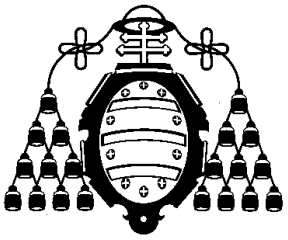

**UNIVERSIDAD DE OVIEDO**

**Trabajo de Investigación**

# DISEÑO DE MÁQUINAS ABSTRACTAS BASADAS EN REFLECTIVIDAD COMPUTACIONAL

**Lenguajes y Sistemas Informáticos**

**Departamento de Informática**

**Autor:** Francisco Ortín Soler
**Director:** Juan Manuel Cueva Lovelle
Junio, 1999

# Tabla de Contenidos

# Tabla de Ilustraciones

# Capítulo 1:

# Introducción

En este trabajo de investigación se realizará un estudio sobre las posibilidades que tiene el diseño de máquinas abstractas basadas en reflectividad computacional. Una vez identificados una serie de conceptos y puntos de interés se definirán un conjunto de líneas de investigación centradas en este punto.

El concepto de máquina abstracta no es un concepto nuevo. La especificación de una plataforma mediante la semántica de sus instrucciones sin indicar la implementación de ésta ha tenido distintas utilidades. La explotación de sus características ha llevado al éxito en determinados entornos de computación. Un ejemplo actual es la plataforma Java [Kramer96]. Su utilización es satisfactoria si se busca portabilidad, movilidad y seguridad del software.

La capacidad de reflexión de un sistema (reflectividad) posibilita el análisis y modificación de él mismo. Existen diversas líneas de investigación entorno a la reflectividad así como sistemas comerciales que hacen uso, en mayor o menor grado, de ella. La reflectividad computacional es una clasificación del concepto de reflectividad por la cual se puede modificar la semántica o interpretación (el funcionamiento) de un sistema. Es en este punto donde existen más líneas de investigación abiertas y donde todavía no se ha empleado de forma clara en sistemas comerciales.

La flexibilidad en la especificación de una plataforma independiente hace de la reflectividad un criterio interesante para el diseño de las máquinas abstractas. Así mismo, el procesamiento multiplataforma de una máquina abstracta es un sistema ideal para el desarrollo de prototipos de investigación en lo referente a las posibilidades de la reflectividad computacional.

La especificación de una máquina abstracta dotada de reflectividad puede ser un entorno de programación que simplifique y mejore las arquitecturas de objetos distribuidos existentes. Sobre este sistema de computación se podrán definir un conjunto de capas constituyentes de un sistema integral orientado a objetos. Sobre este último punto, existen multitud de campos de investigación, uno de ellos el que está siendo desarrollado en la Universidad de Oviedo [Cueva96, Alvarez98].

## 1.1 Organización del Documento.

A continuación desglosaremos brevemente el contenido del documento, organizado por capítulos.

En el capítulo 2 se define el concepto de máquina abstracta. Se muestra una evolución en su utilización, centrada en el objetivo buscado en su uso. Se describen las más conocidas y sus similitudes. Las arquitecturas de objetos distribuidos son estudiadas en el capítulo 3. Se comentan sus objetivos y alternativas y se estudian en su unión con las máquinas abstractas. En el capítulo 4 se describen las posibilidades de un sistema integral orientado a objetos y la estructura del proyecto Oviedo3. Los lenguajes orientados a objetos basados en prototipos y sus posibles ventajas son descritos en el capítulo 5. El capítulo 6 define y clasifica de diversas maneras el concepto de reflectividad, estudia los sistemas existentes y la investigación llevada a cabo e identifica trabajos relacionados con ésta. Finalmente se señalan posibles líneas de investigación en el capítulo 7. Varias de ellas están siendo actualmente investigadas por el autor de este trabajo de investigación.

Otros capítulos del documento son dos apéndices desarrollados como ejemplos de utilización de reflectividad. En los apéndices A y B se han implementado prototipos en Java y en Lisp que muestran las posibilidades de la reflectividad estructural y computacional respectivamente.

Finalmente se muestra el conjunto de referencias bibliográficas utilizadas.

# Capítulo 2:

# Máquinas Abstractas

Un programa, es un conjunto ordenado de instrucciones que se dan al ordenador indicándosele las operaciones o tareas que se desea que realice [Cueva94]. Estos programas son interpretados por un procesador.

Un procesador interpreta las instrucciones propias de un programa modificando o trabajando sobre los datos propios de éste. La implantación de un procesador puede ser *hardware* (física) o *software* (lógica).

## 2.1 Procesadores Hardware.

Un procesador hardware es un intérprete de programas implementado físicamente. Los procesadores más extendidos son los procesadores digitales síncronos: están formados por una unidad de control, una unidad de memoria y una unidad aritmética y lógica interconectadas entre sí [Mandado73]. En función de la implantación de la unidad de control, se puede establecer la siguiente clasificación de procesadores:

Procesadores digitales secuenciales síncronos cableados.

Procesadores digitales secuenciales síncronos microprogramables.

Los microprogramables ofrecen una mayor flexibilidad puesto que la unidad de control puede ser programada. De esta forma es posible modificar su diagrama de secuencia de operaciones sin alterar el cableado de su unidad de control.

## 2.2 Procesadores Software.

Un emulador o procesador software (denominado también intérprete) es un programa que interpreta a su vez programas de otro procesador [Cueva92]. Es decir, es aquel programa capaz de interpretar o emular el funcionamiento de un determinado procesador.

La programación de la unidad de control de un procesador mediante un microcódigo es menos flexible que la modificación de un emulador o procesador software. Esto hace que los emuladores software se utilicen, entre otras cosas, como herramientas de simulación encaminadas a la implementación física del procesador que emulan.

La principal desventaja de este tipo de procesadores frente a los procesadores *hardware* o físicos es la velocidad de ejecución. Puesto que los procesadores software establecen un nivel más de computación (son ejecutados o interpretados por otro procesador), esto hace que inevitablemente requieran mayor número de computaciones para interpretar un programa que su versión hardware.

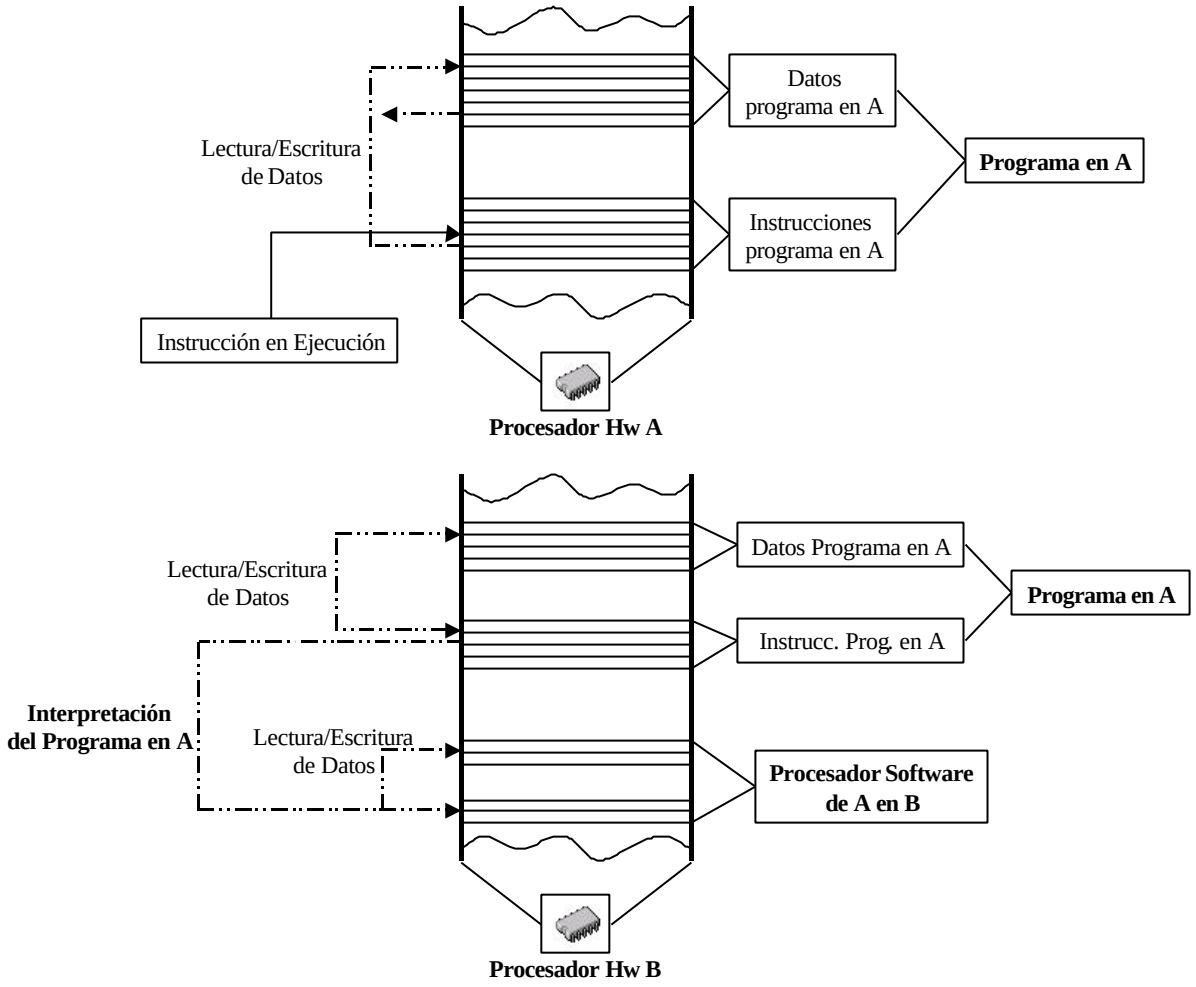


**Figura 1: Ejecución de un programa en un procesador *hardware* y en un procesador *software*.**

En la Figura 1 se muestran las diferencias entre un procesador físico y un procesador *software*. El la parte superior de la figura se muestra cómo el procesador físico **A** va interpretando las distintas instrucciones máquina. La interpretación de las instrucciones implica la lectura y/o escritura de los datos.

En el caso de interpretar a nivel *software* el programa, el procesador es a su vez un programa en otro procesador físico (procesador **B** en la Figura 1). Vemos como existe una nueva capa de computación frente al ejemplo anterior. Esto hace que se requieran más computaciones[1] o cálculos que en el primer caso.

## 2.3 Especificación de Máquinas Abstractas.

El nombre de máquina abstracta indica que la máquina o procesador posee unas determinadas especificaciones que definen su funcionamiento [Alvarez98]. La especificación de una máquina abstracta viene dada por:

La definición formal del formato de programa que es capaz de procesar.

La descripción semántica o del funcionamiento del formato anterior.

[1] Mayor número de computaciones no implica mayor tiempo de ejecución. Si en el ejemplo de la Figura 1 identificamos el procesador A como un Z80 y el procesador B como un Pentium, seguramente conseguiremos mayor velocidad en tiempo de ejecución aun realizando un mayor número de computaciones.

En la especificación de una máquina abstracta no es necesario, por lo tanto, identificar la estructura interna de emulador software ni la implantación física del procesador hardware. De esta forma, una máquina abstracta puede estar implementada física o lógicamente y en cada caso con distintas estrategias. En todas las alternativas, sin embargo, se deberá obtener el resultado especificado en la descripción semántica de la máquina.

## 2.4 Plataformas Independientes.

Aunque todos los procesadores *hardware* son realmente implementaciones físicas de máquinas abstractas, se utiliza el concepto de máquina abstracta o máquina virtual para designar las especificaciones de plataformas cuyo único objetivo final no es la implementación en silicio.

La característica más explotada de las especificaciones de máquinas abstractas, es la independencia de la implementación del procesador que interpreta sus programas. Esto hace que los programas de una máquina abstracta no dependan de una plataforma propia. La especificación de una máquina abstracta con sólo los puntos expuestos en 0 identifican pues una plataforma independiente.

Al igual que un compilador de un lenguaje de alto nivel genera código para una máquina específica, la compilación a una máquina abstracta hace que ese programa generado sea independiente de la plataforma que lo ejecute. Dicho programa podrá ser ejecutado por cualquier procesador –ya sea hardware o software –que cumpla la especificación de la máquina abstracta. Además cada procesador o emulador de la máquina podrá estar implementado acorde a las necesidades y características del sistema real.

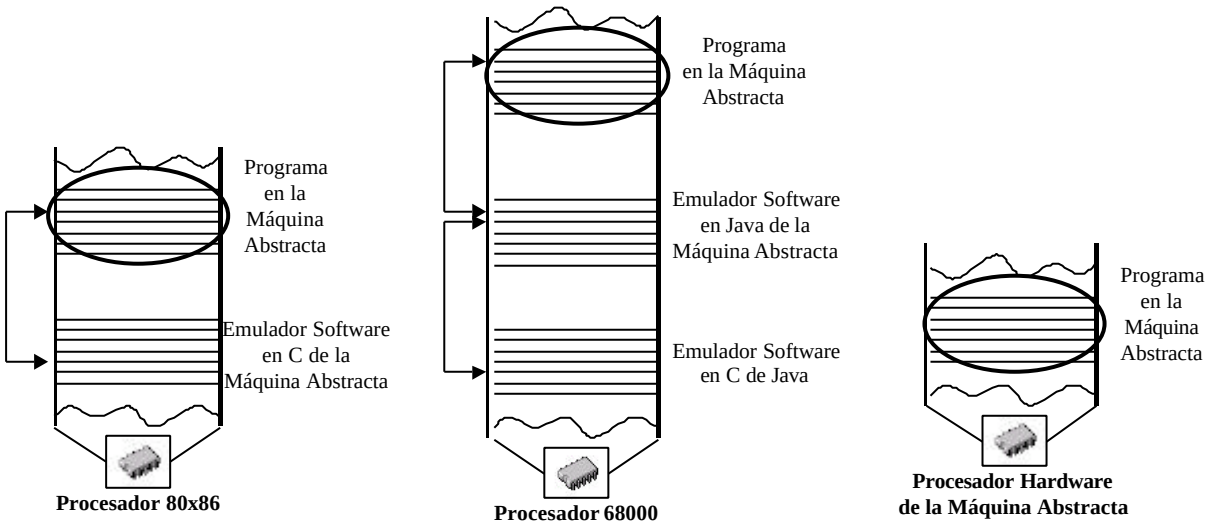


**Figura 2: Ejemplo de tres contextos en los que se aprecia la independencia de una plataforma.**

En la Figura 2 se aprecian tres casos en los que se procesa un programa de una máquina abstracta. En el primer caso, se ejecuta la aplicación mediante un intérprete software. En el segundo gráfico se muestra cómo se puede emular el funcionamiento de la máquina abstracto sobre un procesador *software*. Este último se ejecuta en un micro distinto al del primer ejemplo. Finalmente vemos cómo sería posible implementar físicamente la especificación de la máquina abstracta.

Lo realmente interesante del ejemplo real mostrado en la Figura 2 es el hecho de poder ejecutar el mismo programa en tres plataformas diferentes: en un i80x86, en un Motorola 68000 y en una implantación física del intérprete de la máquina abstracta.

La forma en la que los tres intérpretes ejecutan la aplicación tan sólo ha de cumplir la especificación semántica de funcionamiento de la máquina abstracta. Los tres han de conseguir el mismo resultado de la ejecución pero cada uno implementará la simulación de la manera que más le convenga[2].

## 2.5 Evolución Histórica de las Máquinas Abstractas.

Como hemos comentado en el punto 2.4, las implementaciones *hardware* de intérpretes de lenguajes son máquinas abstractas aunque este término se suele aplicar normalmente a aquellas especificaciones que no tienen una implantación física como principal objetivo. En el resto de este documento nos referiremos, con el término de máquina abstracta, tan sólo al segundo tipo identificado.

En este capítulo se identifica la evolución de un conjunto de máquinas abstractas existentes, comentando el objetivo principal por el que fueron diseñadas.

### 2.5.1 Máquinas Abstractas en la Simplificación de Compiladores

En la implementación de compiladores se empezó a utilizar el concepto de máquina abstracta para simplificar el diseño de compiladores [Cueva92].

El proceso de compilación toma un lenguaje de alto nivel y genera un código intermedio. Este código intermedio es propio de una máquina abstracta. Se elige la máquina abstracta lo más general posible, de forma que se pueda traducir de esta máquina a cualquier máquina real existente. Para generar el código binario de una máquina real, tan sólo hay que traducir el código de la máquina abstracta a una máquina real, independientemente del lenguaje de alto nivel que haya sido utilizado.

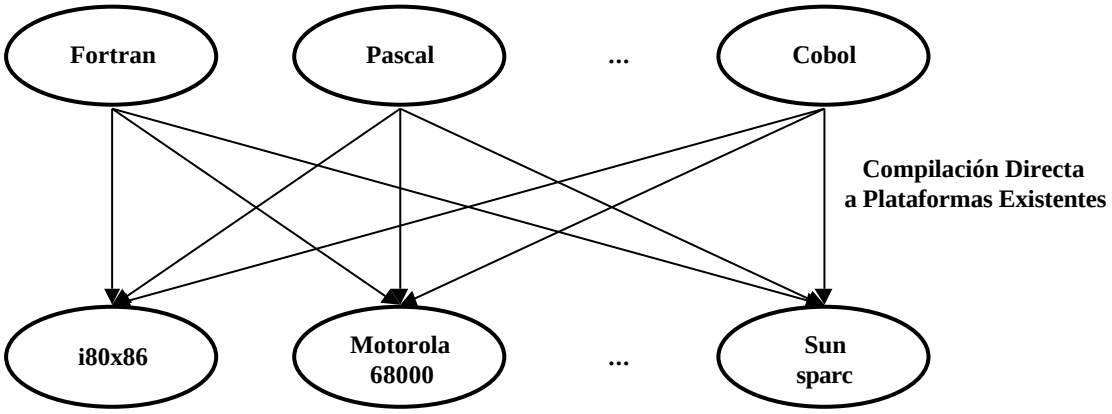


**Figura 3: Compilación directa de lenguajes a máquinas reales.**

[2] Lo más común es ver cómo el emulador ahorma el funcionamiento de la máquina abstracta al conjunto de sus recursos (a su plataforma concreta). De esta forma se consigue que el proceso de interpretación sea lo más simple posible.

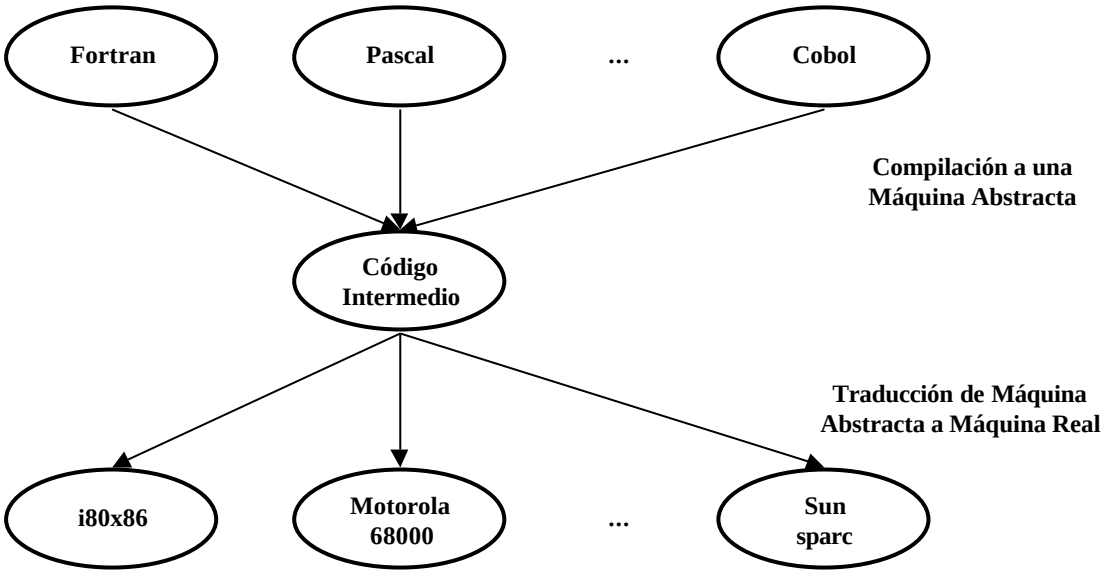


**Figura 4: Compilación de lenguajes a máquinas reales pasando por la generación de código intermedio.**

En la Figura 3 y en la Figura 4 se observa cómo el número de traducciones y compilaciones se reduce cuando tenemos varios lenguajes fuente y varias máquinas destino existentes. Esta práctica ha sido adoptada por varias compañías que desarrollan diversos tipos de compiladores [Borland91]. A nivel interno, definen una máquina abstracta que englobe las características comunes a los procesadores existentes, generando código intermedio para ésta y traduciéndolo posteriormente.

El proyecto UNCOL (*Universal Computer Oriented Language*) proponía un lenguaje intermedio universal para el diseño de compiladores [Steel60]. El objetivo de este proyecto era especificar una máquina abstracta universal para que los compiladores generasen código intermedio a una plataforma abierta. No fue pensada sin embargo, para definir una plataforma independiente.

ANDF (*Architecture Neutral Distribution Format)* [Macrakis93] tuvo como objetivo un híbrido entre la simplificación de compiladores y la portabilidad del código (punto siguiente). Un compilador podría generar código para la especificación de la máquina ANDF siendo este código portable a distintas plataformas.

Este código ANDF no era interpretado por un procesador software sino que era traducido (o instalado) a código binario de una plataforma específica. De esta forma se conseguía lo propuesto con UNCOL, la distribución de un código de una plataforma independiente.

La primera implementación de la máquina-p (próximo punto) también se utilizó como código intermedio de un compilador del lenguaje Pascal [Nori76].

### 2.5.2 Portabilidad de Código Generado.

La principal razón por la que hoy en día se ha difundido el concepto y la utilización de las máquinas abstractas es la portabilidad del código propio éstas y por lo tanto la búsqueda de plataformas independientes (como veremos en 2.5.3).

#### 2.5.2.1 Máquina-p.

El código-p era el lenguaje intermedio propio de la máquina abstracta maquina-p [Nori76], utilizada inicialmente en el desarrollo de un compilador del lenguaje Pascal [Jensen91].

La Universidad de California en San Diego (UCSD) desarrolló un procesador que ejecutaba código binario de la máquina-p (código-p). Adoptó pues la especificación de la máquina abstracta para desarrollar así un proyecto de Pascal portable. Se llegó a disponer de soporte para multitarea y se desarrolló el p-System: un sistema operativo portable, codificado en Pascal y traducido a código-p [Campbell83].

Al igual que los lenguajes C y Algol, el Pascal es un lenguaje orientado a bloques (orientado a marcos de pila[3] desde el punto de vista de implementación) [Jensen91] y esto hizo que el criterio fundamental en la especificación de la máquina-p fuese orientarla a una estructura de pila.

El p-System implementado, tenia la siguiente distribución de memoria, desde las direcciones superiores a las inferiores:

El código (p-código) propio del sistema operativo (p-System).

La pila del sistema (creciendo en sentido descendente).

La memoria *heap* (creciendo en sentido ascendente).

El conjunto de las pilas propias de hilos según se iban demandando en tiempo de ejecución.

Los segmentos globales de datos (de constantes y variables).

El intérprete o simulador de la máquina abstracta.

La máquina abstracta llegó a tener un procesador *hardware.* Western Digital implementó en 1980 la máquina-p en el WD9000 Pascal Microengine. Éste estaba basado en el microprocesador programable WD MCP-1600.

#### 2.5.2.2 Forth.

Otro ejemplo de especificación de una máquina abstracta para conseguir portabilidad de código es la máquina virtual de Forth [Brodie87]. Este lenguaje fue desarrollado en la década de los 70 por Charles Moore para el control de telescopios. Es un lenguaje sencillo, rápido y ampliable que es interpretado en una máquina virtual, consiguiendo ser portable y útil para empotrarlo en sistemas.

El simulador de la máquina virtual de Forth posee dos pilas. La primera es la pila de datos: los parámetros de una operación son tomados del tope de la pila y el resultado es posteriormente apilado. La segunda pila es la pila de valores de retorno: se apilan los valores del contador de programa antes de una invocación a una subrutina, para poder retornar al punto de ejecución original una vez finalizada ésta.

### 2.5.3 Máquinas Abstractas Orientadas a Objetos. Plataformas Independientes.

Con la aparición de los lenguajes de alto nivel orientados a objetos, no tardaron en aparecer máquinas abstractas que adoptaron este paradigma en su código máquina. Dentro de este grupo de máquinas abstractas podemos destacar principalmente dos: la máquina de Smalltalk [Goldberg89] y la de Java [Gosling96].

[3] *Stack Frame Oriented:* Por cada bloque se apila un marco o contexto propio de la ejecución de ese bloque.

#### 2.5.3.1 Smalltalk-80.

El sistema Smalltalk-80 tiene sus raíces en el centro de investigación de Xerox, Palo Alto. Empezó en la década de los setenta pasando por la implementación de tres sistemas principales: Smalltalk 72, 76 y 80. El número corresponde con el año en el que fueron diseñados.

Los esfuerzos del grupo investigador buscaban la obtención de un sistema que fuese manejable por personas que no fuesen informáticos. Para llegar a esto se apostó por un sistema basado en gráficos, interfaces interactivos y visuales y una mejora en la flexibilidad a la hora de programar. En la flexibilidad de la programación podemos añadir a la programación orientada a objetos, el cómodo acceso a clases y objetos que otorga el sistema.

El sistema Smalltalk-80 está dividido básicamente en dos grandes componentes [Krasner83]:

La imagen virtual: colección de objetos, instancias de clases, que proporcionan estructuras básicas de datos y control, primitivas de texto y gráficos, compiladores, descompiladores y manejo básico de la interfaz de usuario.

La máquina virtual: intérprete de la imagen virtual y de cualquier aplicación del usuario. Se divide en:

El gestor de memoria: Se encarga de gestionar los distintos objetos en memoria, sus relaciones y su ciclo de vida. Para ello implementa un recolector de basura de objetos.

El intérprete de instrucciones: Analiza y ejecuta las instrucciones en tiempo de ejecución. Las operaciones que se utilizan son un conjunto de primitivas que operan directamente sobre el sistema.

Aunque exista una especificación formal de la máquina abstracta [Goldberg83], con la existencia de ésta no se buscó directamente una plataforma independiente sino aprovechar las características propias de un lenguaje interpretado[4].

Todas las aplicaciones constituyentes del sistema de Smalltalk-80 están escritas en el propio lenguaje y, al ser éste interpretado, se puede acceder dinámicamente a todos los objetos existentes en tiempo de ejecución. Un ejemplo puede ser el *Browser* [Mevel87]: es una aplicación que recorre todas las clases (los objetos derivados de Class) del sistema (del diccionario Smalltalk) y nos visualiza la información de ésta:

Categoría a la que pertenece la clase.

Un texto que la describe.

Métodos.

Atributos.

Código de creación de cada uno de éstos.

Esta información es modificable en todo momento. Además tenemos la documentación real, puesto que se genera dinámicamente, de todas las clases, objetos, métodos y atributos existentes en el sistema.

El aplicar estos criterios con todos los programas del sistema, hace que la programación depuración y análisis de éste sea muy sencilla. La figura de una máquina abstracta que interpreta el código ayuda a conseguir estas características.

[4] Smalltalk no es interpretado directamente. Pasa primero por una fase de compilación a un código binario en la que se detecta una serie de errores. Este código binario será posteriormente interpretado por el simulador o procesador software de la máquina abstracta.

Smalltalk-80 es pues un sistema de computación que consigue, mediante una máquina virtual, una integración entre todas sus aplicaciones y una independencia de plataforma.

#### 2.5.3.2 SELF.

Posteriormente a Smalltalk-80 se creó el proyecto SELF [Ungar87]. Se especificó una máquina abstracta reduciendo la representación en memoria de los objetos que tenía el Smalltalk. El criterio principal de diseño fue representar el sistema sólo con objetos –eliminado el concepto de clase. Abordaremos este tipo de lenguajes en el capítulo 5.

La simplicidad y la pureza en los conceptos de esta máquina, hicieron que se realizasen múltiples optimizaciones en la compilación llegando a resultados interesantes [Chambers91]. En la búsqueda de esta optimización se descubrieron nuevos métodos de compilación como la compilación continua unida a la optimización adaptable [Hölze94]. Estas técnicas de compilación y optimización avanzadas están siendo utilizadas en la mejora de máquinas abstractas como la de Java (The Java "HotSpot" Virtual Machine) [Sun98].

#### 2.5.3.3 Java.

Oak fue un lenguaje orientado a objetos creado por Sun Microsystems que se renombró a Java como hoy en día se conoce [Gosling96]. Adoptando una sintaxis similar a la del lenguaje C++ [Stroustrup98], trata de eliminar los conceptos no pertenecientes a la programación orientada a objetos y los inconvenientes más comunes de la programación C++.

Un programa en Java como lenguaje se compila para ser ejecutado sobre una plataforma independiente [Kramer96]. Esta plataforma de ejecución independiente está formada básicamente por:

La máquina virtual de Java.

La interfaz de programación de aplicaciones en Java o API.

Esta dualidad que consigue la independencia de una plataforma es igual que la que hemos identificado en Smalltalk-80: imagen virtual y máquina virtual. El API es un conjunto de clases que están compiladas en el formato de código binario de la máquina abstracta [Sun95]. Estas clases son utilizadas por el programador para realizar aplicaciones de una forma más sencillas.

La máquina virtual de Java es el procesador del código binario de esta plataforma. El soporte a la orientación a objetos está definido en su código binario aunque no se define su arquitectura. Por lo tanto la implementación del intérprete y la representación de los objetos en memoria queda a disposición del diseñador del simulador.

La creación de la plataforma de Java se debe a la necesidad existente de abrir el espacio computacional de una aplicación. Las redes de computadores interconectan distintos tipos de ordenadores y dispositivos entre sí. Se han creado múltiples aplicaciones, protocolos, *middlewares* y arquitecturas cliente servidor para resolver el problema de lo que se conoce como programación distribuida [Orfali96].

Una red de ordenadores tiene interconectados distintos tipos de dispositivos y ordenadores, con distintas arquitecturas hardware y sistemas operativos. Java define una máquina abstracta para conseguir implementar aplicaciones distribuidas que se ejecuten en todas las plataformas existentes en la red (plataforma independiente). De esta forma, cualquier elemento conectado a la red que posea un procesador de la máquina virtual y el API correspondiente será capaz de procesar una aplicación –o una parte de ésta –implementada en Java.

Para conseguir una este tipo de programación distribuida, la especificación de la máquina abstracta de Java se ha realizado siguiendo fundamentalmente tres criterios [Venners98]:

Búsqueda de una plataforma independiente.

Especificación de un mecanismo de seguridad robusto.

Movilidad del código a lo largo de la red.

La característica de definir una plataforma independiente está implícita en la especificación de una máquina abstracta. Sin embargo, para conseguir la movilidad del código a través de las redes de computadores, es necesario tener en cuenta otra cuestión: la especificación de la máquina ha de permitir obtener código de una aplicación a partir de una máquina remota.

La distribución del código de una aplicación Java distribuida queda directamente soportada por la funcionalidad de los "*class loaders*" [Gosling96]: Se permite definir la forma en la que se obtiene el software (en este caso las clases), pudiendo éste estar localizado en máquinas remotas. De esta forma la distribución de software es directa puesto que se puede centralizar todo el código en una sola máquina y ser actualizado desde los propios clientes al principio de cada ejecución.

Finalmente cabe mencionar la importancia que se le ha dado a la seguridad. Las redes representan un instrumento para aquellos programadores que deseen destruir información, escamotear recursos o simplemente molestar. Un ejemplo de este tipo de aplicaciones puede ser un virus distribuido que se ejecute en nuestra máquina una vez demandado por la red.

El sistema de seguridad que proporciona la máquina virtual, viene proporcionado por el "*security manager*": una aplicación cliente puede definir un *security manager* de forma que se limite, por la propia máquina virtual, el acceso a todos los recursos que se estime oportuno. Se puede definir así los límites del software obtenido.

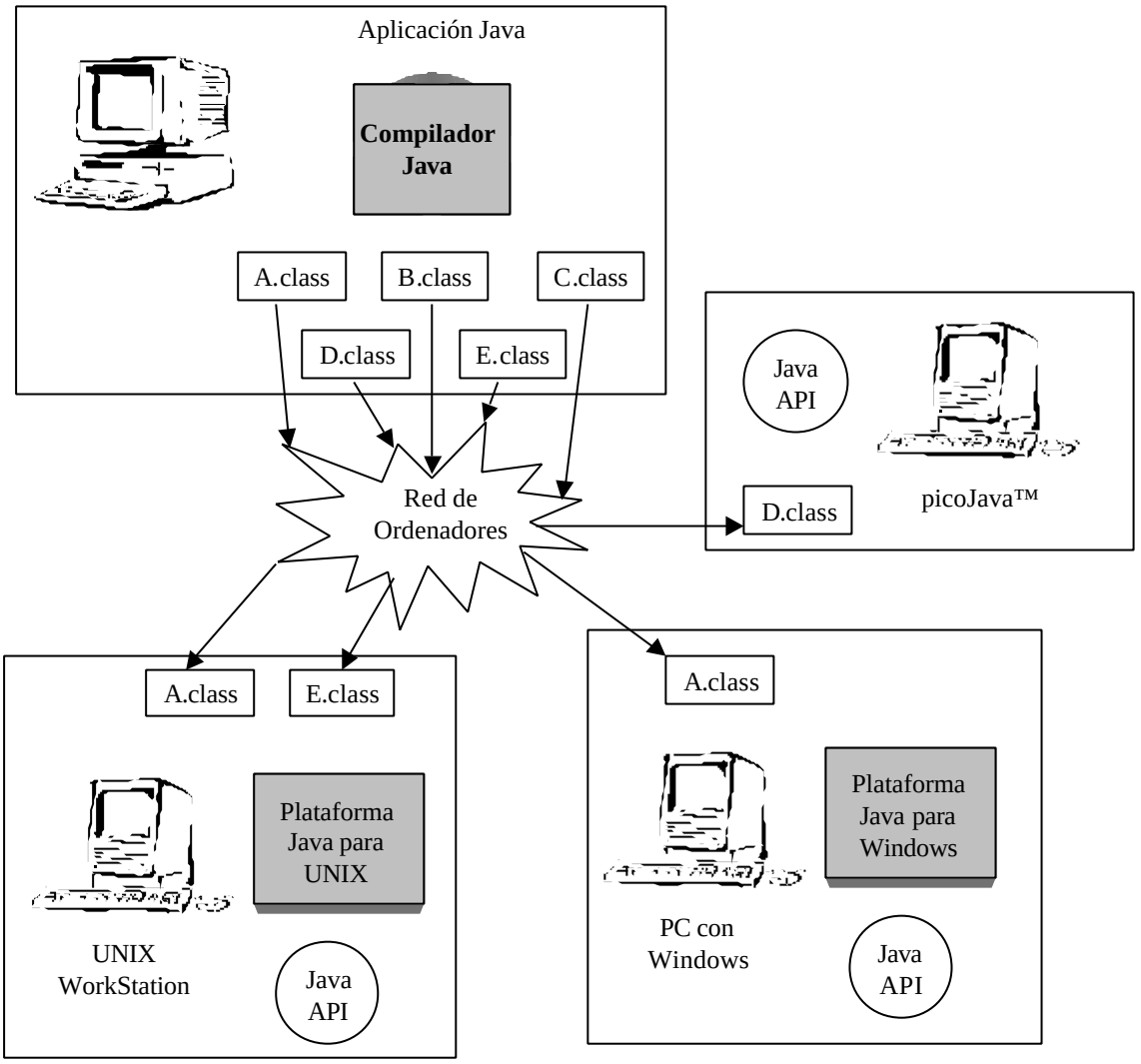


**Figura 5: Entorno de programación distribuida en Java.**

En la Figura 5 se aprecia cómo se enlazan los distintos conceptos mencionados. En un determinado equipo, se diseña una aplicación y se compila al código especificado como binario de la máquina abstracta. Este código ha de ser independiente de la plataforma en la que fue compilado, pudiéndose interpretar en cualquier arquitectura.

Parte de esta aplicación es demandada por una máquina remota. La porción del software solicitado es obtenido a través de la red de comunicaciones y es interpretado por la implementación del procesador en el entorno cliente. La forma en la que la aplicación cliente solicita el código a la máquina remota, es definida en su *class loader* (un ejemplo típico es un applet que se ejecuta en el intérprete de un *browser*).

El código obtenido puede ser "controlado" por un *security manager* definido en la aplicación cliente. De esta forma la máquina virtual chequea los accesos a recursos no permitidos. Finalmente se identifican distintas plataformas clientes ejecutando la misma aplicación.

