UNIVERSIDAD DE OVIEDO

Departamento de Informática

TESIS DOCTORAL

ADAPTACIÓN DINÁMICA DE PERSISTENCIA DE OBJETOS MEDIANTE REFLECTIVIDAD COMPUTACIONAL

Presentada por

Benjamín López Pérez

para obtención del título de Doctor por la Universidad de Oviedo

Dirigida por el

Profesor Doctor D. Francisco Ortín Soler

Oviedo, Marzo de 2006

RESUMEN

El principio de la separación de incumbencias o competencias se centra en la capacidad de modularizar aquellas partes diferentes de una aplicación relevantes a un concepto, objetivo, tarea o propósito específico. Una separación apropiada de los distintos aspectos de un sistema reduce la complejidad del software, mejora su comprensión y mantenibilidad, y facilita la reutilización de código.

Considerando la persistencia como una incumbencia típica en la mayoría de las aplicaciones, la separación de ésta del código principal del sistema permite el desarrollo de programas sin tener en cuenta sus requisitos persistentes, añadiendo y adaptando éstos en fases posteriores. Esta separación permite al desarrollador manejar la persistencia de los programas de forma independiente a su funcionalidad, así como desarrollar la lógica de negocio de la aplicación sin tener en cuenta sus requisitos persistentes. La persistencia sería, en dicho caso, una competencia ortogonal que podría ser reutilizada para cualquier aplicación, independientemente de sus requisitos funcionales.

Tras analizar las distintas alternativas existentes con el objetivo de conseguir una separación total del aspecto de persistencia, se puede comprobar que si bien unas ofrecen mayor transparencia que otras, ninguna permite desarrollar una aplicación cuyo código sea absolutamente independiente del aspecto relacionado con las características de persistencia. Del mismo modo, la mayoría de las alternativas existentes son dependientes de un lenguaje específico.

Adicionalmente a las carencias detectadas en la separación del aspecto de persistencia, las aproximaciones existentes no abordan su asignación, separación, ni adaptación en tiempo de ejecución. Una adaptación dinámica de los distintos parámetros relacionados con la persistencia de un sistema, así como su asignación y eliminación en tiempo de ejecución, son relevantes en sistemas adaptables y adaptativos a contextos surgidos en tiempo de programación.

La reflectividad computacional es una técnica que permite adaptar la estructura y comportamiento de un sistema en tiempo de ejecución. Sobre una plataforma reflectiva no restrictiva que ofrece reflectividad computacional en tiempo de ejecución, se ha desarrollado un sistema de persistencia en el que se demuestra cómo este mecanismo puede ser empleado para ofrecer una separación total de la incumbencia de persistencia de un modo independiente al lenguaje de programación seleccionado. Adicionalmente, la reflectividad computacional ofrece un elevado nivel

de adaptabilidad que permite cambiar dinámicamente las características persistentes de un programa en ejecución de un modo programático.

PALABRAS CLAVE

Persistencia, Reflectividad Computacional, Separación de Incumbencias, Desarrollo de Software Orientado a Aspectos, Persistencia Ortogonal, Orientación a Objetos.

ABSTRACT

The Separation of Concerns principle is based on the capability of modularizing those parts of an application that are exclusive of a specific concept, goal, task or purpose. The appropriate separation of different concerns in a computing system reduces software complexity, improving its legibility and maintainability, and it also facilitates code reuse.

Taking into account that persistence is a common concern in most of applications, the separation of this aspect from the program's main logic implies the development of software without taking into consideration their persistent requirements. Persistence features can be added and adapted in later stages. This separation of concerns offers the programmer the management of persistence issues regardless of programs functionality, making possible the development of the application logic without considering its persistence needs. Following this development principle, application's persistence would be an orthogonal concern capable of being reused in any application, regardless of its functional requirements.

After analyzing different approaches to achieve a complete separation of the persistence concern from application's logic, we have realized that, although there exist several degrees of transparency, none of them allow the development of applications which code is absolutely independent of the persistence aspect.

In addition to the inability of existing systems to separate the persistence concern form application's functionality, none of them tackle with dynamic assignment, adaptation and deletion of persistent issues. The dynamic adaptation of different parameters in a persistence system (as well as adding and deleting them at runtime) is a relevant matter to adaptable and adaptive programs that need to dynamically act in response to runtime emerging contexts.

Computational reflection is a technique that allows the adaptation of system's structure and behaviour at runtime. A persistence system has been developed on the basis of a non restrictive reflective platform in order to demonstrate that this mechanism can be used to provide a full separation of the persistence concern, in a programming language-independent way. Additionally, computational reflection offers a higher level of adaptability that allows the adaptation of persistent features dynamically, in a programmatic way.

KEYWORDS

Persistence, Computational Reflection, Separation of Concerns, Aspect Oriented Software Development, Orthogonal Persistence, Object Orientation.

TABLA DE CONTENIDOS

Capítulo 1
INTRODUCCIÓN

A lo largo de este capítulo se describen los principales objetivos buscados en el desarrollo de esta Tesis Doctoral, estableciendo un marco de requisitos generales a cumplir y posteriormente demostrada mediante la creación y evaluación de una serie de prototipos. Posteriormente se presenta la organización de la memoria, estructurada tanto en secciones como en capítulos.

1.1 Introducción

Actualmente la mayoría de las aplicaciones que manejan objetos persistentes son diseñadas con uso explícito de Sistemas Gestores de Bases de Datos (SGBD). Éstos pueden ser orientados a objetos o relacionales, empleando en el segundo caso algún mecanismo de traducción. Por tanto, el modo más común de desarrollar aplicaciones persistentes es entremezclando el código de la aplicación con sentencias OQL (*Object Query Language*) o SQL (*Structured Query Language*) de control de persistencia.

La no separación del código principal de la aplicación de las sentencias de gestión de la persistencia provoca una serie de inconvenientes:

1. Legibilidad y mantenibilidad. Puesto que el código de persistencia está enmarañado con el funcional, la comprensión, modificación y depuración de la lógica principal del programa se hacen más complejas.
2. Portabilidad. Existe una dependencia directa entre el mecanismo de persistencia y la implementación de la aplicación. Así, requerir cambios relativos a la persistencia provocarán cambios en la implementación del programa.
3. Reutilización de rutinas de persistencia. Es común encontrarse repetidamente con rutinas persistentes similares en las que lo único que varían es la estructura de los datos que manipulan. Este código debería estar factorizado y reutilizado independientemente de los objetos que manejen.
4. Adaptabilidad. El adaptar las características persistentes de una aplicación requiere modificar el código fuente y recompilar éste. No es posible

adaptar la configuración del aspecto de persistencia en tiempo de ejecución sin tener que modificar y recompilar la aplicación.

El principio de la separación de incumbencias o intereses (*separation of concerns*) surge para superar estos problemas comunes, existentes a lo largo ciclo de vida del software [Hürsch95]. La idea de este principio se basa en identificar y separar diferentes incumbencias de una aplicación, ortogonales entre sí. Siguiendo este principio, la persistencia de una aplicación debería poder añadirse por separado a una aplicación una vez ésta haya sido desarrollada. El código fuente de la misma no debería verse modificado.

Justificamos así la necesidad de estudiar las alternativas en la creación de un sistema de persistencia totalmente transparente al programador y reutilizable independientemente de sus requisitos funcionales. Éste debería ser, además, independiente del lenguaje de programación seleccionado. La adaptabilidad dinámica de los parámetros que constituyen el sistema de persistencia así como la actualización programática (adaptativa) de éstos, permitiría adecuar dichos parámetros a contextos surgidos en tiempo de ejecución, no previstos en tiempo de desarrollo.

A lo largo de esta memoria estudiaremos las distintas alternativas, y enunciaremos otras, para crear un sistema de persistencia transparente y adaptable dinámicamente, en el que se satisfagan todos los objetivos enunciados someramente en el párrafo anterior.

1.2 Objetivos

Enunciaremos los distintos objetivos generales propuestos para la creación del sistema de persistencia adaptable dinámicamente previamente mencionado, posponiendo hasta el siguiente capítulo la especificación formal del conjunto integral de requisitos impuestos a nuestro sistema.

1.2.1 Separación de la Incumbencia, Transparencia

El grado de transparencia alcanzado por el sistema de persistencia debe ser total. Esto significa que la forma de acceder a a los datos en las aplicaciones que hagan uso de un sistema de persistencia será la misma que si no se hiciese uso de él. El código fuente de las aplicaciones no necesitará ser modificado para poder utilizar los mecanismos de persistencia y esta regla deberá verificarse sin ninguna excepción.

De este modo, los desarrolladores de aplicaciones que usen el sistema de persistencia sí percibirán el aspecto de la persistencia como una incumbencia totalmente ortogonal, que podrá ser añadida o quitada de las aplicaciones dinámicamente sin tener que modificar el código fuente de éstas.

1.2.2 Independencia del Lenguaje de Programación y Plataforma

El sistema de persistencia no estará limitado a ninguna plataforma hardware o software concreta. Tampoco se limitará a funcionar con aplicaciones desarrolladas en un lenguaje de programación específico, ni con un conjunto restringido de lenguajes de programación. El sistema deberá proporcionar persistencia a múltiples aplicaciones escritas en un número arbitrario de diferentes lenguajes de programa-

ción. Se permitirá la interacción simultánea de dichas aplicaciones con el sistema de persistencia.

1.2.3 Adaptabilidad Dinámica

El sistema de persistencia permitirá su personalización a través de la configuración dinámica de los componentes de persistencia que lo conforman. Un usuario podrá, en tiempo de ejecución, configurar cualquier parámetro del sistema de persistencia.

El diseño del sistema de persistencia comprenderá múltiples parámetros de configuración, los cuales serán definidos en el Capítulo 2. El diseño del sistema de persistencia permitirá la modificación de estos parámetros de una manera sencilla para el usuario.

1.2.4 Adaptabilidad Programática

El sistema de persistencia será capaz de cambiar su comportamiento para adaptarse a los cambios que se produzcan en su entorno. A diferencia del objeto anterior, en este caso no interviene el usuario. Es el propio sistema (u otra aplicación) el que analiza el contexto en el que se está ejecutando y realiza acciones para adaptarse a él.

La adaptabilidad programática permite que el sistema de persistencia pueda abordar requisitos que no hayan sido tenidos en cuenta en tiempo de diseño. El sistema podrá autoconfigurarse a contextos dinámicos.

El objetivo principal de esta Tesis Doctoral es investigar y definir una tecnología para obtener un sistema de persistencia que cumpla los cuatro objetivos principales que acabamos de definir, así como los requisitos que se enumerarán en el siguiente capítulo. Además, se demostrará la validez de nuestra tesis con la imlementación de un prototipo del sistema de persistencia diseñado en este trabajo. Puesto que la adaptabilidad es uno de los objetivos principales del sistema, siendo ésta comúnmente contraria al rendimiento, dejaremos fuera de nuestros principales objetivos el buscar un sistema de computación más eficiente que los estudiados – pero sí más flexible. Diversas técnicas para obtener una mejora en el rendimiento de la tecnología diseñada podrán ser aplicadas un futuro; éstas se analizan en § 16.1.

1.3 Organización de la Memoria

A continuación mostramos la estructura de este documento, agrupando los capítulos en secciones con un contenido acorde.

1.3.1 Introducción y Requisitos del Sistema

En este capítulo narraremos la introducción, objetivos y organización de esta memoria. En el capítulo siguiente estableceremos el conjunto de requisitos impuestos a nuestro sistema, que serán utilizados principalmente para:

- Evaluar las aportaciones y carencias de los sistemas estudiados en la próxima sección.

- Fijar la arquitectura global de nuestro sistema.
- Evaluar los resultados del sistema propuesto, comparándolos con otros sistemas existentes estudiados.

1.3.2 Sistemas Existentes Estudiados

En esta sección se lleva a cabo un estudio del estado del arte de los sistemas similares al buscado. En el Capítulo 3 se realiza un amplio estudio de diversos sistemas de persistencia existentes, tanto en el ámbito empresarial como en el académico.

En el Capítulo 4 se detalla el análisis de cómo determinados sistemas ofrecen flexibilidad computacional sin utilizar técnicas de reflectividad. Uno de estos sistemas, la programación orientada a aspectos, será estudiado con mayor detalle en el Capítulo 5 debido a su importancia en la actualidad.

La técnica de la reflectividad (reflexión), sus conceptos principales y distintas clasificaciones, son introducidos en el Capítulo 6. Finalmente, la evaluación de múltiples sistemas reflectivos, sus aportaciones y limitaciones frente a los requisitos impuestos, son presentadas en el Capítulo 7.

1.3.3 Diseño del Sistema

En esta sección se introduce el sistema de persistencia propuesto basándose en los requisitos impuestos y los sistemas estudiados. La arquitectura global del sistema y su descomposición en capas es presentada en el Capítulo 8; la arquitectura de la primera capa, el sistema reflectivo no restrictivo, en el Capítulo 9; el sistema de persistencia transparente y adaptable dinámicamente es presentado en el Capítulo 10.

Detallamos los capítulos anteriores presentando el diseño del sistema reflectivo no restrictivo en el Capítulo 11. Sobre él, desarrollamos el sistema de persistencia cuyo diseño se presenta en Capítulo 12.

1.3.4 Aplicaciones, Evaluación, Conclusiones y Trabajo Futuro

Un conjunto de posibles aplicaciones prácticas de nuestro sistema se presenta en el
Capítulo 13, encontrándonos actualmente en fase de desarrollo de parte de ellas –haciendo uso de los prototipos existentes.

La evaluación del sistema presentado en esta Tesis, comparándolo con otros existentes, es llevada a cabo en el Capítulo 14 bajo los requisitos establecidos al comienzo de éste. En el Capítulo 15 se muestran las conclusiones globales de la investigación realizada y las principales aportaciones realizadas frente a los sistemas estudiados. Finalmente, en el Capítulo 16 se presentan las futuras líneas de investigación y el trabajo a realizar a partir de los resultados obtenidos.

1.3.5 Apéndices

Como apéndices de esta memoria se presentan:

- Manual de usuario del sistema reflectivo no restrictivo (apéndice A), cuyo diseño fue detallado en el Capítulo 11.
- Descripción del lenguaje Java implementado (apéndice B) en el sistema cuyo diseño se presenta en el Capítulo 12.
- El apéndice C constituye el conjunto de referencias bibliográficas utilizadas en este documento.

Capítulo 2
REQUISITOS DEL SISTEMA

En este capítulo especificaremos los requisitos del sistema buscado en esta Tesis Doctoral. Se identificarán los distintos requisitos y se describirán brevemente, englobando éstos dentro de distintas categorías funcionales. Los grupos de requisitos serán los propios referentes a transparencia del sistema de persistencia, adaptabilidad de éste en tiempo de ejecución y los concernientes a la portabilidad del sistema.

La especificación de los requisitos del sistema llevada a cabo en este capítulo, tiene por objetivo la validación del sistema diseñado así como el estudio de los distintos sistemas existentes similares al buscado. Los requisitos del sistema nos permiten reconocer los puntos positivos de sistemas reales –para su futura adopción o estudio– así como cuantificar sus carencias y justificar determinadas modificaciones y/o ampliaciones.

La especificación de los requisitos nos ayudará a establecer los objetivos buscados en el diseño de nuestro sistema, así como a validar la consecución de dichos objetivos una vez que éste haya sido desarrollado.

2.1 Requisitos de Transparencia

El sistema de persistencia creado deberá ofrecer un grado total de transparencia al programador. Cuando éste se encuentre desarrollando la aplicación, se centrará en los requisitos funcionales del sistema sin tener que entremezclar este tipo de código con llamadas a cualquier tipo de sistema de persistencia.

2.1.1 Separación de la Competencia de Persistencia

La competencia o incumbencia de la persistencia de cualquier aplicación será identificada, configurada y añadida como cualquier otro componente reutilizable. Cualquier cambio en los requisitos de persistencia de la aplicación, bien sean estáticos o dinámicos, no deberá desencadenar ninguna modificación de la implementación de la funcionalidad del mismo. Deberán tratarse como competencias ortogonales e independientes.

2.1.2 Reutilización Independiente de la Funcionalidad

El modo en el que se diseñe el sistema de persistencia no deberá tener dependencia alguna de la funcionalidad central de la aplicación. Sea cual fuere dicha funcionalidad, el sistema de persistencia deberá poseer la flexibilidad suficiente para adaptarse a éste, reutilizándose de un modo independiente a la funcionalidad de la aplicación.

2.1.3 Reutilización Independiente de la Estructura

Las estructuras de datos empleadas en las aplicaciones software conllevan cambios en el rendimiento de su actualización en un almacén. Adicionalmente, determinados sistemas emplean estructuras de datos predefinidas para llevar a cabo los procesos de actualización, eliminación y adición de elementos. El diseño del sistema no deberá imponer este tipo de requisitos y deberá ser independiente del las estructuras de datos empleadas en las aplicaciones.

2.1.4 Automatización Integral de la Persistencia

El sistema de persistencia deberá ser capaz de conocer, de un modo automático e implícito, el momento oportuno en el que se deban llevar a cabo las llamadas a las primitivas de transacción, inserción, actualización, consulta y eliminación de elementos en el almacén persistente. Esta automatización integral estará ligada a los requisitos persistentes del sistema que podrán variar, incluso dinámicamente, tal y como se especifica en el siguiente punto.

2.1.5 Ausencia de Impacto en los Procesos de Desarrollo

El sistema de persistencia no requerirá en tiempo de desarrollo ningún tipo de tratamiento de los datos que son persistentes. Existen diversos sistemas que requieren el uso de herramientas automáticas que tratan las estructuras de datos con objeto de prepararlas para su persistencia.

Estas herramientas actúan habitualmente sobre el código fuente o incluso sobre el código intermedio generado en tiempo de compilación. Aunque el impacto sobre el desarrollador es mucho menor que la modificación manual del código, merman la transparencia del sistema, puesto que el desarrollador debe modificar su entorno y proceso de desarrollo para poder utilizar el sistema de persistencia. Por ello, el diseño del sistema evitará la necesidad de utilización de este tipo de herramientas.

2.2 Requisitos de Adaptabilidad

Una de las facetas más importantes del sistema de persistencia es que posea capacidad de ser adaptado dinámicamente. A continuación describiremos una serie de requisitos relacionados con esta demanda, pero, en general, cualquier tipo de parámetro en el que el usuario pueda estar interesado a la hora de flexibilizar el sistema de persistencia debería ser factible su adaptación. Se busca utilizar una tecnología que ofrezca un mecanismo de adaptabilidad sin limitaciones a priori.

2.2.1 Adaptabilidad Dinámica

Los distintos parámetros del sistema de persistencia deberán ser configurables por un usuario administrador en tiempo de ejecución, sin necesidad de tener que especificar esto en el código fuente de la aplicación ni llevar a cabo algún tipo de diseño distinto al que se realizaría si la aplicación no fuese persistente en absoluto.

2.2.2 Adaptabilidad Programática

El sistema de persistencia deberá ser programático dinámicamente: él mismo (u otra aplicación) podrá adaptarse en tiempo de ejecución. En función de las estructuras de datos empleadas en una aplicación, determinados mecanismos de indexación de bases de datos se comportan mejor que otros. Un sistema adaptativo deberá ser capaz, por ejemplo, de detectar las estructuras utilizadas y seleccionar el mecanismo de indexación más adecuado para cada escenario.

2.2.3 Mecanismos de Indexación

Los índices son estructuras de almacenamiento físico empleados para acelerar la velocidad de acceso a los datos. Juegan un papel fundamental en los sistemas de persistencia, siendo más propicios unos que otros en función de determinadas características estructurales de la aplicación en ejecución. Por tanto, nuestro sistema de persistencia adaptable dinámicamente deberá ofrecer una adaptación implícita de éstos en función de dichos parámetros, cuando sean detectados en tiempo de ejecución.

2.2.4 Sistemas de Almacenamiento

A la hora de almacenar la información persistente de una aplicación informática puede ser necesaria la elección de distintos almacenes y formatos de persistencia en función de los requisitos estipulados. La utilización de ficheros, documentos XML, bases de datos relacionales y orientadas a objeto son ejemplos de diversos sistemas de almacenamiento necesarios en distintos escenarios.

El diseño que se lleve a cabo de nuestro sistema ha de permitir la variación del sistema de almacenamiento en tiempo de ejecución, así como la coexistencia de varios de ellos dinámicamente –tanto para distintos objetos como para el mismo conjunto de ellos, dando lugar a replicación de la información.

2.2.5 Políticas de Actualización

En la actualización de los objetos al sistema de persistencia existen diversas políticas y algoritmos de utilización utilizados. Cuando un objeto persistente es modificado en memoria, debería verse actualizado antes o después en el sistema de persistencia. El momento o las condiciones del volcado son definidos por una política de actualización. La adaptación, modificación o incluso adición de nuevas políticas de actualización deberán poder llevarse a cabo de un modo dinámico, pudiendo coexistir varias al mismo tiempo.

2.2.6 Selección de Objetos Persistentes

En diversos sistemas de persistencia desarrollados, la condición de persistencia de un objeto es definida por medio de un mecanismo de "alcance" o cierre transitivo. Existe un objeto raíz del sistema de persistencia que hace que todo objeto al que haga referencia, directa o indirectamente, lo hace persistente por definición. Ni esa implementación de ejemplo de selección de objetos persistentes, ni cualquier otra predefinida, será válida en nuestro sistema de persistencia. La selección de objetos persistentes se podrá definir y modificar de un modo totalmente abierto por el usuario, por el propio programa o por el administrador, en función de los requisitos persistentes de cada programa específico.

2.3 Requisitos de Portabilidad

El sistema de persistencia deberá ofrecer una garantía de no dependencia de elementos de una plataforma determinada, garantizando así su portabilidad y facilidad de utilización.

2.3.1 Independencia del Lenguaje

La programación de aplicaciones persistentes deberá poder realizarse mediante cualquier lenguaje de programación, permitiendo incluso la coexistencia de distintos lenguajes en un mismo almacén persistente. Su diseño no se debe enfocar a determinadas peculiaridades propias de un lenguaje de programación.

La elección del nivel de abstracción del modelo computacional es una tarea difícil: debe ser lo suficientemente genérica (bajo nivel de abstracción) para la mayoría de lenguajes, pero ofreciendo una semántica comprensible (mayor nivel de abstracción) para facilitar la interacción entre aplicaciones.

2.3.2 Independencia del Sistema Operativo

El sistema deberá poder implantarse en cualquier sistema operativo, y por lo tanto no deberá ser diseñado con características particulares de algún sistema concreto.

2.3.3 Independencia del Hardware

La interfaz de acceso a la plataforma no deberá ser dependiente del hardware en el que haya sido implantado. Todo el sistema de persistencia deberá poder instalarse sobre distintos sistemas hardware.

Capítulo 3
SISTEMAS DE PERSISTENCIA

En este capítulo se definen los conceptos de persistencia y otros relacionados que estarán presentes a lo largo de todo este trabajo. Analizaremos diferentes sistemas de persistencia comenzando por los sistemas más utilizados en la actualidad. El objetivo es obtener una panorámica general de las características positivas y negativas de los sistemas con mayor presencia en el mercado. A continuación se analizarán diversos grupos de sistemas que, por sus características y fundamentos, presentan interés de cara a los objetivos expuestos en § 1.2.

En cada caso estableceremos una descripción y estudio del sistema, para posteriormente analizar sus puntos positivos aportados y sus carencias, en función de los requisitos definidos en el Capítulo 2.

3.1 Definiciones

3.1.1 Persistencia

Se define persistencia de unos datos como el período de tiempo durante el cual los datos existen y son utilizables [Atkinson86]. Si bien el término persistencia se utiliza de diversas formas en el lenguaje corriente, la definición utilizada en este documento se refiere a mantener los valores de los datos durante todo su tiempo de vida, no importa cómo de breve o largo pueda ser éste.

En el ámbito del paradigma de la orientación a objetos, el concepto de persistencia se corresponde con uno de los tres principios denominados secundarios del modelo de objetos [Booch94]. En este contexto se entiende persistencia como la cualidad de un objeto de mantener su identidad y relaciones con otros objetos con independencia del sistema o proceso que lo creó [Martínez2001].

3.1.2 Persistencia Ortogonal

La persistencia ofrecida por un sistema es ortogonal si:

1. Todos los objetos pueden persistir, independientemente de su tipo

2. No hay diferencias para el programador a la hora de manejar datos persistentes y no persistentes
3. El mecanismo para identificar los objetos persistentes no está relacionado con el sistema de tipos, es decir, es ortogonal al universo del discurso del sistema.

Éste es un breve resumen de las conclusiones expuestas en [Atkinson95]. Los sistemas de persistencia ortogonal serán estudiados en § 3.4.

3.1.3 Base de Datos

Una base de datos puede definirse como una colección de registros almacenados de una manera sistemática en un ordenador, de tal modo que una aplicación informática pueda consultarla para utilizar los datos almacenados. Para optimizar la recuperación y ordenación, cada registro se organiza habitualmente como un conjunto de elementos de datos (o hechos). La aplicación informática utilizada para gestionar y consultar la base de datos se conoce como **sistema gestor de base de datos** (SGBD).

Muchos autores consideran que una colección de datos constituye una base de datos únicamente si presenta ciertas propiedades, por ejemplo, si se gestiona la integridad y calidad de los datos, si se permite el acceso a los mismos por varios usuarios, si presentan un esquema, o si se dispone de un lenguaje de consulta. Sin embargo, no existe un consenso a la hora de definir estas propiedades.

Existe normalmente una descripción de la estructura de los tipos de los datos almacenados en la base de datos: esta descripción se conoce como **esquema**. El esquema describe los objetos que se representan en la base de datos y las relaciones entre ellos. Existen diversas formas de organizar un esquema, o lo que es lo mismo, de modelar la estructura de la base de datos. Esto da lugar a la aparición de los **modelos de bases de datos** (o modelos de datos) entre los que se encuentran el jerárquico, el modelo en red, el relacional, el multidimensional o el orientado a objetos.

El modelo de datos que domina el mercado actualmente es el relacional. Este modelo fue introducido en un artículo académico por E. F. Codd en 1970 [Codd70] y su principal objetivo era conseguir que los sistemas de gestión de bases de datos fuesen independientes de las aplicaciones que los usasen. Para ello plantea abstraer la descripción de la estructura de la información separándola de la descripción de los mecanismos físicos de acceso. Para describir el modelo abstracto de la información propone un modelo matemático definido en términos de lógica de predicados y teoría de conjuntos

El modelo de **bases de datos orientadas a objeto** resulta de la aplicación del paradigma de la orientación a objetos a la tecnología de base de datos. Su motivación es acercar el mundo de las bases de datos a los lenguajes de programación, donde el paradigma de objetos domina actualmente en el mercado, permitiendo evitar la sobrecarga de convertir la información entre su representación en la base de datos como tablas y su representación en el lenguaje de programación como objetos, es decir, conseguir la desadaptación de impedancias [Maier89].

3.1.4 Sistema Gestor de Bases de Datos

Un sistema gestor de bases de datos (SGBD) es una aplicación informática (o un conjunto de ellas) diseñada para gestionar una base de datos y para ejecutar sobre ella las operaciones que soliciten diversos clientes.

Las funcionalidades básicas de cualquier sistema gestor de bases de datos incluyen:

- Un lenguaje de modelado para describir el esquema de cada base de datos gestionada con el SGBD (lenguaje de definición de datos).
- Un lenguaje que permita realizar consultas sobre la base de datos (lenguaje de consulta de datos)
- Un lenguaje que permita actualizar los datos contenidos en la base de datos (lenguaje de manipulación de datos).
- Un mecanismo de transacciones que idealmente garantice las propiedades ACID (atomicidad, consistencia, integridad y disponibilidad) de la información, con el objeto que asegurar la integridad de los datos independientemente de los accesos concurrentes de los usuarios (control de concurrencia) y de los fallos (tolerancia a fallos).

Los SGBD se clasifican según el modelo de datos que utilicen. El mercado está dominado actualmente por los SGBD relacionales, que utilizan como lenguaje de definición, consulta y manipulación de datos SQL [Melton93] o una variante propietaria del mismo. Son ejemplos de estos sistemas Ingres, Oracle, DB2 o SQL Server.

Los SGBD orientados a objetos, si bien han influido con sus ideas en los SGBD relacionales, han tenido una penetración muy baja en el mercado debido, entre otras razones, al rendimiento (notablemente inferior con respecto a los relacionales), a la ausencia de estándares en un primer momento y al fracaso en la adopción de la normalización propuesta después. Los SGBD orientados a objetos son estudiados con mayor detalle en § 3.2.7.

3.1.5 Sistemas de Indexación

Un índice, en terminología de base de datos, es una estructura de almacenamiento físico empleada para acelerar la velocidad de acceso a los datos. Los índices juegan un papel fundamental en las bases de datos, tanto relacionales como orientadas a objetos, siendo un soporte básico e indispensable para acelerar el procesamiento de las consultas [Bertino99].

En el ámbito de las consultas de un sistema de persistencia orientado a objetos, conviene resaltar tres características que vienen directamente impuestas por el modelo de objetos [Martínez2001]:

- **El mecanismo de la herencia (Jerarquías de Herencia)**. La herencia provoca que una instancia de una clase sea también una instancia de su superclase. Esto implica que el ámbito de acceso de una consulta sobre una clase en general incluye a todas sus subclases, a menos que se especifique lo contrario.

- **Predicados con atributos complejos anidados (Jerarquías de Agregación)**. Mientras que en el modelo relacional los valores de los atributos se restringen a tipos primitivos simples, en el modelo orientado a objetos el valor del atributo de un objeto puede ser un objeto o conjunto de objetos. Esto provoca que las condiciones de búsqueda en una consulta sobre una clase se puedan seguir expresando de la forma `<atributo operador valor>`, al igual que en el modelo relacional, pero con la diferencia básica de que el atributo puede ser un atributo anidado de la clase.

- **Predicados con invocación de métodos**. En el modelo de objetos, los métodos definen el comportamiento de los objetos y, al igual que en el predicado de una consulta puede aparecer un atributo, también podrá aparecer la invocación de un método.

Las características anteriores exigen técnicas de indexación que permitan un procesamiento eficiente de las consultas bajo estas condiciones [Martínez99]. Así, son muchas las técnicas de indexación en orientación a objetos que se han propuesto y que clásicamente se pueden clasificar en [Bertino95]:

- **Estructurales.** Se basan en los atributos de los objetos. Estas técnicas son muy importantes porque la mayoría de los lenguajes de consulta orientados a objetos permiten consultar mediante predicados basados en atributos de objetos. A su vez se pueden clasificar en:

  - Técnicas que proporcionan soporte para consultas basadas en la **Jerarquía de Herencia**. Ejemplos de los esquemas investigados en esta categoría son SC [Kim89], CH-Tree [Kim89], H-Tree [Chin92], Class Division [Ramaswamy95] y hcC-Tree [Sreenath94].
  - Técnicas que proporcionan soporte para predicados anidados, es decir, que soportan la **Jerarquía de Agregación**. Ejemplos de los índices de esta categoría son, entre otros, Nested, Path, Multiindex [Bertino89] y Path Dictionary Index [Lee98].
  - Técnicas que soportan tanto la **Jerarquía de Agregación** como la **Jerarquía de Herencia**. Nested Inherited e Inherited MultiIndex [Bertino95] son ejemplos de esta categoría.

- **De Comportamiento.** Proporcionan una ejecución eficiente para consultas que contienen invocación de métodos. La materialización de métodos (*method materialization* [Kemper94]) es una de dichas técnicas. En este campo no existe una proliferación de técnicas tan grande como en los anteriores.

La Tabla 1 recoge las consultas para las cuales son propicias las técnicas de indexación recogidas en la clasificación anterior. Estos resultados están extraídos del desarrollo realizado en [Martinez2001]:

Fundamento	Técnica	Consulta recomendada
Jerarquía de herencia	Single Class (SC)	Consultas sobre una única clase de la jerarquía.
	CH-Tree	Consultas que implican varias clases

Fundamento	Técnica		Consulta recomendada
			de la jerarquía.
	H-Tree		Consultas de recuperación que afectan a pocas clases de la jerarquía.
	hcC-Tree	CHP (Class Hierarchy Point)	Consultas puntuales sobre todas las clases de una jerarquía de clases que comienza con la clase consultada.
		SCP (Single Class Point)	Consultas puntuales sobre una clase simple.
		CHR (Class Hierarchy Range)	Consultas de rango sobre todas las clases de una jerarquía de clases que comienza con la clase consultada.
		Range)SCR (Single Class	Consultas de rango sobre una clase simple.
Jerarquía de agregación	Path (PX)		Consultas que impliquen la evaluación de predicados anidados sobre todas las clases del camino[1].
	Nested (NX)		Consultas de recuperación con jerarquías de agregación
	Multiindex (MX)		Consultas de actualización
	Path Dictionary Index (PDI)		Consultas que implican búsquedas asociativas y recorrido entre objetos
	Join Index Hierarchy (JIH)		Consultas con navegaciones frecuentes y actualizaciones poco frecuentes
Jerarquía de herencia y de agregación	Nested Inherited (NIX)		Consultas en las que los atributos en el camino sean simples, y el número de predicados anidados elevado
	Inherited MultiIndex		Consultas sobre la última clase del camino indexado
Invocación de métodos	Materialización de métodos (MM)		Consultas de recuperación que implican invocación de métodos

Tabla 1. Técnicas de indexación en orientación a objetos.

De los resultados recogidos en la Tabla 1 puede deducirse que unas técnicas proporcionan mejores rendimientos que otras para determinadas características del modelo de objetos. De hecho, es precisamente la riqueza del modelo de objetos la que complica las técnicas de indexación clásicas con el fin de proporcionar un rendimiento óptimo ante las diferentes posibilidades que ofrece dicho modelo [Martínez2001].

Además, determinadas técnicas de indexación tienen un rendimiento probado con determinados tipos de datos, pero no se puede asegurar su comportamiento para otros nuevos tipos de datos. Así por ejemplo, los árboles B+ se comportan

[1] Un camino es una rama en una jerarquía de agregación que comienza con una clase C y termina con un atributo anidado de C [Martínez2001].

bien con tipos de datos básicos (por ejemplo, enteros), pero no se puede asegurar su comportamiento para otros tipos de datos definidos por el usuario.

Finalmente, se puede desear indexar un atributo cuyo tipo ha sido definido por el usuario y que, por tanto, no fuese conocido por la técnica de indexación cuando ésta fue implementada.

Por todo esto, no se puede concluir por lo tanto que una técnica de indexación sea mejor que las demás en todas las circunstancias con los estudios realizados. Así, el mecanismo de indexación utilizado por un sistema de persistencia debería ser adaptable, permitiendo al usuario del sistema de persistencia cambiar los parámetros relativos al sistema de indexación y adaptar de este modo su comportamiento. En concreto, un mecanismo de indexación extensible debería permitir:

- Añadir al sistema nuevas técnicas de indexación [Stonebraker86]. Dos escenarios en los que sería deseable añadir nuevas técnicas de indexación serían la utilización de un nuevo tipo de dato que sugiera una técnica determinada y la utilización de una técnica específica para el tipo de consulta más frecuentemente realizada en el sistema. Existen diversos sistemas que implementan esta característica, como Starburst [Lindsay87], POSTGRES [Aoki91] u Oracle [Oracle99], y por algunos generadores de bases de datos como EXODUS [Carey86].

- Añadir nuevos tipos al sistema permitiendo utilizar sobre ellos las técnicas de indexación existentes.

En [Martínez2001] se propone un diseño orientado a objetos basado en patrones de diseño[2] [GOF94] para evitar el impacto sobre el sistema a la hora de implementar la parametrización descrita. En concreto, se propone utilizar el patrón *Strategy* para soportar nuevas técnicas de indexación, y el patrón *Command* para que cada técnica de indexación pueda funcionar con diferentes operadores según el tipo de dato que se indexe.

Por otra parte, existen circunstancias en las que el sistema está capacitado para seleccionar la técnica de indexación idónea en unas circunstancias determinadas, sin necesidad de actuación por parte del usuario. Por ejemplo, el sistema podría seleccionar una técnica de indexación concreta en función de las características de una consulta realizada y del modelo de objetos (ver Tabla 1). Otro ejemplo sería que el sistema detectase cuál es la técnica de indexación que mejor se comporta para un tipo de datos dado. Estos ejemplos sugieren un mecanismo de indexación adaptativo, es decir, capaz de adaptarse de una manera transparente a las necesidades del problema.

3.2 Desarrollo Habitual de Aplicaciones Persistentes

En este apartado se analizará una muestra representativa de los sistemas más utilizados en la actualidad a la hora de desarrollar aplicaciones persistentes.

El modelo de datos dominante en la actualidad es el modelo relacional, representado a nivel práctico por el lenguaje SQL [Melton93]. Tomando el lenguaje Java como ejemplo, el programador suele utilizar SQL, ya sea de un modo directo, utilizando JDBC [Fisher2003] o SQLJ [Clossman98]); o indirecto, empleando algún

[2] Se detectan los patrones de diseño aunque estos no se referencian expresamente.

sistema de traducción objeto-relacional como JDO [Sun2003] o Hibernate [Bauer2004]; así como un *framework* persistencia específico (los *frameworks* de persistencia se analizan en § 3.6).

El mecanismo más básico para dar persistencia a una aplicación es utilizar un sistema de almacenamiento basado en ficheros. Centrándonos en Java como ejemplo, esta plataforma incluye una tecnología de serialización de objetos [Sun97c]. Adicionalmente, XML [W3C98] se está empleando como formato popular de formato de ficheros. En la plataforma Java se ha introducido un sistema de serialización de objetos Java Beans con XML [Sun2003c]

Finalmente se analizará una alternativa menos extendida que las anteriores en la actualidad: la utilización de sistemas de gestión de bases de datos orientadas a objetos.

3.2.1 SQL

SQL [Melton93] (*Structured Query Language*) es el lenguaje más utilizado para crear, modificar y recuperar los datos de los sistemas de gestión de bases de datos relacionales [Melton93].

Sus orígenes se encuentran en la década de los 70, en el trabajo realizado en el centro de investigación San José de IBM, donde se buscaba desarrollar un sistema de bases de datos denominado "System R", basado en el modelo relacional de Codd [Codd70]. El lenguaje *Structured English Query Language* se diseñó con el objeto de manipular y recuperar los datos almacenados en System R. El acrónimo SEQUEL fue más tarde reducido a SQL por ser el primero una marca registrada. Aunque el trabajo de Codd [Codd70] tuvo mucha influencia sobre el diseño de SQL, el lenguaje fue diseñado en IBM por Donald D. Chamberlin y Raymond F. Boyce [Chamberlin74], publicándose en 1974 con el objeto de incrementar el interés en SQL.

SQL fue estandarizado por ANSI (*American Nacional Standards Institute*) en 1986 y por ISO (*International Organization for Standardization*) en 1987. En la Tabla 2 se muestran la fecha de publicación del estándar SQL y sus revisiones.

Año	Nombre	Alias	Comentarios
1986	SQL-86	SQL-87	Publicado originalmente por ANSI. Ratificado por la ISO en 1987.
1989	SQL-89		Revisión menor.
1992	SQL-92	SQL-2	Revisión fundamental. Se trata de la versión más ampliamente soportada por los sistemas de gestión de bases de datos actuales.
1999	SQL:1999	SQL-3	Se añaden expresiones regulares, consultas recursivas, *triggers*, tipos no escalares y algunas características orientadas a objetos (las dos últimas características presentan cierta controversia y aún no están ampliamente soportadas) [Melton99]
2003	SQL:2003		Introduce características relacionadas con XML, secuencias normalizadas, columnas con valores generados

Año	Nombre	Alias	Comentarios
			automáticamente.

Tabla 2. Estándares SQL y sus revisiones.

Aunque SQL está normalizado tanto por ANSI como por la ISO, existen numerosas extensiones y variaciones de los estándares. La mayor parte de éstas son de una naturaleza propietaria, como por ejemplo PL/SQL de Oracle o Sybase, SQL PL/SQL (*SQL Procedural Language*) de IBM y Transact-SQL de Microsoft. Tampoco es extraño que las implementaciones comerciales no soporten características básicas del estándar, como por ejemplo los tipos `DATE` o `TIME`, y utilicen en su lugar alguna variación propietaria. En consecuencia, el código SQL rara vez puede ser portado entre sistemas de bases de datos sin tener que realizar modificaciones sustanciales. Existen varias razones que justifican esta falta de portabilidad:

- La complejidad y tamaño del estándar hace que la mayor parte de las bases de datos no lo implementen en su totalidad.
- Los estándares no especifican el comportamiento de la base de datos en diversas áreas fundamentales, por ejemplo en los índices, dejando esta decisión en manos de las diferentes implementaciones.
- Si bien el estándar especifica con precisión la sintaxis que un sistema de bases de datos debe implementar para ser conforme con la norma, la especificación de la semántica de las construcciones del lenguaje está mucho menos definida, existiendo áreas de ambigüedad.
- Muchas empresas desarrolladoras de sistemas de gestión de bases de datos habían desarrollado sus sistemas antes de la aparición del estándar. Por ello, se han visto obligadas a llegar a un acuerdo entre romper la compatibilidad hacia atrás con sus sistemas o implementar correctamente el estándar.

SQL fue diseñado con un propósito específico: recuperar los datos contenidos en una base de datos relacional. Se trata de un lenguaje declarativo basado en conjuntos, al contrario que otros lenguajes imperativos como C++ [Stroustrup98] o Java [Gosling96] diseñados para poder resolver un conjunto mucho más amplio de problemas. Para integrar SQL con estos lenguajes de programación existen varias alternativas.

1. Por un lado existen extensiones propietarias al estándar, como PL/SQL, que convierten SQL en un lenguaje de programación completo.
2. Otra alternativa es incrustar el código SQL dentro del lenguaje de programación utilizado para desarrollar la aplicación, denominado en este caso "lenguaje anfitrión". Esta es la aproximación implementada, con alguna variación, por JDBC y SQLJ, que serán analizados en los apartados 3.2.2 y 3.2.3 respectivamente.
3. Finalmente, SQL puede ser utilizado indirectamente desde el lenguaje de programación, haciendo uso de un mecanismo de persistencia en el que se delegan los detalles de implementación de la misma. Dentro de los productos utilizados habitualmente en el desarrollo empresarial, esta es la opción más avanzada y donde más alternativas existen.

3.2.2 Java Database Connectivity (JDBC)

Sun Microsystems introdujo con la versión 1.1 de la plataforma Java la API *Java Database Connectivity* (JDBC) [Fisher2003] como la manera estándar de comunicarse con una base de datos relacional utilizando el lenguaje SQL. Actualmente, esta API es considerada la más exitosa dentro de la plataforma Java [Jordan2004].

Durante el diseño de la API JDBC, Sun consideró tres objetivos como fundamentales [Reese2000]:

- JDBC debería ser una API a nivel de SQL.
- JDBC debería tener en cuenta las características de las APIs de bases de datos existentes.
- JDBC debería ser simple.

Una API a nivel de SQL significa que JDBC permite al desarrollador construir sentencias SQL e incrustarlas dentro de llamadas Java a la API. Es decir, se utiliza básicamente SQL pero JDBC suaviza la transición entre el mundo de las bases de datos y de las aplicaciones Java. Por ejemplo, los resultados de la base de datos son devueltos como objetos Java y los problemas de acceso que sucedan son notificados mediante excepciones.

Por otra parte, la idea de Sun de proporcionar una API de acceso a bases de datos universal no es nueva, puesto que mucho antes de su aparición ya existía una gran confusión provocada por la proliferación de APIs de acceso a bases de datos propietarias. De hecho, Sun recogió a la hora de diseñar JDBC las mejores características de otra API de este tipo: Open DataBase Connectivity (ODBC) [Sanders98]. ODBC fue desarrollada para crear un único estándar para el acceso a bases de datos en entornos Windows. Aunque la industria aceptó ODBC como el medio principal de acceso a bases de datos en Windows, este estándar no encaja bien en el mundo de Java principalmente por tratarse de una API C que requiere APIs intermedias para otros lenguajes, además de ser una API con una notable complejidad y tamaño.

Además de ODBC, JDBC está enormemente influido por otra API existente, la X/OPEN SQL Call Level Interface (CLI). Sun quiso reutilizar las principales abstracciones de estas APIs con el fin de facilitar la aceptación de JDBC por parte de los vendedores de bases de datos, además de facilitar la migración de los desarrolladores de ODBC y SQL CLI.

Sun se dio cuenta además de que derivar una API de las APIs existentes podía favorecer un desarrollo rápido de las soluciones que utilizasen motores de bases de datos que soportasen los antiguos protocolos. En concreto, se desarrolló un puente ODBC que asocia llamadas JDBC a llamadas ODBC, dando de este modo a las aplicaciones Java acceso a cualquier sistema gestor de bases de datos que soportase ODBC.

Finalmente JDBC fue diseñado con el fin de que fuese tan sencilla como fuera posible ofreciendo a los desarrolladores la máxima flexibilidad. Un criterio clave [Reese2000] utilizado fue el de ofrecer un interfaz sencillo para las tareas más comunes, mientras que las tareas menos habituales se ofrecían mediante interfaces especializados. Por ejemplo, tres interfaces manejan la gran mayoría de accesos a una base de datos, mientras que para el resto de tareas, menos comunes y más complejas, JDBC ofrece diversos interfaces.

Con respecto a la arquitectura de la API, JDBC se estructura en torno a un conjunto de interfaces que los vendedores de sistemas de bases de datos deben implementar (ver Figura 3.1). El conjunto de clases que implementan los interfaces JDBC para un SGBD concreto se denomina *driver* JDBC.

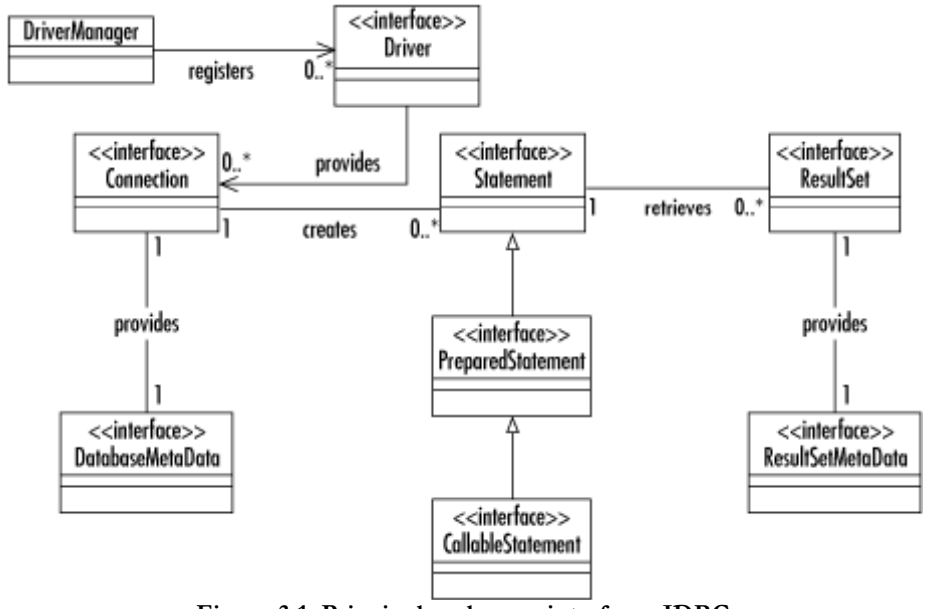


Figura 3.1. Principales clases e interfaces JDBC.

Sun clasifica los *drivers* JDBC en base a los siguientes tipos [Fisher2003]:

- **Tipo 1**. Estos *drivers* implementan un puente para acceder a la base de datos. Un ejemplo de este tipo de *driver* es el puente JDBC-ODBC que viene con la JDK 1.2. Proporciona una puerta de enlace a la API ODBC. Serán las implementaciones de esta API las que realmente realicen el acceso a la base de datos. Las soluciones basadas en puentes normalmente requieren software instalado en los clientes, lo que implica que no son adecuadas para aplicaciones que no permitan instalar software en dichas máquinas.

- **Tipo 2**. Son *drivers* nativos. Esto significa que el *driver* contiene código Java que hace llamadas nativas a C o métodos C++ proporcionados por los vendedores de bases da datos, que son los que realizan realmente el acceso a la base de datos. Esta solución también requiere software instalado en el cliente.

- **Tipo 3**. Estos *drivers* proporcionan al cliente una API de red genérica que será utilizada en los accesos concretos a la base de datos a nivel del servidor. Es decir, el *driver* JDBC en el cliente utiliza *sockets* para llamar a una aplicación middleware en el servidor que traduce las peticiones del cliente en una API específica al *driver* deseado. Este tipo de *driver* es extremadamente flexible debido a que no requiere código instalado en el cliente y un simple *driver* puede proporcionar acceso a múltiples bases de datos.

- **Tipo 4**. Estos *drivers* se comunican directamente con la base de datos utilizando *sockets* y protocolos de red propietarios del SGBD. Se trata de la solución completamente Java más directa. Debido a que raramente la documentación de esos protocolos de red está disponible, este tipo de *driver* casi siempre es proporcionado por el fabricante del SGBD.

La principal ventaja de la arquitectura de JDBC es que el desarrollador no tiene que preocuparse de la implementación de las clases que subyacen a la API más allá de las sentencias relativas a la utilización del *driver* concreto. En el mismo sentido, una aplicación puede cambiar el SGBD utilizado modificando únicamente el código de utilización del *driver*.

JDBC hace uso de diversas características de la plataforma Java [Kramer96], como la gestión automática de memoria, el manejo adecuado de cadenas de caracteres y la API de contenedores para simplificar el uso programático de SQL.

JDBC proporciona además un mapeo transparente de los tipos básicos de datos tales como `String` o `int` con los tipos SQL correspondientes. En concreto, JDBC define un conjunto de "tipos JDBC" [Fisher2003] que actúan como posibles intermediarios entre los tipos Java y los tipos SQL, y que son mapeados de una manera transparente al tipo adecuado de la base de datos subyacente por los *drivers* JDBC.

JDBC gestiona la complejidad del mantenimiento de las conexiones a la base de datos y también proporciona un abanico de mecanismos para mejorar el rendimiento y la escalabilidad. Por ejemplo, las "sentencias preparadas" (*prepared statements*) son habitualmente compiladas y cacheadas por el SGBD y pueden reutilizarse muchas veces con diferentes argumentos. También pueden agruparse numerosas sentencias por lotes con el fin de reducir el número de accesos al servidor.

Para permitir interactuar con las variaciones de los diferentes sistemas de bases de datos SQL, JDBC ofrece un rico interfaz de gestión de metadatos que permite a las aplicaciones descubrir en tiempo de ejecución qué características específicas soporta un tipo de base de datos concreta, así como el acceso al esquema de la base de datos.

La API JDBC continúa evolucionando. Las mejoras de la actual versión 3.0 incluyen *pooling* de sentencias, que permite almacenar en caché sentencias preparadas para su reutilización por parte de múltiples conexiones lógicas, y también un mejor soporte para características avanzadas del estándar SQL 99 [Melton99].

3.2.3 SQLJ

SQLJ [Clossman98] es una tecnología que permite a un programa Java acceder a una base de datos utilizando sentencias SQL incrustadas en el código de la aplicación. SQLJ fue desarrollado por *The SQLJ Group*, un consorcio compuesto por Sun Microsystems y los principales fabricantes de bases de datos, Oracle, IBM, Compac, Informix y Sybase. Una vez desarrollada la especificación del estándar, ésta fue remitida al Comité técnico H2 sobre Bases de Datos del INCITS *(International Committee for Information Technology Standard*). La especificación SQLJ comprende tres partes:

- La parte 0 especifica cómo incrustar sentencias SQL en código Java. Únicamente permite incrustar sentencias SQL estáticas, debiendo ser las sentencias SQL dinámicas gestionadas con sentencias JDBC. Soporta mapeo de tipos entre Java y SQL definido por JDBC.
- La parte 1 abarca el uso de Java desde rutinas SQL. Permite invocar métodos escritos en Java desde código SQL. Los métodos Java proporcionan la implementación de los procedimientos SQL. Para asociar los pro-

cedimientos SQL y los métodos Java, que deberán ser estáticos, cada parámetro SQL y su equivalente Java deben ser compatibles, es decir, mapeables, así como los retornos de las funciones.

- La parte 2 define extensiones SQL para utilizar clases Java como tipos de datos de SQL. Permite el mapeo de tipos definidos por el usuario según la especificación SQL 99 [Melton99] a clases Java. También permite importar un paquete Java dentro de la base de datos SQL mediante la definición de tablas conteniendo columnas cuyos tipos de datos se especifican como clases Java. Los tipos estructurados son asociados con clases, los campos con atributos, y los inicializadores (*initializers*) con constructores.

SQLJ consta de dos componentes: el traductor (*translator*) y las librerías de tiempo de ejecución (*runtime libraries*). El traductor es un preprocesador que lee los archivos de código fuente Java que contienen las sentencias SQL incrustadas y las traduce en llamadas a las librerías de tiempo de ejecución. El traductor se encarga también de compilar el código fuente Java resultante. Las llamadas a las librerías de tiempo de ejecución en el código compilado son las que realizan realmente las operaciones con la base de datos.

En un programa SQLJ una sentencia SQLJ puede aparecer en cualquier lugar donde sería válida una sentencia Java normal. Todas las sentencias comienzan con el símbolo reservado `#sql` con el fin de diferenciarlas de las sentencias Java normales.

```
int id = 2;
String first_name = null;
String last_name = null;
java.sql.Date dob = null;
String phone = null;

#sql {
SELECT
first_name, last_name, dob, phone
INTO
      :first_name, :last_name, :dob, :phone
FROM
      customers
WHERE
      id = :id
};
```

Figura 3.2. Ejemplo de uso de SQLJ.

En la Figura 3.2 se muestra un ejemplo de código SQL incrustado en una aplicación SQLJ. Nótese como las referencias a los objetos temporales utilizados en la aplicación pueden utilizarse desde el código SQL incrustado anteponiendo "`:`" al identificador correspondiente.

SQLJ fue concebido como una alternativa a JDBC, si bien SQLJ utiliza JDBC en las rutinas generadas por el preprocesador. A pesar del soporte recibido por grandes empresas como Oracle su impacto en el mercado ha sido notablemente reducido. En [Reese2002] se apunta cómo la principal razón es que se trata de una alternativa basada en un modelo de acceso a bases de datos desfasado: el paradigma utilizado por SQLJ resulta familiar para programadores de C o COBOL pero va en contra de la naturaleza orientada a objetos de Java.

3.2.4 Mecanismo de Serialización de Objetos Java

Se trata del mecanismo estándar de serialización de objetos introducido en la plataforma Java desde su versión 1.1 [Sun97c].

Una clase se declara serializable implementando el interfaz `java.io.Serializable`. Este interfaz no define ningún método, únicamente identifica los objetos que pueden ser serializables con el objeto de prevenir la serialización de datos sensibles [Jordan2004].

Presenta un mecanismo de persistencia por alcance [Atkinson95]. Empezando por un objeto raíz, la serialización incluye todos los objetos alcanzables desde éste, transitivamente, desde la raíz, siguiendo aquellos campos del objeto que referencian a otros objetos. El mecanismo de serialización se basa en el sistema de flujos de la plataforma Java (*streams*). Para serializar, se escribe en el flujo una representación de cada objeto, incluyendo su clase y sus campos. La deserialización consiste en leer un flujo previamente escrito. La información de la clase se limita al nombre y a la codificación de la signatura de los métodos. En concreto, el código de los métodos no se serializa.

El sistema presenta un mecanismo general para que el desarrollador intervenga en el proceso de serialización, lo que puede ser utilizado, entre otras cosas, para transformar u omitir ciertos campos del objeto. Para ello, deben implementarse los siguientes métodos:

```
private void writeObject(java.io.ObjectOutputStream out)
     throws IOException
 private void readObject(java.io.ObjectInputStream in)
     throws IOException, ClassNotFoundException;
```

Figura 3.3. Métodos a implementar para intervenir en el proceso de serialización.

En la práctica, este mecanismo debe ser utilizado mucho más a menudo de lo que puede parecer en un principio, con el fin de incrementar la eficiencia del mecanismo de serialización. El mecanismo de serialización y deserialización de objetos se realiza en una misma operación, no importa cuantos objetos comprenda el árbol que se almacena. Esto causa graves problemas de rendimiento cuando el número de objetos es numeroso [Jordan2004].

3.2.5 Serialización de Java Beans con XML

Este mecanismo de serialización fue inicialmente desarrollado para serializar objetos de la API gráfica de Java Swing [Drye99]. El objetivo era poder intercambiar modelos de interfaces gráficos entre los diferentes entornos de desarrollo que soportaban su diseño visual. Finalmente, el mecanismo fue introducido como un mecanismo de serialización de Java Beans [Sun96] en la versión 1.4.1 de la plataforma Java.

El mecanismo, en su funcionamiento por defecto, es capaz de serializar y deserializar cualquier objeto que:

- Utilice los patrones de nomenclatura de los Java Beans [Sun96] para definir sus propiedades.
- Todo su estado venga definido en base a dichas propiedades.

El lenguaje XML describe los objetos serializados –lo que realmente describe es el conjunto de invocaciones que deben realizarse para reconstruir el estado de dicho objeto. Por ejemplo, en la Figura 3.4 se muestra un botón Swing. Para reconstruirlo únicamente hará falta llamar a su constructor por defecto e invocar al método `setText()` con el parámetro "`Hello, world`".

```
<object class="javax.swing.JButton">
  <void method="setText">
    <string>Hello, world</string>
  </void>
</object>
```

Figura 3.4. Ejemplo de un botón Swing serializado con XML.

El sistema utiliza un algoritmo para minimizar el tamaño del fichero XML generado: evita guardar propiedades que tengan valores por defecto. Este mecanismo implica pagar el precio de clonar la estructura inicial de datos, lo que puede resultar un problema de escalabilidad cuando se tienen grandes volúmenes de datos.

Si un objeto no expone todo su estado a través de propiedades Java Beans el sistema por defecto no funciona. En este caso, se permite la personalización de cómo se construye el fichero XML. Para ello, se ofrece una API orientada a objetos con la que construir sentencias Java (invocaciones a métodos, a constructores, establecimiento de propiedades). Utilizando esta API, pueden construirse objetos "delegados" que se asociarán con la clase del objeto que serializan, y que serán los encargados de realizar la generación de sentencias Java o realizar su interpretación. Estos objetos serán invocados durante el proceso de serialización/deserialización de cada objeto de la clase a la que están asociados.

3.2.6 Herramientas de Mapeo Objeto/Relacional

Las herramientas de mapeo objeto/relacional (*Object/Relational Mapping*, ORM), también denominadas mediadores objeto-relacionales, envoltorios o *wrappers* [Devis97], son productos que ofrecen capas de software orientado a objetos que permiten trabajar sobre una base de datos relacional. Estos sistemas ofrecen al desarrollador la sensación de estar trabajando con un SGBDOO (Sistema de Gestión de Bases de Datos Orientadas a Objetos), ocultando los mecanismos necesarios para traducir las clases en tablas relacionales y los objetos en tuplas. Estas herramientas generalmente emplean SQL para interactuar con la base de datos relacional.

Una solución ORM comprende 3 componentes [Bauer2004] básicos:

- Una API para ejecutar las operaciones CRUD (*Create, Retrieve, Update, Delete*) básicas sobre los objetos de las clases persistentes.
- Una API o un lenguaje para especificar consultas que hagan referencia a las clases o a las propiedades de las clases.
- Un mecanismo para especificar los metadatos de mapeo. Las herramientas ORM necesitan un formato de metadatos para que la aplicación especifique cómo se realizará la traducción entre clases y tablas, propiedades y columnas, asociaciones y claves ajenas, tipos Java y tipos SQL.

Las herramientas ORM pueden ser implementadas de diversas maneras. Mark Fussel [Fussel97] define los siguientes niveles de calidad para las herramientas de mapeo objeto/relacional.

- **Relacional pura (***Pure relational***)**. Toda la aplicación, incluido el interfaz de usuario, se diseña siguiendo el modelo relacional y las operaciones relacionales basadas en SQL. Como principal ventaja destaca que la manipulación directa de código SQL permite muchas optimizaciones. Sin embargo, inconvenientes tales como la dificultad de mantenimiento y la falta de portabilidad son muy significativos. Las aplicaciones en esta categoría utilizan intensivamente procedimientos almacenados (*stored procedures*), desplazando parte de la carga de trabajo desde la capa de negocio hasta la base de datos.

