaje de programación lo que hay que implementar.

En el texto "Práctical Software Requirements" Kovitz [Kovitz99] define el proceso de desarrollo de software como:

> "Configurar la máquina M para producir los efectos R en el dominio D"

Siendo R los requisitos y D el dominio. Pero nótese que el objetivo es *configurar* una máquina o codificar. La actividad de obtener requisitos no debe perder esto de vista y deben dirigir la codificación de una forma *concreta*.

Sin embargo en las metodologías más extendidas ([Booch94] [Rumbaugh91] [Jacobson99]) el salto del diseño a código se da por hecho sin entrar en los problemas aquí expuestos. Se centran totalmente en el análisis y el diseño considerando la codificación como una actividad menor en comparación con las anteriores.

Kent Beck afirma en [Beck94] que

> "Si no *codificas* no has hecho el proyecto. Si no *testeas* no sabes cuando has acabado de codificar[36]. Si no *escuchas* al cliente no sabes qué testear. Y si no diseñas una *arquitectura adecuada* no podrás cambiar lo que escuches ni testearlo ni implementarlo *indefinidamente*"

Por tanto Beck afirma que una metodología de desarrollo tiene que contemplar y ayudar a estructurar los cuatro aspectos fundamentales del desarrollo:

- La Codificación.
- El Testing (como mecanismo de validación, o sea, de cumplimiento de requisitos; no solo de verificación).
- Cómo escuchar al cliente y extraer los requisitos.
- Cómo diseñar una arquitectura que soporte *el cambio de todos* los anteriores.

Generalmente en las metodologías citadas anteriormente se contempla adecuadamente las dos últimas fases (análisis y diseño) pero suponen menos importante los demás aspectos. Sin embargo se deben entrar a detallar el cómo codificar y cómo testear de manera integrada junto con el resto de las tareas. La tarea de programación no es menos importante que las demás. Es una fase más que si se menosprecia no se obtendrá un producto de calidad (si alguna fase no fuera importante se prescindiría de ella).

Extreme Programming, a diferencia de las demás, integra el testing durante todo el proceso de desarrollo. En el aspecto de codificación propone el refactoring. Sin embargo el refactoring supone afrontar todos los problemas expuestos inicialmente en esta tesis. En el diseño no se concreta como modelar las entidades, las relaciones ni las operaciones. Por tanto aquí se propone integrar en la metodología Extreme Programming la Arquitectura RDM junto con su codificación para completar los cuatro aspectos del desarrollo.

En definitiva se propone una metodología que ofrezca métodos para las cuatro partes que identifica Beck:

- Los métodos que forman parte del refactoring para tratar con la codificación.

[36] Esto no se refiere a los errores de [Myers79] sino a los requisitos del usuario. Recuérdese que XP utiliza los test como medio de comprobar el cumplimiento de los requisitos.

- Los métodos de Testing de [Hightower01].

- Los métodos del Planning Game para escuchar al cliente y extraer los requisitos inmediatos [Beck00]

- Los métodos de RDM expuestos anteriormente para formar la arquitectura que soporte el cambio de todos los anteriores durante la fase de desarrollo (especialmente del refactoring).

Sección 6. Apéndices

Para acabar el documento se ha incluido una sección sección final en la que se recogen dos apéndices.

- En el primero, titulado "El servicio de Contextualización" se explica con más detalle una de las líneas de investigación propuestas. Dicha línea fue extraída del capítulo 13, dedicado a tal efecto, debido a su extensión y para facilitar así no perder la visión de dicho capítulo centrándose demasiado en una línea específica.

- El último apéndice es el manual de usuario del Editor de Procedimientos del aeropuerto de Madrid-Barajas que ha servido como demostración de la validez de la arquitectura propuesta.

14 Apéndice A. El Servicio de Contextualización

14.1 Introducción

En este apéndice se mostrará un problema de transferencia de información en las notificaciones de sucesos aeroportuarios y la adecuación de una implementación distribuida de la Arquitectura RDM. Esta versión no está implementada actualmente y podría ser la base para una nueva tesis.

14.2 Objetivo

En un aeropuerto surgen con frecuencia sucesos que afectan a las distintas actividades en curso o que deben iniciar nuevas actividades. Para el adecuado funcionamiento del aeropuerto es fundamental que las notificaciones lleguen a las personas adecuadas junto con la información necesaria sobre el mismo para tomar las medidas pertinentes.

Por ejemplo cuando ocurre el suceso de baja visibilidad en la torre de control es necesario avisar al Servicio de Extinción de Incendios (SEI) ya que automáticamente deben salir los bomberos a las pistas (independientemente de que haya fuego o no). Ese mismo suceso de baja visibilidad también interesa al Servicio de Pista y Plataforma (SPP) ya que debe pasar al estado de alerta (con la realización de las actividades preventivas que ese estado supone). Pero con esa notificación debe llegar también la visibilidad actual en metros. De esa información dependerá, por ejemplo, que cierren determinadas pistas de aterrizaje. Es decir, ante una notificación de un suceso, distintos receptores *pueden necesitar distinta información* sobre el mismo.

Pero esto no solo debe considerarse para las personas directamente involucradas en un procedimiento. Por ejemplo un ejecutivo de servicio está interesado en averiguar cuantos avisos de baja visibilidad se producen al cabo de un mes. En función de los datos obtenidos prevé que podría plantear algún tipo de procedimiento optimizado para ciertas etapas del año. La notificación a esta persona no forma parte del procedimiento de baja visibilidad ya que no va a realizar ninguna labor inmediata. Pero ser avisado de dichos sucesos le será útil para realizar su trabajo de control.

Cada persona podrá seleccionar los eventos del aeropuerto en los que está interesado y podrá suscribirse aunque no participe en las acciones a tomar. De la misma manera esa persona podrá eliminar la suscripción si no obtiene datos relevantes y podrá suscribirse a otras notificaciones que intuya que puedan aportar ideas a la mejora del funcionamiento general o específico de parte del aeropuerto. Pero estas suscripciones deben ser ágiles y que *no supongan modificar los procedimientos* de suscripción o variar las propias notificaciones en función de las nuevas suscripciones.

En definitiva el objetivo es que *cualquier persona* pueda ser notificada de *cualquier suceso* que necesite de manera que pueda acceder a los datos relevantes, *desde su punto de vista*, del mismo.

14.3 Problemas

Supóngase que se está en el proceso de análisis de la aplicación que realiza las notificaciones de sucesos aeroportuarios. Concretamente se está intentado averiguar la secuencia de pasos a realizar cuando, por ejemplo, se rompe una cinta de facturación. El analista informático determina que en esos casos hay que avisar a las personas P1 y P2. Por tanto en el dominio aparece un nuevo evento denominado "Cinta Rota" y se determina que se les debe enviar un mensaje XML a estas personas en el cual irán los datos relativos al incidente. Pero... ¿qué etiquetas se pone en dicha notificación? Es decir ¿Qué información necesitarán P1 y P2 cuando se rompa una cinta de facturación?

Habría que entrevistar a P1 y a P2 para averiguar qué necesitan (la puerta de facturación, el terminal, el agente handling, etc). Al fin y al cabo en eso consiste la actividad de análisis. Una vez averiguados estos elementos se añade un campo para cada uno de ellos en el mensaje.

El problema es ¿todo suscriptor necesita siempre la misma información sobre un mismo suceso? Puede que en ciertas ocasiones necesiten saber, además, otro dato que no habían previsto. Por ejemplo supóngase que el empleado de mantenimiento encargado de arreglar la cinta recibe la notificación con el número de mostrador de facturación a la que pertenece (el dato que pidió al analista). Una ver finalizada la reparación, al volver a su puesto de trabajo, necesita contactar con el agente handling que estaba facturando en dicho mostrador. En la notificación no está ya que en el análisis nadie dijo que se necesitara. El encargado de mantenimiento tampoco podría justificar que a partir de ahora viniera esa información en el suceso, ya que es para una consulta puntual.

El problema por tanto no es solo la ausencia de atributos necesarios. Si, por ejemplo, se necesita saber qué compañía aérea ocupaba dicha puerta de embarque para cobrarles la reparación, como es una información que siempre será necesaria, se deberá añadir al mensaje a partir de ese momento. El problema es la información *que se necesita puntualmente* y que no tiene utilidad en el caso general. No sólo es imposible prever que vaya a ver casos impensables que requieran esta información sino que además, si se incluyeran todos, el mensaje sería un amasijo de datos demasiado extenso. Pero el acceso a estos datos permiten realizar las operaciones de forma ágil y son los que tienen que acabar siendo obtenidos por ese recolector de carencias del sistema informático que es el teléfono (lo cual implica saber a quién hay que preguntar, cual es su teléfono y que esté disponible en ese momento).

Aparte del problema de la información puntual hay otros problemas. Por ejemplo ¿qué pasa cuando ese suceso también haya que notificárselo a una tercera persona P3? El analista tendría que ir a hablar con él para saber qué información necesita que le aparezca en la notificación del suceso. Es decir, habría que hacer una matriz con todos los sucesos. Para *cada uno* de ellos identificar a todas las personas que lo van a recibir. Y a cada uno de ellos preguntarles qué información necesitan en cada caso. Finalmente habría que consolidar toda la información en un DTD que defina la estructura de ese suceso. A medida que alguien necesite más información se repetirá el proceso continuo de entrevistar y revisar el XML.

Pero es que, finalmente, gente que no se conoce puede suscribirse dinámicamente a los sucesos que les interese ¿Qué información necesitan todos ellos sobre el suceso? En definitiva se tiene un conjunto *variable* de sucesos a los cuales están suscritos un conjunto *variable* de personas que necesitan un conjunto *variable* de información sobre los mismos. ¿Cómo contentar a todos? ¿Cómo conseguir que no haya que estar modificando continuamente los XML y evitar además que ante la falta de información la gente vuelva a usar el teléfono para obtenerla (disminuyendo la utilidad de la inversión realizada en el sistema informático)?

Y aunque se pudiera averiguar toda esta información tampoco se habría avanzado mucho. Supóngase que por arte de magia se consiguiera obtener el conjunto de todos los datos que toda persona necesite ante un suceso. Se podría por fin declarar todo en un DTD. Pero hay otro problema mayor. Cuando alguien quiera *notificar que ha ocurrido* ese suceso ¿se le pregunta toda la información que todo el mundo requiera? ¿Y si no lo sabe? Supóngase, por ejemplo, que una agente handling detecta fuego. Coge su terminal portátil y se encuentra que para enviar esa notificación tiene que rellenar 30 campos (10 personas interesadas en 3 datos cada uno).

14.4 Requisitos de la Solución

El servicio de notificaciones desarrollado de la forma tradicional tiene los siguientes inconvenientes:

- La notificación de un suceso *debe conocer toda* la información que debe incorporar sobre el mismo.

- Lo mismo ocurre con las vistas que permiten introducir las notificaciones en el sistema (el terminal portátil del agente handling que detectó el fuego).

- Las vistas de una notificación *deben conocer* la información que necesita *un determinado* usuario.