Aunque en la especificación de la máquina virtual de Java [Sun95] no se identifica una implementación, el comportamiento de ésta[5] se describe en términos de subsistemas, zonas de

[5] Cuando identificamos la especificación de una máquina abstracta en 2.3, un punto necesario era la descripción semántica de sus instrucciones. Esta especificación identifica el comportamiento de la máquina frente a la ejecución de cada instrucción.

memoria, tipos de datos e instrucciones. En la Figura 6 se muestran los subsistemas y zonas de memoria nombrados en la especificación.

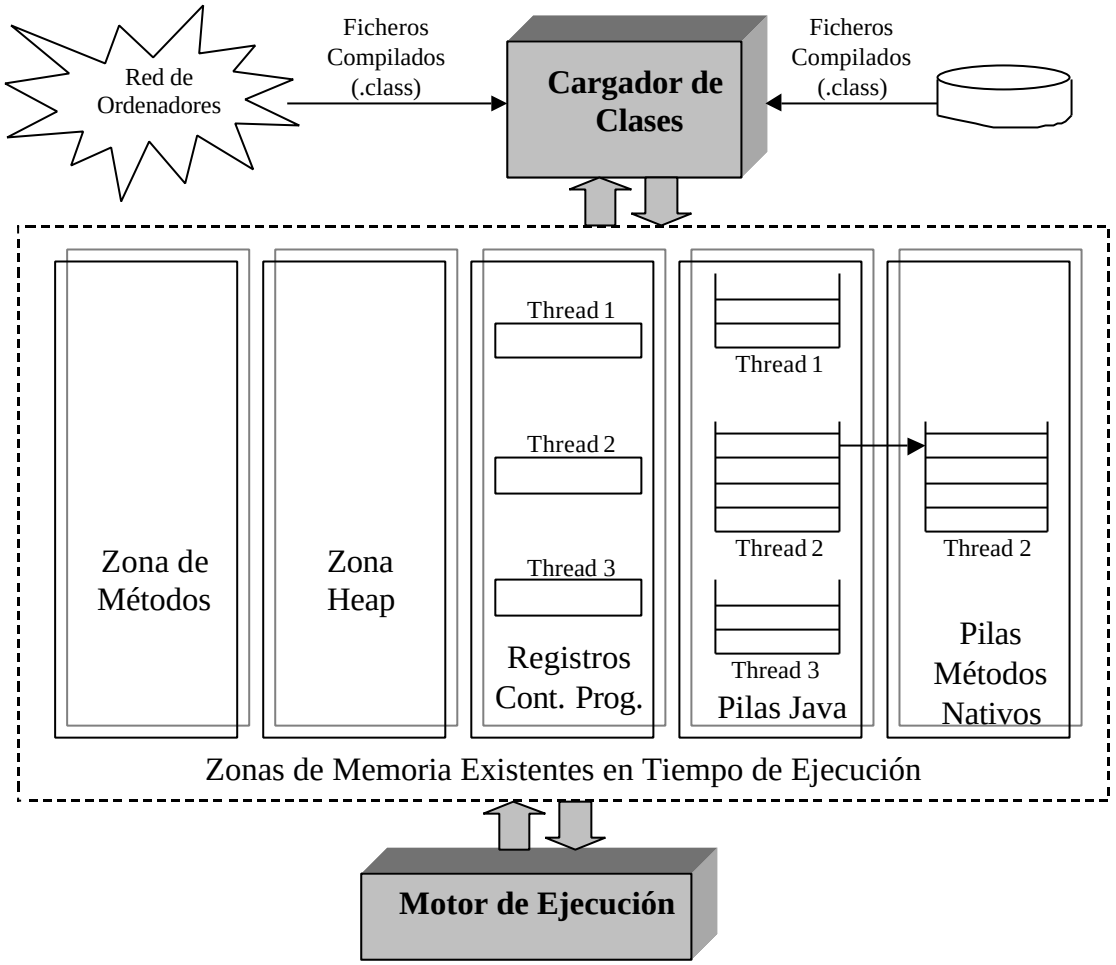


**Figura 6: Arquitectura interna de la máquina virtual de Java.**

La máquina virtual tiene un subsistema de carga de clases (*class loader*): mecanismo utilizado para cargar en memoria tipos –clases e *interfaces*. La máquina también tiene un motor de ejecución (*execution engine*): mecanismo encargado de ejecutar las instrucciones existentes en los métodos de las clases cargadas.

La máquina virtual identifica zonas de memoria que necesita para ejecutar un programa. Existen dos tipos de zonas de memoria:

Las inherentes a la ejecución de la máquina virtual (una zona por cada ejecución de la máquina).

Las inherentes a los hilos (*threads*) de ejecución dentro de una máquina (una zona por cada hilo existente en la máquina).

En el primer grupo tenemos el área de métodos y el área *heap.* En la zona de métodos se introduce básicamente la información o los datos propios de las clases de la aplicación. En la zona *heap* se representan los distintos objetos existentes en tiempo de ejecución.

Por cada hilo en ejecución se crea una zona de pila, un registro contador de programa y una pila de métodos nativos. En cada hilo se va incrementando el contador de programa por cada ejecución, se van apilando y desapilando contextos o marcos en su pila y se crea una pila de método nativo si se ejecuta un método de este tipo.

Finalmente comentar que, tras el éxito de la plataforma Java, Javasoft está trabajando con Sun Microelectronics™ para desarrollar la familia de procesadores JavaChip™ [Kramer96]: picoJava™, microJava™ y ultraJava™ [Sun97]. Estos microprocesadores son implementaciones físicas de intérpretes de la máquina abstracta de Java optimizados para las demandas de esta plataforma como son la multitarea y la recolección de basura.

El sistema operativo de Java, JavaOS [Madany96], puede estar en la RAM de un micro JavaChip™. De esta forma es realmente fácil diseñar ordenadores cuya plataforma de ejecución sea Java [Kramer96].

## 2.6 Diferencias y Similitudes entre las Máquinas Existentes.

Hemos visto como a lo largo de la historia, se han desarrollado diversas especificaciones e implementaciones de máquinas abstractas con objetivos, en algunos casos similares, y en otros bastante distintos.

En un principio el concepto de máquina abstracta fue utilizado en el desarrollo de compiladores. Se mostró cómo en ocasiones era interesante generar un código intermedio de una máquina abstracta, para después traducirlo a múltiple plataformas [Cueva92]. También se simplificaba en el caso de los intérpretes: es más sencillo interpretar un código intermedio ya compilado, que directamente el lenguaje de alto nivel.

Posteriormente se utilizó este código intermedio como una herramienta de portabilidad. El código intermedio generado en cualquier plataforma, sería portable a aquél sistema que tuviese un intérprete de esta máquina [Campbell83]. Las diferencias entre la traducción a una plataforma existente y su interpretación son las propias de la dualidad compilador-intérprete:

| | Compilador | Intérprete |
|---|---|---|
| Tiempo de Ejecución | - | + |
| Tiempo de Compilación | + | - |
| Flexibilidad | - | + |
| Portabilidad de Código | - | + |

Un ejemplo de esta diferencia es la gran portabilidad del lenguaje interpretado Tcl/Tk [Flynt98] y su menor eficiencia en tiempo de ejecución frente a otros lenguajes totalmente compilados a la plataforma destino como el C++ [Stroustrup98].

El siguiente paso lo dio la aparición de la máquina virtual de Smalltalk-80. Se diseñó un sistema entero sobre la especificación de la máquina buscando una flexibilidad y facilidad a la hora de programar aplicaciones. El paradigma de programación de la propia máquina era incluso de un alto nivel de abstracción –orientación a objetos, eventos, primitivas gráficas y la famosa arquitectura “modelo vista controlador” [Mevel87]–rompiendo con las especificaciones existentes.

Aunque se consiguió una portabilidad del código, el aspecto más buscado con la especificación de una máquina abstracta era obtener las ventajas propias de un intérprete en un sistema homogéneo en el que tan sólo existen objetos. Así la depuración de aplicaciones, la documentación de éstas, el almacenar el estado del sistema y el conocer en todo momento qué objetos existen son facilidades propias del Smalltalk-80.

Finalmente Sun identifica el lenguaje de programación Java orientado a objetos, con sintaxis similar a C++, más sencillo de programar y mantener y con un API sencillo y potente [Gosling96]. Para obtener la portabilidad del código y poder realizar aplicaciones distribuidas

multiplataforma, especifica una máquina abstracta que da soporte directo a este lenguaje [Lindholm96].

Buscando una plataforma independiente, distribuida y comercial, añade en la máquina virtual un sistema de seguridad (*security manager*) y la posibilidad de implementar un sistema de distribución de software (*class loader*). Añadiendo al API estándar un *middleware* de comunicación, se obtiene una plataforma dispuesta a ejecutar aplicaciones distribuidas.

Un ejemplo de cómo realiza una aplicación distribuida en Java, es haciendo uso de sus conocidos Applets. Un escenario común se muestra en la Figura 7:

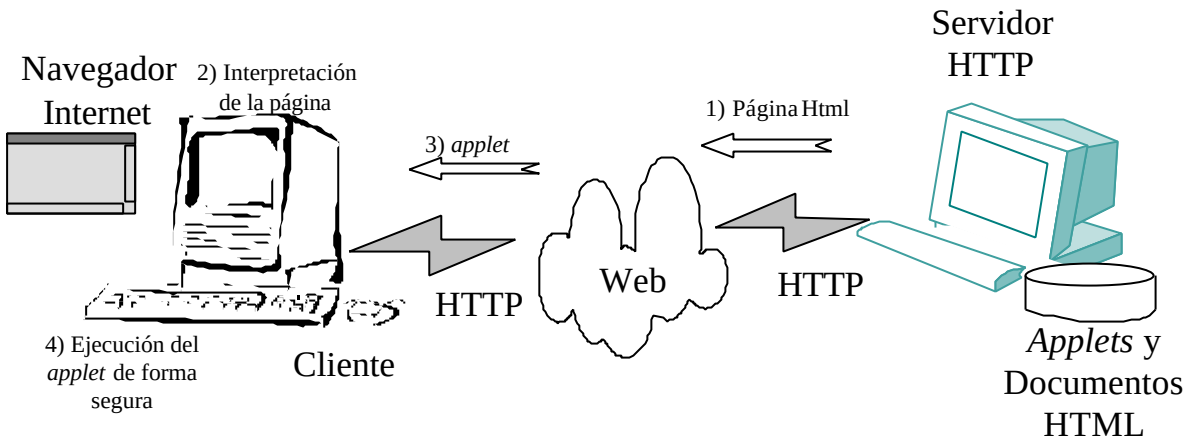


**Figura 7: Ejemplo de una aplicación distribuida sobra la plataforma Java.**

En una máquina servidora se ejecuta un servidor de HTTP (Hypertext Transfer Protocol) [Beners96]. Ésta tiene en su disco un conjunto de documentos HTML (Hypertext Markup Language) [Beners93, Raggett97]. El cliente ejecuta un navegador de Internet (*Web Browser* ) y solicita un documento HTML al servidor. El cliente obtiene el documento gracias al protocolo HTTP que existe sobre la conexión física.

El cliente interpreta el documento obtenido y encuentra un enlace a un *applet* (una parte de la aplicación compilada para la máquina de Java). Obtiene este *applet* del servidor HTTP y lo ejecuta en el intérprete software de la máquina que tiene implementado el propio navegador. El sistema de seguridad (*security manager*) de la máquina del navegador impide que el *applet* acceda a recursos de la máquina cliente.

En este ejemplo se muestra cómo Java es una plataforma independiente, su código es fácilmente distribuible y posee un sistema de seguridad incorporado.

# Capítulo 3:

# Arquitecturas de Objetos Distribuidos

La idea de identificar un ordenador como una "isla", encerrada computacionalmente, esta empezando a ser sustituida por el paradigma de la computación distribuida y cooperativa entre distintos ordenadores interconectados [OMG97].

Con la aparición de las redes de computadores e Internet, cada vez más se demanda la existencia de sistemas distribuidos, que han sufrido una evolución a lo largo de la historia:

**Sistemas monolíticos**: En un único ordenadores potentes o *mainframes* se guardaba todos los módulos de computación y los datos. El software se mantenía de forma centralizada y daba servicio a múltiples clientes, distribuidos físicamente, a través de las terminales.

Los sistemas eran monolíticos, es decir los distintos módulos de la aplicación (interfaz de usuario, la lógica central de la aplicación y los sistemas gestores de bases de datos) estaban todos unidos en la misma aplicación. El hecho de tener un conjunto de terminales para acceder a ésta, implicaba tan sólo una simple distribución física puesto que las terminales no tenían su propio sistema de procesamiento.

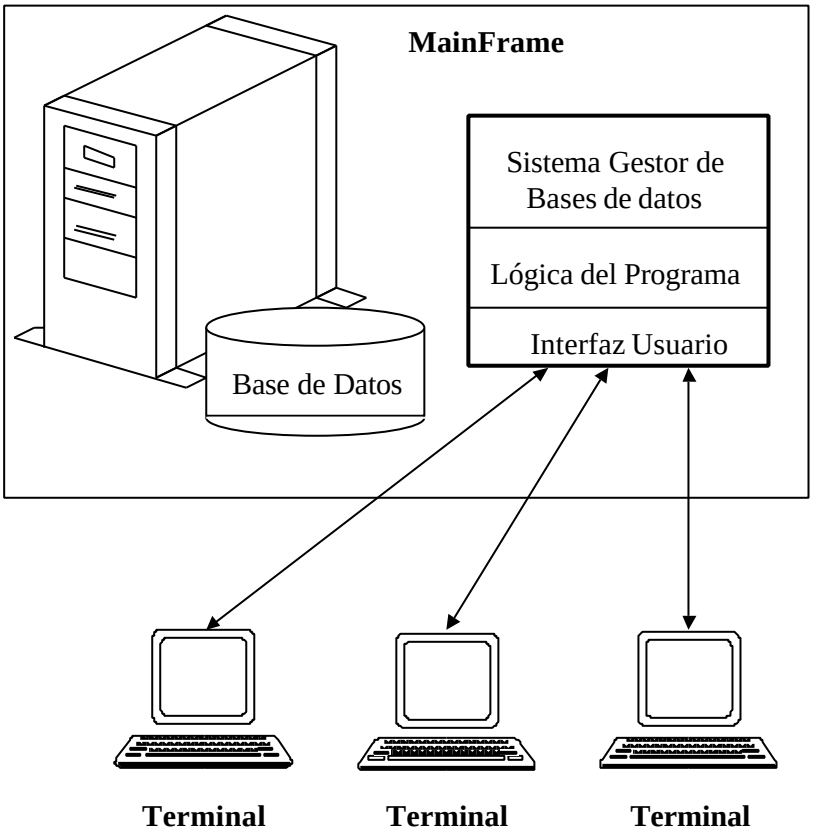


**Figura 8: Ejemplo de aplicación sobre un sistema monolítico.**

**Arquitectura Cliente/Servidor**: Con la aparición del ordenador personal, se hizo posible un cambio en los sistemas monolíticos existentes. Los PCs procesan una parte de la aplicación global (normalmente la parte de la interfaz), descargando al ordenador central de procesar determinados módulos de computación.

La división de la aplicación en las dos zonas de procesamiento cliente y servidor, reducen las necesidades de recursos de procesamientos del ordenador central –propias de un mainframe –y permiten al usuario cliente ejecutar localmente aquellas aplicaciones no distribuidas.

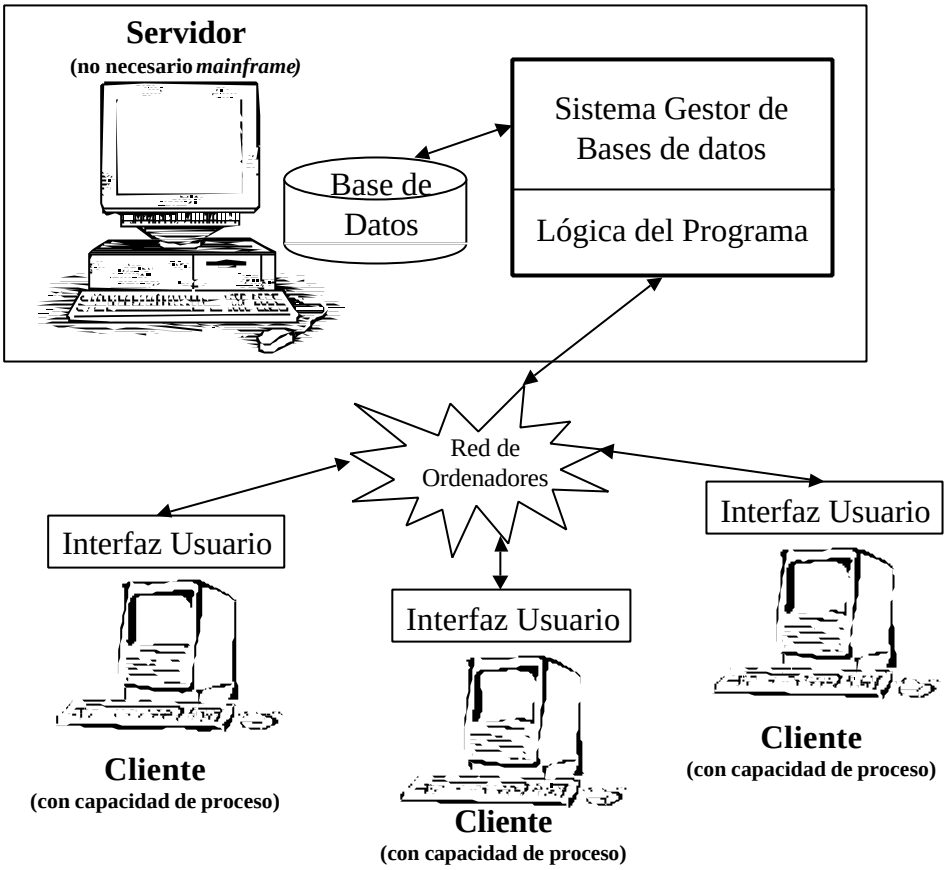


**Figura 9: Ejemplo de una aplicación sobre una arquitectura cliente/servidor.**

**Arquitecturas multicapa (multitier architecture)**: Ampliando el concepto de división computacional propio de la arquitectura Cliente/Servidor, surge el concepto de arquitecturas multicapa: La distribución de la aplicación se hace en un determinado número de niveles o capas. De esta forma una aplicación cliente/servidor puede ser vista como una aplicación de dos niveles.

Las ventajas propias de esta división son velocidad de procesamiento (al utilizar varios ordenadores de forma concurrente) y el mantenimiento del software (propios de la modulación del mismo) [Meyer97]. Un tipo de arquitectura muy utilizada comercialmente es la aplicación de 3 capas separando la parte del cliente, la aplicación central o servidora y el servidor de la base de datos [Edwards97]. Este tipo de aplicaciones son eficientes y fácilmente mantenibles.

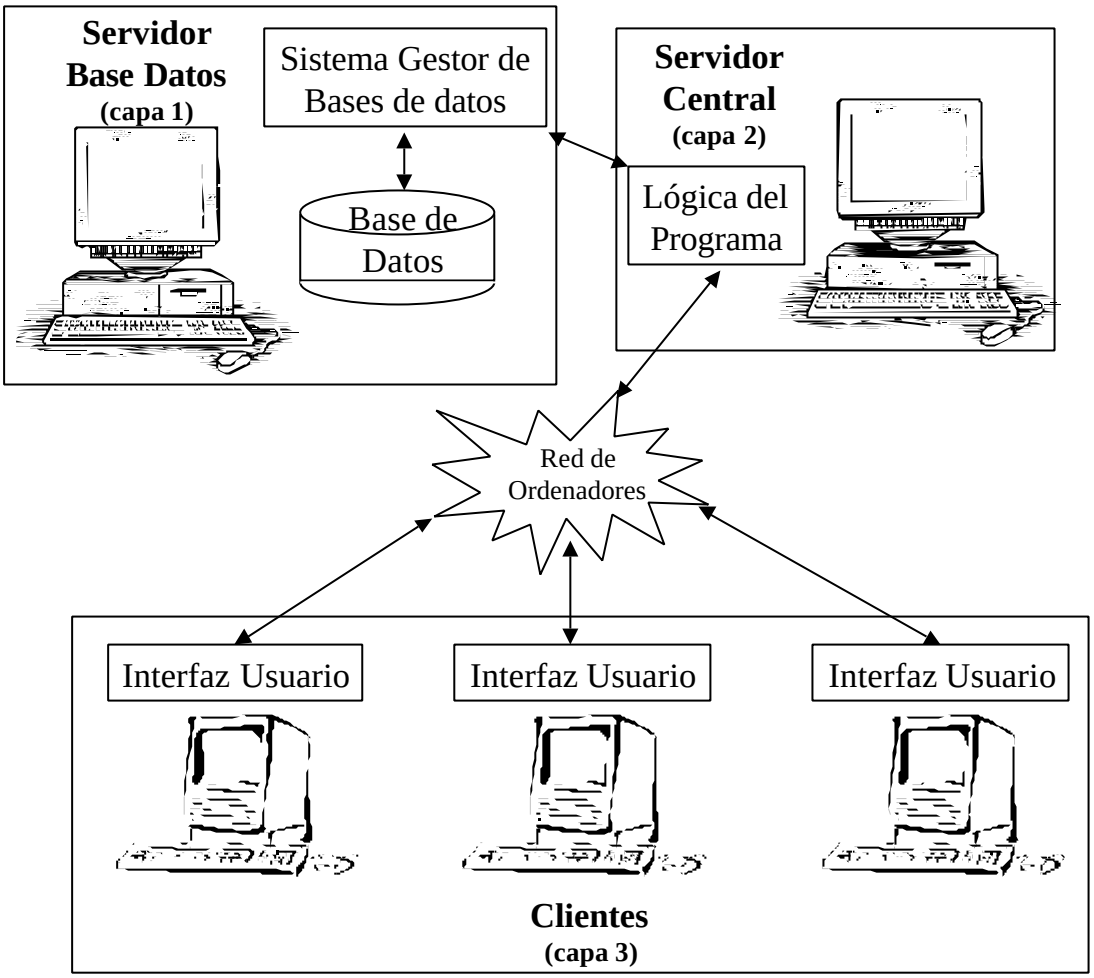


**Figura 10: Ejemplo de división de una aplicación distribuida en 3 capas.**

**Sistemas de objetos distribuidos**: Se está trabajando en descripciones de modelos de arquitecturas de sistemas distribuidos. En lugar de diferenciar entre capas de la aplicación, toda ella se divide en objetos y éstos se ubican en distintas máquinas bajo una misma arquitectura. La identificación de clientes o servidores es inicua, puesto que un objeto puede operar bajo las dos responsabilidades de forma indiferente.

Las arquitecturas de sistemas distribuidos deberán ser totalmente abiertas, pudiendo formar parte de ésta cualquier tipo de protocolos, plataformas, sistemas operativos o sistemas de comunicaciones [OMG97]. Se busca que la reutilización de los recursos software y hardware existentes sea máxima.

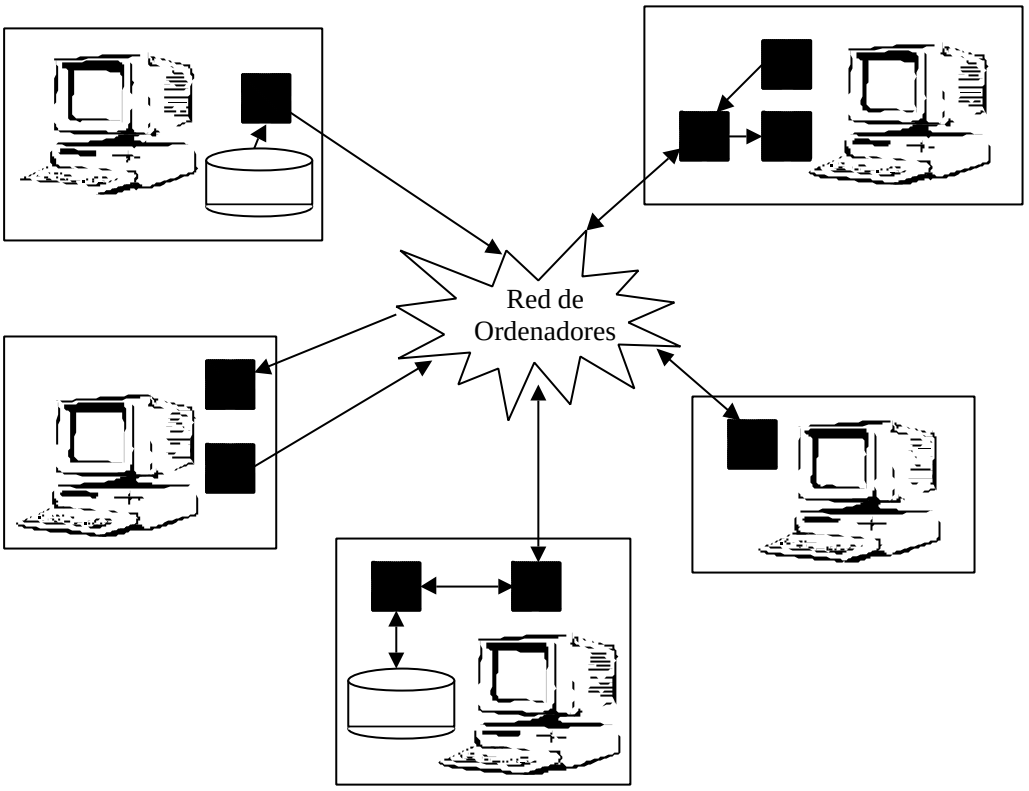


**Figura 11: Ejemplo de una aplicación sobre una arquitectura de objetos distribuidos.**

## 3.1 Alternativas a las Arquitecturas Distribuidas.

Para el desarrollo de aplicaciones distribuidas existen una serie de alternativas a las arquitecturas o sistemas distribuidos. En función de los intereses de una aplicación, puede convenir utilizar una determinada arquitectura o hacer uso de las alternativas comúnmente más sencillas.

Los distintos mecanismos de interconexión de aplicaciones han evolucionado a lo largo del tiempo y se utilizan incluso como primitivas para el protocolo de comunicaciones de las arquitecturas distribuidas. Cabe destacar:

Programación mediante *sockets*: En muchos sistemas, las comunicaciones entre distintas máquinas, y entre procesos de una misma máquina, se realizan mediante el uso de *sockets*. Se crea un *socket* en un puerto software de un ordenador y las aplicaciones pueden comunicarse mediante la lectura y escritura de información en este *socket.*

La interfaz de programación de un *socket* es de un nivel de abstracción bastante bajo (se utilizan las primitivas propias de un archivo). Esto hace que la sobrecarga de las comunicaciones sea realmente poca y por lo tanto puede interesar su utilización en determinados tipos de aplicaciones.

Llamada a procedimientos remotos RPC (*Remote Procedure Call*): Un salto por encima de la programación en *sockets* es la llamada a procedimientos remotos –el uso de RPCs –que proporciona un nivel de comunicación orientado a la invocación de funciones.
El uso de RPCs ofrece una mayor facilidad a la hora de programar de forma distribuida frente a los *sockets.* No nos tendremos que preocupar de cómo se debe repre-

sentar la información entre las distintas plataformas, sino que nos limitamos a invocar un procedimiento remoto.

DCE: Entorno de Computación Distribuida. Define un conjunto de estándares guiados por la fundación de software abierto OSF (*Open Software Fundation*), incluyendo una especificación estándar de RPCs, servicio de nombrado o denominación distribuido, servicio de sincronización, sistemas de archivos distribuidos, servicio de seguridad de red y un paquete de hilos (*threads*) [OSF96].
La especificación del protocolo de comunicaciones de DCOM (*Distributed Object Component Model*) ha sido realizada sobre este entorno [Brown98].

Java RMI (*Remote Method Invocation*) [Sun97b]: Sin llegar a ser una arquitectura distribuida y abierta, se acerca mucho al objetivo buscado. Se basa en permitir la invocación remota a métodos de objetos no ubicados en el mismo equipo. Su principal inconveniente es que es una forma de comunicar aplicaciones Java: tanto el servidor como el cliente han de ser objetos ejecutándose sobre la plataforma Java [Kramer96]. Actualmente se han desarrollado puentes entre RMI y las arquitecturas distribuidas existentes.

Ante todas las alternativas existentes, la decisión de qué sistema elegir para implementar una aplicación está en función de los requisitos de la propia aplicación. Sin embargo, las arquitecturas o sistemas de objetos distribuidos, han sido concebidos como un *middleware* de desarrollo de aplicaciones distribuidas, llevando a cabo todas las operaciones repetitivas que comúnmente se realizan en este tipo de aplicaciones (establecimiento de un protocolo, representación de la información, recepción de mensajes,...) [OMG95].

## 3.2 Objetivos de las Arquitecturas de Objetos Distribuidos.

El principal objetivo de las arquitecturas de objetos distribuidos es definir un *middleware* para el desarrollo de aplicaciones orientadas a objetos distribuidas [Orfali98]. Se trata de dar un soporte a componentes u objetos distribuidos, para que puedan ser combinados e interactúen para formar una aplicación en varios ordenadores.

La arquitectura deberá ser capaz de interconectar componentes entre distintas plataformas, gestionando el acceso a sus propiedades y métodos y controlando sus eventos. El *middleware* deberá subir el nivel de abstracción de programación para conseguir estos objetivos sin tener que codificar su funcionalidad en alguna de las alternativas vistas en 3.1.

Las aplicaciones distribuidas tienen una serie de características y requerimientos que hacen que aparezcan dificultades a la hora de definir un entorno de objetos distribuidos [Longshaw99]. Los problemas más destacables son:

La latencia de las llamadas remotas: La invocación remota de métodos requiere tiempos de ejecución mayores que las invocaciones a métodos locales. Esto supone un problema en los métodos selectores y modificadores de los atributos de un objeto[6].

Gestión de memoria: La gestión de distintos espacios de memoria de varios equipos distribuidos complica el modo en el que se maneja un solo espacio de direcciones. Por ejemplo, la invocación de métodos remota de Java [Sun97b] implementa un recolector de basura distribuido, mientras que CORBA [OMG95] carece de éste.

Errores parciales en la aplicación: Los sistemas distribuidos tienen una mayor probabilidad de error en la invocación a métodos remotos, a causa de la fiabilidad en las

[6] Comúnmente conocidos como métodos *getters* y *setters*. Obtienen y modifican los valores de los atributos de un objeto. Actualmente forman parte de la especificación de componentes Java [Sun96].

comunicaciones. Se puede perder enlace con un conjunto de componentes de otra plataforma, dejando la aplicación global en un estado de "error parcial".

Control del "ciclo de vida" de los objetos: La creación de objetos se realizará a través de la red mediante objetos intermedios, puesto que no se pueden crear directamente de forma remota. Estos objetos deben proporcionar un servicio de manejo de referencias para que el objeto creado pueda ser accedido desde cualquier máquina (realizando la inicialización de una referencia).

Utilización de servicios: En este tipo de aplicaciones aparecen una serie de necesidades comunes a distintas partes de la aplicación. Puesto que se trabaja en un entorno distribuido, los servicios podrán ser requeridos por cualquier objeto que esté ubicado en cualquier máquina. Estos servicios han de estar muy bien diseñados para su reutilización, actuando como componentes disponibles para el desarrollador de aplicaciones. Servicios comunes son:

Nombrado o denominación (*Naming Service*).

Control del ciclo de vida (*Life Cycle Service*)

Persistencia y manejo de estado de objetos (*Persistence and State Management*).

Propagación de eventos (*Event Propagation*).

Paso asíncrono de menajes (*Asynchronous Messaging*).

Seguridad de acceso (*Security Enforcement*).

Soporte de transacciones (*Transaction Support*).

Mediante una arquitectura de objetos distribuidos como se aprecia en la Figura 10, se podrá diseñar una aplicación distribuyéndola en distintas máquinas y pudiendo acceder a todos los objetos con independencia de la plataforma, sistema operativo, red de comunicaciones o protocolo que se esté utilizando.

## 3.3 Arquitecturas de Objetos Distribuidos Existentes.

Las arquitecturas de objetos distribuidos poseen un gran potencial debido a la posibilidad de servir como marco de trabajo de componentes u objetos distribuidos. Se trata de crear aplicaciones a través del ensamblado de componentes software que interoperen a través de las redes, ejecutándose sobre distintas plataformas y sistemas operativos.

El desarrollo del software está orientado al uso de componentes distribuidos. Especificaciones de este tipo de componentes son los EJB (*Enterprise Java Beans* [Sun98b]) de Sun, los componentes COM+ (*Component Object Model* [Kirtland97]) de Microsoft y los CCA (*CORBA Component Architecture* [OMG98]) de OMG.

Actualmente existen dos estándares en el mercado: CORBA de OMG y COM de Microsoft.

### 3.3.1 CORBA.

CORBA (*Common Object Request Broker Architecture*) es un ambicioso proyecto para el diseño de un *middleware* que proporcione una arquitectura de objetos distribuidos en la que puedan definirse e interoperar distintos componentes [OMG97].

CORBA es un producto de un consorcio llamado OMG (*Object Management Group*) que incluye un conjunto de 800 compañías, entre las que se encuentra uno de sus principales competidores: Microsoft (con su especificación COM).

CORBA utiliza el concepto de objeto como el único elemento existente en la arquitectura distribuida o, visto desde otro punto de vista, como la interfaz propia de cualquier aplicación que desee utilizar este *middleware*. La especificación de estos objetos es a nivel de interfaz: Sólo se define el aspecto del objeto; nunca su implementación. Esto hace que la especificación CORBA sea lo suficientemente abierta como para incorporar cualquier aplicación existente a la red.

En el diseño de CORBA, los componentes son inteligentes puesto que son capaces de descubrirse los unos a los otros en tiempo de ejecución e interoperar entre ellos. Otros servicios que nos proporciona son la creación y borrado de objetos, su búsqueda por el nombre, el almacenamiento en un soporte secundario, el acceso a su estado y la definición relaciones entre ellos.

CORBA es parte del proyecto OMA (*Object Management Architecture*) [OMG96] en el que se especifica un conjunto de servicios universales para la gestión de objetos [OMG95b] y un conjunto de facilidades en el desarrollo de aplicaciones específicas [OMG95c].

La principal característica de CORBA es que crea especificaciones de los objetos mediante simples interfaces. Estas especificaciones están escritas en un lenguaje de especificación de interfaces neutral denominado IDL (*Interface Definition Language*) [OMG95]. La interfaz define el conjunto de operaciones que un cliente puede reclamar a ese objeto como servidor. La carencia de especificación de la implementación del objeto hace que el lenguaje y plataforma utilizado sean totalmente abiertos.

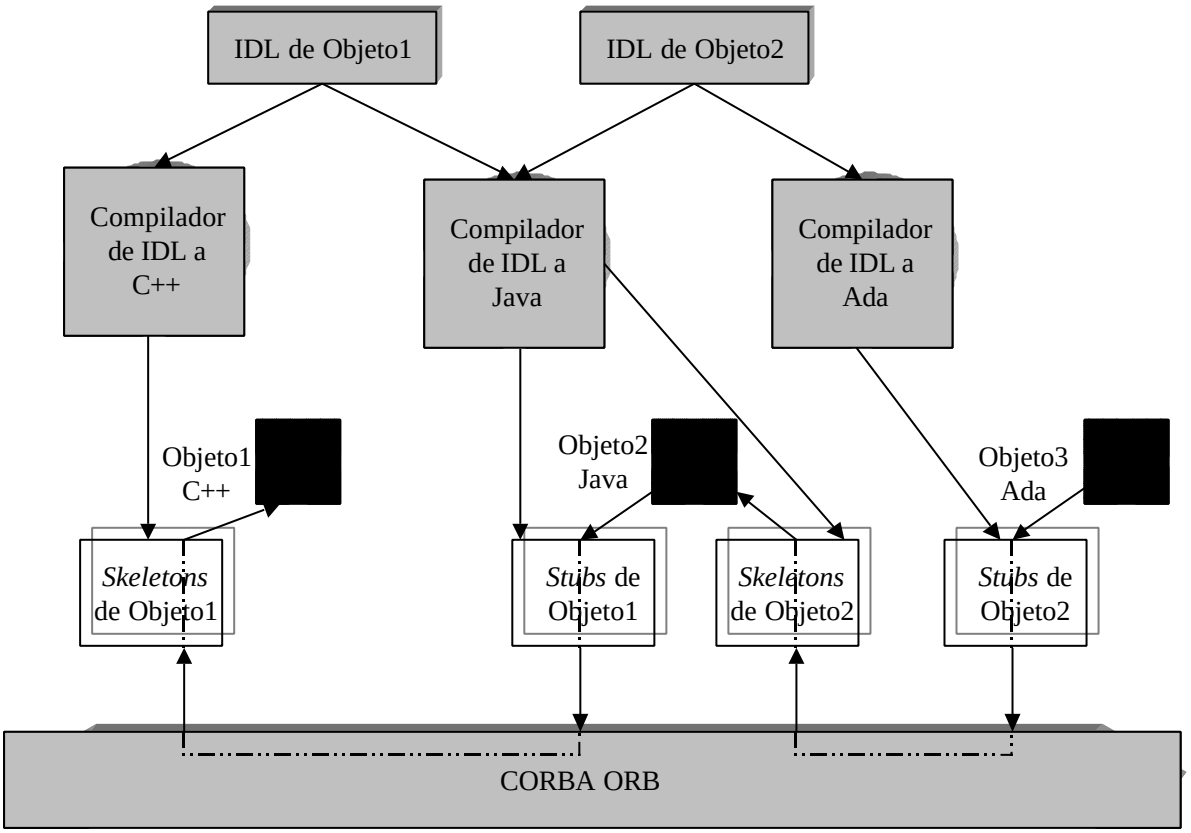


**Figura 12: Ejemplo de interconexión de objetos CORBA mediante un ORB.**

En la Figura 12 se muestra cómo se pueden interconectar objetos CORBA. Se especifica en IDL la interfaz de los objetos que van a ser servidores. Con un compilador de IDL, se

generan las traducciones de la interfaz para la petición de servicios (*client stubs*) por parte del cliente y para la ejecución de eventos (*server skeletons*) por parte del servidor.