- **Mapeo de objetos ligero (***Light object mapping***)**. Las entidades se representan como clases que se mapean de forma manual sobre tablas relacionales. La codificación manual de SQL puede quedar separada de la lógica de negocio si se utilizan patrones de diseño como *Data Access Object* (DAO) [Alur2001]. Esta alternativa está ampliamente extendida para aplicaciones con un número pequeño de entidades, o aplicaciones con un modelo de datos genéricos dirigido por metadatos.

- **Mapeo de objetos medio (***Medium object mapping***)**. La aplicación se diseña en torno a un modelo de objetos. SQL se genera en tiempo de compilación utilizando una herramienta de generación de código o en tiempo de ejecución utilizando el código de un *framework*. El sistema de persistencia soporta las relaciones entre objetos y las consultas pueden especificarse utilizando un lenguaje de expresiones orientadas a objetos. Una gran parte de las soluciones ORM soportan este nivel de funcionalidad. Son adecuadas para aplicaciones de un tamaño medio, con algunas transacciones complejas, en particular cuando la portabilidad entre diferentes bases de datos es un factor importante.

- **Mapeo de objetos total (***Full object mapping***)**. Estas herramientas soportan características avanzadas del modelado de objetos: composición, herencia, polimorfismo y persistencia por alcance [Atkinson95]. La capa de persistencia implementa persistencia transparente y las clases persistentes no necesitan heredar ninguna clase base especial ni implementar un determinado interfaz. Además se incluyen diversas optimizaciones que son transparentes para la aplicación, tales como varias estrategias para cargar los objetos de forma transparente desde el almacén utilizado cuando se navega por un árbol de objetos, o la utilización de cachés de objetos.

Las herramientas de mapeo objeto relacional presentan diversas ventajas que justifican su gran aceptación como sistema de persistencia en plataformas orientadas a objetos.

- Incrementan la productividad al eliminar evitar que sea el propio programador el que tenga que hacer la conversión de clases y objetos a tablas. El código necesario para esta traducción puede suponer entre un 25% y un 40% del total [Excelon97].

- Favorecen la mantenibilidad de la aplicación, al quedar separado el código que interactúa con la base de datos del código correspondiente a la lógica de negocio.
- Eliminan los errores derivados de la traducción objeto/relacional programática por parte del desarrollador.
- Permiten aprovechar las ventajas derivadas del dominio absoluto que tienen los SGBD relacionales en el mercado actual, en especial, del mayor rendimiento de los motores relacionales sobre los que implementan otros paradigmas. Además, favorecen la no dependencia de un fabricante concreto, puesto que el estándar SQL [Melton93] es ampliamente soportado por los diferentes SGBD.

Por otra parte, la traducción automática entre ambos paradigmas también supone un coste debido a la desadaptación de impedancias [Maier89]: un coste computacional, puesto que se añade una nueva capa software adicional; y un coste de desarrollo, puesto que el desarrollador habitualmente deberá especificar mediante algún mecanismo los metadatos que configuran cómo se realizará la traducción.

3.2.6.1 Hibernate

Hibernate, también conocida como H8, es una herramienta de mapeo objeto/relacional para la plataforma Java. Se trata de un proyecto de código abierto distribuido bajo licencia LGPL que fue desarrollado por un grupo de programadores Java de diversos países liderados por Gavin King.

Hibernate proporciona un *framework* sencillo de utilizar para mapear un modelo dominio orientado a objetos a una base de datos relacional. Ofrece un mecanismo de persistencia cuya transparencia se refleja en su soporte para objetos POJO. El término POJO (*Plain Old Java Object* o también *Plain Ordinary Java Object*) fue acuñado en 2002 por Martin Fowler, Rebecca Parsons y Josh Mackenzie. Se tratan de objetos que son esencialmente JavaBeans [Sun96] utilizados en la capa de negocio (muchos desarrolladores utilizan el término POJO y JavaBeans independientemente). Un objeto POJO declara métodos de negocio que definen comportamiento y propiedades que representan estado. Lo que busca el término es diferenciar este tipo de objetos de los Entity Beans [Roman2005], cuya gestión es mucho más tediosa y son menos naturales desde el punto de vista del desarrollador.

Hibernate está pensado para trabajar con un modelo de dominio implementado mediante POJOs. A este modelo se le imponen una serie de requerimientos que intentan coincidir con las mejores prácticas de desarrollo con POJOs, de tal manera que se asegure la compatibilidad de la mayor parte de objetos POJO sin necesidad de realizar modificaciones. El modelo, por lo tanto, se trata de una mezcla de la especificación de JavaBeans, las mejores prácticas POJO y requerimientos específicos de Hibernate. Estos requerimientos son:

- Un constructor sin argumentos para toda clase persistente.
- Se recomiendan propiedades accesibles mediante métodos de acceso `get()` y `set()`, según la convención de nombres definida para los JavaBeans [Sun96] aunque este requisito no es obligatorio.
- Cuando se modelen asociaciones, deben utilizarse las *Collections* de Java para el atributo que modele la relación y además deben utilizarse para la

referencia del correspondiente atributo interfaces, en lugar de implementaciones concretas.

Se va a ilustrar el funcionamiento de Hibernate con un sencillo ejemplo que hará persistir un objeto de dominio representado por la clase Persona, cuyo código se muestra en la Figura 3.5.

```
package example.hibernate;

public class Person{
	protected String name;

	public void setName(String name){
		this.name = name;
	}

	public String getName(){
		return name;
	}
}
```

Figura 3.5. Código de la clase persona.

Para especificar los metadatos que configuran cómo se realiza el mapeo entre el modelo de objeto y el relacional Hibernate permite utilizar dos aproximaciones distintas.

En primer lugar puede utilizarse un documento XML que especifica cómo se realiza la traducción. En la Figura 3.6 se muestra un fichero XML que especifica cómo realizar el mapeo para la clase Persona. En este fichero se especifica que la clase `Person` se almacenará en la tabla `PERSON`, que el identificador de cada objeto persona se almacenará en la columna `PERSON_ID` como clave primaria y será gestionado internamente por Hibernate (posteriormente se explicarán los diferentes mecanismos de gestión identidad en hibernate), y finalmente que la propiedad `name` de cada objeto persona se almacenará en la columna `NAME` de la tabla `PERSON`. Debe señalarse que, si bien muchos desarrolladores optan por escribir este fichero XML a mano, existen diversas herramientas que pueden generar este documento automáticamente, como por ejemplo Xdoclet [Walls2003].

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
	<class
		name="example.hibernate.Person"
		table="PERSON">
		<id column="PERSON_ID">
			<generator class="native"/>
		</id>

		<property
			name="name"
			column="NAME"
			type="string"/>
	</class>
</hibernate-mapping>
```

Figura 3.6. Documento XML con los metadatos de mapeo para la clase Persona

La segunda aproximación para especificar los metadatos de mapeo que soporta Hibernate es utilizar la denominada programación orientada a atributos, que

fue introducida dentro de la plataforma Java de la mano de XDoclet [Walls2003]. Este sistema permite utilizar el formato de las etiquetas Javadoc (`@attribute`) para especificar atributos de metadatos a nivel de clases, atributos o métodos. En la Figura 3.7 se muestra como puede anotarse el código fuente de la clase persona utilizando XDoclet, con el objeto de incluir la misma información que en el fichero recogido en la Figura 3.6.

```
package example.hibernate;

/**
* @hibernate.class
* table="PERSON"
*/

public class Person{
       protected String name;

       public void setName(String name){
              this.name = name;
       }

       /**
       * @hibernate.property
       */
       public String getName(){
              return name;
       }
}
```

Figura 3.7. Código de la clase persona anotado con XDoclet

Con el lanzamiento de la versión 1.5 estándar de la plataforma Java se introdujo un soporte para las introducir metadatos en el código fuente en forma de anotaciones. Se trata de un mecanismo mucho más sofisticado que el ofrecido por XDoclet. En el momento de escribir este documento el soporte por parte de Hibernate del sistema de anotaciones de la JDK 1.5 está siendo desarrollado.

Para almacenar un objeto en la base de datos, Hibernate utiliza un gestor de persistencia que viene definido por el interfaz `Session`. El código necesario para hacer persistir un objeto persona se muestra en Figura 3.8.

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
Person person = new Person();
session.save(person);
tx.commit();
session.close();
```

Figura 3.8. Ejemplo de persistencia de un objeto con Hibernate.

El ciclo de vida de los objetos se modela mediante estados. Hibernate define únicamente tres estados, escondiendo la complejidad de su implementación interna al código del cliente. Estos estados son: *transient*, *persistent* y *detached* [Hibernate2005] y el correspondiente diagrama de estados se ilustra con la Figura 3.9.

- ***Transient*** **(temporal)**. Un objeto es temporal si acaba de ser instanciado usando el operador `new` pero aún no ha sido asociado con una `Session` Hibernate. Este tipo de objetos no tiene una representación persistente ni tampoco un identificador asociado.

- ***Persistent*** **(persistente)**. Una instancia persistente tiene una representación en la base de datos y un identificador asociado. Está por definición dentro del ámbito de un objeto `Session`, bien sea porque haya sido guardado o recuperado de la base de datos. Hibernate detectará cualquier cambio que se realice sobre el estado del objeto y lo sincronizará con la base de datos cuando finalice la unidad de trabajo en curso.

- ***Detached*** **(separado)**. Una instancia "separada" es un objeto que ha sido persistente, pero cuya sesión asociada ha sido cerrada. La referencia al objeto todavía es válida y su estado podría ser modificado. Una instancia separada puede volver a ser unida con una nueva `Sessión` en un momento posterior, haciéndola a ella y a todas las modificaciones realizadas persistentes de nuevo.

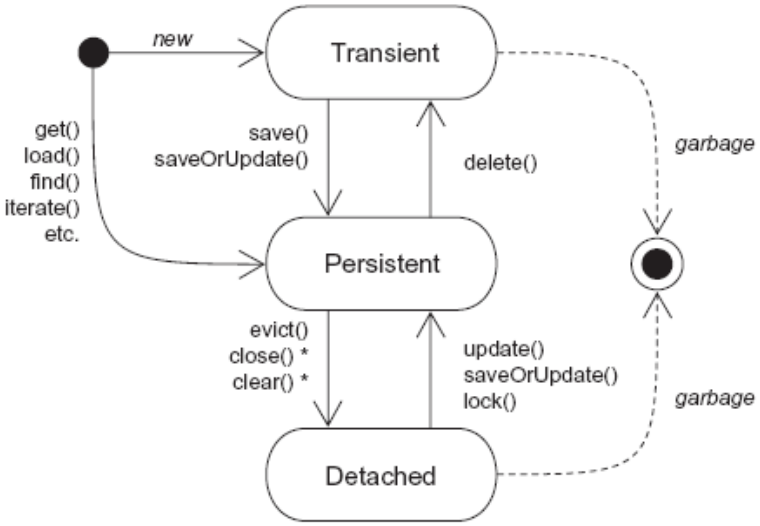


Figura 3.9. Ciclo de vida de un objeto persistente.

Para recuperar objetos de la base de datos Hibernate proporciona los siguientes mecanismos [Bauer2004]:

- Navegar por todo el grafo de objetos, una vez que un objeto ya ha sido cargado. El proceso de carga mientras se navega por el grafo es realizado por Hibernate de una manera transparente al usuario. Se ofrece además la posibilidad de configurar cómo se traen los objetos desde la base de datos a memoria permitiendo la configuración de distintas estrategias. Un ejemplo es la estrategia *Lazy fetching* mediante la cual sólo se consulta la base de datos para traer los objetos o colecciones de objetos a memoria la primera vez que son accedidos (a no ser que el objeto asociado haya sido cacheado).

- Recuperar los objetos a partir de su identificador, que es el método más eficiente cuando se conoce el identificador de un objeto.

- Utilizar el lenguaje HQL (*Hibernate Query Language*), el cual es un dialecto orientado a objetos de la familia de lenguajes de consulta relacionales SQL, si bien a diferencia de SQL no incluye características de definición ni manipulación de datos.

- Utilizar la API Hibernate *Criteria*, la cual proporciona a través del interfaz `Criteria` un modo seguro y orientado a objetos de ejecutar consultas sin necesidad de manipular cadenas de caracteres.

- Utilizar sentencias nativas SQL. En este caso Hibernate se encarga de mapear los resultados de las consultas JDBC (*result sets*) a grafos de objetos persistentes.

Con respecto a la actualización de los objetos persistentes, ésta se realiza de forma transparente al usuario, que únicamente tiene que codificar explícitamente la finalización de las unidades de trabajo abiertas. En el caso más sencillo, al confirmarse una transacción se actualizan en la base de datos todos los cambios que haya sufrido el estado del objeto en memoria. Respecto al borrado de objetos persistentes, debe invocarse explícitamente utilizando el objeto `Session`. Hibernate ofrece además multitud de operaciones que dan control al usuario sobre cómo y cuándo se realiza la actualización [Hibernate2005].

Con respecto a la gestión de las identidades de los objetos Hibernate ofrece dos alternativas:

- Definir una propiedad "identificador" para cada clase persistente. En este caso el identificador de la propiedad siempre deberá ser `id`. Su valor será la clave primaria de la fila correspondiente en la tabla relacional. En este caso puede optarse por una gestión manual de la asignación de identificadores o dejar que Hibernate asigne automáticamente un valor a la propiedad `id`.

- Dejar que sea Hibernate quien gestione totalmente la generación y asignación de claves. En este caso el identificador asociado a una instancia puede obtenerse mediante el método `Session.getIdentifier(Object object)`.

En ambos casos, si se opta por delegar la generación de identificadores a Hibernate, se permite la configuración de diferentes estrategias de generación de identificadores. La estrategia más utilizada es la denominada `native` (nativa) la cual selecciona automáticamente entre las estrategias restantes según el SGBD concreto que haya sido configurado.

Para la persistencia de las asociaciones entre objetos éstas serán codificadas siguiendo las convenciones habituales para objetos POJO: con atributos para las asociaciones uno a uno y objetos `Collection` para los casos de cardinalidad múltiple, proporcionando los correspondientes métodos de acceso `get()` y `set()` en ambos casos. La única restricción es que los atributos del tipo `Collection` se declaren utilizando un interfaz, no una implementación concreta. Será en la especificación de los metadatos de mapeo donde se defina la cardinalidad de la relación y si ésta es bidireccional o no para permitir a Hibernate la gestión de su persistencia.

Cuando se utilicen objetos del tipo `Collection` hay que prestar una atención especial a los métodos `equals()` y `hashcode()` definidos en la clase `java.lang.Object`, de la que heredan todos los objetos Java. Por ejemplo, una colección del tipo `Set` (conjunto) utilizará el método `equals()` de cada objeto que se introduzca para prevenir elementos duplicados. La implementación por defecto de `equals()` realiza una comparación de las identidades Java de los objetos comparados, es decir, compara el valor de las referencias. Esta implementación resulta válida siempre y cuando sólo se mantenga una sesión abierta, puesto que Hi-

bernate garantiza que sólo habrá una única instancia por cada fila de la base de datos dentro de una misma sesión.

Si se manejasen instancias desde múltiples sesiones, sería posible tener un conjunto conteniendo dos objetos, cada uno de los cuales representan la misma fila de la tabla de la base de datos, pero que no tienen la misma identidad Java. Se trata por tanto de un caso semánticamente incorrecto que deberá ser solucionado sobrescribiendo el método `equals()` en las clases persistentes. Además, el método `equals()` debe de ser consistente con el método `hashcode()`, de tal manera que si dos objetos son iguales tengan el mismo código *hash* [Gosling96]. Por ello, si se redefine el método `equals()` el método `hashcode()` debe ser redefinido también de la manera apropiada. Una estrategia para la implementación de estos métodos es utilizar los identificadores de los objetos. En caso de `equals()` se compararían los identificadores de los objetos comparados y en caso de `hashcode()` se devolverían el *hash* del identificador del objeto. La limitación de esta estrategia es que debe proporcionarse una propiedad identificador a cada objeto del dominio.

Hibernate ofrece dos mecanismos de transitividad para la persistencia [Bauer2004].

- **Persistencia por alcance**: Es un mecanismo ampliamente utilizado [Atkinso95]. Su fundamento es que cualquier instancia se convierte en persistente en el mismo momento en el que la aplicación crea una referencia a dicha instancia desde otra instancia que ya es persistente.

- **Persistencia en cascada** (*cascading persistence*). Se trata de un mecanismo propio de Hibernate. Se basa en el mismo principio que la persistencia por alcance, pero permite especificar una estrategia de propagación en cascada para cada asociación, lo que ofrece una mayor flexibilidad y control para las transiciones entre los estados de persistencia.

Hibernate expone el metamodelo de los metadatos utilizados para configurar el mapeo objeto/relacional con el fin de que éste pueda ser consultado y modificado en tiempo de ejecución [Hibernate2005]. En el código de la Figura 3.10 se muestra cómo puede modificarse la configuración del mapeo utilizada para hacer persistente la clase `Person`, de tal manera que se añada una nueva propiedad `age` que se almacenará en la columna `AGE` de la tabla `PERSON`. Debe señalarse que, una vez que una `SessionFactory` ha sido creada, su configuración de mapeo es inmutable, por lo que para volver a obtener una nueva sesión con la nueva configuración es necesario crear una nueva fábrica. No obstante, una aplicación sí puede leer el metamodelo de la configuración utilizando el método `getClassMetadata()`.

```
Configuration configuration = new Configuration();

// Obtiene el metamodelo de la configuración del mapeo
objeto/relacional
PersistentClass personMapping =
      configuration.getClassMapping(Person.class);

// Define una nueva columna AGE para la tabla PERSON
Column column = new Column();
column.setType(Hibernate.INT);
column.setName("AGE");
column.setNullable(false);
column.setUnique(true);
personMapping.getTable().addColumn(column);
```

```
// Envuelve la columna en un objeto Value
SimpleValue value = new SimpleValue();
value.setTable( userMapping.getTable() );
value.addColumn(column);
value.setType(Hibernate.INT);

// Define la nueva propiedad para la clase Persona
Property prop = new Property();
prop.setValue(value);
prop.setName("age");
personMapping.addProperty(prop);

// Se construye una nueva SessionFactory utilizando la nueva
configuración
SessionFactory sessionFactory = configuration.buildSessionFactory();
```

Figura 3.10. Ejemplo de manipulación del metamodelo de configuración Hibernate.

En cuanto a su funcionamiento interno, Hibernate no trabaja procesando el *bytecode* Java de las clases compiladas como hace JDO [Roos2003]. En sus orígenes, utilizaba únicamente las posibilidades de introspección de la plataforma Java ofrecidas por la API *Reflection* [Sun97b]. No obstante, para implementar la estrategia *lazy fetching* a la hora de navegar por el árbol de objetos persistentes esta API se mostraba insuficiente.

Para implementar la estrategia de *lazy fetching* Hibernate usa *proxies* [GOF94]. Un *proxy* es un objeto que implementa el interfaz público de otro objeto e intercepta todos los mensajes enviados a ese objeto por sus clientes. En el caso de Hibernate, esta interceptación es utilizada para cargar el estado del objeto la primera vez que es utilizado. En el caso de las asociaciones con cardinalidad múltiple en alguno de sus extremos Hibernate utiliza implementaciones propias de los interfaces *Collection* definidos en `java.util`. En el caso de asociaciones de cardinalidad simple, es decir, un objeto que hace referencia a una clase definida por el usuario, se necesita un mecanismo más sofisticado.

Para este último caso, Hibernate no hace uso de las *Java Dynamic Proxy Clases* introducidas en la versión 1.3 de la plataforma [Sun99]. Este mecanismo permite crear en tiempo de ejecución un *proxy* que implemente una serie de interfaces. El problema de cara a su aplicación en Hibernate es que no sirve para construir un *proxy* si el objeto al que se accede no implementa ningún interfaz. Por ello, Hibernate apostó por utilizar CGLIB [CGLIB], un proyecto de código abierto que implementa un paquete de reflectividad alternativo al ofrecido por Java. En concreto, CGLIB permite crear *proxies* en tiempo de ejecución que hereden de una clase y que implementen interfaces, permitiendo de esta manera a Hibernate implementar la estrategia de *lazy fetching* de una manera completamente transparente.

Para actualizar un objeto en la base de datos cuando finalice una unidad de trabajo en curso y su estado haya cambiado es necesario utilizar un mecanismo que permita comprobar si el objeto está en un estado "sucio" o no. Hibernate utiliza para implementar esta funcionalidad una estrategia denominada "inspección" (*inspection)* [Bauer2004]. Lo que hace es comprobar el valor de las propiedades de un objeto al finalizar una transacción con una copia de su estado guardada cuando fue cargado desde la base de datos.

Otra alternativa para implementar este mecanismo es la estrategia denominada "interceptación" (*interception*). En este caso la herramienta objeto/relacional intercepta todas las asignaciones de valores a los campos persistentes del objeto de-

tectando así cuando se modifica su estado. Debido a las características de la máquina virtual de Java [Gosling96] está estrategia sólo puede ser implementada interviniendo en tiempo de compilación o en tiempo de carga de clases. Esta es la estrategia que utilizan las implementaciones de JDO [Roos2003].

3.2.6.2 Java Data Objects (JDO)

Java Data Objects (JDO) es una especificación [Sun2003] de un sistema de persistencia de objetos para el lenguaje Java basado en interfaces. Esta especificación describe el almacenamiento, consulta y recuperación de objetos de almacenes de datos.

Las raíces de JDO se encuentran en ODMG (*Object Data Management Group*) [Cattell94]. Este grupo definió un enlace o *binding* de su modelo de objetos [Cattell96] para el lenguaje Java[3] y los diversos fabricantes de bases de datos orientadas a objetos desarrollaron implementaciones para este *binding*. Este intento fue un fracaso, entre otras razones debido a que la total ausencia de estándares en el terreno de las bases de datos orientadas a objetos marcó el desarrollo del estándar ODMG, así como la no definición por parte del ODMG de un conjunto de pruebas de conformidad que asegurasen la interoperabilidad de las distintas implementaciones [Jordan2004]. Con la incorporación de la *Java Community Process* (JCP) se reemplazó este intento por la definición de una propuesta más ambiciosa: Java Data Objects.

La característica fundamental de JDO es que soporta la persistencia transparente, lo cual queda reflejado en los siguientes aspectos [Roos2003]:

- JDO gestiona de una manera transparente el mapeo de instancias JDO al almacén de datos subyacente, es decir, la desadaptación de impedancias [Maier89].

- JDO es transparente a los objetos Java que se hacen persistentes. No es necesario que las clases persistentes hereden de una clase raíz, añadir a las clases nuevos métodos o atributos ni alterar los modificadores de visibilidad de los miembros de éstas. En concreto, se soportan atributos privados así como atributos que no presenten métodos de acceso `get()` y `set()`.

- Con JDO puede trabajarse contra multitud de tipos de almacenes de datos que responden a distintos paradigmas. Entre otros: bases de datos relacionales, bases de datos orientadas a objetos, sistemas de archivos, documentos XML [W3C98] y datos almacenados en aplicaciones heredadas.

- JDO es transparente a su vez al almacén de datos utilizados, de manera que las aplicaciones pueden ser portadas para utilizar cualquier almacén de datos para el cual esté disponible una implementación JDO adecuada. La especificación JDO garantiza la compatibilidad binaria de las diferentes implementaciones, lo que significa que esta característica puede conseguirse incluso sin necesidad de recompilación.

- Si una aplicación hace referencia a un objeto persistente y altera de cualquier forma su estado persistente en memoria, la implementación JDO se encarga implícitamente de actualizar el almacén de datos cuando la

[3] ODM *Java Binding*

transacción sea confirmada. Así se elimina la necesidad de que el programador codifique reiteradamente operaciones explícitas de almacenamiento.

En comparación con otros estándares de la plataforma Java, JDO busca eliminar el código dedicado necesario cuando se utiliza JDBC [Fisher2003] y SQL [Melton93], que nada tiene que ver con los requerimientos de negocio de una aplicación. Por otra parte trata de evitar la complejidad del modelo de persistencia de los *Enterprise Java Beans* (EJB) [Sun2003d], estudiados en § 3.6.1.

Para utilizar JDO, en primer lugar se codifican las clases para las cuales se necesitan los servicios de persistencia, que habitualmente constituyen el modelo del dominio de objetos. Con el objetivo de mostrar cómo funciona JDO se va a desarrollar un sencillo ejemplo donde los objetos de dominio serán personas. El código de la clase persona se recoge en la Figura 3.11.

```
package example.jdo;

public class Person{
	protected String name;

	public void setName(String name){
		this.name = name;
	}

	public String getName(){
		return name;
	}
}
```

Figura 3.11. Código de la clase Persona.

A continuación se escribe un documento XML denominado "descriptor de persistencia". Este documento XML, en su forma más sencilla, únicamente identifica los nombres de las clases persistentes. Se ofrece la posibilidad de especificar qué elementos se hacen persistentes con un mayor nivel de detalle, por ejemplo, especificar qué campos de la clase son persistentes. El descriptor de persistencia que hace persistente la clase persona utilizada en el ejemplo se muestra en la Figura 3.12.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE jdo SYSTEM "file:///jdos/dtd/jdo.dtd">
<jdo>
<package name="example.jdo">
	<class name="Person" />
</package>
</jdo>
```

Figura 3.12. Descriptor de persistencia.

Finalmente, debe lanzarse un proceso de "realce" (*enhancement process*) que se encarga de leer el descriptor XML y añadir dentro de cada clase a la que se haga referencia los métodos adicionales que son necesarios para el establecimiento y recuperación de atributos por parte de una implementación JDO. Estos métodos vienen definidos en el interfaz `PersistenceCapable`. Las diferentes implementaciones de JDO realizan este realce a nivel de *bytecode* Java [Gosling96] aunque el estándar JDO [Sun2003] no especifica ningún mecanismo. Si bien no se recomienda, también es posible implementar el interfaz `PersistenceCapable` manualmente, sin necesidad de tener que utilizar una herramienta de realce.

En la invocación del proceso pueden generarse *scripts* en el lenguaje de definición de datos correspondiente para la definición de la estructura de almacenamiento necesaria en un almacén de datos específicos. Algunos vendedores de implementaciones JDO proporcionan una herramienta de definición de esquemas específica con este propósito, habitualmente para bases de datos relacionales, mientras que este paso no suele ser requerido para bases de datos orientadas a objetos.

Como es natural, no serán los objetos de dominio los que invoquen los servicios de persistencia, sino que serán los objetos de aplicación los que realicen esta tarea con el objetivo de hacer persistir y recuperar los objetos de dominio. El desarrollador codifica estas invocaciones utilizando un interfaz estándar JDO denominado `PersistenceManager`. En la Figura 3.13 se muestra un ejemplo de cómo se puede hacer persistente un objeto de la clase persona.

```
PersistenceManagerFactory persistenceManagerFactory = ... ;
PersistenceManager persistenceManager=
      persistenceManagerFactory.getPersistenceManager();
Transaction transaction = persistenceManager.getCurrentTransaction();
Person person = new Person();
person.setName("Pepe");

transaction.begin();
persistenceManager.makePersistent(person);
transaction.commit();
```

Figura 3.13. Ejemplo de persistencia de un objeto con JDO.

El ciclo de vida de los objetos se modela mediante estados, los cuales son potencialmente visibles para la aplicación. Por ejemplo, el estado "*hollow*" (hueco) especifica que los campos de un objeto aún no han sido leídos del correspondiente almacén de datos. La transición entre estados se realiza la mayor parte de las veces explícitamente con invocaciones por parte del desarrollador. En la

Figura 3.14 puede verse la transición de estados correspondientes a la operación de hacer una instancia persistente. También existen transiciones de estados implícitas, como por ejemplo cuando se modifica un campo de una instancia y esta pasa a marcarse como sucia, tal y como se muestra en la Figura 3.15. El borrado de objetos persistentes siempre será explícito haciendo uso de los servicios del objeto `PersistenceManager`.

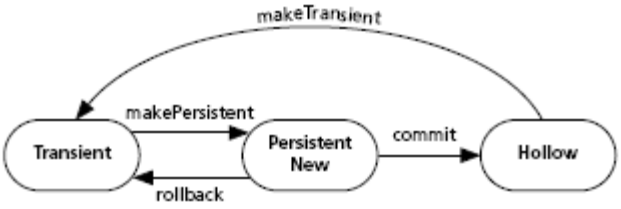


Figura 3.14. Estados correspondientes a la operación de hacer una instancia persistente.

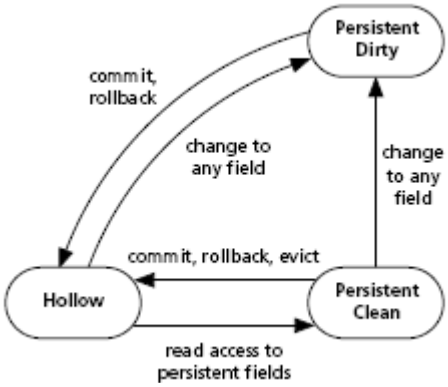


Figura 3.15. Estados correspondientes a la modificación de los campos de una instancia.

Con respecto a la recuperación de instancias persistentes del almacén de datos, JDO permite que las aplicaciones naveguen de una manera transparente por los atributos de un objeto persistente que referencian a otros objetos persistentes. Cuando la aplicación utiliza uno de estos objetos, aquellos que no están presentes en la caché del gestor de persistencia son cargados desde el almacén de datos. De esta manera se ofrece a la aplicación la impresión de que todo el grafo interconectado de instancias persistentes está disponible al instante en memoria. Así pues, el principal problema desde el punto de vista de la aplicación es cómo obtener la primera instancia persistente. JDO ofrece tres métodos [Roos2003]:

- Utilizar el identificador del objeto persistente para obtener una instancia de éste a través del gestor de persistencia. La aplicación debe por lo tanto ser capaz de construir el objeto identificador adecuado para realizar la consulta. Posteriormente se analizará como realiza JDO la identificación de objetos.

- Extensiones JDO. Se denomina extensión (*extent*) de una clase a todas las instancias persistentes de una clase o jerarquía de clases. Puede obtenerse la extensión de una clase a través del objeto `PersistenceManager`. y recorrerse las instancias persistentes haciendo uso de un iterador [GOF94]. Esta opción es recomendable cuando se desean procesar todas las instancias pero es muy ineficiente cuando reintenta recuperar una instancia específica o un grupo pequeño de ellas. En la Figura 3.17 se muestra un ejemplo de cómo recuperar todas las instancias de la clase persona que han sido almacenadas imprimiendo su nombre en la pantalla.

- Sistema de consultas JDO [Sun2003]. Este caso se trata de un sistema ofrecido por JDO para construir y ejecutar consultas mediante un mecanismo orientado a objetos e independiente del almacén de datos utilizado. Las consultas serán representadas mediante objetos que implementen el interfaz `Query`. Estos objetos consulta pueden construirse a través del gestor de persistencia y configurarse utilizando el interfaz definido. JDO define además el lenguaje JDOQL (JDO *Query Language*). Se trata de un sencillo lenguaje utilizado para poder personalizar determinados aspectos de las consultas tales como el orden de las mismas, la declaración de parámetros o los criterios de búsqueda, que serán representados a través de objetos filtro. Varias de las operaciones definidas en el interfaz `Query` (Figura 3.16) reciben como parámetro una cadena de caracteres que debe responder a este lenguaje. Su implementación corresponde al vendedor

de la implementación JDO y siempre utilizarán los métodos de ejecución más eficientes disponibles en el almacén de datos concreto utilizado.

Una vez que el descriptor de persistencia ha sido escrito, los objetos de dominio realzados, el medio de almacenamiento definido de la manera adecuada y se ha codificado la invocación al gestor de persistencia por parte de la aplicación, la aplicación está lista para ser ejecutada.

Figura 3.16. Interfaz Query

```
transaction.begin();
Extent personExtent =
      persistenceManager.getExtent(Person.class, false);
Iterator iterator = personExtent.iterator();

while (iterator.hasNext()) {
      System.out.println(iterator.next());
}

personExtent.close(iterator);
t.commit();
```

Figura 3.17. Recuperación de las instancias persistentes.

Debe hacerse hincapié en que el almacén de datos que utilice JDO no está limitado a un tipo concreto. JDO por sí mismo proporciona los interfaces mediante los cuales pueden invocarse los servicios de persistencia. Estos servicios serán invocados sobre una implementación JDO concreta, la cual será escogida no por su funcionalidad, que ya está especificada por JDO, sino precisamente por la calidad que ofrezca para trabajar contra un almacén de datos en concreta. Por ejemplo, para funcionar contra una base de datos relacional concreta, debería definirse el esquema de almacenamiento en esa base de datos y utilizar una implementación JDO adecuada. Del mismo modo debería hacerse para funcionar contra una base de datos orientada a objetos utilizando una implementación JDO alternativa. Lo más importante es que, desde el punto de vista del desarrollador, no se requiere trabajo para

migrar una aplicación que utilice JDO desde un almacén de datos a otro, ni siquiera cuando cambie el paradigma de almacenamiento.

Con el fin de mejorar la transparencia con la cual JDO puede ser aplicado a los modelos de objetos existentes, JDO define su propio concepto de identidad de objetos, que es independiente de los conceptos de igualdad y equivalencia del lenguaje Java [Gosling96]. En JDO existen tres tipos de mecanismos para establecer la identidad de los objetos [Roos2003]:

- **Identidad del almacén de datos (*datastore identity*)**. Se trata del mecanismo de identidad por defecto. Se asocia una identidad al objeto en el momento en el que éste se hace persistente. La manera de asignar un objeto identidad y la naturaleza depende de cada implementación JDO y del almacén de datos. Una vez que la identidad ha sido fijada puede ser utilizada en futuras peticiones para recuperar un objeto en particular.

- **Identidad de aplicación (*application identity*)**. En este caso es la aplicación la responsable de las identidades de las instancias, las cuales serán derivadas de los valores de un subconjunto de sus atributos persistentes. En el descriptor de persistencia se especifican uno o más campos persistentes que conformarán una clave primaria para la instancia, así como el nombre de la clase que representará el objeto identidad a establecer. Esta clase normalmente es programada por el desarrollador de la aplicación, aunque determinadas herramientas de realce son capaces de generarlas cuando realzan las clases que utilizan este mecanismo de identidad.

- **Identidad no duradera (*non-durable identity*)**. Se utiliza para objetos persistentes donde no tiene sentido distinguir unos de otros. Se utiliza por razones de eficiencia, para permitir la persistencia rápida de nuevas instancias.

El mecanismo de asignación de identidad deseado para cada instancia se especifica en el descriptor de persistencia. Internamente, la implementación de JDO es responsable de asegurar que existe, como mucho, una instancia persistente JDO asociada con un objeto específico del almacén de datos por cada gestor de persistencia. Este proceso es denominado en inglés *uniquing*.

JDO divide las instancias en objetos de primera clase y objetos de segunda clase [Sun2003]. Esta distinción es muy importante para el desarrollador, puesto que la manera de trabajar con instancias compartidas es muy diferente dependiendo de la clase de la instancia.

Los objetos de primera clase son instancias de una clase `PersistenceCapable` que tienen una identidad JDO y por lo tanto soportan la unicidad (*uniquing*) en la caché del `PersistenceManager`.

Cuando se modifican los valores de los atributos de un objeto de primera clase, el estado de ese objeto se marca como "sucio", estado en el que permanecerá hasta que se confirme la transacción a realizar (*commit*) o se cancele (*rollback*).

Un objeto de segunda clase es una instancia de una clase `PersistenceCapable`, o, en el caso de las clases de la API estándar de Java, una instancia de una clase que no implementa el interfaz `PersistenceCapable` pero para la que la implementación JDO proporciona un soporte específico de persistencia. Se diferencian de los objetos de primera clase porque no tienen una identi-

dad JDO y por lo tanto no soportan el proceso de unicidad (*uniquing*). Únicamente serán almacenados como parte de un objeto de primera clase.

Cuando se modifica un objeto de segunda clase éste no asume por sí mismo un estado sucio. En lugar de eso, transmite el hecho de que ha cambiado a su objeto de primera clase poseedor, el cual sí se marca como sucio.

La principal razón de ser de los objetos de segunda clase es permitir diferenciar los artefactos de programación que no tienen representación en el almacén de datos pero que son utilizados únicamente pare representar relaciones. Por ejemplo, una `Collection` de objetos `PersistenteCapable` podría no almacenarse en el almacén de datos, sino que sería utilizada para representar la relación en memoria. A la hora de efectuarse la transacción y hacer persistente el objeto, el artefacto en memoria se descarta y la relación se representa únicamente con las relaciones del almacén de datos.

Aunque la especificación de JDO no utiliza este término, en [Roos2003] se denominan a los *arrays* Java objetos de tercera clase. Esto es debido a que su soporte por parte de las implementaciones JDO es opcional. Desde el punto de vista práctico, el realce a nivel de *bytecode* de los *arrays* no es válido puesto que los *arrays* en la plataforma Java son objetos especiales que no tienen un fichero ".class" asociado. Lo que se recomienda en este caso es implementar la funcionalidad mediante la cual los cambios en los contenidos del *array* se reflejen marcando como sucio el objeto de primera clase contenedor del *array*.

JDO ofrece un sistema de persistencia por alcance [Atkinson95] a través del concepto de cierre transitivo de un objeto. Todos los objetos referenciados, directa o indirectamente, desde un objeto persistente son hechos persistentes. Esto significa que, aunque un objeto no sea marcado específicamente como persistente en el descriptor de persistencia, si forma parte del cierre de objetos referenciado a través de los atributos persistentes de otra instancia persistente, entonces el objeto es a su vez persistente.

JDO ofrece además la ilusión de que la aplicación tiene acceso en memoria al cierre transitivo de todas las instancias persistentes referenciadas desde cualquier instancia persistente, independientemente del hecho que la mayor parte de este conjunto de instancias, potencialmente enorme, estará normalmente en disco y no en realmente en memoria.

3.2.7 Sistemas de Gestión de Bases de Datos Orientadas a Objetos

Según Kim [Kim91] una base de datos orientada a objetos es una colección de objetos en los que su estado, comportamiento y relaciones son definidas de acuerdo con un modelo de datos orientado a objetos, y un sistema de gestión de bases de datos orientadas a objetos (SGBDOO) es un sistema de base de datos que permite la definición y manipulación de una base de datos orientada a objetos.

Los SGBD orientados de objetos, o lenguajes de programación de base de datos orientados a objetos según Cattell [Cattell94], combinan características de orientación a objetos y lenguajes de programación orientados a objetos con capacidades de bases de datos. Estos sistemas están basados en la arquitectura de un lenguaje de programación de bases de datos. Las aplicaciones son escritas en una extensión de un lenguaje de programación existente, y el lenguaje y su implementación (compilador, preprocesador, entorno de ejecución) han sido extendidos para

incorporar funcionalidad de base de datos. El objetivo de estos sistemas es llegar a integrarse con múltiples lenguajes, aunque esto puede suponer un problema debido a lo cercano de la asociación que se requiere entre el sistema de tipos de la base de datos y el sistema de tipos del lenguaje de programación.

La primera descripción realizada formalmente sobre los rasgos y características principales que debe tener un SGBDOO para poder calificarlo como tal, fueron especificadas en el *Object-Oriented Database System Manifesto* [Atkinson89]. En este manifiesto se clasifican las características en tres categorías: obligatorias, optativas y abiertas.

Las características obligatorias son aquellas características que se consideran esenciales en un SGBDOO. Un SGBDOO debe satisfacer dos criterios (Figura 3.18): debe ser un SGBD, y debe ser un sistema orientado al objeto; es decir, en la medida de lo posible deberá ser consistente con el actual conjunto de lenguajes de programación orientados a objetos. El primer criterio se traduce en cinco características: persistencia, gestión del almacenamiento secundario, concurrencia, recuperación y facilidad de consultas. El segundo criterio se traduce en ocho características: objetos complejos, identidad de objetos, encapsulamiento, tipos y clases, herencia, sobrecarga combinada con ligadura tardía, extensibilidad y completitud computacional.

Las características optativas son características que mejoran el sistema claramente, pero no son obligatorias para hacer de él un sistema de bases de datos orientado a objetos. Algunas de estas características son de naturaleza orientada a objetos (herencia múltiple) y se incluyen en esta categoría porque, aunque hacen que el sistema esté más orientado a objetos, no pertenecen a los requisitos primordiales. Otras características son simplemente características de la base de datos (por ejemplo, gestión de las transacciones de diseño) que mejoran la funcionalidad del sistema, pero no pertenecen a los requisitos primordiales.

Estas características son:

- Herencia múltiple.
- Chequeo e inferencia de tipos.
- Distribución.
- Transacciones de Diseño.
- Versiones.

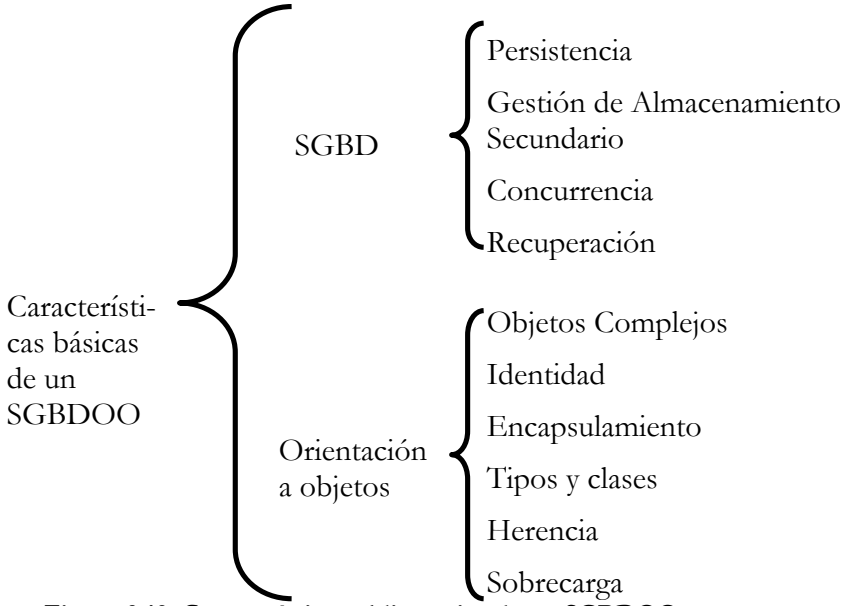


Figura 3.18. Características obligatorias de un SGBDOO.

Finalmente las características abiertas ofrecen grados de libertad para los desarrolladores del sistema de base de datos orientado a objetos. Difieren de las obligatorias en el sentido de que todavía no se ha llegado a consenso alguno respecto a ellas en la comunidad científica. Se diferencian también de las opcionales en que no se conoce cuáles de sus alternativas están más o menos orientadas al objeto.

Estas características son:

- Paradigma de programación.
- Sistemas de representación.
- Sistema de tipos.
- Uniformidad.

A pesar de las especificaciones realizadas en este manifiesto los SGBDOO presentaban muy pocas características en común que facilitasen la portabilidad entre las aplicaciones [Martínez2001]. En 1991, Rick Cattell encabezó la formación del grupo ODMG (*Object Data Management Group*) comenzándose a trabajar en la elaboración de un estándar. Un primer borrador fue publicado en 1992, y la primera versión del mismo en 1993 [Cattell94b]. La última versión (3.0) ha sido publicada en septiembre de 1999 [Cattell99]. Este estándar se construye sobre estándares existentes tanto de lenguajes de programación como de bases de datos, y trata de simplificar el almacenamiento de objetos y garantizar la portabilidad de la aplicación.

La especificación ODMG define una API, un lenguaje de consulta, un lenguaje de metadatos y ligaduras (*bindings*) para los lenguajes de programación C++, SmallTalk y Java[4], con el objeto de que éstos puedan actuar como lenguajes anfitriones (*host languages*). Aunque la mayor parte de los SGBDOO ofrecen cierto soporte al estándar no existe ninguno que lo soporte en su totalidad [Bauer2004], es decir, el intento de estandarizar las bases de datos orientadas a objetos fracasó. El grupo ODMG fue disuelto en 2002. A continuación, se revisan diversos problemas

[4] El *binding* para Java evoluciono hacia el estándar Java Data Objects (§ 3.2.6.2).

existentes en los SGBDOO y a los que el estándar ODMG pretende dar respuesta [Martínez2001].

En primer lugar, la mayoría de los SGBDOO vienen en forma de API (Figura 3.19), que será utilizada desde las aplicaciones que emplean un lenguaje de programación orientado a objetos como C++ o Java. El principal inconveniente es que cada gestor suele emplear su propia API, lo que dificulta la legibilidad y el mantenimiento del código, restringe la flexibilidad y la portabilidad de la aplicación, y desde el punto de vista del usuario incrementa la complejidad.

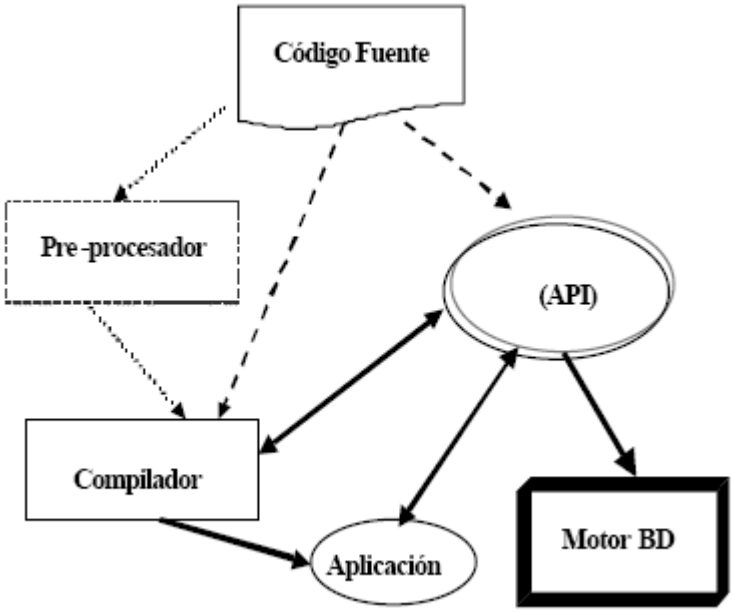


Figura 3.19. Inclusión de funcionalidades de bases de datos en lenguaje de programación.

Como estos problemas eran evidentes, y además una de las principales lacras con las que se encontraban los SGBDOO, ODMG [Cattell99] define una API estándar para interactuar con el SGBDOO desde el lenguaje de programación. El principal problema es que, aunque, inicialmente la mayoría de los vendedores de BDOO se comprometieron a adoptar dicho estándar, hoy en día cada uno emplea el API a conveniencia.

Otro problema presente en las diferentes implementaciones de SGBDOO es la carencia de interoperabilidad. Uno de los objetivos ideales perseguidos por los SGBDOO es la integración con múltiples lenguajes de programación, de forma que objetos que hayan sido creados en una base de datos utilizando un lenguaje de programación puedan ser recuperados desde otra base de datos con otro lenguaje de programación orientado a objetos diferente. Es decir, cualquier base de datos ha de poder recuperar los objetos almacenados independientemente del lenguaje de programación y de la base de datos con la que hayan sido creados.

El estándar ODMG también intenta dar solución a este problema. Propone, por un lado, un lenguaje de definición de objetos (ODL) basado en el IDL de CORBA [OMG98], que facilita la portabilidad de esquemas entre SGBDs, y siempre que los sistemas sean compatibles con el estándar. Las ligaduras del ODL a lenguajes como C++, Smalltalk y Java están diseñadas para ser introducidas fácilmente en la sintaxis declarativa de su lenguaje de programación, sin embargo, debido a las diferencias inherentes en los modelos de objetos nativos a estos lenguajes de programación, no siempre es posible mantener una semántica consistente entre lenguajes de programación (versiones específicas de ODL).

Por otro lado, ODMG propone un formato para el intercambio de objetos (OIF), que permite el intercambio de objetos entre bases de datos. Este lenguaje se emplea para volcar y cargar el estado actual de una base de datos de objetos desde un fichero o conjunto de ficheros. Sin embargo, esto implica que sería necesario un proceso independiente para la importación y exportación de los datos.

Algunos sistemas como POET [Poet2000], por ejemplo, soportan cierta interoperabilidad entre los *bindings* de ODMG para Java y C++ pero siempre dentro del propio producto e imponiendo ciertas restricciones. Así, el *binding* de C++ para conseguir interoperabilidad con el de Java no debe soportar, por ejemplo, herencia múltiple.

Otros problemas existentes en los SGBDOO actuales son su falta de portabilidad y su limitada extensibilidad [Martínez2001]. La mayoría de los SGBDOO existentes son construidos para una determinada plataforma, lo que dificulta la migración tanto del propio gestor como de los datos hacia otras plataformas. Además, éstos se configuran como auténticas cajas negras construidas en torno a mecanismos muy rígidos que no contemplan la incorporación de nuevas técnicas, por ejemplo, la posibilidad de añadir nuevas técnicas de indexación.

3.2.7.1 POET

Poet [Poet2000] es un SGBDOO que permite a un programa Java o C++conectarse a cualquier Servidor de Objetos de Poet, posibilitando de esta forma acceder a los mismos objetos (base de datos) desde cualquier aplicación Poet (independiente del *binding* con el que haya sido desarrollada), pudiéndose incluso copiar la base de datos a máquinas que se ejecutan sobre sistemas operativos diferentes. El formato físico es binario, compatible entre plataformas y lenguajes de programación.

Este SGBDOO soporta el estándar ODMG [Cattell96] implementando los *bindings* para Java y C++. Como lenguaje de consulta utiliza OQL.

Poet, en su versión para Java, trabaja con cuatro elementos básicos: la base de datos propiamente dicha, el diccionario, un preprocesador y un fichero de configuración.

La base de datos almacena todos los objetos que crea y usa la aplicación. El diccionario, llamado también esquema de clases, almacena para cada clase que puede ser persistente el nombre y el identificador de la clase, descripciones de los miembros de cada clase y descripciones de los índices definidos por la clase (si los hay). Además, conoce también las superclases, subclases y las interfaces implementadas. Puede ser compartido por cualquier número de bases de datos, de forma que si la aplicación abre varias bases de datos que comparten un diccionario, éste es cargado una única vez. El fichero de configuración, en el que se especifican entre otras cosas las clases susceptibles de ser persistentes, es la entrada para el preprocesador. Tanto el diccionario como la base de datos son generados automáticamente por el preprocesador `ptj`.

La base de datos puede estar formada por un único fichero (por defecto), o por varios: `objects.dat`, que es el fichero de datos que contiene los objetos almacenados y, `objects.idx`, que contiene la información de los índices para los objetos en la base de datos.

En relación con la persistencia en POET hay que tener en cuenta las siguientes consideraciones:

- Es necesario especificar todas aquellas clases que son susceptibles de ser persistentes en el fichero de configuración (`ptj.opt`). Este fichero de configuración es leído por el procesador, y éste registra la información en el diccionario y añade el código, métodos y clases auxiliares a los ficheros de clases Java, para convertirlas en clases capaces de ser persistentes.

- Una vez que una clase ya es susceptible de contener instancias persistentes, para convertirse realmente en persistentes éstas deberán ser referenciadas por otros objetos persistentes.

- Las extensiones de las clases en POET son creadas por el programador a conveniencia pero son mantenidas por el propio gestor, por lo que no disponen de métodos para añadir o eliminar elementos, pero si para navegar en la extensión, recuperar un elemento, avanzar al siguiente, etc. Para poder crear un objeto de tipo `Extent` asociado a una clase, es necesario haber especificado en el fichero de configuración que dicha clase va a disponer de extensión.

- Es posible asociar un nombre a un objeto para su posterior recuperación, pero también pueden existir objetos sin nombre que pueden ser recuperados empleando la extensión de la clase.

POET dispone de un recolector de basura (`ptgc`) que se encarga de borrar todo objeto que no es alcanzable desde cualquier objeto ligado a la base de datos. El principal problema con el recolector es que elimina cualquier objeto que no tenga nombre asociado. Es por eso que para evitar este problema el recolector acude a un esquema simple que se basa en no borrar el objeto si está bajo una de estas cuatro condiciones:

- El objeto tiene asociado un nombre.

- La clase del objeto es marcada como no débil en la declaración de la clase en el fichero de configuración de POET. Por defecto todas las clases se consideran débiles, lo que quiere decir que el recolector puede borrar sus instancias. Cuando se marca una clase como no débil todas las extensiones de sus subclases son consideradas también como no débiles.

- La clase de los objetos hereda la "no debilidad" recursivamente desde una de sus clases base.

- El objeto es alcanzable desde otro objeto que cumple una o más de las tres condiciones precedentes.

En POET todas las operaciones de base de datos deben realizarse dentro de una transacción. Las transacciones en POET soportan *checkpoints* que son puntos de ejecución internos que no finalizan la transacción pero que escriben en la base de datos las modificaciones realizadas en los objetos hasta ese momento, aunque manteniendo sus bloqueos. Estos *checkpoints* son útiles sobre todo cuando hay un gran número de objetos modificados en una transacción y se quiere evitar una gran operación `commit` al finalizar la transacción, o bien cuando se quiere hacer una consulta basada en los datos de los objetos modificados sin necesidad de salir de la transacción. POET también soporta transacciones anidadas (pero no *checkpoints* dentro de éstas).

En cuanto al bloqueo, POET permite un bloqueo implícito por defecto (de lectura cuando se recupera un objeto, o de escritura cuando se intenta modificar) o un bloqueo explícito con cuatro niveles (*read, upgrade, write* y *delete*).

POET Java soporta aplicaciones, con un único hilo que emplea una transacción en cada momento, con múltiples hilos y cada hilo asociado con una transacción, y con múltiples hilos que comparten una transacción (en este caso el programador debe añadir sus propios controles de concurrencia).

Una clase siempre tiene al menos un índice basado en los identificadores asociados internamente a los objetos, y que determina el orden de los objetos cuando se recuperan desde la extensión. Pero POET permite definir índices adicionales para la extensión de la clase y proporciona métodos para determinar y asignar que índice es usado. Estos índices deben ser especificados en el fichero de configuración, asociados con la extensión de la clase correspondiente, y pueden contener más de un campo como clave. El mantenimiento de los índices es realizado automáticamente por el sistema.

A diferencia de otros sistemas comerciales existentes en el mercado, POET no contempla la extensibilidad del sistema no permitiendo ni siquiera la incorporación de nuevos métodos de acceso.

3.2.7.2 Jasmine

Jasmine es una base de datos que soporta estructuras de datos complejas, herencia múltiple, propiedades y métodos a nivel de instancia y a nivel de clase, colecciones y métodos a nivel de colección, especificación de integridad referencial, y a diferencia de otros gestores la clase contiene su propia extensión. Proporciona dos posibilidades para el desarrollo de aplicaciones:

- **API Jasmine C.** Es un conjunto de llamadas a funciones que manipulan la base de datos Jasmine, a través de C, C++ y otros lenguajes de programación. Cuando se programa con este API, ODQL proporciona acceso a la base de datos, e incluye funciones para definiciones, consultas y manipulación de objetos. El lenguaje anfitrión permite realizar las operaciones no relacionadas con la base de datos y operaciones con datos externos.

- **JADE** (*Jasmine Application Development Environment*). Es un entorno de desarrollo gráfico que permite trabajar directamente con las clases e incluso con las instancias definidas en una base de datos. Permite diseñar, hacer prototipos y desarrollar aplicaciones con Jasmine con una interfaz de arrastrar y soltar.

Además de con C y C++, Jasmine proporciona un *binding* para Java creando una clase Java para cada clase Jasmine. Proporciona también una interfaz para cualquier entorno que soporte ActiveX, y mediante un conjunto de herramientas (WebLink), proporciona una forma de acceso a la base de datos mediante Internet empleando páginas HTML.

La arquitectura de Jasmine [CAI98] (Figura 3.20) está compuesta por cuatro módulos principales: el servidor de la base de datos, el cliente, los almacenes, y las librerías de clases.

- **Almacenes.** Los almacenes son los contenedores físicos de datos dentro de la base y contienen tanto metadatos como datos de usuario. Cada al-

macén contiene una o más familias de clases, dónde una familia de clases es una colección de clases relacionadas entre sí a las que se les da un nombre. Físicamente un almacén es un conjunto de ficheros, cuyo tamaño es definido por el usuario cuando lo declara.

- **Servidor de Base de Datos.** Es el servidor que controla todo tipo de operaciones sobre los datos almacenados en los almacenes. Jasmine se basa en una filosofía de clientes ligeros (*thin client)*, de forma que los métodos complejos son ejecutados en el servidor, encontrándose así bajo su sistema de transacciones y seguridad.

- **Clientes.** Representan el entorno donde se ejecutan las aplicaciones que acceden a la base de datos.

- **Librería de clases.** Son grupos de familias de clases que están predefinidas y distribuidas con Jasmine. Entre ellas destacan la librería de clases multimedia (para el almacenamiento de diferentes datos multimedia), y la familia de clases SQL que permiten el acceso y actualización de datos de otras bases de datos, incluso relacionales (Oracle, Sybase, Informix, etc.).

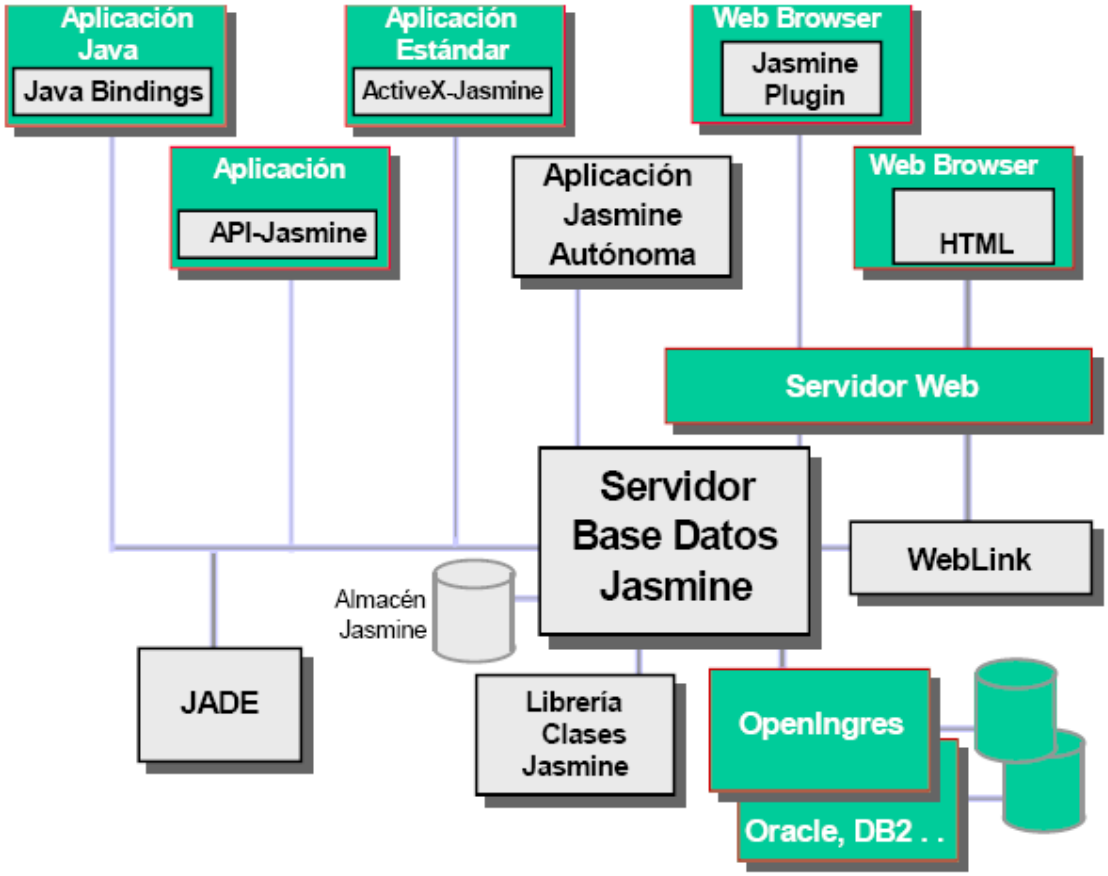


Figura 3.20. Arquitectura de Jasmine.

La implementación del gestor de base de datos está organizada en capas [Ishikawa96] con dos partes principales (Figura 3.21), la capa del tratamiento de datos, y la capa de tratamiento de objetos. La capa de tratamiento de datos está basada en una ampliación de la tecnología relacional.

El subsistema de manejo de datos implementa los índices y las relaciones entre objetos soportando únicamente cuatro tipos de relaciones: secuenciales, B-

Tree, *hash* e internas, siendo la estructura de la tupla independiente del tipo de relación. La capa relacional proporciona funciones para ejecutar operaciones sobre conjuntos, como un álgebra relacional extendida. La capa de tupla proporciona funciones a nivel de tuplas (*scan, raster, delete, insert*, etc.), y la capa de almacenamiento proporciona funciones que incluyen E/S de disco, *buffering* de páginas, transacciones y control de concurrencia y recuperación.

En el subsistema de manejo de objetos, el sistema realiza automáticamente la traducción de objetos en relaciones; la información sobre esta traducción, se almacena en las clases. Todas las instancias intrínsecas a una clase son almacenadas en una única relación, haciendo corresponder una instancia a una tupla, y un atributo a un campo. Las instancias intrínsecas a una superclase y las intrínsecas a subclases son almacenadas en relaciones separadas, de tal forma que al crear o destruir instancias intrínsecas a una clase dada, no se tiene que propagar ninguna modificación a las clases o subclases asociadas. El código fuente y compilado de los métodos y los *demons* es almacenado también en relaciones de las que se recuperan y utilizan durante la optimización de las consultas. Los métodos polimórficos son traducidos en sus correspondientes funciones de implementación.

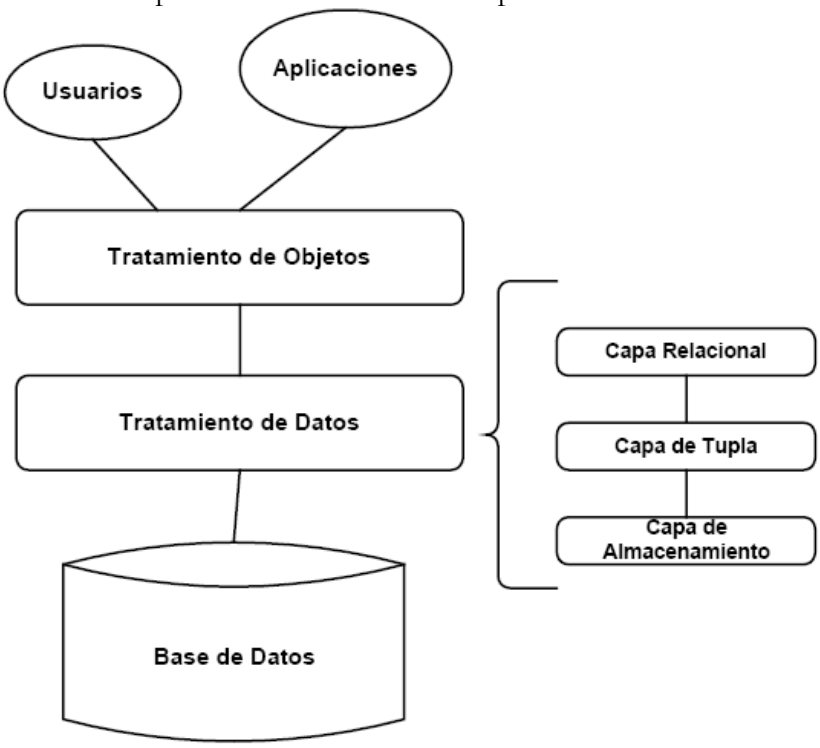


Figura 3.21. Modelo en capas del gestor de bases de datos Jasmine.

Jasmine incorpora como lenguaje de programación para la construcción de aplicaciones avanzadas un lenguaje (Jasmine/C) que integra un lenguaje de propósito general C y un lenguaje de base de datos en un contexto orientado a objetos. El compilador de Jasmine está implementado para que utilice un compilador de C. Los programas de aplicación, escritos en Jasmine/C, son precompilados a programas C, los cuales son compilados y enlazados con la *runtime support library*. El preprocesamiento, se utiliza para obtener el máximo rendimiento de la portabilidad y la optimización de código del compilador de C.

El lenguaje de consulta de Jasmine es ODQL (*Object Database Query Language*). Es un lenguaje utilizado para hacer consultas y manipular los objetos de la base de datos. Expresa la lógica de las operaciones a través de métodos definidos en la

propia base de datos, que son ejecutados en el servidor. ODQL puede emplearse, además de con API C, interactivamente en un terminal utilizando el intérprete de ODQL.

Con relación a las consultas hay que tener en cuenta las siguientes consideraciones:

- Este lenguaje permite consultas dinámicas.
- Una consulta en Jasmine devuelve todas las instancias intrínsecas de la clase, y las intrínsecas a sus subclases que cumplan las condiciones necesarias.
- Cuando en una consulta se hace una llamada a un método definido en una superclase, va a ser invocado automáticamente ese método sobre instancias de la propia clase, y sobre las de cada subclase definida, siendo posible en algunos casos que el tratamiento de las instancias sea distinto en unas clases o en otras, debido a la posibilidad de redefinición de métodos heredados.
- Es posible introducir métodos en el cuerpo de las consultas, e inversamente también es posible que los métodos puedan llevar especificaciones de consultas (atributos virtuales). Además, el usuario puede invocar consultas directamente desde los programas de aplicación. Para ello se introducen variables conjunto, que pueden contener los conjuntos de objetos devueltos por la consulta.

3.2.7.3 FastObjects .NET

FastObjects .NET es un sistema de persistencia para la plataforma .NET que trabaja sobre la base de datos orientada a objetos de Poet FastObject [Poet2004].

El sistema puede utilizarse con todos los lenguajes gestionados (*managed languages*) de la plataforma .NET [Thai2002], si bien impone ciertos requisitos al modelo de datos a usar. Persigue ofrecer persistencia transparente para la plataforma .NET, eliminando la necesidad de utilizar sentencias SQL para recuperar, insertar o actualizar objetos en la base de datos.

A la hora de recuperar un árbol de objetos de la base de datos, utiliza un sistema de "carga perezosa" (*lazy loading*). Únicamente los objetos a los que realmente se accede son recuperados de la base de datos, los que no son accedidos no existen en memoria. Tan pronto como un objeto es accedido se recupera de la base de datos. Desde el punto de vista del usuario, esta estrategia de recuperación es totalmente transparente, una vez recuperado explícitamente el conjunto de objetos raíz del árbol.

FastObjects.NET implementa también una estrategia de "seguimiento de cambios" (*change tracking*), que permite la acometida de todos los cambios hechos en la base de datos sin utilizar ninguna sentencia específica de almacenamiento, aunque la transacción deberá ser delimitada programáticamente.

Las clases que pueden ser hechas persistentes en .NET son escritas de la misma forma que cualquier otra clase con la adición de una serie de atributos a las clases y sus miembros. FastObjects .NET hace uso de los atributos personalizados (*custom attributes*) .NET [Thai2002] para proporcionar los metadatos necesarios para

la persistencia de objetos. Para designar una clase como "capaz de persistir" (*persistence capable*) debe añadirse el atributo `Peristent` a la definición de la clase. En la Figura 3.22 puede se muestra cómo puede hacerse persistente a una clase Persona escrita en C#.