Hay que eliminar todo este conocimiento para simplificar la implementación y promover su reutilización. La solución pasa por que a partir de la información que se *obtenga* inicialmente (sea la que sea) cada persona visualice la información que *necesite*. Que no se base en que todo el mundo tenga que predefinir lo que va a necesitar ni que se exija a alguien que rellene toda esa información. El que notifica el suceso dirá *lo que sepa* y a los interesados les aparecerá en pantalla *lo que necesiten o* bien *estará a su alcance*, debiendo ser en cualquier caso más cómodo y práctico que llamar por teléfono.

El sistema, para cada tipo de notificación, deberá tratar con la siguiente información:

- El emisor tendrá que incluir únicamente la información que conozca. El terminal no puede negarse a emitir una notificación de fuego porque el usuario no sepa exactamente donde es (pero si huelo a humo...). A la información que incluya el usuario en la notificación se la denominará conjunto A.

- El receptor debe poder acceder a toda la información que necesite. A la información que ese usuario requiere *en esa determinada situación* se la denominará B_{us}. Dado que B_{us} puede ser distinto en distintas recepciones *del mismo* tipo de suceso se define B_u como la unión de toda la información que pueda requerir ese usuario sobre ese

suceso ($B_u = \cup B_{us)}$. A su vez el conjunto de información que se necesitará obtener de ese suceso para todo usuario y en cualquier ocurrencia será denominado B ($B = \cup B_{u)}$.

- El sistema es por tanto el que debe proveer los medios para que se pueda cubrir las distancia entre los dos conjuntos A y B_{us}. Es decir, deberá aportar la información que el usuario necesita en esa notificación y que el emisor desconocía. Esta información forma el conjunto $C_{us} = B_{us} - A$ de cada receptor.

Los problemas surgen por que tradicionalmente *se obliga* a que el conjunto A tenga que ser igual a el conjunto B para intentar así eliminar el proceso que supone el tercer punto anterior. Sin embargo el conjunto B es demasiado grande y variable. Por tanto la solución debe tratar los conjuntos más pequeños A y B_{us}. Hay que plantear entonces como el emisor define el conjunto A; como el receptor define su conjunto B_{us} y como y quién llega a B_{us} a partir de A (es decir, como se aporta C_{us}).

14.5 Solución Propuesta

En el capítulo dedicado al proyecto H24 de AENA se describía brevemente el bus de objetos como parte de la arquitectura del proyecto. Este broker permitía añadir servicios que hicieran accesibles nuevos objetos, operaciones o fuentes de notificaciones a través del bus. Ante un nuevo proyecto se podía reutilizar el bus así como dichos servicios de manera independiente.

Un nuevo servidor candidato a ser añadir al bus es el *Servidor de Relaciones*. Este servicio administra las relaciones entre los distintos objetos remotos accesibles a través del bus. Es equivalente al framework RDM usado en el editor para implementar los modelos pero para objetos distribuidos. Este servicio, al igual que ocurría con la versión local del editor, trabaja con el rol *Entidad*. Para que un objeto pueda cumplir el rol de entidad basta con que tenga un identificador único global.

No se puede hacer un diagrama de clases de un aeropuerto. No se pueden determinar todos los objetos y sus relaciones. Se necesita una arquitectura que permita ir añadiendo tipos de objetos, servicios y relaciones a medida que se vayan identificando. Y los nuevos objetos deberán poder ser relacionados con los existentes pero sin afectar a estos últimos. El Bus se encarga de *permitir nuevos objetos* y servicios; el Servidor de Relaciones se encarga de *permitir relacionarlos* y navegar entre ellos de forma uniforme.

El servidor de relaciones, lógicamente, no conoce los objetos que relaciona. Eso permite que si se añade un nuevo servidor de objetos al bus automáticamente sus instancias se pueden relacionar con los objetos existentes. Nótese que los registros de una base de datos antigua en un *legacy system* transparentemente pueden ser relacionados con otros objetos sin necesidad de modificar la aplicación ni las tablas[37] (cosa que por otra parte no se podría o no sería rentable en la mayor parte de los casos). O incluso se podrían relacionar con objetos que estén en otro aeropuerto (ya que se pueden hacer buses virtuales que unifiquen objetos de distintas organizaciones). Y como todo esto es transparente las vistas pueden navegar por los objetos y obtener la información sin saber todo estos detalles de los mismos.

[37] Por supuesto si existe un servicio que cumpla el papel de conector y haga accesibles los registros a través de XML como se indica en [Linthicum99].

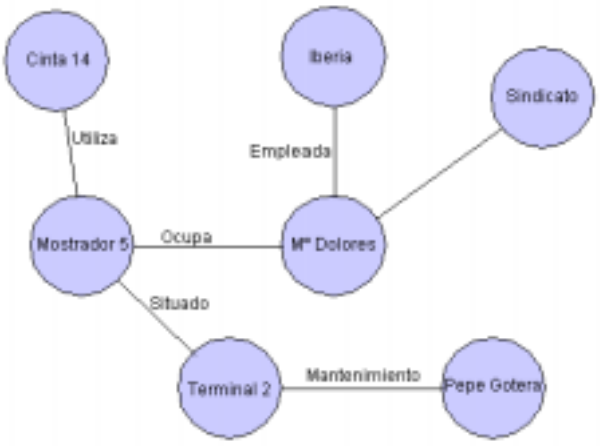

Imagen 14.5-1. Objetos del aeropuerto (posiblemente remotos) relacionados en el bus.

Volviendo al problema de las notificaciones XML ¿Cuál es el papel del servidor de relaciones dentro la solución al problema? El servidor de relaciones es la pieza central que permite independizar a las notificaciones, a los emisores y a los receptores.

Las notificaciones serán objetos del bus. Esto permitirá que otras notificaciones hagan referencia a ellas (por ejemplo una notificación que diga que un peligro ya ha pasado podrá referenciar – mediante una relación - a la notificación que avisó del inicio del peligro). Pero lo que es más importante es que su contenido será a su vez un conjunto variable de identificadores de otros objetos del bus. Es decir, la notificación implementará el Variable State Pattern, o lo que es lo mismo, no tiene un conjunto fijo y predefinido de atributos.

14.5.1 Contextualización de los Sucesos

¿Cómo obtener el conjunto A? Supóngase que un emisor quiere enviar al bus una notificación de "cinta de facturación rota". La pantalla que permite enviar notificaciones, una vez elegida la notificación que se desea enviar, le permite elegir aquellos objetos del bus que quiera relacionar con la notificación (situación, objetos afectados, etc). Cuanta más información se quiera dar mejor pero no se está obligado a ello. En cualquier momento se puede enviar la notificación con la información que se sepa o incluso vacía. De todas formas, para un conjunto reducido de eventos habituales, pueden hacerse plantillas que recomienden los campos a introducir (pero *únicamente* como guía, no como obligación).

Una vez emitida la notificación se tratará de *contextualizar* el suceso para aumentar el conjunto A. Contextualizar consiste en *ampliar la información que se tiene de un suceso*, es decir, definir el contexto o la situación en que sucedió. Si, por ejemplo, se rompe una cinta de facturación contextualizar dicho suceso sería añadir la información de que ha sido en el Terminal 3, es decir, se indica en qué contexto sucedió esa rotura, o lo que es lo mismo, que circunstancias lo rodearon.

Hay dos formas de contextualización:

- La *Contextualización Automática* se basa en utilizar las relaciones existentes en los objetos incluidos en la notificación para añadir más datos a lo que ha dicho el emisor. Por ejemplo, basándose en la situación de la figura anterior, cuando Mº Dolores diga que se ha estropeado una cinta de facturación no será necesario que

diga mas datos para saber que actualmente está en el mostrador 51 y que la cinta rota entonces es la número 14.

El problema con la contextualización automática es ¿Qué relaciones hay que seguir? ¿Con qué nivel de profundidad? Es decir, no se trata de aplicar recursivamente el proceso para que a partir de un objeto se saquen todos los relacionados directa o indirectamente con él. Serían todos los objetos del aeropuerto. Podrían ponerse límites del tipo "incluir todos los objetos que estén a dos relaciones de distancia del objeto incluido en la notificación". Sin embargo, aunque es mejor que nada, unas veces sobrarían objetos y otras faltaría profundidad en alguna de las ramas.

- Otra forma de ampliar el conjunto A es mediante la *Contextualización Colaborativa*. Hay que recordar que el sistema va a ser usado por distintos grupos de personas que quieren acceder a la información del suceso pero que también pueden tener información adicional que puedan compartir. El *rol de emisor no debe quedar fijado* en la persona que envió la notificación. Cualquiera puede tomar ese rol momentáneamente para ampliar la información. Consiste en hacer algo parecido a lo que hacen los informativos de televisión cuando están siguiendo una noticia en directo. Cada vez que se recibe un nuevo dato se añade en tiempo real a la información que se tenía. Pero no se espera a saber quien ha sido el ladrón para informar del robo. Lo mismo ocurre aquí: lo importante es avisar del suceso y ya se irá ampliando la información.

  Pero ¿qué ocurre cuando varias personas añaden la misma información a un suceso? Por ejemplo supóngase que se recibe una notificación de fuego pero no se dice donde. Varios receptores a la vez añaden esta información ¿No produciría esto gran cantidad de información redundante en cada suceso? La contextualización colaborativa se basa en que los sucesos son objetos del bus. Por ello varias notificaciones pueden hacer referencia a un mismo suceso indicando que se ha ampliado la información. Pero las operaciones de modificación se dirigen a este objeto y es éste el que comprueba si ya la tiene. La operación de añadir información no son mails que se envíen a todo el mundo; se aplican sobre el objeto. Es este el que emitirá un evento si ha cambiado de estado. Por tanto las operaciones para añadir una misma información no cambian el estado del objeto y no producirán aviso a los suscriptores; sólo se les avisará cuando haya alguna noticia *nueva* sobre el suceso.

Por tanto se ha liberado al emisor de la *responsabilidad* de conocer y adjuntar toda la información del suceso. Esta responsabilidad se ha repartido entre un proceso automático basado usar las relaciones disponibles (contextualización automática) y la colaboración de cualquier persona que pueda aportar dicha información (contextualización colaborativa).

14.5.2 Declaración de B_{us}

Una vez obtenido el conjunto A (por el emisor, automáticamente o de forma colaborativa) queda saber como declarar B_{us} y obtener C_{us}, es decir, cómo dice el receptor qué información quiere que le aparezca cada vez que se produzca el suceso y quién la obtiene.

Obviamente ellos son los únicos que conocen B_{us}. Está claro que no pueden comunicar ese conjunto a una tercera persona para que les adapte su interfaz de usuario. No sería un proceso ágil por lo que dejaría de usarse. La única solución por tanto es que sean *ellos mismos* los que seleccionen la información C_{us} que quieran sobre cada suceso en *cada momento*. No puede haber un analista del sistema que esté en una fase infinita de análisis preguntando constantemente si alguien quiere más información. En algún punto se pasaría a

la implementación y al día siguiente la información que se presentara ya no es la que se quiere.

Por tanto deben ser ellos mismos los que cojan lo que quieran. Supóngase la siguiente situación.