El código generado para cada cliente y servidor, al traducir los archivos IDL, ha de ser el propio elegido en la implementación de éstos (se supone la posesión de compiladores de IDL a cualquier lenguaje). Este código generado se enlaza con la implementación de los objetos y con el código necesario para realizar la interconexión de objetos: el ORB (*Object Request Broker*).

Una vez ejecutadas todas las partes de la aplicación distribuida, cuando un cliente invoque una operación o requiera una propiedad de un objeto servidor, el ORB enviará esta petición al objeto servidor. Éste podrá devolver en la dirección contraria, el resultado de la solicitud del cliente apoyándose de nuevo en el ORB.

#### 3.3.1.1 Estructura de un ORB.

En la Figura 13 se muestra detalladamente las partes de un ORB en la especificación CORBA 2.0 [OMG95] distinguiendo desde el punto de vista del cliente y del servidor [Orfali97]. La unión de todos los módulos busca dos objetivos:

Que el ORB intercepte la invocación de un método servidor desde un objeto cliente, busque el objeto remoto que implemente la petición, le pase los parámetros, se ejecute el método y se devuelva el resultado.

Que la invocación del método servidor se pueda realizar de forma estática (decidiendo a que método de que objeto se invocará en tiempo de compilación) o de forma dinámica (consultando y seleccionando las propiedades y operaciones de los objetos remotos en tiempo de ejecución).

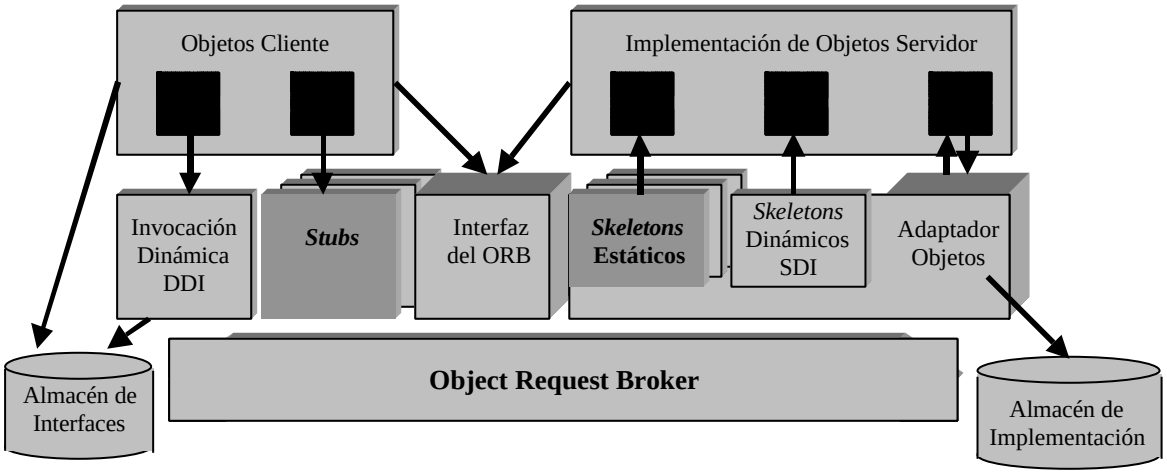


**Figura 13: Estructura en módulos de un ORB.**

Los módulos con los que interactúa el objeto cliente son:

**Los *stubs* del cliente**: Proporcionan la interfaz de llamada estática a los servicios de objetos remotos. Como se muestra en la Figura 12, el compilador de IDL genera este módulo para todo cliente del objeto especificado en el IDL. El código generado actúa como un *proxy* local para las llamadas a métodos remotos.

**Interfaz de invocación dinámica DII (*Dynamic Invocation Interface*)**: Permite al cliente descubrir los métodos de un objeto en tiempo de ejecución. CORBA define unas APIs estándar para buscar los metadatos que definen la interfaz de un objeto remoto.

**APIs de manejo del almacén de interfaces**: Permiten obtener y modificar la descripción de todas las interfaces de los objetos registrados, los métodos que implementan y los parámetros que éstos requieren. El almacén de interfaces es una base de datos en la que se almacena la descripción de todos los objetos de la aplicación. La información que almacena –los IDLs de los componentes –es el soporte para la invocación dinámica de métodos.

**La interfaz del ORB**: es un conjunto de librerías de servicios locales que son de interés para desarrollar la aplicación desde el punto de vista del cliente. Un ejemplo puede ser una función para pasar una referencia a un objeto a una cadena de caracteres.

El servidor no distinguirá entre las invocaciones estáticas o la realizadas dinámicamente mediante el DII. En ambos casos, el ORB utilizará el módulo del adaptador de objetos para invocar al objeto servidor. A continuación definimos brevemente los módulos que se aprecian en el lado del servidor, en la Figura 13:

**Los *skeletons* estáticos**: Proporcionan una interfaz estática para cada servicio expuesto por el servidor. Este módulo es creado, al igual que los *stubs* del cliente, mediante la compilación de un archivo IDL (como se aprecia en la Figura 12).

**Interfaz de *skeletons* dinámicos DSI (*Dynamic Skeletons Interface*)**: Proporcionan el enlace dinámico para los servidores que no tienen una versión de *skeletons* generada mediante un compilador de IDL. Este módulo analiza los valores de los parámetros en una invocación, e interpretan el objeto al que se está invocando. Se utiliza a menudo para implementar interconexiones entre distintos ORBs.

**El adaptador de objetos**: Se sitúa en la parte del servidor interactuando directamente con el ORB y aceptando las peticiones de mensajes en nombre de los objetos servidores. Proporciona un entorno en tiempo de ejecución para la creación de objetos, el paso de mensajes y la asignación de referencias a objetos. Además registra las clases y los objetos existentes en el almacén de implementaciones.

**Almacén de implementaciones**: Es un almacén gestionado en tiempo de ejecución que alberga información sobre las clases de los objetos servidores, las instancias de éstas y sus identificadores o referencias.

**La interfaz del ORB**: Consiste en unas APIs de servicios locales que son idénticas a las proporcionadas en la parte del cliente.

### 3.3.2 Microsoft COM.

El modelo de componentes de Microsoft (COM) es un estándar binario para la creación y manejo de objetos en máquinas Windows[7], bajo un determinado número de reglas [Microsoft95]. La regla principal está basada en una negociación de interfaces para conseguir la tabla de métodos virtuales de un objeto.

COM ofrece servicios para manejar el ciclo de vida de los objetos mediante un contador de referencias, servicio de autorización o seguridad y servicio gestor de eventos además de soportar encapsulamiento, polimorfismo y agregación (como recurso para la herencia dinámica).

Se propuso que el modelo de componentes COM pudiese ser extendido a la conexión de componentes entre distintas máquinas y en consecuencia construirlo utilizando RPCs del

[7] Se suele denominar *Máquina Windows* a aquel ordenador que utiliza como sistema operativo Windows (NT o 95).

entorno de computación distribuida (DCE) definido por OSF (*Open Software Fundation*) [OSF96]. Se le proporcionaron a COM identificadores de interfaces de 128 bits denominándolos identificadores globales únicos GUIDs (Globaly Unique Identification). En las primeras versiones de COM no se permitían accesos a objetos remotos hasta que se diseñó el COM distribuido o DCOM que se introdujo primero en Windows-NT 4 y posteriormente en Windows 95 [Brown98].

COM ha sido utilizado en el desarrollo de aplicaciones distribuidas sobre máquinas Windows pero tuvo carencias frente a la arquitectura CORBA. Las carencias venían dadas por las ventajas que tiene el uso de un ORB (*Object Request Broker*) [OMG95]: un mínimo de soporte para gestionar creación de objetos, reutilización y su destrucción automática en nombre del cliente. El resultado de esta diferencia era que la programación de servidores COM era más compleja que su similar sobre CORBA.

Para resolver el problema del diseño de servidores COM y la gestión de transacciones, en 1997 Microsoft lanzó su producto *Microsoft Transaction Server* (MTS) [Microsoft98]. En éste se implementa un entorno de transacciones y un sistema de petición de objetos u ORB, creando y manejando los objetos bajo las peticiones del cliente.

MTS proporciona una interfaz simple para la seguridad basada en roles, manejo automático de hilos facilitando el diseño de servidores, creación dinámica de objetos bajo pedido y persistencia de objetos. Permite así a los objetos creados en COM trabajar juntos para crear sofisticadas aplicaciones distribuidas.

COM+ será entregado con Windows 2000 y Windows-NT 5 [Kirtland97]. COM+ añade a COM y a MTS un conjunto de servicios para el desarrollo de aplicaciones distribuidas basadas en componentes. Ha sido la respuesta comercial a la especificación de EJB propuesta por Sun, IBM y Oracle [Sun98b]. Actualmente está en vías de desarrollo.

La gran diferencia entre CORBA y COM está entre el tipo de componente u objeto distribuido que utilizan. CORBA apuesta por el software abierto, dejando la definición de un componente (IDL) como único requisito para tratar el objeto. La implementación de éste se producirá en cualquier lenguaje y entorno que pueda implantar la interfaz definida.

COM define un formato de componentes binarios (ActiveX) que puede ser ejecutado en una plataforma Windows [Robinson97]. Es independiente del lenguaje porque puede ser generado por diversos compiladores, pero su formato binario no es definitivamente abierto (actualmente se pueden ejecutar en plataformas Windows y Macintosh y se está tratando de implantar en diversas plataformas Unix).

Existen también especificaciones de interconexión entre las dos arquitecturas COM y CORBA [OMG98b].

## 3.4 La Unión de Arquitecturas Distribuidas y Plataformas Independientes.

En lo referente a las arquitecturas de objetos distribuidas, hemos visto cómo se busca la especificación de un *software* intermedio o *middleware* que permita realizar aplicaciones distribuidas de una forma genérica y estándar. Con esto se consigue lo que en el capítulo 3 definimos como sistemas de objetos distribuidos. Una aplicación se compone en un conjunto de objetos en ejecución en distintas máquinas, interconectados y sin la separación estática de puesto cliente y máquina servidora.

### 3.4.1 Características de las Arquitecturas Distribuidas.

En las arquitecturas existentes se identifican una serie de características comunes como la búsqueda de ese *middleware* para desarrollo de aplicaciones distribuidas, la independencia de las comunicaciones existentes, la especificación como un estándar y el uso de las tecnologías orientadas a objetos. También se apuesta por una arquitectura abierta aunque la forma de conseguirla difiera notablemente.

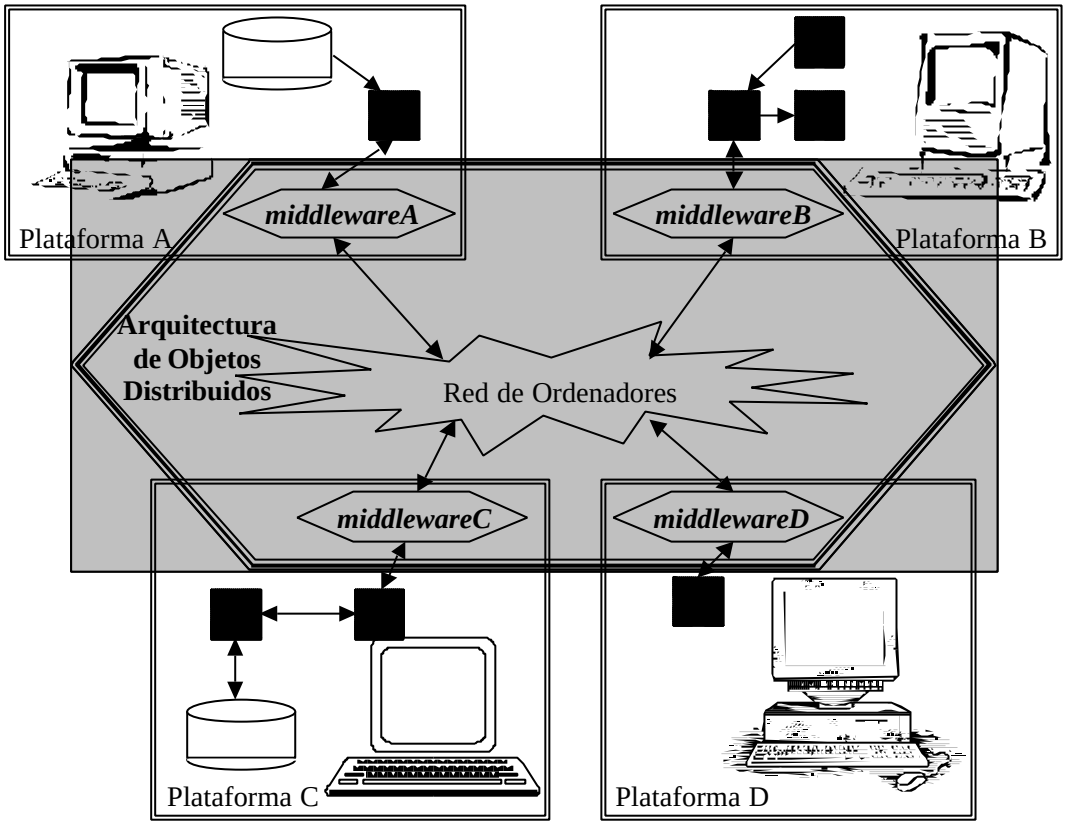


**Figura 14: Ámbito de una arquitectura de objetos distribuidos.**

En lo referente a Microsoft COM+ [Kirtland97] existe una especificación binaria del formato de los objetos o componentes ActiveX. La forma en la que se interconectan estos componentes a través de las redes se especifica en un protocolo DCOM [Brown98] que amplía el modelo de componentes a un entorno distribuido. Este protocolo utiliza el entorno de comunicaciones estándar DCE [OSF96] y por lo tanto abierto.

El problema fundamental en la utilización de la arquitectura COM no viene dado por la comunicación mediante un protocolo sino por la especificación binaria de sus componentes. El formato de éstos es binario para la plataforma Windows y por lo tanto poco portable[8]. La condición de arquitectura abierta pasa por la interpretación del formato de los componentes en cualquier plataforma. Esta tarea obviamente no es sencilla por la dependencia existente entre éstos y un tipo de sistema operativo específico.

La forma en la que CORBA ofrece una arquitectura de objetos distribuidos abierta es distinta al modelo COM. CORBA identifica la existencia de los objetos mediante las especificaciones de sus interfaces con un lenguaje neutro denominado IDL (*Interface Definition Language*)

[8] En la fecha de Marzo de 1999, la especificación del modelo de componentes propuesto por Microsoft está implementada en las plataformas Windows y Macintosh. Se está trabajando en incorporarlo a la mayoría de las plataformas Unix, aunque todavía no está disponible.

[OMG95]. Los propios objetos que proporcionan servicios y facilidades a las aplicaciones CORBA, están simplemente definidos mediante IDL.

El consorcio OMG se encarga de establecer traducciones estándar de este lenguaje a todos los lenguajes existentes, sean o no orientados a objetos. De esta forma, la implementación de los componentes u objetos distribuidos no están limitados ni a un lenguaje, ni a un sistema operativo y ni siquiera a un paradigma de programación como el orientado a objetos [Booch94]. Se pueden por lo tanto introducir cualquier tipo aplicación existente en un entorno distribuido, con tan solo amoldar la aplicación existente a una envoltura (*wrapper*) de objetos.

En CORBA se pueden realmente desarrollar aplicaciones en múltiples redes de ordenadores, lenguajes de programación, sistemas operativos e implementación de componentes obteniendo así una independencia total de plataforma.

### 3.4.2 Características de las Plataformas Independientes.

Como hemos comentado en el capítulo 2, uno de los objetivos buscados con la especificación de máquinas abstractas, era la obtención de una plataforma independiente.

Inicialmente con la máquina-p [Campbell83] se buscó la portabilidad de la plataforma de ejecución de las aplicaciones desarrolladas en Pascal. Posteriormente, con la especificación de la máquina virtual de Java [Sun95], se consiguió una plataforma independiente cuyo código binario se ejecutó en diversos entornos.

La búsqueda de una plataforma independiente se consigue en Java mediante la especificación de una máquina abstracta que puede ser interpretada en la mayoría de los equipos existentes. Se elimina cualquier dependencia entre la especificación y una determinada plataforma o sistema operativo existente en el mercado. Esta característica es la que hace precisamente que los componentes COM no sean catalogados comúnmente como independientes de una plataforma.

### 3.4.3 Unión de Conceptos.

El concepto de arquitectura distribuida abierta finaliza justo donde empieza el concepto de plataforma independiente. La especificación de una interfaz para los objetos hace que la arquitectura deje la implementación de éstos sin especificar. Una plataforma independiente identifica la implementación de una aplicación en un formato binario pero interpretable por cualquier plataforma real existente.

La unión de los dos conceptos consigue obtener un sistema de programación distribuido completamente portable y movible. En un sistema de estas características se resuelven tareas como la distribución de las aplicaciones en sus diversas versiones, el paso de objetos por valor y la movilidad de objetos. Además la implementación de todos los servicios y facilidades propuestos por OMG [OMG96] para el desarrollo de aplicaciones podrán ser implementados una única vez en una plataforma y utilizados en cualquier tipo de aplicación, sea cual sea la plataforma donde se ejecute.

En la Figura 15 se aprecia el ámbito de la unión de los conceptos. Denominamos la plataforma independiente como *plataforma X*. Toda la implementación del *middleware*, se realiza en una única plataforma y el conjunto global de los objetos se ejecutan bajo ésta. Para conseguir una portabilidad del sistema, nos limitamos a realizar un procesador de la máquina abstracta en cada uno de los sistemas existentes. Es importante resaltar cómo cualquier máquina en cualquier entorno sigue pudiéndose unir a este sistema de computación como se aprecia en la Figura 14, aunque no implemente el procesador de la plataforma X.

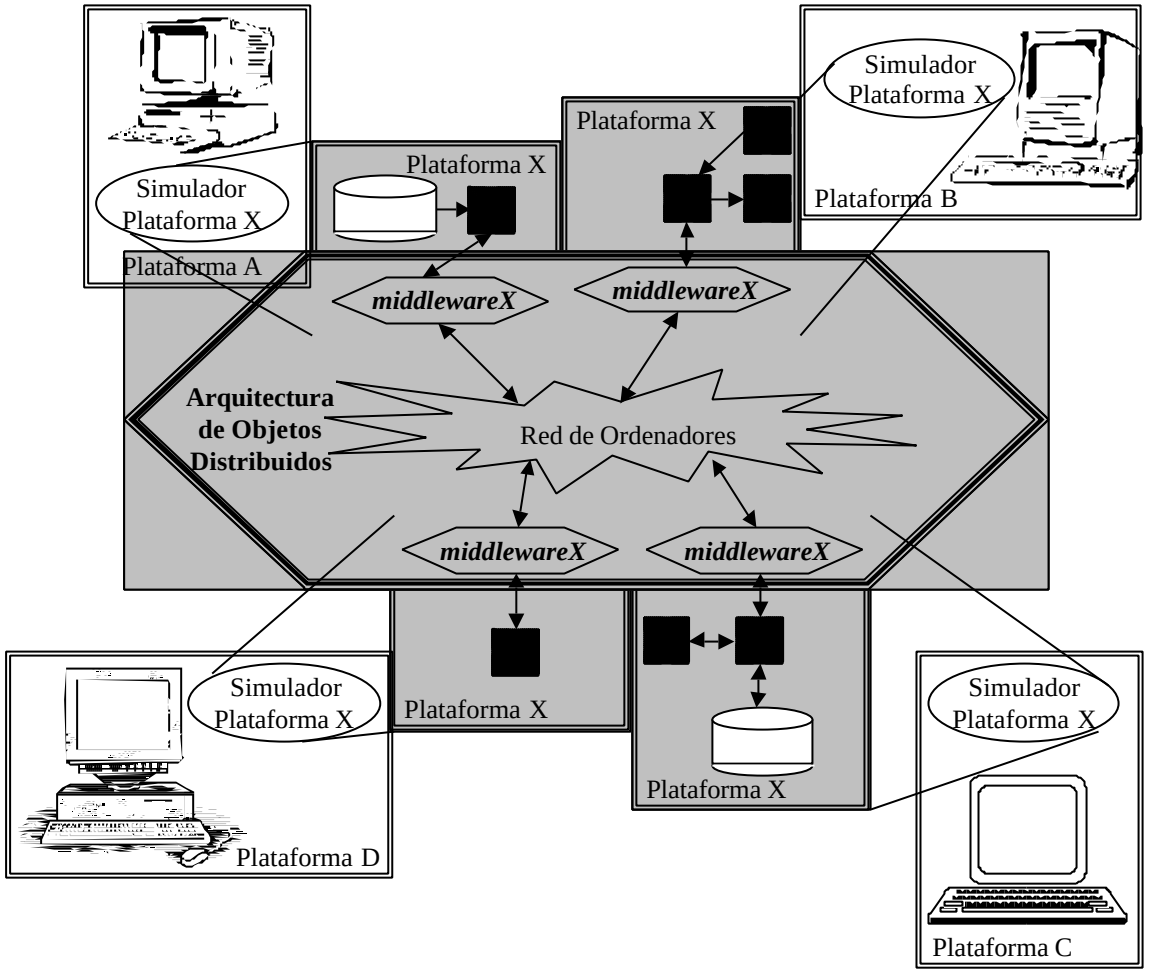


**Figura 15: Unión de una arquitectura y una plataforma independiente.**

Esta unión de conceptos es lo que busca Microsoft con la implementación de COM en diversas plataformas que hagan que ActiveX y DCOM formen un sistema de computación distribuido y móvil universal [Robinson97].

Actualmente se utiliza la potencia de la arquitectura abierta CORBA y la plataforma independiente de Java en el desarrollo de aplicaciones avanzadas que facilitan la posibilidad de distribuir su código por la red, gracias a la máquina virtual de Java, y se comunican de una forma estándar y totalmente abierta a otras plataformas, gracias a CORBA [Orfali98]. La unión de los dos conceptos en el ejemplo CORBA y Java, queda latente en la existente especificación de los componentes EJB sobre CORBA [Sun98c] (sobre COM todavía no han sido especificados).

# Capítulo 4:

# Sistema Integral Orientado a Objetos

En la actualidad, el paradigma de la orientación a objetos ha conseguido una alta distribución comercial en el campo de la informática, debido principalmente a las ventajas que presenta de cara a la reusabilidad, extensibilidad y robustez de las aplicaciones [Cueva98]. La reducción del tiempo de mantenimiento y desarrollo en el ciclo de vida de un sistema informático, reduce su coste total.

En contraposición al éxito de este paradigma, las plataformas tradicionales existentes no están diseñadas bajo este criterio[9] y por lo tanto no soportan bien las herramientas de desarrollo orientadas a objetos. Esta inadaptación o impedancia entre paradigmas hace que los sistemas pierdan eficiencia y no sean homogéneos al verse obligados a introducir una serie de capas que permitan adaptar los sistemas existentes a la orientación a objetos.

Actualmente se está trabajando en la inclusión de componentes distribuidos basados en la orientación a objetos sobre los sistemas informáticos existentes –así lo hemos visto en 3.3. Sin embargo, en la mayoría de los casos, la orientación a objetos se incluye de una manera parcial (sin todas las características de la misma). Ejemplos de esta incorporación parcial son aplicaciones implantadas sobre sistemas distribuidos orientados a objetos como CORBA [OMG95], que interactúan con el sistema operativo a nivel de funciones y tipos de datos básicos.

En la práctica tenemos sistemas que están basados completamente en la orientación a objetos como por ejemplo las bases de datos orientadas a objetos. Sin embargo, estos elementos se diseñan de manera independiente a otros módulos del sistema, con lo que no se aprovecha completamente la potencia de la orientación a objetos y es necesario establecer interfaces de adaptación de paradigmas entre los distintos módulos.

Un sistema integral orientado a objetos es por lo tanto, un entorno informático concebido en su totalidad con la filosofía de la orientación a objetos. Un sistema en el que desde la máquina base hasta la interfaz de usuario se integran de manera fluida al utilizar el mismo paradigma orientado a objetos, aprovechando totalmente las ventajas de éste.

El entorno propio de un sistema integral orientado a objetos, contempla un conjunto de ordenadores interconectados entre sí, todos ellos basados en el paradigma de la orientación a objetos y trabajando en un sistema operativo común. La programación en este sistema soportará el acceso homogéneo a las distintas máquinas y a sus recursos utilizando un mecanismo de seguridad integrado y acorde con el paradigma [Díaz98]. Una característica adicional, de interés en este tipo de sistemas, es su posibilidad de constituirse como un sistema abierto a otras arquitecturas comerciales existentes.

## 4.1 El Proyecto Oviedo3.

El proyecto de investigación Oviedo3 se basa en la planificación y desarrollo de un sistema integral orientado a objetos [Cueva96]. Las características que soporta son las mínimas

[9] Actualmente la plataforma de Java y sus procesadores software y hardware poseen un soporte a la orientación a objetos. Se está investigando en la implementación física de éstos aunque los microprocesadores Java [Sun97] no han gozado todavía de un gran éxito comercial.

requeridas a un sistema orientadas a objetos [Booch94] y otras definidas por las demandas del software existentes en este paradigma:

Abstracción e identidad de los objetos.

Encapsulamiento o encapsulación.

Herencia.

Polimorfismo.

Sistema de comprobación de tipos.

Concurrencia.

Persistencia.

Distribución y soporte a componentes.

El sistema está soportado por la especificación de una máquina abstracta orientada a objetos. Ésta constituye el núcleo del sistema, estableciendo ya el paradigma de la orientación a objetos desde la raíz del sistema. La especificación de la máquina se denomina Carbayonia y el lenguaje de ésta Carbayón [Izquierdo96]. La característica que mejor define a la máquina es que el concepto más básico es un objeto. Para manejar los objetos y alcanzar los requisitos inherentes al sistema, surgen otros conceptos como el paso de mensajes, la definición de métodos, la existencia de hilos, etc.

Una vez establecida en la máquina abstracta la base de tratamiento homogéneo de la información, se amplía ésta con las distintas capas del sistema sin necesidad de tener que modificar ya el paradigma utilizado. El sistema resultante obtiene la característica de portabilidad apoyándose en la implementación del procesador de la máquina abstracta en múltiples plataformas. Un esquema de las partes del sistema se aprecia en la Figura 16.

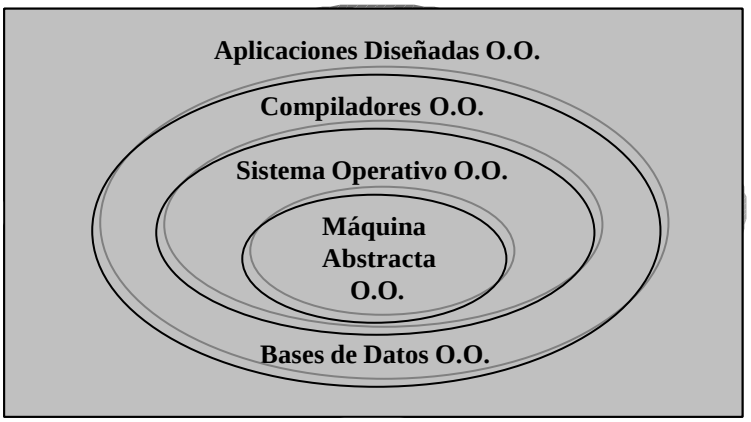


**Figura 16: Esquema del sistema integral orientado a objetos Oviedo3.**

Oviedo3 define su propio sistema operativo orientado a objetos, codificado en Carbayón, que gestiona las tareas básicas del sistema dando un soporte de nivel de abstracción mayor al resto de los módulos [Alvarez98]. Sobre éste se desarrollan compiladores y bases de datos orientados a objetos. Estos últimos se apoyan directamente en el sistema de persistencia [Ortín97, Ortín97b] y en la noción de objeto frente a tabla, cambiando la identificación de la información propia de las bases de datos relacionales [Martínez98].

La ventaja principal de esta arquitectura es el desarrollo rápido de aplicaciones informáticas con tecnologías orientadas a objetos. Con este soporte se desarrollan sistemas operativos, lenguajes, sistemas de gestión de bases de datos, así como sistemas CAD, todos ellos

orientados a objetos. Además esta arquitectura constituye un buen laboratorio de pruebas de robustez de sistemas software complejos y de altas prestaciones.

El diseño de aplicaciones en esta plataforma utiliza directamente tecnologías orientadas a objetos basándose en la reutilización de objetos distribuidos o componentes. Esta forma de programar se está extendiendo cada vez más por sus ventajas en la modulación y reutilización del código. La esencia de Oviedo3, así como sus características de distribución, persistencia, concurrencia, herencia, encapsulamiento y la comprobación de tipos, hacen que se convierta en un entorno de programación poseedor de las características más avanzadas demandadas actualmente y que hemos comentado en 3.3.

### 4.1.1 La Máquina Abstracta.

Siguiendo con la evolución de máquinas abstractas vista en 2.5, la máquina Carbayonia podemos identificarla dentro de las que considerábamos como plataformas independientes orientadas a objetos. Los objetivos de esta máquina son los propios de la clasificación identificada, pero el enfoque principal está más orientado a conseguir un sistema integral –más amplio que el propuesto con Smalltalk-80 (2.5.3.1) –que a la mera distribución y movilidad del código vista en 2.5.3.3.

La plataforma Java está pensada como una plataforma portable que ejecuta un código independiente. Las aplicaciones desarrolladas sobre ésta utilizan un nivel de abstracción alto poseyendo primitivas gráficas y de comunicaciones. Se supone un sistema operativo por debajo de la que proporcione estas primitivas[10]. Existe la especificación de un operativo para la plataforma Java (JavaOS [Madany96]) pero hace uso de la interfaz de llamadas a métodos nativos [Sun97c], perdiendo así su portabilidad. no ha gozado de una alta distribución comercial

La máquina abstracta de Oviedo3 no está pensada como una idea comercial para conseguir una plataforma de implantación de aplicaciones comerciales portables. Su principal objetivo es la creación de un sistema integral basado en el paradigma de la orientación a objetos [Cueva98].

### 4.1.2 Estructura de la Máquina Abstracta.

Carbayonia está compuesta fundamentalmente por cuatro áreas. Un área se podría considerar como una zona o bloque de memoria de un micro tradicional. En Carbayonia no se trabaja nunca con direcciones físicas de memoria, si no que cada área se puede considerar a su vez como un objeto el cual se encarga de la gestión de sus datos, y al que se le envía mensajes para que los cree o los libere. El direccionamiento de un objeto se realiza a través de una referencia.

[10] Las primitivas gráficas del AWT [Campione97] se establecen como llamadas, mediante código nativo de la plataforma existente [Sun97c], al sistema operativo.

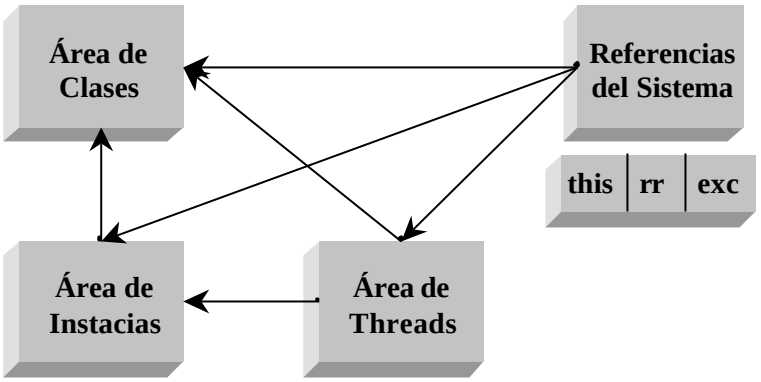


**Figura 17: Áreas de la máquina abstracta Carbayonia.**

A continuación se expone una breve descripción de las áreas que componen Carbayonia [Izquiedo96]:

**Área de Clases**: En este área se guarda la descripción de cada clase. Esta información está compuesta por los métodos que tiene, qué variables miembro la componen, de quien deriva, etc. Este área es fundamental para conseguir temas con la relación entre instancias y la persistencia de instancias.

Aquí ya puede observarse una primera diferencia con los micros tradicionales, y es que en Carbayonia realmente se guarda la descripción de los datos. Por ejemplo, en un programa en ensamblador del 80x86 hay una clara separación entre instrucciones y directivas de declaración de datos: las primeras serán ejecutadas por el micro mientras que las segundas no. En cambio Carbayonia ejecuta las declaraciones de las clases y va guardando esa descripción en éste área.

**Área de Instancias**: Es donde realmente se almacenan los objetos. Cuando se crea un objeto se deposita en éste área, y cuando éste se destruye se elimina de aquí. Se relaciona con el área de clases de manera que cada objeto puede acceder a la información de la clase a la que pertenece.

La forma única por la que se puede acceder a una instancia es mediante una referencia que posee como identificador el mismo que la instancia. La única forma de acceder a una instancia mediante una referencia es accediendo a los métodos del área de instancias.

**Referencias del Sistema**: Son una serie de referencias que están de manera permanente en Carbayonia y que tienen funciones específicas dentro el sistema[11]:

this: apunta al objeto con el que se invocó el método en ejecución.

exc: apunta al objeto que se lanza en una excepción.

rr (*return reference*): referencia donde los métodos dejan el valor de retorno.

**Área de *Threads***: En la ejecución de una aplicación se crea un hilo (*thread*) principal con un método inicial en la ejecución. Mediante las herramientas de concurrencia se podrán ir creando más *threads*, coordinarlos y finalizarlos. Cada *thread* posee una pila de contextos y el conjunto de *threads* existentes en una ejecución se almacenan en este área.

[11] En la versión persistente de la máquina abstracta [Ortin97], se añadió una referencia del sistema denominada *Persistence*. Esta referencia permite acceder a un objeto del sistema que nos facilita todas las primitivas del sistema de persistencia.

La simulación realizada se basa en la creación de objetos dentro del simulador que identifiquen a las distintas partes de la arquitectura de la máquina virtual. Sus métodos son los servicios que éstas partes ofrecen al resto de la máquina. Conforme se va interpretando el código fuente, se van modificando los valores de esta estructura.

### 4.1.3 Especificación del Lenguaje.

La especificación de la máquina abstracta Carbayonia pasa por identificar su lenguaje Carbayón. En la especificación de éste se describe la semántica de cada una de sus instrucciones [Izquierdo96] sobre la estructura de áreas mostrada con anterioridad.

La unidad básica de procesamiento es el objeto. Para definir objetos es necesario declarar previamente las clases a las que pertenecen. La Figura 18 muestra básicamente la estructura de la declaración de una clase.

```
Class Nombre de la Clase
Isa Clases Base
Aggregation Objetos Agregados
Association Referencias a Objetos Asociados
Methods Métodos de Invocación Síncrona
Messages Métodos de Invocación Asíncrona
EndClass
```

**Figura 18: Estructura del código Carbayón.**

Se puede identificar relaciones entre clases de general específico o herencia mediante la declaración *isa* (permitiendo la herencia múltiple). Se establecen relaciones todo/parte entre objetos mediante *aggregation* y enlaces entre abstracciones mediante la declaración *association*. Se pueden definir métodos y mensajes. La diferencia es que los primeros son invocados de forma síncrona y los segundos crean un nuevo hilo o *thread* de ejecución en cada invocación.

Los cuerpos de los métodos vienen definidos por una consecución de instrucciones primitivas entre las que se encuentran la creación y destrucción de objetos así como la invocación y pasos de mensajes [Alvarez98]. Existen además un conjunto de clases y objetos del sistema que podemos utilizar para manejar objetos enteros, cadenas de caracteres y vectores entre otros.

El código de la máquina abstracta tiene un formato en el que la unidad mínima de información es el *byte*[12]. Por sencillez a la hora de generar código para la máquina Carbayonia, se desarrolló una herramienta que incorporaba un entorno de desarrollo de traducción del código ensamblador a éstos *bytecodes*[13] [Izquierdo96].