```
[Persistent]
class Person
{
      Person() {
            // . . .
      }
      // . . .
}
```

Figura 3.22. Declaración de una clase como persistente en FastObjects .NET.

FastObjects.NET hace uso de una proceso de realce (*enhancement*) que reescribe el código intermedio (MSIL, *Microsoft Intermediate Language*) de en los archivos ensamblados .NET. Este proceso añade el código necesario para la implementación de los mecanismos de persistencia, además de los métodos de acceso especiales que permiten realizar el seguimiento de cambios.

Todas las clases persistentes deben tener un constructor sin argumentos. Este constructor puede ser privado. Si la clase no presenta este constructor, la herramienta de realce de FastObjects notificará un error.

El atributo `[Persistent]` no se hereda. Debe ser utilizado con todas las clases de una jerarquía que se deseen hacer persistir. Esto significa que una clase puede ser persistente incluso aunque la clase padre no lo sea. El usuario puede, de este modo, extender clases que no fueron diseñadas para ser persistentes.

Los objetos persistentes en una aplicación .NET se gestionan dentro del contexto de un objeto "ámbito" (*scope*) representado por una instancia que implementa el interfaz `IObjectScope`. Las instancias `IObjectScope` se obtienen de una instancia de la base de datos utilizada.

Esta base de datos se representa utilizando una instancia de la clase `Database`. La base de datos FastObjects utilizada realmente se especifica utilizando una URL. En la Figura 3.23 se muestra cómo realizar la conexión con la base de datos FastObjects.

```
Database db = Database.Get( "fastobjects://LOCAL/MyBase" );
```

Figura 3.23. Obtención de una instancia de la base de datos.

Una vez que se ha establecido una conexión con la base de datos, se utiliza un objeto `IObjectScope` para acceder a los objetos persistentes. Este objeto gestionará los objetos persistentes de la aplicación. En la Figura 3.24 se muestra cómo puede obtenerse un objeto "ámbito" a partir de la base de datos creada en el código de la Figura 3.23.

```
IObjectScope scope = db.GetObjectScope();
```

Figura 3.24. Obtención de un objeto IObjectScope.

La primera vez que se crea un nuevo objeto, independientemente de que se corresponda con una clase persistente o no, el objeto se encuentra en un estado "*transient*" (volátil o temporal). Los objetos en este estado no se almacenan en la base de datos. Para hacerlos persistentes debe invocarse al método `Add()` del objeto

"ámbito". Este método toma un único parámetro, el objeto que se desea hacer persistir. La persistencia de dicho objeto será gestionada a partir de dicha llamada por la instancia `IObjectScope` utilizada. Todos los objetos de clases persistentes que sean referenciados desde los objetos explícitamente introducidos dentro del objeto `IObjectScope,` serán a su vez introducidos dentro de dicho ámbito. FastObjects implementa de este modo un mecanismo de persistencia por alcance [Atkinson95] para la transitividad de la persistencia. En la Figura 3.25 se muestra un ejemplo de cómo puede añadirse un objeto raíz al ámbito de la persistencia gestionada por FastObjects .NET. Nótese que todas las operaciones con objetos persistentes tienen que ser realizadas dentro de una transacción explícitamente activada.

```
IObjectScope scope = Database.Get( dbURL ).GetObjectScope();
scope.Transaction.Begin();

Person person = Person ();
scope.Add(person);
scope.Transaction.Commit();
scope.Dispose()
```

Figura 3.25. Añadir un objeto persistente raíz a la base de datos.

Cuando se modifique el valor del estado de los objetos éstos serán movidos a un estado "sucio" automáticamente. Los objetos en este estado serán actualizados en la base de datos cuando se acometa la transacción activa.

El *framework* .NET define su modelo de datos a través del Sistema Común de Tipos (*Common Type System,* CTS) [Thai2002]. Para añadir la capacidad de persistir a los tipos de datos del *framework* .NET, FastObjects.NET define su propio modelo de datos, que es muy similar al Sistema Común de Tipos, aunque presenta algunas limitaciones.

FastObjects .NET soporta un conjunto restringido de clases e interfaces de la librería de colecciones .NET, recogida en el espacio de nombres `System.Collections`:

- `ICollection`
- `IList`
- `IDictionary`
- `ArrayList`
- `Queue`
- `Stack`
- `HashTable`
- `SortedList`

Además, también pueden utilizarse clases que deriven de `System.Array`.

No obstante, en la implementación actual de FastObjects .NET únicamente se soportan *arrays* persistentes de una dimensión. Tampoco es posible utilizar *arrays* de *arrays* persistentes.

Cualquiera de estas clases o interfaces pueden utilizarse para definir datos miembros de clases persistentes definidas por los usuarios. Sin embargo, no pueden

almacenarse instancias de estos tipos directamente en la base de datos. Tienen que formar parte de una clase definida por el usuario. Estos objetos siempre serán almacenados, recuperados y eliminados junto con los objetos que los contienen.

Para la recuperación de objetos persistentes FastObjects .NET ofrece diversos mecanismos.

- **Navegación de objetos**. Una vez obtenido un objeto de la base de datos utilizando cualquiera de los demás métodos, puede navegarse a través de los objetos referenciados por él, utilizando las sentencias habituales del lenguaje .NET que haya sido seleccionado. La carga de objetos se realizará de manera transparente al programador.

- **Utilizando el identificador del objeto**. Todos los objetos en las bases de datos FastObjects tienen un identificador único. Utilizando este identificador, pueden recuperarse objetos persistentes de la base de datos. Los identificadores de los objetos se representan mediante el interfaz `IObjectId`. Para obtener los identificadores utilizados se utiliza el método `GetObjectId()` del interfaz `IObjectScope`, y para obtener un objeto a partir de su identificador, se utiliza el método `GetObjectById()`.

- **Resolución de objetos**. Se trata de un mecanismo que permite forzar la carga de objetos desde la base de datos. En vez de utilizar la estrategia por defecto de carga perezosa de objetos, mediante la cual las referencias a objetos se rellenan únicamente con datos de la base de datos cuando son utilizadas, fuerza a que a que, dado un objeto, se carguen todos los datos a los que referencia.

- **Extensión de clase (*class extent*)**. Una extensión de una clase es la colección de todas las instancias de dicha clase y sus subclases. Su principal propósito es permitir el acceso a una enumeración de todos los objetos instanciados de una clase persistente dada. Para obtener la extensión de una clase se utiliza el método `GetExtent()` del interfaz `IObjectScope`.

- **Consultas**. FastObjects soporta el uso de OQL (*Object Query Language*) [Cattell99]. Las sentencias OQL serán especificadas como cadenas de caracteres.

3.2.8 Aportaciones y Carencias de los Sistemas Estudiados

3.2.8.1 Transparencia

Desde el punto de vista de transparencia de la persistencia, los sistemas estudiados ofrecen resultados muy diferentes.

La necesidad de implementar el interfaz `java.io.Serializable` que impone el mecanismo de serialización de la plataforma Java viola el requisito § 2.1.1, así como la necesidad de implementar métodos especiales en las clases persistentes para escribir y recuperar el estado de los objetos de dominio cuando se necesiten realizar tratamientos especiales.

El mecanismo de serialización de Java Beans con XML introducido en la JDK 1.4 requiere que los objetos a serializar expongan su estado a través de propiedades siguiendo los patrones de nomenclatura de los Java Beans [Sun96]. Si no se desean modificar las clases persistentes, se permite utilizar objetos delegados que se encarguen de la persistencia, pero siempre se mantiene el requisito fundamental de este sistema: que exista algún modo de recrear el estado del objeto persistente utilizando únicamente los métodos que se proporcionan en el interfaz público del objeto. Por ello, tampoco este sistema verifica el requisito § 2.1.1, puesto que si las clases persistentes no cumplen este principio fundamental deberán ser modificadas.

En cuanto a JDBC y SQLJ, los requisitos de transparencia (§ 2.1) y de adaptabilidad (§ 2.2) no son aplicables plenamente puesto que su único fin es servir de puente entre el lenguaje Java y el lenguaje SQL. Ambas opciones facilitan la interacción con un SGBD relacional utilizando sentencias SQL. Este código SQL queda incrustado dentro del código Java por lo que el desarrollador debe aplicar artefactos de diseño como el patrón *Data Access Object* (DAO) [Alur2001] para separar explícitamente la incumbencia de la persistencia de la lógica de negocio (§ 2.1.1). SQLJ requiere utilizar un preprocesador que analice las sentencias incrustadas dentro del código Java por lo que viola el requisito § 2.1.5. JDBC utiliza una API Java y sentencias SQL en forma de cadenas de caracteres Java por lo que no necesita un proceso de desarrollo diferente al de cualquier aplicación Java.

Los mejores resultados en cuanto a transparencia en los sistemas estudiados se obtienen con las herramientas de mapeo objeto/relacional, si bien con ninguna se consigue separar la competencia de la persistencia de una manera total. En el caso de JDO, puede conseguirse la persistencia de los objetos de dominio especificando los metadatos necesarios en descriptores de persistencia XML, sin que el código de las clases persistentes tenga que ser modificado (requisito § 2.1.1). No obstante, la distinción entre objetos de primera clase y objetos de segunda clase viola el requisito § 2.1.3. En concreto, la especificación no obliga a que los *arrays* sean considerados objetos de primera clase, lo que obliga a descartar la persistencia de la mayor parte de las clases `Collection` desarrolladas por el usuario. Para mitigar esto, se obliga a que las implementaciones soporten algunas variantes específicas de las clases `Collection`, por ejemplo, `HashSet`, pero no se requiere soporte para todas las clases de la plataforma.

La especificación de JDO obliga a que todas las clases persistentes desarrolladas por el usuario implementen el interfaz `PersistenceCapable`. Aunque esta implementación puede hacerse manualmente (violación del requisito § 2.1.1), la especificación define un proceso de realce estándar del *bytecode* contenido en el archivo `.class` que puede ser realizado automáticamente por una herramienta sobre las clases compiladas (violación del requisito § 2.1.5).

En el caso de Hibernate, si se utiliza el sistema de anotaciones para especificar los metadatos de persistencia, se debe modificar el código fuente de las clases persistentes, quedando la incumbencia de la persistencia mezclada con la lógica de negocio (violación del requisito § 2.1.1). Si se utilizan ficheros XML para especificar estos metadatos, no necesita modificarse el código fuente de las clases persistentes (requisito § 2.1.1) pero la implementación de Hibernate impone ciertas restricciones.

Gracias a la utilización de *proxies* para generar el código relativo a los mecanismos de persistencia y a la utilización del método de *inspection* (inspección) para la detección de los estados "sucios", no se necesita un proceso adicional de realce a

nivel de *bytecode* (requisito § 2.1.5). No obstante, la utilización de *proxies* obliga a que, cuando se utilicen objetos `Collection` como atributos de clases persistentes, éstos sean declarados en base a sus interfaces. Aunque Hibernate soporta el uso de *arrays*, recomienda el uso de objetos `Collection` con los que puede utilizar *proxies* específicos para implementar la estrategia de *lazy fetching* por razones de eficiencia (violación del requisito § 2.1.3). Además, la utilización de *proxies* y el método de inspección imponen a las clases persistentes el requisito de proporcionar un constructor sin argumentos. Finalmente existe la restricción de tener que implementar de manera apropiada los métodos `hashcode()` y `equals()` cuando se utilicen varias sesiones y objetos `Collection` tal y como se vio en § 3.2.6.1. Todo ello hace que el requisito § 2.1.1 no sea satisfecho plenamente por Hibernate.

FastObjects .NET hace uso del sistema de anotaciones mediante atributos .NET para proporcionar los metadatos de persistencia, por lo que viola el requisito § 2.1.1 al tener que modificar el código fuente de las clases que se desean hacer persistir. Además requiere un proceso de realce a nivel de código IL (violación del requisito § 2.1.5). Con respecto al soporte las colecciones del *framework* .NET, FastObjects .NET únicamente soporta un determinado número de interfaces y clases del *framework*. En el caso de los *arrays*, impone la restricción de que éstos sean unidimensionales No se verifica por lo tanto el requisito § 2.1.3 en este sistema de persistencia.

Hibernate, FastObjects.NET y JDO pueden ser utilizados como sistema de persistencia por cualquier aplicación cuyo modelo de dominio pueda representarse como un modelo de objetos complejo. De hecho, el gran valor de estas herramientas es su soporte para la persistencia de objetos "planos". El programador puede representar el modelo de dominio utilizando los mecanismos orientados a objetos de la plataforma utilizada, sin necesidad de utilizar librerías, clases o interfaces adicionales (con las determinadas restricciones ya expuestas). Estas herramientas satisfacen, por lo tanto, el requisito § 2.1.2.

Ninguno de los sistemas estudiados ofrece la posibilidad de ejecutar de un modo automático y transparente las rutinas de persistencia descritas en requisito § 2.1.4. Todas las herramientas de mapeo objeto/relacional y los SGBDOO estudiados requieren código específico para la demarcación de transacciones y para el borrado de objetos persistentes (violación del requisito § 2.2.1). Los sistemas más avanzados como Hibernate, JDO y FastObjects .NET implementan una estrategia de persistencia por alcance, donde se requiere programar explícitamente el almacenamiento y la obtención del objeto persistente raíz. A partir de la obtención del conjunto de objetos persistentes raíz, la consulta y modificación de los objetos en el almacén persistente se realiza de forma transparente, con la única limitación de tener que especificar cuando comienzan y finalizan las transacciones. En el caso del mecanismo de serialización de objetos de la plataforma Java y en el de la serialización de Java Beans con XML se utiliza también un mecanismo de persistencia por alcance pero la serialización y deserialización de objetos debe hacerse siempre explícitamente.

3.2.8.2 Adaptabilidad

Analizando los sistemas estudiados desde la perspectiva de su adaptabilidad se observa que están muy lejos de dar respuesta a los requisitos planteados en § 2.2. Ninguno de los sistemas estudiados ofrece la posibilidad de configurar los parámetros del mismo en tiempo de ejecución sin modificar su código (requisito § 2.2.1).

Además, tampoco ofrecen soporte para una adaptación programática dinámica (requisito § 2.2.2) más allá de unas determinadas circunstancias prefijadas. Tanto Hibernate como JDO permiten proporcionar los datos de configuración y los metadatos de mapeo al sistema de persistencia en tiempo de ejecución. Para proporcionar los metadatos ambos sistemas permiten el uso de XML. Utilizando cualquiera de las APIs para el tratamiento de XML [W3C98], en Java puede generarse un árbol XML en tiempo de ejecución con los datos de mapeo necesarios y configurar de este modo cómo realizará la traducción el gestor de persistencia. Con respecto a los datos de configuración, también pueden ser suministrados dinámicamente con XML en el caso de Hibernate o con un objeto `Properties` en el caso de JDO. La principal limitación es que tanto los datos de configuración como los metadatos de mapeo deben suministrarse en el momento de instanciar el gestor de persistencia correspondiente antes de comenzar a utilizarlo dentro de la aplicación, con lo que se restringe la posibilidad de realizar una adaptación realmente dinámica (requisito § 2.2.2). Hibernate expone el metamodelo de los metadatos de configuración a través de una API orientada a objetos que permite su consulta y modificación, pero ofrece las mismas limitaciones que la generación de XML de cara a la adaptabilidad del sistema de persistencia

Ninguno de los sistemas estudiados permite la adaptación dinámica de los mecanismos de indexación (requisito § 2.2.3). Únicamente los SGBDOO ofrecen un mecanismo de indexación pero ninguno de los tres sistemas estudiados permite configurar el tipo de índices a utilizar ni en tiempo de diseño ni en tiempo de ejecución.

El único sistema que permite configurar diferentes formatos y paradigmas de almacenamiento es JDO, aunque no permite su configuración de forma dinámica (requisito § 2.2.4). La especificación de JDO [Sun2003] deja a los desarrolladores de implementaciones la posibilidad de decidir cual será el sistema que soporte en última instancia la persistencia. Además, la especificación JDO y su implementación de referencia (DORI) garantizan la compatibilidad binaria del código realzado, con lo que puede cambiarse entre implementación de JDO sin necesidad de recompilar. El resto de sistemas están restringidos a un formato de persistencia determinado: el mecanismo de serialización estándar de Java utiliza un formato binario propio; el mecanismo de serialización de Java Beans utiliza XML; Hibernate puede configurase únicamente con SGBD relacionales que soporten el estándar JDBC, aunque para favorecer la portabilidad soporta la configuración de diferentes "dialectos" de SQL; finalmente, los SGBD orientados a objetos estudiados contienen un sistema de almacenamiento propio en forma de base de datos orientada a objetos. Ninguno de estos sistemas verifica por lo tanto el requisito § 2.2.4.

Dentro de los sistemas estudiados, ninguno soporta la configuración dinámica de las políticas de actualización de objetos (requisito 2.2.5). Únicamente Hibernate permite establecer diferentes algoritmos de *fetching* predefinidos para configurar cómo se traen los objetos desde el almacén persistente a memoria cuando se navega a través de un árbol de objetos. La actualización de objetos se realiza automáticamente en Hibernate, JDO y FastObjects .NET, pero debe demarcarse programáticamente la transacción y no se tiene ningún control sobre los algoritmos de actualización utilizados. La demarcación declarativa de transacciones se permite en JDO e Hibernate haciendo uso de la *Java Transaction API* (JTA) cuando se integran en un entorno gestionado por contenedor (§ 3.6.1) o cuando se hace uso del sistema de transacciones de Spring (§ 3.6.2).

Con respecto al mecanismo de selección de los objetos persistentes, la mayor parte de los sistemas estudiados implementan un algoritmo de persistencia por alcance, donde los objetos raíz son hechos explícitamente persistentes y los objetos referenciados desde estos objetos raíz directa o indirectamente son hechos a su vez persistentes. Este mecanismo de persistencia por alcance está en todos los casos fijado de antemano por los propios sistemas, por lo que ninguno verifica el requisito § 2.2.6. En el caso de Hibernate, se ofrece, además del mecanismo de persistencia por alcance, el mecanismo de persistencia por cascada. Esta opción permite un mayor control sobre cómo se propaga la persistencia en cada asociación de los objetos persistentes. Se acerca por lo tanto a las necesidades expuestas en el requisito § 2.2.6, pero no las satisface plenamente por tratarse de un mecanismo predefinido.

JDBC permite trabajar contra cualquier SGBD relacional que proporcione un *driver*, gracias a la arquitectura descrita en 3.2.2. Todos los sistemas de persistencia estudiados basados en la plataforma Java y que permite en uso de SGBD relaciones utilizan JDBC, incluida la alternativa SQLJ. El problema de portabilidad entre diferentes gestores relacionales viene dado por los diferentes dialectos SQL existentes, debido al soporte desigual a los estándares SQL y a la introducción de variaciones propietarias por parte de los vendedores. Por ello, la portabilidad estará ligada realmente a utilizar código SQL estándar siendo el estándar más ampliamente soportado SQL-92 [Melton93].

3.2.8.3 Portabilidad

Analizando los aspectos de portabilidad de las soluciones estudiadas, únicamente FastObjects .NET soporta su utilización desde diferentes lenguajes de programación (requisito § 2.3.1). Esta portabilidad de lenguajes se deriva del soporte a múltiples lenguajes de la plataforma .NET [Thai2002] y al trabajo de FastObjects .NET al nivel del código MSIL (*Microsoft Intermediate Language*) de los ensamblados .NET.

En el caso del SGBDOO Poet, ofrece soporte a Java y C++ en forma de *bindings* específicos, permitiendo una cierta interoperabilidad entre ellos. Esta interoperabilidad se ve limitada debido en gran parte a la no adopción de un modelo de objetos común. Por ejemplo, C++ soporta herencia múltiple [Stroustrup98] y Java no, con lo cual de cara a conseguir una plena interoperabilidad esta característica de C++ no debería emplearse. En el caso del sistema de base de datos orientado a objetos de FastObjects .NET este problema se soluciona definiendo modelo de datos muy similar al modelo de datos de que ofrece .NET a través de su Sistema Común de Tipos (*Common Type System*) [Thai2002].

En el diseño de la plataforma Java y de la plataforma .NET se ha buscado la independencia del sistema operativo y del hardware que soportase su ejecución, por lo que los sistemas de persistencia basados en estas plataformas alcanzan los requisitos de independencia planteaos en § 2.3.2 y § 2.3.3.

Finalmente, aunque no es uno de los requisitos planteados en el Capítulo 2, debe señalarse como un aspecto positivo común de los sistemas estudiados sus posibilidades de optimización con el objetivo de incrementar la eficiencia. En el caso de JDBC y SQLJ esta eficiencia puede conseguirse mediante optimizaciones de las sentencias SQL basadas en las características del modelo relacional. Pero es en sistemas como Hibernate o las diferentes implementaciones de JDO donde se ofrecen características de optimización más avanzadas y transparentes de cara al usuario, que sólo debe preocuparse de su configuración.

3.3 Sistemas de Persistencia Adaptativos

Un sistema de persistencia adaptativo es un sistema de persistencia capaz de cambiar su comportamiento en función de los cambios que se produzcan dentro de él mismo o en su entorno. Dentro de este ámbito de los sistemas de persistencia, los esfuerzos de investigación se han centrado en la adaptación dinámica de las políticas de *clustering* en función del empleo dinámico de los objetos. Por ello, antes de revisar algunos de estos sistemas se explicará el concepto de *clustering* y por qué es interesante su adaptación dinámica.

Clustering [Manolis92] es básicamente una técnica utilizada para agrupar objetos juntos con el fin de minimizar el número de accesos a disco. Normalmente, un cluster se carga debido a una referencia a uno de los objetos que contiene. La estrategia de *clustering* empleada tendría éxito si el cluster cargado contiene todos los objetos que se van a utilizar en el futuro. Por ejemplo, todos los objetos que van a ser referenciados por el objeto que causó el fallo de objeto[5].

Se han investigado diversos algoritmos de *clustering* con el fin de mejorar el rendimiento de los sistemas de bases de datos orientados a objetos. En [Manolis92] se evalúa el rendimiento de los algoritmos más conocidos utilizando diferentes cargas de trabajo y el *benchmark* Tektronix. Los algoritmos comparados fueron BFS, DFS y WDFS [Stamos84], Árboles de colocación (*Placement Trees*) [Benzaken90], Cactis [Drew90], PRP [Yue73] y *clustering* estocástico (*stochastic clustering*) [Tsangaris91]. Los resultaron arrojaron que el *clustering* estocástico es el algoritmo que mejor rendimiento ofrece para todas las métricas utilizadas y también el más costoso computacionalmente. Pero lo realmente destacable del estudio es que, para cada prueba realizada, era posible encontrar un algoritmo más barato computacionalmente y que ofreciese un rendimiento similar, siendo este algoritmo diferente dependiendo de la carga de trabajo utilizada en cada caso. Se demuestra además que, incluso cuando la carga de trabajo y el grafo de objetos están fijados, la elección del algoritmo de *clustering* depende de los objetivos específicos del sistema.

Lo anterior justifica el interés en los algoritmos de *clustering* adaptables dinámicamente según diferentes parámetros del sistema de persistencia en ejecución. De hecho, un criterio utilizado para clasificar los algoritmos de *clustering* es la distinción entre estrategias estáticas y dinámicas [Darmon2000]. A continuación se revisan algunos sistemas de persistencia orientados a objetos que implementan una estrategia de *clustering* adaptable dinámicamente.

3.3.1 IK

El sistema de programación persistente IK [Sousa94] implementa un algoritmo de *clustering* adaptable. Los clusters tienen un objeto cabeza (*head*), a partir del cual el resto pueden ser accedidos. El sistema periódicamente verifica si los otros objetos pueden o no ser clasificados como objetos cabeza, creando clusters para ellos y los objetos relacionados.

[5] Un fallo de objeto se produce cuando se solicita un objeto y éste no está en el cluster. Se denomina fallo de objeto por analogía con los fallos de página en la gestión de memoria virtual de un sistema operativo.

3.3.2 Orion

El SGBDOO Orion [Kim91] utiliza segmentos con el fin de agrupar objetos en el almacén persistente. Los segmentos se componen simplemente de una clase y todos sus objetos y son gestionados de forma automática por Orion.

A veces es útil agrupar en el mismo segmento objetos de varias clases, debido a la existencia de reglas de asociación entre ellos. Para esos casos, Orion proporciona un mecanismo para que el programador indique estas situaciones especiales de forma manual.

La estrategia de *clustering* de Orion constituye por lo tanto una aproximación mixta, basada en una estrategia de agrupamiento fijado y un mecanismo manual para casos más complejos.

3.3.3 O_2

En el caso del SGBDOO O_2 [Deux91] se utiliza la técnica de Árbol de Colocación (*Placement Tree*) [Benzaken95]. Un árbol de colocación es una representación de relaciones entre objetos y se utiliza con el objeto de definir clústeres.

Los árboles de colocación pueden variar dinámicamente encargándose el sistema de cambiar los clústeres asignados automáticamente. Sin embargo, la ubicación de los índices dentro de un clúster u otro se puede hacer de un modo explícito, pero no dinámico.

3.3.4 Aportaciones y Carencias de los Sistemas Estudiados

El principal inconveniente de los sistemas estudiados es que el único parámetro que permiten flexibilizar es el algoritmo de *clustering*, por lo que no se verifica el principio fundamental que subyace a los requisitos recogidos en § 2.2: la posibilidad de modificar cualquier parámetro del sistema de persistencia.

Incluso centrando el análisis en la estrategia de *clustering*, no se ofrece soporte para la adaptabilidad dinámica (§ 2.2.1) y programática (§ 2.2.2) de la estrategia utilizada.

En el caso de Orion, no se ofrece realmente una adaptación dinámica de la estrategia de *clustering* puesto que el agrupamiento de objetos debe ser indicado expresamente por el programador.

El sistema IK sí ofrece un algoritmo de *clustering* adaptable pero éste está prefijado y el sistema no ofrece la posibilidad de configurar otras estrategias, es decir, se trata de un parámetro del sistema que no se puede modificar.

Finalmente, el sistema O_2 permite, a través del artefacto de los árboles de colocación, la adaptación dinámica de los clusters creados. El problema es que los árboles de colocación son creados por los administradores del sistema basándose en su propia experiencia [Manolis92], es decir, su gestión por parte del sistema no es automática y por lo tanto tampoco lo es la de los clusters que se basan en ellos. No se puede hablar en este caso tampoco de una adaptabilidad dinámica y transparente del parámetro de *clustering*.

3.4 Sistemas de Persistencia basados en Contenedores

El modelo basado en contenedores [Schofield2002c], se basa en establecer una división visible del almacenamiento persistente de objetos en contenedores para obtener las siguientes ventajas [Schofield2003]:

- Eliminar la diferenciación entre memoria principal y secundaria, siendo un todo continuo para el programador.
- Eliminar una parte de la aplicación que es tediosa de programar, repetitiva y que supone un tamaño considerable de líneas de código, minimizando los errores derivados de la programación de las partes de la aplicación anteriores.
- Ofrecer una mejora en el rendimiento, en comparación con otros sistemas de persistencia más transparentes (que veremos en puntos posteriores) [Schofield2002b], gracias a la utilización de compiladores nativos en lugar de mecanismos de interpretación.

Los contenedores son gestionados como directorios de objetos, siendo éstos totalmente independientes en cuanto a solapamientos y permisos se refiere. Así, se trata de aunar las mejores características de la persistencia con las mejores características de los sistemas de archivos. El lenguaje soportado por un sistema que implementa este modelo debe proporcionar, de un modo u otro, un modo de de acceso a estos directorios.

La principal característica que diferencia a los sistemas de persistencia basados en contenedores es la técnica de *clustering* utilizada en el almacenamiento persistente. Este tipo de sistemas utiliza *clustering* basado en grupos de objetos relacionados entre sí. Un objeto estará ubicado en el mismo *cluster* (contenedor) que sus subobjetos y sus clases. De esta forma, se espera que al cargar un objeto se pueda precargar la mayoría de los objetos que van a necesitarse a continuación. En el modelo de contenedores el *clustering* lo realiza el usuario al crear los directorios, y es por tanto totalmente manual, lo que es muy eficiente y poco transparente, y a la vez asegura que todos los objetos necesarios van a ser cargados con el *cluster* (contenedor). Ofrecen menor transparencia que los mecanismos de *clustering* adaptativo, a un mejor rendimiento.

3.4.1 Barbados

Barbados es un entorno de programación persistente, en el que el código fuente, el código ejecutable, los compiladores y los datos coexisten en el almacén de persistencia [Cooper96].

Un sistema Barbados consiste en un directorio jerárquico donde se almacenan, en lugar de datos, elementos de C++ tales como objetos, clases, enteros o registros. Las estructuras que elija el programador podrán utilizar cualquier tipo y serán almacenadas en disco automáticamente, cuando el usuario abandone la sesión, independientemente de su tipo o estructura. Los identificadores siguen las pautas descritas en este lenguaje de programación.

El entorno de programación consiste en un intérprete de comandos donde el usuario introduce las sentencias C++ y sus declaraciones. Para codificar un programa, el usuario sólo tiene que escribir su código C++ en el intérprete de coman-

dos; para ejecutarlo, sólo tiene que invocarse a su función principal. Todo el código escrito y los objetos creados sobreviven a la ejecución del programa.

Barbados ofrece un sistema de compilación incremental e interactiva donde el lenguaje C++ es el lenguaje utilizado en el intérprete de comandos. Aunque Barbados fue construido sobre el API de Windows 95/NT (Win32), éste podría ejecutarse como un sistema operativo propiamente dicho –de hecho, el objetivo de un sistema de persistencia es aunar los lenguajes de programación y sistemas operativos, utilizando objetos persistentes en lugar de ficheros.

Para utilizar el concepto de contenedor, el lenguaje C++ fue ampliado con un conjunto de funciones necesarias para llevar a cabo la gestión de un sistema de persistencia basado en contenedores:

- Función `cd()`: Función que ofrece el cambio de contenedor. La estructura jerárquica de contenedores está anidada mediante el carácter `/`, siendo `/` el contenedor raíz.
- Función `dir()`: Muestra todos los elementos existentes en un contenedor.
- Función `mkdir()`: Crea un nuevo contenedor.
- Función `del()`: Elimina objetos del sistema de persistencia.
- Función `rmdir()`: Elimina contenedores del sistema de persistencia.

A modo de ejemplo, lo siguiente puede ser una sesión de programación interactiva en Barbados:

```
cd(/);
mkdir(test);
Barbados> test: container
cd(test);
mkdir(program);
Barbados> program: container
cd(program);

// Container /test/program
class Counter {
       float count;
  public:
       float getCount(void) { return ++count; }
       void reset(void) { count = 0; }
};
barbados> class Counter {};

cd(/);
mkdir(data);
barbados> data: container
/test/program/Counter c;
barbados> c: Counter

c.reset();
c.getCount();
barbados> 1
```

Figura 3.26. Ejemplo de creación de un programa en Barbados.

El programador se sitúa, con el comando `cd`, en el contenedor raíz. El intérprete de comandos muestra el resultado de cada opción con un mensaje en consola –en gris, en la Figura 3.26. Una vez allí, creará el contenedor `test` y `program`,

jerárquicamente anidados, mediante `mkdir`. Es allí, en el contendor `/test/program` donde crea una nueva clase `Counter`. Nótese cómo el sistema, de un modo interactivo nos indica el resultado de compilar y ejecutar los comandos que le indiquemos. Una vez creada la clase, se crea una instancia de ésta en el contenedor `/data` –es posible acceder a elementos distintos ubicados en otro contenedor. Pasándole el mensaje `reset` y `getCount`, dejaremos su estado a 1, pudiendo acceder a este objeto, con su nuevo estado, la próxima vez que ejecutemos el sistema.

La principal ventaja de Barbados está basada en la política explícita de *clustering* que presenta. Delega en el humano la tarea de establecer las relaciones entre los elementos de los programas, de forma que, gracias a la cohesión identificada por el programador, ofrece un mayor rendimiento cargando los contenedores en memoria conforme vayan siendo requeridos. Una extrapolación de mediciones en programas persistentes obtuvo un beneficio del 28,33% respecto al almacenamiento explícito de datos [Schofield2002b]. Sin embargo, esta ventaja es contraria a la transparencia buscada a lo largo de este estudio.

3.4.2 Zero

Zero es realmente una máquina virtual pequeña y simple, sin soporte de tipos a bajo nivel, orientada a objetos pura, y basada en prototipos [Schofield2004]. El sistema Zero, se compone de:

- Ensamblador: Permite obtener las operaciones básicas directamente sobre la máquina virtual, desde crear objetos y mandar mensajes, hasta control de excepciones.
- Macroensamblador para programar la máquina virtual Zero, basándose en el ensamblador de muy bajo nivel. Si bien no llega a la comodidad de un lenguaje de programación, es adecuado para proyectos de tamaño medio.
- Máquina virtual: Ejecuta los objetos creados por el ensamblador, el macroensamblador, o los lenguajes Prowl o J--. La máquina virtual es multiplataforma, existiendo versiones para Linux y Windows.
- Librería estándar interna de Zero.

Las características básicas de esta máquina virtual son: herencia simple, dinámica (implementada mediante delegación), creación y clonación de objetos (y prototipos, indistinguibles de los primeros), paso de mensajes, manejo de excepciones y persistencia basada en contenedores.

La máquina virtual está basada en registros (que guardan referencias a objetos), estructurándose en dos grandes grupos: el acumulador (`__acc`), que guarda la referencia resultado de la instrucción anterior, el registro que guarda el objeto que está ejecutando el método (`__this`), y el registro que guarda la excepción que se haya producido (`__exc`); y en un segundo grupo los registros generales que pueden ser utilizado para cualquier propósito (`__gp`*n*).

Para facilitar el desarrollo de aplicaciones se han desarrollado dos compiladores de lenguajes de alto nivel: uno, J--, es un subconjunto de Java, mientras el otro es similar a Self [Chambers89].

Lo más interesante de esta plataforma es el modelo de objetos que utiliza, basado en prototipos y dotado de reflectividad estructural en tiempo de ejecución, y cómo aprovecha éste para implementar un mecanismo de persistencia basado en contenedores con evolución del esquema.

En el modelo computacional basado en prototipos que utiliza Zero, no existe el concepto de clase; la única abstracción existente es el objeto [Borning86]. Un objeto describe su estructura (conjunto de atributos), su estado (los valores de éstos) y su comportamiento (la implementación de los métodos que pueda interpretar).

La herencia es un mecanismo jerárquico de delegación de mensajes existente también en el modelo de prototipos. Si se le envía un mensaje a un objeto, se analiza si éste posee un método que lo implemente y, si así fuere, lo ejecuta; en caso contrario se repite este proceso para sus objetos padre, en el cado de que los hubiere.

La relación de herencia entre objetos basados en prototipos es una asociación más, dotada de una semántica adicional –la especificada en el párrafo anterior. En el caso de Zero, sigue el criterio del lenguaje de programación Self [Ungar87], la identificación de esta semántica especial es denotada por la definición del miembro `parent`. El objeto asociado mediante este miembro es realmente el objeto padre. Al tratarse la herencia como una asociación, es posible modificar en tiempo de ejecución el objeto padre al que se hace referencia, obteniendo así un mecanismo de herencia dinámica.

Vemos cómo, al eliminar el concepto de clase en el modelo, la aproximación de prototipos resulta más sencilla. Sin embargo, ¿es posible agrupar objetos con el mismo comportamiento, al igual que lo hacen las clases?

Utilizando únicamente el concepto de objeto también podremos agrupar comportamientos. Si se crean objetos que únicamente posean métodos, éstos describirán el comportamiento común de todos sus objetos derivados. Este tipo de objetos se denomina de característica o rasgo (*trait*) [Lieberman86]. En la Figura 3.27, el objeto *trait* `Object` define el comportamiento `toString` de todos los objetos. Del mismo modo, `Point` define el comportamiento de sus dos objetos derivados.

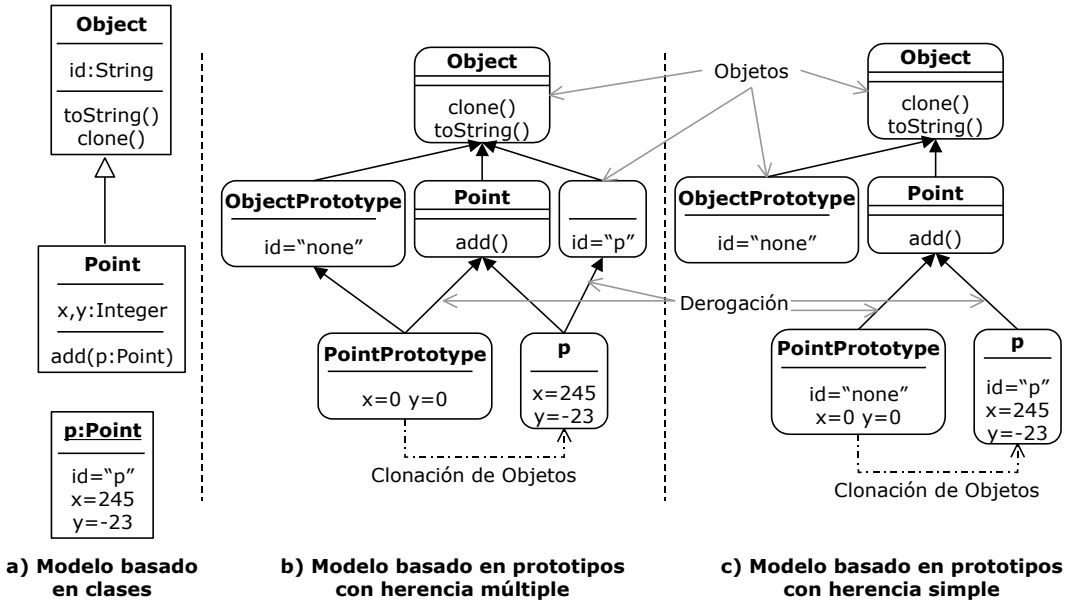


Figura 3.27: Representación de clases y objetos en los dos modelos.

Del mismo modo que hemos agrupado objetos de igual comportamiento, podemos cuestionarnos la posibilidad de agruparlos por estructura. Un prototipo es un objeto descriptor de una estructura común, utilizado para hacer copias exactas de él –clonaciones. En este modelo computacional, mediante la utilización de prototipos y la primitiva de clonación, se obtiene la misma funcionalidad que la de instanciación o creación de objetos a través de una clase en su modelo homólogo.

En la Figura 3.27, la creación de un punto pasa por la clonación de su prototipo. Éste posee la estructura común de todos los puntos (atributos `x` e `y`), su estado inicial (ambos valores iguales a cero) y el comportamiento común definido por el objeto *trait* `Object`. Vemos en la figura también, cómo es posible representar el diseño mediante la utilización de un sistema computacional dotado tan solo de herencia simple.

El mecanismo empleado para clonar los objetos es la reflectividad estructural. Además de las ventajas que aporta la reflectividad a un modelo computacional (tal y como veremos a lo largo de esta Tesis), ésta permite implementar un mecanismo de evolución del esquema no soportado por el modelo de clases de un modo coherente [Schofield2002].

El sistema de contenedores que utiliza está basado en la composición jerárquica de objetos. El objeto raíz del sistema de persistencia es denominado `psRoot`. El objeto *trait* que es utilizado para crear nuevos contenedores (clones de un hijo prototipo) es el contenedor `psRoot` es, de hecho, un objeto derivado de este *trait*. Mediante la adición de atributos a los objetos contenedores, mediante reflectividad estructural, se hacen persistentes los atributos añadidos –persistencia por alcance. Si a un contenedor se le añade otro contenedor, se establecerá una composición jerárquica de contenedores. Veamos un ejemplo:

```
object DisneyPersistente
  method + doIt()
  {
    reference disney = Container.createChild( "Disney" );
    reference donald = Persona.createChild( "donald" );
    reference daisy  = Persona.createChild( "daisy" );

    donald.ponNombre( "Donald" );
    daisy.ponNombre( "Daisy" );

    disney.add( donald );
    disney.add( daisy );
    psRoot.add( disney ;
    return;
  }
endObject
// * Los objetos donald y daisy ya son persistentes
```

Figura 3.28: Haciendo objetos persitentes en Zero.

Esto hará que el contenedor `disney`, que cuelga del contenedor raíz del almacenamiento persistente (`psRoot`) se guarde con los objetos que contenga. Una vez éste proceso termina, y el contenedor está por tanto guardado en el almacenamiento persistente, siendo posible recuperarlo mediante una referencia simple en cualquier otro proceso (`psRoot.Disney.donald`). Esto se aprecia en el siguiente código:

```
object MostrarDisney
  method + doIt()
  {
```

```
    System.console.write( psRoot.Disney.donald.toString() );
    System.console.lf();
    return;
  }
endObject
```

Figura 3.29: Acceso a un objeto persistente.

3.4.3 Aportaciones y Carencias de los Sistemas Estudiados

Una de las ventajas de los sistemas de persistencia basados en contenedores es el modo en el que une la programación y los sistemas de almacenamiento, comúnmente separados en el desarrollo de aplicaciones. Se gana considerablemente en transparencia respecto al desarrollo habitual de aplicaciones (§ 2.1), haciendo que el programador, con mecanismos sintácticos (Barbados) o semánticos (Zero) adicionales, pueda identificar elementos persistentes de un modo similar a la programación no persistente.

No obstante, aunque es un paso hacia delante en la obtención de una mayor transparencia, no se llega a ofertar los requisitos buscados en esta Tesis. No separa totalmente el código persistente del funcional (§ 2.1.1), aunque sí permite utilizar el mismo mecanismo de persistencia independientemente de la funcionalidad (§ 2.1.2) y estructura (§ 2.1.3) de los programas. La consecución de estos dos requisitos es debida a la utilización de reflectividad estructural [Foote90]. No permite conocer de un modo implícito el momento en el que se deben recuperar ni eliminar (§2.1.4) los objetos del sistema de persistencia –han de ser explicitados por el programador. Adicionalmente, también se requiere el desarrollo de aplicaciones con entornos propietarios (§ 2.1.5).

La adaptabilidad dinámica es uno de los inconvenientes de este tipo de sistemas. No permite modificar los distintos componentes del sistema de persistencia en tiempo de ejecución (§ 2.2.1, § 2.2.2, § 2.2.3, § 2.2.4 y § 2.2.5), a excepción de la selección de objetos persistentes (§ 2.2.6) que, si bien es factible, debe hacerse de un modo explícito por parte del programador.

En lo referente a la portabilidad (§ 2.3), el modelo basado en componentes no supone una restricción a priori. De hecho, Barbados es dependiente del lenguaje C++ (§ 2.3.1), del sistemas operativos win32 (§ 2.3.2) y del hardware Intel (§ 2.3.3). Sin embargo, Zero es independiente del sistema operativo (§ 2.3.2), ha sido diseñada con el modelo de objetos basado en prototipos que da un soporte computacional directo a cualquier lenguaje orientado a objetos (§ 2.3.1) y, aunque actualmente sólo está implementado para procesadores Intel de 32 bits, no presente restricciones en su diseño concernientes al hardware utilizado (§ 2.3.3).

La principal ventaja de los sistemas de persistencia basados en contenedores es que sacrifican transparencia por parte del programador, ofreciendo un rendimiento muy interesante. Estudios empíricos han indicado cómo, extrapolando unas mediciones en base a la optimización que un compilador pueda realizar, el rendimiento de aplicaciones persistentes podría ser superior a la gestión explícita de las primitivas de persistencia.

Otros beneficios obtenidos, únicamente en el caso del sistema Zero, son la evolución del esquema y un nivel de transparencia, gracias a la utilización de reflectividad estructural junto con un modelo de objetos basados en prototipos. Estas

dos ideas serán fundamentales para llevar a cabo el desarrollo de un nuevo sistema innovador, presentado a lo largo de esta Tesis Doctoral.

3.5 Sistemas de Persistencia Ortogonal

La persistencia ortogonal fue un concepto introducido por M. P Atkinson y R. Morrison [Atkinson95]. Un sistema de persistencia ortogonal es un sistema de persistencia que soporta un tratamiento uniforme de los objetos con independencia de sus tipos, permitiendo dotar a los valores de todos los tipos de la longevidad que se desee, cualquiera que sea ésta.

El problema que intentan resolver los sistemas de persistencia ortogonal es la desadaptación de impedancias que se produce al utilizar dos herramientas básicas en la construcción tradicional de aplicaciones persistentes: las bases de datos y los lenguajes de programación. Estas inconsistencias se derivan de las diferentes filosofías y circunstancias que rodean cada campo [Atkinson95].

La comunidad de bases de datos se centra en resolver complejos problemas de ingeniería, como el preservar enormes cantidades datos de una manera fiable o soportar muchos procesos trabajando contra estos datos de una forma eficiente. Por el contrario, no se consideran prioritarios aspectos tales como los hábitos de diseño de un analista de sistemas o permitir que equipos aislados de programadores puedan desarrollar una aplicación persistente de forma independiente. En definitiva, el campo de las bases de datos está dominado por las soluciones técnicas.

Por otra parte, los diseñadores de lenguajes de programación se han centrado en permitir a los programadores ser precisos y hacer los programas entendibles. Se han buscado principios de diseño [Strachey67] que se tradujesen en lenguajes con reglas regulares. Estas reglas debían poder combinarse bien con las intuiciones de los programadores, ser fácilmente definibles, y favorecer que los programas se ejecutasen con una eficiencia razonable.

Además, el desarrollo independiente de ambos campos y las inconsistencias derivadas de éste tienden a perpetuarse y crecer. En este contexto, la solución que proponen los sistemas de persistencia ortogonal pasa por desarrollar un entorno integral que soporte el desarrollo de aplicaciones persistentes, en vez de intentar unir dos mundos separados por una brecha que no sólo se perpetúa, sino que tiende a crecer.

Para entender con mayor detalle la motivación de los sistemas de persistencia ortogonal conviene repasar precisamente el concepto de persistencia. Según la definición dada en § 3.1.1, con el término "persistencia" se hace referencia a mantener los valores de los datos durante todo su tiempo de vida, no importa cómo de breve o largo pueda ser éste. El tiempo de vida de los valores de los datos es el período que va desde su creación hasta que ya no son usados más por la aplicación persistente. La Tabla 3 muestra este rango [Atkinson95].

Rango	Tiempo de vida de los valores de los datos
1	Resultados temporales en la evaluación de una expresión
2	Variables locales
3	Variables globales y elementos en el montón (*heap*)

Rango	Tiempo de vida de los valores de los datos
4	Datos que duran una ejecución completa de un programa
5	Datos que duran varias ejecuciones de diversos programas
6	Datos que duran mientras un programa esté siendo usado
7	Datos que sobreviven a las diferentes versiones de un programa
8	Datos que sobreviven a las diferentes versiones de los sistemas que dan soporte a la persistencia.

Tabla 3. Los diferentes tiempos de vida de los valores de los datos.

Habitualmente se utilizan los lenguajes de programación para abordar los ciclos de vida recogidos en las filas 1 a 4 mientras que se utilizan bases de datos y ficheros para las filas 5 a 8. Los sistemas de persistencia ortogonal se diseñan de manera que el tratamiento de los valores de los datos sea uniforme e independiente de su longevidad, tamaño y tipo. Además, se aspira a conseguir esta uniformidad para todos los servicios del sistema de persistencia, desde la definición de los datos y operaciones, hasta la integridad, concurrencia y distribución. Es decir, el principal objetivo de estos sistemas es proporcionar un modelo computacional sencillo y uniforme para todos los aspectos de una aplicación que traten con datos persistentes. Esta capacidad se define con tres principios [Atkinson95].

- **Independencia de la persistencia**. La forma de un programa es independiente de la longevidad de los datos que manipula. Los programas son iguales independientemente de si manipulan datos persistentes a corto o largo plazo.

- **Ortogonalidad respecto al tipo**. Todos los objetos podrán ser persistentes con cualquier plazo de longevidad independientemente de su tipo. No hay casos especiales donde no se permita a un objeto tener una persistencia a largo plazo ni donde no se permita que sea temporal.

- **Identificación de la persistencia**. La elección de cómo se identifican los objetos persistentes es ortogonal al discurso del sistema. En concreto, el mecanismo para identificar los objetos persistentes no está relacionado con el sistema de tipos.

La independencia de la persistencia libera al programador de la responsabilidad que tener que codificar explícitamente el movimiento de datos a lo largo de la jerarquía de dispositivos de almacenamiento, así como de tener que programar la traducción entre las representaciones persistentes a corto y largo plazo.

La ortogonalidad respecto al tipo favorece el modelado del dominio de objetos. Estos modelos podrán ser completos e independientes de la persistencia que tengan los objetos modelados. Dentro de este principio está contenido el principio de diseño de lenguajes de programación denominado principio de la completud de tipos [Strachey67]: debe permitirse cualquier combinación o construcción de datos en todos los tipos. El programador no debe enfrentarse a un sistema donde unos tipos de datos pueden ser persistentes y otros no.

En cuanto a la identificación de la persistencia, existen varios métodos para la identificación de los objetos que deben ser persistentes. Algunos de ellos implican marcar un objeto como persistente utilizando un identificador concreto, modi-

ficando su tipo en la declaración o haciendo que extiendan una determinada clase o implementen un determinado interfaz. Estos métodos no están permitidos bajo el punto de vista del principio de la identificación de la persistencia, además de no ser apropiados por otras razones [Atkinson86].

El método más utilizado para la identificación de la persistencia es el denominado identificación por alcance (*identification by reachability*, *persistence by reachability* o *transitive persistence*). Con este método, la identificación de los objetos persistentes se realiza automáticamente por el sistema, computando el cierre transitivo de todos los objetos alcanzables desde un conjunto de objetos raíz marcados como persistentes.

Existen diversos investigadores sobre persistencia que opinan que, aunque los tres pilares sobre los que se sustenta la persistencia otogonal ofrecen una transparencia indudable al programador, pueden llegar suponer desentajas como no tener una forma de organizar los objetos persistentes, limitar la gama de lenguajes orientados a objetos soportados o un menor rendimiento [Roselló2001]. No obstante, estos principios se fundamentan en conseguir la máxima productividad para los desarrolladores de aplicaciones. Siempre que cualquiera de ellos no se consiga en su totalidad, el desarrollador tendrá que pagar el coste de escribir un código más complejo [Atkinson96].

Existen distintos ejemplos de sistemas de persistencia ortogonal. En el caso de Java, existe una extensión de la especificación del lenguaje Java [Gosling96] denominada *Orthogonally Persistent Java* (OPJ) [Jordan2000] que proporciona persistencia ortogonal a la plataforma Java siguiendo los principios definidos en [Atkinson95].

Se han desarrollado una serie de prototipos que implementan la especificación OPJ donde los más conocidos son PJama [Atkinson96] y PVM [Lewis2000]. Estos prototipos están disponibles para fines de investigación o evaluación.

3.5.1 PJama

PJama es un sistema de programación persistente cuya evolución puede seguirse a través de [Atkinson96], [Jordan98] y [Atkinson2000]. Su desarrollado ha sido dirigido por Sun Microsystems como parte del proyecto de investigación de Sun denominado *"Forest"*.

Un sistema Pjama comprende dos componentes fundamentales [Jordan2004]:

- Una variante de una máquina virtual Java estándar modificada para soportar la persistencia ortogonal (*Persistent Java Virtual Machine*, PJVM).
- Un almacén persistente: una entidad que existe en un medio de almacenamiento estable y que contiene el estado persistente de la computación que se está ejecutando en la máquina virtual de Java.

La máquina virtual Java persistente (PJVM) es en sí misma una máquina sin estados (*stateless*). Toda la información que se necesita para retornar la ejecución en un momento dado se mantiene en el medio de almacenamiento estable. El estado del sistema se refresca periódicamente en el almacén bien sea mediante una llamada explícita desde la aplicación o cuando la PJVM finaliza normalmente. Este refresco es atómico, cuando se produce una excepción o un error del sistema se deja el al-

macén en el estado que contenía cuando se realizó el último punto de control (*checkpoint*).

En la implementación de PJama el medio de almacenamiento se representa con un fichero de disco, si bien este aspecto no es visible para la aplicación. De hecho, una consecuencia del modelo OPJ [Atkinson96] es que es la PJVM quien se encarga de abrir el medio de almacenamiento y no la aplicación Java.

PJama añade una clase especial a la API de Java: la clase `PersistentStore`. Para almacenar objetos en el sistema persistente debe declarase el objeto raíz del árbol de objetos a almacenar como un objeto raíz persistente. De este modo, el cierre transitivo de este objeto raíz es almacenado en el sistema persistente mediante el mecanismo de persistencia por alcance. En la Figura 3.30 se muestra una aplicación para ilustrar el trabajo con PJama. Las líneas de código mostradas en negrita son las sentencias adicionales necesarias para gestionar objetos persistentes. Mediante los métodos `getPRoot()` y `newRoot()` de la clase `PersistentStore` pueden recuperarse y almacenarse los objetos persistentes raíz, que por el mecanismo de persistencia por alcance permiten acceder y hacer persistir los objetos contenidos en su cierre transitivo.