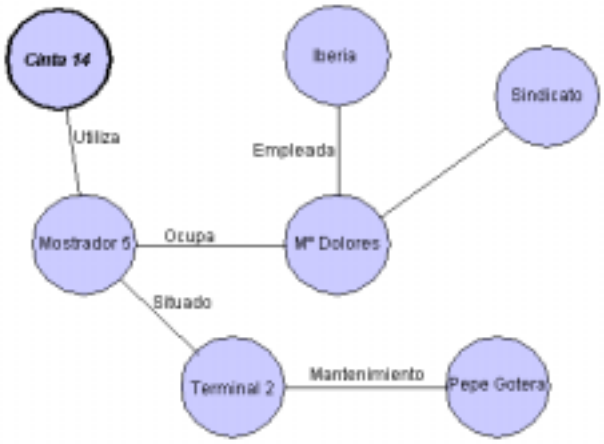

Imagen 14.5-2

Cuando la cinta se rompe la agente handling notifica simplemente que se ha estropeado la cinta 14. Como se comentó anteriormente las notificaciones de cualquier suceso estarán formadas por un identificador del suceso seguidos por los identificadores de uno o más objetos del bus. En este caso, por tanto, la notificación trae el identificador de la cinta 14. Al llegar a los receptores lo que se les presenta con la notificación es una *vista* reducida del espacio global de objetos del bus; una vista en la que sólo están los objetos referenciados en la notificación; el conjunto A (en este caso sería el objeto en negrita de la imagen). Lo que haría cada usuario es ir expandiendo esta vista siguiendo las relaciones que le interesen.

Supóngase, por ejemplo, como sería la presentación de la notificación en HTML [Beners93]. Aparecería la notificación con la descripción de objeto al que referencia (la cinta 14) y la de los objetos con los que está relacionado en un primer nivel. Es decir, se le pediría al servidor de relaciones todas las relaciones que tuviera la "cinta 14". Éste devolvería una relación [cinta14, utiliza, mostrador5]. Con esto se obtiene el objeto "mostrador 51", el cual se presenta en la página HTML. Lo que se hace también es presentar las relaciones que parten desde los objetos mostrados de tal manera que el usuario pueda navegar a otros objetos a partir de ahí.

Notificación: ***Cinta Rota***
Cinta: Cinta14. <atributos de la cinta 14> **Mostrador**: Mostrador 51. <atributos del mostrador 51>
Información relacionada Ocupado por: Mº Dolores Situado en: Terminal 2

Al pinchar sobre alguno de los links se repetiría el proceso pero centrándose en el objeto seleccionado. Así, por ejemplo, al pinchar sobre Mª Dolores aparecería el link hacia Iberia. Así cada usuario contextualiza los sucesos (es decir, añade más información a su vista de la notificación) de la forma que le resulte conveniente en *ese* momento.

De esta manera el espacio de objetos puede verse como un conjunto de páginas (los objetos) con links entre ellos (las relaciones). A partir de la información que reciba cualquier persona podrá navegar hasta la información que quiera sin que el que emite el mensaje tenga que dárselo ni el receptor tenga que declararlo de antemano. En definitiva puede considerarse un sistema hipertexto ya que cumple todas las condiciones que da Nielsen [Nielsen95].

14.6 Reutilización de Vistas

El ejemplo presentado anteriormente es simplemente una sugerencia de presentación. Otras podrían ser:

- Cada vez que se seleccione un objeto éste se añade a los que hay en pantalla, pero no se retiran los anteriores. Eso permitiría navegación visual entre lo que ya se ha visto sin necesidad de tener que recordar por donde volver a un determinado objeto.

- Podría representarse en modo gráfico con círculos como se muestra en la figura superior. Cada vez que selecciona una entidad se añaden con las que está relacionada, ampliando de esta forma la vista de las relaciones. Esto proporcionaría sensación de localización [Flemming98].

- También podría haber vistas no gráficas. Por ejemplo podría usarse una consola de texto que pudiera emitir consultas (o incluso estar integrada en otras vistas gráficas) de tal manera que se pudiera usar el sistema de relaciones a través de un lenguaje simple de navegación. Por ejemplo "mostrador5.utiliza.situado.mantenimiento" devolvería a "Pepe Gotera".

Podría argumentarse que aquí se presenta el problema de usabilidad de tener que recorrer las relaciones una y otra vez para obtener siempre la misma información B_{us} ante el mismo

evento. Sin embargo esto se soluciona haciendo que la vista *recuerde* las relaciones que navegó el usuario en la última recepción de una notificación del mismo tipo. Por tanto ya se le presentan los datos de la notificación junto con los de las relaciones usadas en la última vez. De esta manera sólo tendrá que navegar si quiere algún dato nuevo. Es decir, el usuario se crea mediante personalización su propia vista. No es necesario una herramienta de creación de pantallas para que el usuario decida lo que quiere (no la utilizaría) ni hace falta personal informático dedicado a crear las distintas pantallas para los distintos usuarios.

Aquí es donde tiene su sentido la *Contextualización Automática*. El problema que tenía este tipo de contextualización era ¿ qué relaciones hay que navegar y hasta que nivel hay que profundizar? Ahora la respuesta es sencilla: *las mismas que se navegaron la última vez*. La contextualización automática no hay que aplicarla para una notificación; no se sabe hacia donde navegar. Hay que aplicarla a una tupla formada por *una notificación* y *un usuario*. Por tanto cuando se genera un nuevo suceso se aplica *una contextualización colaborativa* para todos y *muchas contextualizaciones automáticas* (una por cada usuario que visualiza la notificación).

Por tanto el problema queda reducido a la primera recepción de un tipo de suceso (ya que na ha habido una vez anterior). Pero incluso esto también se puede mitigar haciendo plantillas para las personas más importantes. Estas plantillas no serían más que una primera navegación (hecha por el personal informático con una notificación de prueba) que quedaría así grabada para dichas personas y que indicara por tanto las relaciones que se quieren navegar para que automáticamente les salga en la primera notificación real.

En cualquier caso elegir la forma más adecuada de presentar la información y de personalizar la vista sería tema de otra tesis que se centrara en el estudio de la usabilidad en la búsqueda y presentación de información en entornos aeroportuarios. Lo que aquí se pretende es que una vez hayada esa forma usable de localizar la información, sea la que sea, pueda ser *reutilizada con otros modelos de datos en otros dominios*. A las vistas se les ha proporcionado una capa de abstracción para acceder a los objetos y a las relaciones independientemente de cuales sean estos. Por tanto estas vistas podrán usarse con otras estructuras de objetos en otro dominio.

15 Apéndice B. Manual de usuario del Editor

15.1 Introducción

El presente documento se inscribe en el marco del Proyecto H24, expediente Nº: 1893/01 titulado "Desarrollo de una Herramienta de Ejecución y Seguimiento de Procesos en Tiempo Real" desarrollado por Comercio Electrónico B2B 2000 S.A. para AENA – Madrid Barajas.

El presente documento detalla el manual de usuario del Editor Gráfico de Procesos. Su lectura es necesaria para una correcta utilización del producto.

15.2 Criterios de Creación del Editor

El Editor Gráfico de Procesos ha sido diseñado por Comercio Electrónico B2B 2000 S.A. AENA-Madid Barajas bajo los criterios de sencillez y extensibilidad.

Criterio de sencillez

> El editor se ha diseñado para facilitar la interacción con el usuario, y para que la adaptación al mismo se realice en un intervalo de tiempo reducido.

Criterio de extensibilidad

> El editor está preparado para que el usuario añada nueva información a procedimientos y procesos de forma transparente. Cualquier información extra que el usuario quiera añadir podrá hacerlo mediante herramientas sencillas.

15.3 Edición del Lenguaje MADML

15.3.1 Pantalla principal

Al iniciar la aplicación se mostrará la siguiente pantalla. Desde ella podremos abrir y crear procesos como procedimientos. Podemos hacerlo desde dos puntos, el primero es la barra de herramientas y el segundo es desde el menú del editor.

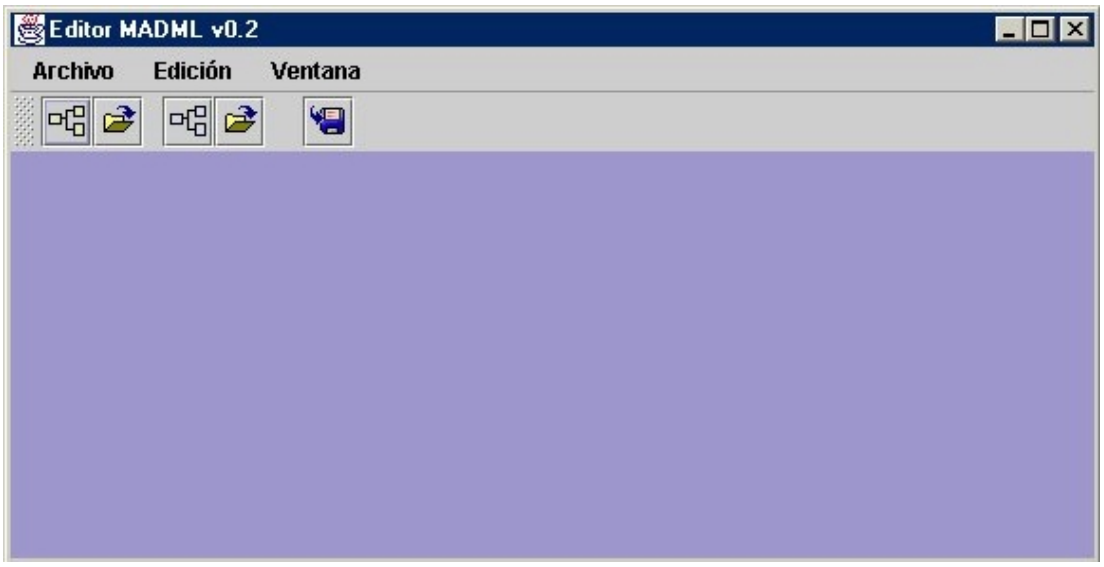

Ilustración 15.3-1: Pantalla principal

15.3.1.1 Creación y apertura de procesos

Desde la barra de herramientas podemos abrir y cerrar procesos de la siguiente forma:

Ilustración 15.3-2:creación, apertura procesos

El primer botón crea un proceso y el segundo botón abre una lista de procesos para que el usuario escoja uno.

15.3.1.2 Creación y apertura de procedimientos

Desde la barra de herramientas podemos abrir y cerrar procedimientos de la siguiente forma:

Ilustración 15.3-3:creación, apertura procedimientos

El primer botón crea un procedimiento y el segundo botón abre una lista de procedimientos para que el usuario escoja uno.

15.3.1.3 Guardar todos los procesos abiertos

El usuario puede guardar todos los procesos y procedimientos que tenga abiertos en el editor. El botón que debe pulsar es el siguiente:

Ilustración 15.3-4: Guardar ventanas activas

15.3.1.4 Cerrar editor

Para cerrar el editor se debe ejecutar la opción "*Salir*" del menú Archivo:

Ilustración 15.3-5: Salir del editor

Una vez que seleccionada esta opción, se pregunta al usuario si quiere archivar cada una de los procesos y procedimientos que tiene activos.