[12] A este tipo de código se le conoce comúnmente como *bytecode*.

[13] Actualmente existen varias versiones del entorno de programación así como un ensamblador simple y portable, diseñado en C++, que opera por línea de comandos.

# Capítulo 5:

# Lenguajes Orientados a Objetos Basados en Prototipos

En este capítulo veremos brevemente qué son los lenguajes orientados a objetos basados en prototipos, qué modelo de objetos utilizan y cuales son sus ventajas frente a los que utilizan el concepto de clase.

Una posible clasificación de los lenguajes orientados a objetos divide a los lenguajes de programación en [Evins96]:

Lenguajes orientados a objetos basados en clases

Lenguajes orientados a objetos basados en prototipos.

En los lenguajes basados en clases todos los objetos deben pertenecer a una determinada clase (a un tipo) declarada previamente y estableciéndose así una relación de instanciación entre clases y objetos [Booch94]. De forma contraria, en los lenguajes basados en prototipos existe tan solo la entidad de objeto eliminando la existencia de las clases [Borning86].

## 5.1 Modelo de Objetos Basado en Prototipos

Existen determinados objetos denominados *traits* que identifican comportamiento [Lieberman86]. Otros objetos denominados *prototipos* poseen un estado determinado y un comportamiento heredado de los *traits* mediante *delegación* o herencia dinámica [Ungar87]. La creación de objetos a partir de una clase en los lenguajes basados en clases es sustituida por la copia o clonación de un objeto prototipo.

En la Figura 19 se representa en la izquierda, en UML [UML98], una instancia de una clase derivada en un lenguaje orientado a objetos basado en clase. Vemos cómo existen dos clases relacionadas mediante la herencia. Una instancia de la clase derivada tiene como estado la unión de los valores de los atributos de la clase a la que pertenece y de sus clases base.

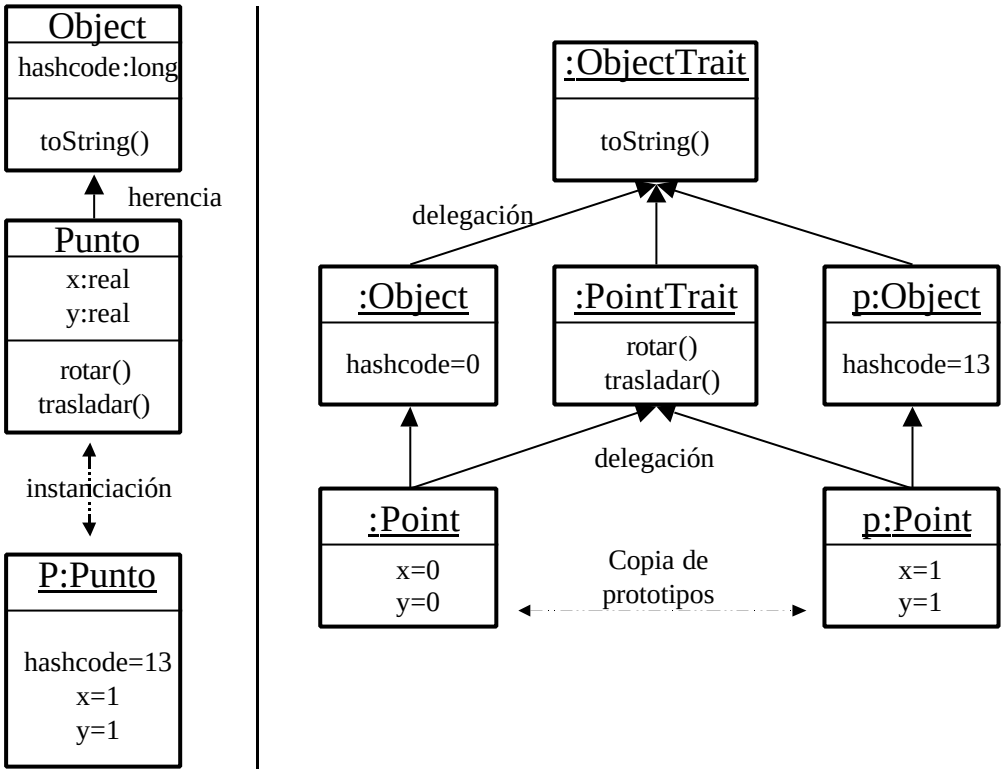


**Figura 19: Representación de una jerarquía de clases con una instancia en un lenguaje orientado a objetos basado en clases y en otro basado en prototipos.**

En la parte derecha de la Figura 19 vemos cómo representar esto en un lenguaje basado en prototipos. Tenemos un objeto *trait* que identifica la funcionalidad de cada abstracción. Este objeto delega el manejo de los valores de los atributos en otro objeto prototipo. El objeto prototipo se utiliza para realizar copias sustituyéndose así la creación de instancias de clases. Para crear un objeto de tipo punto, nos limitamos a copiar el prototipo de punto existente. Este objeto tiene una serie de enlaces dinámicos que le permitirá acceder a su comportamiento y a la parte propia de su abstracción como Object.

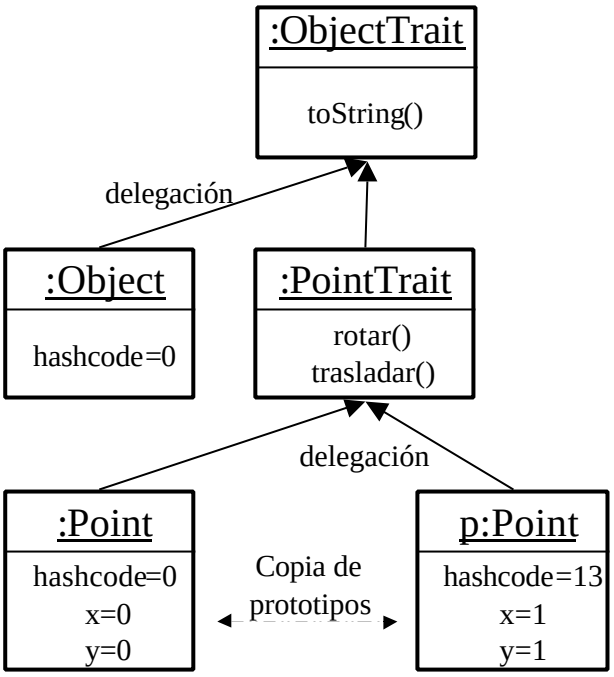


**Figura 20: Representación del modelo utilizando sólo herencia simple.**

La representación del modelo de la Figura 19 utiliza herencia múltiple. Es posible representar la misma información utilizando herencia simple ().

Existen varios lenguajes orientados a objetos basados en prototipos como el Self [Ungar87, Chambers89], ObjectLisp, Cecil [Chambers93], NewtonScript [Swaine94] y PROXY [Leavenworth93]. En estos lenguajes de programación se consiguen las características propias de un lenguaje orientado a objetos (abstracciones, encapsulamiento, herencia y tipos [Booch94]) sin la necesidad de que exista la noción de clase [Ungar91] y sin perder por ello capacidad de representación.

## 5.2 Utilización de los LOO Basados en Prototipos.

La principal diferencia del modelo de objetos propuesto por la copia de prototipos frente a la instanciación de clases es la simplicidad en los conceptos utilizados. No se pierde poder representación [Ungar91], se tiene un modelo más simple [Ungar87] y se identifican todas las características básicas de un lenguaje orientado a objetos.

Los lenguajes de programación basados en clases existentes que ofrecen la implantación de características avanzadas de la orientación a objetos [Cueva98] en el propio lenguaje facilitan sin duda la tarea del programador. Lenguajes como Java [Gosling96], C++ [Stroustrup98] o Eiffel [Meyer92] proporcionan útiles y elevadas abstracciones. ¿Por qué utilizar entonces un nivel conceptual más reducido?

La utilidad del modelo de objetos basados en prototipos viene por la abstracción de los conceptos básicos en un bajo nivel. Su utilización en especificación de máquinas que utilizan el paradigma de la orientación a objetos ha sido satisfactoria [Chambers91].

Un compilador de un lenguaje avanzado puede generar código al modelo de prototipos para posteriormente ser procesado (interpretado). La codificación de éste intérprete, ya sea física o lógica (2.1 y 2.2), se realizará de forma más simple que con otro modelo más complejo.

# Capítulo 6:

# Reflectividad Computacional

La programación de un ordenador viene impuesta por los requerimientos de sus usuarios. El entorno de computación de una oficina de Ingeniería Civil demandará aplicaciones relacionadas con el Diseño Asistido por Computación o CAD. Sin embargo, el ordenador de una secretaría tendrá instalado un alto número de aplicaciones de informática.

Cuando se trata de resolver un determinado problema para el que no existe ninguna aplicación que nos dé los resultados requeridos, se suele desarrollar una aplicación en uno o varios lenguajes de programación para obtener la solución. La elección del lenguaje de programación es una tarea influenciada también por el tipo de problema a resolver. Así existen lenguajes para la resolución de problemas como por ejemplo de líneas de espera (QNAP), acceso a bases de datos (SQL), problemas matemáticos (Matlab y Derive) o cálculos estadísticos (SPSS y BMDP). A estos lenguajes se les conoce como lenguajes de propósito específico u orientados a problemas concretos [Cueva92].

Otros lenguajes orientados a la resolución de cualquier tipo de problema son los conocidos como lenguajes de propósito general. Este tipo de lenguajes es el más difundido actualmente. La principal diferencia a la hora de resolver un problema específico (los comentados en el párrafo anterior) con estos lenguajes, es el tiempo y por lo tanto el coste de desarrollo. La utilización de un lenguaje orientado al problema acelera su desarrollo. La ventaja principal es el amplio abanico de problemas que pueden resolver.

Aún en el ámbito de los lenguajes de propósito general, la elección de uno u otro para la resolución de un problema viene influenciada también por el tipo de problema. Si buscamos una aplicación que sea portable, la elección del lenguaje Java [Gosling96] puede ser más acertada que el C++ [Stroustrup98]. Sin embargo, si el requisito crítico es la eficiencia en tiempo de ejecución, la mejor elección será la contraria.

Existen sistemas cuya naturaleza restringe la utilización de un tipo de lenguaje. Si necesitamos implantar un sistema de control en tiempo real así como si vamos a gestionar grandes volúmenes de información (como por ejemplo en banca), el tipo de lenguaje o entorno a utilizar serán completamente distintos.

En este capítulo introduciremos el concepto de reflectividad[14]. Los lenguajes y sistemas dotados de reflectividad, poseen una gran flexibilidad. Muchas de las facetas proporcionadas por la característica de la reflectividad aún no han sido exploradas así como otras forman ya parte de sistemas muy utilizados[15]. Desde el punto de vista de los lenguajes, lo utilizaremos como un mecanismo para la ampliación de éstos. Un mismo programa, con una determinada sintaxis, podrá representar distintos significados semánticos, apoyándose en la reflectividad.

## 6.1 Definiciones y Conceptos.

En esta sección definiremos y comentaremos una serie de conceptos que utilizaremos en el resto del documento.

Definimos ***reification*** o **cosificación** como la capacidad de acceder al estado de computación de una aplicación como si fuese un conjunto de datos [Smith82]. Cuando se produce

[14] Definiremos distintos tipos y clasificaciones de reflectividad.
[15] Ejemplos como la introspección de Java y los Meta-Datos de CORBA.

la cosificación, se "salta" a otro entorno de computación en el que nuestro anterior entorno es accesible[16].

La **reflectividad** o **reflexión** (*reflection)* es la propiedad de un sistema computacional que le permite razonar y actuar sobre él mismo así como ajustar su comportamiento en función de ciertas condiciones [Maes87]. El dominio computacional de un sistema reflectivo añade al dominio de un sistema convencional, la estructura y la computación de sí mismo.

Un sistema reflectivo es pues un sistema que es capaz de acceder, analizarse y modificarse a sí mismo como si de una serie de estructuras de datos se tratase.

El **sistema base** es aquel sistema de computación que es a su vez dominio de otro sistema de computación distinto, llamado meta-sistema [Golm97].

El **meta-sistema** es un sistema computacional que posee en su dominio de computación otro sistema, el sistema base.

Un meta-sistema está ejecutándose por lo tanto por debajo de un sistema base sin que éste lo vea. La forma de cambiar desde un sistema base al meta-sistema es mediante la cosificación. En la Figura 21 se aprecian todos esos conceptos en los dos entornos de computación.

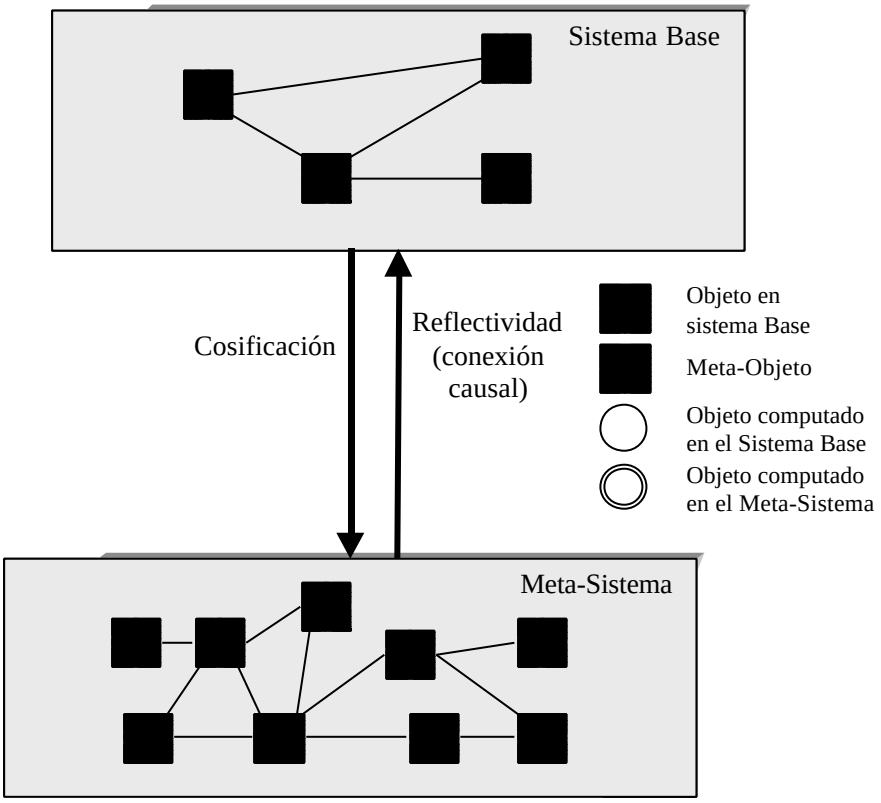


**Figura 21: Dos entornos de computación en un sistema reflectivo**

Un sistema es **causalmente conexo** si su dominio alberga el entorno computacional de otro sistema y, al modificar la representación de éste, los realizados causan un efecto en su computación [Maes87b].

Esta conexión se debe dar cuando el meta-sistema modifica los datos propios relacionados con el entorno del sistema base, para que se produzca la reflectividad. La forma en la que se produce la conexión causal, varía de unos sistemas a otros.

[16] De ahí el término cosificar: Transformar una representación, idea o pensamiento en cosa.

Si identificamos la orientación a objetos como paradigma a utilizar, podremos definir un **meta-objeto** como un objeto del meta-sistema que contiene información relativa al sistema base [Kiczales91]. La información a la que se refiere no es necesariamente otro objeto del sistema base sino cualquier parte de su entorno de computación.

## 6.2 Reflectividad Estructural. Introspección.

La clasificación más utilizada del concepto de reflectividad divide a ésta en dos tipos: reflectividad estructural (*structural reflection*) y reflectividad computacional (*computational reflection*) o de comportamiento (*behavioral reflection*) [Ferber89].

La reflectividad estructural: es la más obvia y actualmente la más conocida. Se refiere al acceso al estado estructural del sistema en tiempo de ejecución [Ferber88] tales como las clases, el árbol de herencia, las instancias y los tipos existentes en tiempo de ejecución. Uno de los ejemplos clásicos es la identificación dinámica de tipos (RTTI) del C++ [Stroustrup98].

En la reflectividad estructural se accede al estado de la ejecución de una aplicación en un determinado momento. Se puede conocer su estado, acceder a distintas partes del mismo y modificarlo si se estima oportuno. De esta manera, una vez reanudada la ejecución del sistema base, los resultados pueden ser distintos a los que se hubiesen obtenido si la modificación de su estado no se hubiera llevado acabo.

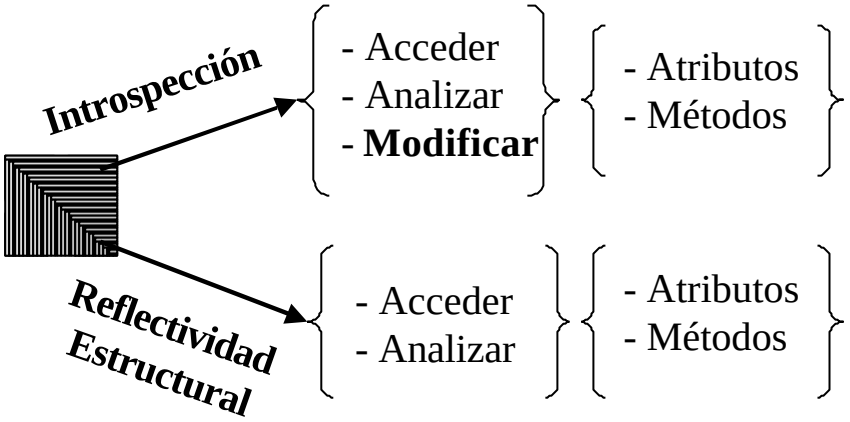


**Figura 22: Posibilidades de la introspección y de la reflectividad estructural.**

La introspección[17] es la capacidad para poder inspeccionar –pero no modificar –los objetos, o la estructura de los objetos, existentes en un entorno de computación en tiempo de ejecución. Podemos pensar en ver la introspección como una reflectividad estructural "de lectura", es decir un reflectividad estructural que no permite modificar la estructura de los objetos en tiempo de ejecución, sólo conocerla.

A continuación identificaremos casos prácticos y utilidades que se han dado a este par de conceptos.

### 6.2.1 Smalltalk-80.

Es uno de los entornos de programación que más se ha basado en la reflectividad estructural. Como ya hemos comentado en 2.5.3.1, el sistema Smalltalk-80 se puede dividir en dos elementos:

[17] El significado de la palabra introspección es "operación de observar uno sus propios actos o estados de ánimo o de conciencia". Vemos como se amolda muy bien a su semántica en el campo de la computación.

La imagen virtual que es una colección de objetos que proporcionan funcionalidades de diversos tipos.

La máquina virtual que interpreta de la imagen virtual.

En un principio se carga la imagen virtual en memoria y la máquina va interpretando el código. Si deseamos conocer la estructura de alguna de las clases existentes, una descripción, el conjunto de los métodos que posee o incluso la implementación de éstos, podemos utilizar el *browser* [Mevel87] (Figura 23).

El *browser* es una aplicación diseñada en Smalltalk que accede a todos los objetos clase[18] existentes en el sistema y muestra su estructura y descripción. De la misma forma se puede acceder a los métodos de éstas. Este es un caso propio de la introspección del sistema. Gracias a la introspección se consigue una documentación siempre actualizada de las clases del sistema.

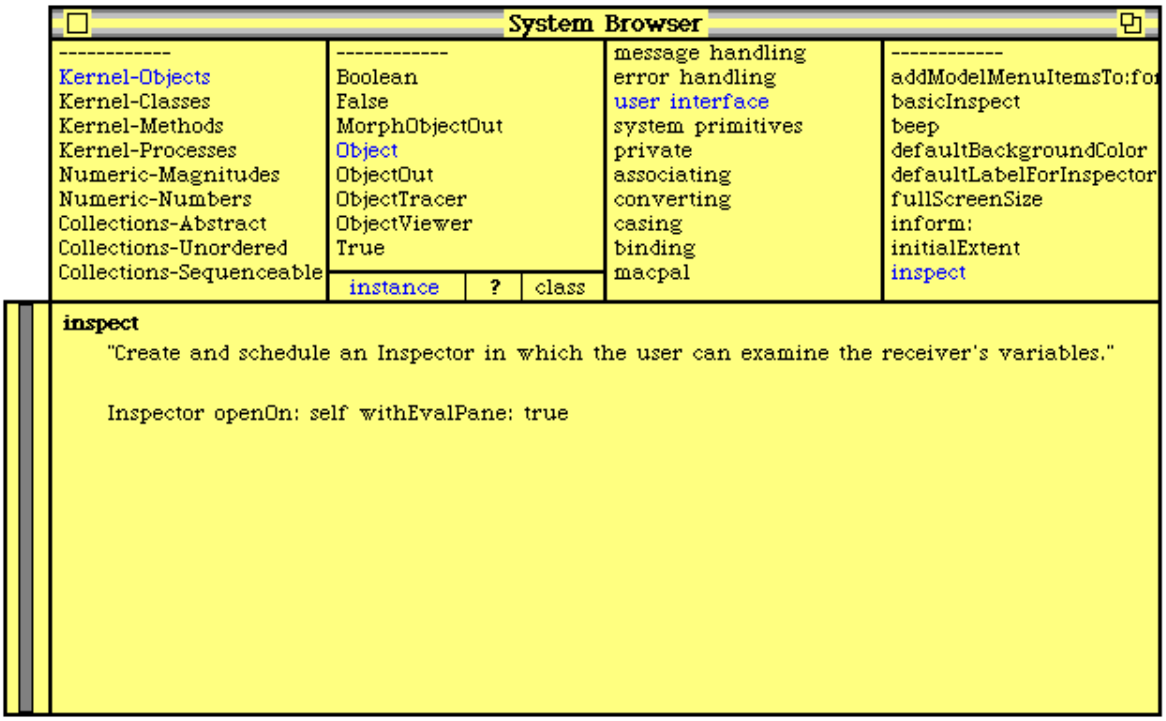


**Figura 23: Análisis del método *inspect* del objeto de clase *Object* con la aplicación *Browser* de Smalltalk-80**

De la misma forma que las clases, los objetos existentes en tiempo de ejecución pueden ser inspeccionados gracias al mensaje *inspect* [Goldberg89]. Podremos comprobar así como modificar por ejemplo, los valores de los distintos atributos de los objetos. Vemos en la Figura 24 cómo también es posible hacer uso de la reflectividad estructural para modificar la estructura de una aplicación en tiempo de ejecución. Una utilidad dada al mensaje *inspect* es la depuración de una aplicación sin necesidad de generar código intruso a la hora de compilar.

[18] En Smalltalk-80 todas las clases se identifican como un objeto en tiempo de ejecución. Los objetos que identifican clases son instancias de clases derivadas de la clase *Class*.

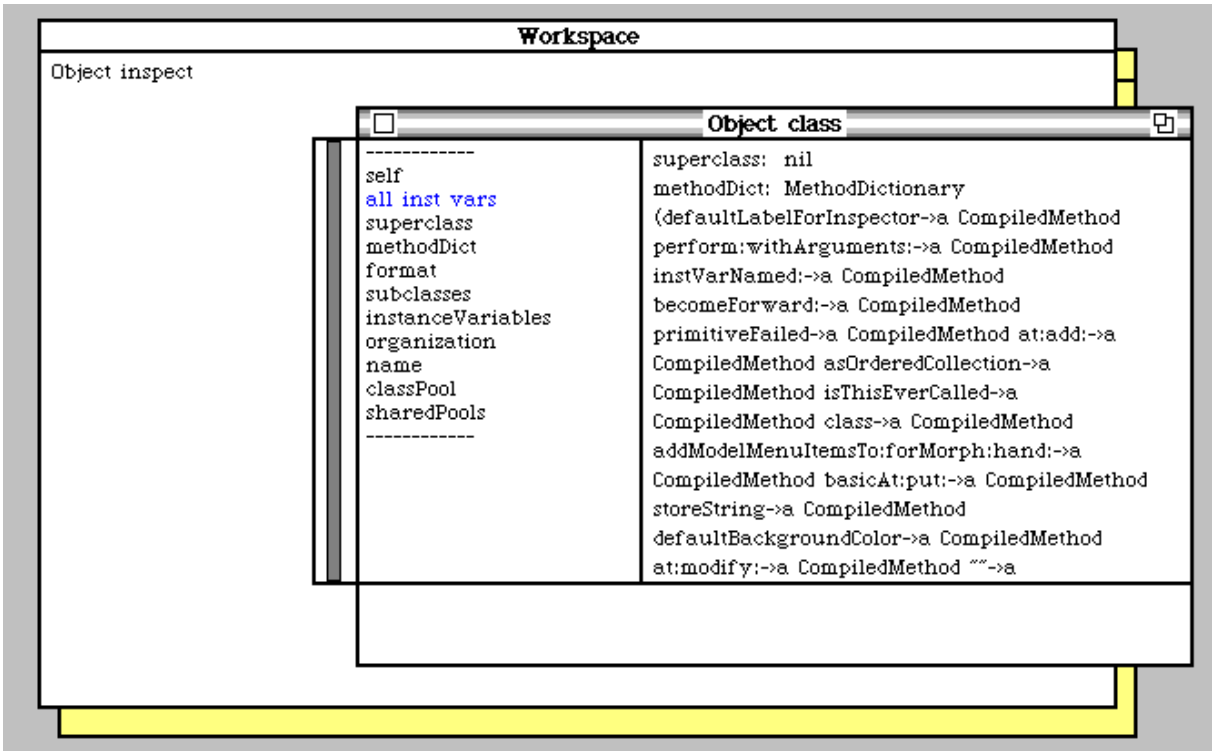


**Figura 24: Invocando al método *inspect* del objeto *Object* desde un espacio de trabajo de Smalltalk-80, obtenemos un acceso a las distintas partes de la instancia**

Lo buscado con el diseño de Smalltalk era obtener un sistema que fácilmente manejable por personas que no fuesen informáticos. Para ello se identificó el paradigma de la orientación a objetos y la reflectividad estructural. Una vez diseñada una aplicación consultando la información necesaria las clases en el *browser*, se pude depurar accediendo y modificando los estados de los objetos en tiempo de ejecución. Se accede al error de un método de una clase y se modifica su código, todo ello haciendo uso de la reflectividad estructural.

Vemos cómo en Smalltalk-80 se utilizaron de forma básica los conceptos de reflectividad estructural e introspección para obtener un entorno sencillo de manejar y autodocumental. Es por lo tanto una plataforma de desarrollo de prototipos ideal para probar utilidades de la reflectividad estructural.

### 6.2.2 Java y JavaBeans.

Java define para su plataforma una interfaz de desarrollo de aplicaciones (API) que proporciona reflectividad estructural restringida, denominada *The Java Reflection API* [Sun97d]. Este API representa en tiempo de ejecución las clases, interfaces y objetos que existen en la máquina virtual.

El API considera un conjunto "común" de operaciones a realizar mediante reflectividad estructural restringiendo así el número de operaciones posibles en el acceso al entorno de computación. Las posibilidades que ofrece son:

Determinar la clase de la que un objeto es instancia.

Obtener información sobre los modificadores de una clase, sus métodos, campos, constructores y clases base (o ancestras).

Encontrar las declaraciones de métodos y constantes pertenecientes a un *interface*.

Crear una instancia de una clase totalmente desconocida en tiempo de compilación.

Obtener y asignar el valor de un atributo de un objeto en tiempo de ejecución.

Invocar un método de un objeto en tiempo de ejecución, aun siendo éste desconocido en tiempo de compilación.

Crear un nuevo *array*[19] y modificar los valores de sus elementos.

La restricción de utilizar sólo este tipo de operaciones pasa por la imposibilidad de modificar el código de un método en tiempo de ejecución. La única tarea que se puede realizar con respecto a los métodos es su invocación dinámica. La inserción, modificación y borrado de métodos no es viable en el *Reflection API.*

Sobre este paquete de reflectividad estructural se define el paquete *java.beans* que ofrece un conjunto de clases e *interfaces* para conseguir introspección sobre una especificación de componentes software: los *JavaBeans* [Sun96]. La implementación de este paquete se basa en la utilización del API de reflectividad puesto que, como ya hemos especificado, la introspección es reflectividad estructural "de lectura".

Cabe preguntarse por qué es necesaria la introspección para el desarrollo de los componentes. Un componente está constituido de métodos, propiedades y eventos. Para conocer el conjunto de estas tres partes de forma flexible, se utiliza la introspección: podemos acceder a los métodos de instancia públicos y conocer su signatura.

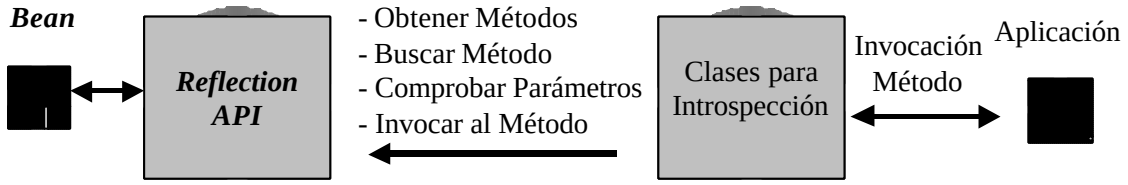


**Figura 25: Implantación de las clases de introspección sobre el API de reflectividad de Java.**

El paquete de introspección identifica una forma segura de acceder al API de reflectividad y facilita el conocimiento y acceso de métodos, propiedades y eventos de un componente en tiempo de ejecución. En la Figura 25 se aprecia cómo una aplicación solicita el valor de una propiedad de un Bean, el paquete de introspección, apoyándose en el de reflectividad, invoca al método apropiado y devuelve el valor de la ejecución.

Como apéndice de este documento se ha desarrollado un ejemplo del uso de reflectividad en la máquina de Java [Sun97d]. Por teclado se pide al usuario el nombre de una clase demostrando así que no es conocida en tiempo de compilación. Se visualiza toda su información (introspección), se crea una instancia, se muestra los valores de los objetos y se invoca a un método de éste (reflectividad estructural). El paquete de introspección para el desarrollo de aplicaciones ofrece estas primitivas a un mayor nivel de abstracción.

### 6.2.3 CORBA.

Para una arquitectura de objetos distribuidos, el concepto de introspección en clave a la hora de hacer aplicaciones portables y flexibles [OMG95].

Como comentamos en 3.3.1, CORBA posee una forma de realizar invocaciones dinámicas a métodos de objetos remotos. Esto pasa por conectarse al ORB, solicitar la descripción de un objeto, analizar su interfaz e invocar posteriormente al método adecuado.

CORBA proporciona metadatos de los objetos existentes en tiempo de ejecución guardados en el almacén de interfaces (*Interface Repository*), que proporcionarán la información nece-

[19] En Java los *arrays* son objetos primitivos.

saria a cerca de los objetos servidores. El módulo de invocación dinámica (DII) se apoya en estas descripciones de los objetos.

El acceso dinámico a métodos de objetos distribuidos tiene una serie de ventajas.

No es necesaria la existencia de *stubs* para el cliente, el acceso se determina en tiempo de ejecución.

Para compilar las llamadas a los clientes no es necesario haber implementado el servidor.

En las modificaciones del servidor no es necesario recompilar la especificación IDL para el cliente.

La depuración del cliente es más sencilla puesto que podemos ver el aspecto del servidor desde la plataforma cliente.

Los mensajes de error no producen una parada en la ejecución sino que pueden ser controlados desde la aplicación cliente.

El control de versiones de objetos servidores puede ser llevado a cabo mediante el descubrimiento de nuevos objetos (coexistiendo así varias versiones).

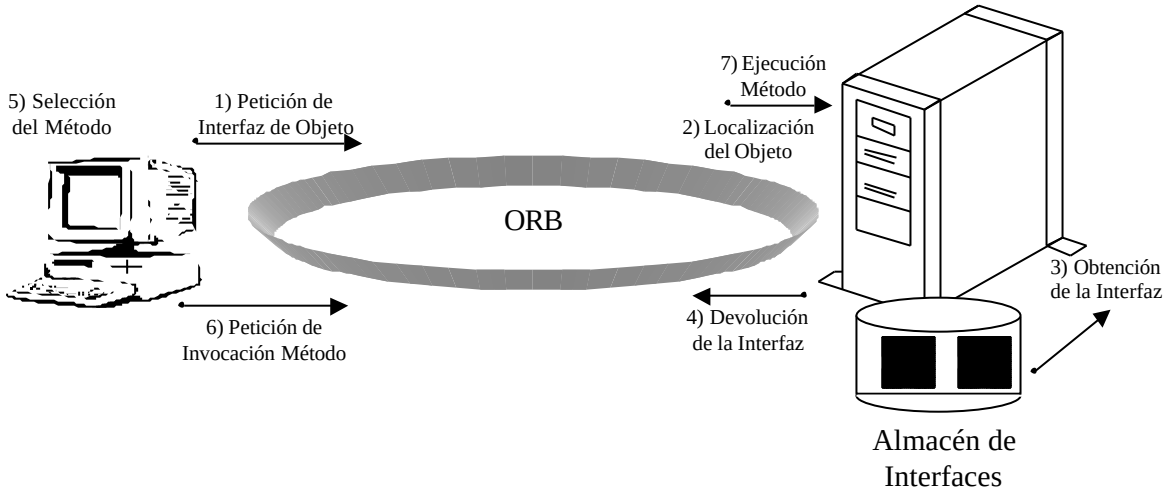


**Figura 26: Secuencia de una invocación dinámica en CORBA, apoyándose en los metadatos**

En la Figura 26 se aprecia cómo un objeto cliente se conecta a un ORB y solicita la interfaz de un objeto. El objeto servidor es localizado y, accediendo al almacén de interfaces, se devuelve su interfaz al cliente. El cliente busca el método en el que está interesado y envía un mensaje al objeto servidor.

En la misma figura se puede suponer el caso negativo a la hora de encontrar el objeto o el método solicitado. El cliente puede estar programado para indicar una posible desconexión del objeto remoto y seguir operando con cualquier otro objeto.

## 6.3 Reflectividad Computacional.

Se puede definir la reflectividad computacional[20] como el comportamiento exhibido por un sistema computacional que se puede computar a sí mismo apoyándose en un mecanismo de conexión de causalidad [Maes87b]. Esto quiere decir que la reflectividad computacional es el mecanismo para cosificar la computación de un sistema y por lo tanto su comportamiento.

[20] Este concepto aparece en bibliografías como *computational, procedural* o *behavioral reflection*.

Un lenguaje de programación posee unas especificaciones léxicas [Cueva93], sintácticas [Cueva95] y semánticas [Cueva95b]. Un programa correcto es aquél que cumple las tres especificaciones descritas. La semántica (distinto del análisis semántico) del lenguaje de programación identifica el significado de cualquier programa codificado en este lenguaje, es decir cuáles serán sus repercusiones en tiempo de ejecución. La semántica del uso del operador *new* del lenguaje C++ [Stroustrup98], es la petición de reserva de zonas de memoria *heap* para gestionar información.

La gran diferencia entre la reflectividad estructural y la computacional es que la primera refleja[21] en tiempo de ejecución el entorno o contexto de computación de un sistema mientras que la segunda refleja la semántica o comportamiento del sistema base (no simplemente los datos).

Si nos fijamos en un sistema de computación básico y físico –el caso de un microprocesador –mediante la reflectividad computacional podríamos modificar el significado de las instrucciones máquina, pudiendo cambiar así la unidad de control, accesos a memoria o incluso la forma de operar de la unidad aritmética y lógica. Así un programa con una determinada sintaxis carece de un significado si lo podemos modificar mediante reflectividad computacional.

Como ejemplo de reflectividad computacional se ha desarrollado un prototipo de evaluador de expresiones aritméticas computacionalmente reflectivo (apéndice B). En este ejemplo se muestra como una operación semántica del lenguaje como la división puede cambiarse de devolver números racionales a devolver números enteros. Otra modificación mostrada en el apéndice B, es la visualización de las secuencias de evaluaciones realizadas para expresiones complejas.

### 6.3.1 La Torre de Intérpretes.

Para entender e implementar prototipos de demostración de uso de reflectividad computacional, Smith identificó lo que en reflectividad se conoce como la torre de intérpretes [Smith82]. Desarrolló un entorno de trabajo (*framework*) en el que cada intérprete de un lenguaje estaba siendo interpretado a su vez por otro intérprete.

El programa de usuario se ejecuta en un nivel que podemos llamar nivel 0. Este programa es interpretado por un intérprete que es ejecutado en el nivel 1. El segundo intérprete es a su vez interpretado por otro más que se ejecuta en un nivel superior: el 2. De esta forma, podemos identificar la computación de un sistema como una torre de intérpretes.

En la realidad existe un nivel máximo existente que es el que ejecuta un intérprete de forma física. Son intérpretes de bajo nivel muy eficientes e implementados en *hardware*: Los microprocesadores. Un microprocesador es un procesador físico de su lenguaje binario (2.1).

[21] El sistema base es reflejado en el meta-sistema.