```
import java.util.ArrayList;
import org.opj.store.*;

public class Autores {
  private ArrayList autores;

  public Autores() {
    // Obtiene el almacenamiento
    PJavaStore pjs=PjavaStore.getStore();
    // Obtiene los autores
    try { autores=(ArrayList)pjs.getPRoot("autores");
    } catch (PJSException) {
      autores=new ArrayList();
      // Hace los autores persistentes
      pjs.newPRoot("autores",autores);
    }
  }

  public void añade(Autor autor) { autores.add(autor); }
  public Autor toma(int índice) { return (Autor)autores.get(índice); }
  public int numAutores() { return autores.size(); }
  public String toString() { return autores.toString(); }

  public static void main(String[] args) {
    Autores autores=new Autores();
    if (autores.numAutores()==0) {
      autores.añade(new Autor("Antonio", "Machado", null));
      autores.añade(new Autor("Miguel","Unamuno","Jugo"));
    }
    System.out.println("Autores: "+autores);
  }

}
```

Figura 3.31. Ejemplo de una aplicación persistente en PJama.

PJama crea la ilusión de que existe una computación continua para toda la PJVM, aunque en la práctica el estado persistente de la computación sigue una secuencia de estados discretos con un intervalo determinado por la frecuencia de control (*checkpoint frequency*) [Jordan2004]. El ámbito de la computación persistente se limita al estado gestionado por la PJVM, requiriéndose la intervención de la aplica-

ción para capturar el estado que hace referencia a los objetos que están fuera del ámbito de la PJVM, como por ejemplo las conexiones de *sockets*.

Es importante señalar que PJama y OPJ convierten en persistente todo lo que se necesita para retornar la computación como si ésta nunca hubiese sido suspendida, al menos desde un punto de vista conceptual. Algunos corolarios que se derivan de esta característica son:

- Una clase se carga y se inicializa independientemente del número de veces que la computación sea suspendida y retornada.
- El enlace entre una instancia y su objeto `Class` asociado nunca cambia.
- El estado de ejecución, representado con instancias de la clase `Thread`, es persistente.
- La gestión de memoria automática se extiende al dominio de la persistencia. El modelo OPJ necesita herramientas adicionales para la recolección de basura en almacenes persistentes.

3.5.2 JSpin

JSpin [Kaplan96, Ridgway2000] es un sistema de programación persistente basado en Java. Forma parte del framework SPIN, cuyo principal objetivo es soportar diversos lenguajes sobre una plataforma persistente [Kaplan2000], siendo Java uno de los lenguajes y JSpin el nombre de la parte Java.

El framework SPIN define una plataforma persistente compuesta por un sistema de gestión de bases de datos orientado a objetos (SGBDOO) y un interfaz, diseñado para ser utilizable por más de un lenguaje. Los tres lenguajes soportados en la actualidad son C++, Java y CLOS. La interacción con la base de datos se realiza utilizando el interfaz TI/DARPA Open OODB [Wells92].

Los objetivos de JSpin son similares a los de PJama; busca ofrecer un sistema de persistencia ortogonal para la plataforma Java. La principal diferencia es que en su implementación opta por no modificar la máquina virtual Java, sino el compilador.

JSpin se presenta en forma de una API que proporciona un sistema de persistencia orientado a objetos para las aplicaciones Java. Esta API incluye los métodos que deben añadirse a cada clase procesada por JSPIN, junto con diversas clases específicas del sistema.

El objetivo es ofrecer la ilusión de que todos los métodos se añaden a la clase Object, de la cual heredan todas las clases Java. En la realidad, no se hace así debido a restricciones con el tipo de retorno. En concreto, el método `fetch()` de una clase debe devolver un objeto de esa clase y por lo tanto debe ser específico para la clase.

La API básica añade los siguientes métodos a cada clase [Ridgway97]:

- `public void persist([String name])`. Cuando se invoca sobre cualquier objeto, el objeto y todos los objetos que son alcanzables desde él se vuelven persistentes. El parámetro `name` es opcional y puede utilizarse para asignar un nombre al objeto persistente a partir del cual puede recuperarse más tarde. Si no se le asigna ningún nombre, el objeto

únicamente podrá ser recuperado si es referenciado desde otro objeto persistente.

- `public static Object fetch(String name)`. Cuando se invoca, devuelve la instancia persistente de la clase que se corresponde con el nombre proporcionado con el parámetro `name`.

Además de estos métodos, existen métodos relacionados en la clase `PersistentStore` que proporcionan una funcionalidad similar (Figura 3.32).

```
public abstract class PersistentStore {
      public void beginTransaction();
      public void commitTransaction();
      public void abortTransaction();
      public void persist(Object obj, String name);
      public void persist(Object obj);
      public Object fetch(String name);
}
```

Figura 3.32. Métodos de la clase PersistentStore.

La adición de los métodos a cada clase la realiza un compilador Java modificado. La implementación de los métodos añadidos, tanto a las clases persistentes como a las clases específicas JSPIN, se realiza principalmente mediante llamadas al sistema de persistencia subyacente.

3.5.3 Aportaciones y Carencias de los Sistemas Estudiados

Los sistemas estudiados presentan un alto nivel de transparencia a la hora de proporcionar persistencia a las aplicaciones. Si se analizan bajo la perspectiva de los tres principios fundamentales de la persistencia ortogonal [Atkinson95], PJama y JSpin verifican plenamente el principio de ortogonalidad respecto al tipo, puesto que todos los tipos del sistema son persistentes (requisito § 2.1.3); también verifican el principio de identificación de la persistencia, puesto que se realiza mediante persistencia por alcance [Atkinson95] y es independiente del sistema de tipos.

El principal inconveniente de estos sistemas es que el principio de persistencia por alcance hace que no cumplan de un modo completo el criterio de independencia de la persistencia [Jordan96, Schofield2002]. Los objetos persistentes raíz deben ser almacenados y recuperados con código explícito. Por ello, la persistencia no es tenida en cuenta como una incumbencia totalmente separada de la lógica de la aplicación (violación del requisito § 2.1.1). No obstante, el programador no tiene que realizar ninguna tarea más allá de la declaración de los objetos raíz. Este hecho, junto con la verificación del principio de la ortogonalidad respecto al tipo, hace que el sistema de persistencia pueda integrarse con cualquier aplicación Java sin imponer restricciones a ésta (requisito § 2.1.2).

Los sistemas estudiados sí consiguen una automatización integral del sistema de persistencia (requisito § 2.1.4). En concreto, no es necesario delimitar programáticamente el ámbito de las transacciones, al contrario de lo que sucedía en los sistemas analizados en § 3.2. Sin embargo, no se ofrece la posibilidad de modificar o configurar los algoritmos utilizados para sincronizar los objetos en memoria y los persistentes (requisito § 2.2.5).

Por otra parte, el mecanismo de persistencia por alcance implementado en los sistemas estudiados impide flexibilizar el número de objetos que son hechos

persistentes (requisito § 2.2.6). Cuando un objeto es persistente, todos aquellos que estén asociados a él también lo serán.

En cuanto a la adaptabilidad de los sistemas estudiados, ninguno permiten parametrizar las distintas variables del sistema de persistencia, ni programáticamente (requisito § 2.2.2) ni a través de un administrador externo (requisito § 2.2.1).

PJama no ofrece posibilidad de modificar el sistema de almacenamiento utilizado (requisito § 2.2.4). JSpin ofrece una mayor flexibilidad en este punto gracias a la utilización del interfaz TI/DARPA Open OODB [Wells92], pero sigue viéndose limitado a sistemas de bases de datos orientados a objetos que ofrezcan dicho interfaz.

Los sistemas estudiados no ofrecen independencia del lenguaje (requisito § 2.3.1). Se trata de una consecuencia directa de la naturaleza de los sistemas de persistencia ortogonal propuesta en [Atkinson95]: entornos integrales para el desarrollo de aplicaciones persistentes. Debe señalarse que el framework SPIN [Kaplan2000] tiene el objetivo de permitir gestionar objetos que hayan sido total o parcialmente creados por cualquiera de los lenguajes soportados. Es posible crear un conjunto de objetos relacionados en un lenguaje y acceder a ellos en otro, aunque no es posible modificar esos objetos utilizando un lenguaje distinto del que fue usado para crearlos.

3.6 Frameworks de Persistencia

Un *framework* puede definirse como una colaboración de clases adaptables que definen una solución para un problema dado. Un *framework* define una colección de abstracciones fundamentales junto con sus interfaces y especifica las interacciones entre los objetos, con el fin de poder reutilizar la arquitectura y el diseño. Además, suele proporcionar una serie de implementaciones por defecto para favorecer la reutilización de código.

En este apartado se analizarán dos *frameworks* de persistencia que utilizan aproximaciones muy diferentes: Enterprise Java Beans (EJB) [Sun2003d] y Spring [Johnson2005].

3.6.1 Enterprise Java Beans

Enterprise Java Beans (EJB) [Sun2003d] define una arquitectura de componentes para el desarrollo de aplicaciones distribuidas. EJB forma parte de la especificación de la plataforma Java 2 Enterprise Edition (J2EE). Los desarrolladores que hagan uso de esta arquitectura, obtienen un soporte transparente para los aspectos de distribución, persistencia, transacciones y seguridad [Jordan2004].

Un *enterprise bean* es un componente software que se ejecuta en el lado del servidor. Necesitan un servidor de aplicaciones J2EE que disponga de un contenedor (*container*) que debe implementar la especificación EJB. Los *enterprise beans* se despliegan sobre ese contenedor que se será el gestor del ciclo del ciclo de vida del *bean* y se interpondrá en todos los accesos al mismo.

Existen tres tipos diferentes de *enterprise beans* [Roman2005]*:*

- ***Session beans***. Modelan los procesos de negocio. En el caso más típico representan un único tipo de cliente dentro del servidor de aplicaciones.

Para acceder a una aplicación que ha sido desplegada en el servidor, el cliente invoca a los métodos del *session bean* y éste ejecuta las tareas de negocio dentro del servidor, ocultando su complejidad al cliente.

- ***Entity beans***. Representan un objeto de negocio dentro de un mecanismo de almacenamiento persistente. El sistema de almacenamiento típico es una base de datos relacional pero la arquitectura permite otros mecanismos tales como bases de datos orientadas a objetos con el requisito de que el fabricante del servidor de aplicaciones los soporte.

- ***Message-driven beans***. Son *enterprise beans* que permiten a una aplicación J2EE procesar mensajes de una manera asíncrona. Actúan como escuchadores de los mensajes que son enviados a cualquier componente J2EE, permitiendo procesar dichos mensajes y realizar acciones cuando se intercepten.

Desde el punto de vista de este trabajo de investigación, los componentes que mayor interés representan de cara a la persistencia son los *entity beans*. Centrándonos en ellos, puede hablarse de dos mecanismos de gestión de persistencia dependiendo de quién se encargue de la interacción con el almacén de datos:

- ***Bean-managed persistence*** **(Persistencia gestionada por el *bean*)**. En este caso son los propios *entity beans* los que realizan las operaciones necesarias para hacerse persistentes. En otras palabras, los desarrolladores son los responsables de escribir el código necesario para transportar los objetos en memoria al almacén de datos subyacente. Esto incluye las operaciones de guardar, cargar y encontrar datos dentro del componente. Debe utilizarse por lo tanto una API de persistencia, típicamente JDBC.

- ***Container-managed persistence*** **(Persistencia gestionada por el contenedor)**. En este caso es el contenedor J2EE el que se encarga de gestionar todos los accesos a base de datos que requiere el *entity bean*. El código del bean no está ligado a ningún mecanismo específico de almacenamiento persistente, por lo que se podría reutilizar el bean en otro servidor de aplicaciones distinto que utilizase otros sistemas de almacenamiento sin necesidad de recompilar el código del *bean*. La persistencia gestionada por el contenedor puede ser implementada a su vez utilizando algún mecanismo de persistencia, como JDO o Hibernate.

Para desarrollar un *Entity Bean* (y también un *Session Bean*) deben crearse los siguientes ficheros [Sun2003d]:

- **Un descriptor de despliegue (*Deployment descriptor*)**. Es un fichero XML que especifica información acerca del bean, por ejemplo el tipo de persistencia y las características de los mecanismos de transacción.

- **La clase del *Enterprise bean***: Implementa los métodos definidos en los interfaces descritos a continuación.

- **Interfaces**: Los clientes accederán a los *beans* únicamente a través de los métodos definidos en sus interfaces. EJB distingue entre el acceso local y remoto diferenciando dos tipos de interfaces:

  - Para permitir un acceso remoto al bean deben proporcionarse un interfaz remoto y otro *home*. El interfaz remoto define los métodos de negocio espe-

cíficos del componente y el interfaz *home* define los métodos relacionados con el ciclo de vida del componente: `create()` y `remove()`. En el caso específico de los *entity beans* el interfaz *home* también define métodos de búsqueda y métodos *home*. Los primeros se utilizan para localizar los *entity beans*. Los segundos son métodos de negocio que son invocados en todas las instancias de la clase del *entity bean*.

- Para un acceso local, deben proporcionarse un interfaz local y otro *local home*. El interfaz local define los métodos de negocio del componente y el interfaz *local home* define los métodos relacionados con su ciclo de vida y métodos de búsqueda.

Cuando se utiliza el sistema de persistencia gestionada por el contenedor éste se encarga también de realizar la gestión de las relaciones entre objetos. Para ello, se declara en el descriptor de persistencia información acerca de las relaciones que deben ser gestionadas. Con esta información, el contenedor se encarga de generar todo el código que gestiona las relaciones dentro de una nueva clase que hereda del *entity bean* correspondiente. Ésta es una de las razones por las cuales el código de los *entity beans* difícilmente puede ser utilizado fuera de los contenedores J2EE.

Con respecto a la gestión de las claves primarias, también cambia dependiendo del escenario arquitectónico que se usen los EJB. En el caso de la persistencia gestionada por el propio *bean*, el desarrollador debe codificar cómo se generan las claves. Cuando la persistencia la gestiona el contenedor J2EE, puede optarse por una generación automática de claves. En ambos casos, pueden utilizarse clases definidas por el usuario que representen los objetos clave si bien estas clases deberán cumplir unos determinados requisitos, que serán más estrictos en el caso de la persistencia gestionada por el contenedor.

EJB define además un lenguaje de consulta denominado EJBQL (*EJB Query Language*), el cual está basado en una mezcla de SQL y OQL. EJBQL puede ser utilizado para implementar los métodos de búsqueda, que son usados por clientes externos del *entity bean*, así como por métodos de selección, utilizados habitualmente por los *entity beans*.

3.6.1.1 Enterprise Java Beans 3.0 (JSR-220)

Los Enterprise Java Beans fueron concebidos como un componente clave de la especificación J2EE, sin embargo, paradójicamente es el estándar de J2EE que más críticas ha recibido. Existe una crítica generaliza en la comunidad Java a su complejidad [Jordan2004]. Además, se desaconseja utilizar la persistencia gestionada por lo *beans*, porque diversos fallos en la especificación EJB impiden el desarrollo de implementaciones eficientes. Centrándonos únicamente en la persistencia gestionada por contenedor, pueden señalarse diversas deficiencias [Bauer2004]:

- Los componentes se definen con una correspondencia uno a uno con las tablas cuando se usa el modelo relacional, es decir, se fuerza al modelo del dominio a estar en primera forma normal. Ello impide en muchas ocasiones aprovechar la ventaja del rico modelo de tipos de Java.

- Los *entity beans* no soportan las asociaciones polimórficas ni consultas polimórficas.

- Los *entity beans* no son portables en la práctica puesto que, a pesar de la existencia del estándar, las implementaciones de los diferentes fabrican-

tes difieren entre sí. En concreto, los metadatos que configuran como se realizan los mapeos entre el modelo de objetos y el almacén utilizado varían a menudo dependiendo de la implementación utilizada.

- Los *entity beans* no son serializables. Una opción cuando se necesitan transportar datos a un cliente remoto es utilizar el patrón *Data Transfer Object* (DTO). El problema del uso intensivo de este patrón es que degenera en la aparición de jerarquías de clases paralelas, donde cada entidad del modelo del dominio queda representada a la vez por un DTO y por un *entity bean*.

- EJB es un modelo intrusivo. Obliga a utilizar un estilo de desarrollo Java muy alejado del modelo natural y hace que la reutilización de código fuera de un contenedor específico sea extremadamente difícil.

Algunos autores [Bauer2004] señalan que la principal razón para el fracaso de los EJB es que el estándar en su totalidad fue creado por un comité. Por el contrario, otras alternativas que sí han tenido más éxito, como la serialización, el patrón DAO (*Data Access Object*) [Alur2001] o las herramientas de mapeo objeto relacional, son el resultado de años de experiencia e investigación.

Es por ello que para la nueva versión del estándar EJB 3.0, en la actualidad en fase de desarrollo y recogida en el JSR-220 [Sun2005], se ha considerado realizar una modificación sustancial de la arquitectura. Dentro de esta revisión, los cambios más notables se han producido en el modelo de *Entity Beans* y persistencia. Esta parte está recogida en un documento independiente dentro del JSR-220.

La especificación JSR-220 reconoce el interés y el éxito del paradigma del mapeo objeto/relacional transparente. Lo que hace es estandarizar la API básica así como los metadatos necesarios para cualquier mecanismo de persistencia objeto/relacional.

En primer lugar, el nuevo sistema de persistencia hace una apuesta clara por objetos POJO (*Plain Old Java Object*). Para proporcionar los metadatos que configuran el mapeo objeto/relacional propone utilizar el sistema de anotaciones estándar de la plataforma Java publicado con la JDK 1.5 o bien utilizar descriptores XML independientes [Sun2005].

Los objetos de dominio se denominan entidades. Serán objetos POJO anotados (o denotados en el descriptor XML) como una entidad (*entity*) y a los que se imponen ciertas restricciones:

- Poseer al menos un constructor sin argumentos.

- No puede ser final, ni sus métodos tampoco serlo.

- El estado de una entidad únicamente estará disponible a los clientes únicamente a través de métodos de acceso (bien con métodos `get()` y `set()` o siguiendo otra nomenclatura).

Estas características permiten que el estándar sea implementado con unas características muy similares a las vistas para Hibernate en § 3.2.6.1: utilizando clases proxies para la carga transparente de objetos persistentes cuando se navega por un grafo de objetos y una estrategia de "inspección" para detectar el estado modificado de un objeto al finalizar la unidad de trabajo.

El ciclo de vida de cada entidad es gestionado por un objeto `EntityManager`. Cada instancia `EntityManager` está asociada con un contexto de persistencia. Un contexto de persistencia es un conjunto de instancias de entidades en el cual para cada entidad persistente únicamente hay una instancia de entidad. Dentro del contexto de persistencia se gestionan las instancias de las entidades así como su ciclo de vida.

El interfaz `EntityManager` define por tanto los métodos que se utilizan para interactuar con el contexto de persistencia. Su ciclo de vida puede estar gestionado tanto por la aplicación (*application-managed entity manager*) como por el contenedor (*container-managed entity manager*).

Con respecto al ciclo de vida de las entidades, cada instancia de una entidad puede estar en uno de los siguientes estados Sun2005:

- ***New*** **(nueva)**. La instancia aún no tiene identidad y todavía no ha sido asociada a un contexto de persistencia.
- ***Managed*** **(gestionada)**. La instancia posee una identidad de persistencia y que está asociada con un contexto de persistencia.
- ***Detached*** **(separada**). La instancia tiene una identidad de persistencia que ya no está asociada con un contexto de persistencia.
- ***Removed*** **(borrada)**. La instancia tiene una identidad de persistencia, está asociada con un contexto de persistencia pero cuya eliminación de la base de datos ya ha sido planificada.

Al igual que Hibernate (§ 3.2.6.1), el sistema de transitividad de la persistencia puede configurarse por alcance o en cascada, ofreciendo la última opción un mayor control sobre cómo se propaga la persistencia.

En cuanto a las consultas, se incluye un nuevo API de consultas orientado a objetos a través del interfaz `Query`. Además, se enriquece la especificación previa del lenguaje EJB-QL [Sun2003d].

La especificación del sistema de persistencia recogido en la JSR-220 ha sido tomada como punto de partida para desarrollar un nuevo estándar de persistencia que aúne los esfuerzos realizados por el grupo de trabajo de JDO 2.0 (JSR-243) y por el propio grupo de trabajo de EJB 3.0 (JSR-220). Este nuevo estándar de persistencia será distribuido de una manera independiente al estándar EJB y funcionará tanto en las plataformas J2SE como J2EE. En el momento de escribir estas líneas ésta se trata de una iniciativa en curso que se conoce con el nombre genérico de *Java Persistence API*.

3.6.2 Spring

El *framework* Spring [Johnson2005] se define como una solución ligera para el desarrollo rápido de aplicaciones J2SE y J2EE. El *framework* se presenta con una arquitectura modular con el objeto de que el desarrollador pueda utilizar módulos concretos de funcionalidad sin tener que preocuparse del resto. Esta arquitectura se muestra en la Figura 3.33.

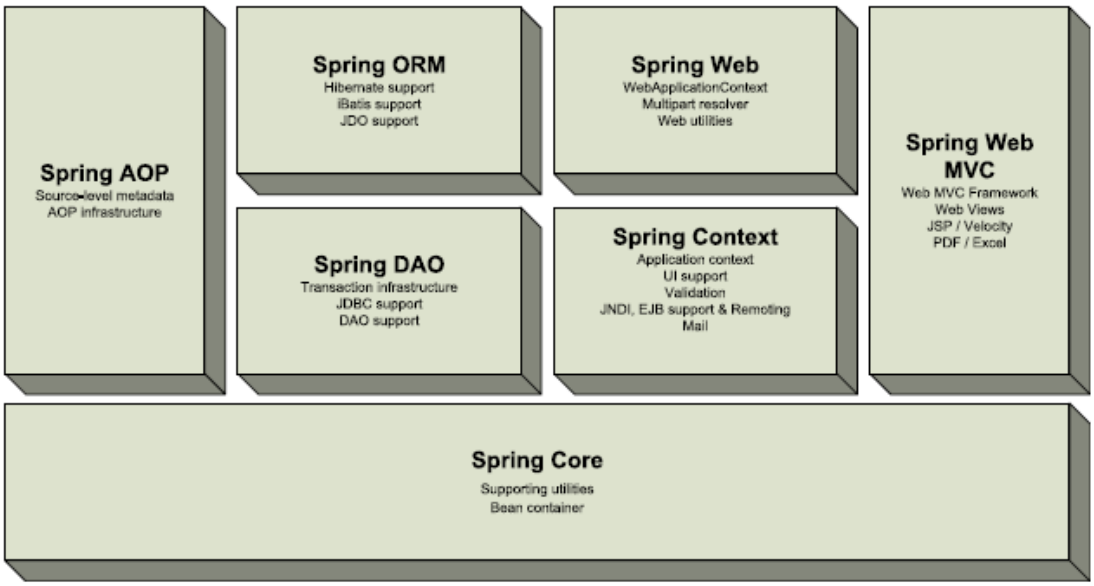


Figura 3.33. Arquitectura de Spring.

Desde el punto de vista de este documento, los módulos que mayor interés presentan de cara a la persistencia son los módulos Spring DAO y Spring ORM (*Object/Relational Mappers*). Antes de estudiar estos módulos es necesario explicar los fundamentos de funcionamiento de Spring, los cuales vienen explicados por el patrón *Dependency Injection* (Inyección de dependencia), también conocido como *Inversion of Control* (Inversión de control, IoC) [Fowler2004].

El módulo Spring Core representa el corazón del *framework* e implementa las funcionalidades del patrón *Dependency Injection*, el cual es utilizado intensivamente por todos los módulos de Spring para la gestión y configuración de recursos. El componente básico es el interfaz `BeanFactory`, el cual aplica el patrón *factory method* [GOF94] para definir una fábrica de Java Beans [Sun96]. Esta fábrica elimina la necesidad crear y configurar programáticamente los Java Beans, típicamente utilizando fábricas *singleton* [GOF94]. Permite desacoplar la configuración y especificación de las dependencias de la lógica del programa. La manera de especificar cómo se crean y configuran los objetos puede ser potencialmente cualquiera, mediante diferentes implementaciones del interfaz `BeanFactory`. El mecanismo más utilizado es la configuración mediante ficheros XML utilizando la implementación dada por la clase `XMLBeanFactory`. En la nomenclatura de Spring, tanto el interfaz `BeanFactory` como el interfaz `ApplicationContext`, que extiende al primero y añade nuevas funcionalidades, se denominan contenedores IoC (*Inversion of control*).

El funcionamiento de la Inversión de control se ilustrará con un ejemplo. El java bean a configurar vendrá dado en la clase `Person` cuyo código se muestra en la Figura 3.34.

```
package example.spring;

public class Person{
	protected String name;
	protected Person spouse;

	public void setName(String name){
		this.name = name;
	}
```

```
        public String getName(){
                return name;
        }

        public void setSpouse(Person spouse){
                this. spouse = spouse;
        }

        public Person getSpouse(){
                return spouse;
        }

}
```

Figura 3.34. Código de la clase persona.

En la Figura 3.35 se muestra el fichero XML necesario para configurar un objeto persona de nombre Pepe cuyo cónyuge es otra nueva persona de nombre María. El atributo `singleton` permite configurar si existirá una única instancia del objeto para un identificador dado o si por el contrario cada vez que se solicite el objeto se creará una nueva instancia.

```
<bean id="pepe" class="example.spring.Person" singleton="false">
  <property name="name"><value>Pepe</value></property>
    <property name="spouse">
      <bean class="org.springframework.beans.TestBean">
        <property name="name"><value>Maria</value></property>
      </bean>
    </property>
</bean>
```

Figura 3.35. XML de configuración del objeto persona.

Desde el código de la aplicación sería necesario instanciar un objeto `XMLBeanFactory` y configurarlo a partir del fichero XML creado. El código necesario se muestra en la Figura 3.36.

```
XmlBeanFactory factory = new XmlBeanFactory(new
       FileInputStream("beans.xml"));
Person pepe = (Person)factory.getBean("pepe",
       example.spring.Person.class);
```

Figura 3.36. Utilización de un BeanFactory para obtener un Bean.

El modulo Spring DAO (*Data Access Object*) facilita la utilización del patrón DAO [Alur2001]. Este patrón propone utilizar un objeto, denominado *Data Access Object*, que medie entre el cliente que accede a unos datos y el sistema de almacenamiento que contiene los datos, de tal manera que los mecanismos necesarios para interactuar con el almacén queden ocultos de cara al cliente. Esto permite poder cambiar el sistema de almacenamiento sin necesidad de modificar el código de la aplicación cliente, puesto que el código de acceso al almacén de datos y el código de la aplicación son independientes. En la Figura 3.37 se muestra el diagrama de clases de este patrón en su versión más general. Además del objeto DAO, el objeto `Data` se corresponde con otro patrón J2EE, el patrón *Transfer Object*, utilizado para transportar datos puros [Alur2001]. Su uso en este patrón es opcional.

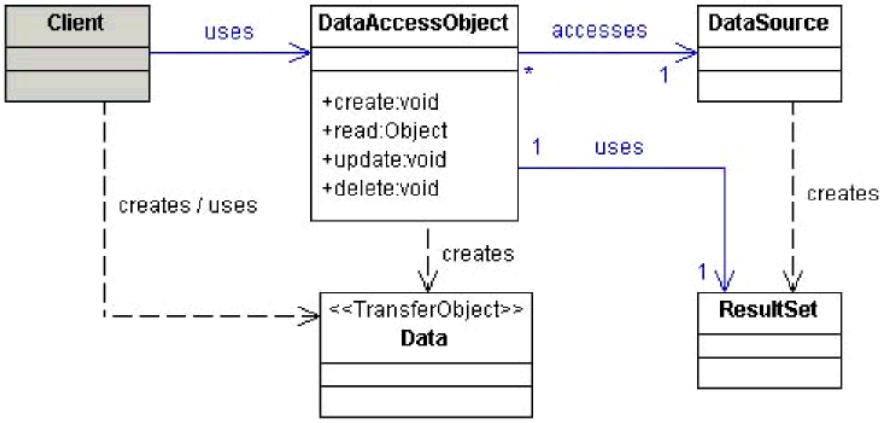


Figura 3.37. Diagrama de clases del patrón Data Access Object.

El soporte DAO en Spring se enfoca principalmente a facilitar el trabajo con diferentes tecnologías como JDBC, Hibernate o JDO haciendo que éste se realice de una manera estándar y uniforme, con el objetivo de poder cambiar la tecnología escogida de una manera sencilla. Para ello ofrece una jerarquía común de excepciones y un mapeo automático de las excepciones específicas de cada tecnología con las excepciones comunes de Spring.

Spring proporciona un conjunto de clases abstractas DAO que pueden ser extendidas por el desarrollador. Estas clases abstractas proporcionan métodos para establecer la fuente de datos (*data source*) y otros aspectos de configuración que son específicos de la tecnología utilizada. Estas clases se denominan `JdbcDaoSupport`, `HibernateDaoSupport` y `JdoDaoSupport` dependiendo de la tecnología a la que den soporte. Están pensadas para ser extendidas facilitando la creación de objetos DAO. Para ello ofrecen acceso a un objeto denominado respectivamente `JdbcTemplate`, `HibernateTemplate` o `JdoTemplate`. Este objeto permite ejecutar acciones específicas con la tecnología subyacente encargándose de realizar gran parte de las tareas repetitivas que son necesarias para ejecutar la acción, como por ejemplo la necesidad de adquirir y liberar recursos. El enfoque utilizado para permitir la configuración de acciones específicas y su ejecución en un contexto determinado es utilizar el patrón *Command* [GOF94] para suministrar las acciones que serán ejecutadas vía *callback* en el contexto de la plantilla. Se trata por tanto del patrón *Template method* (GOF94) siendo el método a ejecutar modelado como un objeto comando.

Para el caso de JDBC, el soporte al patrón DAO lo proporciona principalmente la clase `JdbcTemplate`. Su función es simplificar el uso de JDBC encargándose de gestionar la creación y liberación de recursos. Ayuda a evitar errores comunes como por ejemplo olvidarse de cerrar la conexión. Se encarga de ejecutar las tareas comunes de JDBC como la creación de sentencias y su ejecución, dejando que sea el código de la aplicación el encargado de proporcionar el SQL necesario y extraer los resultados. Además se encarga de capturar las excepciones JDBC y las traduce a la jerarquía genérica de excepciones DAO de Spring. En la Figura 3.38 puede observarse la utilización de `JdbcTemplate` para ejecutar una sentencia SQL. La creación de un objeto con la propiedad `dataSource` permite configurar dicha propiedad mediante inversión de control.

```
public class ExecuteAStatement {
      private JdbcTemplate jdbcTemplate;
```

```
	private DataSource dataSource;

	public void doExecute() {
		jdbcTemplate = new JdbcTemplate(dataSource);
		jdbcTemplate.execute("create table mytable (id integer,
		name varchar(100))");
	}
	public void setDataSource(DataSource dataSource) {
		this.dataSource = dataSource;
	}
}
```

Figura 3.38. Ejecución de una sentencia SQL utilizando el soporte de Spring.

Además, se ofrece un conjunto de objetos que permiten modelar las consultas, actualizaciones y procedimientos almacenados como objetos independientes. Spring implementa esta funcionalidad utilizando JDO [Roos2003], si bien los objetos devueltos no están obviamente ligados a la base de datos.

El soporte para la integración de herramientas de mapeo objeto/relacional viene implementado en el módulo Spring ORM. Spring puede integrarse[6] con Hibernate, JDO, Oracle TopLink, Apache OJB e iBATIS SQL Maps. Esta integración se produce en términos de gestión de recursos, soporte para la implementación del patrón DAO[7] y estrategias de transacciones. Además es consistente con las jerarquías genéricas de excepciones DAO y transacciones de Spring.

Existen dos enfoques para esta integración: extender las clases DAO de Spring o codificar los DAOs utilizando la API de la tecnología correspondiente. En ambos casos, los DAO se configurarán utilizando la Inyección de Dependencias (*dependency injection*).

Con respecto a la gestión de recursos, Spring permite gestionar mediante inversión de control la creación y configuración de los objetos necesarios para utilizar las herramientas de mapeo objeto/relacional soportadas.

Por ejemplo, en la Figura 3.39 se muestra cómo puede configurarse un `DataSource` JDBC y utilizarlo como una de las propiedades con las que se configura un `SessionFactory` Hibernate. De este modo, la fábrica de sesiones Hibernate podría estar disponible desde cualquier objeto de la aplicación, incluso pudiendo obtenerse de una manera declarativa como parte de una propiedad de un java bean que se configure mediante XML.

```
<beans>
  <bean id="myDataSource"
class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>
    <property name="username" value="sa"/>
    <property name="password" value=""/>
  </bean>

  <bean id="mySessionFactory"
class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource"/>
```

[6] En el momento de escribir estas líneas la última versión de Spring es la 1.2.6.

[7] Aunque se utilice una herramienta de mapeo objeto/relacional, la creación de objetos DAO es considerada una buena práctica para limpiar la capa de aplicación de cualquier acceso a datos o gestión de transacciones [Bauer2004].

```
    <property name="mappingResources">
      <list>
        <value>product.hbm.xml</value>
      </list>
    </property>

    <property name="hibernateProperties">
      <props>
        <prop
key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>
      </props>
    </property>
  </bean>
...
</beans>
```

Figura 3.39. Configuración de los recursos Hibernate desde Spring.

La clase de `HibernateTemplate` se encarga de asegurar que las sesiones hibernate son abiertas y cerradas adecuadamente, además de interactuar automáticamente con el sistema de transacciones. Las acciones a ejecutar se crean implementando el interfaz `HibernateCallback`. Spring proporciona además la clase base `HibernateDaoSupport` que define la propiedad `sessionFactory`, que podría ser configurada por inversión de control, y el método `getHibernateTemplate()` para que sea utilizado por las subclases. De este modo, se simplifica la creación de objetos DAOs para los requerimientos más habituales. El soporte a JDO en Spring es muy similar al de Hibernate.

Una de las abstracciones más importantes del *framework* Spring es la gestión de transacciones. Entre las características que ofrece destacan [Johnson2005] las siguientes:

- Proporciona un modelo de programación consistente transversal a diferentes APIs de transacciones como las proporcionadas por JTA, JDB, Hibernate y JDO.
- Proporciona una API para la gestión programática de transacciones más sencilla de utilizar que la mayor parte de las APIs anteriores.
- Se integra con la abstracción de acceso a datos de Spring.
- Soporta la gestión de transacciones declarativa.

El elemento clave del sistema de transacciones de Spring es la "estrategia de transacción". Esta estrategia se representa con el interfaz `PlatformTransactionManager` (Figura 3.40).

```
public interface PlatformTransactionManager {
  TransactionStatus getTransaction(TransactionDefinition definition)
    throws TransactionException;
  void commit(TransactionStatus status) throws TransactionException;
  void rollback(TransactionStatus status) throws TransactionException;
}
```

Figura 3.40. Interfaz PlatformTransactionManager.

Las diferentes implementaciones de este interfaz se definirán como cualquier otro objeto dentro de un contenedor IoC. En la Figura 3.41 se muestra el código XML necesario para crear un gestor de transacciones Hibernate.

```
<bean id="txManager" class="org.springframework.orm.hibernate.
      HibernateTransactionManager">
      <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

Figura 3.41. Creación de un gestor de transacciones Hibernate.

Para la gestión programática de transacciones, se proporciona el objeto `TransactionTemplate`. Este objeto funciona de forma similar a otros objetos *template* como `HibernateTemplate`. Utiliza un enfoque de *callback* para evitar a la aplicación la necesidad de adquirir y liberar recursos. Para poder utilizar un objeto `TransactionTemplate` éste debe ser configurado con un objeto `PlatformTransactionManager`. Esta configuración se realizará normalmente mediante inversión de control. En la Figura 3.42 se muestra un ejemplo de utilización del objeto `HibernateTemplate`. Es importante señalar que, si se desease cambiar la tecnología de acceso a datos utilizada, únicamente habría que modificar el fichero XML donde se configura el gestor de transacciones específico (Figura 3.41).

```
Object result = transactionTemplate.execute(new
     TransactionCallback() {
          public Object doInTransaction(TransactionStatus
          status) {
               operation1();
               return operation2();
          }
});
```

Figura 3.42. Gestión programática de transacciones.

Spring ofrece además la posibilidad de realizar una gestión de transacciones declarativa. Para ello hace uso del módulo Spring AOP (*Aspect Oriented Programming*), el cual permite utilizar características de programación orientada a aspectos para extender la funcionalidad del *framework*. La programación orientada a aspectos será revisada en profundidad en el Capítulo 5.

La principal ventaja de la gestión de transacciones declarativa es que reduce al mínimo el impacto de ésta en el código de la aplicación. Otras características destacables son:

- La gestión declarativa de transacciones funciona con cualquiera de los entornos de persistencia soportados.
- El sistema puede aplicarse a cualquier objeto POJO (*Plain Old Java Object*).
- Pueden configurarse reglas de *rollback* de una forma declarativa. Estas reglas permiten especificar qué excepciones deberían provocar un *rollback* de una transacción al ser lanzadas. Esta especificación se realiza de forma declarativa. Esto evita que los objetos de negocio dependan de la infraestructura de transacciones. En concreto, no necesitan importar ninguna API de Spring ni de transacciones.
- Spring ofrece la oportunidad de personalizar el comportamiento transaccional utilizando programación orientada a aspectos. Por ejemplo, puede

insertarse un determinado comportamiento a ejecutarse cuando se produzca un *rollback* de una transacción.

La manera habitual de trabajar con transacciones declarativas es utilizar la clase `TransactionProxyFactoryBean`. El módulo de AOP de Spring es implementado haciendo uso de *proxies* generados dinámicamente (si bien acepta su integración con otros sistemas AOP más sofisticados, como AspectJ). Para ello hace uso de las *Java Dynamic Proxy Classes* [Sun99] para crear *proxies* para clases que implementan interfaces y de GCLIB [CGLIB] cuando no hay interfaces implementados en la clase para la cual se genera el *proxy*. `TransactionProxyFactoryBean` es únicamente una versión especializada de la clase genérica `ProxyFactoryBean` que, además de crear un *proxy* para envolver el objeto correspondiente (*target object*), le añade un objeto `TransactionInterceptor`. Este objeto contiene la información relativa a qué métodos y propiedades requieren un tratamiento transaccional y cómo se configura ese tratamiento.

En la Figura 3.43 se muestra un ejemplo de la gestión declarativa de transacciones con Spring. En él se crea un *proxy* que envolverá al objeto Persona de identificador `pepe` cuya configuración se muestra en la Figura 3.35. Este *proxy* se configura para hacer uso del gestor de transacciones con el identificador `txManager` (Figura 3.41). Finalmente se proporcionan los atributos que configurarán el comportamiento transaccional. Los atributos se basan en la nomenclatura utilizada por los contenedores EJB. En el caso del atributo `PROPAGATION_REQUIRED`, declara que la transacción utilizada para ejecutar el método asociado podrá ser una de las transacciones en curso o, de no existir ninguna, deberá crearse una nueva. En el ejemplo, esta propiedad se ha asociado a los métodos del tipo `set()`. Al resto de métodos del objeto se les asocia además la propiedad de "transacción de sólo lectura" mediante el atributo `readOnly`.

```
<bean id="personStore"
class="org.springframework.transaction.interceptor.TransactionProxyFact
oryBean">
      <property name="transactionManager" ref="txManager"/>

      <property name="target" ref="pepe"/>

      <property name="transactionAttributes">
            <props>
                  <prop key="set*">PROPAGATION_REQUIRED </prop>
                  <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
            </props>
      </property>
</bean>
```

Figura 3.43. Gestión de transacciones declarativa con Spring.

Debe señalarse que el sistema de gestión de transacciones de Spring no necesita un servidor de aplicaciones para funcionar como el caso de los Enterprise Java Beans con las transacciones gestionadas por el contenedor (*Container managed transactions*, CMT) [Roman2005].

3.6.3 Aportaciones y Carencias de los Sistemas Estudiados

El sistema de persistencia propuesto por los Enterprise Java Beans (EJB) ofrece resultados muy pobres en cuanto a su transparencia. En primer lugar, para transformar cualquier clase Java existente en un EJB (en el caso de objetos del mo-

delo del dominio en un *Entity Bean*) requiere cambios considerables y el conjunto de clases e interfaces resultantes es muy dependiente del *framework* EJB, por lo que la incumbencia de la persistencia no queda separada de la lógica de la aplicación (requisito § 2.1.1). Al igual que otros sistemas vistos en § 3.2, los EJB siguen un ciclo de vida definido, pero es más visible al programador debido a que se refleja a través del gran número de métodos que deben ser implementados en las clases de los EJB para proporcionar las llamadas *callback* al contenedor [Jordan2004]. Esto obliga a realizar un gran esfuerzo de programación para cumplir los requisitos del contenedor, no los requisitos del dominio del problema.

El proceso de desarrollo es a su vez más complejo debido a la necesidad de empaquetar las clases EJB junto con un descriptor de despliegue XML en archivos JAR que serán gestionados por el servidor de aplicaciones (violación del requisito § 2.1.5). Los EJB pueden funcionar únicamente en el ámbito de un contenedor EJB funcionando en un servidor de aplicaciones, por lo que no pueden adaptarse a la funcionalidad de cualquier aplicación J2SE que se ejecute aisladamente (requisito § 2.1.2).

En el caso concreto de la persistencia gestionada por el contenedor, las relaciones entre objetos serán gestionadas íntegramente por el contenedor, lo que impide reutilizar el modelo de dominio en aplicaciones que no se basen en EJB. Esto impide además hacer uso las relaciones entre objetos representadas a través de código Java utilizando objetos `Collection` (requisito § 2.1.3). Además, EJB define además un modelo de objetos propio que es menos potente que el definido por el lenguaje Java. Existen diversas características del lenguaje como hilos y herencia cuyo uso está desaconsejado con EJB.

En cuanto a los requisitos de adaptabilidad, EJB ofrece una cierta adaptabilidad sobre el sistema de persistencia a través de los mecanismos declarativos de configuración soportados en los descriptores de despliegue XML. Las posibilidades de configuración están enfocadas principalmente a requisitos de eficiencia y escalabilidad. No obstante esta adaptabilidad no es dinámica (§ 2.2.1), puesto que no se puede cambiar la configuración en tiempo de ejecución sino cuando se despliega el EJB. Tampoco se ofrece ningún soporte para la adaptabilidad programática (§ 2.2.2).

EJB permite configurar distintos almacenes de objetos (requisito § 2.2.4), aunque esta característica depende de los fabricantes de contenedores EJB. A través de la arquitectura J2EE Connector se definen una serie de contratos a nivel de sistema entre el servidor de aplicaciones y conectores que permiten integrar otros sistemas. Por ejemplo, a través de esta arquitectura puede utilizarse JDO para la persistencia de los *Entity Beans*.

EJB sí ofrece una automatización integral de la persistencia (§ 2.1.4). Cuando se utilizan las transacciones gestionadas por el contenedor, estas pueden ser especificadas declarativamente sin tener que modificar el código. Será el contenedor el que se encargue de gestionar todas las unidades transaccionales y la actualización de objetos en el almacén. Sin embargo, no se ofrece la posibilidad de añadir o modificar las políticas de actualización existentes tal y como se recoge en el requisito § 2.2.5.

En cuanto a la selección de objetos persistentes, para cada objeto del modelo de dominio persistente debe construirse un *Entity Bean* y en los descriptores de despliegue deben modelarse las relaciones entre éstos. Se obtiene por lo tanto un gran control sobre qué objetos son persistentes (requisito § 2.2.6) a costa de un alto

coste de desarrollo. Por otra parte, los EJBs se han mostrado como componentes muy pesados. Esto ha hecho descartar la noción original de que un *entity bean* representa una fila en una tabla de la base de datos. Incluso se recomienda [Alur2001] a los desarrolladores utilizar *entity beans* para representar objetos gruesos con el fin de evitar problemas de rendimiento y escalabilidad.

Con respecto a la nueva propuesta de especificación de los Enterprise Java Beans 3.0 (JSR-220) revisada en § 3.6.1.1, las conclusiones relativas a las aportaciones y carencias del nuevo sistema de persistencia son análogas a las realizadas para Hibernate en § 3.2.6.1 debido a las similitudes existentes entre ambas aproximaciones.

Analizando ahora el sistema de persistencia propuesto en Spring, lo primero que debe señalarse es que este *framework* no ofrece un sistema de persistencia en sí mismo, sino que ofrece la posibilidad de integrar diferentes sistemas de persistencia minimizando el impacto que tengan éstos en la lógica de la aplicación. Por ello, aunque sus objetivos están alineados con el requisito § 2.1.1, los requisitos recogidos en el Capítulo 2 no son plenamente aplicables.

Gracias al uso del patrón de Inversión de Control se simplifica mucho la configuración de los recursos necesarios para utilizar los diferentes mecanismos de persistencia soportados. Esto se traduce en una reducción considerable de las líneas de código fuente necesarias, además de facilitar los cambios de configuración en los mismos. Características tales como el soporte para facilitar el desarrollo de objetos DAO (Data Access Object), la definición de una jerarquía de excepciones común y la traducción automática de excepciones permiten intercambiar con mayor facilidad los mecanismos de persistencia subyacentes, pero este cambio siempre requerirá modificaciones a nivel de código fuente, por lo que no se verifica el requisito § 2.2.4. Bajo esta perspectiva, Spring supone un avance cuantitativo para el desarrollador que utiliza los sistemas de persistencia que Spring integra.

El sistema de transacciones que propone Spring sí supone un avance cualitativo con respecto a los sistemas que integra, puesto que permite realizar una gestión declarativa de las transacciones liberando al programador de tener que realizar esta tarea programáticamente. De este modo acerca a los sistemas que integra, que incluyen JDO e Hibernate ((§ 3.2.6), a un sistema con automatización integral de la persistencia (§ 2.1.4). Además, en este caso sí se permite intercambiar el tipo de gestor de transacciones sin modificar el código fuente gracias al uso de la Inversión de control.

En cuanto a la independencia del sistema operativo (§ 2.3.2) y del hardware (§ 2.3.3) se consiguen tanto en EJB como en Spring gracias a las características de portabilidad de la plataforma Java.

3.7 Motores de Persistencia Transparentes

Los motores de persistencia transparentes persiguen que las aplicaciones que los utilicen no tengan que realizar explícitamente gran parte de las tareas habitualmente asociadas al trabajo con mecanismos de almacenamiento. Ejemplos de estas tareas son guardar los objetos en el sistema de almacenamiento, comprobar si han sufrido cambios en su estado y actualizarlos, o tener que realizar una consulta para obtener cada uno de los objetos persistentes almacenados.

Con este enfoque, los objetos de aplicación no deberían contener código relativo al sistema de almacenamiento, como por ejemplo sentencias SQL u operaciones con ficheros. Además, los objetos persistentes podrían ser obtenidos a través de una navegación normal por el grafo de objetos, siendo éstos cargados de manera transparente desde el sistema de almacenamiento cuando fuese necesario. Sobre este planteamiento normalmente se realizan ciertas relajaciones. Es habitual la navegación transparente por el grafo de objetos se realice partiendo de un objeto raíz que sí haya sido explícitamente recuperado del sistema de persistencia. Además, se acepta que desde el código de la aplicación se generen eventos relacionados con el sistema de persistencia, como por ejemplo la confirmación de una transacción, si bien estos eventos deberían ser automatizados tanto como fuese posible.

Varios de los sistemas de persistencia estudiados en § 3.2 persiguen ofrecer en mayor o menor grado transparencia en la persistencia, lo que demuestra el éxito de este enfoque en el ámbito de desarrollo empresarial.

3.7.1 ZOPE Object Database (ZODB)

ZODB (ZOPE *Object Database*) es una base de datos orientada a objetos específicamente diseñada para el lenguaje Python [Rossum2001]. Fue desarrollada como un componente fundamental del gestor de contenidos ZOPE, si bien terminó adquiriendo entidad propia conviviendo ambos en la actualidad como proyectos independientes. Puede ser utilizada de forma aislada como sistema de persistencia para cualquier aplicación Python. La arquitectura de ZODB se muestra en el diagrama de clases de la Figura 3.44.

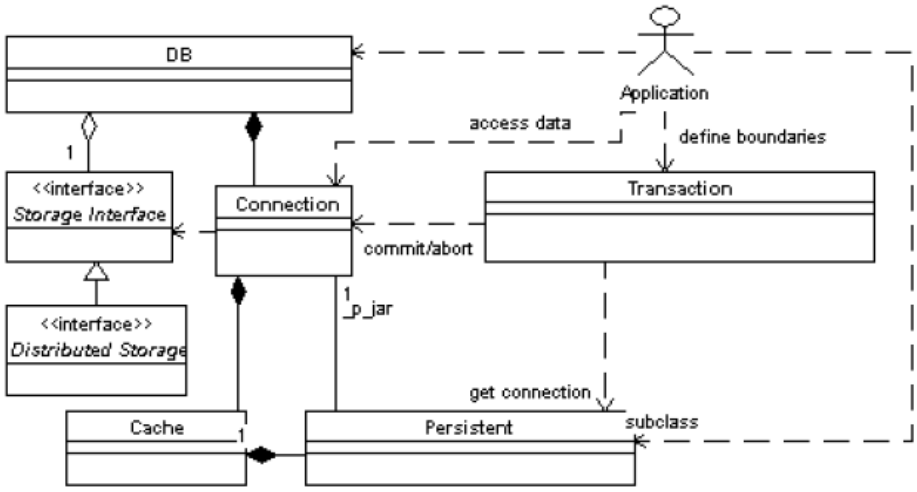


Figura 3.44. Arquitectura de ZODB.

Para utilizar ZOPE el primer paso es declarar como persistentes las clases deseadas. Para marcar una clase como persistente ésta debe extender la clase `Persistent`. El funcionamiento de ZODB se ilustrará con un sencillo ejemplo. En la Figura 3.45 se muestra cómo se declara la clase `User` y se marca como persistente. En este ejemplo, la clase `User` será únicamente un contenedor de atributos por simplicidad. La adición de métodos no tendría impacto en el tratamiento de la clase por parte de ZODB.

```
from persistent import Persistent

class User(Persistent):
```

```
 pass
```

Figura 3.45. Declaración de una clase como persistente en ZODB.

ZODB usa persistencia por alcance [Atkinson95]. Partiendo de un conjunto de objetos raíz, todos los atributos de esos objetos son hechos persistentes, independientemente de que sean tipos de datos Python o instancias de clase. No existen métodos para almacenar explícitamente objetos en una base de datos ZODB. Su almacenamiento en la base de datos se desencadenará por su mera asignación como atributo de un objeto persistente o su almacenamiento en un contenedor persistente. El requisito es que la cadena de contención debe alcanzar finalmente el objeto raíz de la base de datos.

El siguiente paso será preparar el acceso a la base de datos. Primero se abrirá el sistema de almacenamiento representado por el interfaz `Storage`. Luego se creará una instancia de la base de datos que utilizará el mecanismo de almacenamiento declarado. Finalmente se obtendrá una conexión a la base de datos sobre la que trabajar, representado por un objeto `Connection`. En la se muestra el código necesario para configurar una base de datos sobre el fichero `prueba.fs`.

```
from ZODB import FileStorage, DB

storage = FileStorage.FileStorage('/tmp/test-filestorage.fs')
db = DB(storage)
conn = db.open()
```

Figura 3.46. Preparación de la base de datos para su acceso.

Para comenzar a trabajar con la base de datos en primer lugar debe obtenerse el objeto raíz de ZODB utilizando el método `root()` de la instancia `Connection`. Este objeto raíz se comporta como cualquier diccionario Python, de tal manera que puede añadirse simplemente un nuevo par clave/valor para el objeto raíz de la aplicación. Como objeto raíz contenedor de todos los usuarios se utiliza en el ejemplo de la Figura 3.47 un árbol binario `OOBTree`.

```
dbroot = conn.root()

from BTrees.OOBTree import OOBTree

userdb = OOBTree()
dbroot['userdb'] = userdb
```

Figura 3.47. Configuración de los objetos raíz de la aplicación.

Desde este momento puede trabajarse con el objeto raíz de la base de datos o sobre cualquier objeto alcanzable desde este, utilizando el modelo natural de objetos de Python. Los mecanismos relativos a la persistencia se desencadenarán de forma transparente, con la salvedad de la acometida de transacciones que deberá ser realizada de forma explícita. En el ejemplo de la Figura 3.48 se muestra cómo puede crearse un usuario como un objeto Python normal, se le asignan una serie de propiedades y posteriormente se almacena en el objeto raíz de la aplicación, que a su vez está contenido en el objeto raíz de la base de datos. El único código relativo al sistema de persistencia es el relativo a la demarcación de la transacción.

```
import transaction

# Creación de un usuario
```

```
newuser = User()
newuser.id = 'pepe'
newuser.name = 'Pepe';

# Inserción en el objeto raíz de la aplicación
userdb[newuser.id] = newuser

# Acometida del cambio
transaction.commit()
```

Figura 3.48. Interacción con objetos persistentes.

ZODB utiliza varios ganchos proporcionados por Python para capturar el acceso a atributos pudiendo de este modo detectar la mayor parte de las maneras de modificar un objeto. Cuando se detecta que el estado de un objeto ha sido modificado, éste se marca como "sucio" y será actualizado en la base de datos cuando se acometa la transacción en curso.

Este mecanismo para detectar qué objetos han sido modificados impone a su vez ciertas restricciones que afectan a la transparencia de ZODB.

En primer lugar, si se modifica un objeto que es a su vez el valor del atributo de otro objeto, ZODB no puede capturar ese comportamiento y no podrá marcar dicho objeto como sucio. Si el objeto en cuestión es un contenedor Python, pueden utilizarse los envoltorios (*wrappers*) que proporciona ZODB: `PersistentList` y `PersitentMapping`. Si este no es el caso o no se desean utilizar envoltorios, como solución general puede establecerse el estado de un objeto a sucio programáticamente.

El principal gancho de Python utilizado para detectar el acceso a los atributos es la redefinición de los métodos `__getattr__` y `__setattr__` en la clase `Persistent`, superclase de todo objeto persistente en ZODB. Estos métodos especiales de Python permiten detectar cuándo un atributo es leído o modificado. Aquí se encuentra la segunda restricción que impone ZODB a las clases persistentes: si se redefinen estos métodos debe marcarse el objeto como sucio de la manera apropiada y además deben tenerse en cuenta ciertas restricciones en su reimplementación.

Por un motivo similar no se deberían redefinir los métodos `__getstate__` y `__setstate__`. Son llamados por el módulo `pickle` (empleado por ZODB para la transmisión de objetos al almacenamiento) cuando se envía o recupera un objeto de un canal, y por tanto sufren de los mismos problemas.

La última restricción es que una clase persistente nunca debería tener un método `__del__`. La razón es que la base de datos mueve los objetos entre la memoria principal y el medio de almacenamiento de manera transparente al usuario. Por ejemplo, si un objeto no ha sido utilizado en un período de tiempo, podría ser liberado y sus contenidos vueltos a cargar del sistema de almacenamiento la siguiente vez que se use. Debido a que el intérprete de Python no es consciente de la existencia del sistema de persistencia, llamaría al método `__del__` cada vez que el objeto se liberase.

Los objetos persistentes utilizados en ZODB pasan por diversos estados a lo largo de su vida. Estos estados y sus transiciones conforman el modelo de ciclo de vida que se muestra en la Figura 3.49. Los estados son:

- **No guardado**: Cuando un objeto se crea por primera vez, se encuentra en el estado no guardado. Un objeto en este estado puede pasar al estado actualizado cuando se almacena en la base de datos, referenciándolo en un objeto que ya esté almacenado en ella y emitiendo entonces una transacción. Cuando el objeto se guarda por primera vez en la base de datos, se almacena una copia y el objeto entra en el estado actualizado. Un objeto no guardado puede terminar de existir de la misma forma que cualquier otro objeto de Python.

- **Actualizado**: En este estado se encuentran aquellos objetos que han sido guardados en la base de datos y cuyo estado se encuentra cargado en memoria. Un objeto en el estado actualizado pasa al estado cambiado al modificarse. En el instante de la transición, el objeto se registra con el sistema de gestión de transacciones, de forma que su estado se almacene si se acomete una transacción, o vuelto atrás si la transacción se aborta. Si un objeto en el estado actualizado no se usa en un largo periodo de tiempo, el gestor de caché puede decidir desactivarlo y liberar su estado. El objeto sigue en memoria, y transiciona al estado fantasma. Un objeto que esté en el estado actualizado pero sólo esté siendo referenciado por el gestor de caché puede ser eliminado de la memoria.

- **Cambiado**: Los objetos entran en este estado al modificarse. Si se acomete la transacción, el estado del objeto se guarda en la base de datos y el objeto pasa al estado actualizado. Si la transacción en curso se aborta, el objeto se desactiva y pasa al estado fantasma.

- **Fantasma**: El objeto existe en memoria pero sin estado interno, sólo mantiene su presencia. Si se accede a algún atributo del objeto, se carga el estado interno de la base de datos y el objeto pasa al estado actualizado. También se puede cambiar un atributo del objeto, en cuyo caso se pasa al estado cambiado. Un objeto que esté en el estado fantasma pero sólo esté siendo referenciado por el gestor de caché puede ser eliminado de la memoria.

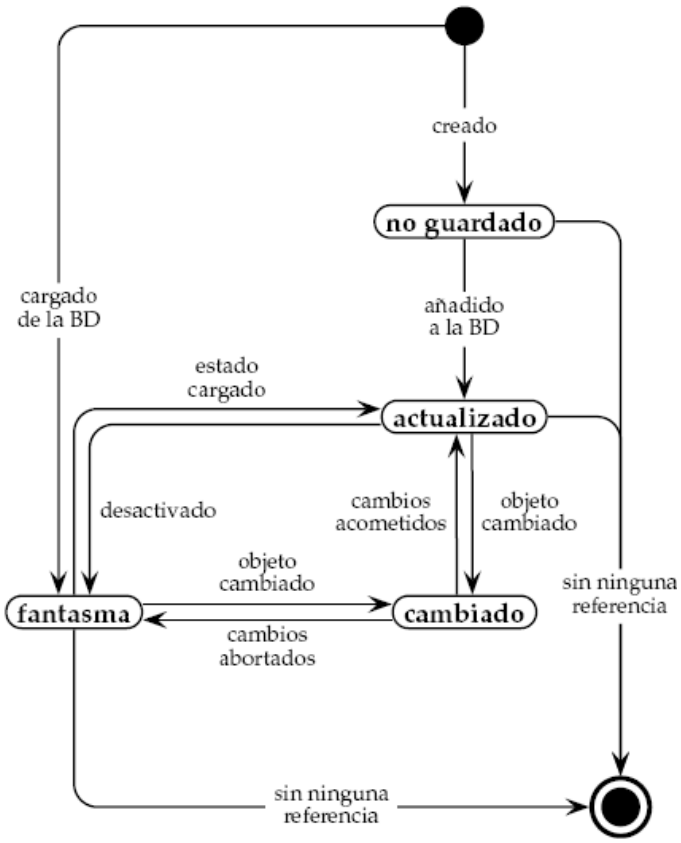


Figura 3.49. Ciclo de vida de los objetos persistentes en ZODB.

3.7.2 APE

APE (*Adaptable Persistence Engine*) es un motor de persistencia construido sobre ZODB. Mientras que ZODB define una base de datos orientada a objetos propia, Ape permite su integración con SGBD relacionales o, en última instancia, con cualquier sistema de almacenamiento que se desee. Permite a los desarrolladores almacenar objetos ZODB en bases de datos y formatos arbitrarios sin necesidad de modificar el código fuente de la aplicación.

La arquitectura de APE comprende los siguientes componentes:

- ***Mappers*** **(Mapeadores)**. Son los componentes encargados de serializar y deserializar objetos, así como de su almacenamiento y carga en el almacén persistente. APE distingue entre los procesos de serializar y almacenar, haciendo posible reutilizar los serializadores con múltiples mecanismos de almacenamiento. Un mapeador proporciona un serializador que se encarga de extraer e insertar el estado de los objetos y una puerta de enlace (*gateway*), que se encarga de almacenar y recuperar el estado de un almacén persistente. La configuración de los mapeadores se realiza mediante ficheros XML donde se especifican los mapeadores y puertas de enlace de cada mapeador.