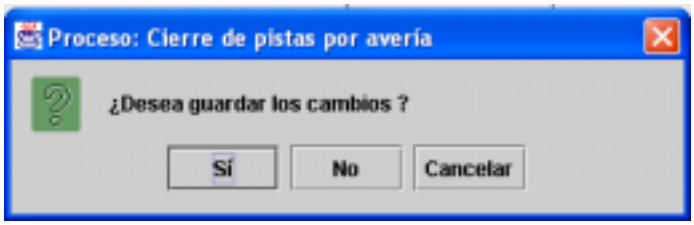

Ilustración 15.3-6: Guardar cambios

15.3.1.5 Ventanas activas

El usuario puede ver y seleccionar cualquiera de las ventanas que está editando actualmente. Para ello deberá abrir el menú *Ventana* y escoger la ventana a abrir.

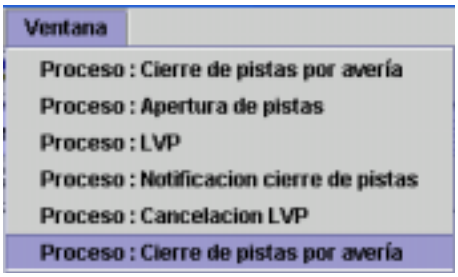

Ilustración 15.3-7: Ventanas activas

15.3.2 Procesos

15.3.2.1 Abrir un proceso

Cuando pulsamos el botón de apertura de proceso, aparece la pantalla que se muestra a continuación. El usuario puede desplazarse por la lista de procesos almacenados y, posteriormente realizar la selección.

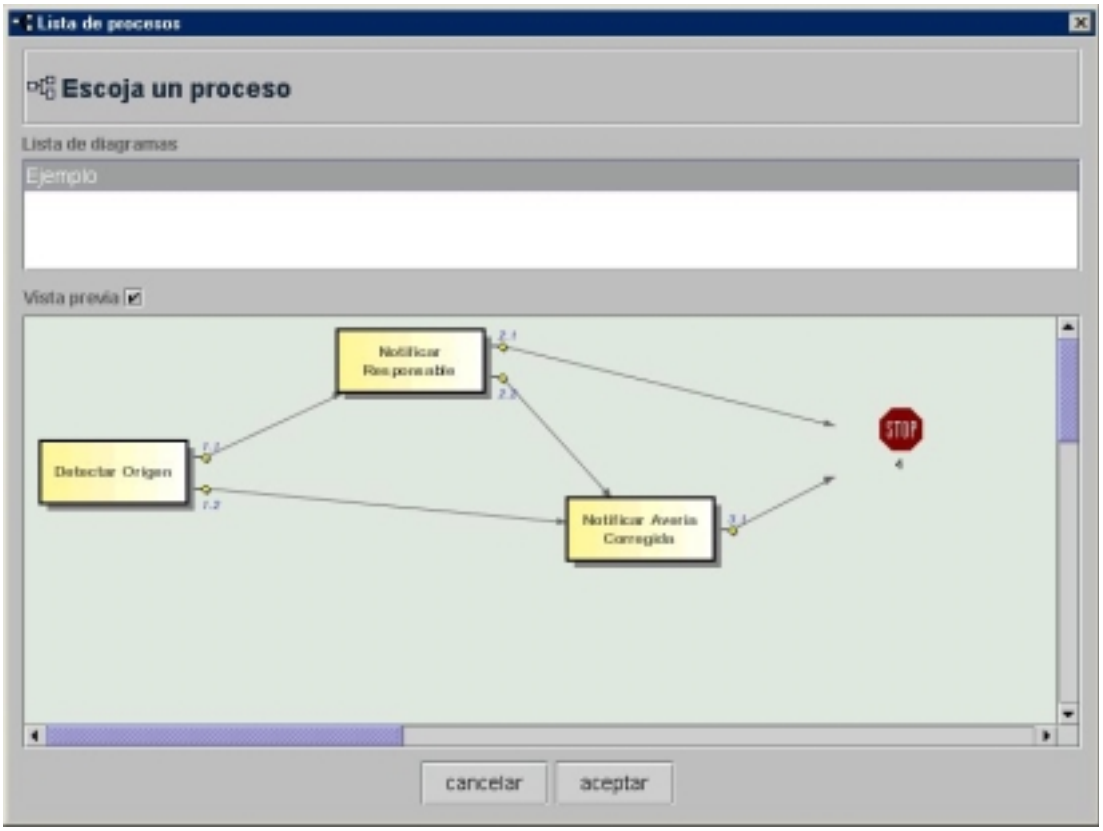

Ilustración 15.3-8: Apertura de un proceso

Al mismo tiempo, mientras itera sobre la lista, podrá ver una vista previa del proceso a editar. Si esto fuese muy costoso en tiempo, o si el usuario no desea mostrar dicha vista, sólo tendrá que desactivarla:

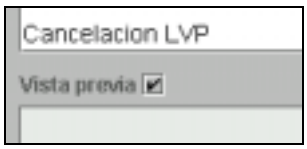

Ilustración 15.3-9: Vista previa

15.3.2.2 Editar propiedades de un proceso

Todo proceso tiene un nombre y una descripción. Desde el panel que se presenta a continuación se puede cambiar el nombre y la descripción del proceso. Para ello sólo debe situarse el cursor sobre el espacio del nombre o la descripción, y escribir el texto deseado.

Para que se active este panel, se ha de pulsar con el ratón sobre el proceso, en cualquier punto en el que no haya ningún subproceso, salida o relación.

Ilustración 15.3-10: Editar las propiedades de un proceso

15.3.2.2.1 Tags no tratados

El editor está preparado para añadir información nueva a procesos y procedimientos (no contemplada todavía en la especificación del lenguaje MADML).

Para ello se deberá pulsar sobre el botón *Tags no tratados*, y escribir la información deseada. Una vez escrita pulsar *aceptar*.

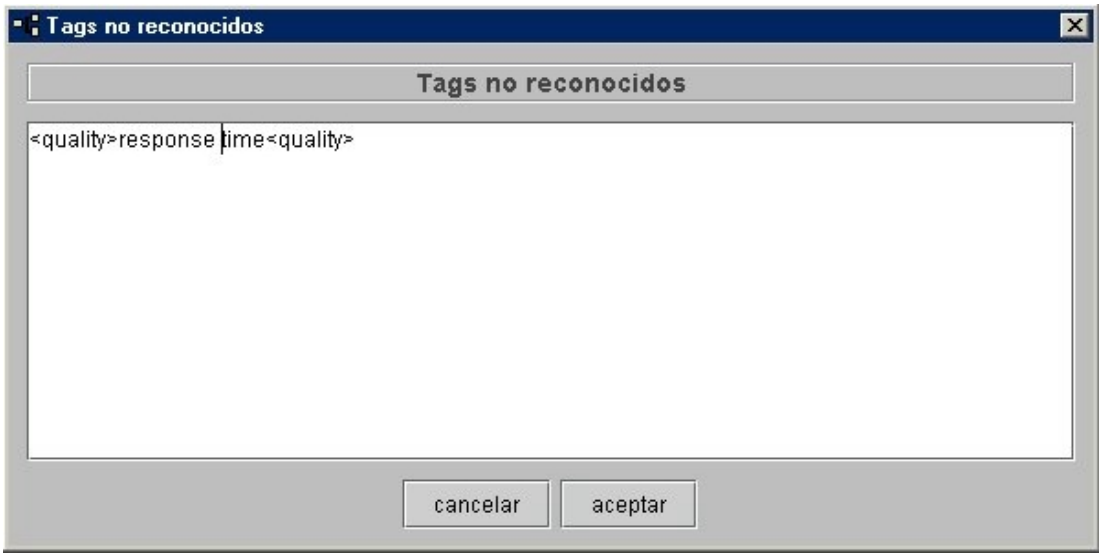

Ilustración 15.3-11: Tags no tratados

15.3.2.3 Modificar el proceso

Para modificar el proceso se utiliza la siguiente barra de herramientas:

Ilustración 15.3-12: Barra de herramientas de proceso

En ella podemos ver los siguientes elementos:

15.3.2.3.1 Barra de herramientas

 Graba el proceso actual

 Herramienta de selección

 Herramienta de creación de subprocesos

 Herramienta de creación de salida de subproceso

 Herramienta de creación de salidas de proceso

 Zoom de proceso

 Creación de ficha de procedimiento

 Crear procedimiento a partir de proceso

 Ampliar información de proceso

 Amplia proceso

 Muestra información de proceso y su esquema asociado

15.3.2.4 Zoom

Pulse en el botón que tiene una lupa para cambiar el zoom del proceso, a continuación seleccione el nivel.

15.3.2.5 Creación de subproceso

Pulse en la herramienta de creación de subproceso, a continuación seleccione el lugar deseado para insertarlo.

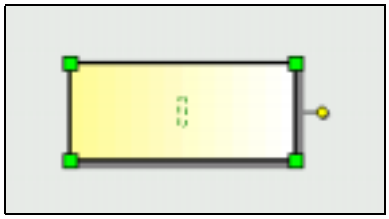

Ilustración 15.3-13: Subproceso

15.3.2.6 Creación de salida de subproceso

Pulse en la herramienta de creación de salida de subproceso, y a continuación seleccione el subproceso al que desea añadir la salida:

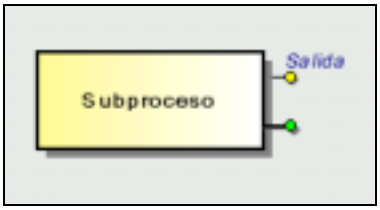

Ilustración 15.3-14: Salida de subproceso

15.3.2.7 Creación de salida de proceso

Al igual que en la creación de un subproceso, pulse en la herramienta de creación de salida de proceso, y a continuación seleccione el lugar donde desea situarla:

Ilustración 15.3-15: Salida de proceso

15.3.2.8 Conexión entre figuras

Para conectar dos figuras, pulse sobre la salida de un subproceso, y a continuación (sin soltar el botón) pulse sobre la figura a la que desea conectarla:

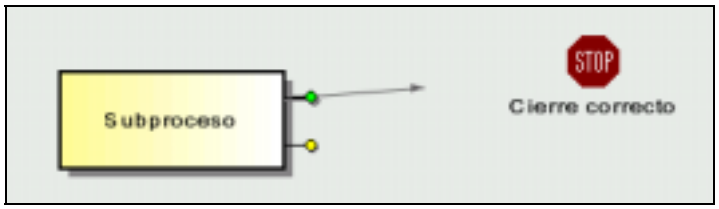

Ilustración 15.3-16: Conexión subproceso, salida

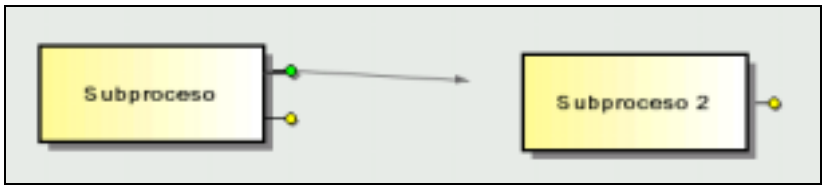

Ilustración 15.3-17: Conexión entre subprocesos

15.3.2.9 Ficha de proceso

En cualquier momento podemos ver la ficha del proceso que se está editando. Para ello pulse sobre la herramienta "*ficha de proceso*", y se mostrará el detalle del proceso.