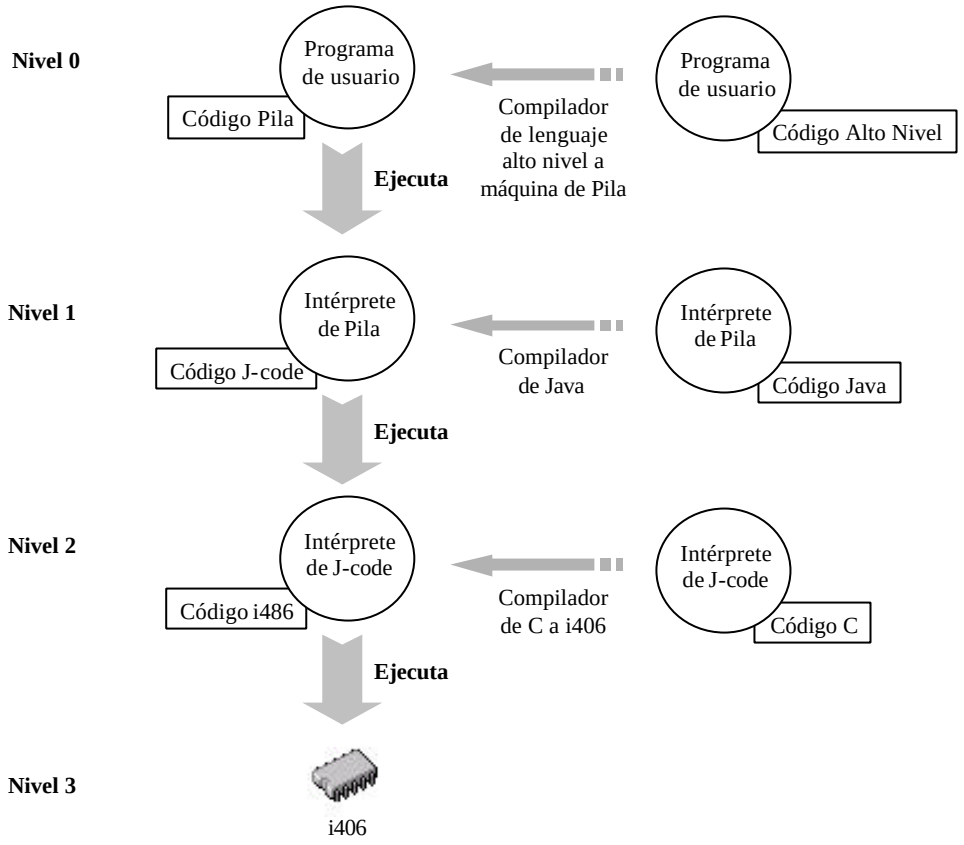


**Figura 27: Ejemplo real de una torre de intérpretes de 4 niveles.**

Vemos en la Figura 27 un ejemplo real de una posible torre de intérpretes[22]. Se trata de implementar un intérprete de un lenguaje intermedio de pila. Este código intermedio habrá sido generado por un compilador y ahora es interpretado. El lenguaje de implementación elegido para desarrollar el intérprete es Java.

El Java es un lenguaje que se compila a un código binario de una máquina abstracta [Lindholm96]. Para interpretar este código podemos utilizar un emulador de la máquina programado en C y compilado para un i486. Finalmente el código binario de este micro es interpretado por el propio procesador, de forma física.

En el ejemplo propuesto tenemos un total de cuatro niveles en la torre de intérpretes.

Si pensamos en el comportamiento o la semántica de un programa como una función de nivel n, ésta podrá ser vista realmente como una estructura de datos desde el nivel n+1. Si el lenguaje de implementación de la función fuese Lisp [Steele90], la representación de ésta en el nivel n+1 sería una lista. De esta forma, seremos capaces de acceder al estado de un sistema en tiempo de ejecución (reflectividad estructural) a la interpretación o semántica del programa (reflectividad computacional o de comportamiento).

Cada vez que en la torre de intérpretes nos movamos en un sentido de niveles ascendente, podemos identificar esta operación como cosificación[23] (*reification*). Un movimiento en el sentido contrario se identificará como reflexión (*reflection*) [Smith82].

[22] Caso real dado como ejercicio de la asignatura de Procesadores de Lenguajes.
[23] Ciertamente hacemos datos (o cosa) un comportamiento.

### 6.3.2 Formalización.

Las bases de la computación han sido formalizadas mediante matemáticas, basándose principalmente en λ-*calculus* [Barendregt81] y extensiones de éste. Un ejemplo es la formalización realizada por Mendhekar en la que define una pequeña lógica para lenguajes funcionales reflectivos [Mendhekar96].

Mendhekar propone un cálculo reducido de computación, denominado $l_v - calculus$, y define sobre él una lógica ecuacional. Se aprecia cómo es realmente difícil formalizar un sistema reflectivo y las limitadas ventajas que proporciona. La investigación acerca de reflectividad computacional, tiene una rama relativa a su formalización y estudio matemático pero su complejidad hace que vaya por detrás de las implementaciones y estudios de prototipos computacionales.

### 6.3.3 Intérpretes Meta-Circulares.

En 6.3.1 hemos definido una torre de intérpretes para comprender el concepto de reflectividad computacional. En estas torres de intérpretes el nivel n está constituido por un intérprete codificado en un lenguaje **I** que interpreta un lenguaje **B**.

Cuando tenemos una torre infinita de intérpretes y el lenguaje **I** y el lenguaje **B** son siempre el mismo (en todos los niveles de la torre), se puede decir que estamos ante un intérprete meta-circular. La forma en la que se accede desde el meta-sistema al sistema base no varía así el lenguaje y se produce simplemente la identificación de lo que antes era computación como datos: cosificación.

El desarrollo de prototipos de intérpretes meta-circulares ha pasado por la implementación de lenguajes sencillos y basados en reflectividad estructural. Implementar la meta-circularidad para sistemas computacionalmente reflectivos implica una elevada complejidad y, como veremos, no aportaría ventajas significativas frente a un sistema de dos niveles.

Uno de los prototipos más conocidos de intérpretes meta-circulares basados en reflectividad estructural es el desarrollado para Lisp y conocido como 3-LISP [Wand88]. El estado computacional que será reflejado en este lenguaje tiene tres partes:

Entorno (*environment*): Identifica el enlace entre identificadores y sus valores en tiempo de ejecución.

Continuación (*continuation*): Define el contexto de control. Recibe el valor devuelto de una función y lo sitúa en la expresión que se está evaluando, en la posición en la que aparece la llamada a la función ya terminada.

Almacén (*store*): Describe el estado global de la computación en el que se incluyen contextos de ejecución e información sobre los sistemas de entrada y salida.

De esta forma, el estado de computación de un intérprete queda definido por tres valores –denominados (e, r, k) –que podrán ser accedidos desde el meta-sistema como un conjunto de tres datos.

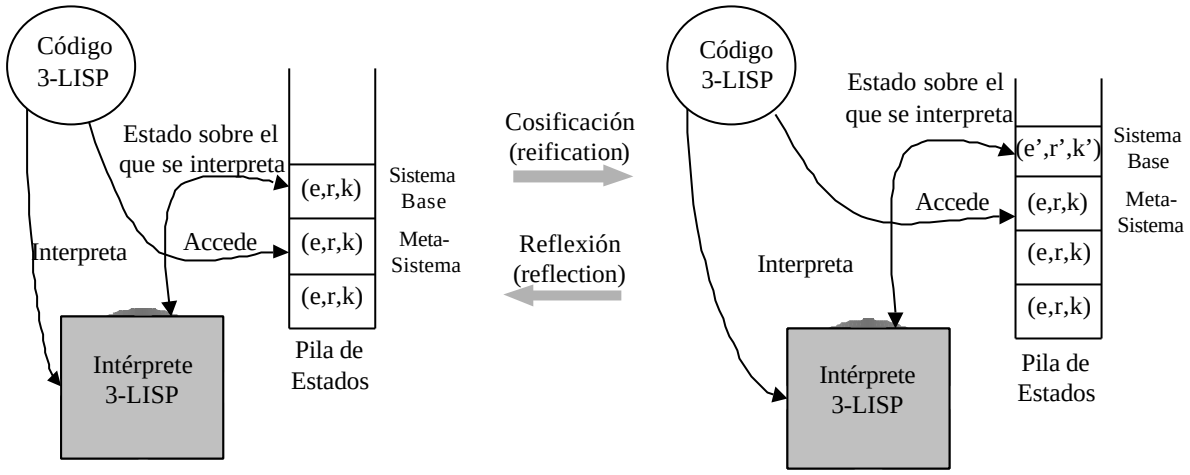


**Figura 28: Ejemplo de implementación de un intérprete meta-circular de 3-LISP.**

En la Figura 28 se aprecia cómo se desarrollo el prototipo de 3-LISP. Existe un programa en 3-LISP que posibilita el cambio de nivel con las operaciones *reify* y *reflect*. Un intérprete del único lenguaje existente va leyendo e interpretando el lenguaje fuente. La interpretación pasa por la lectura y modificación de los tres valores (e, r, k). En la interpretación se puede producir un cambio de nivel de computación:

Cosificación (*reify*): Se apila el valor del estado computacional (e, r, k). Se crea entonces un nuevo estado de computación (e', r', k'). Ahora el intérprete trabaja sobre este contexto y puede modificar los tres valores del nivel anterior como si de datos se tratasen.

Reflexión (*reflect*): Se desapila el contexto actual volviendo al estado anterior existente en la pila.

La posibilidad de implementación de este prototipo pasa por la existencia de un único lenguaje. Así se puede separar un único intérprete de los múltiples estados de computación sobre los que éste trabaja.

Una vez comentado el funcionamiento base de este prototipo, cabe preguntarnos sus ventajas e innovaciones. Sin duda una ventaja es el tratamiento uniforme del sistema al identificar un único lenguaje. En los pasos de cosificación no será necesario pues referirse al estado de computación con otra sintaxis distinta a la utilizada en el nivel actual.

Una pregunta inevitable pasa por preguntarnos la utilidad de la torre infinita. El hecho de que existan tantos niveles aporta al sistema una flexibilidad indudable. Por otro lado esta flexibilidad tan amplia dotada por los infinitos niveles hace que el sistema sea difícil de utilizar y altamente inseguro puesto que se puede modificar todo y obtener resultados imprevistos [Foote90].

Los intereses prácticos pasan por un número fijo de niveles en la torre de intérpretes. Estableciendo este número máximo se puede fijar un mecanismo de seguridad que se apoye en un conjunto de primitivas –hablando en términos de reflectividad computacional –no modificables. Así se obtendrá un sistema seguro, flexible y optimizable[24].

[24] La reflectividad computacional mediante el modelo de la torre de intérpretes registra tiempos elevados de ejecución por las múltiples interpretaciones realizadas.

## 6.4 Evolución en Niveles de Reflectividad.

Existen diversas clasificaciones del concepto de reflectividad. Golm identificó la siguiente clasificación [Golm97]:

Reflectividad en tiempo de compilación, enlace (*link*) y ejecución. Se identifica el tiempo en el que se realiza el enlace entre el sistema base y el meta-sistema. Esta conexión puede identificarse en tiempo de compilación, en tiempo de enlace o carga y en tiempo de ejecución.

Reflectividad estructural y computacional o de comportamiento. Esta clasificación se realiza en función de la información del sistema base que puede obtenerse desde el meta-sistema De esta forma en la estructural podemos acceder al estado de computación del sistema base y en la computación accedemos a su semántica o evaluación de instrucciones.

Reflectividad explícita e implícita. Divide el tipo de reflectividad en función de si se especifica o la transición en el sistema base. Si el "salto" de niveles es visible para el sistema base, la reflectividad es explícita. Si no lo es, estamos ante reflectividad implícita.

En los apartados 6.2 y 6.3 hemos ahondado en la segunda clasificación y hemos identificado ejemplos existentes. En este apartado mostraremos una serie de sistemas existentes basándonos en la capacidad que tienen a la hora de acceder a información del sistema base. Los sistemas aparecerán ordenados de menor a mayor información accesible.

En la identificación de los sistemas se hará también referencia al momento en el que se produce el enlace entre el sistema base y meta-sistema (primera clasificación).

### 6.4.1 Sistemas con Introspección.

Definimos introspección como la capacidad de acceder a la representación de un sistema en tiempo de ejecución. La introspección permite el acceso sólo en lectura, sin permitir así la modificación de los datos y sin producirse la reflexión del sistema por la conexión causal (ver apartado 6.1).

La introspección es probablemente el tipo de reflectividad más utilizado actualmente en los sistemas comerciales. Se ha desarrollado en áreas como la especificación de componentes, arquitecturas de objetos distribuidas y programación orientada a objetos.

En lo referente a los componentes, hemos comentado en 6.2.2 cómo para los JavaBeans se diseñan clases como `java.beans.Introspector` que se apoyan directamente en el API de reflectividad de Java [Sun97d]. Las clases proporcionan una abstracción de alto nivel para que la introspección sea utilizada para el desarrollo de componentes. Así, cualquier objeto recuperado de disco, puede analizarse y obtener sus propiedades, métodos y eventos.

En lo referente a COM, el modelo de componentes de Microsoft (*Component Object Model*), la introspección también está presente [Microsoft95]. El acceso a los componentes se realiza, al igual que en CORBA, mediante el concepto de interfaz (*interface*). Todos los componentes COM han de implementar la interfaz `Iunknown`.

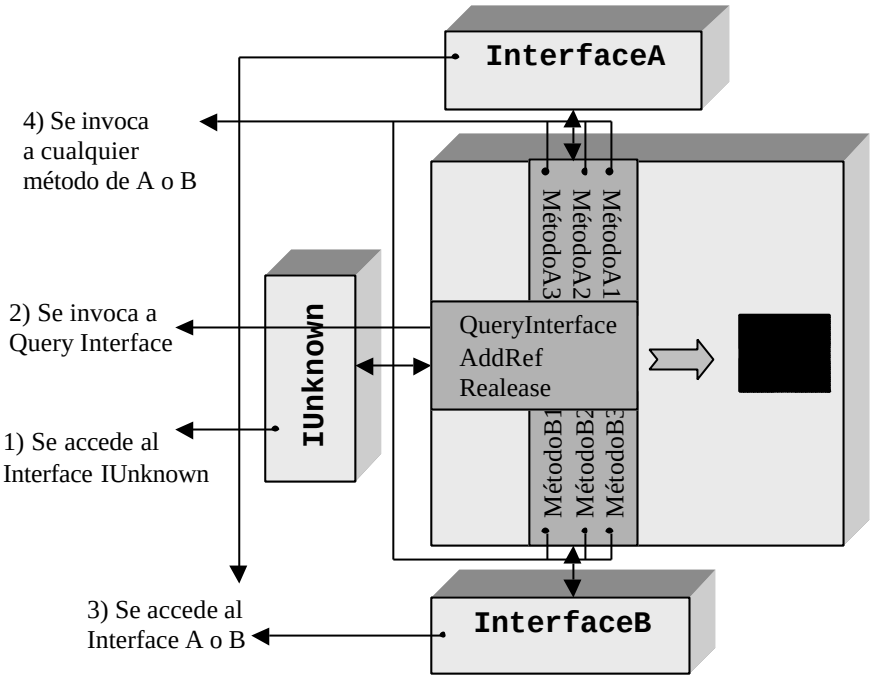


**Figura 29: Ejemplo de uso de Introspección en COM.**

Un objeto puede implementar cualquier interfaz pero, ¿cómo saber cuál es en tiempo de ejecución? Haciendo uso de la introspección. Como se observa en la Figura 29, el componente implementar dos interfaces A y B. Cada uno tiene sus métodos. Para acceder a éstos, se invoca al método `QueryInterface` y se obtienen las interfaces A y B.

Gracias al proceso de introspección, se diseñan modelos de componentes sobre determinadas plataformas. En el caso de Java basándonos en el API de reflectividad estructural que decíamos limitada y en Microsoft con estructuras adicionales creadas para todo componente.

El campo de las arquitecturas de objetos distribuidos, introducido en el capítulo 3, también se hace uso del concepto de introspección. En el caso de Microsoft DCOM amplia el modelo COM mediante un estándar de comunicaciones. La introspección de los objetos distribuidos se basa en la propia de los componentes [Brown98].

CORBA se basa en el concepto de introspección para llevar a cabo el descubrimiento de los objetos distribuidos existentes en la red. Almacena en un repositorio las especificaciones de las interfaces de los objetos servidores. Contra esta base de datos se pueden realizar peticiones y acceder de forma dinámica a estos objetos mediante DII (*Dynamic Interface Invocation*).

El resultado es la posibilidad de diseñar aplicaciones flexibles, dinámicas y fácilmente mantenibles (las ventajas de la introspección en CORBA están enumeradas en el apartado 6.2.3).

Finalmente, el ejemplo más tímido de utilización de introspección es la programación. Existe en varios paradigmas pero vamos a centrarnos en la orientación a objetos. Cuando tenemos una referencia de una clase base, puede hacer referencia a un objeto de una clase derivada ¿cómo podemos saber qué tipo de objeto tiene la referencia[25]?

Este mecanismo de identificación de tipos en tiempo de ejecución ha sido adoptado por diversos lenguajes de programación orientados a objetos. El C++ proporciona el `dyna-`

[25] Suponemos varias clases derivadas. Si la clase base no es abstracta, la instancia puede ser de cualquiera de las clases derivadas o de la propia clase base.

`mic_cast` de su RTTI (*Runtime Type Identification*) [Stroustrup98], el Java proporciona los ahormados descendentes que son compilados a la instrucción máquina `checkcast` [Lindholm96] y el ObjectPascal identifica el operador `as` [Charte96].

### 6.4.2 Sistemas con Reflectividad Estructural.

La principal característica de los sistemas mencionados en el punto anterior es que adoptan la introspección como algo adicional al entorno existente. Exceptuando Java, implementan la introspección mediante añadidos al sistema. La forma en la que se consigue la introspección en cada uno de ellos puede resumirse de la siguiente manera:

| Sistema | Consigue la Introspección |
|---|---|
| Java | Mediante el API de reflectividad |
| COM | Mediante la implementación obligada de una interfaz predefinida |
| CORBA | Mediante el almacenamiento de las interfaces (los archivos IDL) |
| Programación OO | Mediante el añadido de información en tiempo de ejecución. |

Vemos como exceptuando el caso de Java la forma de conseguir la introspección es con programación o restricciones adicionales haciendo que el sistema pierda en flexibilidad.

Smalltalk-80 consigue mediante la especificación de una máquina abstracta y la interpretación de su código binario un sistema basado en la reflectividad computacional [Goldberg83]. La reflectividad que se consigue con Smalltalk tiene más funcionalidad y posibilidades que con Java. Por el contrario es menos segura.

Ejemplos típicos del uso de la reflectividad estructural de Smalltalk, que hemos comentado en 6.2.1, son:

La aplicación *browser* que accede a todas las clases del sistema, describiendo su funcionamiento, su implementación y la de cada uno de sus métodos.

El mensaje *inspect* de todo objeto. Realiza una visualización del estado de cualquier objeto permitiendo su modificación.

Estos son meros ejemplos de las posibilidades de la reflectividad estructural. Smalltalk es un sistema cuyas aplicaciones están implementadas sobre él mismo, siendo todo reflectivo, flexible y modificable [Mevel87]. Es una plataforma ideal para el desarrollo de prototipos basados en reflectividad estructural.

La semántica del lenguaje viene dada por una serie de primitivas de la máquina inaccesibles para el programador. Carece por lo tanto de reflectividad computacional.

### 6.4.3 Reflectividad Computacional.

Actualmente existen una serie de prototipos más o menos desarrollados que identifican propiedades basadas en reflectividad computacional. Existen similitudes, diferencias y limitaciones entre ellos que comentaremos en este apartado.

La mayoría de los sistemas existentes se basan en la especificación de un protocolo de comunicación entre el sistema base y el meta-sistema. Las operaciones computacionales que se pueden modificar y la forma en la que pueden realizarse quedan reflejadas en el protocolo. A este tipo de protocolos se les ha denominado “Protocolos de Meta-Objetos” MOPs (*Metaobject Protocol*) [Kiczales91].

Un meta-objeto es un objeto del meta-sistema que identifica información sobre el sistema base. No necesariamente ha de ser información de un objeto; puede tratarse de métodos, atributos, clases, etc.

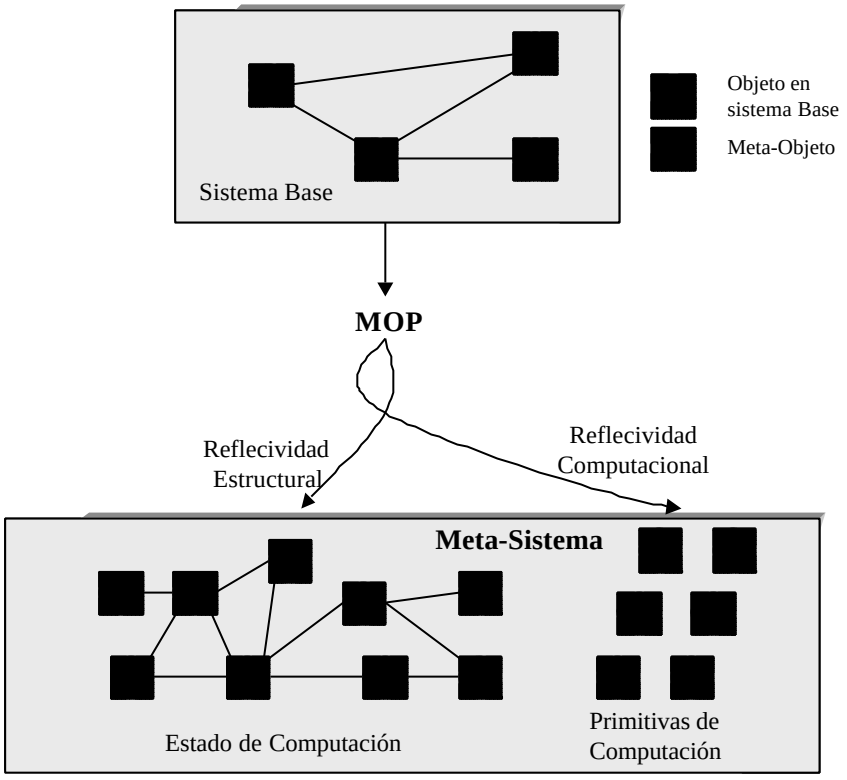


**Figura 30: Reflectividad computacional limitada a unas primitivas mediante el uso de MOPs.**

Los objetos del sistema base pueden modificar su semántica computacional interactuando mediante un protocolo con el meta-sistema. Como se puede apreciar en la Figura 30, los objetos utilizan el protocolo para acceder a los meta-objetos y a las primitivas de computación y poder modificar así su comportamiento y estructura.

Así todos los sistemas usan un protocolo y unas primitivas de computación modificables. Las diferencias residen en el momento y la forma en que se dan.

#### 6.4.3.1 Reflectividad en Tiempo de Compilación.

Una primera identificación de lenguajes agrupa a aquellos que realizan el enlace entre el sistema base y meta-sistema en tiempo de compilación (*Compile-Time Reflective Systems*). Ejemplos de lenguajes de este tipo son OpenC++ [Chiba95], Iguana [Gowing96] y Cognac [Murata95].

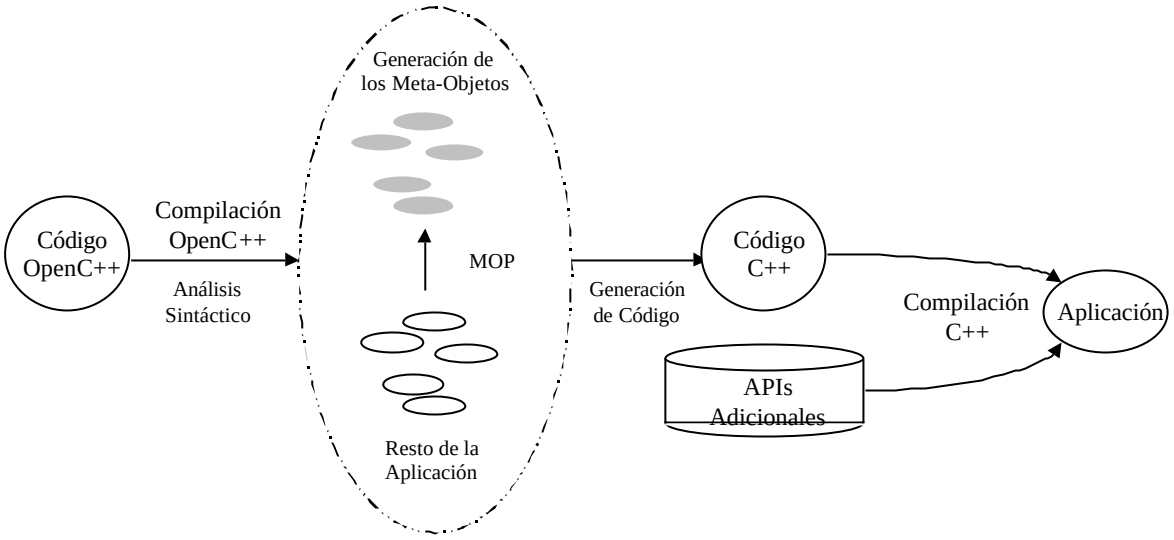


**Figura 31: Reflectividad computacional en tiempo de compilación con OpenC++.**

En el caso del OpenC++ el compilador generará código C++ que será a su vez recompilado. En la primera traducción, existen una serie de primitivas en tiempo de compilación que son llamadas al crearse el árbol sintáctico [Cueva95]. Podemos generar el código C++ que deseemos modificando así la semántica del código fuente. El código generado puede apoyarse en otras librerías adicionales como por ejemplo la implementación de un sistema de persistencia.

El código generado es muy eficiente en comparación con el resto pero es mucho menos flexible puesto que necesita recompilar el código fuente cada vez que se desee modificar la semántica de un programa.

#### 6.4.3.2 Reflectividad en Tiempo de Ejecución.

El siguiente grupo de MOPs asocia el sistema base con el meta-sistema en tiempo de ejecución. Las llamadas al protocolo se realizan realmente cuando la aplicación se está ejecutando. Es por esto por lo que la naturaleza de estos lenguajes es interpretada. Ejemplos de éstos son Closette [Kiczales91] y MetaJava[26] [Kleinöder96].

Closette es un intérprete de un subconjunto de CLOS desarrollado en CLOS [Steele90]. Las primitivas de comunicación entre los dos niveles del sistema se implementan normalmente mediante macros. Estas macros se expanden a código CLOS. Este código accede y modifica la interpretación del lenguaje, consiguiendo así reflectividad computacional sobre unas primitivas.

En la implementación de este intérprete se identifican tres capas de código [Kiczales91]:

Las capa de macros. Estas macros se expanden consiguiendo traducciones a código que interactúa con el intérprete. Se consigue así el enlace entre los dos sistemas en tiempo de ejecución.

La capa de “pegamento”. Forma un conjunto de funciones que trabajan con el intérprete facilitando así la implementación de las macros. Un ejemplo puede ser la fun-

[26] Actualmente MetaJava se denomina MetaXava por problemas comerciales. MetaJava (o MetaXava) no es un producto de Sun Microsystems.

ción `find-class` que busca un meta-objeto de clase que coincida con un nombre.

La capa de nivel más bajo. Es la representación de la estructura y comportamiento de los meta-objetos. Esta implementación representa el modo en el que se interpreta el lenguaje.

El ejemplo de MetaJava es muy similar al de Closette. La principal variación se produce en el modo en el que están implementados. MetaJava, al igual que Java, es un lenguaje que se compila a otro de menor nivel y éste es posteriormente interpretado. La diferencia es que la máquina virtual de MetaJava posee añadidos de ciertas primitivas de reflectividad computacional.

El lenguaje de alto nivel posee un conjunto de clases e interfaces para comunicar el sistema base con el meta-sistema –implementando así un MOP. Este software se compila al código binario de la máquina, haciendo uso de unas primitivas de sincronización entre los dos sistemas.

En el lenguaje de alto nivel se programa:

La aplicación base mediante objetos y clases.

Los meta-objetos haciendo uso del software diseñado para obtener reflectividad.

La conexión entre ambos. Determinados objetos o grupos de objetos se enlazan con los meta-objetos pudiendo así modificar su comportamiento.

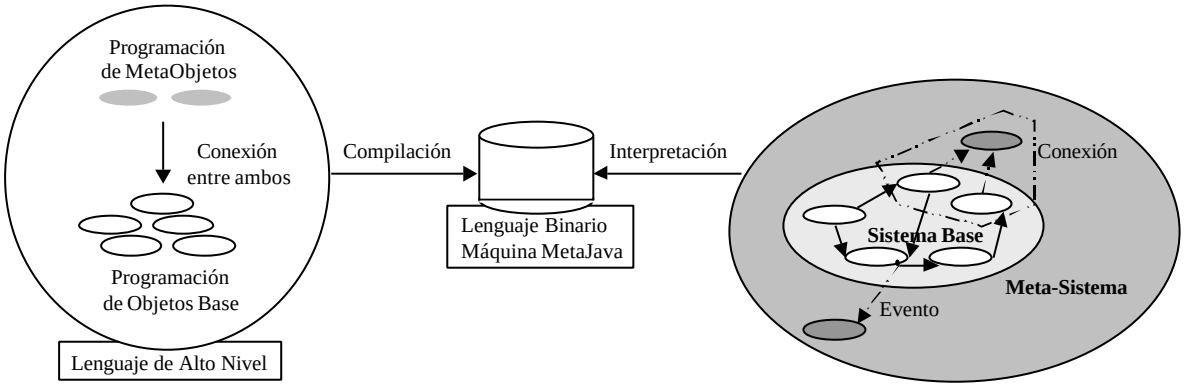


**Figura 32: Reflectividad computacional en tiempo de ejecución con MetaJava.**

En tiempo de ejecución (Figura 32) se ejecuta el sistema base y, en la generación de los eventos capturados en la conexión de los dos sistemas, se ejecutan los métodos propios de los meta-objetos que se hayan conectado con los objetos tratados.

Las similitudes entre estos sistemas reflectivos son que el conjunto de operaciones computacionales modificables por el sistema base están predefinidas. Estos sistemas son más flexibles que los no reflectivos pero su flexibilidad se ve limitada por el número de primitivas de reflectividad existentes.

La principal diferencia entre MetaJava y Closette es el número de niveles de computación existentes en tiempo de ejecución. En el caso de Closette tenemos un intérprete de este lenguaje que se ejecuta sobre un intérprete de CLOS. Se utilizan por lo tanto dos niveles. Por el contrario en Java se ejecutan tanto los meta-objetos como los objetos en el mismo nivel, interpretados por el procesador de la máquina virtual. El enlace entre estos objetos se realiza mediante primitivas de la máquina.

El tener dos niveles de la torre en lugar de uno aporta mayor flexibilidad a Closette. Si queremos introducir una nueva primitiva del protocolo sólo tenemos que programarla sobre el intérprete de nivel inferior (el intérprete de CLOS). La modificación del protocolo en MetaJava pasa por la modificación de la máquina abstracta y el software de alto nivel utilizado para conseguir la reflectividad.

El tener un nivel menos hace que MetaJava sea más eficiente que Closette. Se sacrifica así flexibilidad por eficiencia.

En todos los casos vistos de reflectividad computacional, tenemos limitado el número de operaciones a realizar desde el meta-sistema para acceder al sistema base. Esta limitación viene impuesta por el protocolo utilizado. Identificaremos este tipo de reflectividad computacional como "limitada" en el resto del documento.

## 6.5 Trabajos Relacionados.

Buscando la flexibilidad del software y de sistemas informáticos, existen sistemas que proponen determinadas soluciones para conseguir un software más dúctil. Comentaremos una serie de trabajos relacionados con el objetivo de la reflectividad aunque con un planteamiento distinto.

### 6.5.1 Implementaciones Abiertas.

Las implementaciones abiertas (*open implementations*) [Kiczles96] son módulos software que pueden adaptar o cambiar su implementación interna para cubrir las distintas necesidades de los posibles clientes.

Las implementaciones abiertas rompen con el concepto de caja negra. Aceptan el uso de una interfaz para especificar las funcionalidades del módulo software pero incluyen la posibilidad de identificar posibles modos o implantaciones de llevar estos procedimientos a cabo.

Un ejemplo típico de este problema es el uso de un contenedor software (módulo que alberga un conjunto de elementos). Un contenedor puede identificar funcionalidades como crear, insertar, borrar, buscar,... El uso que se vaya a dar a este contenedor determina una implementación concreta como más eficiente que el resto. En el caso de los compiladores, la tabla de símbolos es un contenedor sobre el que las operaciones a realizar son básicamente inserciones y búsquedas. Una implementación eficiente de este tipo de contenedores utiliza métodos de *hashing* [Cueva92b].

Para conseguir "abrir" el concepto de caja negra, se definen los patrones de uso (*pattern of use*) [Kiczales96b]. Se separa el código que utiliza normalmente el módulo software del código que ha de solicitar o demandar una implementación dada (*ISC code*). En este segundo código, se le pasa a la caja negra el patrón de uso especificado y ésta elige la implementación en función del patrón. En el ejemplo propuesto, el código ISC podría invocar a la función crear pasándole una constante que identificase la implantación –`crear("TablaHash")`.

En este campo de investigación se están desarrollando prototipos de metodologías de análisis y diseño basados en implementaciones abiertas [Maeda97].

### 6.5.2 Programación Orientada al Aspecto.

En este tipo de programación demandan un nuevo paradigma de programación por la ineficacia de los paradigmas existentes para la resolución de un problema.

En muchos casos la resolución de un problema no pasa solamente por la obtención de cualquier resultado válido. Puede requerirse además un modo, aspecto o criterio en la solución

de un problema. De esta forma en la programación orientada al aspecto se identifican dos términos muy distintos [Kiczales96c]:

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 (*threads*).

Para una aplicación que filtre imágenes gráficas podemos tener como componentes las imágenes y los filtros o máscaras. Como aspectos tendremos el ahorro de memoria o la claridad del código.

El programar una aplicación que se divide en componentes con un aspecto u otro, modifica totalmente su implementación. No existe un paradigma que separe los componentes de los aspectos. En los paradigmas actuales la separación de componentes viene ceñida a aspectos predefinidos.

En la programación orientada al aspecto se soporta directamente la diferencia de aspectos y componentes, pudiendo programar las funcionalidades de la aplicación y, sin modificar éstas, variar los aspectos requeridos.

Se codifican los componentes en un lenguaje de componentes y los aspectos en un lenguaje de aspectos. Estos dos archivos son la entrada de un tejedor de aspectos (*aspect weaving*) que genera el programa final en código C.

### 6.5.3 Programación Adaptable.

El proyecto Demeter se basa en el desarrollo de software mediante un método adaptable que eleva el nivel de abstracción de la programación orientada a objetos [Lieberherr95].

Las aplicaciones programadas de forma adaptable tienen un formalismo menor que las diseñadas directamente sobre la programación orientada a objetos. En la programación adaptable se identifican las clases y un conjunto de patrones de propagación (*propagation patterns*) que declaran las relaciones entre éstas. En este nivel de programación se identifican las relaciones entre las clases, sus operaciones básicas y un conjunto de limitaciones pero sin llegar a programar todo su comportamiento.

Las partes en las que se divide un patrón de propagación son:

Especificación de la operación. Es la interfaz de una computación.

Especificación de un camino. Indica el camino en el grafo de relaciones que va a seguir el cálculo de la operación.

Fragmento de código. Se especifica el código propio de la realización de la operación.

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 puede ser formalizada en distintas aplicaciones finales. Para conseguir la aplicación final en C++, debemos especificar aparte una personalización (*customization*) de ésta.

Tenemos dos niveles en la programación adaptable. El primero es un alto nivel de abstracción que especifica clases y relaciones entre ellas. Una vez conseguida esta especificación, implementamos de forma refinada su funcionamiento mediante una personalización.

Distintas personalizaciones de un mismo grafo de propagación dan lugar a distintas aplicaciones.

### 6.5.4 Filtros de Composición.

Los filtros de composición constituyen una extensión del modelo de programación orientada a objetos convencional añadiendo los filtros de composición a la abstracción de objeto [Aksit92]. Los objetos se definen como una instancia de una clase que consiste en métodos, variables de instancia o atributos, condiciones, objetos internos y externos y uno o más filtros.

Los filtros son objetos instanciados de clases de tipo filtro. El propósito de los filtros es el manejo y control del envío y recibo de mensajes. Especifican condiciones de aptitud o rechazo de los mensajes y determinan así la acción resultante. Cada filtro se puede programar cada clase que lo utilice. El sistema se asegura que un mensaje sea procesado por el filtro apropiado antes de que el método correspondiente sea ejecutado: cuando se recibe tiene que ser filtrado por un filtro de entrada y cuando se envía por un filtro de salida.

Las acciones que se llevan a cabo una vez aceptado o denegado un mensaje por un filtro dependen del tipo de filtro. Se han desarrollado filtros para aplicaciones con delegación en la herencia, tiempo real [Aksit94], manejo de errores y sincronización de procesos [Bergmans94].