- ***Schemas*** **(Esquemas).** Los objetos esquema definen el formato de los datos que son intercambiados entre los serializadores y las puertas de enlace. APE define tres esquemas a través de tres clases: `FieldSchema`,

`RowSchema` y `RowSequenceSchema`; el primero permite declarar que los datos intercambiados son campos sencillos, el segundo que son múltiples campos y el último que se intercambia una lista de filas de campos. APE permite utilizar esquemas definidos por el usuario, el único requerimiento es que implementen la operación de igualdad de Pitón, `__eq__`, con el fin de que el sistema pueda verificar que los serializadores y las puertas de enlace son compatibles.

- ***Gateways*** **(Puertas de enlace)**. Las puertas de enlace cargan y almacenan el estado serializado de los objetos. Pueden crearse puertas de enlace para almacenar datos en cualquier tipo de almacén en cualquier formato. El estado devuelto por el método de las puertas de enlace `load()` debe ser conforme al esquema declarado por la puerta de enlace. En el sentido inverso, la puerta de enlace esperará que el estado pasado al método `store()` sea conforme al mismo esquema.

- ***Serializers*** **(Serializadores).** Los serializadores se encargan de extraer los datos que representan el estado de un objeto y de introducirlos dentro de él. El método `serialize()` lee el estado interno de un objeto sin modificarlo. El método `deserialize()` instala el estado dentro del objeto.

- ***Classifiers*** **(Clasificadores).** Los clasificadores son los componentes que se encargan de seleccionar los mapeadores que se utilizarán para cada objeto o registro de la base de datos. Habrá un clasificador por cada base de datos de objetos. Los clasificadores pueden ser sencillos, utilizando siempre un mapeador específico para un identificador de objeto dado o incluso almacenando el nombre del mapeador en la base de datos. Los clasificadores también pueden ser complejos, utilizando atributos o metadatos para elegir el mapeador adecuado.

APE se distribuye con dos mapeadores por defecto: un mapeador para el sistema de archivos de Zope2 y un mapeador de SQL. Además de ser directamente utilizables proporcionan un patrón del que partir a la hora de crear configuraciones de mapeadores propias.

Para cargar un objeto, Ape solicita que la puerta de enlace de un mapeador específico cargue los datos. La puerta de enlace interroga la base de datos y devuelve un resultado. Las puertas de enlace pueden delegar la petición a múltiples puertas de enlace sencillas utilizando puertas de enlace compuestas. Estas puertas de enlace compuestas resultan de aplicar el patrón de diseño *Composite* [GOF94] y se encargan de combinar los resultados en un diccionario que mapea los nombres de las puertas de enlace con los resultados del almacén de datos.

APE se encarga de enviar el resultado obtenido al serializador del mapeador, que podría incluso existir en una máquina diferente a la de la puerta de enlace. El serializador instala los datos cargados en el objeto dentro del objeto que está siendo deserializado. Como las puertas de enlace, los serializadores pueden a su vez organizarse como un serializador compuesto que delega en múltiples serializadores sencillos. Finalmente se devuelve el control a la aplicación.

Cuando se almacenan objetos, el sistema utiliza los mismos componentes pero en el orden inverso. El serializador lee el objeto y los resultados se envían a la puerta de enlace, la cual almacena los datos.

ZODB es la clave que permite cargar y almacenar objetos en el momento adecuado. La clase base `Persistent` notifica a APE cuando un objeto necesita ser almacenado o cargado. No obstante, el *framework* puede ser utilizado fuera del ámbito de ZODB. Por ejemplo, el sistema de mapeadores puede reutilizarse para importar y exportar objetos o para sincronizar un modelo de objetos con un almacén de datos.

3.7.3 Aportaciones y Carencias de los Sistemas Estudiados

ZODB se presenta como un sistema de persistencia altamente transparente, aunque presenta ciertas restricciones que impiden que la incumbencia de persistencia quede completamente separada de la lógica de la aplicación (requisito § 2.1.1).

En primer lugar, tal y como se vio en § 3.7.1 existen circunstancias específicas en las que el programador debe gestionar la designación de objetos como "modificados" (violación del requisito § 2.1.4). Estas circunstancias se darán con frecuencia al utilizar contenedores que forman parte del estado de un objeto, por lo que ZODB proporciona implementaciones especiales de determinados contenedores (`PersistentList` y `PersitentMapping`) violando de este modo el requisito § 2.1.3. Además, debido a los mecanismos de detección de la modificación de objetos y a la utilización del módulo `pickle`, debe prestarse atención a la implementación de determinados métodos especiales en las clases persistentes tal y como se vio en § 3.7.1.

Además, las transacciones deben ser delimitadas programáticamente con lo que no se ofrece una automatización integral de la persistencia (requisito § 2.1.4). En este sentido, la política de actualización de objetos siempre es acometida por las transacciones no permitiéndose la configuración de ninguna otra política (requisito § 2.2.5).

ZODB ofrece un mecanismo de persistencia por alcance, no permitiendo detallar de ningún modo como se propaga la persistencia por transitividad (requisito § 2.2.6). ZODB tampoco ofrece ningún mecanismo de adaptabilidad. Se permite cambiar el sistema de almacenamiento pero no de forma dinámica (requisito § 2.2.4): al crear una conexión se indica el sistema de almacenamiento, que se mantendrá hasta su cierre. Si se desea cambiar el sistema de almacenamiento, habrá que crear una nueva conexión y volver a registrar todos los objetos.

APE permite utilizar un número arbitrario de sistemas y formatos de almacenamiento (requisito § 2.2.4). Además, gracias a su arquitectura modular basada en patrones de diseño facilita la adición de nuevos tipos de objetos y nuevos mecanismos de almacenamiento maximizando la reutilización. Por ejemplo, gracias al uso del patrón `Composite` [GOF94] sería posible utilizar diversos almacenes de datos simultáneamente tal y como se recoge en el requisito § 2.2.4. APE se construye sobre ZODB utilizando el mismo mecanismo para la detección de objetos modificados. Por ello, no aporta mayor transparencia que ZODB.

En cuanto a los requisitos de portabilidad, tanto ZODB como APE verifican la independencia del sistema operativo (§ 2.3.2) y del hardware (§ 2.3.3) gracias a la portabilidad de Python [Rossum2001], aunque no ofrecen independencia del lenguaje (requisito § 2.3.1).

3.8 Conclusiones

Los sistemas estudiados a lo largo de este capítulo ofrecen niveles muy diferentes de transparencia a la hora de dar respuesta a la persistencia de aplicaciones en los entornos comerciales de desarrollo software que existen en la actualidad. Aunque ninguno logra el requisito fundamental de separar completamente la incumbencia de la persistencia de la lógica de negocio (requisito § 2.1.1), los mejores resultados se han obtenido con los sistemas que transforman el código de la aplicación añadiendo a la lógica de negocio las rutinas necesarias para los mecanismos de persistencia. Esta transformación se realiza de una manera transparente al desarrollador, si bien los mecanismos utilizados son muy diferentes: generación de *proxies* dinámica (§ 3.2.6.1), herramientas de procesamiento a nivel de código intermedio (§ 3.2.6.2 y § 3.2.7.3), máquinas virtuales modificadas (§ 3.5.1), compiladores modificados (§ 3.5.2) y mecanismos específicos del lenguaje utilizado (§ 3.7).

Se ha visto cómo cada una de estas aproximaciones conlleva una serie de restricciones que impiden lograr los objetivos de transparencia recogidos en § 2.1. Por ejemplo, la generación dinámica de *proxies* únicamente permite interceptar métodos, mientras que la modificación a nivel de *bytecode*, además de violar el requisito § 2.1.5, impide tratar tipos de datos que no tienen asociada una clase, como los *arrays*.

Los mejores resultados en cuanto a transparencia y automatización se han obtenido con la modificación de la máquina virtual de Java (§ 3.5.1). El hecho de que la máquina virtual de soporte directamente el sistema de persistencia permite evitar diversos problemas derivados de la falta de ortogonalidad en el propio lenguaje [Jordan2004], pero también impide que el sistema de persistencia sea independiente del lenguaje (requisito § 2.3.1).

Únicamente los sistemas de persistencia ortogonal ofrecen una automatización integral de la persistencia (requisito § 2.1.4). El resto de sistemas obliga a realizar la acometida de transacciones de forma explícita. Además, en todos los sistemas que implementan persistencia por alcance, debe especificarse programáticamente cuáles son los objetos raíz. Ello implica que la forma de las aplicaciones persistentes difiere de la de las aplicaciones no persistentes, es decir, que la persistencia no es una incumbencia totalmente separada de la aplicación (requisito § 2.1.1). Del mismo modo, la operación de borrado de un objeto persistente, en aquellos sistemas que la ofrecen, debe ser invocada explícitamente en todos los casos.

Las carencias detectadas en los sistemas estudiados son mucho más acusadas en materia de adaptabilidad (§ 2.2). Se ha observado que ningún sistema ofrece ningún tipo de adaptabilidad dinámica real (requisito § 2.2.1). En determinados sistemas se ofrece la posibilidad de modificar una serie de parámetros, pero esta flexibilización se realiza siempre al configurar el sistema de persistencia, no durante su utilización, y además se restringe a un conjunto prefijado de parámetros.

Incluso centrándose en los componentes predeterminados que se pueden modificar en tiempo de configuración, se ha observado que son un conjunto muy reducido. En concreto, ninguno de los sistemas estudiados permite modificar las políticas de actualización de objetos (requisito § 2.2.5) ni tampoco los mecanismos de selección de los objetos persistentes (requisito § 2.2.6). Hibernate ofrece soluciones parciales a través de las estrategias de *fetching* prefijadas y el mecanismo de transitividad por cascada (§ 3.2.6.1).

Dentro de las facilidades de persistencia que ofrece Spring (3.6.2), debe destacarse el sistema de gestión declarativa de transacciones. Permite limpiar el código de las sentencias específicas de demarcación de transacciones favoreciendo de este modo la separación de la incumbencia de la persistencia (requisito § 2.1.1) y permitiendo controlar cómo se ejecutan las unidades transaccionales (requisito § 2.2.5).

De entre los sistemas estudiados en este capítulo, únicamente JDO (§ 3.2.6.2), EJB (§ 3.6.1) y APE (§ 3.7.2) permiten configurar diferentes sistemas de almacenamiento y formatos. Sin embargo, ninguno de ellos permite el intercambio de formato de almacenamiento en tiempo de ejecución sin tener que volver a almacenar explícitamente todos los objetos en el nuevo almacén (requisito § 2.2.4). Debe destacarse la utilización del patrón *Composite* (GOF94) por parte de APE de cara a permitir la utilización simultánea de varios almacenes tal y como se recoge en el requisito § 2.2.4).

Finalmente, la mayoría de los sistemas estudiados cumplen los requisitos de portabilidad tanto a nivel de hardware (§ 2.3.3) como de sistema operativo (§ 2.3.2) debido a que esta portabilidad está presente en las máquinas virtuales de las plataformas sobre las que funcionan. La independencia a nivel de lenguaje (§ 2.3.1), sin embargo, únicamente es conseguida por FastObjects .NET (3.2.7.3) y Zero (3.4.2). En el primer caso, el beneficio es debido a las características teóricas multilenguaje de la plataforma .NET [Thai2002]; en el segundo gracias a la utilización de un modelo computacional de objetos basado en prototipos, útil para representar cualquier modelo computacional que siga este paradigma [Borning86].

Capítulo 4

SISTEMAS COMPUTACIONALES ADAPTABLES NO REFLECTIVOS

En el capítulo anterior hemos observado cómo ninguno de los sistemas de persistencia analizados cumple el requisito fundamental de entender la persistencia como una incumbencia totalmente separada de la lógica de la aplicación. Además, evaluando los sistemas bajo la perspectiva de su adaptabilidad, los resultados han sido mucho peores. Por ello, en este capítulo analizaremos diversos trabajos existentes en el ámbito de los entornos computacionales de programación flexible con la excepción de una técnica, la reflectividad computacional, que será estudiada en el Capítulo 6. También se identificará la programación orientada a aspectos la cual, debido a su importancia en la actualidad, será motivo de análisis detallado en el capítulo siguiente.

La mayoría de los sistemas estudiados se fundamentan en un paradigma de diseño y programación denominado "separación de incumbencias" (*separation of concerns*) [Hürsch95], que se centra en separar la funcionalidad básica de una aplicación de otros aspectos especiales, como pueden ser la persistencia o distribución. El objetivo final es siempre separar la funcionalidad de una aplicación de sus distintas incumbencias, haciendo éstas variables y consiguiendo así la adaptabilidad global del sistema.

En cada caso estableceremos una descripción y estudio del sistema. Finalmente se analizarán sus puntos positivos aportados y sus carencias, en función de los requisitos definidos en el Capítulo 2.

Es de interés para el desarrollo de este capítulo definir qué se entiende por sistema adaptable y sistema adaptativo:

- Un sistema es adaptativo si es capaz de cambiar su comportamiento (él mismo) de acuerdo con los cambios que se produzcan en su entorno o dentro de él.
- Un sistema es adaptable si el usuario puede producir un nuevo comportamiento del sistema de una forma dinámica.

4.1 Principio de Separación de Incumbencias o Competencias

El principio de la separación de incumbencias (*Separation of Concerns*, SoC) [Hürsch95] representa un paradigma de diseño y programación que se centra en separar la funcionalidad básica de una aplicación de otros aspectos especiales, como pueden ser la persistencia o distribución.

Cuando se desarrolla un sistema, comúnmente se entremezclan en su codificación características tan dispares como su funcionalidad (aspecto principal) y aspectos adicionales como los relativos a su almacenamiento de datos, sincronización de procesos, distribución, restricciones de tiempo real, persistencia o tolerancia a fallos. El hecho de unir todas estas incumbencias en la propia aplicación conlleva:

- Elevada complejidad en el diseño y codificación de la aplicación.
- El programa, al unir todos sus aspectos en una dimensión, es difícil de mantener. Los cambios en una incumbencia implican el cambio global del sistema.
- El código de la aplicación es poco legible; al combinar todos los aspectos, la funcionalidad central de la aplicación no sobresale en un plano superior de abstracción.

La Ingeniería del Software ha utilizado el concepto de separación de incumbencias para gestionar la complejidad del desarrollo de software mediante la separación de las funcionalidades principales de la aplicación de otras partes con un propósito específico tales como la autenticación, administración, rendimiento, persistencia, *logging*, etc. La aplicación final se construye con el código de las funcionalidades principales más el código de propósito específico. Los principales beneficios de esta aproximación son:

- Un nivel de abstracción más alto, debido a que el desarrollador se puede centrar en incumbencias concretas y de forma aislada.
- Una mayor facilidad a la hora de entender la funcionalidad de la aplicación. El código que implementa ésta no está entremezclado con código de otras incumbencias.
- Una mayor reusabilidad al haber un menor acoplamiento.
- Una menor complejidad del código resultante.
- Una mayor flexibilidad en la integración de componentes.
- Un incremento de la productividad en el desarrollo.

La base de una buena separación de incumbencias es la capacidad de identificar y manejar de forma individual los distintos componentes del sistema tanto en la fase de diseño como en la codificación.

A lo largo de los años se han desarrollado aproximaciones y técnicas para conseguir SoC, que han ido evolucionando con el objetivo de hacer frente a las características cambiantes, ámbito y complejidad de las aplicaciones software. En términos generales estas propuestas ofrecen soluciones para la especificación, asociación y composición de incumbencias ortogonales a la funcionalidad básica a nivel de código. Estas aproximaciones están todavía en una etapa de propuesta o de ex-

perimentación. Adicionalmente ofrecen soporte limitado, o simplemente no lo ofrecen, a la definición, y asociación, dinámica de incumbencias.

4.2 Filtros de Composición

Los filtros de composición fueron diseñados para aumentar la adaptabilidad del paradigma de la orientación a objetos, añadiendo a éste extensiones modulares y ortogonales [Bergmans94]:

- La modularidad de las extensiones implica que los filtros de composición pueden ser añadidos a cada objeto, sin necesidad de modificar la definición de éste.
- La ortogonalidad de los filtros requiere que éstos tengan funcionalidades independientes entre sí.

Un ejemplo comparativo de adaptabilidad de objetos mediante filtros de composición puede ser la modificación del funcionamiento de una cámara de fotos. Si las condiciones de luz son pobres y el cuerpo a fotografiar se encuentra alejado, podremos modificar el funcionamiento de la cámara (objeto) añadiéndole filtros de color y lentes de aumento (filtros de composición); estas dos extensiones son modulares porque no es necesario modificar el funcionamiento de la cámara para acoplarlas, y son ortogonales porque sus funcionalidades son independientes entre sí.

En este modelo, 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. El objeto en este modelo se compone de dos partes: un núcleo u objeto interno y una capa de interfaz. El objeto interno se puede ver como un objeto normal (en el sentido tradicional), y la capa de interfaz envuelve al objeto interno y gestiona los mensajes entrantes y salientes (Figura 4.1).

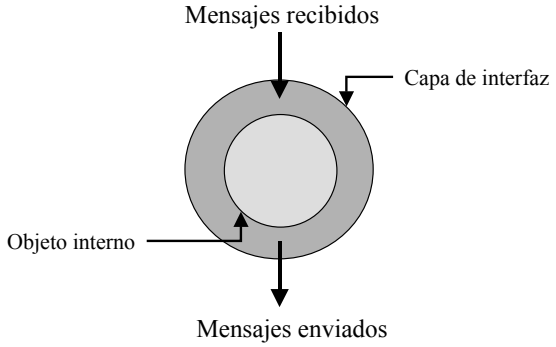


Figura 4.1. Objeto en el modelo de filtros de composición.

Los filtros son objetos instanciados de una clase filtro. El propósito de los filtros es el manejo y control del envío y recepción de mensajes. Especifican condiciones de aptitud o rechazo de los mensajes y determinan así la acción resultante. Cada filtro se puede programar en cada clase que lo utilice. El sistema se asegura de 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.

En términos generales, cuando un objeto envía un mensaje a otro objeto, el mensaje es filtrado por los filtros de salida del objeto emisor, y posteriormente por

los filtros de entrada del objeto receptor. Cada filtro decide qué hacer con el mensaje, aceptarlo o rechazarlo. Cuando se acepta el mensaje se invoca a un método, mientras que un mensaje rechazado pasa al siguiente filtro.

Los filtros de composición se han aplicado para adaptar la orientación a objetos a determinadas necesidades como delegación [Aksit88], transacciones atómicas [Aksit91], integración de acceso a bases de datos en un lenguaje de programación [Aksit92], coordinación de comportamiento entre objetos [Aksit93], restricciones de tiempo real [Aksit94] y especificaciones flexibles y reutilizables de 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 permitiendo la modificación de 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. Además pueden presentarse inconvenientes cuando los filtros no son ortogonales entre sí [Pryor2002].

El primer lenguaje de programación de filtros de composición fue "Sina" [Aksit88]. Posteriormente, los lenguajes de programación Smalltalk y C++ fueron extendidos para poder añadir filtros de composición a sus objetos sin necesidad de modificar éstos [Dijk95, Glandrup95].

4.3 Desarrollo de Software Orientado a Aspectos

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

La Programación Orientada a Aspectos (POA) (*Aspect Oriented Programming*) surge en 1997 [Kiczales97], y desde entonces ha suscitado mucho interés por las posibilidades que ofrece. En la POA hay dos conceptos muy importantes:

- Un componente (*component*) 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 (*aspect*) 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, la persistencia o la sincronización de hilos (*threads*).

Una aplicación orientada a aspectos es el resultado de adaptar (modificar o ampliar el funcionamiento) los componentes de una aplicación por medio de aspectos. Este proceso se denomina tejido (*weave*) y es realizado por el tejedor de aspectos (*aspect weaver*).

Como se muestra en la Figura 4.2, la generación de la aplicación final se produce en tres fases:

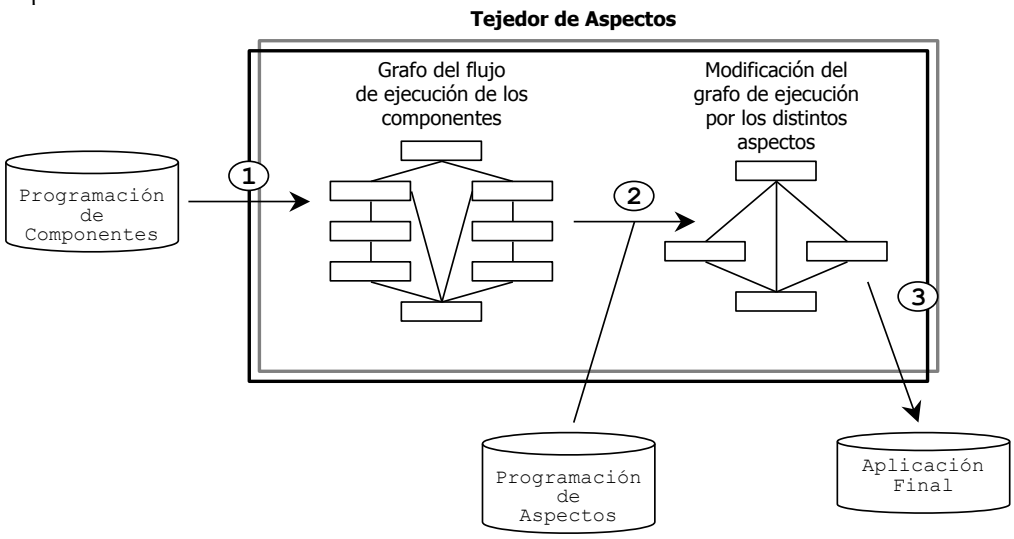


Figura 4.2. Fases en la generación de una aplicación orientada a aspectos.

1. El tejedor construye un grafo del flujo de ejecución del programa de componentes (esto lo puede hacer a partir del código fuente o de código ya compilado, depende del sistema).
2. A continuación se modifica el grafo anterior realizando las modificaciones o adiciones oportunas (en respuesta a los aspectos).
3. Por último se genera el código final de la aplicación (a partir del nuevo grafo generado, que incluye la funcionalidad de los componentes y de los aspectos).

En la actualidad se está realizando mucho trabajo de investigación sobre esta tecnología y han aparecido muchos sistemas que la soportan, siendo la referencia actual AspectJ [Kiczales2001]. Debido a su importancia, la programación orientada a aspectos y su aplicación de cara a la implementación de un sistema de persistencia será estudiada en el Capítulo 5.

4.4 Demeter AOP

Demeter AOP surge de aplicar la programación orientada a aspectos para implementar el método Demeter (*Demeter Method*) [Lieberherr96]. El método Demeter tiene sus fundamentos en la programación adaptable, una extensión de la programación orientada a objetos en la que se flexibiliza las relaciones entre la computación y los datos. La programación adaptable permite expresar la intención general de un programa sin necesidad de especificar todos los detalles de las estructuras de los objetos [Lieberherr96].

El software adaptable toma el paradigma de la orientación a objetos y lo amplía para que éste soporte un desarrollo evolutivo de aplicaciones. La naturaleza progresiva de la mayor parte de los sistemas informáticos, hace que éstos sufran un

número elevado de cambios a lo largo de su ciclo de vida. La adaptabilidad de un sistema guía la creación de éste a una técnica que minimice el impacto producido por los cambios. Software adaptable es aquél que se amolda automáticamente a los cambios contextuales (por ejemplo, la implementación de un método, la estructura de una clase, la sincronización de procesos o la migración de objetos).

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

Cada patrón de propagación de divide en las siguientes partes:

- Signatura de la operación o prototipo de la computación a implementar
- Especificación de un camino. 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 podrá ser formalizada en distintas aplicaciones finales mediante distintas personalizaciones (*customizations*). En cada personalización se especifica la estructura y comportamiento exacto de cada clase en un lenguaje de programación. Se ofrecen implementaciones del método en Java [Gosling96] y en el caso de Demeter, en C++ [Stroustrup98].

Por lo tanto en la programación adaptable se distinguen dos niveles. El primero es un alto nivel de abstracción que especifica clases y relaciones entre ellas. El segundo es el que, partiendo de esta especificación, implementa de forma refinada su comportamiento final mediante una personalización.

La gente involucrada en el grupo Demeter ha estado colaborando con el grupo de Gregor Kiczales, y fruto de esta colaboración surgió la idea de la Programación Orientada a Aspectos (POA) [Kiczales97]. En la actualidad el grupo Demeter está trabajando en la integración de la programación adaptable con la POA, habiendo desarrollado los sistemas DJ [Orleans2001] y Demeter AspectJ [DAJ], que integran ambas ideas.

4.5 Separación Multidimensional de Incumbencias

En la introducción de este capítulo identificábamos la separación de incumbencias como un nuevo paradigma de diseño y programación basado en identificar, encapsular y manipular las partes de un sistema relevantes a sus distintas competencias. La separación de estas competencias o incumbencias, evita la miscelánea de código concerniente a la obtención de distintos objetivos, facilitando el desarrollo de software y su mantenimiento.

El conjunto de incumbencias relevantes en un sistema varía a lo largo del ciclo de vida de éste, añadiendo mayor complejidad a la separación de las competencias. El caso más general es aquél en el que cualquier criterio para la descompo-

sición de incumbencias pueda ser aplicado. A modo de ejemplo, podemos identificar un sistema en el que se reconozcan tres dimensiones de competencias (mostrado en la Figura 4.3): primero, el estado y comportamiento de cada objeto, ubicados en su clase –modificar ésta implica modificar la funcionalidad y estructura de sus instancias; segundo, las características de cada objeto – capacidad de visualización, impresión, persistencia, etc.; tercero, reutilización de artefactos software –*middleware* (COM [Brown98], CORBA [OMG95], etc.), acceso a bases de datos, sistemas de seguridad, etc.

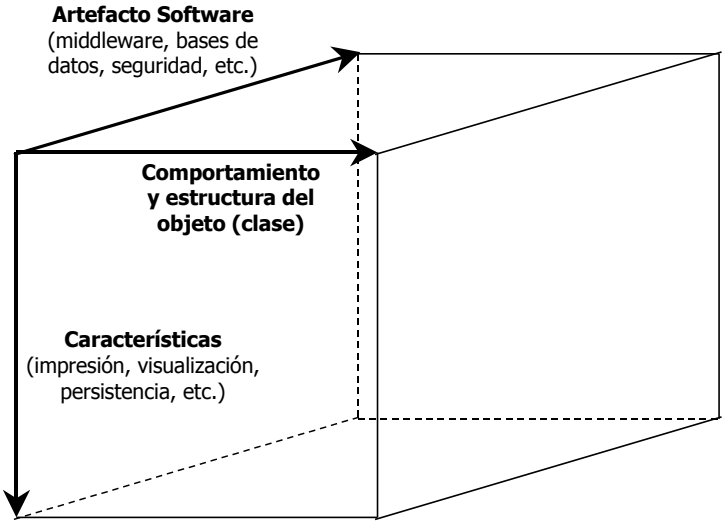


Figura 4.3. Ejemplo de un sistema con tres dimensiones de incumbencias.

El término "separación multidimensional de incumbencias" (*Multi-Dimensional Separation of Concerns*) se ha utilizado para designar a los sistemas capaces de proporcionar la separación incremental, modularización e integración de artefactos software, basados en cualquier número de incumbencias [IBM2000]. Los principales objetivos de estos sistemas son:

- Permitir encapsular todos los tipos de incumbencias de un sistema, de forma simultánea: El usuario podrá elegir y especificar cualquier dimensión de incumbencias del sistema, sin estar restringido a un número determinado ni a un tipo de competencias preestablecidas.

- Utilización de incumbencias solapadas e interactivas: En los sistemas estudiados previamente en este capítulo, las incumbencias separables de un sistema debían ser independientes y ortogonales –en la práctica esto no ocurre. La separación multidimensional de incumbencias permite que éstas se solapen y que la interacción entre ellas sea posible.

- Remodularización de incumbencias: El cambio o aumento de requisitos de un sistema puede implicar nuevas apariciones de incumbencias o modificaciones de las existentes. La creación o modificación de los módulos existentes deberá ser factible sin necesidad de modificar los módulos no implicados.

Las distintas ventajas de la utilización de este paradigma son:

- Promueve la reutilización de código; no sólo aspectos funcionales de computación, sino competencias adicionales (persistencia, distribución, etc.).

- Al separarse los distintos aspectos del sistema, se mejora la comprensión del código fuente.
- Reducción de los impactos en el cambio de requisitos del sistema.
- Facilita el mantenimiento de los sistemas gracias al encapsulamiento individual de las incumbencias identificadas.
- Mejora la trazabilidad de una aplicación centrándonos en aquel aspecto que deseemos controlar.

Existen diversos sistemas de investigación construidos basándose en este paradigma [OOPSLA99, ICSE2000]. Un ejemplo es *Hyperspaces* [Ossher99], un sistema de separación multidimensional de incumbencias independiente del lenguaje, creado por IBM, así como la herramienta Hyper/J [IBM2000b] que da soporte a *Hyperspaces* en el lenguaje de programación Java [Gosling96].

4.6 *Frameworks* Adaptables de Propósito Específico

4.6.1 RDM

RDM (*Rapid Domain Modeling*) [Izquierdo2002] es una arquitectura que intenta solucionar algunos problemas recurrentes de la programación orientada a objetos tradicional. Propone un marco donde los elementos que habitualmente se identifican al modelar un dominio son tratados como entidades independientes. Estos elementos serán:

- Entidades
- Relaciones
- Operaciones
- Reglas de negocio.

El desarrollo de RDM se fundamenta en la dificultad de trasladar estos elementos a código utilizando la programación orientada a objetos tradicional, donde el concepto de clase engloba a todos estos elementos, dificultando su reutilización cuando cambia el dominio. Realizar una separación de los mismos supone:

- Facilitar el mantenimiento de los programas: el programa se encuentra distribuido en elementos independientes entre sí.
- Facilitar la reutilización de elementos: al ser independientes, pueden ser utilizados en distintos dominios.

Lo que plantea RDM es gestionar estos elementos como objetos especializados e independientes entre si, lo que permita poder combinarlos de diferentes maneras bajo diferentes dominios. RDM proporciona un *framework* desarrollado en Java que demuestra la viabilidad de la arquitectura. El *framework* se define en base a interfaces y se suministran implementaciones por defecto de los mismos para que sea directamente utilizable [Gamma95]. A continuación se revisarán los principales elementos de la arquitectura RDM.

Una entidad RDM es la representación independiente y especializada de los datos que engloban los objetos de la programación orientada a objetos tradicional. Esto no significa que RDM proponga volver al concepto de estructuras o registros, las entidades podrán contener métodos, pero siempre que éstos afecten a los datos de la entidad o de entidades agregadas a esta. Lo que se pretende así, es que un objeto (o entidad) no englobe operaciones que afecten a otras entidades del dominio. A nivel del *framework* una entidad es una implementación del patrón *Variable State* [Beck96]. Se proporciona el interfaz `Entity` y su implementación por defecto en `StandardEntity`. Este patrón simplemente propone modelar los atributos de los objetos como una agregación de propiedades representadas por un par nombre-valor, con métodos para añadir, borrar y acceder a las propiedades dinámicamente. En la Figura 4.4 se muestra cómo se puede crear una entidad añadirle una propiedad dinámicamente.

```
Entity persona = new StandardEntity () ;
persona.add( "nombre" , "Pepe" ) ;
persona.get ( "nombre" ) ; //Pepe
```

Figura 4.4. Creación de una entidad RDM.

En RDM, las relacionas al igual que las entidades se corresponden con objetos independientes y especializados. Las relaciones podrán establecerse entre entidades y eliminarse entre cualquier par de entidades en tiempo de ejecución. Dos importantes características acerca de las entidades y relaciones son:

- Las entidades no conocerán las relaciones de las que pueden formar parte a priori.
- Las relaciones no sabrán nada sobre que entidades pueden relacionar.

Sin embargo se podrá acceder desde una entidad hasta las entidades con las que está relacionada. En la Figura 4.6 se muestra el código necesario para representar el modelo mostrado en la Figura 4.5.

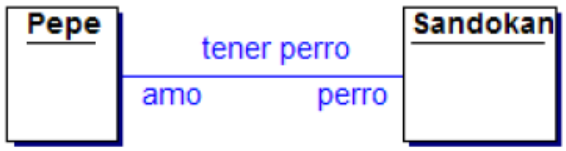


Figura 4.5. Dominio a modelar con relaciones RDM.

```
//Creación de la relación
Relation tenerPerro = new StandardRelation ("amo",
      "tener perro", "perro" , Cardinality.ONETOONE) ;

//Creación de las entidades a relacionar
Persona pepe = new Persona ("Pepe") ;
Perro sandokan = new Perro ("Sandokan") ;

//Dar de alta las entidades en la relación
pepe.add (tenerPerro, "amo" ) ;
sandokan.add(tenerPerro, "perro") ;

//A partir de aquí las entidades de la manera que
//se desee
pepe.set("perro", sandokan) ;
pepe.get("perro") ; //Sandokan
```

Figura 4.6. Establecimiento de relaciones con RDM.

Se observa cómo la manera de acceder a atributos de entidades o a entidades asociadas es igual con RDM. Es transparente de cara al usuario. Por otra parte, puede observarse cómo la relación queda establecida entre ambos extremos. La codificación de asociaciones en la programación orientada a objetos tradicional se muestra como una tarea rutinaria y tediosa (y por ello especialmente propensa a fallos). Por ello las asociaciones se suelen llevar a código como asociaciones unidireccionales.

RDM proporciona la gestión y sincronización automática. Esta sincronización se hace atendiendo a la cardinalidad de la asociación. En RDM puede ser:

- De uno a uno.
- De uno a muchos.
- De muchos a uno.
- De muchos a muchos.

La asociación de agregación representa una dependencia estructural entre el objeto que agrega y el agregado. Esto significa que si un objeto se clona, deberían clonarse los agregados. O si se borra, los agregados deberían ser eliminados también. Al igual que para las asociaciones normales, el *framework* RDM suministra una implementación del interfaz `Relation` que representa una agregación y que se tiene en cuenta en las dos operaciones comentadas:

- Borrado de entidades: en RDM las entidades pueden borrarse. Borrar una entidad desemboca el borrado de las entidades agregadas. Una entidad borrada no podrá volver a utilizarse en la aplicación.
- Clonado de entidades: Al obtener una copia de una entidad, se clonan a su vez todas las entidades agregadas.

Las reglas de negocio también se modelarán de manera independiente con RDM. Esto significa que estarán representadas por objetos especializados independientes de las entidades y relaciones.

Las reglas de negocio se implementarán aprovechando que los modelos construidos con RDM lanzan avisos cuando suceden cambios (patrón *Observer* [GOF94]):

- En las entidades.
- En las relaciones.

De este modo pueden definirse escuchadores que se subscriban al modelo y que definan las acciones a realizar cuando se produzcan cambios. Se utilizarán estos escuchadores para implementar las reglas de negocio. Los dos elementos identificados en la arquitectura que representan las reglas de negocio serán:

- Monitores que se utilizarán para representar restricciones sobre el dominio.
- Reacciones que se utilizarán para extender la funcionalidad del modelo.

Al igual que el resto de los elementos del *framework*, son elementos independientes de las entidades e independientes entre sí.

Los monitores implementan las restricciones sobre el modelo del siguiente modo: son notificados cuando se va a producir un cambio en el modelo y pueden rechazarlo lanzando una excepción, en cuyo caso el cambio no llegaría a producirse. RDM proporciona una serie de monitores por defecto:

- Impedir a una clase relacionarse consigo misma.
- Impedir modificar o borrar un miembro.
- Fijar las clases entre las que puede establecerse una relación.

Las reacciones se utilizan para extender la funcionalidad del modelo, permitiendo que éste evolucione descentralizadamente. Lo que hace una reacción cuando el modelo cambia, es ser notificada y realizar alguna acción. De este modo se producen reacciones en cadena de elementos independientes consiguiendo la funcionalidad deseada.

4.6.2 Adaptive Object-Models

Muchos sistemas de información orientados a objetos comparten un estilo de arquitectura que enfatiza la flexibilidad y adaptabilidad en tiempo de ejecución. Las reglas de negocio se almacenan de manera externa al programa en una base de datos o en ficheros XML [W3C98] en vez de en el código. El modelo de objetos que al usuario le afecta es parte de la base de datos, y el modelo de objetos del código es un intérprete de modelos de objetos de usuario. A estos sistemas se les denomina "Modelos de Objetos Adaptativos" (*Adaptive Object-Models*) [Yoder2001] ya que el modelo de objetos del usuario es interpretado en tiempo de ejecución y puede ser cambiado con efectos inmediatos en el sistema que lo interpreta.

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

Las arquitecturas que pueden adaptarse en tiempo de ejecución a los requisitos del usuario en ocasiones son denominadas "arquitecturas reflectivas" o "meta-arquitecturas". Los "Modelos de Objetos Adaptables" son un tipo de estas arquitecturas que también se las conoce por otros muchos nombres. Se las ha llamado "*Type Instance Pattern*" en [Gamma95]. También, dado que generalmente se utilizan para modelar sistemas que gestionan productos, se les denomina "*User Defined Product Arquitecture*" [Johnson98]. También es conocida como "*Active Object Models*" [Foote98] y como "*Dynamic Object Models*" [Riehle2000].

Un Modelo de Objetos Adaptativo es un sistema que representa clases, atributos y relaciones como metadatos. El sistema se modela basándose en instancias en vez de en clases. Los usuarios cambian los metadatos para reflejar los cambios del dominio. Estos cambios modifican el comportamiento del sistema. En otras palabras, el sistema almacena el modelo de objetos en la base de datos y lo interpreta. Por tanto, el modelo de objetos es activo, cuando éste se cambia el sistema cambia inmediatamente.

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

Un modelo de objetos adaptativo no modela las entidades como clases. En su lugar, modelan las descripciones de las clases (metadatos) que son interpretadas en tiempo de ejecución. Así, cuando es requerido un cambio del negocio, estas descripciones son cambiadas e inmediatamente reflejadas en la aplicación en ejecución.

Esta arquitectura normalmente se compone de varios patrones de diseño. El patrón *Type Object* [Johnson98b] separa las entidades de su tipo. Las entidades tienen atributos que son implementados con el patrón *Property* y de nuevo es vuelto a aplicar el patrón *Type Object* para separa a éstos de su tipo. Finalmente el patrón *Strategy* [GOF94] se ocupa del modelado de las operaciones.

4.6.2.1 Patrón Type Object

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

4.6.2.2 Patrón Property

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

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

4.6.2.3 Patrón Strategy

Una estrategia es un objeto que encapsula a un algoritmo [GOF94]. El patrón *Strategy* define un interfaz estándar para una familia de algoritmos para que los clientes puedan trabajar con cualquiera de ellos de la misma manera. Si el comportamiento de un objeto se define en función de una o más estrategias entonces su comportamiento puede ser modificado dinámicamente.

Cada aplicación del patrón *Strategy* produce una interfaz diferente y por lo tanto una jerarquía de clases distinta. Las estrategias pueden evolucionar para llegar a ser complicadas reglas de negocio que son construidas o interpretadas en tiempo

de ejecución. Estas pueden ser reglas primitivas o combinación de reglas mediante la aplicación del patrón *Composite* [GOF94].

El modelado de reglas complicadas junto con los interfaces de usuario dinámico usualmente es la parte más difícil de los modelos de objetos adaptativos y es la razón por la que no son implementables en un *framework* genérico [Yoder2001].

4.6.2.4 Relaciones

Los atributos son propiedades que usualmente se refieren a tipos primitivos. Estas asociaciones son usualmente unidireccionales. Las relaciones son propiedades que se refieren a otras entidades y son generalmente bidireccionales. Esto generalmente se modela mediante dos subclases de propiedades, una para los atributos y otra para las relaciones. Estas dos subclases se suelen modelar de tres formas [Yoder2001].

- La primera es volviendo a aplicar otra vez el patrón *Property* para las relaciones al igual que se aplicó para los atributos.
- La segunda manera es utilizar la misma construcción que se hizo para los atributos pero poniendo como clase padre a la clase `Property` y subdividiendo mediante herencia en `Attribute` y en `Association`. Una `Association` conocería su cardinalidad.
- La tercera forma sería discriminando por el valor de la propiedad. Si dicho valor es otra entidad se supone que es una relación.

4.6.2.5 Interfaces de Usuario para la Definición de Tipos

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

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

4.6.2.6 Arquitectura de los Modelos de Objetos Adaptativos

Los modelos de objetos adaptativos se construyen aplicando los patrones descritos en conjunción con otros patrones de diseño como *Composite*, *Interpreter* y *Builder* [GOF94]. El patrón *Composite* se usa para construir el árbol dinámico de la estructura de tipos o reglas [Yoder2001]. Los patrones *Builder* e *Interpreter* se utilizan habitualmente para reconstruir las estructuras a partir del metamodelo o para interpretar los resultados.

No obstante los artefactos descritos son patrones, no un *framework* para construir modelos de objetos adaptativos. Cada modelo de objetos adaptativo que se construya constituirá un *framework* en sí mismo, pero aún no se ha implementado un *framework* genérico para construir estos modelos. En [Yoder2001] se señala que, si bien su implementación sería factible, el trabajo realmente duro está relacionado

con el modelado de las reglas de negocio, las cuales son específicas del dominio y varían con mayor frecuencia.

4.7 Conclusiones

En este capítulo hemos visto cómo la programación adaptable puede ser utilizada en distintos campos, y cómo se han desarrollado distintas técnicas para conseguirla. A continuación analizaremos las principales aportaciones y carencias de los sistemas estudiados bajo la perspectiva de su aplicación para desarrollar un sistema de persistencia que verifique los requisitos planteados en el Capítulo 3.

Con la excepción de los *frameworks* de propósito específico, los sistemas estudiados han sido desarrollados buscando un entorno en el que la parte funcional de la aplicación se pudiese separar del resto de incumbencias.

Los filtros de composición permiten modificar el mecanismo computacional base en la orientación a objetos: el paso de mensajes. Con esta técnica se permite modificar el comportamiento de un grupo de objetos (los instanciados de una determinada clase). La flexibilidad de las aplicaciones se consigue especificando su funcionalidad básica mediante un lenguaje orientado a objetos, y posteriormente ampliando ésta mediante la programación de filtros de composición. Sin embargo, para que un objeto pueda utilizar un filtro de composición, deberá especificarlo a priori (en tiempo de compilación) en la interfaz de su clase; esto limita la flexibilidad dinámica del sistema y lo invalida de cara a permitir la adaptabilidad dinámica de los parámetros del sistema de persistencia (§ 2.2).

La programación adaptable, en la que se fundamenta Demeter AOP, más que buscar entornos de computación flexible, está enfocada a minimizar el impacto en los cambios producidos en entornos de desarrollo evolutivos. Los programas adaptables tratan de representar de algún modo la funcionalidad captada en tiempo de análisis y diseño. Es por esta razón por la que este tipo de sistemas tampoco ofrecen la flexibilidad necesaria para cumplir los requisitos de adaptabilidad del sistema de persistencia dados en § 2.2.

La principal aportación de los sistemas basados en separación multidimensional de incumbencias, en comparación con los estudiados en este capítulo, es el grado de flexibilidad otorgado: mientras que el resto de sistemas identifican un conjunto limitado de aspectos a describir, en este caso no existen límites.

Por otro lado, si bien el grado de flexibilidad aumenta al aumentar la posibilidad de identificar más incumbencias, el mecanismo para obtener dicha flexibilidad no se modifica. Dicho mecanismo sigue los pasos de descripción de competencias y procesamiento de éstas para generar el sistema final. Aunque se llega a un compromiso de flexibilidad para obtener eficiencia de la aplicación final, éste está lejos de ser el mecanismo flexible dinámico (accesible en tiempo de ejecución) que permita dar respuesta a los requisitos recogidos en § 2.2.

A diferencia de los sistemas anteriores, la arquitectura RDM no fundamenta su adaptabilidad en la separación de incumbencias sino en el tratamiento especializado de los elementos del dominio más habituales. El mantener estos elementos como entidades independientes facilita afrontar los cambios en el dominio, cuyo impacto estará localizado sobre abstracciones de mayor nivel que el tradicional concepto de clase en orientación a objetos. RDM ofrece un grado de adaptabilidad limitado de cara al sistema de persistencia que se quiere desarrollar, puesto que los

cambios que permite afrontar están restringidos a los que afectan a los elementos de dominio más habituales (cambios en las relaciones, modificación de atributos, borrado de entidades, etc.). Sin embargo, en § 2.2 se identificaba como un requisito fundamental del sistema que cualquier parámetro del sistema fuese parametrizable dinámicamente, lo que implica que las operaciones deben poder ser modificadas dinámicamente. RDM identifica las operaciones como un elemento de dominio pero no propone ningún sistema para modelarlas dinámicamente.

RDM podría utilizarse para modelar los dominios de objetos persistentes de la aplicación desarrollando una implementación del *framework* que ofrezca persistencia transparente, pero este enfoque presenta tres graves limitaciones. En primer lugar, la implementación actual del *framework* para Java se presenta altamente intrusiva, teniendo que modificar el código de las clases de los objetos de dominio sustancialmente (violación del requisito § 2.1.1). En segundo lugar, si los mecanismos de persistencia transparente descansan sobre la implementación del *framework* no podría ofrecerse persistencia a objetos que no fuesen entidades RDM, lo que supone una violación grave de los requisitos de transparencia (§ 2.1).

En cuanto a los modelos de objetos adaptativos, al igual que RDM su adaptabilidad también está enfocada a cambios en el dominio del modelo de objetos. La arquitectura permite modelar las reglas de negocio mediante el patrón *Strategy* [GOF94], siendo los objetos estrategia los metadatos que representan las reglas de negocio. Al margen de la dificultad de modelar reglas complejas utilizando este enfoque [Yoder2001], estas estrategias serán definidas habitualmente en tiempo de diseño, con lo que no sería posible verificar los requisitos de adaptabilidad dinámica recogidos en § 2.2. Si se desease que estas reglas pudiesen ser definidas en tiempo de ejecución, debería utilizarse un lenguaje de reglas. Sin embargo, la elección del lenguaje utilizado para introducir reglas de negocio en el sistema se muestra como una de las mayores dificultades a la hora de implementar un modelo de objetos adaptativo [Foote98].

Capítulo 5

DESARROLLO DE SOFTWARE ORIENTADO A ASPECTOS

En el capítulo anterior hemos estudiado diversos sistemas que ofrecían diferentes aproximaciones para implementar el principio de separación de incumbencias. También se definió la programación orientada a aspectos. En este capítulo analizaremos con mayor profundidad el desarrollo de software orientado a aspectos, una prometedora disciplina actualmente en auge basada en aplicar la separación de incumbencias en todos los puntos del ciclo de vida del desarrollo de software.

En primer lugar realizaremos una introducción a esta disciplina, explicaremos sus fundamentos y se proporcionarán una serie de definiciones de sus conceptos fundamentales. A continuación estudiaremos diversos sistemas desarrollados en el ámbito de la programación orientada a aspectos. Se clasificarán utilizando el criterio más habitual: la posibilidad de tejer los aspectos estática o dinámicamente.

Finalmente analizaremos las principales aportaciones y carencias la programación orientada a aspectos, evaluada a través de los sistemas analizados, en función de los requisitos definidos en el Capítulo 2.

5.1 Introducción

El término Programación Orientada a Aspectos (POA) es propuesto por Gregor Kiczales [Kiczales97] en 1997, aunque el equipo Demeter había estado trabajando sobre este campo desde mucho antes. La primera definición temprana de "aspecto" también la dio este grupo en 1995 [Demeter96].

La Programación Orientada a Aspectos (POA) dota al sistema de la capacidad de modularizar aspectos que se diseminan a través de la funcionalidad básica. Los aspectos expresan funcionalidad que afecta al sistema de una manera ortogonal a la funcionalidad básica y permiten al desarrollador diseñar el sistema básico sin la necesidad de incluir los aspectos ortogonales, dotándole además de un punto único para realizar modificaciones. La separación del código que se encarga de la funcionalidad básica del código de los aspectos ortogonales permite que el código resultante no esté entremezclado, siendo más fácil de depurar, mantener y modificar

[Parnas72]. Es decir, se pretende poder implementar un sistema de forma eficiente y fácil, lo que redundará en una mejor calidad del software.

Puesto que la POA es una aproximación de la Separación de Incumbencias (SoC), antes de definir formalmente el concepto de aspecto conviene señalar qué se entiende por incumbencia (o competencia) (*concern*).

"Incumbencia es todo aquello que incumbe al software" [Tarr99]. Básicamente incumbencia es todo lo que sea importante para la aplicación, ya sea código, infraestructura, requerimientos, elementos de diseño, etc.

En la POA existen dos conceptos fundamentales [Kiczales97]:

- Un componente (*component*) 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 (*aspect*) 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 a la ejecución o semántica de los componentes. Ejemplos de aspectos son la persistencia o la sincronización de hilos (*threads*).

Un aspecto es una clase particular de incumbencia. La definición formal más aceptada es: "Un aspecto es una unidad modular que se disemina por la estructura de otras unidades funcionales. Los aspectos existen tanto en la etapa de diseño como en la de implementación. Un aspecto de diseño es una unidad modular del diseño que se entremezcla en la estructura de otras partes del diseño. Un aspecto de programa o de código es una unidad modular del programa que aparece en otras unidades modulares del programa" [Kiczales97].

De una manera más informal se puede decir que los aspectos son elementos que se diseminan por todo el código y son difíciles de describir con respecto a otros componentes.

En la Figura 5.1 se pueden ver los requerimientos de un sistema como un haz de luz que pasa a través de un prisma el cual la descompone en las distintas incumbencias. Por un lado está la lógica de negocio, y por otro una serie de incumbencias que se diseminan por todo el código. El conjunto de todas ellas es lo que forma el sistema [Laddad2002].

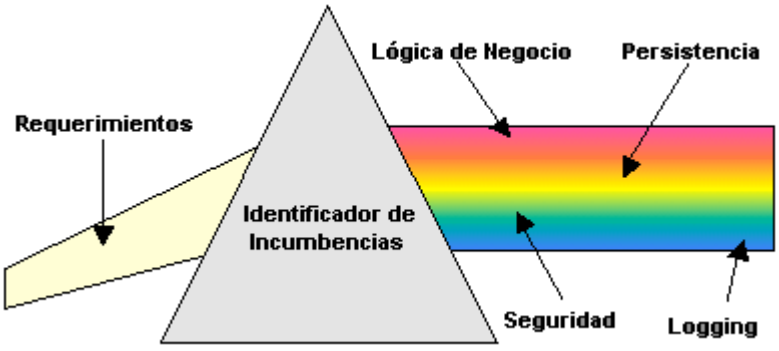


Figura 5.1. Requerimientos del sistema según POA.

La estructura de un programa orientado a aspectos se muestra en la Figura 5.2. Se puede ver que el programa es una combinación de distintos módulos. Unos contienen la funcionalidad básica (modelo de objetos), mientras que los demás recogen otro tipo de características como son la seguridad, persistencia, *logging* (registro), gestión de memoria, etc.

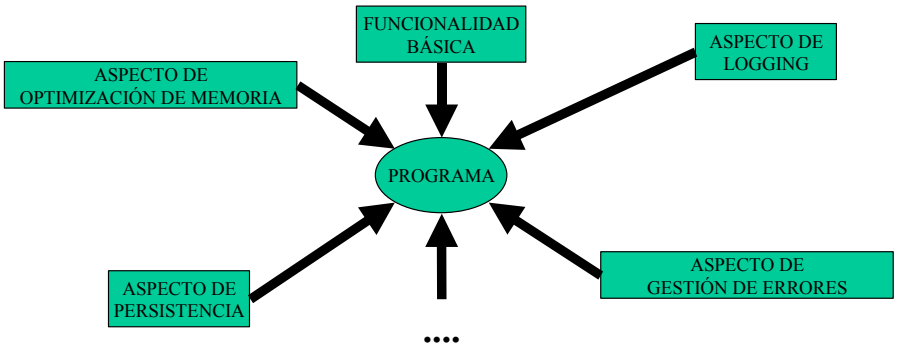


Figura 5.2. Estructura de un programa orientado a aspectos.

En la parte izquierda de la Figura 5.3 se puede ver la forma que tiene el código de un programa siguiendo una metodología tradicional. Se puede observar que el código está entremezclado siendo difícil de entender, depurar y modificar. En cambio siguiendo la POA, mostrada en la parte derecha de la figura, las diferentes incumbencias están separadas con lo que son más fáciles de entender y, por lo tanto, de trabajar con ellas.

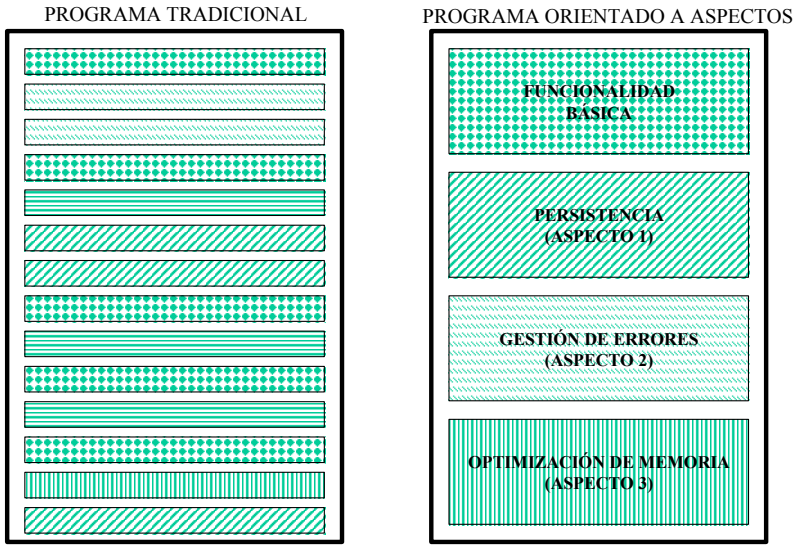


Figura 5.3. Código programa tradicional vs. POA.

Básicamente en el funcionamiento de la POA se pueden identificar tres pasos (Figura 5.4):

- **Descomposición en aspectos**: se descomponen los requerimientos con el fin de identificar las distintas incumbencias (las comunes y aquéllas que se entremezclan con el resto).

- **Implementación de incumbencias**: se implementan de forma independiente las incumbencias detectadas anteriormente, tanto la funcionalidad básica como los aspectos ortogonales.

- **Recomposición de aspectos**: este proceso, denominado tejido (*weaving*), toma todos los módulos implementados anteriormente (funcionalidad básica y aspectos) y los teje para formar el sistema completo. Es realizado por el tejedor de aspectos (*aspect weaver).*

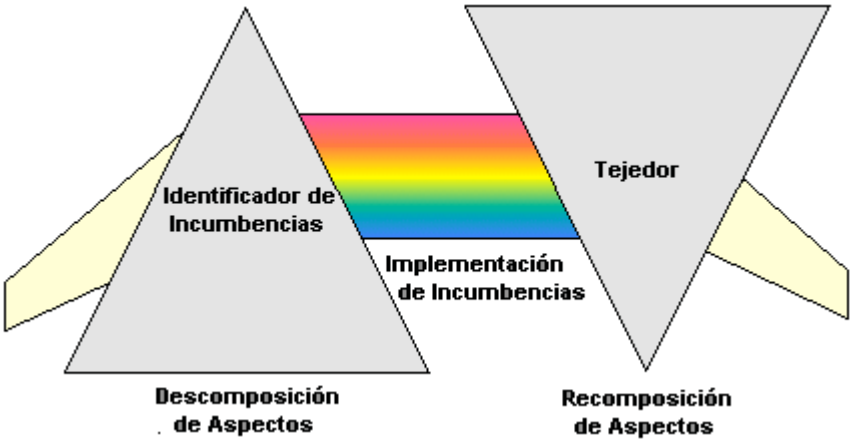


Figura 5.4. Pasos básicos en la POA.

Los lenguajes orientados a aspectos definen una nueva unidad de programación de software, el aspecto, para encapsular las funcionalidades que están diseminadas (*crosscut*) y enmarañadas (*tangled*) por todo el código. A la hora de formar el sistema se ve que hay una relación entre los componentes y los aspectos, y que, por lo tanto, el código de los componentes y de estas nuevas unidades de programación tiene que interactuar de alguna manera. Para que los aspectos y los componentes se puedan mezclar, deben tener fijados los puntos donde puedan hacerlo, que son lo que se conoce como puntos de enlace (*joinpoint*).

Los puntos de enlace son un interfaz entre los aspectos y los módulos de componentes que define en qué lugares se pueden aumentar el comportamiento de los módulos con el comportamiento de los aspectos.

El proceso de realizar esta unión se conoce como tejido (*weave*), y el encargado de realizarlo es el tejedor de aspectos (*aspect weaver*).

El tejedor, además de los aspectos, los módulos y los puntos de enlace, recibe unas reglas que le indican cómo debe realizar el tejido. Estas reglas son los llamados puntos de corte (*pointcuts*), que indican al tejedor qué aspecto tiene que aumentar a qué módulo a través de qué punto de enlace.

El código de los aspectos normalmente recibe el nombre de *advice* al ser el término utilizado en el sistema AspectJ [AspectJ] y que han adoptado la mayoría de sistemas posteriores.

Como se puede ver en la Figura 5.5, en las metodologías tradicionales el proceso de generar un programa consistía en pasar el código a través de un compilador o un intérprete para así disponer de un ejecutable. En la POA no se tiene un único código del programa sino que está separado el código que implementa la funcionalidad básica y el código que implementa cada uno de los aspectos. Todo este código debe pasar no sólo a través del compilador, sino que debe ser tratado por el

tejedor, que es el que se encarga de crear un único programa con toda la funcionalidad, la básica más los aspectos

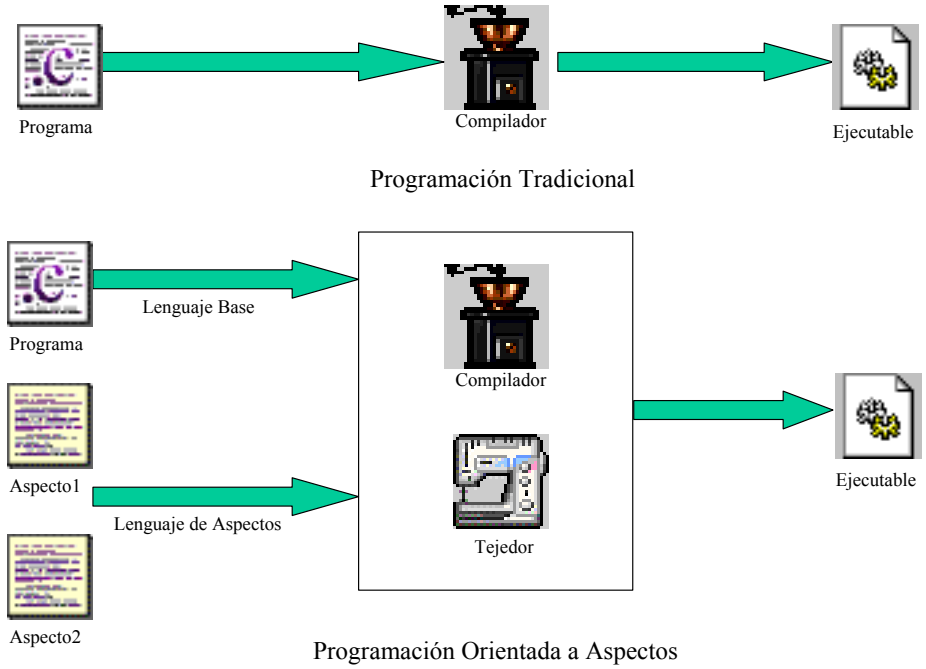


Figura 5.5. Implementación tradicional vs. POA.