15.3.2.10 Borrado de entidades

Para borrar las entidades del proceso, se deben seleccionar y después pulsar el botón *suprimir*.

15.3.2.10.1 Selección multiple

Puede seleccionar varias entidades al mismo tiempo para ser borradas. Hay dos maneras: 1) pulse y arrastre el ratón hasta englobar mediante el cuadro de selección las entidades deseadas, 2) mantenga pulsada la tecla shift mientras selecciona varias figuras.

15.3.3 Procedimientos

15.3.3.1 Abrir un procedimiento

La forma de abrir un procedimiento es la misma que la de un proceso. Se muestra al usuario un diálogo con una lista de procedimientos y su vista previa asociada (ésta puede ser desactivada por el usuario si lo desea)

15.3.3.2 Barra de herramientas

A continuación se presenta una descripción de los elementos que componen la barra de herramientas que nos permite actuar sobre el modelo.

 Graba el proceso actual

 Herramienta de selección

 Herramienta de creación de actividad

 Herramienta de creación de salida de actividad

 Herramienta de creación de actividad de invocación

 Herramienta de creación de salidas de proceso

 Zoom de proceso

 Creación de ficha de procedimiento

 Edita propiedades procedimiento

15.3.3.3 Editar propiedades de un procedimiento

Para editar las propiedades de un procedimiento, el usuario deberá pulsar el botón de edición de propiedades de procedimiento. A continuación se presentará la siguiente pantalla:

Ilustración 15.3-18: Edición de propiedades de procedimiento

Desde este panel podemos cambiar el nombre, la descripción, los parámetros de entrada y los resultados de ejecución del procedimiento. También podemos, al igual que en el resto de entidades, añadir y leer información no tratada por el momento por el editor.

Para añadir un parámetro o un resultado pulsaremos el botón que tiene un "+" en verde. A continuación se nos mostrará el siguiente diálogo:

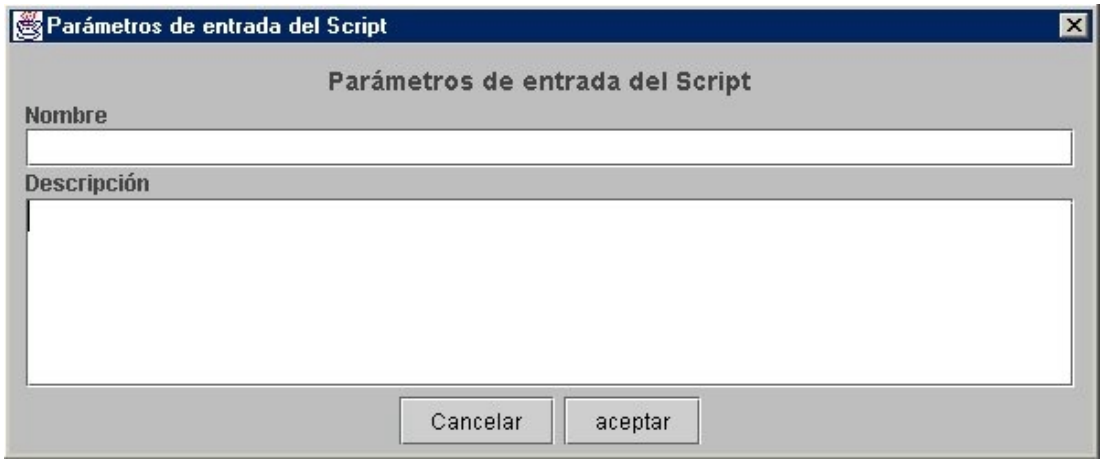

Ilustración 15.3-19: Parámetro de entrada

Introduciremos el nombre y la descripción, y pulsaremos el botón aceptar. En ese momento, la información extra ha sido añadida; si el usuario desea cambiarla, sólo tendrá que volver a pulsar el botón "*Tags no tratados*" y realizar los cambios pertinentes.

Para borrar un parámetro o resultado lo seleccionaremos y pulsaremos el botón "-"

15.3.3.3.1 Mapeo del procedimiento a un proceso

Para mapear el procedimiento a un proceso pulsaremos sobre el botón que tiene tres puntos; podemos verlo en la siguiente figura.

Ilustración 15.3-20: Mapeo de procedimiento a proceso

A continuación se mostrará al usuario un diálogo de selección proceso (el mismo que el de abrir proceso) y se solicita al usuario que escoja uno.

15.3.3.4 Creación de una actividad

Pulse en la herramienta de creación de actividad y a continuación seleccione el lugar deseado para insertarla:

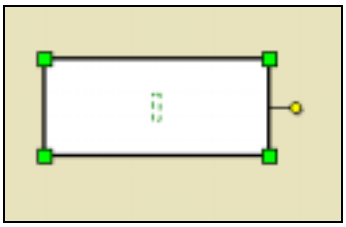

Ilustración 15.3-21: Creación de actividad

15.3.3.5 Creación de una actividad de invocación

Pulse en la herramienta de creación de actividad de invocación. Se le preguntará entonces si desea seleccionar el procedimiento a invocar. Si elige sí, podrá escoger procedimiento del diálogo principal de abrir procedimiento, sino elegirá el nombre de manera maual:

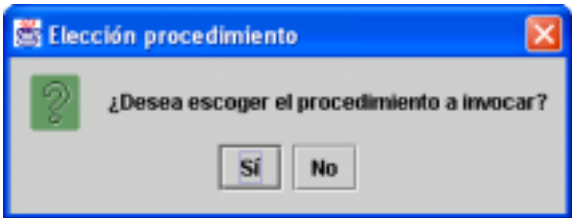

Ilustración 15.3-22: Elección del procedimiento a invocar

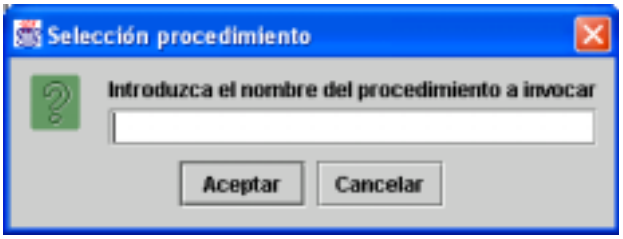

Ilustración 15.3-23: Elección manual del procedimiento

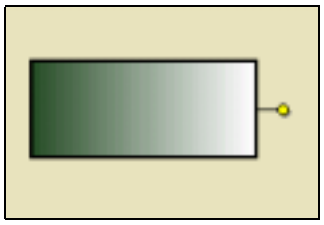

Ilustración 15.3-24: Actividad de invocación

15.3.3.6 Creación de una salida de actividad

La forma de crear una salida de actividad (o actividad de invocación) es la misma que en el subproceso; se pulsa sobre la herramienta y después sobre la actividad.

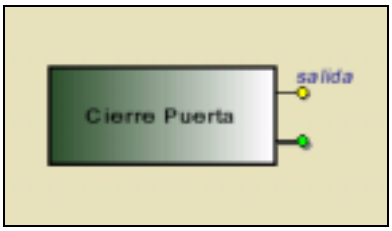

Ilustración 15.3-25: Salida de actividad

15.3.3.7 Creación de una salida de procedimiento

Se pulsa sobre la herramienta de creación de salida de procedimiento, y a continuación se selecciona el lugar donde colocarla:

Ilustración 15.3-26: Salida de procedimiento

15.3.3.8 Editar propiedades de actividad

Para editar las propiedades de una actividad disponemos del siguiente panel:

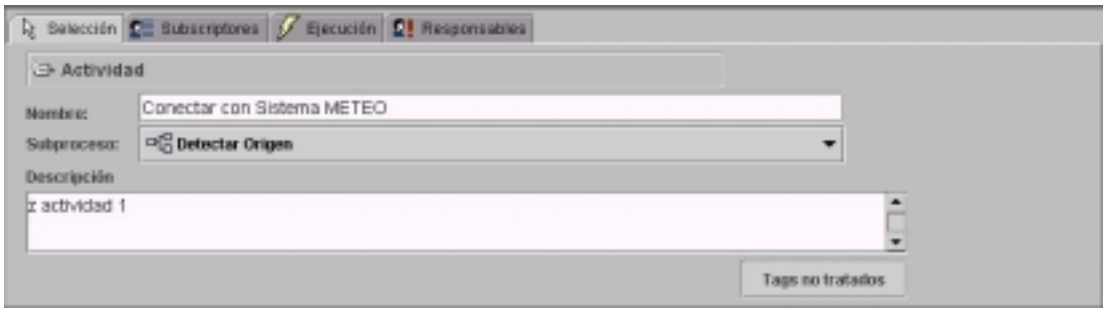

Ilustración 15.3-27: Editar propiedades de actividad

Desde este panel podemos cambiar el nombre o la descripción de la actividad. Además podemos seleccionar el subproceso al que se mapea esta actividad. Para ello desplegaremos la lista de subprocesos disponibles.

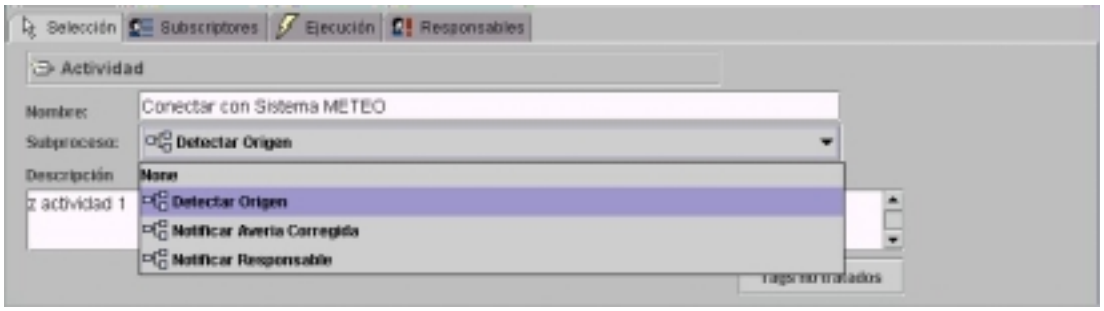

Ilustración 15.3-28: Mapeo de subproceso

15.3.3.9 Editar propiedades de actividad de invocación

Las actividades de invocación disponen del siguiente panel para editar sus propiedades:

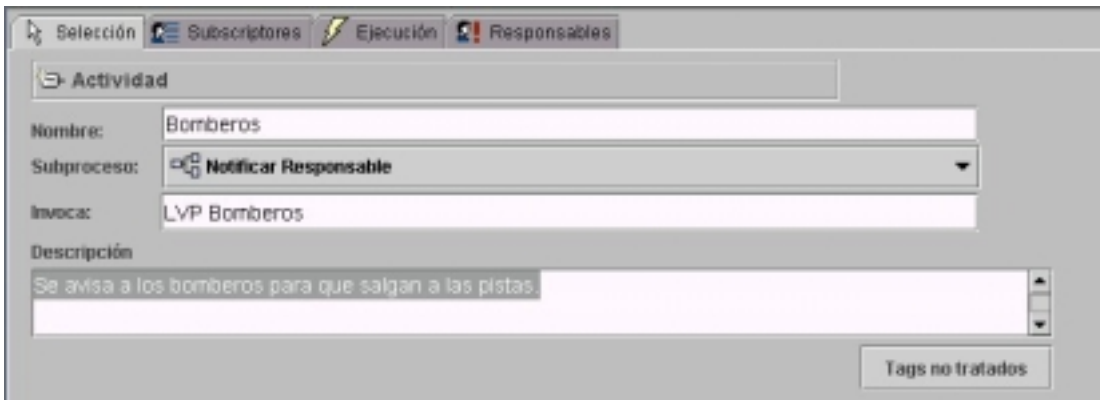

Ilustración 15.3-29: Panel edición actividad de invocación

Al igual que en las actividades normales, podemos cambiar el nombre, la descripción, y mapear la actividad a un subproceso. La única diferencia con el panel de edición de una actividad normal, es la selección del procedimiento de invocación.