Los filtros aportan al programador la oportunidad de poder atrapar el envío y recepción de mensajes y poder llevar a cabo determinadas acciones antes de que el método sea realmente ejecutado. Se separan así dos niveles de abstracción, el de mayor nivel (método) y el dependiente de la implantación o de bajo nivel (filtro). Un ejemplo de la distinción clara de estos dos niveles puede ser una aplicación en tiempo real.

La forma en la que se consigue una aplicación flexible es pudiendo modificar la semántica de los pasos de mensajes y de la ejecución de los métodos, mediante el uso de filtros de composición. La flexibilidad está pues limitada al paso y recepción de mensajes.

# Capítulo 7:

# Líneas de Investigación

En este trabajo de investigación se han introducido una serie estudios relacionados con diversos campos dentro del mundo de la informática. De forma resumida, los temas tratados pueden englobarse dentro de uno de los siguientes puntos:

Reflectividad computacional.

Máquinas abstractas y arquitecturas distribuidas.

Sistemas integrales orientados a objetos.

Siguiendo con la introducción realizada en estos temas, se proponen ciertas líneas de investigación en las que se pueden tratar más a fondo determinados puntos. En las distintas líneas de investigación se utilizan diversos conceptos de todos los puntos introducidos y en cada una de ellas se señalan distintas vertientes para la profundización en la investigación.

Las distintas líneas de investigación en las que iremos analizando a lo largo de este capítulo pueden enumerarse de la siguiente forma:

Especificación de una plataforma y arquitectura orientada a objetos flexible.

Reflectividad computacional.

Implantación de una máquina abstracta como sistema eficiente de computación.

Sistema integral orientado a objetos.

## 7.1 Especificación de una Plataforma y Arquitectura Orientada a Objetos Flexible.

En el capítulo 2 hablamos de las máquinas abstractas y en el capítulo 3 de las arquitecturas distribuidas orientadas a objetos y en el segundo de ellos veíamos cómo un sistema de computación potente y portable era la unión de los dos conceptos.

Sin embargo, esta unión posee también una serie de carencias. Comentaremos el porqué de estas limitaciones así como las carencias existentes en las máquinas abstractas más utilizadas. Finalmente fijaremos unos criterios de diseño para un sistema de computación flexible.

### 7.1.1 Carencias en la unión de plataformas independientes y arquitecturas distribuidas.

El concepto de plataforma independiente finaliza justo donde empieza el concepto de arquitectura distribuida. Los dos conceptos en sí tienen ventajas en su utilización y la unión de ambos aporta ciertas ventajas vistas en 3.4.

El desarrollo de aplicaciones con la plataforma Java [Gosling96] y la arquitectura CORBA [OMG95] es un claro ejemplo práctico de la combinación de estos dos conceptos [Orfali98]. Java garantiza una portabilidad y movilidad (descarga bajo demanda) del software y CORBA asegura la distribución de la computación de una forma abierta a cualquier otra plataforma.

Sin embargo, la creación de plataformas independientes y arquitecturas distribuidas han sido tareas desarrolladas por separado. Dicha separación a dado lugar a la existencia de limita-

ciones que se hubiesen reducido al diseñarse un sistema de computación, es decir los dos conceptos mencionados en sólo uno.

Ejemplos de las limitaciones existentes son el paso por valor de objetos, la gestión de memoria distribuida (el caso de un recolector de basura distribuido), procesos distribuidos (sincronización de procesos remotos), especificación de agentes móviles,...

Las principales causas de las limitaciones existentes son producidas por:

La plataforma independiente sólo especifica la interpretación del código. El no identificar la representación de los objetos[27] y la libre implementación de aspectos como el gestor de procesos y el recolector de basura limita la movilidad de los objetos y la gestión de memoria y procesos por un sistema operativo distribuido.

En lo referente a las arquitecturas distribuidas, no se especifica tampoco nada relativo a la computación. De esta forma el sistema es muy abierto pero se ve limitado en ciertos aspectos[28].

### 7.1.2 Carencias de las Máquinas Abstractas Existentes.

En 2.5 comentamos de forma resumida la evolución de las máquinas abstractas en función de sus características principales. De todas las mencionadas nos centraremos en la máquina abstracta de Smalltalk-80 [Goldberg83] y de Java [Sun95]. Ambas poseen características similares a las buscadas en la línea de investigación propuesta.

La máquina de Smalltalk aporta un conjunto amplio de características interesantes entre las que podemos destacar su identificación de sistema integral (todo se desarrolla sobre el propio lenguaje), su reflectividad estructural y su gran flexibilidad (aportada por la unión de las dos características anteriores).

Por otra parte, posee una serie de inconvenientes que hace que sea insuficiente para nuestros objetivos. Las principales carencias son:

Alto número de primitivas. Smalltalk-80 posee un número elevado de primitivas para tratar su especificación. Esto hace que el sistema sea difícil de interpretar –por lo tanto poco portable –y mucho menos eficiente que sus competidores [Chambers91].

Especificación abierta. En la referencia [Goldberg83] se especifica la máquina de Smalltalk. Esta especificación constituyó un estándar que se solía modificar en cada implementación.
La plataforma Smalltalk no estaba pensada fundamentalmente para la portabilidad del código así que no existe realmente una especificación abierta que se siga rigurosamente.

Carece de distribución. No tiene propiedades de distribución como la invocación remota de métodos o la movilidad de objetos.

Posee un sistema de multitarea indirecto mediante clases que no es lo suficientemente abierto –manejable por un sistema operativo.

[27] En la especificación 1.1 de Java se define una representación simple del estado de un objeto mediante la serialización. Este método es utilizado con RMI para obtener el paso por valor de objetos. La clase ha de estar definida también en la máquina receptora.

[28] El criterio adaptado no es realmente un error. Se trata de alcanzar un objetivo distinto al que nosotros buscamos. Si se trata de diseñar una arquitectura abierta, el modo en que se realiza la computación puede no interesar.

La otra especificación de máquinas abstractas a estudiar es el caso de Java. Como comentamos en 2.5.3.3, esta máquina ha revolucionado el mercado del software por utilizar el concepto de máquina abstracta para conseguir fundamentalmente una plataforma portable, un mecanismo de seguridad integral y movilidad y distribución del código.

Los objetivos buscados en el diseño de la máquina de Java son también exigidos a la plataforma que buscamos. La creación de nuestro sistema busca además una plataforma que de soporte a una arquitectura de objetos y a un sistema integral. Java fue diseñada para conseguir simplemente una nueva plataforma de programación y esto hace que posea limitaciones para acaparar nuestros objetivos.

A continuación enumeraremos sus principales carencias para el desarrollo de nuestro sistema de computación.

**Varias plataformas**. La plataforma Java se basa en una máquina virtual y un código denominado *core* escrito sobre ésta. La existencia de una única plataforma identificaría este par como obligado en toda la simulación de esta plataforma, pero no es así.

En los orígenes de la creación de la plataforma, Java se utilizaba para interconectar distintos dispositivos. Posteriormente fue aumentando y se introdujo en los ordenadores. La idea de plataforma independiente mantiene la conexión con dispositivos, pero para ello se definen tres plataformas adicionales:

*The Java Embedded Platform*. Para sistemas empotrados. Tiene un API menor que el *core*.

*The Java Personal Platform*. Reduce el API de la plataforma empotrada.

*The Java Card Platform*. No solo limita el API, sino que reduce el juego de instrucciones.

La reducción del API básico –*core* –y sobre todo la reducción del juego de instrucciones, identifica la plataforma base como compleja y disminuye el grado de portabilidad. Así se habla de plataformas distintas [Venners98].

El problema planteado viene por el intento de resolver distintos entornos de programación lo suficientemente distintos con una única especificación de máquina abstracta. Para conseguir esto es necesario un mecanismo de flexibilidad como el propuesto para las *Virtual Virtual Machines* [Folliot98].

**Planificación de hilos (*threads*)**. Java define dos mecanismos de sincronización de procesos basados en monitores: exclusión mutua a cooperación [Gosling96]. Sin embargo, la planificación de los distintos procesos de la máquina no viene definida en la especificación siendo por lo tanto dependiente de la implementación.

Una aplicación portable no deberá nunca suponer una determinada planificación aunque el entorno de implementación la utilice. Si transportamos ese código a otra plataforma, puede darse que la planificación del nuevo simulador de la máquina abstracta sea distinta.

En la búsqueda de una máquina flexible para el diseño de un sistema de computación, el criterio llevado a cabo con la máquina de Java es demasiado estricto. Tanto para el diseño de un sistema operativo como para determinados tipos de aplicaciones (por ejemplo las aplicaciones en tiempo real) es conveniente poder especificar un sistema de planificación de procesos.

**Código nativo**. En la plataforma de Java se especificó una interfaz de acceso a código nativo [Sun97c]–de la máquina real –para cubrir fundamentalmente los siguientes objetivos:

Acceso a características de la plataforma real no contempladas en las APIs de la máquina de Java.

Para acceder a un sistema existente o bien a unas librerías escritas en otro lenguaje distinto a Java.

Para acelerar la ejecución de una aplicación, se puede implementar la parte crítica del código mediante código nativo. Así los tiempos de ejecución se reducen considerablemente puesto que no se realiza la interpretación de dicho código.

El hecho de implementar un solo método en código nativo implica la imposibilidad de migrar el código. Para conseguir la ejecución de la aplicación en varias plataformas, necesitamos implementar los métodos nativos en todas las máquinas en las que se vaya a ejecutar ésta.

Un primer ejemplo de la contraposición entre la portabilidad de la plataforma y el código nativo es que el propio API *core* de la plataforma tiene métodos nativos[29]. Así para ejecutar una aplicación Java se necesita, además de la implantación del simulador, la compilación a la máquina real de todos los métodos nativos de las APIs.

**Gestión de memoria**. Al igual que en el caso de los procesos, la gestión de la memoria y por lo tanto el recolector de basura no son modificables y son totalmente dependientes de la implementación de la máquina.

Java identifica una serie de características como la carga de las clases –con el *Class Loader* –y las restricciones de seguridad –con el *Security Manager* –como flexibles permitiendo modificar su semántica mediante el polimorfismo. Sin embargo, otras facetas como la gestión de procesos y memoria no gozan de esta flexibilidad. Esto limita la gestión global en un sistema distribuido y por lo tanto su flexibilidad.

Bajo estas demandas de flexibilidad existen campos de investigación que identifican todos los módulos del sistema de computación y los abren a distintas implementaciones. Ejemplos son las máquinas virtuales adaptables (*Adaptive virtual machine*) [Baillarguet99] y las máquinas virtuales virtuales (*Virtual virtual machines*) [Folliot98].

**Complejo juego de instrucciones**. En el bajo nivel de Java se identifican muchas facetas que pueden implementarse en un nivel superior y compilarse a éste, dejando así la ciertas abstracciones como tarea de traducción al nivel inferior. El resultado es un elevado número de instrucciones máquina produciendo una complejidad para su interpretación así como una difícil optimización.

Un ejemplo del elevado número de instrucciones se observa en las distintas formas existentes de invocar a un método: 10 códigos distintos. Esto se produce por identificar en la abstracción de la máquina conceptos como métodos de clase, de instancia, interfaces, constructores y optimizaciones con nuevas instrucciones.

La optimización de sistemas más simples como el Self [Ungar87] ha superado en eficiencia y tiempo de implementación, a máquinas más complejas como la de Smalltalk [Chambers91]. Las distintas técnicas utilizadas en Self como la compilación dinámica adaptable [Hölzle94] y otras adicionales están siendo añadidas a la máquina virtual de Java para obtener la "*Java HotSpot Virtual Machine*" [Sun98d].

**Representación de los objetos**. Como comentamos en 7.1.1, la plataforma de Java no identifica la representación completa de un objeto sino que la deja como dependiente de implementación. Este requerimiento simplifica la implantación de la máquina pero limita su funcionamiento en casos como los agentes móviles y el paso por valor.

Java goza actualmente de un mecanismo muy básico de representación de objetos denominado serialización. Se representa una parte del estado de un objeto y se puede volcar éste a un *Stream* y por lo tanto a disco o a otra máquina. Problemas como la representación de su clase o la asociación de objetos no han sido resueltos de forma general y completa.

[29] Un ejemplo puede ser el método *hashCode* de *java.lang.Object*. Este método devuelve un entero que representa la clave *hash* de este objeto para el uso de tablas.

Vemos pues cómo los criterios de diseño de la máquina abstracta buscada son distintos a los existentes. La búsqueda de un sistema ortogonal de dotación de flexibilidad a la máquina es el principal criterio de diseño.

### 7.1.3 Criterios de Diseño de la Máquina Abstracta.

Definiremos una serie de criterios identificados para el diseño de la máquina abstracta que de soporte, como veremos, a distintas líneas de investigación. Generalmente se proporcionan unos criterios básicos de su diseño que la hacen distinta a las anteriormente mencionadas.

#### 7.1.3.1 Orientación a Objetos Basada en Prototipos.

Una clase es un conjunto de objetos que comparten una estructura común y un comportamiento común [Booch94]. La clase es un mecanismo de programación creado para identificar características comunes de los objetos.

Los objetos modelan un problema asimilándolo a la realidad. Las clases, sin embargo identifican abstracciones de agrupaciones de objetos. La programación orientada a objetos basada en prototipos elimina el concepto de clase, simplificando así el modelo de objetos.

Existen distintos niveles en los que se reconocen lenguajes con la eliminación o substitución del concepto de clase:

Programación OO basada en prototipos. La programación en Self elimina el concepto de clase e instanciación substituyéndolo por objetos prototipo, copia y delegación [Smith95].

Objetos representando clases. En el caso de Smalltalk-80, no existe la programación de clases sino la programación de objetos que representan comportamiento [Goldberg89]. Así todo objeto ha de estar enlazado con otro –mediante el atributo *class* –que indique su comportamiento. Esto da lugar a las metaclases que son clases de clases.

Existen clases y objetos que representan clases. En Java se separan de forma clara los conceptos de clase y objeto definiendo así un sistema de tipos [Campione97]. Sin embargo, para dotar a la plataforma de introspección en tiempo de ejecución, las clases son representadas por objetos en tiempo de ejecución [Sun97d].

Clases y objetos totalmente separados. En C++ [Stroustrup98] una clase existe en tiempo de compilación para determinar el sistema de tipos de los objetos [Cardelli97]. En tiempo de ejecución no posee representación accesible.

Para definir una plataforma de objetos distribuida, reflectiva y persistente, el concepto de clase conlleva una serie de inconvenientes. En el caso de la distribución, cuando queremos realizar un paso por valor o bien mover un agente, debemos controlar la localización de su clase. Enviar el objeto no es suficiente.

Cuando hablamos de reflectividad las clases también complican la representación del sistema. Si nos centramos en el caso de MetaJava [Golm97b], ¿cuál es el mecanismo seguido para modificar un método de un solo objeto (sin modificar el resto de las instancias)? La creación de unas subclases denominadas *shadow classes*. La representación de estas en el meta-sistema poseen también meta-objetos y deben guardar enlaces con las clases originales para mantener el grafo de herencia. En [Golm97b] podemos apreciar la complejidad de su tratamiento.

La persistencia de un objeto pasa por la persistencia de la clase de la que es instancia [Booch94]. La modificación de la clase representa una posible incoherencia para sus objetos asociados [Ortin97].

Finalmente, la simplicidad conceptual del modelo de objetos basados en prototipos hace más atractivo su uso para representaciones de bajo nivel –como nuestra máquina abstracta – sin limitar la capacidad de representación del sistema (en [Evins96] se dan patrones de traducción de un modelo basado en clases a uno basado en prototipos). Así tenemos ejemplos de lenguajes como Self [Ungar87] y sistemas reflectivos como Moostrap [Mulet93] basados en prototipos.

#### 7.1.3.2 Modelo Activo de Objetos.

La máquina abstracta ha de proporcionar un mecanismo de computación que sea uniforme, potente, flexible y adaptable por el sistema operativo [Tajes96]. La máquina abstracta computará las invocaciones a los distintos métodos pudiéndose modificar la política de planificación por el sistema operativo.

En la máquina abstracta se ofrece como abstracción base la del objeto. La localización del objeto ha de ser absolutamente transparente para proporcionar directamente la computación de objetos sobre distribución y persistencia. La invocación de un método de un objeto podrá ser llevada a cabo indistintamente si es persistente o temporal o si es local o remoto. De esta forma, podemos ampliar el concepto de estado propuesto por [Booch94]: "El estado de un objeto abarca todas las propiedades (normalmente estáticas) del mismo más los valores actuales (normalmente dinámicos) de cada una de estas propiedades".

Ampliamos la definición de estado de un objeto agregando "el estado computacional en que se encuentran todos los procesos que se están ejecutando sobre dicho objeto". De esta forma se aboga por un modelo de objetos activo en el que éste encapsula su estado y la computación que se está llevando a cabo. No existe una entidad independiente que realice esta labor [Tajes97].


**Figura 33: Estructura de un objeto autocontenido.**

Como se aprecia en Figura 33 los objetos en este modelo son autocontenidos. Poseen un estado en tiempo de ejecución que viene definido por los valores de sus atributos. Pueden describir comportamiento (apartado 7.1.3.1) al igual que lo hace una clase en un modelo orientado a clases y almacenan también la parte de su estado relacionado con su computación.

Las ventajas de la existencia de objetos autocontenidos son relativas a la independencia de la ubicación de éstos, permitiendo así su movilidad entre computadores y su almacenamiento en disco. Se ha desarrollado un sistema de persistencia para una máquina abstracta [Ortin97b] en el que se almacenaban los objetos pero no la computación de éstos –persistencia

de computación. Con los objetos autocontenidos, la persistencia computacional es mucho más viable.

#### 7.1.3.3 Sistema de Referencias Globales.

La forma identificada para acceder a los distintos objetos autocontenidos del sistema es mediante referencias globales.

El primer prototipo de máquina abstracta poseía el concepto de referencia a objetos que se estaban ejecutando en esa máquina [Izquierdo96]: para acceder a un objeto, antes había que obtener una referencia a éste. Para poder acceder a objetos persistentes, las referencias se modificaron para poder acceder a un sistema de memoria virtual [Ortin97].

Para poder acceder a objetos distribuidos en distintas máquinas e independientes de su estado de persistencia, el sistema de referencias ha de poder hacer referencia a cualquier objeto. De esta forma se ha de identificar un "sistema de nombrado" [OMG95b].

Para poder acceder a un objeto desde una máquina y que éste pueda moverse entre el sistema de computación se utilizará un sistema de indireccionamiento en cada máquina utilizado en sistemas operativos [Deitel93] y en diversas implantaciones de gestores de memoria de la máquina de Smalltalk-80 [Krasner83].

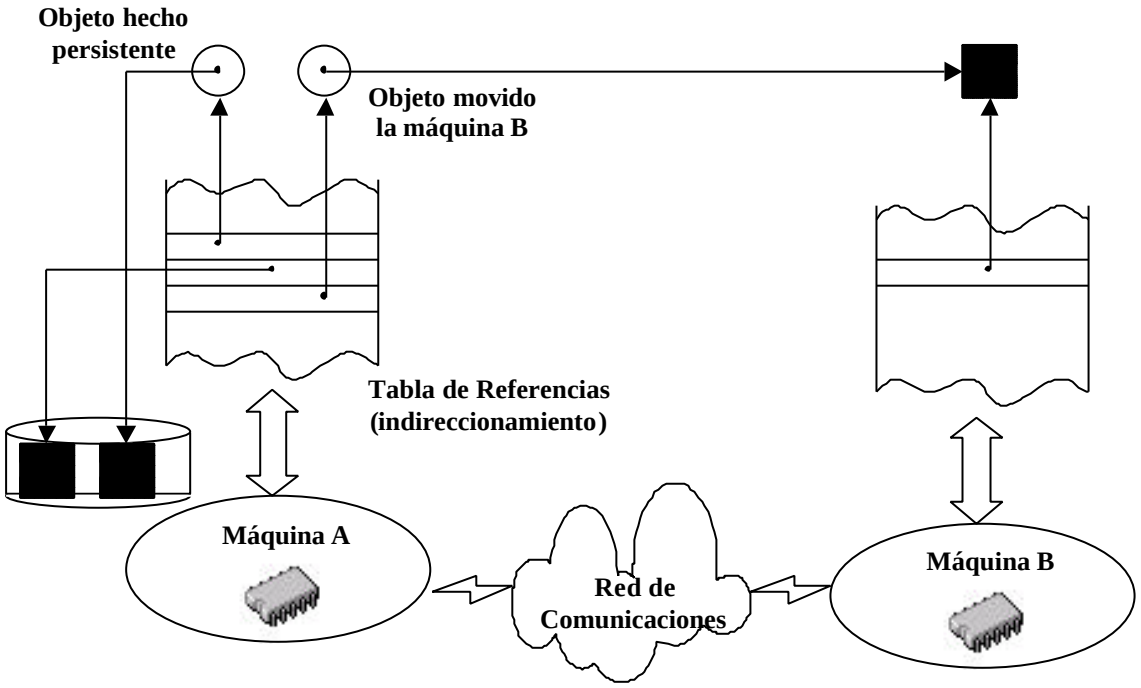


**Figura 34: Indireccionamiento mediante una tabla de referencias globales.**

Cada máquina abstracta en ejecución posee una tabla de indireccionamiento de objetos (Figura 34). Cada elemento de esta tabla posee una referencia global. La máquina tiene tantas entradas a esta tabla como referencias esté utilizando el programador para acceder a este objeto. Cuando un objeto pase a estado persistente o se mueva trasladándose a otra máquina, simplemente habrá que modificar la referencia global de la tabla, puesto que el acceso al objeto se hace mediante una indirección.

#### 7.1.3.4 Reflectividad.

El objetivo de proporcionar a la máquina abstracta una arquitectura reflectiva es para que pueda ser extendida de manera transparente por objetos del sistema operativo, proporcionados en forma de objetos de usuario [Alvarez98].

Podemos diferenciar la extensión de la máquina en dos sentidos:

Extensión del nivel de abstracción. Los objetos podrán adquirir propiedades adicionales como persistencia y distribución, programando el acceso a éstos mediante reflectividad estructural. Este tipo de reflectividad proporciona un mecanismo de ampliación, mantenimiento y modificación del modelo básico o primitivo de la máquina.

Funcionalidad flexible. Mediante la reflectividad computacional podemos modificar la semántica del juego de instrucciones de la máquina. Así una aplicación en código nativo de la máquina puede ser implantada en un comportamiento de control de sistemas en tiempo real –modificando la semántica de sus instrucciones.

La reflectividad estructural es un criterio básico para el desarrollo de un sistema operativo sobre la máquina abstracta. Al igual que un sistema operativo comercial accede a la máquina física para obtener un nivel de abstracción superior [Deitel93], la reflectividad estructural es el mecanismo de acceso del sistema operativo a la máquina abstracta.

Existen prototipos de sistemas operativos orientados a objetos que utilizan el concepto de reflectividad. Apertos [Yokote93] divide el sistema en un *micro kernel* (*MetaCore*) que posee las primitivas de computación, un meta-sistema y un sistema base. La comunicación entre los dos sistemas se realiza mediante *Reflectors*. El sistema operativo gestiona la concurrencia y distribución apoyándose en la arquitectura reflectiva.

En lo referente a la modificación de la semántica de las instrucciones de la máquina mediante reflectividad computacional, hemos identificado la línea de investigación "Reflectividad computacional." en la que profundizaremos en las posibilidades de este concepto.

## 7.2 Reflectividad computacional.

El diseño de una arquitectura reflectiva da lugar a distintos campos posibles de investigación. En el punto anterior hemos visto cómo se puede subir el nivel de abstracción de un entorno de programación gracias a un nuevo concepto de sistema de computación. En esta línea se investigan las mejoras propias del diseño de una arquitectura y plataforma basada en reflectividad.

Un campo muy poco explorado es la investigación de las posibilidades de la reflectividad computacional en la informática. Los sistemas actuales pasan principalmente por el uso de la introspección (6.2). El desarrollo de un prototipo de sistema de computación basado en los criterios de diseño identificado en la línea de investigación anterior, nos servirá para estudiar a fondo las posibilidades de la reflectividad computacional.

En este capítulo propondremos un nuevo nivel de reflectividad –superior a los niveles identificados en 6.4 –y las distintas posibilidades y campo de investigación en las que puede ser utilizado.

### 7.2.1 Reflectividad Computacional sin MOPs.

Hemos visto como el mayor nivel de reflectividad venía otorgado por una reflectividad computacional "limitada" a priori por un protocolo de comunicación entre el sistema base y el

meta-sistema: MOP (*Meta Object Protocol*). De éstos, los más flexibles son los que establecen el enlace en tiempo de ejecución (como vimos en 6.4.3.2).

Existen diversas líneas de investigación basadas en el desarrollo de MOPs y en su posible utilización. Sin embargo, este tipo de reflectividad no es todo lo flexible que pudiese ser. Ya en el campo de la investigación, se ha propuesto un sistema con un mayor grado de flexibilidad [Ortin99].

El sistema propuesto modela una torre de intérpretes de dos niveles. El nivel que soporta el meta-sistema está basado dotado de reflectividad estructural (puede ser la máquina abstracta de la anterior línea de investigación). Sobre este nivel se ejecuta un intérprete de un lenguaje –pudiendo ser el mismo –dotado de reflectividad computacional.

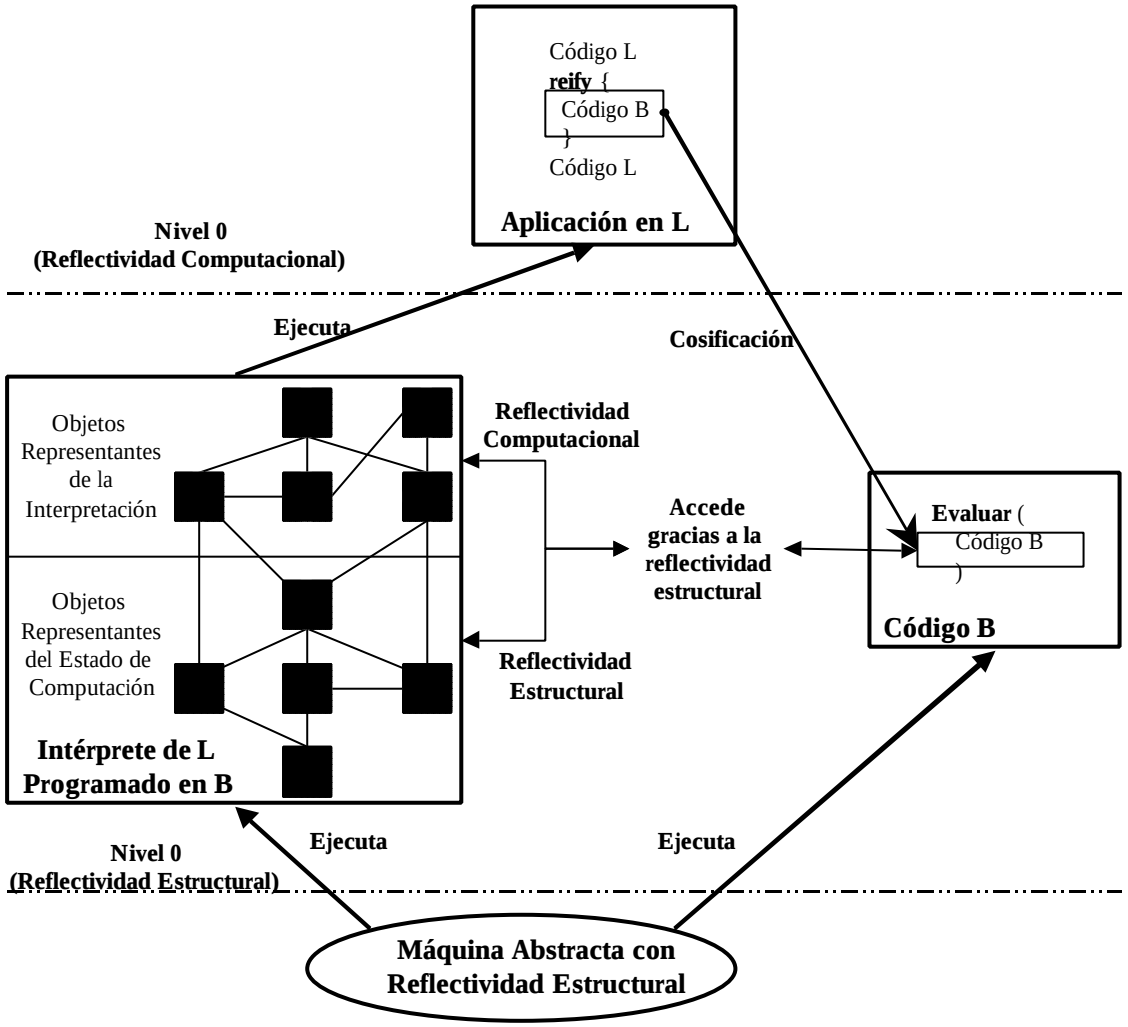


**Figura 35: Torre de dos niveles para conseguir reflectividad computacional mediante reflectividad estructural.**

En la Figura 35 se aprecia una máquina virtual dotada de reflectividad estructural, ejecutando un intérprete del lenguaje **L** codificado en el lenguaje **B**. En tiempo de ejecución, el intérprete tendrá objetos que representan objetos del programa y objetos que representan su semántica y computación. Por lo tanto:

El acceso a los objetos representantes del estado del programa facilitan la reflectividad estructural de éste.

El acceso a los objetos que identifican la computación del lenguaje **L**, proporciona reflectividad computacional a dicho lenguaje.

Hay que darse cuenta de que la manipulación de estos objetos siempre es posible gracias a la reflectividad estructural de la máquina virtual. El único requisito necesario para llevar a cabo todo lo propuesto es, ¿cómo poder ejecutar en la máquina código nativo desde el lenguaje **L**? Es decir, ¿cómo es posible realizar la cosificación[30]?

La respuesta a la pregunta propuesta se satisface con la implementación de una instrucción de la máquina –en el lenguaje **B** –que permita evaluar una representación de datos como instrucciones[31]. La instrucción identificada deberá ser la implementación de la función `eval` del Lisp, en el paradigma de la orientación a objetos [Steele90].

Vemos cómo el nivel de reflectividad ya no queda limitado a la especificación a priori –en tiempo de diseño del lenguaje –de un protocolo de comunicación como sucede en el caso de los MOPs. La flexibilidad ganada queda reflejada en el hecho de que la semántica del lenguaje **L** puede ser modificada en su totalidad, expresándolo en el lenguaje **B**.

En el apéndice B se muestra un ejemplo del sistema propuesto. Como lenguaje de alto nivel (lenguaje **L**) se implementó un simple evaluador de expresiones aritméticas. El lenguaje **B** es el lenguaje de programación Lisp [Steele90] que es a su vez interpretado. Se muestran ejemplos de las posibilidades de la reflectividad computacional. Lo más interesante es ver cómo no hay un protocolo predefinido de comunicación entre el sistema base y el meta-sistema: podemos evaluar cualquier expresión en Lisp.

Las limitaciones del prototipo mostrado quedan impuestas por la simplicidad del lenguaje interpretado y por las carencias eminentes del lenguaje Lisp en lo referente a la reflectividad estructural (para más información consultar el apéndice B).

### 7.2.2 Reflectividad Computacional Aplicada.

Veremos en los diversos puntos de esta sección cómo se puede aplicar el término de reflectividad computacional a diversos campos de la informática, muchos de ellos en fase de investigación. En todos ellos se aprecia la posibilidad de tener un lenguaje que se permita modificar su semántica (reflectividad computacional) para distintos usos.

#### 7.2.2.1 Ingeniería del Software. Separación de Incumbencias.

Hemos visto en 6.5 cómo existen líneas de investigación relacionadas con software flexible y por lo tanto con la reflectividad. Ejemplo de éstos son el análisis y diseño de implementaciones abiertas (*Open Implementation Analisys and Design*) [Maeda97], programación orientada al aspecto (*Aspect Oriented Programming*) [Kiczales96c] y software adaptable (*Adaptive Software*) [Lieberherr95].

Todos estos estudios están relacionados con la forma de conseguir lo que se conoce como “Separación de Incumbencias” (*Separation of Concerns*) [Hürsch95]. Cada trabajo identifica una manera de separar los algoritmos básicos de resolución de un problema, con los aspectos propios de su implementación.

En la separación de incumbencias se puede resolver un problema modelando un conjunto de clases y objetos que interactúan entre sí. Esta parte se define como el nivel conceptual. El diseño formal de este modelo puede desarrollarse mediante una determinada metodología como por ejemplo UML [UML98].

---

[30] La operación *reification* o el salto en la torre de intérpretes propuesta en 6.1.
[31] Vemos como precisamente ésta es la definición de cosificación: ver una idea como “cosa”.

Una vez especificado el nivel conceptual cabe definir la parte de implementación. Cómo se interpreta la invocación de los métodos y temas relacionados con la sincronización, distribución, restricciones de sistemas en tiempo real y tolerancia a fallos son tareas de este nivel.

La separación de incumbencias es identificada como un paradigma en la ingeniería del software. Los mecanismos utilizados para llevarla a cabo se denominan técnicas de separación. Entre las distintas técnicas propuestas está la meta-programación [Hürsch95].

La reflectividad computacional juega pues un papel determinante en las distintas técnicas de separación de incumbencias en ingeniería del software flexible y adaptable.

#### 7.2.2.2 Sistemas de Bases de Datos Orientadas a Objetos.

Los lenguajes de programación orientados a objetos están completamente extendidos, pero el dotar de persistencia a éstos no es una tarea trivial [Booch94]. Actualmente está en fase de investigación y de desarrollo diversas técnicas, estándares y lenguajes para dotar de persistencia a las tecnologías orientadas a objetos.

Los distintos entornos de programación orientada a objetos que soportan persistencia pasan por la ampliación de un lenguaje o la utilización de un conjunto de librerías o APIs para dotar de persistencia a un conjunto de objetos de una aplicación [Cattell94]. La condición de persistencia de un objeto pasa por introducir código adicional en la aplicación y el mantenimiento de ésta por parte del programador.

Haciendo uso del sistema de reflectividad computacional definido en 7.2.1, se ha propuesto en [Ortin99] un nuevo modelo de persistencia de objetos denominado “Persistencia Implícita”. Sobre un lenguaje computacionalmente reflectivo se diseña un motor de bases de datos orientado a objetos y un *middleware* encargado de hacer persistente una aplicación –o parte de ésta –sin necesidad de definir código intruso por parte del programador.

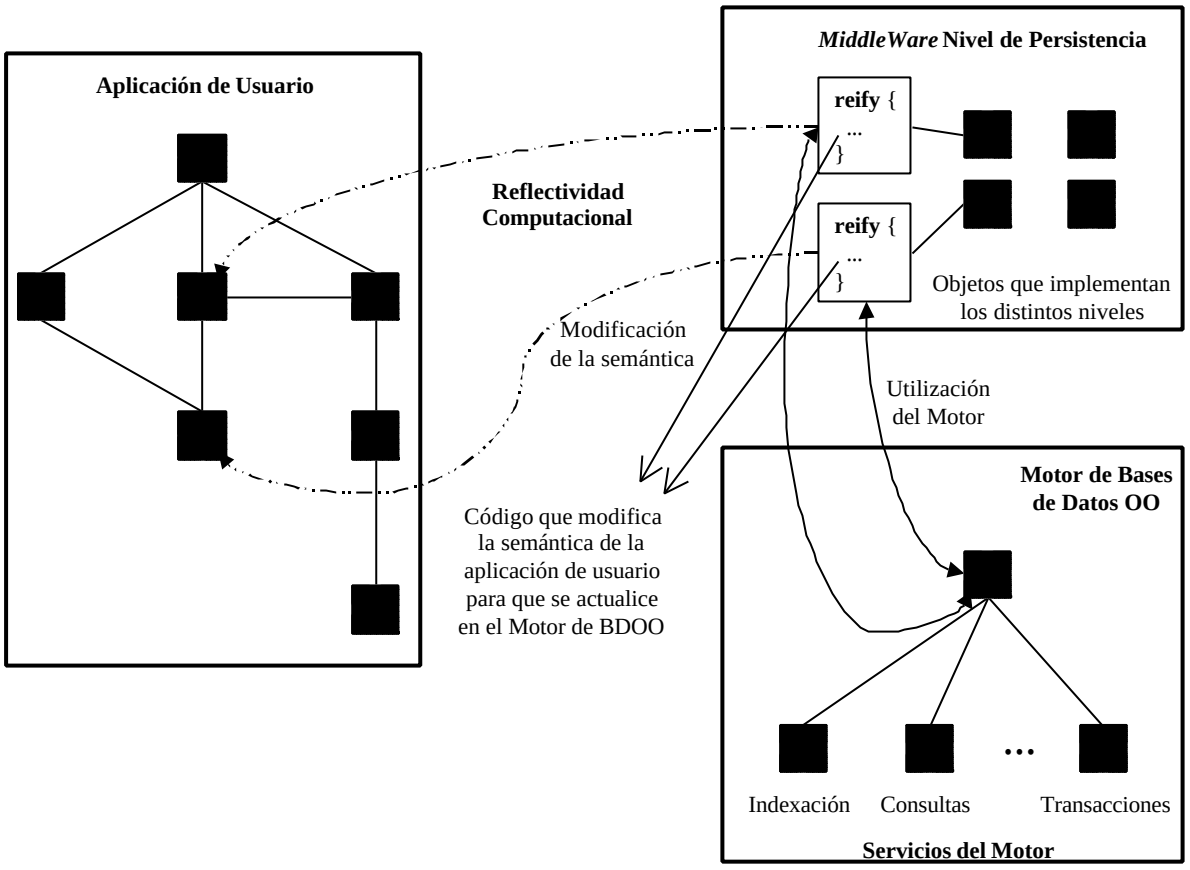


**Figura 36: Implementación de un sistema de "persistencia implícita" apoyándose en un sistema con reflectividad computacional.**

En Figura 36 el *middleware* "nivel de persistencia" se encarga de enlazar una aplicación con el motor de bases de datos:

Mediante reflectividad estructural el *middleware* es capaz de conocer toda la estructura de una aplicación.