Conviene revisar la relación existente entre la programación orientada a aspectos y la programación orientada a objetos. La programación orientada a objetos (POO) es la metodología más utilizada en la actualidad en la mayoría de los nuevos proyectos de desarrollo de software. De hecho la POO ha demostrado su fuerza a la hora de modelar la funcionalidad básica dominante del sistema (normalmente la funcionalidad original para la que surge la aplicación). Sin embargo, no es capaz de tratar de forma adecuada aspectos como la Persistencia, que no forman parte de la funcionalidad básica o dominante del sistema, encontrándose éstos diseminados por muchos módulos, a menudo sin relación entre ellos, con el consiguiente problema de pérdida de claridad en el código (lo que afecta negativamente a la calidad del software). Se puede afirmar por lo tanto que las técnicas tradicionales no gestionan bien la "separación de incumbencias" para aspectos que no forman parte de la funcionalidad básica del sistema [Tarr99].

La POA no pretende desbancar a la POO, al contrario, simplemente va un paso más allá al entender que no es suficiente con lo que hay. Al igual que dentro de la POO tiene cabida la descomposición funcional, dentro de la POA tiene cabida la POO. La POA no es una extensión de la POO ya que se puede utilizar con otros paradigmas existentes, como la programación estructurada.

5.1.1 Definiciones

A continuación se definen diversos términos habitualmente utilizados en el ámbito de la programación orientada a aspectos.

- **Aspecto**: es aquel módulo software que no puede ser encapsulado en un procedimiento. Los aspectos no son unidades funcionales en las que se pueda dividir un sistema, sino propiedades que afectan a la ejecución o

semántica de los componentes. Ejemplos de aspectos son la gestión de memoria y la sincronización de hilos.

- **Punto de enlace** (*joinpoint*): es un punto de la ejecución de un programa bien definido donde se podrá añadir código (funcionalidad). No todos los puntos de ejecución de un programa son puntos de enlace, sólo aquéllos que puedan ser tratados de forma controlada. Cada sistema que soporta POA ofrece una serie de puntos de enlace distintos. Ejemplos de puntos de enlace pueden ser invocaciones a métodos, acceso a campos, etc. Por el contrario, la ejecución de la línea 33 de la clase C, no es un punto de enlace.

- **Punto de corte** (*pointcut*): es un conjuntos de instrucciones que se le pasan al tejedor con el fin de que sepa qué código (*advice* del aspecto) se le debe añadir en qué punto de enlace a una aplicación. Un ejemplo podría ser invocar el método X del aspecto cuando se produzca un acceso de lectura al campo Y de la clase C.

- *Advice*: es el código del aspecto que se ejecuta en los puntos de enlace seleccionados por un punto de corte. Normalmente desde su código se puede acceder al contexto de ejecución del punto de enlace (se puede acceder a los valores de variables, etc.).

- **Tejedor** (*weaver*). Es el proceso encargado de realizar la ampliación del comportamiento de los módulos de componentes con el comportamiento de los aspectos. Dependiendo de en qué momento se realice el tejido puede hablarse de tejido estático o tejido dinámico.

- **Tejido estático**. El proceso de tejido de los aspectos se realiza en tiempo de compilación.

- **Tejido dinámico**. El proceso de tejido de los aspectos se realiza en tiempo de ejecución.

- **AOSD** (*Aspect Oriented Software Development*): Desarrollo de Software Orientado a Aspectos, es el término usado actualmente para referirse al uso de técnicas orientadas a aspectos a lo largo del ciclo de vida del software. Incluye todo el proceso de desarrollo del software (análisis, diseño, implementación, etc.).

5.2 Separación Estática de Aspectos

La mayoría de las actuales implementaciones de POA están basadas en el tejido estático. El tejido estático consiste en la modificación en tiempo de compilación del código de la aplicación, insertando llamadas a las rutinas específicas de los aspectos. Los lugares donde estas llamadas se pueden insertar son los puntos de enlace (*joinpoints*), definidos por cada sistema.

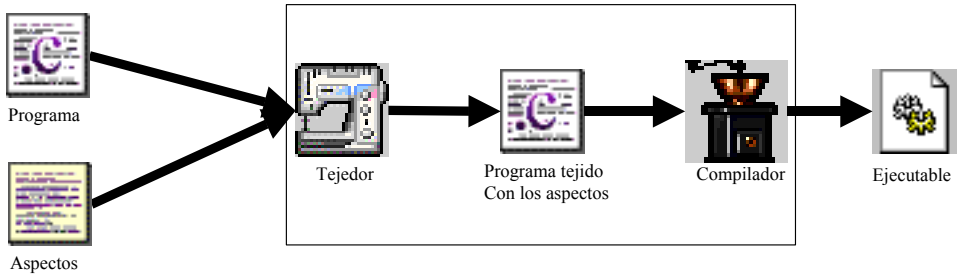


Figura 5.6. Tejido estático.

En la Figura 5.6 se puede ver el proceso que se sigue en el tejido estático. Se parte del programa con la funcionalidad básica implementada en un lenguaje base (C++, Java, etc.) y además se tiene uno o varios aspectos, escritos en un lenguaje de aspectos (que puede ser una extensión de algún lenguaje normal, o uno específicamente definido con este fin). Por aspecto se entiende el código que se debe añadir y las instrucciones que recibe el tejedor sobre dónde y cómo insertar el código de los aspectos en el código principal (estas instrucciones pueden venir expresadas en el mismo lenguaje o en otro particular, y pueden encontrarse en el mismo fichero o en otro). El tejedor (*weaver*) realiza la composición de código, insertando el proveniente de los aspectos en el código de la funcionalidad básica siguiendo las instrucciones recibidas (del tipo "insertar llamada a método X antes de la invocación al método Y" o "invocar al método Z después de acceder al campo K de la clase C"). El resultado es un nuevo código fuente, de estilo tradicional, que pasará a través de un compilador el cual generará el programa ejecutable. Hay herramientas que realizan el proceso de tejido de forma interna en un único paso desde el punto de vista del usuario.

El tejido estático presenta algunos inconvenientes como la imposibilidad de adaptarse a cambios en el entorno de ejecución debido a que la aplicación final es generada tejiendo la funcionalidad básica con los aspectos en tiempo de compilación; cualquier funcionalidad que necesite ser adaptada en tiempo de ejecución implica que se debe parar la ejecución de la aplicación, ésta se debe recompilar con los nuevos aspectos (es decir pasar a través del tejedor y del compilador), y debe ser iniciada de nuevo. Ésta es la razón por la que el uso del tejido estático no es factible en aplicaciones que no puedan detenerse y que necesiten ser adaptadas.

Al mismo tiempo, la depuración de una aplicación que haya sido compilada y se le hayan añadido ya los aspectos es una tarea compleja porque el código que se debe analizar es el resultante del proceso de tejido: el código original está entremezclado con el de los aspectos, el cual estará diseminado por toda la aplicación.

La principal ventaja que ofrecen estos sistemas es que se evita que el uso de la POA derive en una penalización en el rendimiento, ya que antes de la compilación se dispone de la totalidad del código pudiendo ser optimizado por el compilador.

Una desventaja muy importante que suelen presentar las herramientas que ofrecen tejido estático (y las que ofrecen tejido dinámico) es que son dependientes del lenguaje; esto implica que el sistema sólo sirve para un lenguaje específico, no pudiendo crear distintos aspectos en distintos lenguajes.

A continuación se va a analizar la herramienta AspectJ [AspectJ]. Los orígenes de esta herramienta han estado unidos a los de la misma programación orienta-

da a aspectos y por ello se ha convertido en un estándar *de facto* tomado como referencia por el resto de herramientas que trabajan con aspectos. Por ello, será la única herramienta analizada dentro de la categoría de la programación orientada a aspectos estática.

5.2.1 AspectJ

AspectJ [AspectJ] fue el primer sistema comercial que ofreció POA y es el más extendido en la actualidad, siendo un modelo a seguir y de comparación para el resto de sistemas que pretenden ofrecer POA. No es casualidad que haya sido desarrollado inicialmente en Xerox PARC [PARC] por el mismo grupo de personas que propusieron el concepto de programación orientada a aspectos [Kiczales97].

AspectJ es una especificación de lenguaje y también una implementación de lenguaje que soporta POA. La especificación del lenguaje define varias construcciones y su semántica para soportar los conceptos de la orientación a aspectos. La implementación del lenguaje ofrece herramientas para compilar, depurar y documentar código [Laddad2002].

Las construcciones de AspectJ extienden el lenguaje Java [Gosling96], por lo que cualquier programa válido escrito en Java es también válido en AspectJ (no ocurriendo así a la inversa).

El compilador de AspectJ genera clases Java puras (conformes al estándar), por lo que pueden ser ejecutadas en cualquier máquina virtual de Java (JVM).

Entre las herramientas que proporciona AspectJ se encuentra un compilador (que es el tejedor de aspectos), un depurador y un generador de documentación. Además cuenta con un navegador de aspectos.

Los conceptos que AspectJ ha añadido a Java son:

- *Joinpoints* (puntos de enlace).
- *Pointcut designators* o *Pointcuts* (puntos de corte).
- *Advice* (código del aspecto).

Las definiciones de estos conceptos se han visto anteriormente en § 5.1.1. Estos conceptos han sido adoptados por la mayoría de los sistemas que implementan orientación a aspectos.

La forma de trabajar de AspectJ se puede ver en la Figura 5.7. De forma separada se programan los componentes (que contendrán la funcionalidad básica de la aplicación) en Java estándar, y por otra parte se programan los aspectos (entendiéndose como tal el conjunto de *pointcuts* y *advice*) mediante las extensiones de AspectJ, todo esto (aspectos y componentes) se procesa por parte del compilador de AspectJ (`ajc`, *AspectJ compiler*) [O'Brien2001] el cual genera los ficheros de clases (estándar) que son los que se ejecutarán por parte de la JVM.

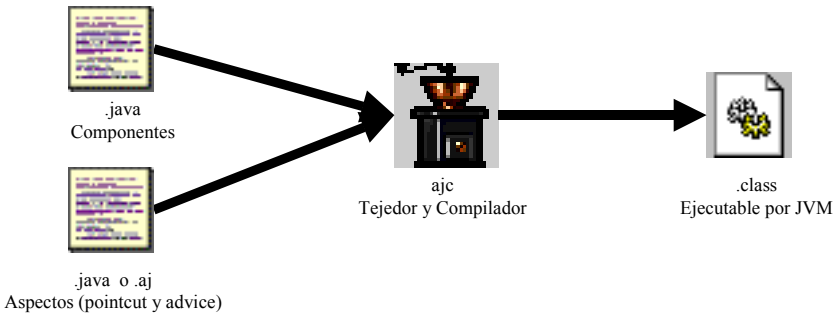


Figura 5.7. Proceso de compilación en AspectJ.

También se puede usar el compilador estándar de Java, actuando en este caso el compilador de AspectJ como un mero precompilador que genera nuevo código Java que es el que se le pasa al compilador estándar para que sea este el que genere el código ejecutable (en un principio era así como funcionaba).

En el caso de que se necesite añadir un nuevo aspecto, modificar alguno existente, o eliminarlo, es necesario volver a compilar la aplicación. Es decir, es un sistema que soporta orientación a aspectos de forma estática.

Los aspectos codificados contienen uno o varios puntos de corte, cuya función es seleccionar puntos de enlace en el componente, y *advice* que lo que hacen es asignar código a los puntos de corte (o combinaciones de ellos), de tal forma que cuando se verifica un punto de corte se ejecute el código asignado por el *advice*.

5.2.1.1 Puntos de Enlace

Al ser el primer sistema que ofreció orientación a aspectos y por ser actualmente el patrón a seguir por el resto de sistemas tiene mucha importancia el conjunto de puntos de enlace que soporta, ya que los demás sistemas intentan soportar los mismos o un subconjunto de ellos, por lo que ha pasado a ser el patrón de comparación.

Los puntos de enlace que soporta son:

- Invocación a un método.
- Invocación a un constructor.
- Ejecución de un método.
- Ejecución de un constructor.
- Inicialización de objetos que se crean con el constructor.
- Preinicialización de atributos (asignación en la declaración) realizada antes de la invocación al constructor del padre (`super`).
- Inicialización de bloque estático.
- Acceso de lectura a campo.
- Acceso de escritura a campo.

- Cuando una excepción `IOException` (o un subtipo de la misma) se trata en un bloque `catch`.
- Ejecución de cualquiera de los *advice* inyectados.

Estos son los puntos de enlace donde se puede añadir código en un componente en AspectJ. Los puntos de enlace son seleccionados mediante los puntos de corte que se muestran a continuación.

5.2.1.2 Puntos de Corte

Los puntos de corte definen una agrupación de puntos de enlace a capturar. Se expresan mediante las primitivas de los puntos de enlace y, parametrizándolas, mediante tipos, modificadores (`public`, `static`,...), expresiones regulares (operador `*`) y operadores de consulta (`||` o `&&`). A continuación se muestran los puntos de corte existentes:

- `call`, invocación a un método.
- `call`, invocación a un constructor.
- `execution`, ejecución de un método.
- `execution`, ejecución de un constructor.
- `initialization`, inicialización de objetos que se crean con el constructor.
- `preinitialization`, preinicialización de atributos (asignación en la declaración) realizada antes de la invocación al constructor del padre (`super`).
- `staticinitialization`, inicialización de bloque estático.
- `get`, acceso de lectura a campo.
- `set`, acceso de escritura a campo.
- `handler`, cuando una excepción `IOException` (o un subtipo de la misma) se trata en un bloque `catch`.
- `adviceexecution`, ejecución de cualquiera de los *advice* inyectados.

Hasta aquí se han visto puntos de corte que se corresponden con puntos de enlace de forma directa, los puntos de corte que se muestran a continuación sirven para, combinándolos con los anteriores mediante operadores, poder seleccionar puntos de enlace de una forma más potente:

- `within`, selecciona cualquier punto de enlace donde el código asociado se define dentro de (una clase, un paquete, etc.).
- `withincode`, selecciona cualquier punto de enlace donde el código asociado se define dentro de un método.
- `withincode`, selecciona cualquier punto de enlace donde el código asociado se define dentro del constructor.

- `cflow`, selecciona cualquier punto de enlace que se produce en el flujo de control de una llamada a un método (incluye la llamada).
- `cflowbelow`, selecciona cualquier punto de enlace que se produce a debajo del flujo de control de una llamada a un método (no incluye la llamada).
- `if`, selecciona cualquier punto de enlace donde se cumpla una condición (dada como parámetro).
- `this`, selecciona cualquier punto de enlace en el que el objeto que se está ejecutando es una instancia de una clase determinada (como parámetro).
- `target`, selecciona cualquier punto de enlace en el que el objeto destino es una instancia de una clase determinada (como parámetro).
- `args`, selecciona cualquier punto de enlace donde los argumentos cumplan ciertas reglas (dadas como parámetros) como pueden ser su tipo, su número o su orden.

A continuación se muestran algunos ejemplos de puntos de corte muy sencillos:

- `target(Point) && call(int *())`, selecciona cualquier método que devuelve un `int`, que no tiene parámetros y que se realiza en un objeto que es una instancia de la clase `Point`.
- `within(*) && execution(*.new(int))`, selecciona la ejecución de cualquier constructor que recibe exactamente un parámetro de tipo `int` sin que importe desde donde se realiza la llamada.
- `execution(!static * *(..))`, selecciona cualquier ejecución de cualquier método no estático, con cualquier número de parámetros.

Los puntos de corte se definen y son posteriormente usados en la declaración de los *advice*.

5.2.1.3 Advice

En AspectJ a la hora de declarar un *advice*, además del código que se va a añadir, se selecciona la composición de puntos de corte en los que se va a insertar el código, y además se indica si va a ejecutarse antes (*before*), después (*alter*) o en vez de (*around*) el punto de corte alcanzado.

- `before`, el código se ejecuta antes ejecutar el punto de enlace (una invocación a un método, un acceso a un campo, etc.).
- `after`, el código se ejecuta después.
- `after returning`, el código se ejecuta después si se devuelve lo que se haya declarado.
- `after throwing`, el código se ejecuta después en caso de que se haya producido una excepción.

- `around`, se ejecuta en vez de ejecutar el punto de enlace.
- `around throws`, se ejecuta en vez de ejecutar el punto de enlace pero puede lanzar una excepción.

Además de esto, en un *advice* se puede acceder a:

- `thisJoinPoint`, que proporciona información reflectiva sobre el punto de enlace.
- `proceed`, (sólo accesible en los de tipo `around`) que permite ejecutar el código original del punto enlace.

En la Figura 5.8 se muestra un ejemplo sencillo de *advice*.

```
//utilizando un punto de corte con nombre
pointcut setter(Point p1, int newval): target(p1) && args(newval)
                                        (call(void setX(int) ||
                                         call(void setY(int)));

  before(Point p1, int newval): setter(p1, newval) {
      System.out.println("se va a fijar algo en " + p1 +
                         " al nuevo valor de " + newval);
  }
//lo mismo utilizando punto de corte anónimo
before(Point p1, int newval): target(p1) && args(newval)
                               (call(void setX(int)) ||
                                call(void setY(int))) {
      System.out.println("se va a fijar algo en " + p1 +
                         " al nuevo valor de " + newval);
  }
```

Figura 5.8. Ejemplo de advice con AspectJ.

En este ejemplo se muestran dos formas de realizar lo mismo, la primera crea un punto de corte y le da un nombre, para posteriormente utilizarlo en un *advice* (o en varios); la segunda utiliza un punto de corte de forma anónima (lo crea dentro del *advice*, con lo que no se puede utilizar en más *advice*). Lo que realiza este punto de corte es seleccionar las invocaciones a los métodos `setX` y `setY`, que reciben como parámetro un único `int`, y cuyo objeto es una instancia de la clase `Point`. Lo que hace el *advice* es fijar que antes (`before`) de que se produzca el punto de corte se ejecute el código (en este caso simplemente muestra por pantalla una frase, pero podría ser la invocación a un método, o cualquier cosa que se necesitase).

Este sistema presenta un conjunto de puntos de enlace muy rico, permitiendo adaptar las aplicaciones de una forma muy potente. Los principales inconvenientes que presenta este sistema son la dependencia del lenguaje Java, lo que implica que programas escritos en otros lenguajes no pueden ser adaptados mediante este sistema, y la necesidad de disponer del código fuente del componente a adaptar.

5.3 Separación Dinámica de Aspectos

Usando un tejedor estático, el programa final se genera tejiendo el código de la funcionalidad básica y el de los aspectos seleccionados en la fase de compilación. Si se quiere enriquecer la aplicación con un nuevo aspecto, o incluso eliminar uno de los aspectos actualmente tejidos, el sistema debe ser recompilado y reiniciado de nuevo.

Aunque no todas las aplicaciones necesitan ser adaptadas mediante aspectos en tiempo de ejecución, hay aspectos específicos que se benefician de un sistema con tejido dinámico; puede haber aplicaciones que necesiten adaptar sus competencias específicas en respuesta a cambios en el entorno de ejecución [Popovici2001]. Como ejemplo, técnicas relacionadas se han empleado en gestionar requerimientos de calidad del servicio en sistemas distribuidos de CORBA [Zinky97], en la gestión de sistemas de "*cache prefetching*" (precarga en memoria) en un servidor Web [Devillechaise2003], y en distribución de incumbencias basada en el balanceo de carga [Matthijs97]. Otro ejemplo de uso de POA de forma dinámica es lo que recientemente se ha llamado software autónomo (*autonomic software/computing*): software capaz de auto repararse, gestionarse, optimizarse o recuperarse [Autonomic].

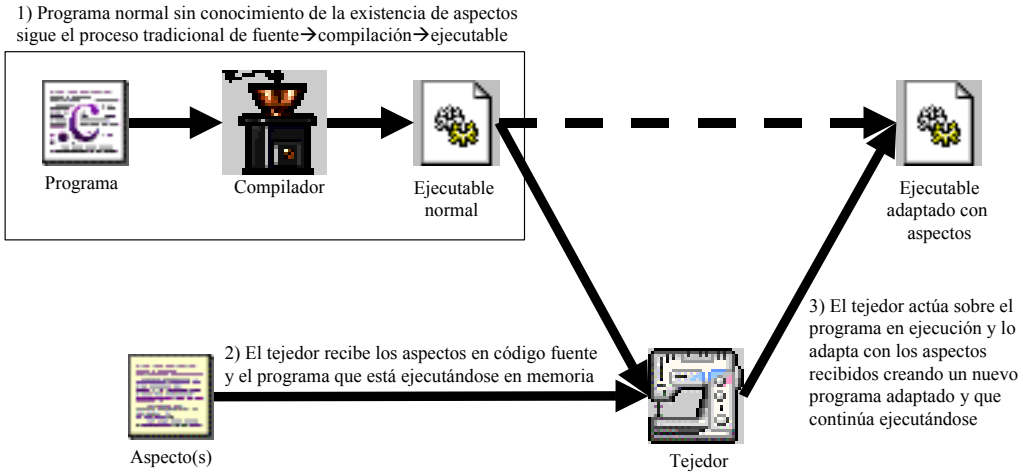


Figura 5.9. Tejido dinámico.

En la Figura 5.9 se puede ver representado el proceso del tejido dinámico. En un primer paso, un programa escrito en un lenguaje normal pasa por el compilador y se genera un ejecutable que se pone en ejecución (este es el proceso tradicional). Este programa no tiene por qué tener ningún conocimiento de qué va a ser adaptado. Cuando se necesita adaptar (añadir o modificar funcionalidad) el programa que está ejecutándose (porque surgen nuevos requerimientos o en respuesta a cambios en el entorno), sin detener la ejecución, se procesa el programa en ejecución (no el programa ejecutable, sino el que está en memoria) junto a los aspectos que van a adaptarlo, y por medio del tejedor se modifica el programa en memoria añadiéndole la funcionalidad de los aspectos, dando lugar a una nueva versión del programa que continúa la ejecución. El programa ejecutable inicial no ha sido modificado con lo que se puede utilizar de nuevo sin las modificaciones realizadas en memoria.

En sistemas que usan tejido dinámico de aspectos, la funcionalidad básica permanece separada de los aspectos en todo el ciclo de vida del software, incluso en la ejecución del sistema. El código resultante es más adaptable y reutilizable, y los aspectos y la funcionalidad básica pueden evolucionar de forma independiente [Pinto2002].

Con el fin de superar las limitaciones del tejido estático han surgido diferentes sistemas que ofrecen tejido dinámico. Estos sistemas ofrecen al programador la posibilidad de modificar de forma dinámica el código del aspecto asignado a puntos de enlace de la aplicación de forma similar a los sistemas reflectivos en tiempo real

basado en protocolos de meta objetos (*meta object protocols*) (MOP) [Maes87, Kiczales92, Sullivan2001, Baker2002].

Muchas de las herramientas que afirman ofrecer un tejido dinámico de aspectos realmente ofrecen un híbrido entre el tejido estático y el dinámico, pues los aspectos deben conocerse y definirse en el momento del diseño e implementación, para posteriormente, en ejecución, poder instanciarlos. Aunque esta solución aporta alguna ventaja respecto al tejido estático hay casos en que no es suficiente, por ejemplo cuando una vez que la aplicación está funcionando surge un requerimiento nuevo, que no se tuvo en cuenta en el diseño.

Al igual que ocurre con las herramientas de tejido estático, la mayoría de las herramientas que ofrecen tejido dinámico son dependientes del lenguaje como se verá más adelante.

La principal desventaja del tejido dinámico respecto al estático es el rendimiento en tiempo de ejecución [Böllert99].

5.3.1 PROSE

El sistema PROSE (*PROgramable extenSions of sErvices*) [Popovici2001, Popovici2002], ofrece POA de forma dinámica, permitiendo adaptar en tiempo de ejecución una aplicación sin necesidad de haber definido nada en el momento del diseño.

Su plataforma está implementada en Java, siendo también Java el lenguaje de codificación de los aspectos. Provee al usuario con un subconjunto de las características esenciales de la POA [Popovici2002].

Para implementar el sistema se ha utilizado el interfaz de depuración de la máquina virtual de Java (JVM *Debugger Interface*, JVMDI) y se ha creado un *plug-in* (añadido) para la JVM de tal forma que soporte el concepto de aspecto de forma directa. Este *plug-in* recibe el nombre de interfaz de aspectos de la JVM (JVM *Aspect Interface*, JVMAI) que es el que permite al usuario el tejido dinámico de aspectos.

Los aspectos pueden ser instanciados dinámicamente e insertarse bien localmente, desde una aplicación que se ejecuta en la misma máquina virtual, o bien remotamente, utilizando un interfaz remoto del la implementación PROSE del JVMDI. Mediante esta última opción las instancias de los aspectos sí que pueden ser generados e inicializados fuera de la máquina virtual en la cual van a ser tejidos.

Con posterioridad [Popovici2003] y con la intención de ofrecer mejor rendimiento en ejecución se ha modificado el compilador dinámico (*Just In Time,* JIT) de la máquina virtual de investigación Jikes de IBM (IBM *Jikes Research Virtual Machine*) [Jikes], haciendo que ésta JVM soporte de forma directa la POA, y mediante una API, *Application Program Interface* (interfaz de programación de aplicaciones), se ofrecen puntos de enlace a la capa superior. Su funcionamiento se basa en la introducción de ganchos (*hooks weaving*) directamente en el código nativo generado por el compilador JIT. Estos ganchos se corresponden con todos los puntos de enlace potenciales. Cuando se ejecutan, los ganchos determinan si un *advice* necesita ser invocado o no para cada punto de enlace particular.

El trabajar a nivel del compilador JIT proporciona una mejora en la eficiencia pero una pérdida de portabilidad, puesto que el sistema queda ligado a un compilador y una máquina virtual determinada. Para mitigar este problema se modifico la arquitectura original de PROSE estableciendo un diseño en 2 capas [Popovici2003] separando claramente el soporte a los aspectos incrustado en la máquina

virtual (el monitor de ejecución, *execution monitor*) y el sistema de POA específico de la aplicación (motor de POA dinámico, *dynamic AOP engine*). El monitor de ejecución ofrece al motor un modelo de puntos de enlace en forma de API. El objetivo es facilitar el intercambio de motores AOP. Por ejemplo, para permitir describir los aspectos utilizando un lenguaje distinto a Java.

La última versión de PROSE [Nicoara2005] reemplaza el tejido de ganchos por el reemplazo directo del código de los métodos en tiempo de ejecución. Introduce dos nuevas estrategias: tejido de *advices* para los puntos de enlace que impliquen el reemplazo de métodos y tejido de cabos (*stubs*) para los puntos de enlace que comprendan *advices* externos, que serán invocados vía llamadas *callback*. Debe señalarse que estos métodos se añaden como alternativas adicionales a los métodos ya implementados en PROSE. Por ello, en la última versión se soportan todas las estrategias utilizadas: tejido basado en JVMDI, tejido de ganchos, tejido de cabos y tejido de *advices*.

En esta versión, el mecanismo de funcionamiento se basa en tejer los aspectos en tiempo de ejecución disparando la recompilación de los métodos. Como parte de esta recompilación y dependiendo de la naturaleza del *advice* correspondiente, se teje el *advice* completo o un mecanismo de *callback* en el *bytecode* original. El sistema también aprovecha el compilador JIT: cuando se inserta un aspecto, los métodos afectados se recompilan automáticamente, pero en este caso se permiten utilizar versiones optimizadas del mismo incrementando de este modo la eficiencia. En esta implementación se necesita extender el funcionamiento de la máquina virtual Jikes, para permitir el reemplazo de código en tiempo de ejecución. La arquitectura de esta versión mantiene la separación de dos capas descrita reescribiendo la correspondiente al motor de ejecución [Nicoara2005].

PROSE soporta los siguientes tipos de puntos de enlace:

- **Límites de los métodos (*method boundaries*)**: puntos de entrada y salida de los métodos.
- **Redefinición de métodos**: reemplazo del cuerpo de los métodos.
- **Acceso y modificación de campos**: seguimiento del acceso a las variables.
- **Puntos de enlace relacionados con las excepciones**, incluyendo la captura y lanzamiento de excepciones.

5.3.2 DynamicAspects

DynamicAspects es un *framework* de programación orientada a aspectos dinámica desarrollado para la plataforma Java. Se trata de un proyecto de código abierto [DynamicAspects] cuyo autor es Marco Petris.

Los aspectos se especifican como objetos Java que implementan el interfaz `Advice`. En la versión actual las implementaciones por defecto de este interfaz extienden el interfaz `BeforeAfterAdvice` que a su vez extiende `Advice`, y permite introducir el código del aspecto a través de dos métodos `before()` (antes) y `after()` (después).

Los aspectos pueden instalarse y desinstalarse sobre las clases en tiempo de ejecución. La manera en la que los métodos son instalados se modela mediante el

concepto de "tipo de tejido" representado con la clase `WeaveType`. Los tipos de tejido implementados son:

- ***Execution* (ejecución)**. Permite envolver con el aspecto la ejecución de un método o constructor.
- ***Call* (llamada)**. Permite envolver con el aspecto la llamada a un método o constructor.
- ***CFlow***. Permite restringir la ejecución del aspecto con un número arbitrario de condiciones que se modelan a su vez como objetos.

Para la instalación de un aspecto sobre una clase, además del tipo de tejido a utilizar, deben proporcionarse los puntos de corte. Los puntos de corte vendrán representados por los patrones que describen los métodos cuya llamada o ejecución deberá ser observada por el aspecto. DynamicAspects utiliza el paquete `java.util.regex` para procesar las expresiones regulares que representan los patrones. Se proporciona así mismo una clase `PointcutFactory` que permite construir los patrones programáticamente y proporciona constantes con las expresiones regulares más comunes.

En su implementación utiliza el paquete de instrumentación de bytecode (*Instrumentation package*) que forma parte de la nueva API de *profiling* introducida en la plataforma estándar Java con la JDK 1.5: JVM *Tool Interface* [Sun2004].

El paquete de instrumentación `java.lang.instrument` permite el desarrollo de agentes Java (*Java Agents*) que serán invocados al ejecutar una aplicación con la máquina virtual de Java utilizando el argumento `-javaagent`. Los agentes son librerías Java [Sun2004b] que contienen una clase que implemente un método con la siguiente signatura: `public static void premain(String options, Instrumentation instrumentation)`. El interfaz `Instrumentation` contiene, entre otros, un método que permite redefinir el bytecode de las clases. En la redefinición se puede cambiar los cuerpos de los métodos pero no se permite cambiar el esquema de las clases. Este método es utilizado para tejer el código de los aspectos sobre las clases en tiempo de ejecución.

Aunque DynamicAspects permite instalar y desinstalar aspectos sobre las clases en tiempo de ejecución, las clases de los aspectos deben de ser conocidas en tiempo de compilación por lo que los aspectos tienen que definirse en tiempo de diseño. Los aspectos no pueden ser generados en tiempo de ejecución.

5.3.3 AspectWerkz

AspectWerkz [Boner2003] es un *framework* de programación orientada a aspectos para Java. Permite realizar el tejido del bytecode de las clases en tiempo de compilación, carga de clases y ejecución.

Los aspectos se representan como objetos Java. Los *advices* se corresponderán con los métodos del aspecto. Estos métodos deben recibir como parámetro un objeto del tipo `JoinPoint` para que el sistema pueda reconocerlos en la fase de tejido.

Para especificar los puntos de corte y los *advices* que deben ser insertados en ellos pueden utilizarse diversos métodos:

- Utilizar un fichero de definición XML.
- Usar el sistema de anotaciones introducido en la plataforma J2SE 1.5 [Gosling2004].
- Utilizar *doclets* personalizados del sistema *javadoc* para versiones anteriores a la 1.5 de la plataforma J2SE. Será necesario utilizar un compilador específico de AspectWerkz para procesarlos (*AnnotationC compiler*).

El sistema de anotaciones (o los *doclets* personalizados) permite definir sobre qué puntos de enlace se ejecutará cada *advice* a través de los metadatos asociados a cada método del aspecto. Es decir, requiere modificar el código del aspecto.

Para la selección de los puntos de enlace AspectWerkz define un lenguaje de patrones propio. Los patrones utilizados serán normalmente una combinación de patrones de clases, métodos y campos. Dentro de los patrones pueden utilizarse referencia a anotaciones de las clases y miembros para filtrar los resultados.

Los puntos de corte se especifican mediante una combinación de una serie de identificadores que definen el tipo de punto de corte y los patrones que identifican los puntos de enlace. Los puntos de corte soportados son:

- `staticinitialization()`. Inicialización estática de una clase.
- `execution()`. Ejecución de métodos o constructores.
- `call()`. Llamadas a métodos o constructores.
- `set()`. Modificación de un campo.
- `get()`. Acceso a un campo.
- `handler()`. Cláusulas `catch`.
- `within()`. Permite restringir un conjunto de tipos a un patrón dado. Se utiliza en combinación con otros puntos de corte como `call` y `handler`.
- `withincode()`. Permite restringir el ámbito de los métodos y constructores a un patrón dado. Suele utilizarse en combinación con otros puntos de corte como `call`, `get`, `set` y `handler`.
- `cflow()`. Permite definir un control de flujo sobre una expresión que describa un punto de corte.
- `hashmethod()`. Cualquier clase que tenga un método dado.
- `hashfield()`. Cualquier clase que tenga un campo dado.

AspectWerkz permite la instalación y desinstalación de aspectos en tiempo de ejecución, si bien éstos tienen que estar definidos en tiempo de diseño. Se ofrecen diversas opciones para realizar este despliegue dinámico de los aspectos siendo los dos parámetros básicos la clase del aspecto que se quiere instalar y una cadena de caracteres XML que define los puntos de corte sobre los que se instalará el aspecto.

En cuanto a la implementación del tejido dinámico, desde la versión 2.0 de AspectWerkz soporta el paquete de instrumentación de bytecode introducido en la plataforma J2SE 1.5 como parte de la API de *profiling JVM Tool Interface* [Sun2004] (ver § 5.3.2). En el caso de J2SE 1.4 utiliza la tecnología HotSwap de la arquitectura JPDA (*Java Platform Debugger Architecture*) [Sun2003b] que permite reemplazar clases en tiempo de ejecución.

5.3.4 CLAW

El sistema CLAW (*Cross-Language Load-Time Aspect Weaving*, tejido de aspectos en tiempo de carga de lenguajes cruzados) [Lam2002], inicialmente llamado RAW (*Runtime Aspect Weaver*, tejedor de aspectos en tiempo de ejecución) está implementado sobre la plataforma .NET. El trabajo lo realiza a nivel de código intermedio (MSIL) obteniendo con ello un gran beneficio: la independencia del lenguaje.

En la plataforma .NET cualquier programa escrito en cualquier lenguaje de programación es traducido al código intermedio (MSIL) que es el que se ejecuta por parte del CLR (*Common Language Runtime*). Al introducir los aspectos a este nivel se consigue que cualquier programa escrito en cualquier lenguaje pueda ser adaptado por aspectos escritos en cualquier otro lenguaje, ya que al final ambos son compilados a MSIL y es en este nivel donde son tejidos.

Como se puede ver por el nombre inicial (RAW) y por el definitivo (CLAW) en un principio se argumentaba que este sistema ofrecía tejido en tiempo de ejecución para, posteriormente afirmar que el tejido es en tiempo de carga.

En realidad en este sistema, aunque los aspectos se tejen en tiempo de ejecución, tienen que estar definidos previamente (en tiempo de diseño) lo que implica que el sistema no puede adaptarse ante nuevos requerimientos no previstos en el diseño.

La forma de trabajar de este sistema es utilizar la API de *profiling* que provee .NET [Pietrek2001]. En concreto los dos siguientes interfaces: `IcorProfilerCallback` e `IcorProfilerCallbackInfo`. Estos interfaces funcionan mediante un sistema basado en eventos. Cuando el sistema de aspectos empieza a ejecutarse informa al motor de ejecución de los eventos que quiere monitorizar (los eventos más importantes a monitorizar serian la carga de módulos, carga de "*assemblies*" (ficheros ensamblados ejecutables) o la compilación justo a tiempo (*just in time*, JIT).

En tiempo de ejecución el motor de ejecución llamaría al sistema cuando se produjesen esos eventos, y éste se encargaría (mediante reflectividad) de examinar el MSIL existente, adaptarlo según sea necesario, y escribir el nuevo MSIL adaptado en memoria para que el compilador JIT se encargue de compilarlo y ejecutarlo.

El principal inconveniente de esta forma de trabajar es que el sistema tiene que estar en modo *profiling*, lo que implica una penalización en el rendimiento. Además, para obtener la reflectividad necesaria el autor ha utilizado los interfaces de metadatos no gestionado (*Unmanaged Metadata Interfaces*) los cuales son una conjunto de interfaces COM [Box99] que están accesibles desde fuera del entorno .NET lo que hace que no sea portable al ser tecnología propia de Microsoft.

5.3.5 Rapier-LOOM.NET

El sistema LOOM.NET desarrollado en el *Hasso Plattner Institute* ofrece en principio un tejido estático de aspectos [Schult2002] pero con posterioridad se le ha sido añadido una parte de tejido dinámico. Actualmente se conoce con el nombre Gripper-LOOM.NET a la parte estática y con Rapier-LOOM.NET a la dinámica.

Aunque el sistema se ha implementado para C# los autores argumentan que es extensible para cualquier lenguaje dentro de la plataforma .NET. Se utilizan "*custom attributes*" (atributos personales) de C# para definir los puntos de enlace, y mediante introspección y reflectividad son evaluados en tiempo de ejecución.

Los aspectos se representan como clases .NET que extienden la clase `Aspect`. Los métodos del aspecto constituirán los *advices* y se identificarán anotándolos con atributos que especifican los puntos de corte donde se realizará el tejido. Los puntos de enlace soportados son la invocación de métodos y constructores y pueden categorizarse mediante atributos que especificarán como se realizará el tejido:

- `Before`. Antes de la invocación.
- `After`. Después de la invocación.
- `Instead`. En lugar de la invocación.
- `AfterReturning`. Después de retornar la invocación. No se soporta en constructores.
- `AfterThrowing`. Después de lanzar una excepción. No se soporta en constructores.

Para especificar los métodos concretos que constituirán los puntos de corte se utilizan atributos con los que se especifican sentencias de inclusión y exclusión. Estas sentencias pueden recibir como parámetros referencias a clases específicas o también cadenas de caracteres con las que se permite el uso de comodines.

Para la implementación del tejido dinámico se utiliza una estrategia de generación de *proxies* (GOF94) que actuarán como delegados de las clases modificadas con el código de los aspectos. En tiempo de ejecución deberá utilizarse una fábrica de objetos que, dados el aspecto o aspectos a utilizar y la clase sobre la que realizar el tejido, devuelve una instancia de la clase con el tejido realizado. Este mecanismo impone una limitación importante: debe modificarse el código de la aplicación para instanciar los objetos cuyas clases pueden ser ampliadas con aspectos utilizando la fábrica proporcionada por el sistema. Además, para poder generar el *proxy* que redefina los métodos ampliados, los métodos candidatos tienen que ser virtuales o estar definidos vía un interfaz para poder realizar llamadas polimórficas.

Al igual que ocurría en CLAW (§ 5.3.4) se hace uso de los interfaces de metadatos no gestionados (*Unmanaged Metadata Interfaces*) con la misma pérdida de portabilidad mencionada antes.

Finalmente, debe señalarse que la el tejido dinámico consiste únicamente en la posibilidad de instanciar los aspectos en tiempo de ejecución, pero no se permite la creación de aspectos no contemplados en tiempo de diseño.

5.4 Conclusiones

La programación orientada a aspectos es una técnica de implementación que proporciona un soporte a nivel de lenguaje de programación para modularizar las incumbencias ortogonales a la funcionalidad de la aplicación. Los aspectos expresan funcionalidad que "atraviesa" el sistema de un modo modular, permitiendo diseñar un sistema como composición de aspectos separados. Por ello, esta técnica se muestra a priori como una candidata idónea para separar totalmente la incumbencia de persistencia de la funcionalidad de la aplicación (§ 2.1).

De hecho, en la literatura existente en AOSD es común ver la persistencia como una funcionalidad candidata a ser tenida en cuenta como un aspecto [Mens97, Suzuki99]. Teóricamente, tendría que ser posible:

- Modularizar la persistencia de una aplicación como un aspecto, empleando técnicas propias de la POA.
- Reutilizar los aspectos propios de la persistencia de un modo independiente al tipo de aplicación.
- Desarrollar programas de un modo independiente a la naturaleza persistente de sus datos.

Analizando distintas implementaciones de persistencia como aspectos, nos hemos dado cuenta de que dichos objetivos no se han alcanzado en implementaciones reales.

Como un primer ejemplo, PersAJ [Rashid2000] es un sistema que implementa un prototipo para almacenar aspectos en una base de datos orientada a objetos. Para mantener el modelo de persistencia independiente del sistema de POA, se implementó un aspecto para describir la representación persistente de los aspectos. Su objetivo era proporcionar un modelo de persistencia de aspectos, pero los datos de las aplicaciones y el código de persistencia no pudo ser separado.

Adicionalmente, Kielze y Guerraoui [Kielze2002] proporcionaron una evaluación de POA para separar la concurrencia y tolerancia a fallos de un sistema distribuido. Lo único que implementaron como un aspecto separado referente la persistencia fueron transacciones, sin conseguir el almacenamiento y recuperación de objetos persistentes con POA. Otro ejemplo de la aplicación de la POA para la automatización de la acometida de transacciones ya fue analizada en § 3.6.2.

Otro estudio llevado a cabo por Rashid y Chitchyan se centró en desarrollar un sistema de persistencia con AspectJ [Rashid2003]. Su conclusión fue que el desarrollo de una aplicación persistente no podía ser llevado a cabo independientemente de los aspectos de persistencia. Si bien el almacenamiento y actualización de los objetos persistentes puede desarrollarse aparte, la obtención y borrado de los mismos tiene que ser realizada explícitamente.

Por ejemplo, en Java, debido a la recolección de basura no existe un punto de enlace donde tejerse el aspecto que se encargue del borrado en la base de datos. En este caso sería necesario poder modificar el mecanismo de borrado de objetos implementado en la máquina virtual Java. La programación orientada a aspectos no da respuesta a esta necesidad: permite modelar las operaciones mediante aspectos, pero no permite modelar la semántica del motor computacional que ejecuta dichas operaciones (en el ejemplo, la máquina virtual de la plataforma Java).

Debe destacarse que estas conclusiones coinciden con los resultados obtenidos en el Capítulo 3 dedicado al análisis de sistemas de persistencia existentes: los sistemas que mejores resultados ofrecían en cuanto a transparencia se basaban en la modificación del código de la aplicación para inyectar los mecanismos de persistencia, pero ningún sistema permitía eliminar de la aplicación las sentencias de código necesarias para el borrado de objetos persistentes. Del mismo modo, los mecanismos de persistencia por alcance [Atkinson95] implementados en estos sistemas ofrecían cierta transparencia pero siempre se debía recuperar explícitamente los objetos raíz.

La programación orientada a aspectos representa un mecanismo general y mucho más potente de transformación de código que las diferentes técnicas analizadas en los sistemas del Capítulo 3. Sin embargo, como se ha mencionado, no permite dar respuesta a las operaciones de obtención y borrado, necesarias para que el sistema de persistencia sea completamente transparente a la aplicación (§ 2.1). La razón es que esta respuesta no puede encontrarse en la transformación del código de las aplicaciones, independientemente de que esta transformación sea estática (§ 5.2) o dinámica (§ 5.2).

Las herramientas de POA que ofrecen un tejido dinámico de aspectos pueden ser interesantes para dar respuesta a los requisitos de adaptabilidad (§ 2.2) del sistema de persistencia. Sin embargo se ha observado cómo todas las herramientas de POA dinámicas analizadas, con la excepción de PROSE (§ 5.3.1), no permiten la generación de aspectos en tiempo de ejecución. Los aspectos deben ser conocidos en tiempo de diseño y en tiempo de ejecución únicamente pueden instalarse y desinstalarse de las clases. Esto impide dar respuesta a los requisitos que no hayan sido tenidos en cuenta en tiempo de diseño restringiendo de este modo el grado de adaptabilidad dinámica (§ 2.2.1) y programática (§ 2.2.2) de la aplicación.

En cuanto a los requisitos de portabilidad, se han observado graves carencias en los sistemas POA analizados. Estas carencias son debidas fundamentalmente a la falta de mecanismos estandarizados en las plataformas utilizadas para la modificación del código de las aplicaciones en tiempo de ejecución. Los sistemas analizados que funcionan sobre la plataforma .NET (§ 5.3.4 y § 5.3.5) presentan una independencia del lenguaje (§ 2.3.1) por realizar el tejido a nivel de MSIL, sin embargo se encuentran limitados a entornos Microsoft (violación del requisito § 2.3.2) por los mecanismos utilizados en su implementación. En este sentido, dentro de la plataforma Java supone un gran avance la estandarización de la API de *profiling* [Sun2004] y la inclusión de un mecanismo estándar de instrumentación de bytecode (§ 5.3.2 y § 5.3.3). En el caso de PROSE, las últimas versiones han sacrificado portabilidad por rendimiento, restringiendo su funcionamiento a la máquina virtual Jikes de IBM [Jikes], modificando el compilador utilizado y la propia máquina virtual.

Capítulo 6
REFLECTIVIDAD (REFLEXIÓN)

Previamente a este capítulo, hemos estudiado un conjunto de sistemas que dotaban a distintos entornos computacionales de un cierto grado de flexibilidad. Cada uno de dichos sistemas utilizaba distintas técnicas para conseguir adaptarse a las distintas necesidades del usuario, así como para poder ampliar o extender sus características.

A lo largo de este capítulo, introduciremos los conceptos propios de una técnica utilizada para conseguir un elevado grado de flexibilidad en determinados sistemas: la reflectividad[8] computacional. Haremos un estudio genérico de las posibilidades de los sistemas reflectivos y estableceremos distintas clasificaciones en función de distintos criterios. En el siguiente capítulo, todos los términos definidos y estudiados aquí serán utilizados para analizar los distintos sistemas dotados de reflectividad, así como para, posteriormente, poder especificar y diseñar nuestro propio entorno computacional reflectivo.

6.1 Conceptos de Reflectividad

6.1.1 Reflectividad

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 –el conjunto de información que computa– añade al dominio de un sistema convencional la estructura y comportamiento de sí mismo. Se trata pues de un sistema capaz de acceder, analizar y modificarse a sí mismo, como si de una serie de estructuras de datos propias de un programa de usuario se tratase.

[8] Traducción de *reflection*. Muchos autores la han traducido también como reflexión. Nosotros utilizaremos el término reflectividad a lo largo de esta memoria.

6.1.2 Sistema Base

Aquél sistema de computación que es dominio de otro sistema computacional distinto, denominado metasistema [Golm97]. El sistema base es el motor de computación (intérprete, software o hardware) de un programa de usuario.

6.1.3 Metasistema

Aquél sistema computacional que posee por dominio de computación a otro sistema computacional denominado sistema base. Un metasistema es por lo tanto un intérprete de otro intérprete.

6.1.4 Cosificación

Cosificación o concretización (*reification*) es la capacidad de un sistema para acceder al estado de computación de una aplicación, como si se tratase de un conjunto de datos propios de una aplicación de usuario [Smith82]. La cosificación es la posibilidad de acceso desde un sistema base a su metasistema, en el que se puede manipular el sistema base como si de datos se tratase; es el salto del entorno de computación de usuario al entorno de computación del intérprete de la aplicación de usuario.

Si podemos cosificar[9] un sistema, podremos acceder y modificar su estructura y comportamiento y, por lo tanto, podremos añadir a su dominio computacional su propia semántica; estaremos trabajando pues, con un sistema reflectivo.

[9] Cosificar o concretizar: transformar una representación, idea o pensamiento en cosa.

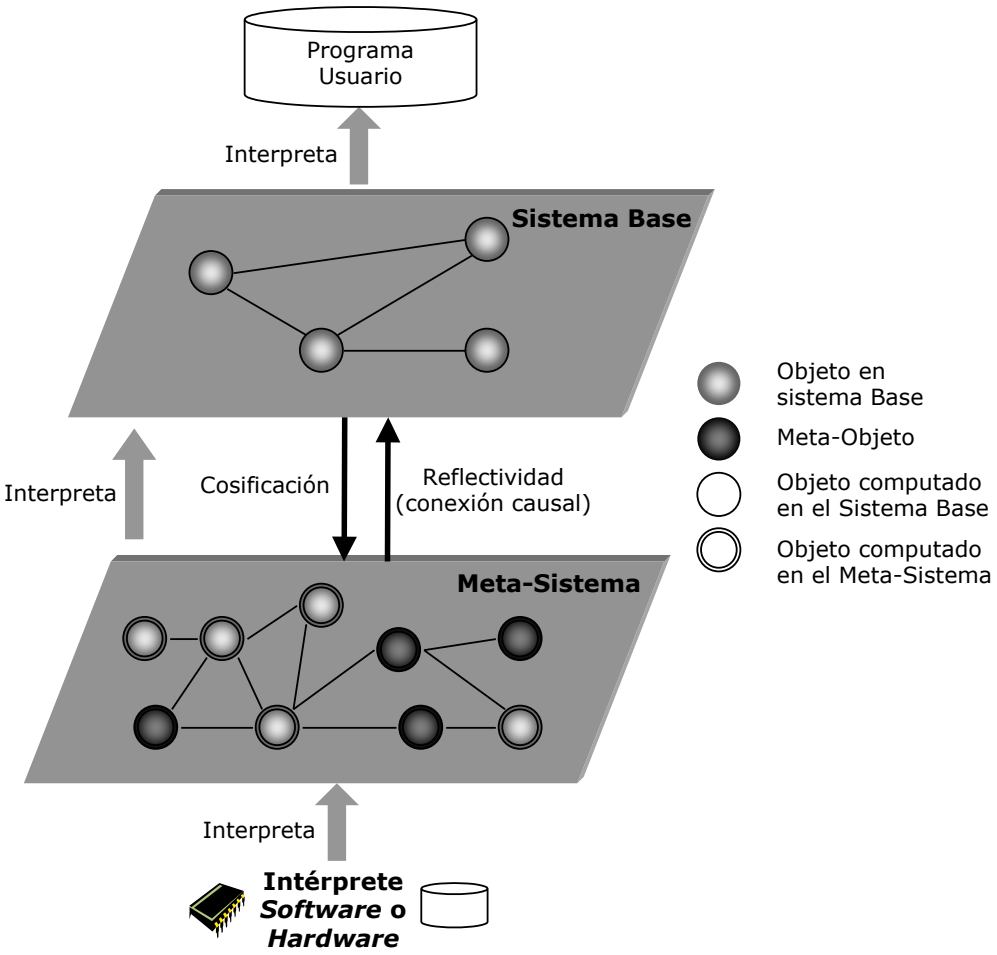


Figura 6.1. Entorno de computación reflectivo.

6.1.5 Conexión Causal

Un sistema es causalmente conexo si su dominio computacional alberga el entorno computacional de otro sistema (permite la cosificación del sistema base) y, al modificar la representación de éste, los cambios realizados causan un efecto en su propia computación (la del sistema base) [Maes87b].

Un sistema reflectivo es aquél capaz de ofrecer cosificación completa de su sistema base, e implementa un mecanismo de conexión causal. La forma en la que los distintos sistemas reflectivos desarrollan esta conexión varía significativamente.

6.1.6 Metaobjeto

Identificando la orientación a objetos como paradigma del sistema reflectivo, puede definirse un metaobjeto como un objeto del metasistema que contiene información y comportamiento propio del sistema base [Kiczales91]. La abstracción del metaobjeto no es necesariamente la de un objeto propio del sistema base, sino que puede representar cualquier elemento que forme parte de éste –por ejemplo, el mecanismo del paso de mensajes.

6.1.7 Reflectividad Completa

La conexión causal no es suficiente para un sistema reflectivo íntegro. Cuando la cosificación de un sistema es total, desde el metasistema se puede acceder a cualquier elemento del sistema base y la conexión causal se produce para cualquier elemento modificado, entonces ese sistema posee reflectividad completa (*completeness*) [Blair97]. En este caso, el metasistema debe ofrecer una representación íntegra del sistema base (todo podrá ser modificado).

Como veremos en el estudio de sistemas reales (Capítulo 7), la mayoría de los sistemas existentes limitan a priori el conjunto de características del sistema base que pueden ser modificados mediante el mecanismo de reflectividad implementado.

6.2 Reflectividad como una Torre de Intérpretes

Para explicar el concepto de reflectividad computacional, así como para posteriormente implementar prototipos de demostración de su utilidad, Smith identificó el concepto de reflectividad computacional como una torre de intérpretes [Smith82].

Diseñó un entorno de trabajo en el que todo intérprete de un lenguaje era a su vez interpretado por otro intérprete, estableciendo una torre de interpretaciones. El programa de usuario se ejecuta en un nivel de interpretación 0; cobra vida gracias a un intérprete que lo anima desde el nivel computacional 1. En tiempo de ejecución el sistema podrá cosificarse, pasando el intérprete del nivel 1 a ser computado por otro intérprete –la reflectividad computacional implica una computación de la computación. En este caso (Figura 6.2.b) un nuevo intérprete', desde el nivel 2, pasa a computar el intérprete inicial. El dominio computacional del intérprete' –mostrado en la Figura 6.2 por un doble recuadro– estará formado por la aplicación de usuario (nivel 0) y la interpretación directa de éste (nivel 1). En esta situación se podrá acceder tanto a la aplicación de usuario (por ejemplo, para modificar sus objetos) como a la forma en la que éste es interpretado (por ejemplo, para modificar la semántica del paso de mensajes). El reflejo de dichos cambios en el nivel 1 es lo que se denomina reflectividad o reflexión.

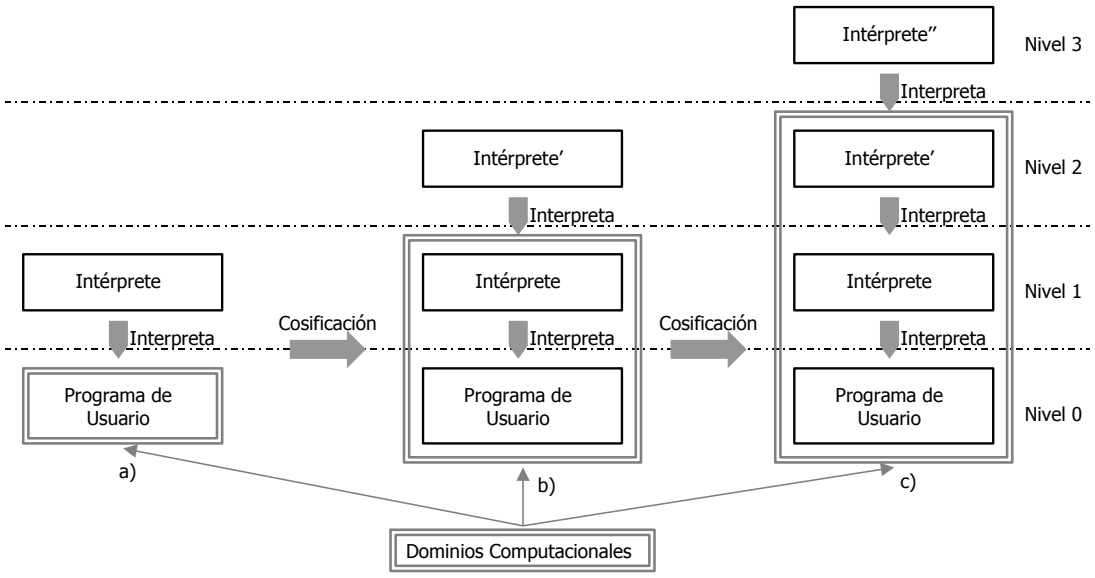


Figura 6.2. Torre de intérpretes definida por Smith.

La operación de cosificación se puede aplicar tantas veces como deseemos. Si el intérprete' procesa al intérprete del nivel 1, ¿podemos cosificar este contexto para obtener un nuevo nivel de computación? Sí; se crearía un nuevo intérprete" que procesare los niveles 0, 1 y 2, ubicándose éste en el nivel computacional 3. En este caso reflejaríamos la forma en la que se refleja la aplicación de usuario (tendríamos una meta-meta-computación).

Un ejemplo clásico es la depuración de un sistema (*debug*). Si estamos ejecutando un sistema y deseamos depurar mediante una traza todo su conjunto, podremos cosificar éste, para generar información relativa a su ejecución desde su intérprete. Todos los pasos de su interpretación podrán ser trazados. Si deseamos depurar el intérprete de la aplicación en lugar de la propia aplicación, podremos establecer una nueva cosificación aplicando el mismo sistema [Douence99].

Este entorno de trabajo fue definido por Smith como una torre de intérpretes [Smith82]. Definió el concepto de reflectividad, y propuso la semántica de ésta para los lenguajes de programación, sin llevar a cabo una implementación. La primera implementación de este sistema fue realizada mediante la modificación del lenguaje Lisp [Steele90], denominándolo 3-Lisp [Rivières84].

La torre infinita de intérpretes siempre tiene un nivel máximo de computación. Siempre existirá un intérprete que no sea interpretado a su vez por otro procesador: un intérprete hardware o microprocesador. Un microprocesador es un intérprete de un lenguaje de bajo nivel, implementado físicamente.

En la Figura 6.3 mostramos un ejemplo real de una torre de intérpretes. Se trata de la implementación de un intérprete de un lenguaje de alto nivel, en el lenguaje de programación Java [Gosling96].

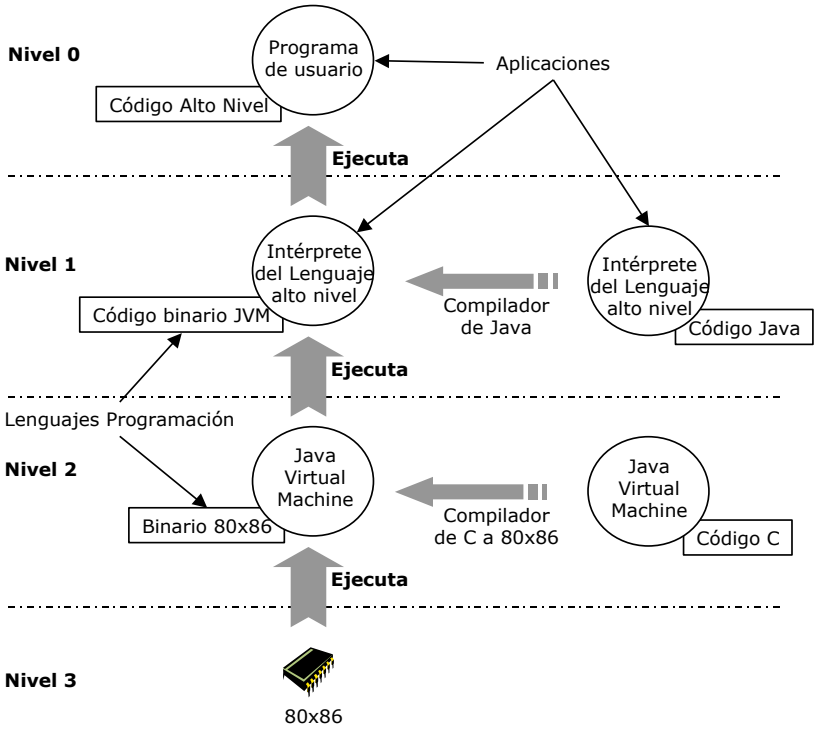


Figura 6.3. Ejemplo de torre de intérpretes en la implementación de un intérprete en Java.

Java es un lenguaje que se compila a un código binario de una máquina abstracta denominada "*Java Virtual Machine*" [Lindholm96]. Para interpretar este código

previamente compilado– podemos utilizar un emulador de la máquina virtual programado en C y compilado, por ejemplo, para un i80X86. Finalmente el código binario de este microprocesador es interpretado una implementación física de éste. En el ejemplo expuesto tenemos un total de cuatro niveles computacionales 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. En el ejemplo, el intérprete del lenguaje de alto nivel define la semántica de dicho lenguaje, y la máquina virtual de Java define la semántica del intérprete del lenguaje de alto nivel. Por lo tanto, el intérprete de Java podrá modificar la semántica computacional del lenguaje de alto nivel, obteniendo así reflectividad computacional.

Cada vez que en la torre de intérpretes nos movamos en un sentido de niveles ascendente, podemos identificar esta operación como cosificación[10] (*reification*). Un movimiento en el sentido contrario se identificará como reflexión (*reflection*) [Smith82].