Ilustración 15.3-30: Selección procedimiento invocado

En el panel anterior podemos ver como esta actividad invoca al procedimiento LVP. En el caso de una invocación manual de un procedimiento podremos cambiar el nombre pulsando sobre el botón de la derecha (tres puntos) Una vez pulsado se abrirá un diálogo en el que se nos preguntará el nuevo nombre del procedimiento.

15.3.3.10 Subscriptores de actividad

Para ver y cambiar los subscriptores de una actividad se recurrirá al siguiente panel. Cuando pulsamos el botón "Cambiar" aparecerá el siguiente diálogo.

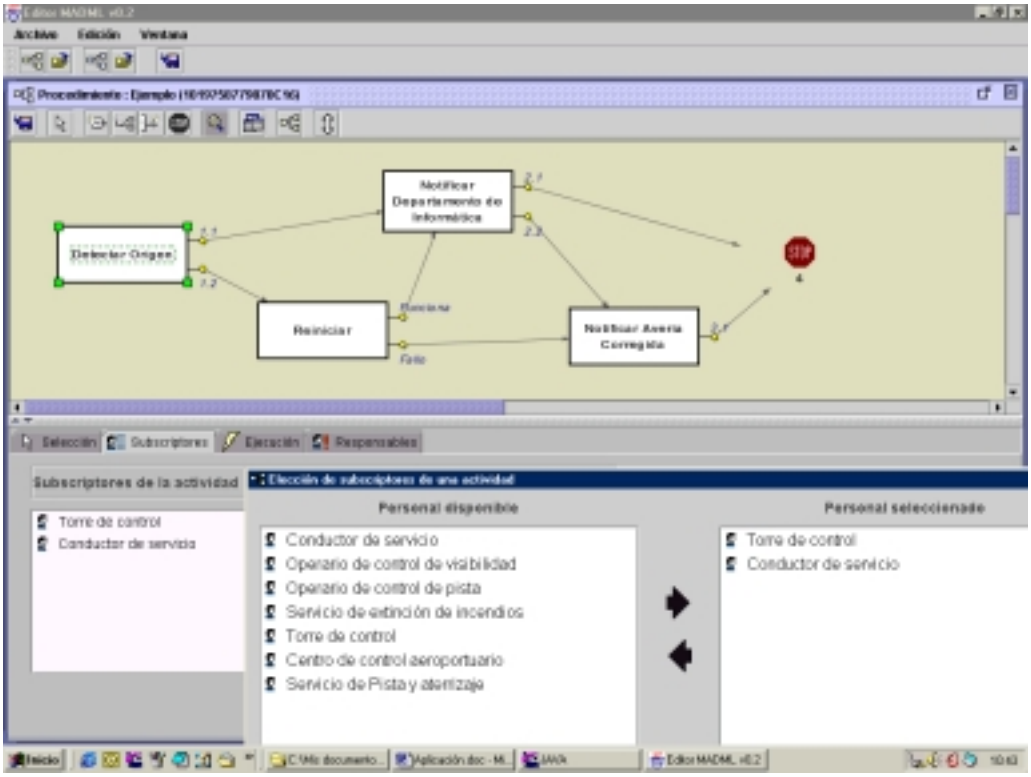

Ilustración 15.3-31: Subscriptores de actividad

Pulsando sobre las dos flechas podremos seleccionar y deseleccionar los subscriptores de la actividad.

15.3.3.11 Script de ejecución de actividad

Cada actividad tiene un script de ejecución propio. La forma de crear proclets y salidas es idéntica a la del procedimiento. Pulsando sobre las diferentes figuras podremos editar (mediante el panel de la derecha) sus propiedades.

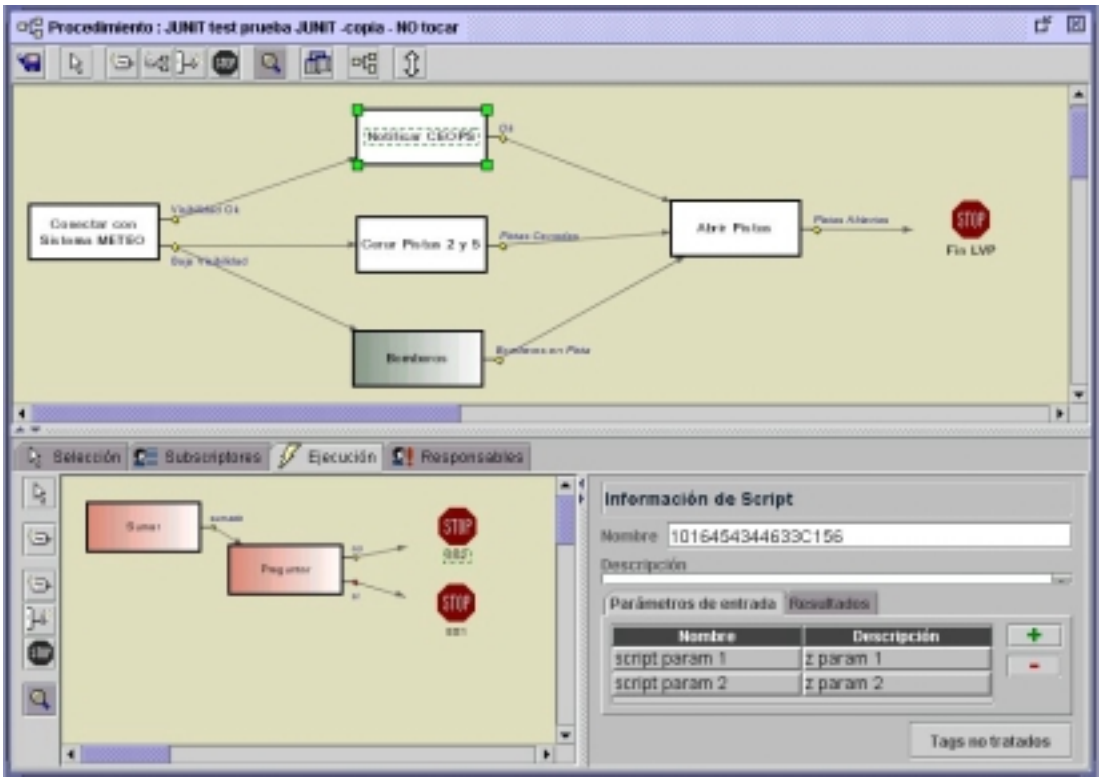

Ilustración 15.3-32: Script de ejecución

15.3.3.11.1 Editar propiedades de un script

Todo Script tiene unos parámetros de entrada y unos resultados. Pueden editarse, al igual que en las propiedades de un procedimiento, mediante la tabla que aparece a la direcha en la figura anterior.

Si deseamos añadir o quitar un parámetro sólo deberemos pulsar los botones identificados con los signos “+” y “-“.

Para ver los resultados del Script pulsaremos sobre la pestaña que pone “resultados”.

Una vez que hemos añadido un parámetro, podemos editarlo haciendo *doble clic* sobre la celda, y escribiendo el nuevo texto.

15.3.3.11.2 Creación de un proclet

La forma de crear un proclet es la misma que la de crear una actividad o subproceso. Primero pulsamos sobre la herramienta de creación de proclet (misma apariencia que la de creación de actividad) y a continuación pulsamos sobre el lugar donde deseamos colocarlo. Una vez realizada la segunda pulsación nos aparecerá un diálogo para que escojamos el proclet:

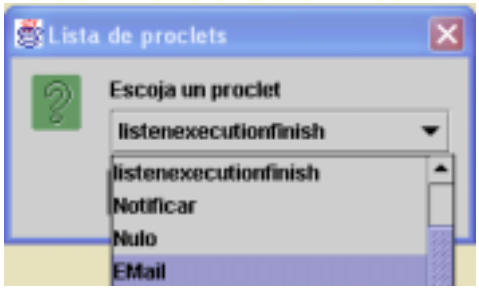

Ilustración 15.3-33: Lista de proclets

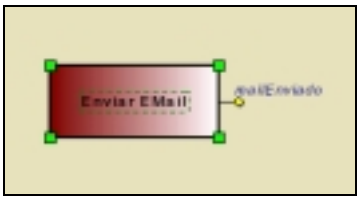

Ilustración 15.3-34: Añadir un proclet a un script

15.3.3.11.3 Edición de un proclet

Cada proclet puede tener su propio panel de edición. En general, todo proclet está formado por unos parámetros de entrada y unos resultados. Los parámetros de entrada pueden tener, aparte de nombre y valor, un origen del dato. Este origen puede ser "*inline*" o "*token*"; si el *inline* el valor del parámetro será literal, es decir, el introducido por el usuario; si el origen es *token* el valor se tomará de la pila del motor. Los parámetros token son también utilizados para referirse a los de entrada de un procedimiento.

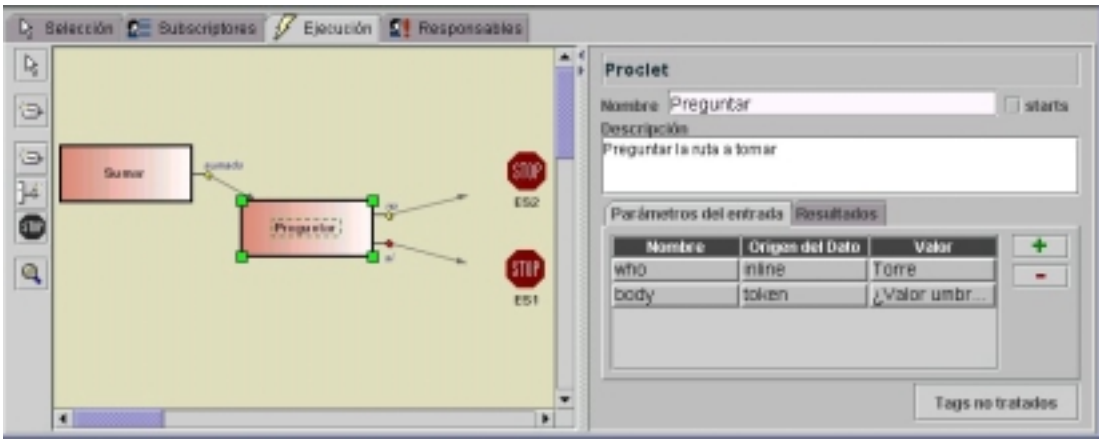

Ilustración 15.3-35: Edición de un proclet

15.3.3.11.4 Mapeo a salida de actividad

Las salidas del Script han de ser mapeadas a las salidas de la actividad que genera dicho Script. Para ello disponemos del siguiente panel:

Ilustración 15.3-36: Mapeo de salida de Script a salida de actividad

Si la actividad tiene como salidas: “error”,”correcto”,”incorrecto”; entonces la salida de Script debe ser mapeada a una de estas salidas (desplegando la lista de la figura anterior)

15.3.3.12 Script de responsabilidad de ejecución

Cada actividad tiene un script de responsabilidad propio. La forma de proceder es la misma que con el script de ejecución; la única diferencia radica en las salidas de script (en este caso son los responsables de la actividad); no deben ser mapeados a ninguna salida de actividad. Sencillamente debemos escribir su nombre.