Mediante reflectividad computacional es capaz de modificar la semántica del lenguaje para los objetos persistentes (por ejemplo la creación, destrucción y paso de mensajes a objetos).

De esta forma se obtiene un sistema de "Persistencia Implícita" sin necesidad de incluir código adicional a la aplicación y sin necesidad de que el usuario mantenga la gestión de persistencia. Existe otro conjunto de ventajas aportadas por la reflectividad computacional a un sistema de persistencia, mencionadas en [Ortin99].

#### 7.2.2.3 Control de Sistemas en Tiempo Real.

El hecho de poder modificar la semántica de un lenguaje, permite definir una determinada computación basada en la máquina física en la que se está interpretando el programa. De esta forma, la reflectividad computacional puede utilizar como un mecanismo para especificar lenguajes y sistemas de control en tiempo real.

Utilizando la reflectividad de una forma más limitada se han especificado sistemas de control en tiempo real mediante, por ejemplo, MetaJava [Golm97c] y filtros de composición (*Composition Filters*) [Aksit94].

Las limitaciones de ejecución de un método en un sistema en tiempo real vienen definidas por su duración. Estas restricciones se pueden agrupar en:

Limitaciones *soft*. La ejecución de un método tiene limitada su duración pero el caso de que no se consiga cumplir, no es necesariamente un error: se puede ejecutar un código adicional.

Limitaciones *real*. La ejecución de un método no puede superar nunca el tiempo impuesto en sus limitaciones.

Los sistemas existentes de control de aplicaciones en tiempo real basados en reflectividad, se basan en el primer tipo de limitaciones del sistema. Por otro lado, los sistemas de control con limitaciones reales, utilizan un lenguaje propio con las limitaciones de tiempo.

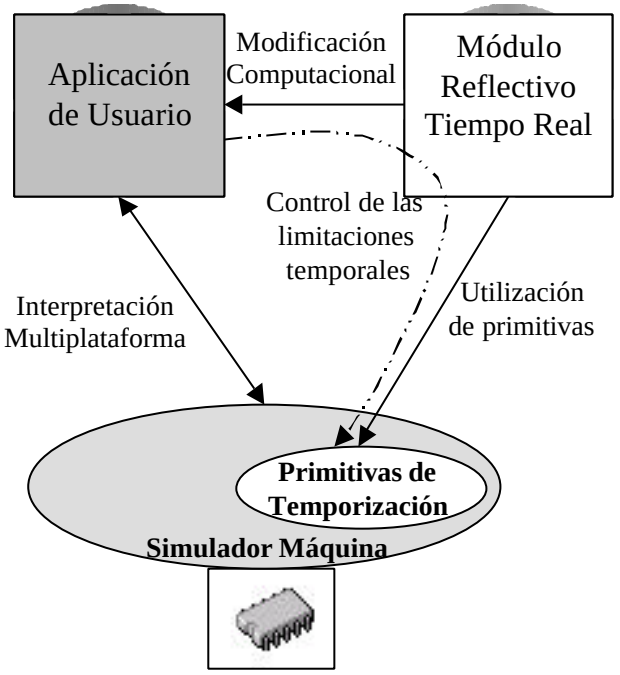


**Figura 37: Aplicación multiplataforma en tiempo real utilizando reflectividad computacional.**

La unión de los conceptos de máquina abstracta y reflectividad computacional permiten el desarrollo de sistemas en tiempo real, multiplataforma y basados en un único lenguaje. La simulación de la máquina consigue la independencia de plataforma y la unicidad del lenguaje utilizado. La reflectividad computacional permite modificar las primitivas del lenguaje para cumplir restricciones temporales, implementadas cada una físicamente en una plataforma.

#### 7.2.2.4 Agentes Móviles.

La movilidad de los objetos a través de las redes de computadores, ha dado lugar a una ampliación en la programación distribuida: los agentes móviles.

Las arquitecturas distribuidas proponen una interoperabilidad de objetos entre distintos sistemas, cada uno ubicado físicamente en una localización [OMG97][32]. Esto ha dado lugar al desarrollo de diversas plataformas de programación de agentes móviles que permitan la movilidad de los objetos. Ejemplos existentes son Java [Kramer96], KQML [Finin93] o CORBA [OMG95].

[32] Se está realizando una especificación del paso de objetos por valor, que será publicada en la versión 3 de CORBA.

La carencia de plataformas con reflectividad estructural y de un modelo de objetos "autocontenidos" como el propuesto en 7.1.3.2, hace difícil el desarrollo de una plataforma robusta y de fácil utilización, para la programación de agentes móviles [Perez98].

De los diversos prototipos desarrollados sobre la plataforma Java, IBM ha popularizado un API para el desarrollo de agentes móviles sobre la plataforma de Java, denominando a éstos *aglets* [Lange97]. Su éxito frente a otras plataformas es debido a las características que ofrece la plataforma de Java. Fundamentalmente su portabilidad y movilidad de código y su API de reflectividad estructural limitada [Sun97d].

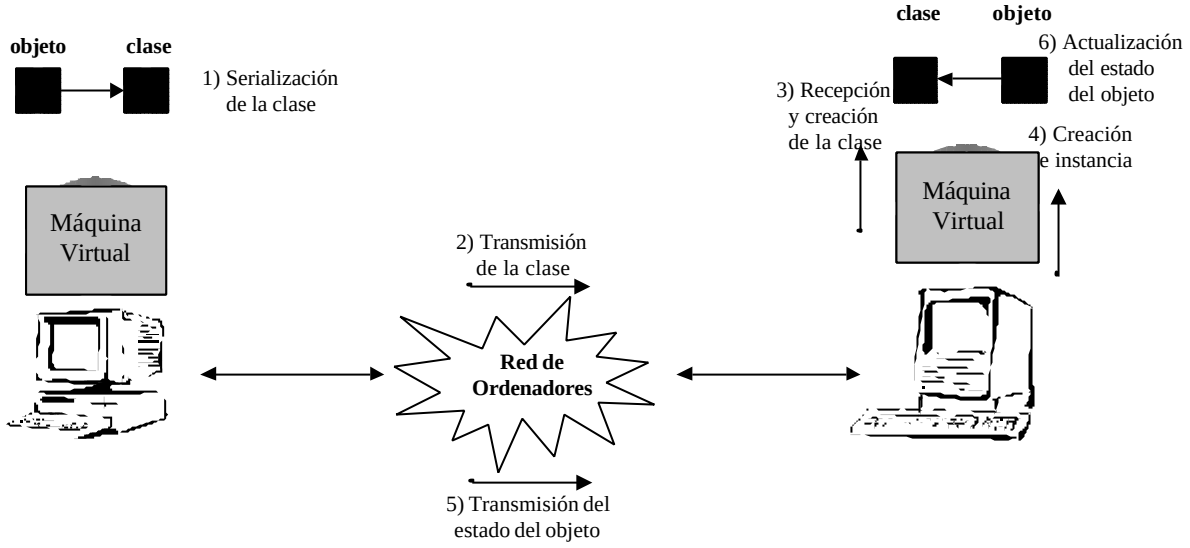


**Figura 38: Distintos pasos llevados a cabo para conseguir la movilidad de un objeto Java.**

En la Figura 38 podemos apreciar los distintos pasos a realizar para conseguir la movilidad de un agente sobre la plataforma Java. Aunque se pueda almacenar el estado de un objeto mediante serialización[33], la máquina remota ha de tener la clase a la que el objeto pertenece. Por lo tanto, lo primero que se debe realizar es serializar la clase, transmitirla por la red de comunicaciones mediante un *Stream* asociado a un *Socket* y crearla en tiempo de ejecución en la máquina remota.

Una vez existente la clase en la nueva máquina, se crea una instancia y se transmite el estado del objeto (del agente) para actualizar el estado de la nueva instancia creada. Finalmente, todo el acceso a los métodos y atributos del agente, en tiempo de ejecución, pueden realizarse gracias al paquete de reflectividad `java.lang.reflect` [Sun97d]. Estos accesos no se pueden realizar en tiempo de compilación puesto que la clase no tenía por qué existir.

Vemos como, ante esta serie de operaciones y limitaciones, la búsqueda de la plataforma reflectiva de la línea de investigación de 7.1 garantiza un entorno de computación más robusto y sencillo que los existentes en la actualidad.

Cabe preguntarse cómo la reflectividad computacional –al margen de la estructural – puede tomar parte en la investigación de agentes móviles. Sobre la plataforma mencionada, la reflectividad computacional se convierte en un mecanismo potente para el desarrollo de un lenguaje de implementación de agentes móviles (un lenguaje flexible y con semántica modificable, como proponíamos en 6.3).

[33] La palabra serialización se refiere a la implementación del *interface* de java `java.io.Serializable`.

Un ejemplo de modificación de la semántica de un lenguaje para obtener la movilidad de los objetos, es la modificación de un método que recibe una dirección física para evaluar su invocación como la migración del objeto receptor a la dirección física pasada.

#### 7.2.2.5 Técnicas de Procesamiento de Lenguajes.

En el campo de los traductores, compiladores e intérpretes la reflectividad ya ha hecho aparición en diversos sistemas. El nivel de reflectividad utilizado es estructural y, en la mayoría de los casos, introspección (capítulo 6.4).

Un ejemplo de utilización de introspección y reflectividad estructural en compiladores es la implementación de depuradores (*debuggers*). La dificultad de poder asociar el código generado por un compilador y el código fuente implicaba la generación de código "intruso" para poder depurar la aplicación. La dificultad de desarrollo y las limitaciones de éstos son superados por la implantación basada en reflectividad.

La programación de un depurador en una plataforma reflectiva, se limita a ejecutar un hilo (*thread*) al mismo tiempo que la aplicación que está siendo apurada. Mediante introspección se pueden conocer los valores del estado de la aplicación. Mediante reflectividad estructural se pueden modificar éstos. Estas dos posibilidades las proporciona Smalltalk-80 con el mensaje *inspect* [Mevel87].

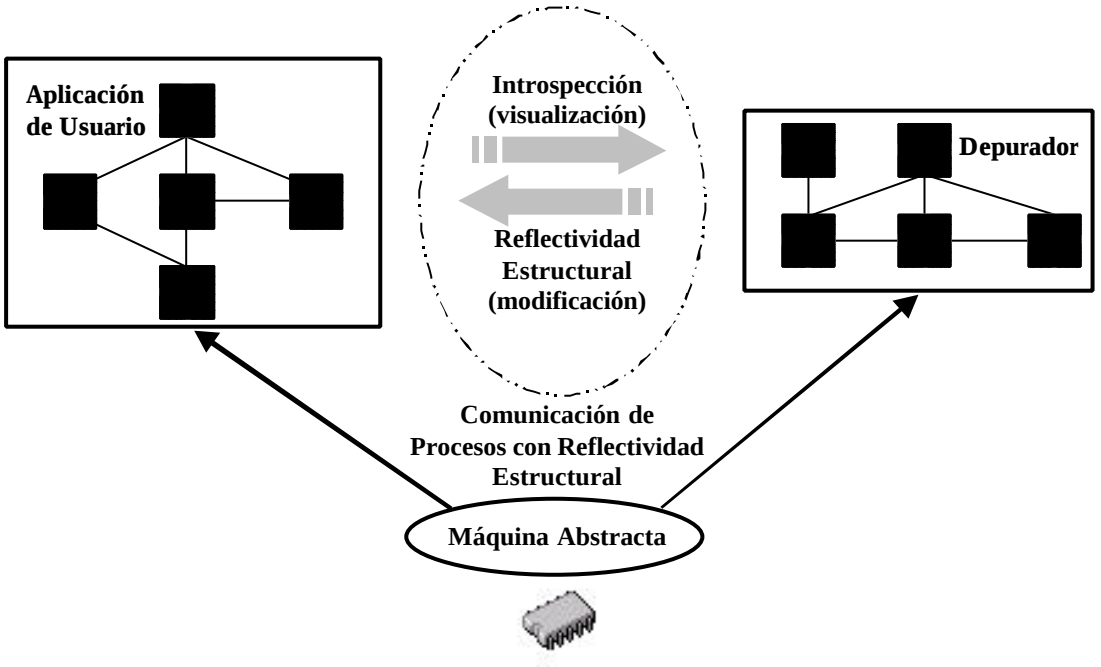


**Figura 39: Introspección y reflectividad estructural en el desarrollo de un depurador.**

En el tema de compilación, la reflectividad estructural se ha utilizado también en Smalltalk-80 de otras maneras. Cuando compilamos normalmente una aplicación C++ [Stroustrup98], debemos tener las declaraciones de las librerías para que el compilador no de errores. En el caso de Java, el problema es el mismo aunque el enlace o *linkado* se realiza de forma dinámica.

Cuando utilizamos cualquier librería tenemos que añadir las declaraciones de ésta. En el campo de la distribución la compilación de accesos remotos es peor. Para realizar llamadas estáticas es necesario obtener apoyos o *stubs* para las invocaciones remotas [OMG95, Sun97b]. Smalltalk-80 resuelve estos problemas mediante reflectividad estructural.

El compilador de Smalltalk-80 está implementado en este lenguaje. Así puede hacer uso de la reflectividad estructural que posee la máquina virtual. Parte de la tabla de símbolos del compilador [Cueva92b] se obtiene en tiempo compilación[34] gracias a la reflectividad estructural. En concreto la existencia de cualquier objeto, su estructura y comportamiento se conocen gracias a la reflectividad estructural sin necesidad de introducirlo en la tabla de símbolos.

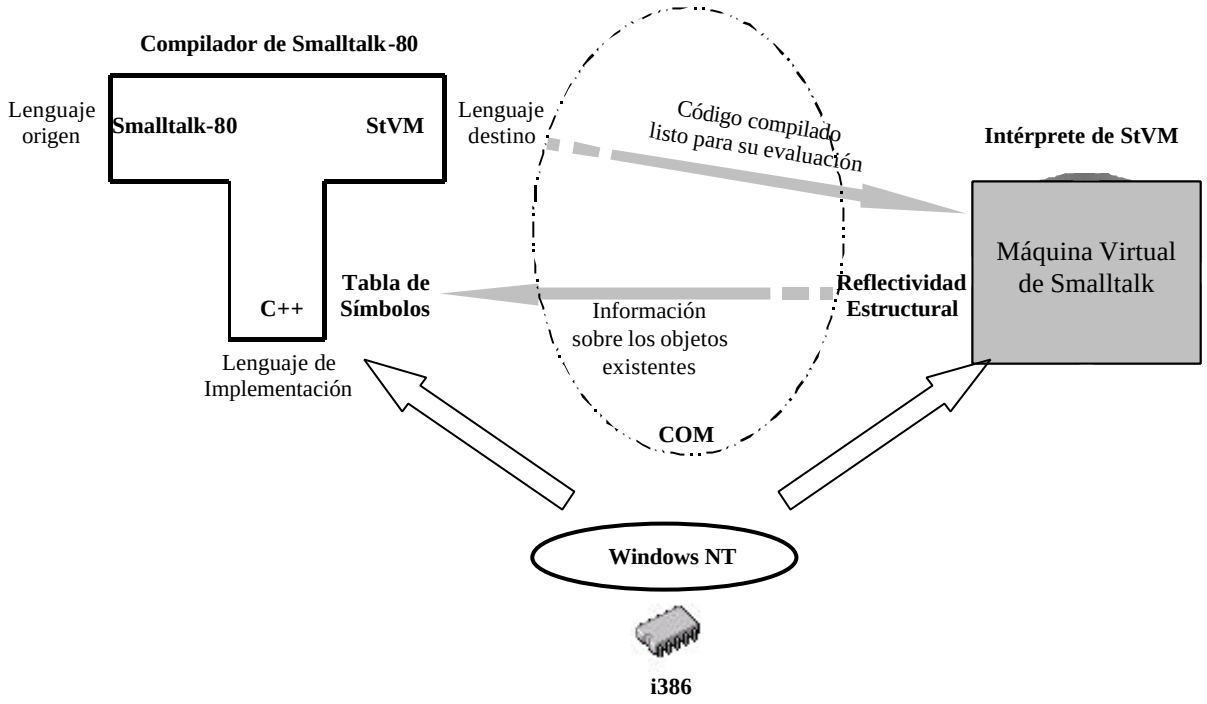


**Figura 40: Utilización de reflectividad estructural en el desarrollo de compiladores.**

De forma similar a la compilación llevada a cabo en Smalltalk-80, se está implementando un compilador con tabla de símbolos reflectiva. Su estructura está reflejada en la Figura 40. Sobre una plataforma se desarrollan un intérprete de la máquina de Smalltalk y un compilador de este lenguaje. El intérprete y el compilador intercambian información mediante un protocolo de comunicación entre componentes COM [Microsoft95].

El compilador genera código binario para la máquina virtual de Smalltalk y se lo pasa al intérprete para que la evalúe. Para llevar a cabo la compilación, es necesario conocer la estructura de los objetos existentes. En lugar de obtener esta información de la tabla de símbolos, el compilador se la pide al intérprete. Éste la adquiere a su vez gracias a la reflectividad estructural de la máquina.

Como en el resto de los campos propuestos, la utilización de la reflectividad está limitada al nivel estructural. La computacional está por investigar, sobre todo en este apartado de procesamiento de lenguajes. En el campo de los intérpretes se puede utilizar para el desarrollo de compiladores bajo demanda (*Just In Time Compilers*) [Yellin96] o la compilación dinámica adaptable [Auslander96].

En el campo de la compilación, la modificación de la semántica del lenguaje implica la modificación de los distintos módulos de un compilador [Cueva92]. Esta ventaja proporcionada por la reflectividad computacional podrá ser utilizada para el desarrollo de herramientas genéricas de desarrollo de compiladores.

[34] En tiempo de ejecución si utilizamos el punto de vista de identificar un compilador como cualquier otro programa.

## 7.3 Implantación de una máquina abstracta

Hemos comentado en diversas ocasiones las ventajas de utilizar una máquina abstracta frente al código nativo de una plataforma. Además la reflectividad computacional la hemos identificado sobre una máquina abstracta. Sin embargo, no todo son ventajas. La utilización de máquinas abstractas reduce la eficiencia de los sistemas de computación.

El hecho de interpretar una plataforma desde otra por medio de un procesador software, añade un nivel de computación al sistema con la consecuente pérdida de eficiencia (apartado 2.2). Esta falta de eficiencia es la principal desventaja del uso de máquinas abstractas.

Existen diversos estudios que se están llevando a cabo para conseguir una mayor velocidad de las aplicaciones desarrolladas sobre máquinas abstractas. Los campos de investigación van desde técnicas de compilación hasta implantaciones *hardware*.

Enlazando con la primera línea de investigación, abriremos una serie de vías de investigación dedicadas a la eficiencia computacional de la máquina abstracta definida.

### 7.3.1 Estudios de Eficiencia.

Aunque hemos dicho que la utilización de máquinas abstractas implica generalmente una menor eficiencia, existe la posibilidad de obtener un sistema computacionalmente más eficiente que los existentes en la actualidad. Como identificamos en 2.1, el proceso de la especificación de la máquina abstracta puede ser *software* o *hardware*.

De esta forma, podemos establecer una serie de pruebas en los pasos del diseño de la máquina abstracta para alcanzar, como objetivo de la investigación, un sistema computacionalmente más eficiente. Los pasos que identificamos en el estudio de eficiencia son:

Estudios con otras máquinas abstractas de características similares. Pasa por comparar el modelo de objetos y el código nativo de otras máquinas estudiando que criterios pueden ser más eficientes.

Estudios con compiladores basados en el mismo paradigma. Se trata de investigar técnicas avanzadas de compilación, interpretación y optimización de código orientado a objetos. Observar cómo se amoldan los existentes a la máquina diseñada.

Estudios con otros paradigmas de programación. Se trata de hacer comparaciones de los lenguajes compilados a código nativo de un procesador existente frente a la ejecución del código de la máquina en un procesador físico (*hardware*).

A continuación profundizaremos en estos puntos.

### 7.3.2 Eficiencia en el Procesamiento de Máquinas Abstractas.

El modelo de objetos y el juego de instrucciones elegido en el diseño de una máquina abstracta, va a determinar en gran parte la eficiencia de su procesamiento.

Respecto al modelo de objetos, son clásicos los estudios de comparación de ejecución de máquinas de un modelo complejo de objetos como Smalltalk-80 [Goldberg83] y el simple Self [Ungar87]. En [Chambers91] se realizó un estudio comparativo en el que entraban en juego Self y Smalltalk (entre otros).

Las ejecuciones de un conjunto de programas programados en Smalltalk-80 y en Self-91 fueron medidas y la ejecución de Smalltalk resultó ser aproximadamente el 20% más lenta que la ejecución de Self. La del Self resultó ser aproximadamente el 50% más lenta que la aplicación codificada en C y compilada con optimización de código. Se han llegado a diseñar intérpretes de Smalltalk-80 sobre Self, más eficientes que la propia máquina de Smalltalk.

La eficiencia de identificar un modelo de objetos simple, como el modelo basado en prototipos utilizado por la máquina de Self, parece resultar más eficiente para el diseño de un modelo de objetos de bajo nivel.

El otro criterio de diseño pasa por la elección de un juego de instrucciones complejo o reducido. Esta diferencia de ideas es la propia de las arquitecturas CISC y RISC existentes en los microprocesadores actuales. Básicamente las diferencias son:

Complejo juego de instrucciones: La máquina de Java está dotada de un gran número de instrucciones. Identifica en ellas los tipos simples para mejorar en eficiencia. La complejidad de interpretar estas instrucciones es llevada a cabo por un intérprete que se ejecuta en código nativo. Su problema viene en la implementación física: es difícil la implantación de un elevado número de instrucciones y casi imposible su optimización.

Reducido juego de instrucciones: El compilador es el que lleva la carga de la complejidad. El bajo nivel con pocas instrucciones permite la implantación sencilla en *hardware* y la optimización de la misma.

La investigación y evaluación de criterios de diseño en este nivel están orientados a otorgar de eficiencia a la máquina, frente a sus competidoras.

### 7.3.3 Técnicas de Compilación e Interpretación.

Los lenguajes de programación basados en el paradigma de la orientación a objetos acercan la máquina al problema, tratando de modelar el *software* de forma similar al sistema real [Booch84]. El código generado, sin embargo, es compilado a las mismas arquitecturas de computadores basadas en el modelo de *Von Neumann* [Myers82].

Para pequeños programas, la codificación de un programa en un lenguaje orientado a objetos produce substancialmente una mayor cantidad de código que si lo programamos en un lenguaje estructurado [Cueva92]. La misma comparación se puede realizar entre los lenguajes estructurados y el ensamblador o código máquina de la plataforma destino.

Una mayor cantidad de código genera ejecutables mayores y menos eficientes en ejecución. Cabe puntualizar que la difícil tarea encomendada al compilador de traducir código en dos niveles de abstracción tan distintos, implica una menor eficiencia en velocidad y tamaño del código generado –frente al que se hubiese escrito directamente sobre el microprocesador.

Es en este punto en el que debemos estudiar las posibilidades de compilación de un código a nuestra máquina frente a otros compiladores (Java, C++, etc.) y las posibilidades de optimización en la interpretación del código generado frente a otras máquinas abstractas (Java, Smalltalk, Self, etc.). Así mismo, existen técnicas de optimización que utilizan la unión de dos conceptos y en las cuales se está investigando.

**Técnicas de compilación**. Son las más estudiadas y, hoy en día, más documentadas y utilizadas [Chambers92]. Las principales características de los lenguajes orientados a objetos que mejoran la duración de las fases de desarrollo del software, generan también dificultades en la compilación y código menos eficiente. Así el estudio las características propias de los lenguajes orientados a objetos han servido para identificar puntos de optimización en la generación de código [Dean96].

Ejemplos básicos de posibilidades de optimización son la substitución de llamada a un método selector por su código (*method inlining*) [Stroustrup98], estudiada a partir del encapsulamiento de datos, o la generación de invocación estática de métodos que no tienen posibilidad de evaluarse mediante enlace dinámico, estudiada a partir del polimorfismo.

Estudios más avanzados se han llevado a cabo. Vortex [Dean96] es un proyecto de optimización de cualquier lenguaje orientado a objetos basándose en las características comunes. Se aprecia cómo las técnicas de optimización son más eficientes en lenguajes orientados a objetos puros que en los híbridos como C++.

**Técnicas de interpretación**. Este estudio está completamente abierto por su gran dependencia del lenguaje interpretado. Cada máquina abstracta tiene sus posibilidades de optimización en la interpretación de su código nativo aunque existen técnicas comunes. El estudio de éstas puede aprovecharse para optimizar la ejecución de las aplicaciones diseñadas para la máquina abstracta en un nuevo grado.

Tomando como ejemplo a la máquina de Java, un primer nivel de optimización en la interpretación de su código es la modificación de determinadas instrucciones a su notación `quick` [Lindholm96]. Esta optimización se centra básicamente en modificar instrucciones dinámicamente cuando se haya resuelto su indireccionamiento en el "*constant pool*". Otras optimizaciones como representar la información de reflectividad como objetos, eliminar un nivel de indireccionamiento en *handles* de objetos (*Handleless Objects*) o implantar mejoras en los algoritmos de recolección de basura se están investigando para presentar un nuevo intérprete de la plataforma de Java: *The HotSpot Virtual Machine* [Sun98].

**Técnicas de optimización híbridas**. Es probablemente en este punto de investigación donde mejores resultados se están obteniendo en la optimización de la utilización de máquinas abstractas. El uso e investigación de estas técnicas presumen un éxito comercial de este tipo de plataformas.

El denominador común de esta técnicas reside en mezclar los conceptos de compilación e interpretación. El tiempo de compilación no es estrictamente anterior al de compilación sino que se entrelazan. Los ejemplos de técnicas más conocidas son las siguientes:

Compilación dinámica (*Dynamic Compilation*). Se trata de realizar las compilaciones al código nativo de la máquina bajo demanda de un intérprete [Chambers92]. El resultado es que al principio de la interpretación de un programa la ejecución es lenta y paulatinamente va acelerando.

Compilación dinámica adaptable (*Adaptive Dynamic Compilation*) [Hölzle94]. Es una modificación de la anterior. En tiempo de ejecución se realizan estudios de los fragmentos de código más utilizados (*Hot Spots*). Una vez detectados, se va realizando optimizaciones de código en tiempo de ejecución.

Compiladores nativos bajo demanda (*Just In Time Compilers*). Son una variación de los primeros. El código es compilado a la máquina nativa en lugar de a la máquina abstracta. En tiempo de ejecución se produce una extensa comunicación entre el procesador de la máquina virtual y el código nativo generado bajo demanda [Yellin96].

### 7.3.4 Eficiencia del Sistema de Computación. Implantación Hardware.

El punto final en el estudio de las posibilidades de eficiencia de utilización de una plataforma abstracta está guiada por la comparación de los sistemas existentes. Dicha comparación será realizada con los distintos lenguajes de programación y paradigmas utilizados, compilados a plataformas físicas existentes.

El hecho de interpretar una plataforma desde otra por medio de un procesador software, añade un nivel de computación al sistema con la consecuente pérdida de eficiencia (apartado 2.2). Si además añadimos otro nivel para conseguir reflectividad computacional –como se había propuesto en 7.2.1 –tenemos un sistema con una naturaleza limitada, en lo que

mo se había propuesto en 7.2.1 –tenemos un sistema con una naturaleza limitada, en lo que a eficiencia se refiere, frente a los existentes.

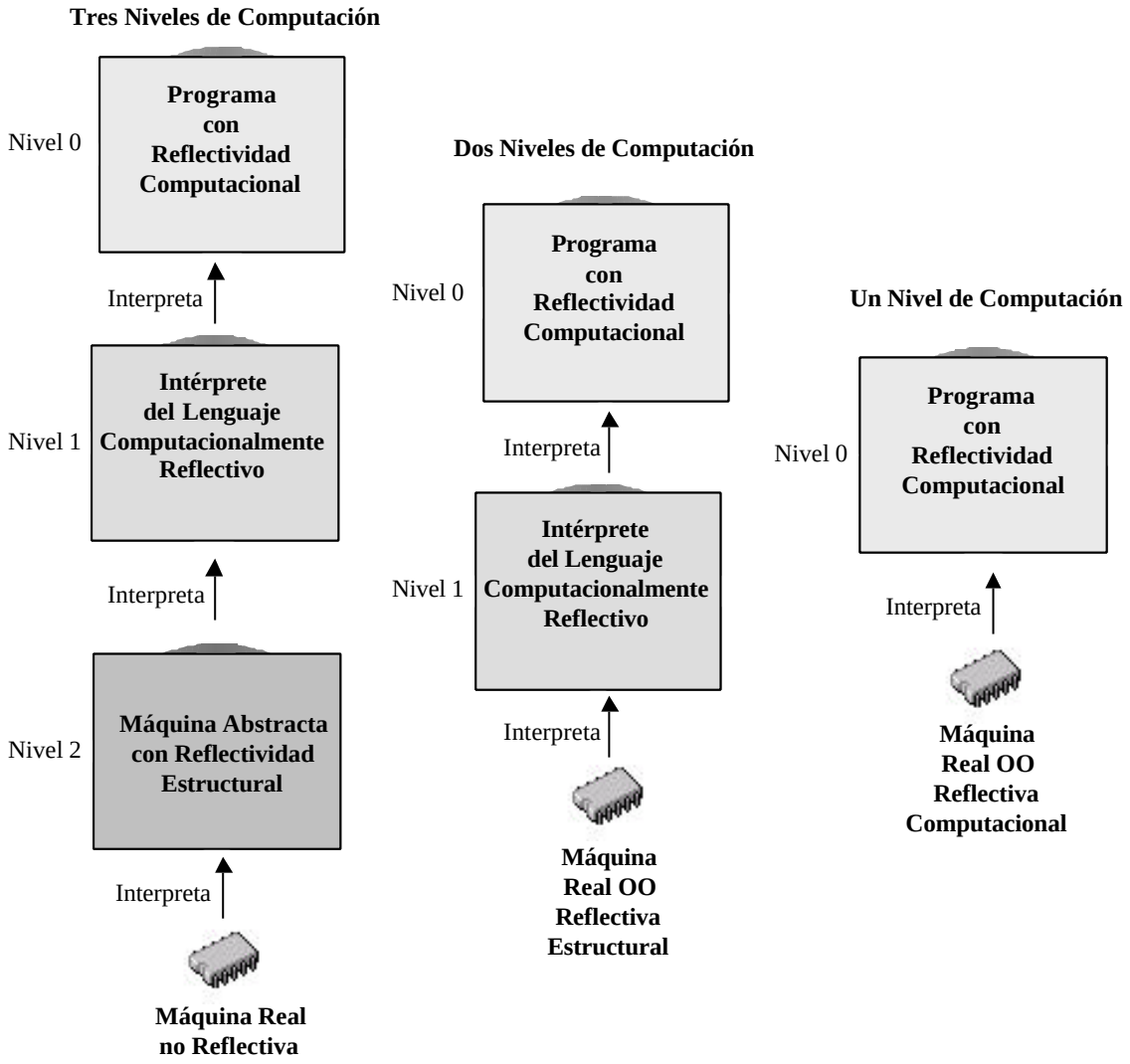


**Figura 41: Distintos escenarios en las optimizaciones de los niveles de computación.**

Podemos apreciar en la Figura 41 cómo tenemos una limitación a priori por la creación de una torre de intérpretes. La primera optimización sería implantar el procesador de la máquina con reflectividad estructural en *hardware*. La segunda necesitaría una implementación física con capacidad de llevar a cabo reflectividad computacional.

La implementación de micros basados en el paradigma de objetos no es una idea nueva [Myers82]. Existen estudios sobre la necesidad de crear plataformas físicas basadas en el paradigma de la orientación a objetos [Hölzle95] así como intentos comerciales de lanzamiento de procesadores con implantaciones propias de este paradigma que fracasaron [Colwel88].

Actualmente Sun Microsystems está investigando y lanzando prototipos de implantaciones *hardware* de su procesador de Java [Sun97]. Una vez llegado al triunfo de la plataforma mediante sus procesadores software, está por ver si es viable la comercialización de un procesador físico.

La segunda proposición de implementación requiere la construcción de una máquina real con la característica de reflectividad computacional. Para modificar la semántica de ins-

trucciones de una plataforma física, debemos apostar por un procesador microprogramable como señalábamos en 2.1.

El micro-código implementa sobre una serie de primitivas la interpretación física de las instrucciones modificables mediante reflectividad estructural. Las primitivas, sin embargo, no permitirán su modificación para obtener así un sistema seguro[35] [Álvarez98]. Este diseño se basa en implementación de la torre de dos niveles propuesta en 7.2.1 en el ámbito *hardware*.

## 7.4 Sistema Integral Orientado a Objetos.

Como se ha mencionado en el apartado 4.1 de este trabajo, el proyecto de investigación Oviedo3 desarrolla estudios para aplicar las tecnologías orientadas a objetos a distintos campos dentro de la informática para conseguir un sistema integral orientado a objetos [Cueva96].

El desarrollo del sistema de computación propuesto en la primera línea de investigación está enfocado también a soportar todos los módulos del sistema integral. Existen varios prototipos de máquinas abstractas [Izquierdo96, Ortin97] que se han utilizado para diversos estudios, demandando una plataforma con las características propuestas en 7.1.3.

Se están investigando en el proyecto Oviedo3 distintos módulos del sistema integral como el desarrollo de un sistema operativo, motor de bases de datos, procesadores de lenguajes, interfaces de usuario, métodos de análisis y diseño y modelos de componentes, todos ellos orientados a objetos. Al respecto hay publicadas dos tesis doctorales [Alvarez98, Perez99].

Esta línea de investigación demasiado amplia como para mencionarla de una forma detallada. Las distintas posibilidades de cada capa del sistema requieren un estudio propio y complejo, sin embargo, el diseño de la máquina abstracta ha de tener en cuenta los requerimientos necesarios para el desarrollo del sistema integral.

[35] La reflectividad computacional ha de tener un control para poder definir un sistema computacional seguro. Este control pasa por definir una serie de primitivas no modificables como se comenta en [Foote90].

# Apéndice A:

# Introspección en la Plataforma Java

A continuación se muestra un ejemplo de la utilización del paquete `java.lang.reflect` de Java [Sun97d]. Muestra cómo se puede realizar una introspección como la que se realiza para la utilización de componentes *Beans* [Sun96].

En el ejemplo se pide por la entrada estándar una clase del API de Java. Se muestra por la salida estándar la información proporcionada por el API de reflectividad. A continuación se crea una instancia de esta clase invocando a un constructor de parámetros. Si la clase no posee éste, la ejecución finalizará en este punto. Finalmente se muestra información de la instancia creada.

La información que se muestra para una clase es:

El nombre de la clase.

Los valores de los modificadores `public`, `abstract` y `final`.

El nombre de la clase ancestra.

Sus interfaces.

Los nombres y tipos de atributos públicos.

La signatura y nombre de los métodos públicos.

Para los objetos:

Muestra los valores de los atributos públicos para el objeto.

Invoca al método público `toString()` heredado de la clase `Object`.