6.3 Clasificaciones de Reflectividad

Pueden establecerse distintas clasificaciones en el tipo de reflectividad desarrollado por un sistema, en función de diversos criterios. En este punto identificaremos dichos criterios, clasificaremos los sistemas reflectivos basándonos en el criterio seleccionado, y describiremos cada uno de los tipos.

Para cada clasificación de sistemas reflectivos identificado, mencionaremos algún ejemplo existente para facilitar la comprensión de dicha división. Sin embargo, el estudio detallado de los distintos entornos reflectivos existentes se llevará a cabo en el Capítulo 7.

6.3.1 Qué se refleja

Hemos definido reflectividad en § 6.1.1 como "la propiedad de un sistema computacional que le permite razonar y actuar sobre él mismo". El grado de información que un sistema posee acerca de sí mismo –aquello susceptible de ser cosificado y reflejado– establece la siguiente clasificación.

6.3.1.1 Introspección

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

Esta característica se ha utilizado de forma pragmática en numerosos sistemas. A modo de ejemplo, la plataforma Java [Kramer96] ofrece introspección mediante su paquete `java.reflect` [Sun97b]. Gracias a esta capacidad, en Java es posible almacenar un objeto en disco sin necesidad de implementar un solo método: el sistema accede de forma introspectiva al objeto, analiza sus atributos y los

[10] Ciertamente hacemos datos (o cosa) un comportamiento.

convierte en una secuencia de bytes para su posterior envío a un flujo[11] [Eckel2000]. Sobre este mismo mecanismo Java implementa además su sistema de componentes JavaBeans [Sun96]: dado un objeto, en tiempo de ejecución, se determinará su conjunto de propiedades y operaciones.

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

6.3.1.2 Reflectividad Estructural

Se refleja el estado estructural del sistema en tiempo de ejecución –elementos tales como las clases, el árbol de herencia, la estructura de los objetos y los tipos del lenguaje– permitiendo tanto su observación como su manipulación [Ferber88].

Mediante reflectividad estructural, se puede acceder al estado de la ejecución de una aplicación desde el sistema base. Podrá conocerse su estado, acceder las distintas partes del mismo, y modificarlo si se estima oportuno. De esta manera, una vez reanudada la ejecución del sistema base (después de producirse la reflexión), los resultados pueden ser distintos a los que se hubieren obtenido si la modificación de su estado no se hubiera llevado a cabo.

Ejemplos de sistemas de naturaleza estructuralmente reflectiva son Smalltalk-80 [Goldberg83] y Self [Ungar87]. La programación sobre estas plataformas se centra en ir creando los objetos mediante sucesivas modificaciones de su estructura. No existe diferencia entre tiempo de diseño y tiempo de ejecución; al poder acceder a toda la estructura del sistema, el programador se encuentra siempre en tiempo de ejecución modificando dicha estructura.

6.3.1.3 Reflectividad Computacional

También denominada reflectividad de comportamiento (*behavioral reflection*). Se refleja el comportamiento exhibido por un sistema computacional, de forma que éste pueda computarse a sí mismo mediante un mecanismo de conexión causal (§ 6.1) [Maes87]. Un sistema dotado de reflectividad computacional puede modificar su propia semántica; el propio comportamiento del sistema podrá cosificarse para su posterior manipulación.

Un ejemplo de abstracción de reflectividad computacional es la adición de las "*Proxy Classes*" [Sun99] a la plataforma Java2. Apoyándose en el paquete de introspección (`java.reflect`), se ofrece un mecanismo para modificar el paso de mensajes: puede manipularse dinámicamente la forma en la que un objeto interpreta la recepción de un mensaje[12]. De esta forma la semántica del paso de mensajes puede ser modificada.

Otros ejemplos reducidos de reflectividad computacional en la plataforma Java son la posibilidad de sustituir el modo en el que las clases se cargan en memoria y el grado de seguridad de ejecución de la máquina abstracta. Gracias a las clases `java.lang.ClassLoader` y `java.lang.SecurityManager` [Gosling96], se

[11] Este proceso se conoce como "serialización" (*serialization*). El mecanismo de serialización de la plataforma Java ha sido analizado en § 3.2.5.

[12] Estudiaremos este mecanismo, en mayor profundidad, en el siguiente capítulo.

puede modificar en el sistema la semántica obtención del código fuente (de este modo se consigue cargar un *applet* de un servidor Web) y el modo en el que una aplicación puede acceder a su sistema nativo (el sistema de seguridad frente a virus, en la ejecución de *applets*, conocido como *sandbox* [Orfali98]).

6.3.1.4 Reflectividad Lingüística

Un lenguaje de programación posee unas especificaciones léxicas [Cueva93], sintácticas [Cueva95] y semánticas [Ortín2004]. Un programa correcto es aquél que cumple las tres especificaciones descritas. La semántica del lenguaje de programación identifica el significado de cualquier programa codificado en dicho lenguaje, es decir cuál será su comportamiento al ejecutarse.

La reflectividad computacional de un sistema nos permite cosificar y reflejar su semántica; la reflectividad lingüística [Ortín2001] nos permite modificar cualquier aspecto del lenguaje de programación utilizado: mediante el lenguaje del sistema base se podrá modificar el propio lenguaje (por ejemplo, añadir operadores, construcciones sintácticas o nuevas instrucciones).

Un ejemplo de sistema dotado de reflectividad lingüística es OpenJava [Chiba98]. Ampliando el lenguaje de acceso al sistema, Java, se añade una sintaxis y semántica para aplicar patrones de diseño [GOF94], de forma directa, por el lenguaje de programación amolda el lenguaje a los patrones *Adapter* y *Visitor* [Tatsubori98].

6.3.2 Cuándo se produce el reflejo

En función de la clasificación anterior, un sistema determina lo que puede ser cosificado para su posterior manipulación. En este punto clasificaremos los sistemas reflectivos en función del momento en el que se puede llevar a cabo dicha cosificación.

6.3.2.1 Reflectividad en Tiempo de Compilación

El acceso desde el sistema base al metasistema se realiza en el momento en el que el código fuente está siendo compilado, modificándose el proceso de compilación y por lo tanto las características del lenguaje procesado [Golm97].

Tomando Java como lenguaje de programación, OpenJava es una modificación de éste que le otorga capacidad de reflejarse en tiempo de compilación [Chiba98]. En lugar de modificar la máquina virtual de la plataforma para que ésta posea mayor flexibilidad, se rectifica el compilador del lenguaje con una técnica de macros que expande las construcciones del lenguaje. De esta forma, la eficiencia perdida por la flexibilidad del sistema queda latente en tiempo de compilación y no en tiempo de ejecución.

Lo que pueda ser modificado en el sistema está en función de la clasificación § 6.3.1. En el ejemplo de OpenJava se pueden modificar todos los elementos; verbigracia, la invocación de métodos, el acceso a los atributos, operadores del lenguaje o sus tipos. El hecho de que el acceso al metasistema se produzca en tiempo de compilación implica que una aplicación tiene que prever su flexibilidad antes de ser ejecutada; una vez que esté corriendo, no podrá accederse a su metasistema de una forma no contemplada en su código fuente.

6.3.2.2 Reflectividad en Tiempo de Ejecución

En este tipo de sistemas, el acceso del metasistema desde el sistema base, su manipulación y el reflejo producido por un mecanismo de conexión causal, se lleva a cabo en tiempo de ejecución [Golm97]. En este caso, una aplicación tendrá la capacidad de ser flexible de forma dinámica, es decir, podrá adaptarse a eventos no previstos[13] cuando esté ejecutándose.

El sistema MetaXa, previamente denominado MetaJava, modifica la máquina virtual de la plataforma Java para poder ofrecer reflectividad en tiempo de ejecución [Kleinöder96]. A las diferentes primitivas semánticas de la máquina virtual se pueden asociar metaobjetos que deroguen el funcionamiento de dichas primitivas: el metaobjeto define la nueva semántica. Ejemplos clásicos de utilización de esta flexibilidad son la modificación del paso de mensajes para implementar un sistema de depuración, auditoría o restricciones de sistemas en tiempo real [Golm97b].

6.3.3 Cómo se expresa el acceso al metasistema

En función del modo en el que se exprese el metasistema desde el sistema base, podremos establecer la siguiente clasificación.

6.3.3.1 Reflectividad Procedural

La representación del metasistema se ofrece de forma directa por el programa que implementa el metasistema [Maes87]. En este tipo de reflectividad, el metasistema y el sistema base utilizan el mismo modelo computacional; si nos encontramos en un modelo de programación orientado a objetos, el acceso al metasistema se facilitará utilizando este paradigma.

El hecho de acceder de forma directa a la implementación del sistema ofrece dos ventajas frente a la reflectividad declarativa:

- La totalidad del sistema es accesible y por lo tanto cualquier parte de éste podrá ser manipulada. No existen restricciones en el acceso.
- La conexión causal es automática [Blair97]. Al acceder directamente a la implementación del sistema base, no es necesario desarrollar un mecanismo de actualización o reflexión.

6.3.3.2 Reflectividad Declarativa

En la reflectividad declarativa, la representación del sistema en su cosificación es ofrecida mediante un conjunto de sentencias representativas del comportamiento de éste [Maes87]. Haciendo uso de estas sentencias, el sistema podrá ser adaptado.

La ventaja principal que ofrece esta clasificación es la posibilidad de elevar el nivel de abstracción, a la hora de representar el metasistema. Las dos ventajas de la reflectividad procedural comentadas previamente, se convierten en inconvenientes para este tipo de sistemas.

La mayor parte de los sistemas existentes ofrecen un mecanismo de reflectividad declarativa. En el caso de MetaXa [Kleinöder96], se ofrece una representación de acceso a las primitivas de la máquina abstracta modificables.

[13] No previstos en tiempo de compilación –cuando la aplicación esté siendo implementada.

6.3.4 Desde Dónde se puede modificar el sistema

El acceso reflectivo a un sistema se puede llevar a cabo desde distintos procesos. La siguiente clasificación no es excluyente: un sistema puede estar incluido en de ambas clasificaciones.

6.3.4.1 Reflectividad con Acceso Interno

Los sistemas con acceso interno (*inward*) a su metasistema permiten modificar una aplicación desde la definición de éste (su codificación) [Foote92]. Esta característica define aquellos sistemas que permiten modificar una aplicación desde sí misma.

La mayoría de los sistemas ofrecen esta posibilidad. En MetaXa [Kleinöder96], sólo la propia aplicación puede modificar su comportamiento.

6.3.4.2 Reflectividad con Acceso Externo

El acceso externo de un sistema (*outward*) permite la modificación de una aplicación mediante otro proceso distinto al que será modificado [Foote92]. En un sistema con esta característica, cualquier proceso puede modificar al resto. La ventaja es la flexibilidad obtenida; el principal inconveniente, la necesidad de establecer un mecanismo de seguridad en el acceso entre procesos.

El sistema Smalltalk-80 ofrece un mecanismo de reflectividad estructural con acceso externo [Mevel87]: el programador puede acceder, y modificar en su estructura, cualquier aplicación o proceso del sistema.

6.3.5 Cómo se ejecuta el sistema

La ejecución del sistema reflectivo puede darse de dos formas: mediante código nativo o mediante la interpretación de un código intermedio.

6.3.5.1 Ejecución Interpretada

Si el sistema se ejecuta mediante una interpretación software, las características de reflectividad dinámica se pueden ofrecer de una forma más sencilla. Ejemplos de este tipo de sistemas son Self [Smith95], Smalltalk-80 [Krasner83] o Java [Kramer96]. Estas plataformas de interpretación ofrecen características reflectivas de un modo sencillo, al encontrase el intérprete y el programa ejecutado en el mismo espacio de direcciones.

Su principal inconveniente es la pérdida de eficiencia por la inclusión del proceso de interpretación.

6.3.5.2 Ejecución Nativa

Se produce cuando se compila código nativo, capaz de ofrecer cualquier grado de reflectividad indicado en § 6.3.1. Un primer ejemplo es el mecanismo RTTI del ANSI/ISO C++, que ofrece introspección en tiempo de ejecución para un sistema nativo [Kalev98].

Esta clasificación es distinta a la indicada en § 6.3.2. Puede producirse el reflejo del sistema en tiempo de ejecución, tanto para sistemas compilados (nativos) como interpretados. Un ejemplo de ello es el sistema Iguana [Gowing96]. Este sis-

tema implementa un compilador de C++ que ofrece reflectividad computacional en tiempo de ejecución. El código generado es una ampliación del estándar RTTI.

La principal ventaja de este tipo de sistemas es su eficiencia; sin embargo, su implementación es más costosa que la de un intérprete, puesto que debemos generar código capaz de ser modificado dinámicamente.

Capítulo 7
SISTEMAS COMPUTACIONALES DOTADOS DE REFLECTIVIDAD

En el capítulo anterior clasificábamos la reflectividad en función de distintos criterios. Utilizando dicha clasificación y los conceptos definidos en el mencionado capítulo, estableceremos un estudio de un conjunto de sistemas existentes que utilizan de algún modo reflectividad.

Agruparemos los sistemas en función de ciertas características comunes y analizaremos su funcionamiento. El objetivo es estudiar, a la luz de implementaciones reales, qué puede aportar la técnica de reflectividad estudiada en Capítulo 6 de cara al diseño del sistema de persistencia que verifique los requisitos definidos en el Capítulo 2. Las conclusiones de este estudio se reflejarán al final de este capítulo.

7.1 Sistemas Dotados de Introspección

En el capítulo anterior, definíamos introspección como la capacidad de acceder a la representación de un sistema en tiempo de ejecución. Mediante introspección únicamente se puede conocer el sistema, sin permitir la modificación de éste y sin producirse, por lo tanto, su reflejo mediante un sistema de conexión causal. Por esto determinados autores definen introspección como reflectividad estructural "de sólo lectura" [Foote90].

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 distribuidos y programación orientada a objetos.

7.1.1 ANSI/ISO C++ RunTime Type Information (RTTI)

El lenguaje de programación C++ es un lenguaje compilado que genera código nativo para cualquier tipo de plataforma [Cueva98]. En la fase de ejecución de dicho código nativo, existe una carencia del conocimiento de los tipos de los objetos en tiempo de ejecución. El estándar ANSI/ISO de este lenguaje amplió su especificación añadiendo cierta información respecto al tipo (RTTI) en tiempo de eje-

cución, para poder dar seguridad respecto al tipo al programador de aplicaciones [Kalev98].

El caso típico en la utilización de este mecanismo es el ahormado descendente (*downcast*) en una jerarquía de herencia [Eckel2000b]. Un puntero o referencia a una clase base se utiliza genéricamente para englobar cualquier objeto de esa clase o derivado (Figura 7.1). Si tenemos un puntero a dicha clase, ¿qué mensajes le podemos pasar? Sólo aquéllos definidos en la clase base; por lo tanto, en el caso de un objeto instancia de una clase derivada, se pierde su tipo –no se puede invocar a los métodos propios de la clase derivada.

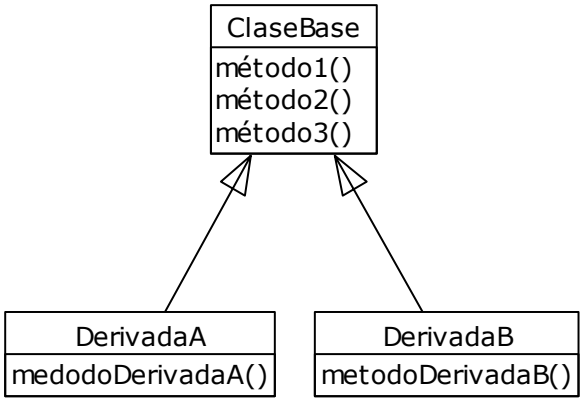


Figura 7.1. Punteros o referencias "Base" permiten la utilización genérica de objetos derivados.

La resolución de este problema se lleva a cabo añadiendo a los objetos información dinámica relativa a su tipo (introspección). Mediante el estándar RTTI podemos preguntarle a un objeto cuál es su tipo y recuperar éste si lo hubiésemos perdido [Stroustrup98]. El siguiente fragmento de código referente a la jerarquía de clases mostrada en la Figura 7.1, recupera el tipo del objeto gracias al operador `dynamic_cast` que forma parte del estándar RTTI.

```
Base *ptrBase;
DerivadaA *ptrDerivadaA=new DerivadaA; // * Tratamiento genérico
mediante herencia ptrBase=ptrDerivadaA;
// * SÓLO se pueden pasar mensajes de la clase Base ptrBase-
>metodoBase();
// * Solicitamos información del tipo del objeto
ptrDerivadaA=dynamic_cast<DerivadaA*>(ptrBase); // * ¿Es de tipo
DerivadaA?
if (ptrDerivadaA)
// * Recuperamos el tipo; se pueden pasar mensajes de Derivada
ptrDerivadaA->mentodoDerivadaA();
```

Al acceder al objeto derivado mediante el puntero base, se pierde el tipo de éste no pudiendo invocar a los mensajes derivados. Gracias a la utilización de RTTI, el programador puede recuperar el tipo de un objeto, e invocar a sus propios mensajes.

7.1.2 Plataforma Java

Java define para su plataforma [Kramer96] una interfaz de desarrollo de aplicaciones (API, *Application Programming Interface*) que proporciona introspección, denominada *The Java Reflection* API [Sun97b]. Este API permite acceder en tiempo de ejecución a la representación de las clases, interfaces y objetos existentes en la máquina virtual. Las posibilidades que ofrece al programador son:

- Determinar la clase de la que un objeto es instancia.
- Obtener información sobre los modificadores de una clase [Gosling96], sus métodos, campos, constructores y clases base.
- 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, sin necesidad de conocer éste en tiempo de compilación.
- Invocar un método de un objeto en tiempo de ejecución, aun siendo éste desconocido en tiempo de compilación.
- Crear dinámicamente un *array*[14] 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 miembros (métodos y atributos) no es posible en la plataforma Java.

Sobre este API de introspección se define el paquete `java.beans`, que ofrece un conjunto de clases e interfaces para desarrollar la especificación de componentes software de la plataforma Java: los JavaBeans [Sun96].

Cabe preguntarse por qué es necesaria la introspección para el desarrollo de un sistema de componentes. Un componente está constituido por métodos, propiedades y eventos. Sin embargo, una aplicación que vaya a utilizar un conjunto de componentes no puede saber a priori la estructura que éstos van a tener. Para conocer el conjunto de estas tres características, de forma dinámica, se utiliza la introspección: permite acceder a los métodos de instancia públicos y conocer su signatura.

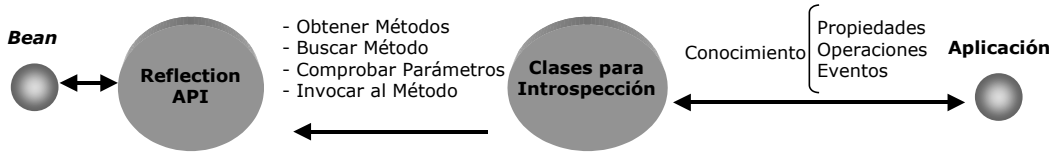


Figura 7.2. Utilización de introspección en el desarrollo de un sistema de componentes.

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 7.2 se aprecia cómo una aplicación solicita el valor de una propiedad de un Bean. Mediante el paquete de introspección se busca el método apropiado y se invoca, devolviendo el valor resultante de su ejecución.

Otro ejemplo de utilización de la introspección en la plataforma Java es la posibilidad de convertir cualquier objeto a una secuencia de *bytes*[15], para posterior-

[14] En Java los arrays son objetos primitivos.
[15] También denominado "serialización" (*serialization*).

mente poder almacenarlo en disco o transmitirlo a través de una red de computadores. Un objeto en Java que herede del *interface* `java.io.Serializable`, sin necesidad de implementar ningún método, podrá ser convertido a una secuencia de *bytes*. ¿Cómo es posible conocer el estado del objeto en tiempo de ejecución? Mediante el acceso introspectivo a éste [Eckel2000].

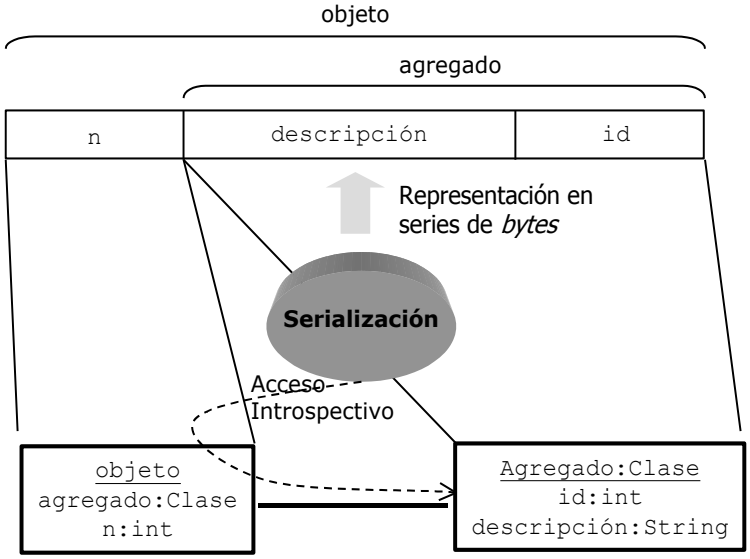


Figura 7.3. Conversión de un objeto a una secuencia de bytes, mediante un acceso introspectivo.

Mediante el paquete `java.reflect` se va consultando el valor de todos los atributos del objeto y convirtiendo éstos a *bytes*. Si un atributo de un objeto es a su vez otro objeto, se repite el proceso de forma recursiva. Del mismo modo es posible crear un objeto a raíz de un conjunto de *bytes* –siguiendo el proceso inverso.

7.1.3 CORBA

CORBA (*Common Object Request Broker Architecture*) es un ambicioso proyecto del consorcio OMG (*Object Management Group*) enfocado a diseñar un *middleware* que proporcione una arquitectura de objetos distribuidos, en la que puedan definirse e interoperar distintos componentes, sin tener en cuenta el lenguaje y la plataforma en la que éstos hayan sido implementados [OMG97].

En el desarrollo de CORBA, se utilizó el concepto de introspección para permitir crear aplicaciones distribuidas portables y flexibles, capaces de reaccionar a futuros cambios producidos en el sistema y, por lo tanto, no dependientes de una interfaz de acceso monolítica. Para ello, CORBA define un modo de realizar invocaciones dinámicas a métodos de objetos remotos –DII, *Dynamic Invocation Interface* [OMG95]– sin necesidad de conocer éstos en el momento de crear la aplicación cliente.

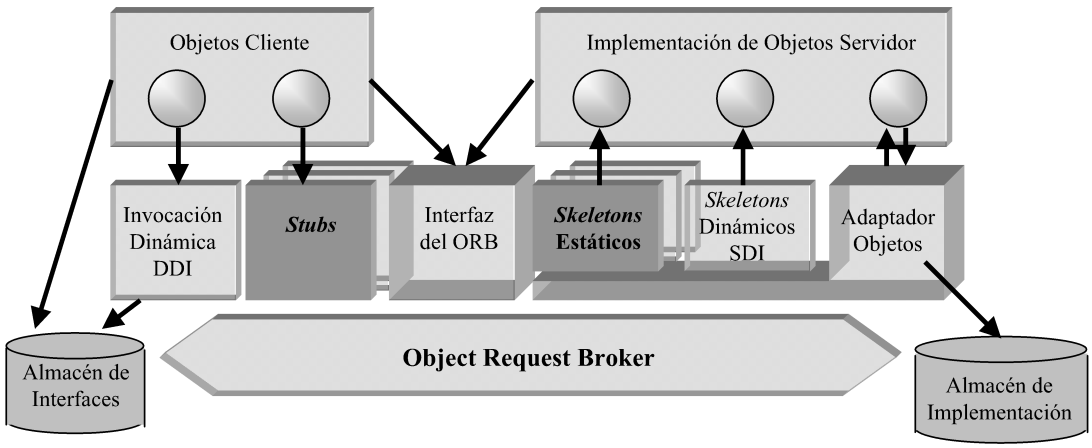


Figura 7.4. Elementos de la arquitectura de objetos distribuidos CORBA.

Como se muestra en la Figura 7.4[16], CORBA proporciona metadatos de los objetos existentes en tiempo de ejecución, guardando en el almacén de interfaces (*Interface Repository*) las descripciones de las interfaces de todos los objetos CORBA. El módulo de invocación dinámica, DII, apoyándose en el almacén de interfaces, permite al cliente conectarse al ORB, solicitar la descripción de un objeto, analizarla, y posteriormente invocar al método adecuado.

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 desarrolla en su totalidad en tiempo de ejecución.
- Para compilar la aplicación cliente que realiza las llamadas a los objetos servidores, no es necesario haber implementado previamente el servidor.
- En las modificaciones del servidor no es necesario recompilar la especificación IDL para el cliente; éste es capaz de amoldarse a los cambios.
- La depuración del cliente es más sencilla puesto que podremos ver el aspecto del servidor desde la plataforma cliente, sin necesidad de llevar una traza paralela de otro proceso.
- 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 los nuevos objetos, coexistiendo así distintas versiones de aplicaciones CORBA.
- Es la base para crear un sistema de componentes CORBA [OMG98].

[16] Para obtener una descripción más detallada del funcionamiento de cada uno de los módulos de la arquitectura CORBA, consúltese [OMG95] o [Orfali98].

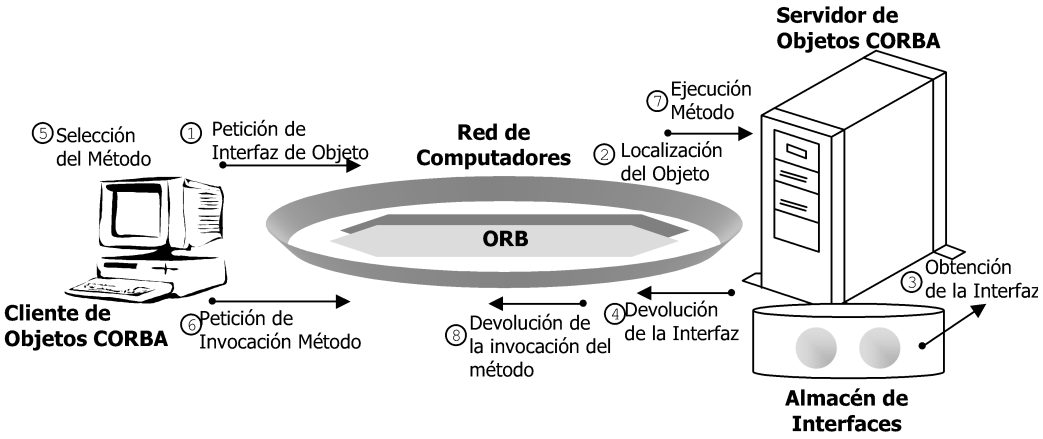


Figura 7.5. Invocación dinámica de un método en CORBA utilizando DII.

En la Figura 7.5 se aprecia cómo un cliente, mediante la utilización del ORB, solicita al módulo DII del servidor de objetos CORBA la interfaz de un objeto. Éste localiza la descripción del objeto en el almacén de interfaces y se lo pasa al cliente. Finalmente, el cliente selecciona el método deseado enviándole al objeto servidor el mensaje oportuno. Todo este proceso se puede llevar a cabo de forma independiente a la implementación del objeto, puesto que en todo momento se trabaja con las descripciones de sus interfaces en único un lenguaje: IDL (*Interface Definition Language*) [OMG96].

Bajo el mismo escenario de la Figura 7.5, se puede suponer el caso de no encontrar el objeto, o método de éste, deseado por el cliente. En este caso no se producirá un error en tiempo de ejecución, sino que el cliente podrá tener una respuesta adicional a esta situación.

El inconveniente del uso introspectivo de CORBA es, como en todos los sistemas que aumentan su grado de flexibilidad, la pérdida de eficiencia en tiempo de ejecución producida por la computación adicional necesaria para obtener la información introspectiva.

7.1.4 COM

COM (*Component Object Model*) es un modelo binario de interacción de objetos construido por Microsoft [Microsoft95], cuyo principal objetivo es la creación de aplicaciones mediante la unión de distintas partes –componentes– creadas en distintos lenguajes de programación [Kirtland99]. El acceso a los componentes se realiza, al igual que en CORBA, a través de interfaces (*interface*).

Al tratarse de código binario, nativo de una plataforma tras su compilación previa, no puede obtenerse directamente la descripción de cada componente. La información necesaria para obtener introspección, al igual que en CORBA, se añade al propio objeto para que pueda ofrecerla en tiempo de ejecución. La gran diferencia frente a la postura de CORBA es que en COM la descripción del componente la posee él mismo y no un almacén de interfaces externo.

En COM un objeto puede implementar un número indefinido de interfaces, pero ¿cómo podemos conocer y obtener éstos en tiempo de ejecución? ¿Cómo se ofrece la introspección? Mediante la utilización de un *interface* predefinido, denominado `IUnknown`.

Todo *interface* en COM hereda de `IUnknown` y éste obliga a implementar tres métodos: `AddRef`, `Release` y `QueryInterface` [Box99]. Este último método, `QueryInterface`, permite conocer a partir de un objeto el conjunto de interfaces que éste implementa. De esta forma un objeto es capaz de ofrecer información acerca de sí mismo: introspección.

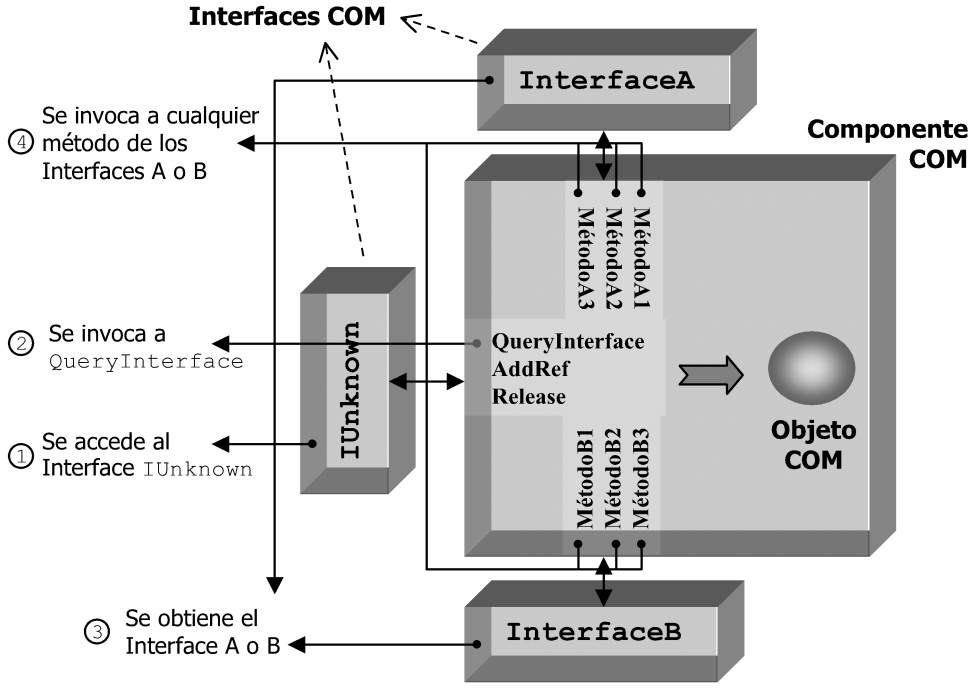


Figura 7.6. Sistema de introspección en el modelo de componentes COM.

El cliente de un componente COM le pide a éste su *interface* por defecto, obteniendo una referencia a un *interface* `IUnknown` –todos los interfaces heredan de éste. Mediante la referencia obtenida, le pregunta al objeto si implementa un determinado *interface* invocando a su método `QueryInterface`. En el caso afirmativo, el cliente consigue una nueva referencia al *interface* deseado, pudiendo pasarle cualquier mensaje que éste especifique.

Microsoft aumentó el sistema de componentes COM haciéndolo distribuido DCOM [Brown98], apoyándose para ello en la introspección descrita. Posteriormente le añadió características propias de sistemas de componentes transaccionales, rebautizándolo como COM+ [Kirtland99b].

7.2 Sistemas Dotados de Reflectividad Estructural

Fijándonos en la clasificación realizada en el capítulo anterior, en § 6.3.1 se define un sistema dotado de la capacidad de acceder y modificar su estructura como estructuralmente reflectivo. Estudiaremos casos reales de este tipo de sistemas y analizaremos sus ventajas e inconvenientes en función de los requisitos definidos en el capítulo 2.

7.2.1 Smalltalk-80

Es uno de los primeros entornos de programación que se han basado en reflectividad estructural. El sistema Smalltalk-80 se puede dividir en dos elementos:

- La imagen virtual, que es una colección de objetos que proporcionan funcionalidades de diversos tipos.
- La máquina virtual, que interpreta la imagen virtual y las aplicaciones de usuario.

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, su descripción, el conjunto de los métodos que posee o incluso la implementación de éstos, podemos utilizar una aplicación desarrollada en el sistema denominada *browser* [Mevel87] (Figura 7.7). El *browser* es una aplicación diseñada en Smalltalk que accede a todos los objetos clase[17] 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 de utilización de la introspección que ofrece el sistema. Gracias a la introspección, se consigue una documentación dinámicamente actualizada de las clases del sistema.

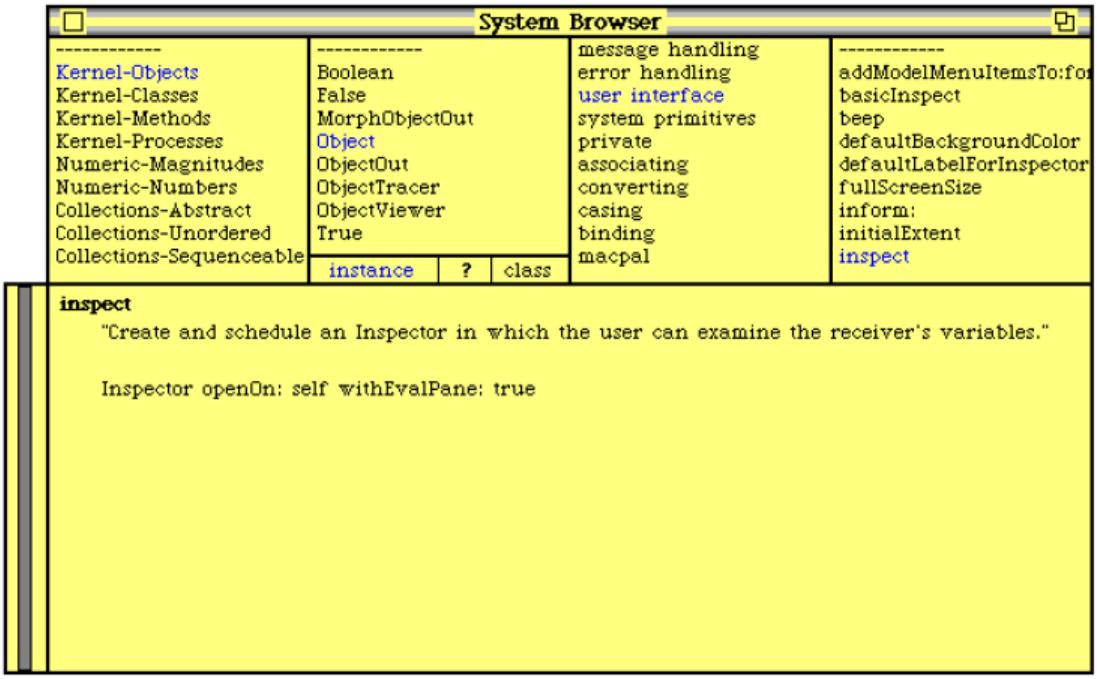


Figura 7.7. Análisis del método *inspect* del objeto de clase *Object*, con la aplicación *Browser* de Smalltalk-80.

De la misma forma que las clases, haciendo uso de sus características reflectivas, los objetos existentes en tiempo de ejecución pueden ser inspeccionados gracias al mensaje `inspect` [Goldberg89]. Mediante esta aplicación también podremos modificar los valores de los distintos atributos de los objetos.

Vemos en la Figura 7.8, cómo 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.

[17] 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`.

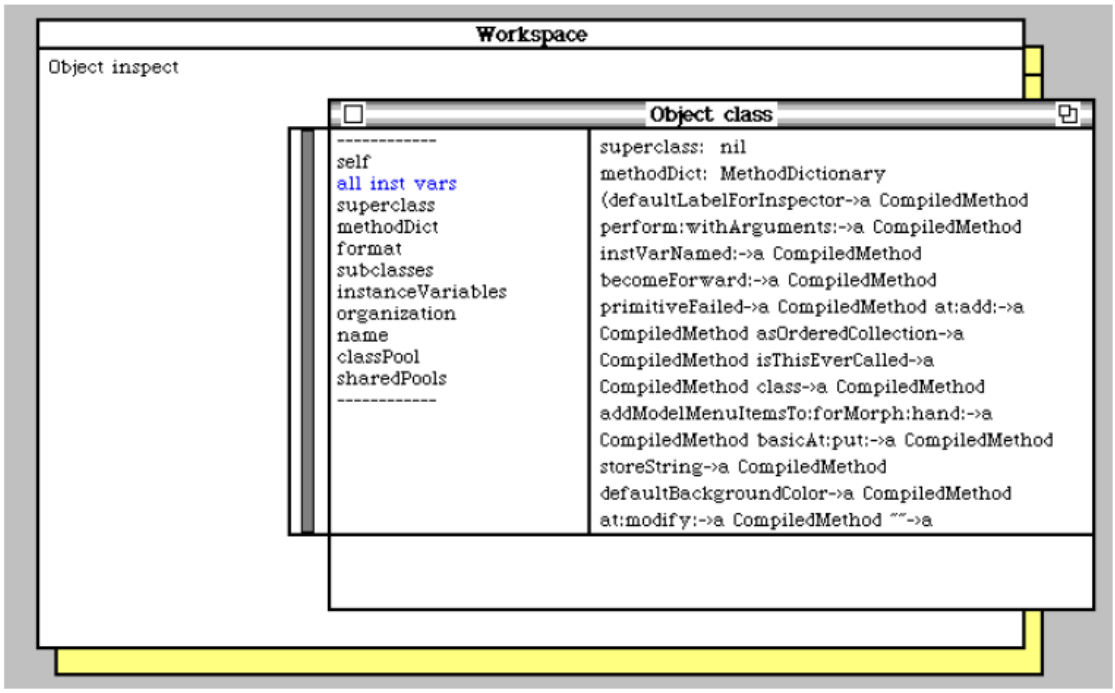


Figura 7.8. Invocando al método inspect del objeto Object desde un espacio de trabajo de Smalltalk-80, obtenemos un acceso a las distintas partes de la instancia.

Lo buscado en el diseño de Smalltalk era obtener un sistema 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 de las clases en el *browser*, se puede depurar ésta 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 en tiempo de ejecución del sistema.

Hemos visto cómo en Smalltalk-80 se utilizaron de forma básica los conceptos de reflectividad estructural e introspección, para obtener un entorno sencillo de manejar y autodocumentado.

La semántica del lenguaje Smalltalk viene dada por una serie de primitivas básicas de computación de la máquina virtual, no modificables por el programador; carece por lo tanto de reflectividad computacional.

7.2.2 Self, Proyecto Merlin

El sistema Self fue construido como una simplificación del sistema Smalltalk y también estaba constituido por una máquina virtual y un entorno de programación basado en el lenguaje Self [Ungar87]. La reducción principal respecto a Smalltalk se centró en la eliminación del concepto de clase, dejando tan sólo la abstracción del objeto. Este tipo de lenguajes orientados a objetos han sido denominados "basados en prototipos".

Este sistema, orientado a objetos puro e interpretado, fue utilizado en el estudio de desarrollo de técnicas de optimización propias de los lenguajes orientados a objetos [Chambers91]. Fueron aplicados métodos de compilación continua y compilación adaptable [Hölzle94], métodos que actualmente han sido implantados a plataformas comerciales como Java [Sun98].

El proyecto Merlin se creó para hacer que los ordenadores fuesen fácilmente utilizados por los humanos, ofreciendo sistemas de persistencia y distribución implícitos [Assumpcao93]. El lenguaje seleccionado para desarrollar este sistema fue Self.

Para conseguir flexibilidad en el sistema, trataron de ampliar la máquina virtual con un conjunto de primitivas de reflectividad (*regions*, *reflectors*, *meta-spaces*) [Cointe92]. La complejidad de ésta creció de tal forma que su portabilidad a distintas plataformas se convirtió en algo inviable [Assumpcao95]. Assumpçao propone en [Assumpcao95] el siguiente conjunto de pasos para construir un sistema portable con capacidades reflectivas:

- Implementar, sobre cualquier lenguaje de programación, un pequeño intérprete de un subconjunto del lenguaje a interpretar. En su caso un intérprete de "tinySelf". El requisito fundamental es que éste tenga reflectividad estructural.

- Sobre lenguaje (tinySelf), se desarrolla un intérprete del lenguaje buscado (Self), con todas sus características.

- Implementamos una interfaz de modificación de las operaciones deseadas. La codificación de un intérprete en su propio lenguaje ofrece total flexibilidad: todo su comportamiento se puede modificar –incluso las primitivas computacionales– puesto que tinySelf posee reflectividad estructural dinámica. Sólo debemos implementar un protocolo de acceso a la modificación de las operaciones deseadas.

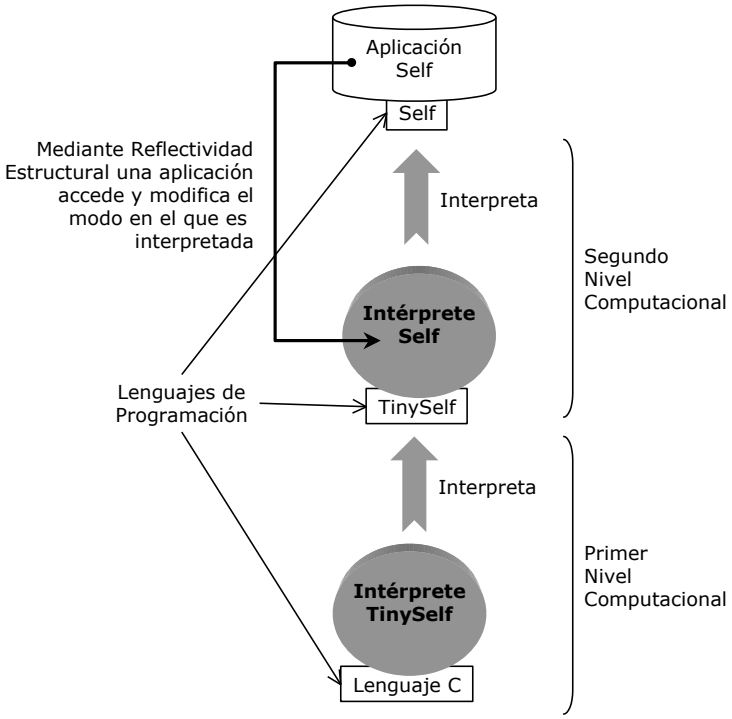


Figura 7.9. Consiguiendo reflectividad mediante la introducción de un nuevo intérprete.

El resultado es un intérprete de Self que permite modificar determinados elementos de su computación, siempre que éstos hayan sido tenidos en cuenta en el desarrollo del intérprete.

El principal problema es la eficiencia del sistema. El desarrollar dos niveles computacionales (intérprete de intérprete) ralentiza la ejecución de la aplicación codificada en Self. Lo que se propone para salvar este obstáculo [Assumpcao95] es implementar un traductor de código tinySelf a otro lenguaje que pueda ser compilado a código nativo. Una vez desarrollado éste, traducir el código propio del intérprete de Self –codificado en el lenguaje tinySelf– a código nativo, reduciendo así un nivel computacional.

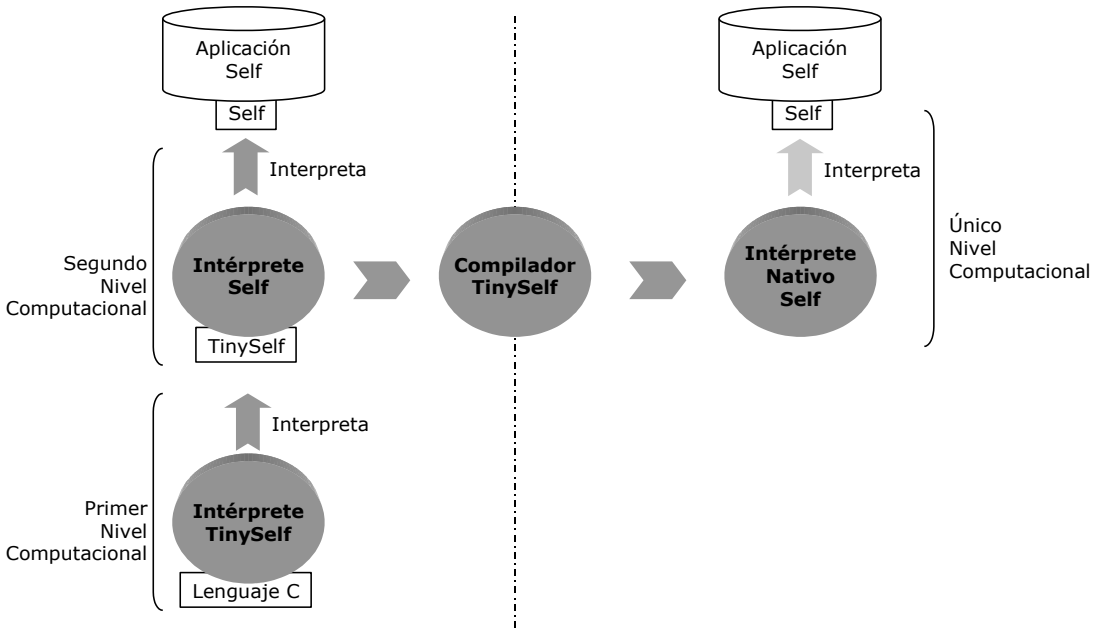


Figura 7.10. Reducción de un nivel de computación en la torre de intérpretes.

El resultado se muestra en la Figura 7.10: un único intérprete de Self en lugar de los dos anteriores. Sin embargo, el producto final tendría dos inconvenientes:

- Una vez que traduzcamos el intérprete, éste no podrá ofrecer la modificación de una característica que no haya sido prevista con anterioridad. Si queremos añadir alguna, deberemos volver al sistema de los dos intérpretes, codificarla, probarla y, cuando no exista ningún error, volver a generar el intérprete nativo.

- La implementación de un traductor de tinySelf a un lenguaje compilado es una tarea difícil, puesto que el código generado deberá estar dotado de la información necesaria para ofrecer reflectividad estructural en tiempo de ejecución. Existen sistemas que implementan estas técnicas como por ejemplo Iguana [Gowing96].

Al mismo tiempo, la ejecución de un código que ofrece información modificable dinámicamente –sus objetos– ocupa más espacio, y añade una ralentización en sus tiempos de ejecución.

7.2.3 ObjVlisp

ObjVlisp es un sistema creado como una evolución de Smalltalk-76 [Ingalls78], desarrollado como un sistema extensible capaz de modificar y ampliar determinadas funcionalidades de los lenguajes orientados a objetos [Cointe88].

El modelo computacional del sistema es definido por la implementación de una máquina virtual en el lenguaje Lisp [Steele90]; ésta está dotada de un conjunto de primitivas que trabajan con la abstracción principal del objeto. El modelo computacional del núcleo del sistema se define con seis postulados [Cointe88]:

- Un objeto está compuesto de un conjunto de miembros que pueden representar información de éste (atributos) y su comportamiento (métodos). La diferencia entre ambos es que los métodos son susceptibles de ser evaluados. Los miembros de un objeto pueden conocerse y modificarse dinámicamente (reflectividad estructural).

- La única forma de llevar a cabo una computación sobre un objeto es enviándole un mensaje indicativo del método que queramos ejecutar.

- Todo objeto ha de pertenecer a una clase que especifique su estructura y comportamiento. Los objetos se crearán dinámicamente como instancias de una clase.

- Una clase es también un objeto instancia de otra clase denominada metaclase.

- Por lo tanto, aplicando el punto 3, una metaclase define la estructura y el comportamiento de sus clases instanciadas. La metaclase inicial primitiva se denomina `CLASS`, y es construida como instancia de sí misma.

- Una clase puede definirse como subclase de otra(s) clase(s). Este mecanismo de herencia permite compartir los miembros de las clases base por sus clases derivadas. El objeto raíz en la jerarquía de herencia se denomina `OBJECT`.

- Los miembros definidos por una clase describen un ámbito global para todos los objetos instanciados a partir de dicha clase; todos sus objetos pueden acceder a dichos miembros.

A partir de los seis puntos anteriores, se deduce que son establecidos, en tiempo de ejecución, dos grafos de objetos en función de dos tipos de asociaciones existentes entre ellos: la asociación "hereda de" –en la que el objeto raíz es `OBJECT`– y la asociación "instancia de" –con objeto raíz `CLASS`. Un ejemplo de ambos grafos se muestra en la siguiente figura:

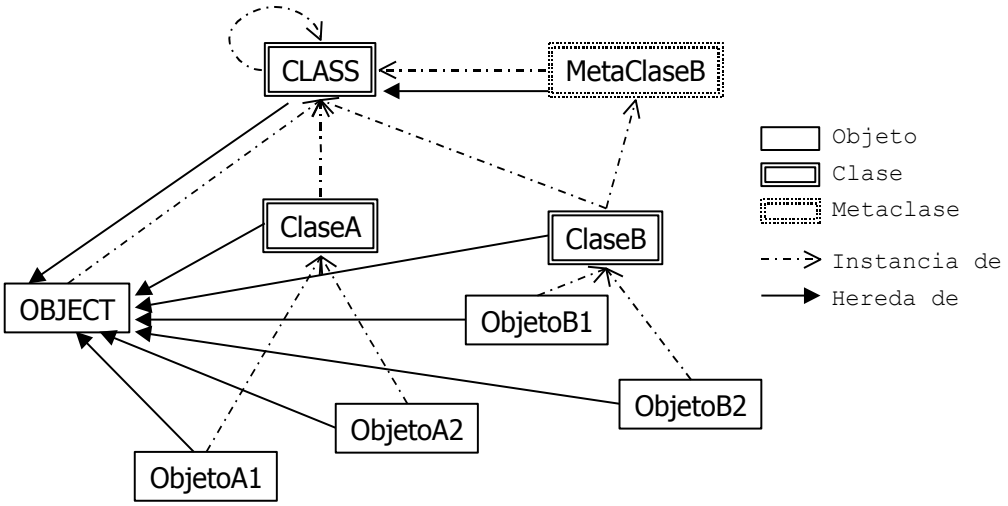


Figura 7.11. Doble grafo de objetos en el sistema ObjVlisp.

La plataforma permite extender el sistema computacional gracias a la creación dinámica de objetos y a la extensión de los existentes. Los nuevos objetos ofrecen un mayor nivel de abstracción al usuario del sistema. Esto es una consecuencia directa de aplicar la reflectividad estructural en tiempo de ejecución.

El sistema soporta un mecanismo de adaptabilidad basado en el concepto de metaclase. Si queremos modificar el comportamiento de los objetos instancia de una clase, podemos crear una nueva metaclase y establecer una asociación entre la clase y ésta, mediante la relación "es instancia de". El resultado es la modificación de parte de la semántica de los objetos de dicha clase. En la Figura 7.11, la semántica de los objetos instancia de la clase B es definida por su metaclase B.

Las distintas modificaciones que puede hacer una metaclase en el funcionamiento de los objetos de una clase son:

- Modificación del funcionamiento del mecanismo de herencia.
- Alteración de la forma en la que se crean los objetos, implementando un nuevo comportamiento de la primitiva `make-object`.
- Cambio de la acción llevada a cabo en la recepción de un mensaje.
- Utilización de otro tipo de contenedor para coleccionar los miembros de un objeto.
- Modificación del acceso a los miembros, estableciendo un mecanismo de ocultación de la información.

7.2.4 Linguistic Reflection in Java

Para Graham Kirby y Ron Morrison *Linguistic Reflection* es la capacidad de un programa para, en tiempo de ejecución, generar nuevos fragmentos de código e integrarlos en su entorno de ejecución[18] [Kirby98]. Implementaron un entorno de trabajo en el que añadían al lenguaje de programación Java esta característica.

La capacidad de un sistema reflectivo para adaptarse a un contexto en tiempo de ejecución es opuesta al concepto de tipo, que limita, en tiempo de compilación, el conjunto de operaciones aplicables a un objeto [Cardelli97]. El objetivo del prototipo implementado es ofrecer la generación dinámica de código sin perder el sistema de tipos del lenguaje Java [Gosling96].

Como se muestra en la Figura 7.12, los pasos llevados a cabo en la evaluación dinámica de código son:

- La aplicación inicial es ejecutada por una implementación de la máquina virtual de Java.
- La aplicación genera código dinámicamente en función de los requisitos propios de su contexto de ejecución.
- Este nuevo fragmento de código es compilado a código binario de la máquina virtual, mediante un compilador de Java codificado en Java. Es-

[18] El concepto definido como "*linguistic reflection*" no es el mismo que definíamos en § 6.3.1 como "reflectividad lingüística".

te proceso se realiza con la comprobación estática de tipos propia del lenguaje.

- La implementación de una clase derivada de la clase `ClassLoader` [Gosling96] permite cargar dinámicamente el código generado, para pasar a formar parte del entorno computacional de la aplicación inicial.

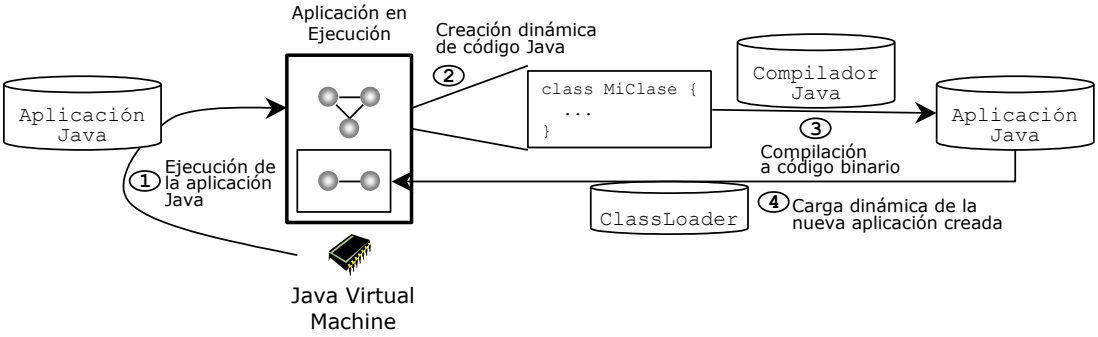


Figura 7.12. Evaluación dinámica de código creado en tiempo de ejecución.

La aportación principal del sistema es soportar la codificación de aplicaciones genéricas en tiempo de ejecución y seguras respecto al tipo. Éste es un concepto similar a las plantillas (*templates*) del lenguaje C++ [Stroustrup98], pero resueltas dinámicamente en lugar de determinar su tipo en tiempo de compilación. Como caso práctico, se han implementado productos cartesianos de tablas de una base de datos relacional, así como contenedores de objetos seguros respecto al tipo, ambos genéricos dinámicamente.

En lo referente a reflectividad, el sistema no aporta nada frente a la introspección ofrecida por la plataforma Java [Sun97b]. Sin embargo, la posibilidad de crear código dinámicamente ofrece facilidades propias de plataformas dotadas de reflectividad estructural como Smalltalk. La implementación mediante un intérprete puro [Cueva98] hubiese mucho más sencilla, pero menos eficiente en tiempo de ejecución.

7.2.5 Python

Python es un lenguaje de programación orientado a objetos, portable e interpretado [Rossum2001]. Su desarrollo comenzó en 1990 en el CWI de Ámsterdam. El lenguaje posee módulos, paquetes, clases, excepciones y un sistema de comprobación de tipos dinámico. La implementación del intérprete de Python ha sido portada a diferentes plataformas como UNIX, Windows, DOS, OS/2, Mac y Amiga.

El entorno de programación de Python ofrece en tiempo de ejecución determinadas funcionalidades de reflectividad estructural [Andersen98]:

- Modificación dinámica de la clase de un objeto. Todo objeto posee un atributo denominado `__class__` que hace referencia a su clase (las clases también son objetos). La modificación de esta referencia implica la modificación del tipo del objeto.

- Acceso al árbol de herencia. Toda clase posee un atributo `__bases__` que referencia una colección de sus clases base modificables dinámicamente.

- Acceso a los atributos y métodos de un objeto. Tanto los objetos como las clases poseen un diccionario de sus miembros (`__dict__`) que puede ser consultado y modificado dinámicamente.

- Control de acceso a los atributos. El acceso a los atributos de una clase puede ser modificado con la definición de los métodos de clases `__getattr__` y `__setattr__`.

Además, el intérprete del lenguaje ofrece numerosa información del sistema en tiempo de ejecución: introspección. Sobre la reflectividad estructural del lenguaje se han construido módulos que aumentan las características reflectivas del entorno de programación [Andersen98], ofreciendo un mayor nivel de abstracción basándose en el concepto de metaobjeto [Kiczales91].

7.3 Reflectividad en Tiempo de Compilación

Basándonos en la clasificación realizada en el capítulo anterior, la reflectividad en tiempo de compilación se produce en fase de traducción independientemente de lo que el sistema permita reflejar. En este tipo de sistemas una aplicación puede adaptarse a un contexto dinámicamente, siempre y cuando los cambios hayan sido tenidos en cuenta en su código fuente –puesto que toda su información se genera tiempo de compilación. Si en tiempo de ejecución aparecieren exigencias no previstas en fase de desarrollo del sistema, éstas no podrían solventarse dinámicamente; debería recodificarse y recompilarse el sistema.

7.3.1 OpenC++

OpenC++ [Chiba95] es un lenguaje de programación que ofrece características reflectivas a los programadores de C++. Su característica principal es la eficiencia dinámica de las aplicaciones creadas: no existe una sobrecarga computacional en tiempo de ejecución. Está enfocado a amoldar el lenguaje de programación y su semántica al problema para poder desarrollar las aplicaciones a su nivel de abstracción adecuado.

El modo en el que son generadas las aplicaciones se centra en un preproceso de código fuente. Una aplicación se codifica en OpenC++ pudiendo hacer uso de sus características reflectivas. Ésta es traducida a código C++ que, tras compilarse mediante cualquier compilador estándar, genera la aplicación final –adaptable en el grado estipulado cuando fue codificada.

La arquitectura presenta el concepto de metaobjeto –popularizado por Gregor Kiczales [Kiczales91]– para modificar la semántica determinados elementos del lenguaje. En concreto, para las clases y funciones miembro, se puede modificar el modo en el que éstas son traducidas al código C++ final.

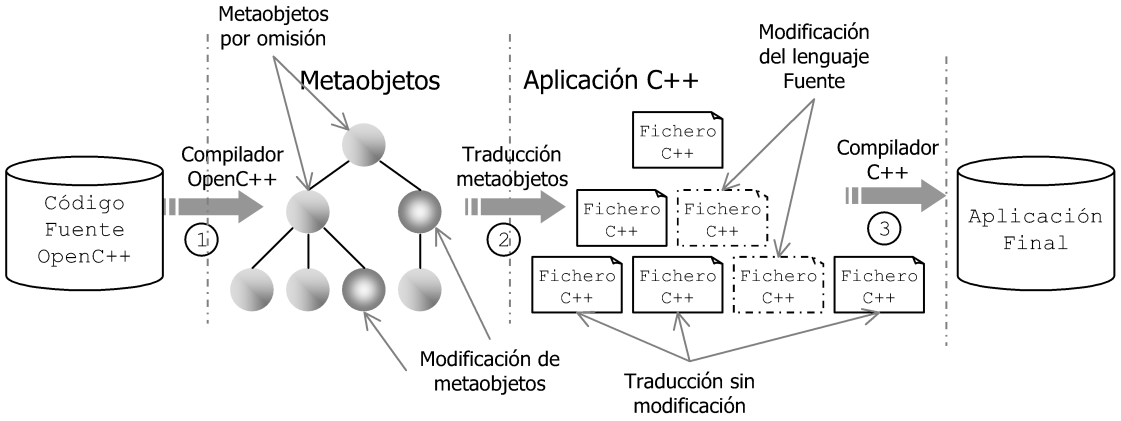


Figura 7.13. Fases de compilación de código fuente OpenC++.