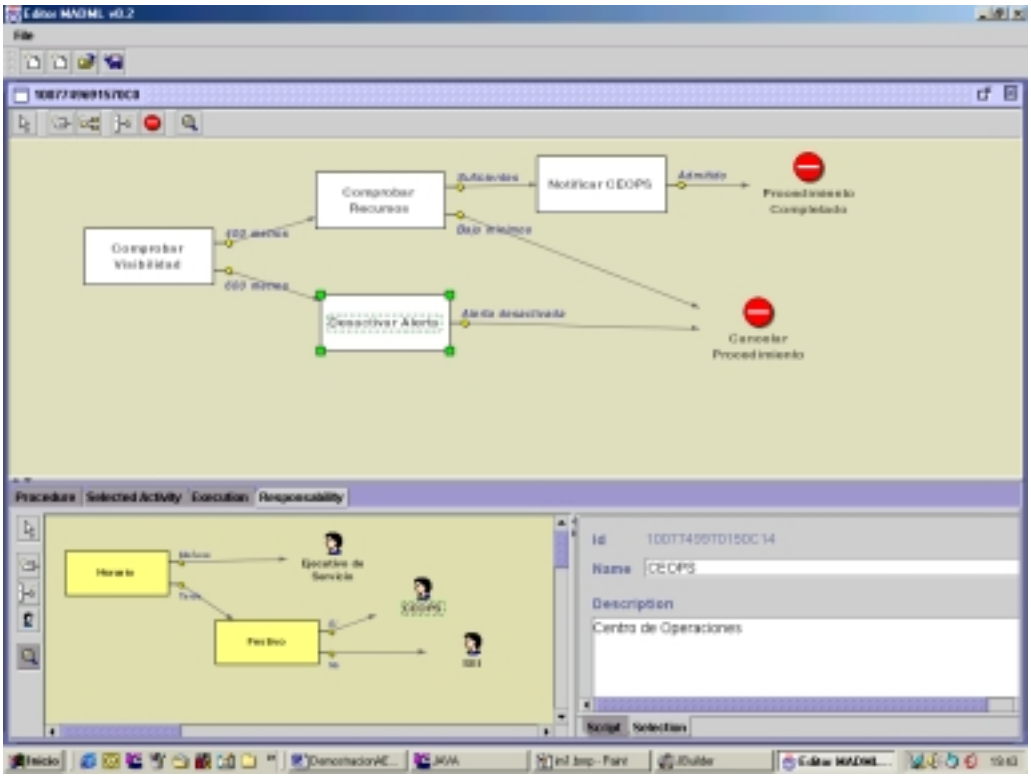

Ilustración 15.3-37: Script de responsabilidad

16 Bibliografía

[Abiteboul99] Serge Abiteboul, Dan Suciu, Peter Buneman. "Data on the Web: From Relations to Semistructured Data and Xml (Morgan Kaufmann Series in Data Management Systems)". Morgan Kaufmann Publishers; ISBN: 155860622X; 1st edition (October 1999).

[Acebal02] César F. Acebal, Raúl Izquierdo Castanedo, Juan M. Cueva Lovelle. "Good Design Principles in a Compiler University Course". ACM SIGPLAN Notices. Vol. 37, No. 4, April 2002.

[Aho90] A. V. Aho. "Compiladores: Principios, Técnicas y Herramientas".Addison-Wesley Iberoamericana. 1990.

[Álvarez96] Darío Álvarez Gutiérrez, Lourdes Tajes Martínez, Fernando Álvarez García, María Ángeles Díaz Fondón, Juan Manuel Cueva Lovelle y Raúl Izquierdo Castanedo. "Un Sistema Operativo para el Sistema Orientado a Objetos Integral Oviedo3". II Jornadas de Informática. Almuñécar, Granada. Julio de 1996.

[Álvarez97] Darío Álvarez Gutiérrez, Lourdes Tajes Martínez, Fernando Álvarez García, María Ángeles Díaz Fondón, Juan Manuel Cueva Lovelle y Raúl Izquierdo Castanedo. "An Object-Oriented Abstract Machine as the Substrate for an Object-Oriented Operating System". 11 th Conference in Object Oriented Programming (ECOOP). Workshop in Object Oriented Operating Systems. Jyväskylä, Finlandia. Junio de 1997.

[Allamaraju00] Subrahmanyam Allamaraju et. al. "Professional Java Server Programming J2EE Edition". Wrox Press 2000.

[Anderson98] Francis Anderson, Ralph Johnson "The Objectiva Architecture". OOpsla 98.

[Arsanjani99] Ali Arsanjani. "Rule Object: A Pattern Language for Adaptive and Scalable Business Rule Construction". IBM Enterprise Java Services National Practice, Raleigh, NC, USA. Maharishi University of Management, Fairfield, Iowa, USA.

[Basanta00] David Basanta, M. Cándida Luengo, Raul Izquierdo, Jose E. Labra, J. Manuel Cueva. "Improving the Quality of Compiler Construction with Object-Oriented Techniques". ACM SIGPLAN, Vol. 35, No. 12, pp. 41-51, December 2000.

[Beck00] Kent Beck, Martin Fowler. “Planning Extreme Programming”. Addison-Wesley Pub Co; ISBN: 0201710919; 1st edition (October 13, 2000)

[Beck96] Kent Beck. “Smalltalk Best Practice Patterns”. Prentice Hall PTR; ISBN: 013476904X; 1st edition (October 3, 1996)

[Beck99] Kent Beck. “Extreme Programming Explained: Embrace Change”. Addison-Wesley Pub Co; ISBN: 0201616416; 1st edition (October 15, 1999)

[Beners93] Tim Beners-Lee. “Hypertext Markup Language. HTML”. 1993.

[Beyer02] Hugh Beyer, Karen Holtzblatt. “Contextual Design : A Customer-Centered Approach to Systems Designs”. Morgan Kaufmann Publishers; ISBN: 1558604111; 1st edition (March 2002)

[Booch94] Grady Booch. “Análisis y diseño orientado a objetos con aplicaciones”.Editorial Addison-Wesley / Díaz de Santos. 1994.

[Borland98] Russell E. Borland “Adaptable and Adaptive Systems Workshop”. ACM Junio 1986

[Borning86] A. H. Borning. “Classes Versus Prototypes in Object-Oriented Languages”.In Proceedings of the ACM/IEEE Fall Joint Computer Conference 36-40. 1986

[Box99] Don Box. “Essential COM”. Addison-Wesley. Reading, Massachusetts (EE.UU.) ISBN 0201634465. 1999.

[Brose01] Gerald Brose, Andreas Vogel, Keith Duddy. “Java Programming with CORBA: Advanced Techniques for Building Distributed Applications”. John Wiley & Sons; ISBN: 0471376817; 3 edition (January 5, 2001)

[Brown98] Nat Brown y Charlie Kindel. “Distributed Component Object Model Protocol. DCOM/1.0”. Microsoft Corporation. Network Working Group. Enero, 1998.

[Buschmann96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. “Pattern-Oriented Software Architecture, Volume 1: A System of Patterns”. John Wiley & Son Ltd; ISBN: 0471958697; 1 edition (August 8, 1996)

[Campione99] Mary Campione, Kathy Walrath. “The Java Tutorial. Second Edition.Object Oriented Programming for Internet”. The Java Series. Sun Microsystems. 1999.

[Cardelli97] Luca Cardelli. “Type Systems”. Handbook of Computer Science and Engineering, Chapter 103. CRC Press. 1997.

[Coffe97] Peter Coffee (Editor), Michael Morrison, Randy Weems, Jack Leong “How to Program Java Beans”. Ziff Davis Pr. Abril 1997

[Cook94] Steve Cook, John Daniels. “Designing Object Systems. Object-Oriented Modelling with Syntropy”. Prentice Hall. Diciembre 1994

[Cooper95] Alan Cooper. “About Face: The Essentials of User Interface Design”. Hungry Minds, Inc; (August 11, 1995)

[Czarnecki00] Krzysztof Czarnecki, Ulrich W. Eisenecker. “Generative Programming. Methods, Tools and Applications”. Addison-Wesley 2000

[Demeter] Manual de usuario de DJ. URL http://www.ccs.neu.edu/research/demeter/DJ/UserManual.htm

[DSouza98] Desmond Francis D'Souza, Alan Cameron Wills. “Objects, Components, and Frameworks With Uml : The Catalysis Approach”. Addison-Wesley Pub Co; ISBN: 0201310120; (October 1998)

[Duffey01] Kevin Duffey, Richard Huss, Vikram Goyal, Ted Husted, Meeraj Kunnumpurath, Lance Lavandowska, Sathya Narayana Panduranga, Krishnaraj Perrumal, Joe Walnes. “Professional JSP Site Design”. Wrox Press Inc; ISBN: 1861005512; 1st edition (November 2001)

[Evins94] M. Evins. “Objects Without Classes”. Computer IEEE. Vol. 27, N. 3,104-109. 1994.

[Flemming98] Jennifer Fleming, Richard Koman. “Web Navigation: Designing the User Experience”. O`Reilly. Julio 1998

[Foote92] Brian Foote. “Objects, Reflection and Open Languages”. Workshop on Object Oriented Reflection and Metalevel Architectures. ECOOP 92. Utrech (Holanda)

[Foote98] Brian Foote, Joseph Yoder. “Metadata and Active Object-Models”. Proceedings of Plo98. Technical Report #wucs-98-25. Octubre 1998. URL: http://jerry.cs.uiuc.edu/~plop/plop98

[Fowler99] Martin Fowler, Kent Beck (Contributor), John Brant (Contributor), William Opdyke, don Roberts. “Refactoring: Improving the Design of Existing Code”. Addison-Wesley Pub Co; ISBN: 0201485672; 1st edition (August 1999)

[Gamma94] Eric Gamma, R. Helm, R. Johnson, J.O. Vlissides. “Design Patterns, Elements of Reusable Object-Oriented Software”. Addison-Wesley Editorial. 1994.

[Gamma95] Eric Gamma, Richard Helm, John Vlissides. “Design Patterns Applied”, tutorial Oopsla 1995.

[Geary97] David M. Geary, Alan L. McClellan. “Graphic Java. Mastering the AWT”. Prentice Hall 1997.