Ejemplo de ejecución para la clase **`java.lang.Object`**:

```
Introduzca una clase a analizar: java.lang.Object

Introspeccion Clase:
--------------------
  Clase: java.lang.Object
  Public: SI.
  Abstract: NO.
  Final: NO.
  SuperClase: La clase Object no tiene ancestra.
  Interfaces:
  Atributos:
  Metodos:
    java.lang.Class getClass( )
    int hashCode( )
    boolean equals( java.lang.Object )
    java.lang.String toString( )
    void notify( )
    void notifyAll( )
    void wait( long )
    void wait( long int )
    void wait( )

Introspeccion Objeto:
---------------------
  Creacion de una Instancia sin parametros: OK.
  Valores de los atributos:
```

```
  Invocacion del metodotoString:
    Resultado: java.lang.Object@20a85b
```

Ejemplo de ejecución para la clase **`java.awt.Button`**:

```
Introduzca una clase a analizar: java.awt.Button

Introspeccion Clase:
-------------------
  Clase: java.awt.Button
  Public: SI.
  Abstract: NO.
  Final: NO.
  SuperClase: java.awt.Component
  Interfaces:
  Atributos:
    int WIDTH
    int HEIGHT
    int PROPERTIES
    int SOMEBITS
    int FRAMEBITS
    int ALLBITS
    int ERROR
    int ABORT
    float TOP_ALIGNMENT
    float CENTER_ALIGNMENT
    float BOTTOM_ALIGNMENT
    float LEFT_ALIGNMENT
    float RIGHT_ALIGNMENT
  Metodos:
    java.lang.Class getClass( )
    int hashCode( )
    boolean equals( java.lang.Object )
    java.lang.String toString( )
    void notify( )
    void notifyAll( )
    void wait( long )
    void wait( long int )
    void wait( )
    java.lang.String getName( )
    void setName( java.lang.String )
    java.awt.Container getParent( )
    .
    .
    .

Introspeccion Objeto:
--------------------
  Creacion de una Instancia sin parametros: OK.
  Valores de los atributos:
    WIDTH = 1
    HEIGHT = 2
    PROPERTIES = 4
    SOMEBITS = 8
    FRAMEBITS = 16
    ALLBITS = 32
    ERROR = 64
    ABORT = 128
    TOP_ALIGNMENT = 0.0
    CENTER_ALIGNMENT = 0.5
    BOTTOM_ALIGNMENT = 1.0
    LEFT_ALIGNMENT = 0.0
    RIGHT_ALIGNMENT = 1.0
  Invocacion del metodotoString:
    Resultado: java.awt.Button[button0,0,0,0x0,invalid,label=]
```

Finalmente mostraremos la implementación de tres clases que obtienen los resultados mostrados anteriormente.

```
/*************************************************************************/
// * Ejemplo de Utilización del paquete java.lang.reflect.
// * Ejemplo de Introspección en Java.
// * Lenguajes y Sistemas Informáticos.
// * Departamento de Informática.
// * Universidad de Oviedo.
// * Trabajo de Investigación.
// * Francisco Ortín Soler.
// * Mayo, 1999.
/*************************************************************************/

// * Ejemplo de Introspección en Java

import java.lang.reflect.*;
import java.io.InputStreamReader;
import java.io.BufferedReader;

// * Clase que muestra las características de
//   objetos y clases en tiempo de ejecución
class Introspeccion {

  public static void main(String args[]) {
    System.out.println("\n\n\n\n");
    System.out.print("Introduzca una clase a analizar: ");
    String s=null;

    try {
      s = new BufferedReader(new InputStreamReader(System.in)).readLine();
      IntrospeccionClase clase=new IntrospeccionClase( Class.forName(s) );
      System.out.println("\n\nIntrospeccion Clase:");
      System.out.println("-------------------");
      clase.mostrar();
      }
    catch (Exception e) {
      System.out.println(e);
      System.exit(-1);
      }

    System.out.println("\nIntrospeccion Objeto:");
    System.out.println("--------------------");
    try {
      IntrospeccionObjeto objeto=new IntrospeccionObjeto(
             Class.forName(s).newInstance() );
      System.out.println("  Creacion de una Instancia sin parametros: OK.");
      objeto.mostrarValoresAtributos();
      objeto.invocarMetodoSinParametros("toString");
      }
    catch (Exception e) {
      System.out.println(e);
      System.exit(-1);
      }
    }

  }

/*************************************************************************/

// * Clase que muestra las características de
//   clases en tiempo de ejecución
class IntrospeccionClase {

  // * Objeto identificativo de Clase
  private Class clase;

  public IntrospeccionClase(Class c) {
    clase=c;
    }
```

```
  public void mostrar() {
    mostrarClase();
    mostrarModificadores();
    mostrarSuperclase();
    mostrarInterfaces();
    mostrarAtributos();
    mostrarMetodos();
    }

  public void mostrarClase() {
    System.out.println( "  Clase: " + clase.getName() );
    }

  public void mostrarModificadores() {
    int m=clase.getModifiers();
    System.out.print("  Public: ");
    if ( Modifier.isPublic(m) )
      System.out.println("SI.");
    else
      System.out.println("NO.");
    System.out.print("  Abstract: ");
    if ( Modifier.isAbstract(m) )
      System.out.println("SI.");
    else
      System.out.println("NO.");
    System.out.print("  Final: ");
    if ( Modifier.isFinal(m) )
      System.out.println("SI.");
    else
      System.out.println("NO.");
    }

  public void mostrarSuperclase() {
    Class superClase=clase.getSuperclass();
    if (superClase!=null)
      System.out.println("  SuperClase: " + superClase.getName() );
    else
      System.out.println("  SuperClase: La clase Object no tiene ancestra.");
    }

  public void mostrarInterfaces() {
    Class interfaces[]=clase.getInterfaces();
    System.out.println("  Interfaces:");
    for (int i=0;i<interfaces.length;i++)
      System.out.println( "    " + interfaces[i].getName() );
    }

  public void mostrarAtributos() {
    Field atributos[]=clase.getFields();
    System.out.println("  Atributos:");
    for (int i=0;i<atributos.length;i++) {
      String nombre=atributos[i].getName();
      String tipo=atributos[i].getType().getName();
      System.out.println( "    " + tipo + " " + nombre );
      }
    }

  public void mostrarMetodos() {
    Method metodos[]=clase.getMethods();
    System.out.println("  Metodos:");
    for (int i=0;i<metodos.length;i++) {
      System.out.print( "    " + metodos[i].getReturnType().getName() );
      System.out.print( " " + metodos[i].getName() + "( " );
      Class parametros[]=metodos[i].getParameterTypes();
      for (int j=0;j<parametros.length;j++)
        System.out.print( parametros[j].getName() + " " );
      System.out.println(")");
      }
```

```
    }

  }

/**************************************************************************/

// * Clase que muestra las características de
//   objetos en tiempo de ejecución
class IntrospeccionObjeto {

  // * Objeto con el que trabajamos
  private Object objeto;

  public IntrospeccionObjeto(Object o) {
    objeto=o;
    }

  public void mostrarValoresAtributos() {
    Field atributos[]=objeto.getClass().getFields();
    System.out.println("  Valores de los atributos:");
    for (int i=0;i<atributos.length;i++) {
      try {
        System.out.print( "    " + atributos[i].getName() + " = " );
        System.out.println( atributos[i].get(objeto).toString() );
        }
      catch (Exception e) {
        System.out.println(e);
        System.exit(-1);
        }
      }
    }

  public void invocarMetodoSinParametros(String m) {
    System.out.println("  Invocacion del metodo" + m + ":" );
    try {
      Method metodo=objeto.getClass().getMethod(m,null);
      String resultado=(String)metodo.invoke(objeto,null);
      System.out.println("    Resultado: " + resultado );
      }
    catch (Exception e) {
      System.out.println(e);
      System.exit(-1);
      }
    }

  }

/**************************************************************************/
```

# Apéndice B:

# EJEMPLO DE REFLECTIVIDAD COMPUTACIONAL

Como apéndice del documento mostraremos un ejemplo de reflectividad computacional basándonos en la torre de dos niveles propuesta en 7.2.1.

El lenguaje que vamos a hacer computacionalmente reflectivo es un simple evaluador de expresiones aritméticas. Las expresiones son introducidas por teclado con la posibilidad de utilizar los operadores +, -, * y /. La precedencia sintáctica de los operadores es igual para todos y se resuelve por su asociatividad a derechas [Cueva95]. Para modificar la precedencia por omisión se puede utilizar el uso de paréntesis.

La torre de intérpretes propuesta hemos identificado dos características fundamentales a la hora de seleccionar el nivel mayor (en nuestro caso el lenguaje de programación del evaluador):

Posibilidades de reflectividad estructural.

Posibilidad de realizar el "salto" entre niveles o cosificación.

Buscando estos requisitos, la selección realizada se ha inclinado por el lenguaje de programación funcional Lisp [Steele90]. El segundo punto está proporcionado por la funcionalidad de las funciones `eval` y `apply` que son capaces de interpretar estructuras de datos como computación.

La carencia de este lenguaje a la hora de implementar el prototipo reside en el primer requerimiento. Lisp no es un lenguaje de naturaleza reflectiva y por lo tanto la codificación del evaluador no sigue una "elegancia" típica de la programación funcional[36]. La única característica que hemos podido utilizar es la posibilidad de redefinir una función mediante `defun`. Las posibilidades de la reflectividad computacional de este lenguaje están limitadas a la reflectividad estructural del Lisp.

Finalmente, comentar que la sintaxis de reflectividad viene dado con la palabra reservada **`reification`**. Así podremos pasarle al intérprete código del lenguaje de expresiones o código Lisp precedido de **`reification`**. El intérprete se encarga de realizar el salto computacional.

A continuación vamos a poner un ejemplo de su utilización:

Se introduce una expresión para que sea evaluada por el intérprete. Vemos cómo el resultado es un número racional resultante de aplicar el operador de división.

Se modifica computacionalmente el lenguaje. Ahora la división ya no es racional sino entera.

Comprobamos cómo la misma expresión introducida en el primer paso es evaluada de forma distinta. Una misma instrucción del lenguaje se interpreta de distintas formas. Hemos variado la semántica del lenguaje gracias a la reflectividad computacional.

[36] Ejemplo de esto es la codificación de dos funciones `evaluarBinario` y `operacionBinaria` que realizan la misma computación. Se ha programado así para poder realizar trazas mediante reflectividad computacional.

Ahora vamos a modificar computacionalmente el intérprete para que realice trazas: Cualquier evaluación de una subexpresión binaria será visualizada secuencialmente. Para conseguir esto hemos modificado la función `evaluarBinario.`

Vemos cómo al evaluar la expresión inicial se comporta con división entera y mostrando las secuencias de evaluaciones realizadas.

La salida de esta ejecución en el intérprete fue la siguiente:

```
> (interprete)

" *******************************"

" * Evaluador de Expresiones dotado de Reflectividad Computacional."
" * Operadores binarios +, -, * y /."
" * Posibilidad del uso de paréntesis. "
" * Reification -> Cosificación. "
" * Final de Evaluación: T o Nil. "
" *******************************"
* (8 + (2 - 3 * 2) / 3)
" * Resultado de la Evaluación: " 20/3
* (reification defun dividir (op1 op2)
      (round (/ op1 op2)))
" * Resultado de la Evaluación: " REFLECTION
* (8 + (2 - 3 * 2) / 3)
" * Resultado de la Evaluación: " 7
* (reification defun evaluarBinario (op1 op op2)
      (progn
          (print 'TRAZA_SUBEXPRESION)
          (prin1 op1)
          (prin1 op)
          (prin1 op2)
          (prin1 '=)
          (prin1 (operacionBinaria op1 op op2))))
" * Resultado de la Evaluación: " REFLECTION
* (8 + (2 - 3 * 2) / 3)
TRAZA_SUBEXPRESION 3*2=6
TRAZA_SUBEXPRESION 2-6=-4
TRAZA_SUBEXPRESION -4/3=-1
TRAZA_SUBEXPRESION 8+-1=7
" * Resultado de la Evaluación: " 7
* nil
" * Final de Evaluación. "
>
```

Mostramos a continuación la implementación realizada del intérprete de expresiones aritméticas sobre, a su vez, un intérprete de lenguaje Lisp:

```
; ***********************************************************************

; * Ejemplo de Reflectividad Computacional.

; * Intérprete de Expresiones Aritméticas Simples.

; * Operadores de Igual Precedencia. Asociatividad a Derechas.
; * Se permite el uso de Paréntesis para Cambiar Precedencias.
; * Lenguajes y Sistemas Informáticos.
; * Departamento de Informática.
; * Universidad de Oviedo.
; * Trabajo de Investigación.
; * Francisco Ortín Soler.
; * Mayo, 1999.
; ***********************************************************************
; * Intérprete del Lenguaje. Entrada Estándar.
; * Se Limita a Llamar a la Función InterpretarInstruccion
;    hasta que se Introduzca nil o T.
(defun interprete ()
```

```
      (progn
          (print " ******************************")
          (print " * Evaluador de Expresiones dotado de Reflectividad Computacio-
nal.")
          (print " * Operadores binarios +, -, * y /.")
          (print " * Posibilidad del uso de paréntesis. ")
          (print " * Reification -> Cosificación. ")
          (print " * Final de Evaluación: T o Nil. ")
          (print " ******************************")
          (print '*)
          (do ((L (read) (read)))
               ((or (equal L 'T) (null L)) " * Final de Evaluación. ")
              (let ((resultado (interpretarInstruccion L)))
                  (progn
                      (print " * Resultado de la Evaluación: ")
                      (prin1 resultado)
                      (print '*))))
          ))

; * Intérprete de Instrucciones.
; * Recibe una instrucción como parámetro.
; * La instrucción puede ser una evaluación o
;   una llamada a la reflectividad computacional
(defun interpretarInstruccion (L)
     (cond
              ((numberp L) L)
              ((atom L) nil)
              ((equal (car L) 'Reification) (progn
                                     (eval (cdr L))
                                     'Reflection))
               (T (evaluar L))))

; ************************************************************************
; * Evaluador de Expresiones con Varios Operadores y Paréntesis
(defun evaluar (L)
     (cond
              ((atom L) L)
              ((null (cadr L)) (evaluar (car L)))
              (T (evaluarBinario (evaluar (car L)) (cadr L) (evaluar (cddr
L))))))

; ************************************************************************
; * Evaluador de Expresiones Binarias
(defun evaluarBinario (op1 op op2)
     (operacionBinaria op1 op op2))
(defun operacionBinaria (op1 op op2)
     (cond
                ((equal op '+) (sumar op1 op2))
                ((equal op '-) (restar op1 op2))
                ((equal op '*) (multiplicar op1 op2))
                ((equal op '/) (dividir op1 op2))
                (T op1)))

; ************************************************************************
; * Operaciones Primitivas del Evaluador de Expresiones
(defun sumar (op1 op2)
     (+ op1 op2))

(defun restar (op1 op2)
     (- op1 op2))

(defun multiplicar (op1 op2)
     (* op1 op2))

(defun dividir (op1 op2)
     (/ op1 op2))
```

# Apéndice C:

# Referencias Bibliográficas


[Aksit92] Mehmet Aksit, Lodewijk Bergmans, Sinan Vural. An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. In O. Lehrman Madsen editor, *European Conference on Object-Oriented Programming ECOOP*. pp. 372-396, Utrecht Holanda. Junio/Julio, 1992.

[Aksit94] Mehmet Aksit, Jan Bosch, William van der Sterren, Lodewijk Bergmans. Real Time Specification Inheritance Anomalies and Real-Time Filters. In mario Tokoro and Remo Pereschi, editors. *European Conference on Object-Oriented Programming ECOOP*. pp 386-407. Bologna, Italia. Julio, 1994.

[Alvarez98] Darío Álvarez Gutiérrez. Persistencia Completa para un Sistema Operativo Orientado a Objetos usando una Máquina Abstracta con Arquitectura Reflectiva. Tesis Doctoral. Departamento de Informática. Universidad de Oviedo. Marzo de 1998.

[Auslander96] Joel Auslander, Matthi Philipose, Craig Chambers, Susan J. Eggers, Brian N. Bershad. Fast, Effective Dynamic Compilation. *In PLDI' 96*. 1996.

[Baillarguet99] Carine Baillarguet, Ian Piumarta. An Highly-Configurable, Modular System Architecture for Mobility, Interoperability, Specialisation and Reuse. *Proceedings of the ECOOP'99 Workshop on Object-Orientation and Operating Systems.* Lisboa, Portugal. 1999.

[Barendregt81] Hendrik Pieter Barendregt. The Lambda Calculus: Its Syntax and Semantics. North-Holland, Amsterdam. 1981.

[Beners93] Tim Beners-Lee. Hypertext Markup Language. HTML. 1993.

[Beners96] Tim Beners-Lee, R. Fielding y H. Frystyk. Hypertext Transfer Protocol – HTTP 1.0. HTTP Working Group. Enero, 1996.

[Bergmans94] Lodewijk Bergmans. Composing Cocurrent Objects. PhD thesis. University of Twente, Enschede, Holanda. Julio, 1994.

[Booch94] Grady Booch. Análisis y diseño orientado a objetos con aplicaciones. Editorial Addison-Wesley / Díaz de Santos. 1994.

[Borland91] Borland International Inc. Borland languages. Open architecture handbook. The Borland Developer's Technical Guide. 1991.

[Borning86] Borning, A.H. Classes Versus Prototypes in Object-Oriented Languages. *In Proceedings of the ACM/IEEE Fall Joint Computer Conference* 36-40. 1986.

[Brodie87] Leo Brodie. Starting Forth: an Introduction to the Forth language and Operating System for Beginners and Professionals. Prentice Hall. 1987.

[Brown98] Nat Brown y Charlie Kindel. Distributed Component Object Model Protocol. DCOM/1.0. Microsoft Corporation. Network Working Group. Enero, 1998.

[Campbell83] F. Campbell. The Portable UCSD p-System. Microprocessors and Microsystems, No. 7. Octubre de 1983. pp. 394-398.

[Campione97] Mary Campione, Kathy Walrath. The Java Tutorial. Object Oriented Pro-

gramming for Internet. The Java Series. Sun Microsystems. 1997.

[Cardelli97] Luca Cardelli. Type Systems. Handbook of Computer Science and Engineering, Chapter 103. CRC Press. 1997.

[Cattell94] R. Cattell. Object Data Management. Object Oriented and Extended Relational Database Systems (Revised Edition ). Addison Wesley, 1994.

[Chambers89] Chambers, C., Ungar, D. and Lee, E. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. *In OOPSLA'89 Conference Proceedings. Published as SIGPLAN Notices*, 24, 10, 49-70. 1989.

[Chambers91] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. *OOPSLA '91 Conference Proceedings*, Phoenix, AZ, Octubre 1991.

[Chambers92] Craig Chambers. The Design and Implementation of the SELF Compiles, and Optimizing Compiler for Object-Oriented Programming Languages. PhD thesis. Universidad de Stanford. Marzo, 1992.

[Chambers93] Craig Chambers. The Cecil Language: Specification and Retionale. Technical Report TR-93-03-05. Department of Computer Science and Engineering. University of Washington, March 1993.

[Charte96] Francisco Charte. Programación con Delphi. 3ª Edición. Anaya Multimedia. 1996.

[Chiba95] Shigeru Chiba. A Metaobject Protocol for C++. *In 10th Conference on Object-Oriented Programming Systems, Languages, and Applications. OOPSLA'95.* 1995.

[Colwel88] Robert P. Colwel, Edward F. Gehringer, E. Douglas Jensen. Performance Effects of Architectural Complexity in the Intel 432. *ACM Transactions on Computer Systems*, Vol. 6, No. 3. Agosto, 1988.

[Cueva92] Juan Manuel Cueva Lovelle. Conceptos Básicos de Traductores, Compiladores e Intérpretes. Cuaderno didáctico número 9. Departamento de Matemáticas. Universidad de Oviedo. Octubre de 1992.

[Cueva92b] Juan Manuel Cueva Lovelle. Tablas de Símbolos en Procesadores de Lenguajes. Cuaderno Didáctico número 54. Departamento de Matemáticas. Universidad de Oviedo. 1992.

[Cueva93] Juan Manuel Cueva Lovelle. Análisis Léxico en Procesadores de Lenguaje. Cuaderno Didáctico número 48. Departamento de Matemáticas. Universidad de Oviedo. 1993.

[Cueva94] J. M. Cueva Lovelle, P.A. García Fuente, B. López Pérez, C. Luengo Díez y M. Alonso Requejo. Introducción a la Programación Estructurada y Orientada a Objetos con Pascal. ISBM: 84-600-8646-1. 1994.

[Cueva95] Juan Manuel Cueva Lovelle. Análisis Sintáctico en Procesadores de Lenguaje. Cuaderno Didáctico número 61. Departamento de Matemáticas. Universidad de Oviedo. 1995.

[Cueva95b] Juan Manuel Cueva Lovelle. Análisis Semántico en Procesadores de Lenguaje. Cuaderno Didáctico número 62. Departamento de Matemáticas. Universidad de Oviedo. 1995.

[Cueva96] Juan Manuel Cueva Lovelle. El Sistema Integral Orientado a Objetos: Oviedo3. *II Jornadas sobre Tecnologías Orientadas a Objetos.* Oviedo, Marzo 1996.

[Cueva98] Juan Manuel Cueva Lovelle. Análisis y Diseño Orientado a Objetos. Cuaderno didáctico número 1. Departamento de Informática, Universidad de Oviedo. Marzo, 1998.

[Dean96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, Craig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. A Research Paper in Language Implementation. University of Washington. 1996.

[Deitel93] Harvey M. Deitel. Introducción a los Sistemas Operativos. Addison-Wesley Iberoamericana. 1993.

[Díaz98] M.A. Díaz, D. Álvarez, A. García-Mendoza, F. Álvarez, L. Tajes, J.M. Cueva. Merging Capabilities with the Object Model of an Object-Oriented Abstract Machine. *Proceedings of the ECOOP'98 Workshop on Distributed Object Security and the 4th Workshop on Mobile Object Systems*, pp. 9-13. Inria Rhône-Alpes, Francia, Julio de 1998.

[Edwards97] Jeri Edwards. 3-Tier Client/Server at Work. Wiley. 1997.

[Evins96] M. Evins. Objects Without Classes. *Computer IEEE*. Vol. 27, N. 3, 104-109. 1994.

[Ferber88] Jacques Ferber. Coceptual Reflection and Actor Ñanguages. Meta-Level Architectures and Reflection. P. Maes and D. Nardi Editors. North-Holland, 1988.

[Ferber89] Jacques Ferber. Computational Reflection in Class Based Object-Oriented Languages. *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications. OOPSLA'89.* New Orleans, USA. Octubre, 1989.

[Finin93] Tim Finin, Jay Weber, Gio. Widerhold. Draft Specification of the KQML Agent Communication Language plus Example of Agent Policies and Architectures. pp. 4, 33. Junio, 1993.

[Flynt98] Clif Flynt. Tcl/Tk for Real Programmers. Academic Press. Diciembre, 1998.

[Folliot98] Bertil Folliot, Ian Piumarta, Fabio Reccardi. Virtual Virtual Machines. Project SOR, INRIA Rocquencourt, B.P. 105, 78153, Les Chesnay Cedex, Francia. 1998.

[Foote90] Brian Foote. Object-Oriented Reflective Metalevel Architectures: Pyrite or Panacea? *ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures.* July, 1990.

[Goldberg83] Goldberg A. y Robson D. Smalltalk-80: The language and its Implementation. Addison-Wesley. 1983.

[Goldberg89] Goldberg A. y Robson D. Smalltalk-80: The language. Addison-Wesley, 1989.

[Golm97] Michael Golm. Design and Impoementation of a Meta Architecture for Java. Friedrich-Alexander-Universität. Computer Science Department. Erlangen-Nürnberg, Germany. Enero, 1997.

[Golm97b] Michael Golm, Jürgen Kleinöder. MetaJava – A Platform for Adaptable Operating – System Mechanisms. *ECOOP '97 Workshop on Object-Orientation and Operating Systems*. Jyväskylä, Finlandia. Junio, 1997.

[Golm97c] Michael Golm, Jügen Kleinöder. Implementing Real-Time Actors with MetaJava. Technical Report TR-14-97-09. Friedrich-Alexander-Universität. Computer Science Department. Erlangen-Nürnberg, Germany. Abril, 1997.

[Gosling96] James Gosling, Bill Joy y Guy Seele. The Java™ Language Specification. Addison-Wesley. 1996.

[Gowing96] Brendan Gowing, Vinny Cahill. Meta-Object Protocols for C++: The Iguana Approach. Distributed Systems Group, Department of Computer Science, Trinity College. Dublin, Irlanda. 1996.

[Hölzle94] Urs Hölzle, David Ungar. A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance. *OOPSLA '94 Conference Proceedings*, Portland, OR. Octubre 1994.

[Hölzle95] Urs Hölzle, David Ungar. Do Object-Oriented Languages Need Special Hardware Support? *ECOOP' 95 Conference Proceedings*, Springer Verlag Lecture Notes on Computer Science 952. Agosto, 1995.

[Hürsch95] Walter L. Hürsch, Cristina Videira Lopes. Separation of Concerns. Technical Report UN-CCS-95-03, Northeastern University, Boston. Enero, 1995.

[Izquierdo96] Raúl Izquierdo Castanedo. Máquina Abstracta Orientada a Objetos. Proyecto Final de Carrera número 9521I. Escuela Técnica Superior de I. I. e Ingenieros Informáticos. Universidad de Oviedo. Septiembre, 1996.

[Jensen91] Jensen K. y Wirth N. PASCAL User Manual and Report ISO Pascal Standard. 4º Edición Springer-Verlag, 1991.

[Kiczales91] Gregor Kiczales, Jim des Rivières, Daniel G. Bobrow. The Art of Metaobject Protocol. MIT Press. 1991.

[Kiczales96] Gregor Kiczales. Beyond the Black Box: Opent Implementations. *IEEE Software.* Vol. 13, No. 1, pp. 8-11. 1996.

[Kiczales96b] Gregor Kiczales, John Lamping, Cristina Videira Lopes, Chris Maeda, Anurag Mendhekar, Gail Murphy. Open Implementation Design Guidelines. *19th International Conference on Software Engineering*. 1996.

[Kiczales96c] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina videira Lopes, jean-Marc Loingtier, John Irwin. Apect-Oriented Programming. *Position Paper for the ACM Workshop on Strategic Directions in Computing Research*, MIT. Junio, 1996.

[Kirtland97] Mary Kirtland. Object-Oriented Software Development Made Simple with COM+ Runtime Services. Microsoft Systems Journal. Noviembre, 1997.

[Kleinöder96] Jürgen Kleinöder, Michael Golm. MetaJava: An Efficient Run-Time Meta Architecture for Java™. Technical Report 14-96-03. Computer Science Department. Friedrich-Alexander-University. Erlangen, Alemania. Junio, 1996.

[Kramer96] Douglas Kramer. The Java Platform. A White Paper. Sun JavaSoft. Mayo 1996.

[Krasner83] Glenn Krasner. Smalltalk-80: Bits of History, Words of Advice. Addison-Wesley. 1983.

[Lange97] Danny B. Lange, Mitsuru Oshima. Programming Mobile Agents in Java – With the Java Aglet API. IBM Research. Mayo, 1997.

[Leavenworth93] Leavenworth B. PROXY: a Scheme-Based Prototyping Language. *Dr. Dobb's Journal*, No. 198, 86-90. March 1993.

[Lieberherr95] K. J. Lieberherr. Adaptive Object-Oriented Software: The Demeter Methods

with Propagation Patterns. PWS Publishing Company. 19995.

[Lieberman86] Lieberman, H. 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.

[Lindholm96] Tim Lindholm y Frank Yellin. The Java™ Virtual Machine Specification. Sun Microsystems. Septiembre, 1996.

[Longshaw99] Andrew Longshaw. Getting Clever with Components. *Application Development Advisor*. Vol. 2. No 4. Marzo / Abril 1999.

[Macrakis93] Stavros Macrakis. Delivering Applications to Multiple Platforms Using ANDF. AIXpert. Agosto 1993.

[Madany96] Peter W. Madany. JavaOS ™: A Standalone Java ™ Environment. A White Paper. JavaSoft. Mayo, 1996.

[Maeda97] Chris Maeda, Arthur Lee, Gail Murphy, Gregor Kiczales. Open Implementation Analysis and Design. *Association for Computing machinery, ACM*. 1997.

[Maes87] Pattie Maes. Computational Reflection. PhD. Thesis. Laboratory for Artificial Intelligence, Vrije Universiteit Brussel. Bruselas, Bélgica. Enero, 1987.

[Maes87b] Pattie Maes. Issues in Computational Reflection. Meta-Level Architectures and Reflection. P Maes and D. Nardi Editors. North-Holland. Bruselas, Bélgica. Agosto, 1987.

[Mandado73] Enrique Mandado. Sistemas Electrónicos Digitales. Marcombo Boixareu Editores. 1973.

[Martínez98] Ana Belén Martínez, Darío Álvarez, Juan Manuel Cueva, Francisco Ortín y Jose Arturo Pérez. Incorporating an Object-Oriented DBMS into an Integral Object Oriented System. Proceedings *4th International Conference on Information Systems, Analysis and Synthesis.* Vol. 2. USA. Julio, 1998.

[Mendhekar96] Anurag Mendhekar, Danel P. Friedman. Towards a Theory of Reflective Programming Languages. Department of Computer Science. Indiana University. USA. 1992.

[Mevel87] Mével A. y Guéguen T. Smalltalk-80. Mac Millan Education, Houndmills, Basingstoke. 1987.

[Meyer92] B. Meyer. Eiffel. The Language. Prentice-Hall. 1992.

[Meyer97] B. Meyer. Object-Oriented Software Construction. 2 Edición. Prentice-Hall. 1997.

[Microsoft95] The Component Object Model Specification. Version 0.9. Microsoft Corporation. Octubre, 1995.

[Microsoft98] Microsoft Transaction Server 2.0. Transactional Component Services. Microsoft Corporation. 1998.

[Mulet93] Philippe Mulet, Pierre Cointe. Definition of a Reflective Kernel for a Prototype-Based *Language. International Symposium on Object Technologies for Advanced Software.* Kanazawa, Japón. Noviembre, 1993.

[Murata95] Kenichi Murata, R. Nigel Horspool, Yasuhiko Yokote, Eric G. Manning, Mario Tokoro. Cognac: a Reflective Object-Oriented Programming System using Dynamic Comilation Techniques. 1995.

[Myers82] G. Myers. Advances in Computer Architecture. Second Edition. New Yor, NY: John Weley and Sons. 1982.

[Nori76] K. V. Nori, U. Ammann, K. Jensen, H. H. Nageli y C. Jacobi. The Pascal-P Compiler: Implementation Notes. Bericht 10, Eidgengössische Technische Hochschule. Zurich, Suiza. Julio 1976.

[OMG95] Object Management Group (OMG). The Common Object Request Broker: Architecture and Specification. Julio, 1995.

[OMG95b] Object Management Group (OMG). CORBA Services Specification. Marzo, 1995.

[OMG95c] Object Management Group (OMG). Systems Management: Commno Management Facilities, Volume1, Version 2. Diciembre, 1995.

[OMG96] Object Management Group (OMG). Description of New OMA Reference Model. Draft 1. Mayo, 1996.

[OMG97] Object Management Group (OMG). A Discussion of the Object Management Architecture. Enero, 1997.

[OMG98] Object Management Group (OMG). CORBA Components. CORBA 3.0 Draft Specification. Noviembre, 1998.

[OMG98b] Object Management Group (OMG). Interworking Sections of CORBA 2.2, OMG Specification. Febrero, 1998.

[Orfali96] Robert Orfali, Dan Harkey y Jeri Edwards. The Essential Client / Server Survival Guide. Second Edición. Wiley. 1996.

[Orfali97] Robert Orfali, Dan Harkey, Jeri Edwards. Instant CORBA. Wiley. 1997.

[Orfali98] Robert Orfali, Dan Harkey. Client/Server Programming with Java and CORBA. 2ª Edición. Wiley. 1998.

[Ortin97] Francisco Ortín Soler. Diseño y Construcción del Sistema de Persistencia en Oviedo3. Proyecto Final de Carrera 972001. Escuela Técnica Superior de I. I. e Ingenieros Informáticos. Universidad de Oviedo. Septiembre, 1997.

[Ortin97b] Francisco Ortín Soler, Darío Álvarez, Raúl Izquierdo, Ana Belén Martínez, Juan Manuel Cueva. El Sistema de Persistencia en Oviedo3. Terceras Jornadas de Tecnologías Orientadas a Objetos. Sevilla, 1997.

[Ortin99] Francisco Ortín Soler, Ana Belén Martínez Prieto, Darío Álvarez Gutiérrez. An Implicit Persistence System on an OO Database Engine Using Reflection. A ser publicado. Mayo, 1999.

[OSF96] Open Software Fundation (OSF). Distributed Computing Environment. Technical Report. 1996.

[Perez98] Jesús Arturo Pérez Díaz, Juan Manuel Cueva Lovelle, Benjamín López Pérez. Different Platforms for Mobile Agents Development, is Java de Dominant Trend? *International Workshop on Intelligent Agents on the Internet and Web. World Congress on Expert Systems*. Mexico City. Méjico. Marzo, 1998.

[Perez99] Nelson Pérez Castillo. Sistema Integrado de Información Geográfica mediante Tecnologías de Objetos Distribuidos. Tesis Doctoral. Departamento de Informática. Universidad de Oviedo. Febrero, 1999.

[Raggett97] Dae Raggett, Arnaud Le Hors, Ian Jacobs. HTML 4.0 Specification. World

Wide Web Consortium. Diciembre, 1997.

[Robinson97] Steve Robinson, Alex Krasilshchikov. ActiveX Magic: An ActiveX Control adn DCOM Sample Using ATL. Microsoft MSDN. Mayo, 1997.

[Smith82] B. C. Smith. Reflection and Semantics in a Procedural Language. MIT-LCS-TR-272. Massachusetts Institute of Technology. 1982.

[Smith95] Randall B. Smith, David Ungar. Programming as an Experience: The Inspiration for Self. Sun Microsystems Laboratories. 1995.

[Steel60] T. B. Steel Jr. UNCOL: Universal Computer Oriented Language Revisited. Datamation. Enero-Febrero 1960.

[Steele90] Jr. Steele, G.L. Common Lisp: The Language. Segunda Edición. Digital Press, 1990.

[Stroustrup98] Bjarne Stroustrup. The C++ Programming Language. Third Edition. Addison-Wesley. October 1998.

[Sun95] The Java Virtual Machine Specification. Release 1.0 Beta Draft. Sun Microsystems Computer Corporation. Agosto 1995.

[Sun96] Java Beans™ 1.0 API Specification. JavaSoft. Sun Microsystems. Diciembre, 1996.

[Sun97] picoJava™ I Data Sheet. Java Processor Core. Sun Microsystems. Diciembre, 1997.

[Sun97b] Java™ Remote Method Invocation Specification (RMI). JavaSoft. Sun Microsystems. Febrero, 1997.

[Sun97c] Java™ Native Interface Specification. JavaSoft. Sun Microsystems. Mayo, 1997.

[Sun97d] Java Core Reflection. API and Specification. JavaSoft. Enero, 1997.

[Sun98] The Java HotSpot Viritual Machine Architecture. White Paper. Sun Microsystems. 1998.

[Sun98b] Enterprise Java Beans 1.0 Specification. Sun Microsystems. Marzo 1998.

[Sun98c] Enterprise JavaBeans™ to CORBA Mapping 1.0. Sun Microsystems. 1998.

[Sun98d] The HotSpot Virtual Machine Architecture. White Paper. Sun Microsystems. 1998.

[Swaine94] Swaine M. Programming Paradigms. Developing for Newton. *Dr. Dobb's Journal*, No. 212, 115-118. March 1994.

[Tajes96] Lourdes Tajes Martínez. Introducción de la Concurrencia en Sistemas Operativos Orientados a Objetos. *II Jornadas sobre Tecnologías Orientadas a Objetos.* Oviedo. 1996.

[Tajes97] Lourdes Tajes Martínez, Fernando Álvarez García, et al. Concurrencia en un Sistema Operativo Orientado a Objetos Basado en una Máquina Abstracta Reflectiva Orientada a Objeto. *VIII Jornadas de Paralelismo*. Cáceres, 1997.

[UML98] James Rumbaugh, Ivar Jacobson, and Grady Booch. The Unified Modeling Language Reference Manual. Addison Wesley. December 1998.

[Ungar87] Ungar, D. and Smith, R. B. SELF: The Power of Simplicity. *In OOPSLA'87*

*Conference Proceedings*. Published as SIGPLAN Notices, 22, 12, 227-241. 1987.

[Ungar91] David Ungar, Craig Chambers, Bay-wei Chang, and Urs Hölzle. Organizing Programs Without Classes. *Lisp and Symbolic Computation: An International Journal*, 4, 3. 1991.

[Venners98] Bill Venners. Inside the Java Virtual Machine. Java Masters. McGraw Hill. 1998.

[Wand88] Mitchell Wand, Daniel P. Friedman. The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower. Meta-Level Architectures and Reflection. P. Maes, D. Nardi Editors. North-Holland. 1988.

[Yellin96] Frank Yellin. The JIT Compiler API. Sun Microsystems. Octubre, 1996.

[Yokote93] Y. Yokote. Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach. *WorkShop on Reflection and Meta-level Architectures. OOPSLA' 93*. 1993.

# Índice