[Goldberg89] Goldberg A. y Robson D. “Smalltalk-80: The language”. Addison-Wesley.1989

[Gosling96] James Gosling, Bill Joy y Guy Seele. The Java Language Specification.Addison-Wesley. 1996.

[Grosso01] William Grosso. “Java RMI”. O'Reilly & Associates; ISBN: 1565924525; 1st edition (October 15, 2001)

[Hightower01] Richard Hightower, Nicholas Lesiecki. “Java Tools for Extreme Programming: Mastering Open Source Tools”. John Wiley & Sons; ISBN: 047120708X; 1st edition (December 15, 2001)

[Hunter] Jason Hunter. URL: www.jdom.org

[Hunter01] Jason Hunter, William Crawford. “Java Servlet Programming”. O'Reilly & Associates; ISBN: 0596000405; 2nd edition (January 15, 2001)

[IBM00e] “Multi-Dimensional Separation of Concerns”. International Business Machines Corporation, IBM Research. 2000

[IBM00f] “HyperJ̃: Multi-Dimensional Separation of Concerns for Java”. International Business Machines Corporation, IBM Research. 2000.

[ICSE2000] Second Workshop on Multi-Dimensional Separation of Concerns in Software Engineering. ICSE’2000. Limerick (Irlanda). Junio de 2000.

[Izquierdo96] Izquierdo Castanedo, Raúl."Maquina Abstracta Orientada a Objetos Carbayonia". Ponencia en Congreso. II Jornadas sobre Tecnologías Orientadas a Objetos. Oviedo. Marzo, 1996.

[Jacobson99] Ivar Jacobson, Grady Booch, James Rumbaugh. “The Unified Software Development Process”. Addison-Wesley Pub Co; ISBN: 0201571692; 1st edition (January 1999)

[Jazayeri98] M. Jazayeri, R. Loos, D. Musser and A. Stepanov. “Generic Programming”. Report of the Dagstuhl Seminar 9817. Abril 1998. http://www-ca.informatik.uni-tuebingen.de/dagstuhl/gpdag_2.html

[Johnson91] Ralph E. Johnson, Brian Foote. “Designing Reuseable Classes”. Journal of Object Oriented Programming. Agosto 1991

[Johnson92] Ralph Johnson. "Documenting Frameworks Using Patterns" (from OOPSLA'92). URL www.jhotdraw.org

[Johnson98] Ralph E. Johnson and Jeff Oakes. “The User Defined Product Framework” 1998. URL http://st.cs.uiuc.edu/pub/papers/frameworks/udp

[Johnson98b] Ralph Johnson, Bobby Wolf. “Type Object”. Pattern Languages of Program Design 3. Addison Wesley 1998

[Kalev98] Danny Kalev. “The ANSI/ISO C++ Professional Programmers Handbook”. Que Editorial. 1998

[Kay01] Michael H. Kay. “XSLT Programmer's Reference 2nd Edition”. Wrox Press Inc; ISBN: 1861005067; 1st edition (April 2001)

[Kiczales01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G. Griswold. “Getting Started with AspectJ”. CACM 2001. aspectj.org. 2001.

[Kiczales97] G.Kiczales. “Aspect-oriented programming”. In ECOOP ’97:European Conference on Object-Oriented Programming, 1997.Invited presentation

[Kiczales97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. “Aspect Oriented Programming”. Proceedings of ECOOP’97 Conference. Finlandia. Junio de 1997.

[Kovitz99] Benjamin L. Kovitz. “Práctical Software Requirements”. Manning Publications 1999

[Kramer96] Douglas Kramer. “The Java Platform. A White Paper”. Sun Micro-Systems JavaSoft. Mayo 1996.

[Lauesen98] Soren Lauesen. “Real-life object-oriented systems”. IEEE Software, 1998, March/April, p. 76-83.

[Lieberherr96] Karl J. Lieberherr. “Adaptive Object Oriented Software: The Demeter Method”. PWS Publishing Company. 1996

[Lieberman86] H. Lieberman. “Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems”. In OOPSLA’86 Conference Proceedings. Published as SIGPLAN Notices, 21, 11, 214-223. 1986

[Linthicum99] David S. Linthicum. “Enterprise Application Integration”. Addison-Wesley Pub Co; ISBN: 0201615835; 1st edition (November 12, 1999)

[Loomis94] Mary E. S. Loomis. “Object Databases : The Essentials”. Addison-Wesley Pub Co; ISBN: 020156341X; 1st edition (October 1994)

[Maguire93] Steve Maguire. “Writing Solid Code : Microsoft's Techniques for Developing Bug-Free C Programs”. Microsoft Press; ISBN: 1556155514; (May 1993).

[Manolescu00] D. Manolescu. “Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Deveopment” PhD thesis, Computer Science Technical Report UIUCDCS-R-2000-2186. University of Illinois. Octubre 2000

[Martine98] Martine Devos, Michel Tilman. “A repository-based framework for evolutionary software development”. ECOOP 98

[Metrica21] MÉTRICA. Versión 2.1. URL: http://www.map.es/csi/pg5m40.htm

[Meyer97]. Bertrand Meyer. “Object-Oriented Software Construction”. 2ª Edición. Prentice-Hall. 1997

[Microsoft00] “C# Language Specification v.0.22”. Microsoft Corporation. Febrero de 2000.

[Microsoft95] “The Component Object Model Specification”. Version 0.9. Microsoft Corporation. Octubre de 1995.

[Myers79] Glenford J. Myers. “The Art of Software Testing”. John Wiley & Sons; ISBN: 0471043281; 1 edition (February 20, 1979)

[Nielsen94] Jakob Nielsen. “Usability Engineering”. Morgan Kaufmann Publishers (October 1994)

[Nielsen95] Jakob Nielsen. “Multimedia and Hypertext : The Internet and Beyond”. Academic Press; ISBN: 0125184085; Reprint edition (February 1995)

[Oopsla99] First Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems. OOPSLA’99. Denver (EE.UU.). Noviembre de 1999.

[Opdyke93] William F. Opdyke, Ralph E. Johnson. “Creating abstract superclasses by refactoring”. Proceedings of the 21st Annual Computer Science Conference (ACM CSC 93). Febrero de 1993

[Ortín01] Ortín, Francisco. “Sistema Computacional de Programación Flexible Diseñado Sobre una Máquina Abstracta Reflectiva no Restrictiva”. Tesis Doctoral. URL “http://di002.edv.uniovi.es/~ortin/”

[Ortín97] Francisco Ortín Soler, Darío Álvarez Gutiщrrez, Raúl Izquierdo Castanedo, Ana Belén Martínez Prieto, Juan Manuel Cueva Lovelle. “The Oviedo3 Persistence System”. III Jornadas Internacionales de Orientación a Objetos. Sevilla (Spain). October 1997.

[Ossher99] H. Ossher, P. Tarr. “Multi-Dimensional Separation of Concerns using Hyperspaces”. IBM Research Report 21452. Abril de 1999.

[Parnas72] D..L.Parnas.”On the criteria to be used in decomposing systems into modules”.Communications of the ACM , 15(12):1053 –1058,December 1972.

[Perkins00] Alan Perkins. “Business rules = meta-data”. Proceedings of 34th International conference on Technology of Object Oriented Languages and Systems. 2000

[Pressman01] Roger S. Pressman. “Software Engineering : A Practitioner's Approach”. McGraw Hill College Div; ISBN: 0072496681; 5th Pkg edition (June 2001)

[Reenskaug95] Reenskaug, Trygve “Working with Objects: The OOram Software Engineering Method” Manning 1995

[Riehle00] Dirk Riehle, Michel Tilman, Ralph Johnson. “Dynamic Object Model”. PLoP 2000

[Roberts98] Don Roberts and Ralph E. Johnson. “Evolve Frameworks into domain-Specific Languages Pattern Languages of Program Design 3” edited by Frank Buschmann, Dirk Riehle, and Robert Martin. Addison Wesley Longman, 1998

[Roman01] Ed Roman, Scott W. Ambler, Tyler Jewell, Floyd Marinescu. “Mastering Enterprise JavaBeans (2nd Edition)”. John Wiley & Sons; ISBN: 0471417114; 2nd Bk&acc edition (December 14, 2001)

[Rouvellou00] I. Rouvellou, L. Degenaro, K. Rasmus, D. Ehnebuske, B. McKee “Extending business objects with business rules”. Proceedings on Tecnology of Object-Oriented Languages, 2000.

[Rumbaugh91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. “Object Oriented Modeling and Design”. Prentice Hall, 1991

[Rumbaugh98] James Rumbaugh, Ivar Jacobson, Grady Booch. “The Unified Modeling Language Reference Manual”. Addison-Wesley. Diciembre de 1998.

[Stepanov95] A. Stepanov, Meng Lee. “The Standard Template Library”. Hewlet-Packard, Palo Alto, CA, 1995.

[Stroustrup98] Bjarne Stroustrup. “The C++ Programming Language”. Third Edition. Addison-Wesley. October 1998

[Sun96] “JavaBeans™ 1.0 API Specification”. Sun Microsystems Computer Corporation. Diciembre de 1996

[Sun97d] “Java Core Reflection. API and Specification”. JavaSoft. Enero de 1997.

[Sundsted98] Todd Sundsted. "MVC Meets Swing," (JavaWorld, April 1998): http://www.javaworld.com/jw-04-1998/jw-04-howto.html

[Tilman99] M. Tilman, M. Devos. “A reflective and Repository Based Framework”. Implementing Application Frameworks”. Wiley 1999.

[Topley99] Kim Topley. “Core Swing: Advanced Programming”. Prentice Hall PTR; ISBN: 0130832928; 1st edition (December 22, 1999).

[Turner98] C.R.Turner, A.Fuggetta,L.Lavazza and A.L.Wolf. “Feature Engineering”.In Proceedings of the 9th International Workshop on Software Specification and Design ,162 –164,April,1998.

[W3C98] “Extensible Markup Language (XML) 1.0”. World Wide Web Consortium. Febrero de 1998.

[W3C98b] “Level 1 Document Object Model Specification”. Version 1.0. W3C Working Draft, WD-DOM-19980720. Julio de 1998.

[Wirfs90] Rebecca Wirks-Brock, Brian Wilkenson, Lauren Wiener. “Designin Object-Oriented Software”. Prentice Hall, Englewood Cliffs. 1990

[Witthawaskul98] Weerasak Witthawaskul. “Position Statement: Active Object-Model” Oopsla 98

[Yoder01] Joseph W. Yoder. “Adaptive Object-Models”. ECOOP 2001. Budapest, Hungary. Enero 2001.

[Yoder01b]. Josepgh Yoder, Federico Balaguer, Ralph Johnson. "The Architectural Style of Adaptive Object-Models". Workshop on Adaptive Object-Models and Metamodeling Techniques. Budapest (Hungría). Junio de 2001

[Yoder01c] Josepgh Yoder, Federico Balaguer, Ralph Johnson. "Architecture and Design of Adaptive Object-Models". ACM Sigplan Notices V 36 Diciembre 2001

[Yoder98] Joseph W. Yoder, "Metadata Pattern Minning Workshop". OOpsla 98. URL http://www.joeyoder.com/Research/metadata/uiucpresentation/