Tomando código OpenC++ como entrada, el traductor de este lenguaje genera C++ estándar como salida. Este proceso consta de 2 fases mostradas en la Figura 7.13. En la primera fase, se crea el árbol sintáctico relativo a las clases y funciones miembros de la aplicación, para crearse un metaobjeto por cada uno de dichos elementos sintácticos.

Por omisión, los metaobjetos son instancias de una clase `Class` (si el elemento sintáctico es una clase), o bien de una clase `Function` (cuando se trate de un método). Estas clases traducen el código sin añadir ninguna modificación. En el caso de que una clase o un método hayan sido definidos mediante un metaobjeto en OpenC++, éstos serán instancias de clases derivadas de las dos mencionadas anteriormente; dichas clases definirán la forma en la que se traducirá el código fuente asociado a los metaobjetos. Ésta es la segunda fase de traducción que concluye con la generación de la aplicación final por un compilador de C++.

El sistema ofrece un mecanismo de preproceso para poder amoldar el lenguaje de programación C++ [Stroustrup98] a las necesidades del programador. Además de poder modificar parte de la semántica de métodos y clases (reflectividad computacional), OpenC++ ofrece la posibilidad de añadir palabras reservadas al lenguaje, para implementar modificadores de tipos, clases y el operador `new` (reflectividad lingüística).

7.3.2 OpenJava

Una vez analizadas y estudiadas las técnicas y ventajas en la utilización de reflectividad en tiempo de compilación, Shigeru Chiba aumentó las características reflectivas del lenguaje de programación Java, denominándose éste OpenJava [Chiba98].

El lenguaje Java ofrece introspección por medio de la clase `java.lang.Class` [Sun97b], entre otras. Mediante la creación de una nueva clase, `OJClass`, y el preproceso de código OpenJava realizado por el *OpenJava compiler*, diseñaron un lenguaje capaz de ofrecer reflectividad en tiempo de compilación de las siguientes características:

- Modificación del comportamiento de determinadas operaciones sobre los objetos, como invocación a métodos o acceso a sus atributos (reflectividad computacional restringida).

- Reflectividad estructural. A la introspección ofrecida por la plataforma de Java (§ 7.1), se añaden a los objetos y clases funcionalidades propias de reflectividad estructural (§ 7.2).

- Modificación de la sintaxis del lenguaje. Pueden crearse nuevas instrucciones, operadores y palabras reservadas (reflectividad lingüística).

- Modificación de los aspectos semánticos del lenguaje. Es posible modificar la promoción o coerción de tipos [Ortín2004b].

OpenJava mejora las posibilidades ofrecidas por su hermano OpenC++, sin necesidad de modificar la implementación de la máquina abstracta –como sucede, por ejemplo, en el caso de MetaXa [Kleinöder96]. Esto supone dos ventajas:

- Al no modificarse la máquina virtual, no se generan distintas versiones de ésta con la consecuente pérdida de portabilidad de su código fuente [Ortín2001].

- Al no ofrecer adaptabilidad dinámica, no supone una pérdida de eficiencia en tiempo de ejecución.

El lenguaje de programación OpenJava se ha utilizado para describir la solución de problemas mediante patrones de diseño [GOF94] en el nivel de abstracción adecuado [Tatsubori98]; el lenguaje se modificó para amoldar su sintaxis a los patrones *Adapter* y *Visitor* [GOF94].

7.3.3 Java Dynamic Proxy Classes

En la edición estándar de la plataforma Java2 (*Java Standard Edition*) versión 1.2.3 se ha aumentado el paquete `java.lang.reflect`, para ofrecer un mecanismo de modificación del paso de mensajes de una determinada clase; este tipo de clase se denomina *proxy* (apoderadas) [Sun99].

Una clase *proxy* especifica la implementación de un conjunto ordenado de interfaces. La clase es creada dinámicamente especificando su manejador de invocaciones (`InvocationHandler`). Cada vez que se invoque a un método de las interfaces implementadas, la clase apoderada ejecutará el método `invoke` de su manejador; éste podrá modificar su semántica en función del contexto dinámico existente. El diagrama de clases que ofrece este marco de trabajo se muestra a continuación:

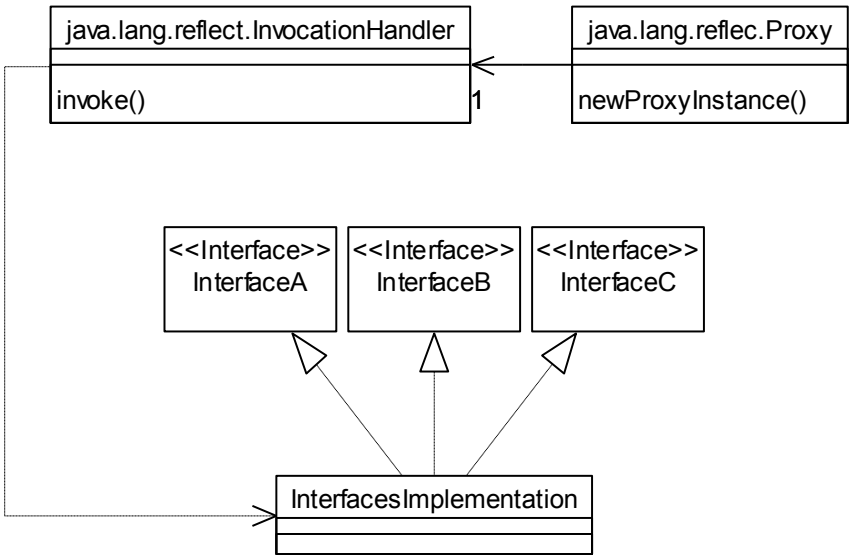


Figura 7.14. Diagrama de clases de la implementación de un manejador de invocaciones.

La clase es creada dinámicamente especificando las interfaces a implementar, un cargador de clases (`ClassLoader` [Gosling96]) y una implementación del manejador de invocaciones (`InvocationHandler`); esta funcionalidad la ofrece el método de clase `newProxyInstance` de la clase Proxy.

La adaptabilidad del sistema se centra en la implementación del método `invoke` del manejador. Este método recibe el objeto real al que se ha pasado el mensaje, el identificador del método y sus parámetros. Haciendo uso del sistema introspectivo de la plataforma Java [Sun97b], podremos conocer e invocar el método que deseemos en función de los requisitos existentes en tiempo de ejecución.

Con este mecanismo, la plataforma de Java ofrece un grado de flexibilidad superior a la introspección existente en versiones anteriores; permite modificar la semántica del paso de mensajes. Sin embargo, las distintas interpretaciones de dicha semántica han de ser especificadas en tiempo de compilación, para ser seleccionadas posteriormente de forma dinámica.

El resultado es un conjunto de clases que, haciendo uso de la introspección de la plataforma, permite simular la modificación de una parte del comportamiento del sistema. No obstante no estamos en un grado computacional de reflectividad, al no permitir la modificación de dicho comportamiento para todo el sistema –se modifica tan solo para los objetos instancia de una clase.

7.4 Reflectividad Computacional basada en Meta-Object Protocols

Los sistemas reflectivos que permiten modificar parte de su semántica en tiempo de ejecución son comúnmente implementados utilizando el concepto de protocolo de metaobjeto (*Meta-Object Protocol*, MOP). Estos MOPs definen un modo de acceso (protocolo) del sistema base al metasistema, permitiendo modificar parte de su propio comportamiento dinámicamente.

En este punto analizaremos brevemente un conjunto de sistemas que utilizan MOPs para modificar su semántica dinámicamente, para posteriormente hacer una síntesis global de sus aportaciones y carencias.

7.4.1 Closette

Closette [Kiczales91] es un subconjunto del lenguaje de programación CLOS [Steele90]. Fue creado para implementar un MOP del lenguaje CLOS, permitiendo modificar dinámicamente aspectos semánticos y estructurales de este lenguaje. La implementación se basó en desarrollar un intérprete de este subconjunto de CLOS sobre el propio lenguaje CLOS, capaz de ofrecer un protocolo de acceso a su metasistema.

Existen dos niveles computacionales: el nivel del intérprete de CLOS (metasistema), y el del intérprete de Closette (sistema base) desarrollado sobre el primero. El acceso del sistema base al metasistema se realiza mediante un sistema de macros; el código Closette se expande a código CLOS que, al ser evaluado, puede acceder a su metasistema. La definición de la interfaz de estas macros constituye el MOP del sistema.

El diseño de este MOP para el lenguaje CLOS se centra en el concepto de metaobjeto. Un metaobjeto es cualquier abstracción, estructural o computacional, del metasistema susceptible de ser modificada por su sistema base. Un metaobjeto no es necesariamente una representación de un objeto en su sistema base; puede representar una clase o la semántica de la invocación a un método. El modo en el que se puede llevar a cabo la modificación de los metaobjetos es definido por el MOP.

La implementación del sistema fue llevada a cabo en tres capas, mostradas en la Figura 7.15:

1. La capa de macros; define la forma en la que se va a interpretar el subconjunto de CLOS definido, estableciéndolo mediante traducciones a código CLOS. En el caso de que no existiese un MOP, esta traducción sería la identidad; el código Closette se traduciría a código CLOS sin ningún cambio.
2. La capa de "pegamento" (*glue*). Son un conjunto de funciones desarrolladas en CLOS que facilitan el acceso a los objetos del metasistema, para así facilitar la implementación de la traducción de las macros. Como ejemplo, podemos citar la función `find-class` que obtiene el metaobjeto representativo de una clase, cuyo identificador es pasado como parámetro.
3. La capa de nivel inferior. Es la representación en CLOS (metasistema) de la estructura y comportamiento del sistema base. Todas aquellas partes del sistema que deseen ser reflectivas, deberán ser implementadas como metaobjetos.

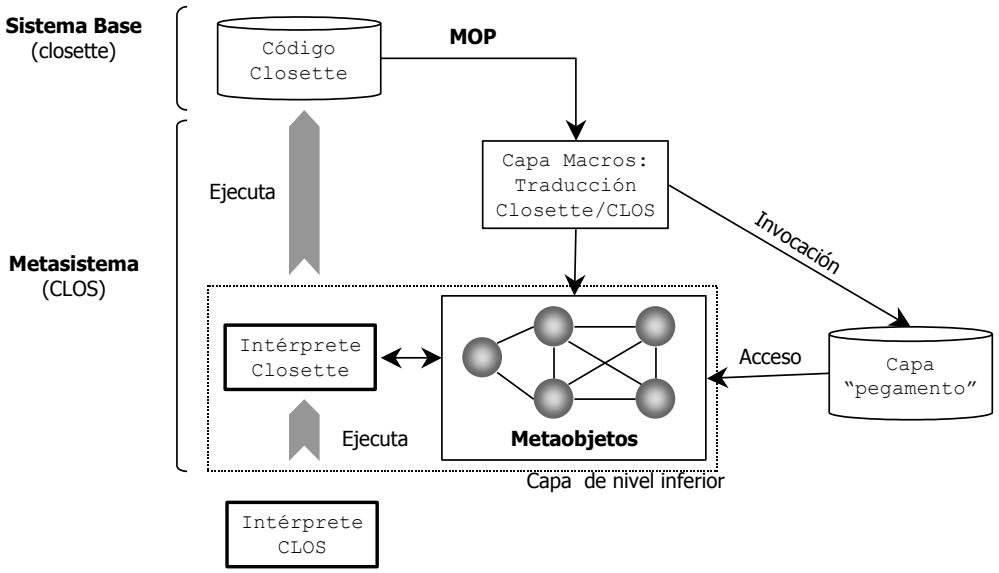


Figura 7.15. Arquitectura del MOP desarrollado para el lenguaje CLOS.

7.4.2 MetaXa

MetaXa[19] es un sistema basado en el lenguaje y plataforma Java que añade a éste características reflectivas en tiempo de ejecución, permitiendo modificar parte de la semántica de la implementación de la máquina virtual [Golm97]. El diseño de la plataforma está orientado a dar soporte a la demanda de creación de aplicaciones flexibles, que puedan adaptarse a requerimientos dinámicos como distribución, seguridad, persistencia, tolerancia a fallos o sincronización de tareas [Kleinöder96].

El modo en el que se deben desarrollar aplicaciones en MetaXa se centra en el concepto de metaprogramación (*meta-programming*) [Maes87]: separación del código funcional del código no funcional. La parte funcional de una aplicación se centra en el modelado del dominio de la aplicación (nivel base), mientras que el código no funcional formaliza la supervisión o modelado de determinados aspectos propios código funcional (metasistema). MetaXa permite separar estos dos niveles de código fuente y establecer entre ellos un mecanismo de conexión causal en tiempo de ejecución.

El sistema de computación de MetaXa se apoya sobre la implementación de objetos funcionales y metaobjetos conectados a los primeros –a un metaobjeto se le puede conectar objetos, referencias y clases; de forma genérica utilizaremos el término objeto. Cuando un metaobjeto está conectado a un objeto sobre el que sucede una acción, el sistema provoca un evento en el metaobjeto indicándole la operación solicitada en el sistema base. La implementación del metasistema permite modificar la semántica de la acción provocadora del evento. La computación del sistema base es suspendida de forma síncrona, hasta que el metasistema finalice la interpretación del evento capturado.

[19] Anteriormente conocido como MetaJava.

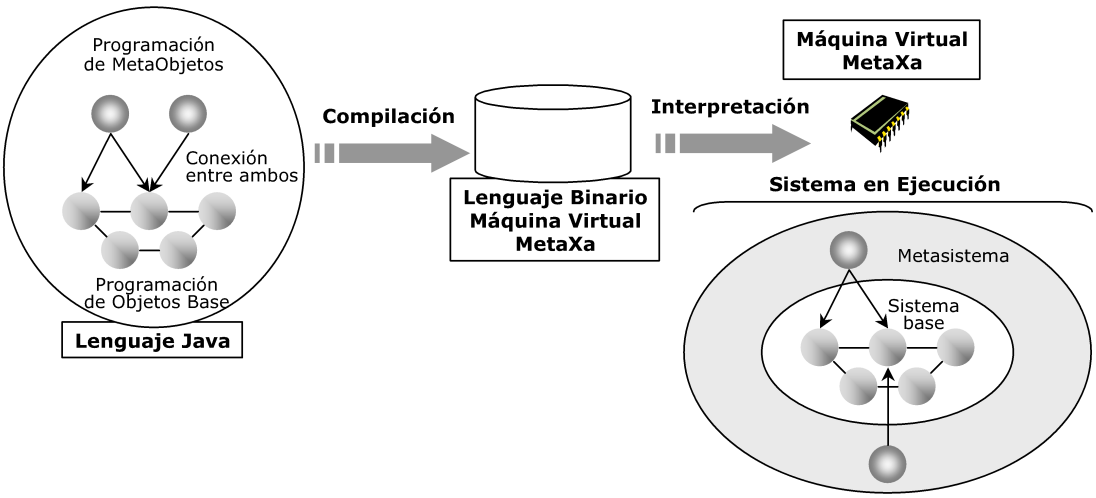


Figura 7.16. Fases en la creación de una aplicación en MetaXa.

Si un metaobjeto recibe el evento `method-enter`, la acción por omisión será ejecutar el método apropiado. Sin embargo, el metaobjeto puede sobrescribir el método `eventMethodEnter` para modificar el modo en el que se interpreta la recepción de un mensaje.

En MetaXa, para hacer el sistema lo más eficiente posible, inicialmente ningún objeto está conectado a un metaobjeto. En tiempo de ejecución se producen las conexiones apropiadas para modificar los comportamientos solicitados por el programador.

La implementación de la plataforma reflectiva de MetaXa toma la máquina virtual de Java [Sun95] y añade sus nuevas características reflectivas mediante la implementación de métodos nativos residentes en una librería de enlace dinámico [Sun97].

Uno de los principales inconvenientes del diseño abordado es el de la modificación del comportamiento individual de un objeto. Al residir el comportamiento de todo objeto en su clase, ¿qué sucede si deseamos modificar la semántica de una sola instancia de dicha clase? MetaXa crea una nueva clase para el objeto denominada clase sombra (*Shadow Class*) con las siguientes características [Golm97c]:

- Una clase y su clase sombra asociada son iguales para el nivel base.
- La clase sombra difiere de la original en las modificaciones realizadas en el metasistema.
- Los métodos y atributos de clase (`static`) son compartidos por ambas clases.

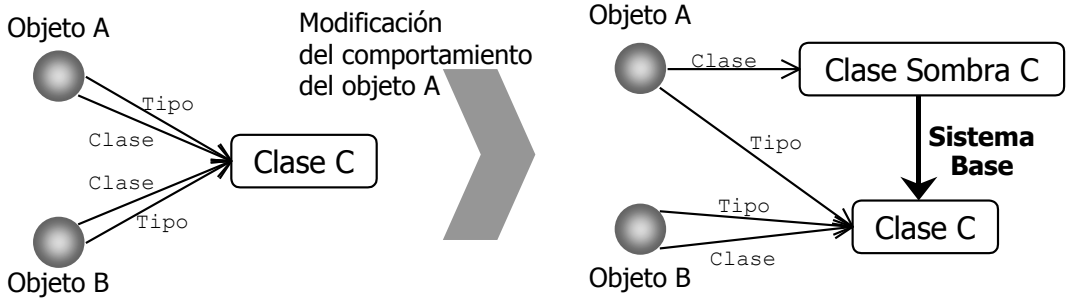


Figura 7.17: Creación dinámica de una clase sombra en MetaXa para modificar el comportamiento.

Hay que solventar un conjunto de problemas para mantener coherente el sistema en la utilización de las clases sombra [Golm97c]:

- Consistencia de atributos. La modificación de los atributos de una clase ha de mantenerse coherente con su representación sombra.
- Identidad de clase. Hay que tener en cuenta que el tipo de una clase y el de su clase sombra han de ser el mismo, aunque tengan distinta identidad.
- Objetos de la clase. Cuando el sistema utilice el objeto representante de una clase (en Java una clase genera un objeto en tiempo de ejecución) han de tratarse paralelamente el de la clase original y el de su sombra. Un ejemplo puede ser las operaciones `monitorenter` y `monitorexit` de la máquina virtual [Venners98], que tratan los objetos clase como monitores de sincronización; ha de ser el mismo monitor el de la clase sombra que el de la real.
- Recolector de basura. Las clases sombra deberá limpiarse cuando el objeto no tenga asociado un metaobjeto.
- Consistencia de código. Tendrá que ser mantenida cuando se produzca la creación de una clase sombra a la hora de estar ejecutándose un método de la clase original.
- Consumo de memoria. La duplicidad de una clase produce elevados consumos de memoria; deberán idearse técnicas para reducir estos consumos.
- Herencia. La creación de una clase sombra, cuando la clase inicial es derivada de otra, obliga a la producción de otra clase sombra de la clase base original. Este proceso ha de expandirse de forma recursiva.

La complejidad surgida por el concepto de clase en un sistema reflectivo hace afirmar a los propios autores del sistema, cómo los lenguajes basados en prototipos como Self [Ungar87] o Moostrap solucionan estos problemas de una forma más coherente [Golm97c].

7.4.3 Iguana

La mayoría de los sistemas reflectivos, adaptables en tiempo de ejecución y basados en MOPs, son desarrollados mediante la interpretación de código; la ejecución de código nativo no suele ser común en este tipo de entornos. La principal razón es que los intérpretes tienen que construir toda la información propia de la estructura y la semántica de la aplicación a la hora de ejecutar ésta. Si un entorno reflectivo trata de modificar la estructura o comportamiento de un sistema, será más sencillo ofrecer esta información si la ejecución de la aplicación es interpretada.

En el caso de los compiladores, la información relativa al sistema es creada en tiempo de compilación –y generalmente almacenada en la tabla de símbolos [Cueva92]– para llevar a cabo todo tipo de comprobación de tipos [Ortín2004b] y generación de código [Aho90]; una vez compilada la aplicación, dicha información deja de existir. En el caso de un depurador (*debugger*), parte de la información es mantenida en tiempo de ejecución para poder conocer el estado de computación

(introspección) y permitir modificar su estructura (reflectividad estructural dinámica). El precio a pagar en este caso es un aumento de tamaño de la aplicación, y una ralentización de su ejecución.

El sistema Iguana ofrece reflectividad computacional en tiempo de ejecución basada en un MOP, compilando código C++ a la plataforma nativa destino [Gowing96]. En la generación de código, de forma contraria a un depurador, Iguana no genera información de toda la estructura y comportamiento del sistema. Por omisión, compila el código origen C++ a la plataforma destino sin ningún tipo de información dinámica. El programador ha de especificar qué parte del sistema desea que sea adaptable en tiempo de ejecución, de modo que el sistema generará el código oportuno para que sea reflectivo.

Iguana define dos conceptos para especificar el grado de adaptabilidad de una aplicación:

Categorías de cosificación (*Reification Categories*). Indican al compilador dónde debe producirse la cosificación del sistema. Son elementos susceptibles de ser adaptados en Iguana; ejemplos son clases, métodos, objetos, creación y destrucción de objetos, invocación a métodos o recepción de mensajes, entre otros.

Definición múltiple de MOPs (*Multiple fine-grained MOPs*). El programador ha de definir la forma en la que el sistema base va a acceder a su información dinámica, es decir se ha de especificar el MOP de acceso al metasistema.

La implementación de Iguana está basada en el desarrollo de un preprocesador que lee el código fuente Iguana –una extensión del C++– y traduce toda la información especifica del MOP, a código C++ con información dinámica adicional adaptable en tiempo de ejecución (el código C++ no reflectivo no sufre proceso de traducción alguno). Una vez que la fase de preproceso haya sido completada, Iguana invocará a un compilador de C++ para generar la aplicación final nativa, adaptable dinámicamente.

7.4.4 Cognac

Cognac [Murata94] es un sistema orientado a objetos basado en clases, cuya intención es proporcionar un entorno de programación de sistemas operativos orientados a objetos como Apertos [Yokote92]. El lenguaje de programación de Cognac es intencionalmente similar a Smalltalk-80 [Goldberg89]; para aumentar su eficiencia, se le ha añadido comprobación estática de tipos [Cardelli97].

Los principales objetivos del sistema son:

- Uniformidad y simplicidad. Para el programador sólo debe haber un tipo de objeto concurrente, sin diferenciar ejecución síncrona de asíncrona.
- Eficiencia. Necesaria para desarrollar un sistema operativo.
- Seguridad. Deberá tratarse de minimizar el número de errores en tiempo de ejecución; de ahí la introducción de tipos estáticos al lenguaje.
- Migración. En el sistema los objetos deberán poder moverse de una plataforma física a otra, para seleccionar el entorno de ejecución que más les convenga.

- Metaprogramación. El sistema podrá programarse separando las distintas incumbencias y aspectos de las aplicaciones, diferenciando entre el código funcional del no funcional.

La arquitectura del sistema está compuesta de cinco elementos:

- El *front-end* del compilador. El código fuente Cognac es traducido a un código intermedio independiente de la plataforma destino seleccionada. El compilador selecciona la información propia de las clases y la almacena, para su posterior uso, en el sistema de clases (quinto elemento).

- El *back-end* del compilador. Esta parte del compilador toma el código intermedio y lo traduce a código binario propio de la plataforma física utilizada. Crea un conjunto de funciones traducidas de cada una de las rutinas del lenguaje de alto nivel.

- Librería de soporte dinámico (*Run-time Support Library)*. Es el motor principal de ejecución. Envía una petición al sistema de clases para conocer el método apropiado del objeto implícito a invocar; una vez identificado éste en el código nativo, carga la función apropiada y la ejecuta.

- Intérprete. Aplicación nativa capaz de ejecutar el código intermedio de la aplicación. Será utilizado cuando el sistema requiera reflejarse dinámicamente. La ejecución del código en este modo se ralentiza frente a la ejecución nativa.

- Sistema de clases. Información dinámica relativa a las clases y métodos del sistema.

El proceso de reflexión dinámica y los papeles de las distintas partes del sistema se muestran en la Figura 7.17. La ejecución del sistema es controlada por la librería de soporte dinámico, que lee la información relativa al mensaje solicitado, busca éste entre el código nativo y lo ejecuta. Cuando se utiliza un metaobjeto dinámicamente, el motor de ejecución pasa a ser el intérprete, que ejecuta el código intermedio de dicho metaobjeto. El comportamiento del sistema es derogado por la interpretación del metaobjeto creado; éste dicta la nueva semántica del sistema.

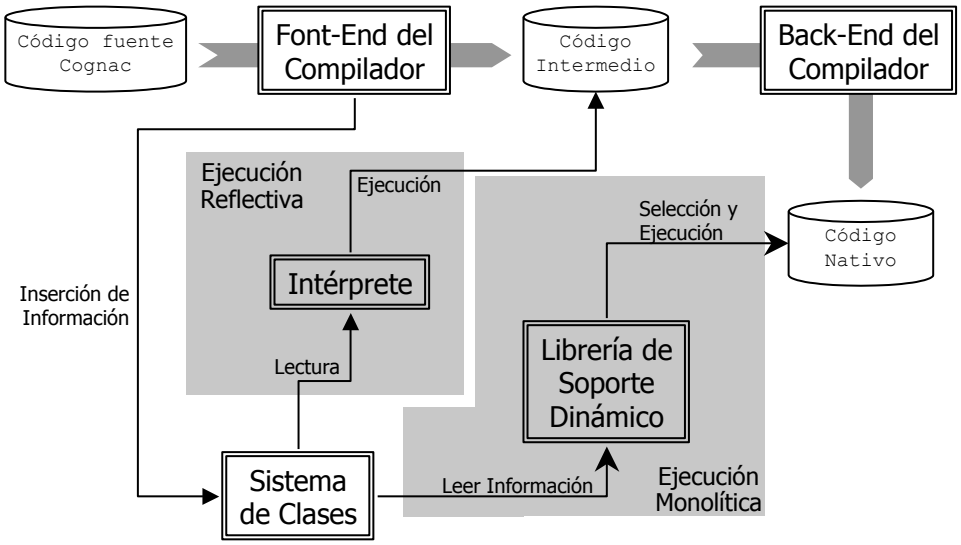


Figura 7.17. Arquitectura y ejecución de una aplicación en Cognac.

7.4.5 Guanará

Para el desarrollo de la librería MOLDS [Oliva98] de componentes de metasistema reusables, enfocados a la creación de aplicaciones de naturaleza distribuida –ofreciendo características de persistencia, distribución y replicación, indiferentemente de la aplicación desarrollada (separación de incumbencias)–, se desarrolló la plataforma computacionalmente reflectiva Guanará [Oliva98b]. Guanará es una ampliación de la implementación de la máquina virtual de Java, Kaffe OpenVM, otorgándole la capacidad de ser reflectiva computacionalmente en tiempo de ejecución mediante un MOP [Oliva98c].

El mecanismo de reflectividad ofrecido al programador está centrado en el concepto de metaobjeto. Cuando se enlaza un metaobjeto a una operación del sistema, la semántica de dicha operación es derogada por la evaluación del metaobjeto. El modo en el que sea desarrollado este metaobjeto definirá dinámicamente el nuevo comportamiento de la operación modificada.

El concepto de metaobjeto es utilizado por multitud de MOPs y, la combinación de éstos, se suele llevar a cabo mediante el patrón de diseño "Cadena de Responsabilidad" (*Chain of Responsibility*) [GOF94]: cada metaobjeto es responsable de invocar al siguiente metaobjeto enlazado con su misma operación, y devolver el resultado de éste. Este esquema de funcionamiento es poco flexible y todo metaobjeto necesita modificar su comportamiento en función del resto de metaobjetos (Figura 7.18).

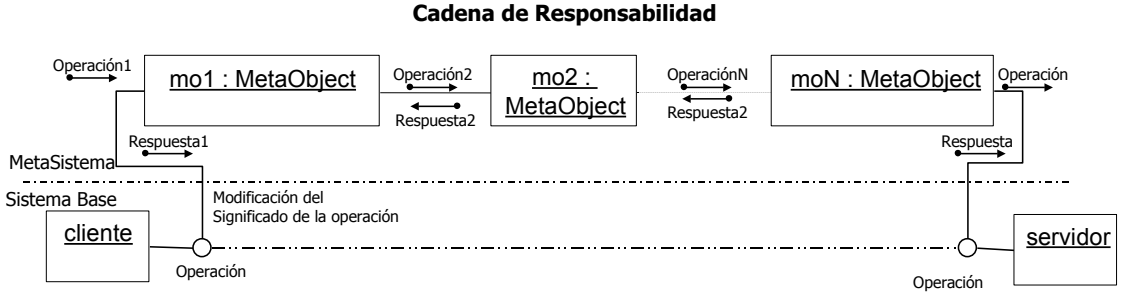


Figura 7.18. Utilización del patrón Chain of Responsibility para asociar múltiples metaobjetos.

Guanará aborda este problema con el uso del patrón "Composición" (*Composite*) [GOF94], permitiendo establecer combinaciones de comportamiento más independientes y flexibles [Oliva99]. Cada operación puede tener enlazado un único metaobjeto primario, denominado compositor (*Composer*). Éste, haciendo uso del patrón "Composición", podrá acceder a una colección de metaobjetos mediante una estructura de grafo. Como se muestra en la Figura 7.19 cada uno de los elementos de un compositor puede ser un metaobjeto u otro compositor.

El modo en el que cada compositor establece el nuevo comportamiento de su operación asociada en función de los metaobjetos utilizados, viene definido por una configuración adicional denominada metaconfiguración. De esta forma, separamos la estructura de los metaobjetos de su secuencia de evaluación, haciendo el sistema más flexible y reutilizable.

La parte realmente novedosa de este sistema radica en la forma en la que se pueden combinar múltiples comportamientos a una operación reflectiva, así como el establecimiento de metacomportamientos de un metaobjeto (metametaobjetos). Establece un mecanismo escalable y flexible, basado en el patrón de diseño *Composite*.

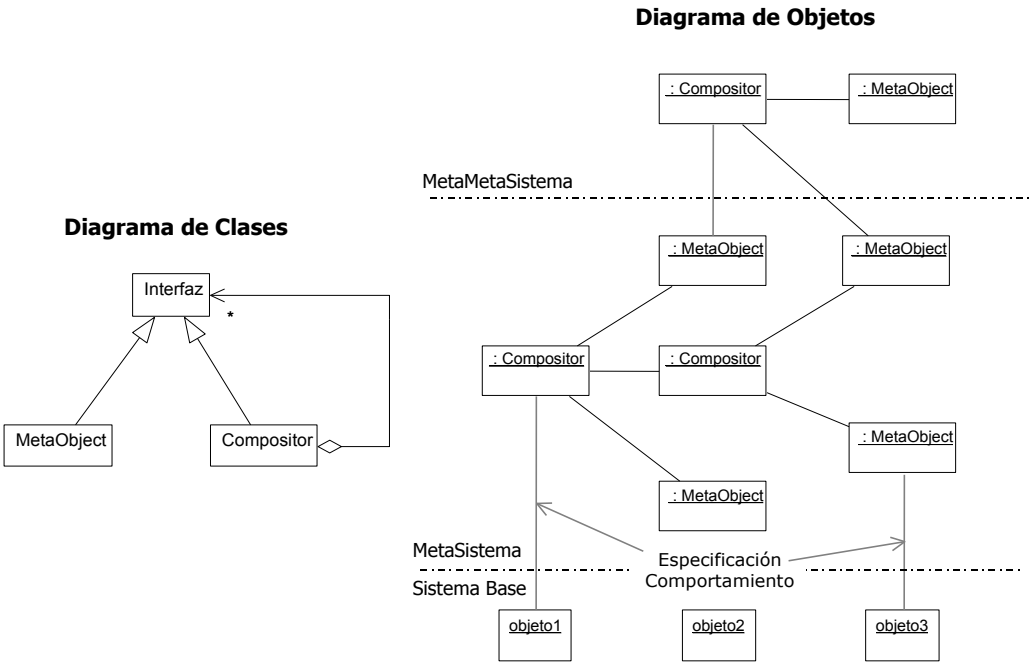


Figura 7.19. Utilización del patrón Composite para asociar múltiples metaobjetos.

7.4.6 Dalang

Dalang [Welch98] es una extensión reflectiva del API de Java [Kramer96], que añade reflectividad computacional para modificar únicamente el paso de mensajes del sistema, de un modo restringido. Implementa dos mecanismos de reflectividad: en tiempo de compilación (estática) y en tiempo de ejecución (dinámica).

El conjunto de clases utilizadas en el esquema estático se muestra en la Figura 7.20. Dada una clase C cuyo paso de mensajes deseemos modificar, Dalang sustituye dicha clase por otra nueva con la misma interfaz y nombre, renombrando la original –esto es posible gracias a la introspección de la plataforma Java [Sun97b]. La nueva clase posee un objeto de la clase original en la que delegará la ejecución de todos sus métodos. Sin embargo, poseerá la capacidad de ejecutar, previa y posteriormente a la invocación del método, código adicional que pueda efectuar transformaciones de comportamiento (en la Figura 7.20, este código reside en la implementación de los métodos `beforeMethod` y `afterMethod`).

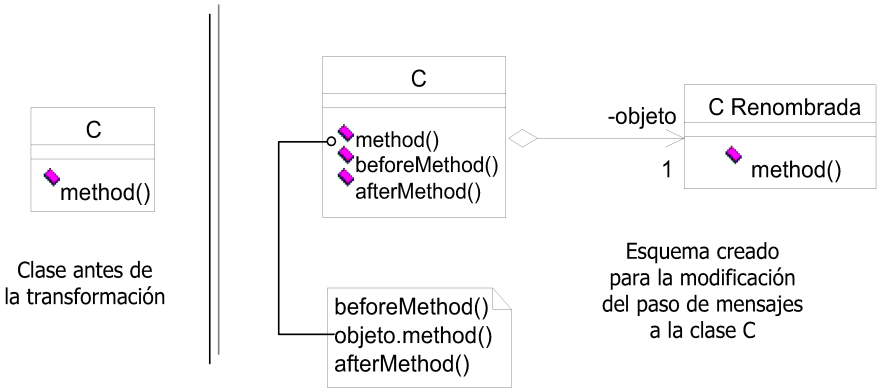


Figura 7.20. Transformación de una clase en Dalang para obtener reflectividad.

La reflectividad dinámica del sistema se obtiene añadiendo al esquema anterior la implementación de un cargador de código capaz de realizar la carga dinámica de las nuevas clases en el sistema. La plataforma Java ofrece fácilmente esta posibilidad mediante la implementación de una clase derivada de `ClassLoader` [Gosling96].

Esta arquitectura posee un conjunto de inconvenientes:

- Transparencia. Se permite modificar el paso de mensajes de un conjunto de objetos (las instancias de la clase renombrada); sin embargo, no es posible modificar la semántica del paso de mensajes para todo es sistema.
- Grado de reflectividad. La modificación de la semántica se reduce al paso de mensajes y se realiza en un grado bastante reducido –ejecución anterior y posterior de código adicional.
- Eficiencia. La creación dinámica de clases requiere una compilación al código nativo de la plataforma, con la consecuente ralentización en tiempo de ejecución.

Su principal ventaja es que no modifica la máquina virtual de Java ni el código existente en su plataforma; de esta forma, el sistema no pierde la portabilidad del código Java y es compatible con cualquier aplicación desarrollada para esta plataforma virtual.

7.4.7 NeoClasstalk

Tras los estudios de reflectividad estructural llevados a cabo con el sistema ObjVlisp (analizado en § 7.2.3) la arquitectura evolucionó hacia una ampliación de Smalltalk [Goldberg83] denominada Classtalk [Mulet94]; su principal objetivo era utilizar esta plataforma como medio de estudio experimental en el desarrollo de aplicaciones estructuralmente reflectivas. Continuando con el estudio de aplicaciones basadas en reflectividad, el desarrollo de un MOP que permitiese modificar el comportamiento de las instancias de una clase hizo que el sistema se renombrase a NeoClasstalk [Rivard96].

La aproximación que utilizaron para añadir reflectividad computacional a NeoClasstalk fue la utilización del concepto de metaclase[20], propio del lenguaje Smalltalk. Una metaclase define la forma en la que van a comportarse sus instancias, es decir, sus clases asociadas –el concepto de metaclase es tomado como un mecanismo para definir la semántica computacional de las clases instanciadas. Sobre este sistema, se desarrollaron metaclases genéricas para poder ampliar las características del lenguaje [Ledoux96] tales como la definición de métodos con pre y poscondiciones, clases de las que no se pueda heredar o la creación de clases que tan sólo puedan tener una única instancia –patrón de diseño *Singleton* [GOF94].

El modo en el que se modifica el comportamiento se centra en la modificación dinámica de la metaclase de una clase. Como se muestra en la Figura 7.21, existe una metaclase por omisión denominada `StandardClass`; esta metaclase define el comportamiento general de una clase en el lenguaje de programación Smalltalk.

[20] Al igual que en los lenguajes orientados a objetos basados en clases, un objeto es una instancia de unaclase, el concepto de metaclase define una clase como instancia de una metaclase – la cual define el comportamiento de todas sus clases asociadas.

El nuevo comportamiento deseado deberá implementarse en una nueva metaclase, derivada de `StandardClass`.

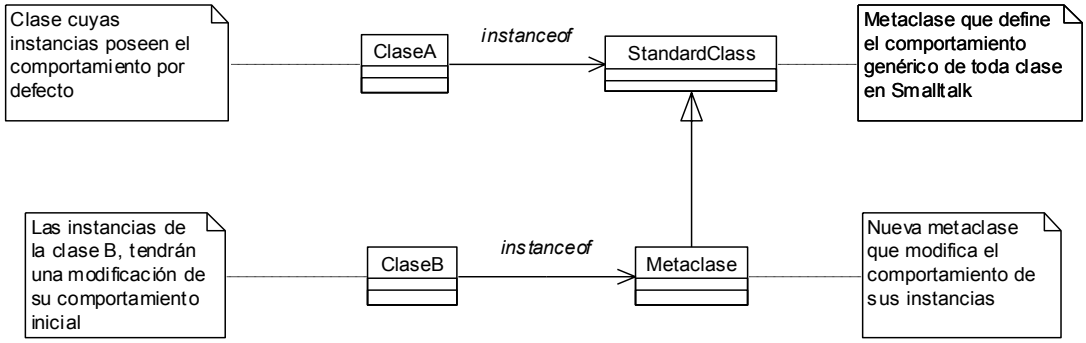


Figura 7.21. MOP de NeoClasstalk utilizando metaclases.

En NeoClasstalk, los objetos pueden cambiar de clase dinámicamente, modificando su asociación `instanceof`. Si modificamos esta referencia en el caso de una clase, estaremos modificando su metaclase y, por tanto, modificando su comportamiento.

Una de las utilizaciones prácticas de este sistema fue el desarrollo de OpenJava [Ledoux99], un ORB capaz de adaptarse dinámicamente a los requisitos del programador CORBA [OMG98]. Mediante la modificación del comportamiento basado en metaclases, añade una mayor flexibilidad al *middleware* CORBA consiguiendo:

- Modificación dinámica del protocolo de comunicaciones. La utilización de objetos delegados (*proxy*) permite ser configurada para modificar el comportamiento del paso de mensajes, seleccionando dinámicamente el protocolo deseado.
- Migración de objetos servidores dinámicamente.
- Replicación de objetos servidores.
- Implementación de un sistema dinámico de caché.
- Gestión dinámica de tipos. Accediendo dinámicamente a las especificaciones de los interfaces de los objetos servidores (archivos IDL), se pueden implementar comprobaciones de tipo en tiempo de ejecución.

El desarrollo de todo el sistema fue llevado a cabo siguiendo la separación de incumbencias: la parte de una aplicación que modele el dominio del problema se deberá separar del resto de código que pueda ser reutilizado para otras aplicaciones, y modele un aspecto global a varios sistemas.

7.4.8 Moostrap

Moostrap [Mulet93] es un lenguaje orientado a objetos reflectivo basado en prototipos, implementado como un intérprete desarrollado en Scheme [Abelson2000].

Acorde a la definición de la mayor parte de los lenguajes basados en prototipos, define un número reducido de primitivas computacionales que va extendiendo mediante la utilización de sus características estructuralmente reflectivas, para

ofrecer un mayor nivel de abstracción en la programación de aplicaciones. La abstracción del objeto –la entidad básica en los lenguajes basados en prototipos– queda definida con primitivas de reflectividad estructural:

- Un objeto viene definido como un conjunto de miembros (*slots*). Éstos pueden constituir datos representativos del estado dinámico del objeto (atributos) o su comportamiento (métodos). La diferencia entre los dos tipos de miembros, es que los segundos pueden ser evaluados, describiendo la ejecución de un comportamiento[21].

- Adición dinámica de miembros a un objeto. Todo objeto posee el miembro computacional primitivo `addSharedSlots` capaz de añadir dinámicamente un *slot* a un objeto.

- Eliminación dinámica de miembros de un objeto, mediante la primitiva `removeSlot`.

Mediante sus capacidades estructuralmente reflectivas, extiende las primitivas iniciales para ofrecer un mayor nivel de abstracción al programador de aplicaciones. Un ejemplo de esto es la definición del mecanismo de herencia, apoyándose en su reflectividad estructural dinámica [Mulet93]: si un objeto recibe un mensaje y no tiene un miembro con dicho nombre, se obtiene su miembro `parent` y se le envía dicho mensaje a este objeto, continuando este proceso de un modo recursivo.

Además de reflectividad estructural, Moostrap define un MOP para permitir modificar el comportamiento de selección y ejecución de un método de un objeto, ante la recepción de un mensaje. La semántica de la derogación de estas dos operaciones viene definida por el concepto de metaobjeto [Kiczales91], utilizado en la mayor parte de los MOPs existentes.

El paso de un mensaje puede modificarse en dos fases: primero, ejecutándose el metaobjeto asociado a la selección del miembro y, posteriormente, interpretando el comportamiento definido por el metaobjeto que derogue la ejecución del método.

Utilizando Moostrap, se trató de definir una metodología para crear metacomportamientos mediante la programación de metaobjetos en Moostrap [Mulet95].

7.5 Intérpretes Metacirculares

Dentro del capítulo anterior, en el § 6.2, razonábamos acerca del concepto de reflectividad computacional utilizando la metáfora de una torre de intérpretes. Cuando una aplicación pueda acceder a su nivel inferior, podrá modificar su comportamiento. Si lo que desea es modificar la semántica de su semántica (el modo en el que se interpreta su comportamiento), deberá acceder en la torre a un nivel computacional dos unidades inferior.

El proceso de acceder, desde un nivel en la torre de intérpretes definida por Smith [Smith82], a niveles inferiores puede, teóricamente, extenderse hasta el infinito –al fin y al cabo, todo intérprete será ejecutado o animado por otro. Las imple-

[21] Este proceso de convertir datos en computación, lo definimos "descosificación" al tratarse del proceso contrario de "cosificación" –definido en el Capítulo 6.

mentaciones de intérpretes capaces de ofrecer esta abstracción se han denominado intérpretes metacirculares (*metacircular interpreters*) [Wand88].

7.5.1 3-Lisp

La idea de la torre infinita de intérpretes propuesta por Smith [Smith82] en el ámbito teórico tuvo distintas implementaciones en un futuro inmediato [Rivières84].

El desarrollo de los prototipos de intérpretes metacirculares comenzó por la implementación de lenguajes de computación sencilla. El diseño de un intérprete capaz de ejecutar un número infinito de niveles computacionales, supone una elevada complejidad que crece a medida que aumentan las capacidades computacionales del lenguaje a interpretar. El primer prototipo metacircular desarrollado, denominado 3-Lisp [Wand88], interpreta un subconjunto del lenguaje Lisp [Steele90], y permite cosificar y reflejar un número indefinido de niveles computacionales.

El estado computacional que será reflejado en este lenguaje está formado por tres elementos [Wand88]:

- 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 ejecutada.
- 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 –denominado metacontinación (*metacontinuation*) [Wand88]– queda definido formalmente por tres valores (e, r, k), que podrán ser accedidos desde el metasistema como un conjunto de tres datos (cosificación). La capacidad de representar formalmente y mediante datos el estado computacional de una aplicación en tiempo de ejecución, aumenta en complejidad al aumentar el nivel de abstracción del lenguaje de programación en el que haya sido codificada. Por esta causa, la mayoría de los prototipos de intérpretes metacirculares desarrollados computan lenguajes de semántica reducida.

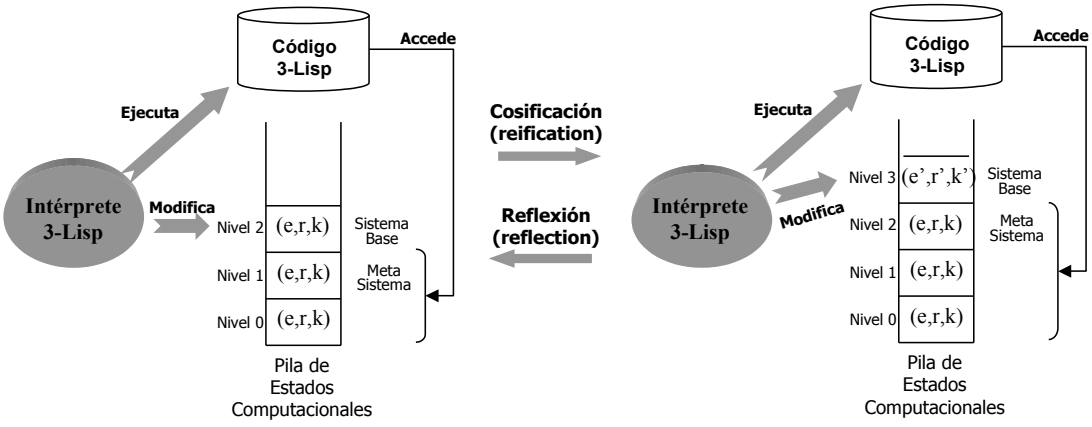


Figura 7.22. Implementación de un intérprete metacircular de 3-Lisp.

En la Figura 7.22 se aprecia el funcionamiento de los prototipos de interpretación de 3-Lisp. Existe un programa codificado en 3-Lisp que posibilita el cambio de nivel con las operaciones *reify* (aumento de nivel) y *reflect* (reducción de nivel). Un intérprete del subconjunto de Lisp definido va leyendo y ejecutando el lenguaje fuente. La ejecución de la aplicación supone la modificación de los tres valores que definen el estado computacional de la aplicación (e, r, k). En la interpretación se puede producir un cambio de nivel de computación:

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

- Reflexión (*reflect*): Se desapila el contexto actual volviendo al estado anterior existente en la pila. La ejecución continúa donde había cesado antes de hacer la última cosificación.

Las condiciones necesarias para implementar un prototipo de estas características son básicamente dos:

- Expresividad computacional mediante un único lenguaje. Puesto que realmente existe un único intérprete (ver Figura 7.23), éste estará obligado a animar un solo lenguaje de programación. En todos los niveles computacionales deberá utilizarse por tanto el mismo lenguaje de programación.

- Identificación formal del estado computacional. La semántica computacional del lenguaje deberá representarse como un conjunto de datos manipulables por el programa. La complejidad de este proceso es excesivamente elevada para la mayoría de los lenguajes de alto nivel.

7.5.2 ABCL/R2

La familia de lenguajes ABCL fue creada para llevar a cabo investigación relativa al paralelismo y orientación a objetos. Inicialmente se desarrolló un modelo de computación concurrente denominado ABCM/1 (*An object-Based Concurrent computation Model*) y su lenguaje asociado ABCL/1 (*An object-Based Concurrent Language*) [Yonezawa90].

En la implementación de un modelo computacional concurrente, la reflectividad computacional ofrece la posibilidad de representar la estructura y la computación concurrente mediante datos (cosificación), utilizando abstracciones apropiadas. En la definición del lenguaje ABCL/R (ABCL reflectivo) [Watanabe88], a partir de todo objeto "x" puede obtenerse su metaobjeto "□ x" que representa, mediante su estructura, el estado computacional de "x". Se implementa un mecanismo de conexión causal, para que los cambios del metaobjeto se reflejen en el objeto original. La operación "□ " puede aplicarse tanto a objetos como a metaobjetos, tratándose pues de una implementación de una torre infinita de intérpretes (intérprete metacircular).

Para facilitar la coordinación entre metaobjetos del mismo tipo, y para definir comportamientos similares de un grupo de objetos, la definición del lenguaje ABCL/R2 [Matsuoka91] añadía el concepto de "metagrupo": metaobjeto que defi-

ne el comportamiento de un conjunto de objetos del sistema base. Para implementar la adición de metagrupos, surgen determinadas ampliaciones del sistema:

- Nuevos objetos de núcleo (*kernel objects*) para gestionar los grupos: *The Group Mangager*, *The Primary Metaobject Generator* y *The Primary Evaluator*.
- Se crea un paralelismo entre dos torres de intérpretes: la torre de metaobjetos (activada mediante la operación "□") y la torre de metagrupos (activada mediante la operación "□□").
- Objetos no cosificables (*non-refying objects*). Se ofrece la posibilidad de definir objetos no reflectivos para eliminar la creación de su metaobjeto adicional y obtener así mejoras de rendimiento.

7.5.3 MetaJ

MetaJ [Douence99] es un prototipo que trata de ofrecer las características propias de un intérprete metacircular para un subconjunto del lenguaje de programación Java [Gosling96]. Expresado siempre en Java, el intérprete permite cosificar objetos para acceder a su propia representación interna (reflectividad estructural) y a la representación interna de su semántica (reflectividad computacional).

Inicialmente el intérprete procesa léxica y sintácticamente el código fuente, creando un árbol sintáctico con nodos representativos de las distintas construcciones sintácticas del lenguaje. El método `eval` de cada uno de estos nodos representará la semántica asociada a cada uno de sus elementos sintácticos.

Conforme la interpretación del árbol se va llevando a cabo, se van creando objetos representativos de los creados por el usuario en la ejecución de la aplicación. Todos los objetos creados poseen el método `reify` que nos devuelve un metaobjeto: representación interna del objeto que nos permite acceder a su estructura (atributos, métodos y clase), así como a su comportamiento (por ejemplo, la búsqueda de atributos o la recepción de mensajes). Podremos obtener así:

- Reflectividad estructural: Accediendo y modificando la estructura del metaobjeto, se obtiene una modificación estructural del objeto; existe un mecanismo de conexión causal que refleja los cambios realizados en todo metaobjeto.
- Reflectividad computacional: Se consigue modificando la clase de una instancia por una clase derivada que derogue el método que especifica la semántica a alterar. En la Figura 7.23 se muestra cómo se modifica la clase del metaobjeto para ser otra con la redefinición del método `lookupMethod`, encargada de gestionar la recepción de mensajes.

La parte novedosa de MetaJ sobre el resto de sistemas estudiados a lo largo de este capítulo reside en la capacidad de poder cosificar metaobjetos en el grado que deseemos. Si invocamos al método `reify` de un metaobjeto, obtendremos la representación de un metaobjeto pudiendo modificar así la semántica de su comportamiento. El acceso reflectivo no posee un límite de niveles, constituyéndose así como un caso particular de un intérprete metacircular.

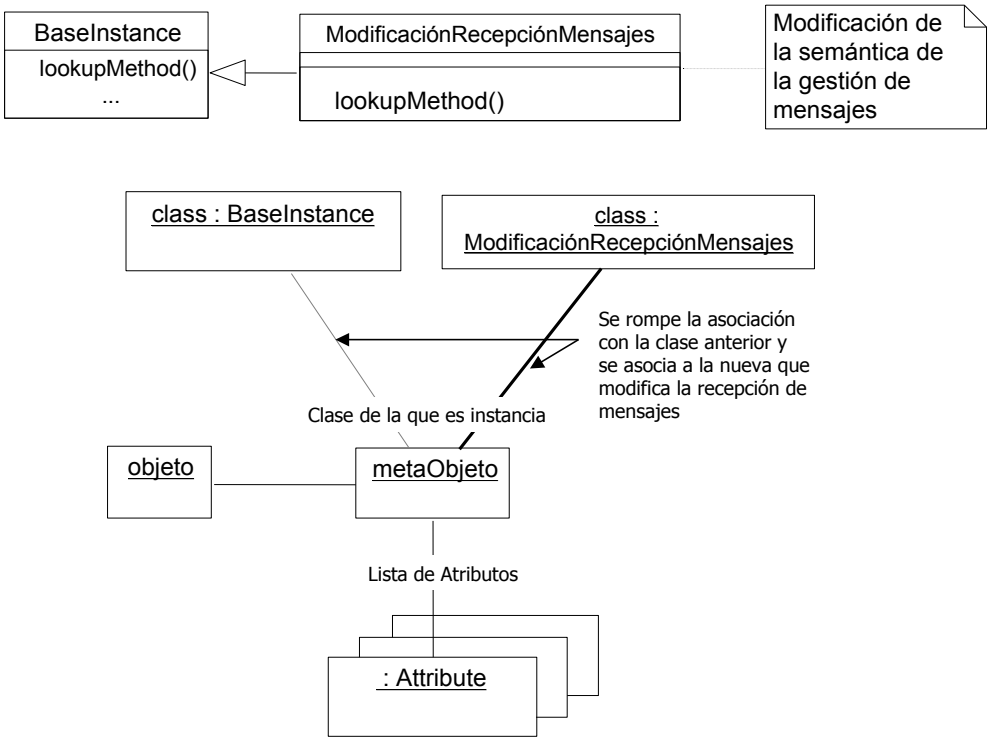


Figura 7.23. Modificación de la clase de un objeto, para obtener la modificación de la semántica de la recepción de mensajes.

7.6 Conclusiones

A lo largo de este capítulo, estudiando distintos tipos de sistemas reflectivos, hemos visto cómo la reflectividad es una técnica que puede ser empleada para obtener flexibilidad en un sistema computacional –en el Capítulo 4 vimos un conjunto de técnicas alternativas. En este punto, analizaremos qué puede aportar esta técnica a los objetivos buscados en esta Tesis (§ 1.2), así como las limitaciones encontradas.

Uno de los objetivos fundamentales buscados para el sistema de persistencia es su transparencia (§ 2.1). En concreto, se desea que la incumbencia de la persistencia sea totalmente independiente de la lógica de la aplicación (requisito § 2.1.1). Como se ha estudiado en los capítulos anteriores, ninguno de los sistemas estudiados permite dar respuesta a este requisito. Existen multitud de aproximaciones al problema, algunas de las cuales ofrecen altos niveles de transparencia pero, tal y como se observó en el Capítulo 3 y en el Capítulo 5, existen dos primitivas de persistencia básicas que siempre deben ser invocadas explícitamente: el borrado y la recuperación de objetos persistentes. En § 5.4 se concluyó que este problema no podía ser abordado con una mera transformación del código de la aplicación ejecutada, puesto que lo que se necesita es poder modificar la semántica del motor computacional que ejecuta la aplicación.

Esta necesidad puede abordarse con reflectividad computacional (§ 6.3.1.3): un sistema dotado de reflectividad computacional puede modificar su propia semántica. Si el motor computacional sobre el que se ejecutase el sistema de persistencia ofreciese reflectividad computacional, se podría modificar la semántica de sus facetas computacionales haciendo implícitas las operaciones de recuperación y bo-

rrado de objetos persistentes. Se podría conseguir, por lo tanto, un sistema de persistencia completamente transparente, de acuerdo con los requisitos recogidos en § 2.1.

Otro de los objetivos fundamentales del sistema de persistencia es que ofrezca la capacidad de adaptar su comportamiento en tiempo de ejecución (§ 2.2) –sea esta adaptación realizada por un usuario (§ 2.2.1) o programáticamente (§ 2.2.2). Ello sugiere que la reflectividad ofrecida debe ser dinámica, es decir, debe poder producirse en tiempo de ejecución (§ 6.3.2.2). Las implementaciones basadas en técnicas reflectivas en tiempo de compilación son las que se emplean en las herramientas estáticas de programación orientada a aspectos, obteniéndose, pues, las mismas limitaciones (§ 5.4).

Una de las técnicas habitualmente utilizadas para implementar reflectividad computacional dinámica es la utilización de protocolos de metaobjeto o MOP (*Meta-Object Protocols*). El concepto de MOP establece un modo de acceso del sistema base al metasistema, identificando el comportamiento que puede ser modificado. Esta técnica presenta ciertas limitaciones de cara a lograr los requisitos de adaptabilidad exigidos al sistema de persistencia (§ 2.2):

- La especificación del MOP supone una restricción a priori. El protocolo debe contemplar lo que será adaptable dinámicamente. Esta especificación a realizar previamente a la ejecución de una aplicación implica que ésta no podrá adaptarse a cualquier requisito surgido dinámicamente –ha de estar contemplado en el protocolo. Por el contrario, el sistema de persistencia buscado en este trabajo debe ofrecer la capacidad de flexibilizar en tiempo de ejecución un número no predeterminado de parámetros (§ 2.2). Es decir estos parámetros no deben estar restringidos a los considerados en tiempo de diseño, podrían generarse y configurarse dinámicamente.

- La estructura establecida en los sistemas que utilizan MOPs es siempre dependiente de un único lenguaje de programación, lo que viola el requisito § 2.3.1.

Otro tipo de sistemas que permiten ofrecer reflectividad computacional dinámica son los intérpretes metacirculares. Estos sistemas ofrecen el mayor nivel de flexibilidad computacional respecto al dominio de niveles computacionales a modificar. El acceso a cualquier elemento de la torre de intérpretes permite modificar la semántica del sistema en cualquier grado. Sin embargo, aunque teóricamente facilita la comprensión del concepto de reflectividad, en un campo más pragmático puede suponer determinados inconvenientes. La posibilidad de acceso simultáneo a distintos niveles puede producir la pérdida del conocimiento de la semántica del sistema, sin conocerse realmente cuál es el significado del lenguaje de programación [Foote90].

En los sistemas estudiados basados en interpretación metacircular, se ha dado precedencia a ofrecer un número indefinido de niveles de computación accesibles, frente a ofrecer un mayor grado de información a cosificar. Si tomamos 3-Lisp como ejemplo, ofrece la cosificación del estado computacional de la aplicación, pero no permite modificar la semántica del lenguaje; el intérprete es monolítico e invariable. Esto impediría la realización implícita de las operaciones de obtención y borrado de objetos persistentes, condición necesaria para alcanzar el grado de transparencia total exigido en § 2.1

Para conseguir este requisito mediante la implementación de infinitos niveles, debería especificarse la semántica del lenguaje en el propio estado de computación, extrayéndola del intérprete monolítico.

En el caso de MetaJ, se restringe a priori el número de operaciones semánticas a modificar, puesto que han de estar predefinidas como métodos de una clase de comportamiento. Esto viola las dos condiciones en las que se fundamentan los requisitos de adaptabilidad para el sistema de persistencia (§ 2.2):

- No existe pues, una flexibilidad no restringida a priori.
- Además, la modificación a llevar a cabo en el comportamiento ha de especificarse en tiempo de compilación.

Como conclusión, en los sistemas estudiados aparecen limitaciones respecto al grado de modificación de la semántica adaptable (§ 7.4 y § 7.5), y a la imposibilidad de modificar el lenguaje de programación (§ 7.4). El sistema de persistencia planteado en esta Tesis debe superar estas restricciones (requisitos § 2.2 y § 2.3.3). Por lo tanto, será necesario utilizar un sistema de reflectividad computacional no restrictivo que ofrezca la capacidad de modificar el lenguaje de programación utilizado, sin limitar a priori qué características computacionales pueden ser adaptadas. Este sistema será descrito en el Capítulo 9.

Capítulo 8
ARQUITECTURA DEL SISTEMA

Una vez descritos todos los objetivos y requisitos a conseguir, estudiado los distintos sistemas y técnicas existentes en la consecución de éstos, y evaluadas las aportaciones y carencias de los mismos, describiremos la arquitectura general del sistema innovador propuesto en esta memoria, analizando brevemente su estructura y los objetivos generales a cumplir por cada uno de sus elementos.

Profundizaremos en capítulos posteriores en la descripción de cada uno de los elementos del sistema, así como en la justificación de las técnicas seleccionadas y el cumplimiento de los objetivos marcados.

8.1 Capas del Sistema

El sistema está dividido en dos capas o elementos bien diferenciados siguiendo el patrón arquitectónico *Layers* [Buschmann96]. Éstos pueden apreciarse en la siguiente figura:

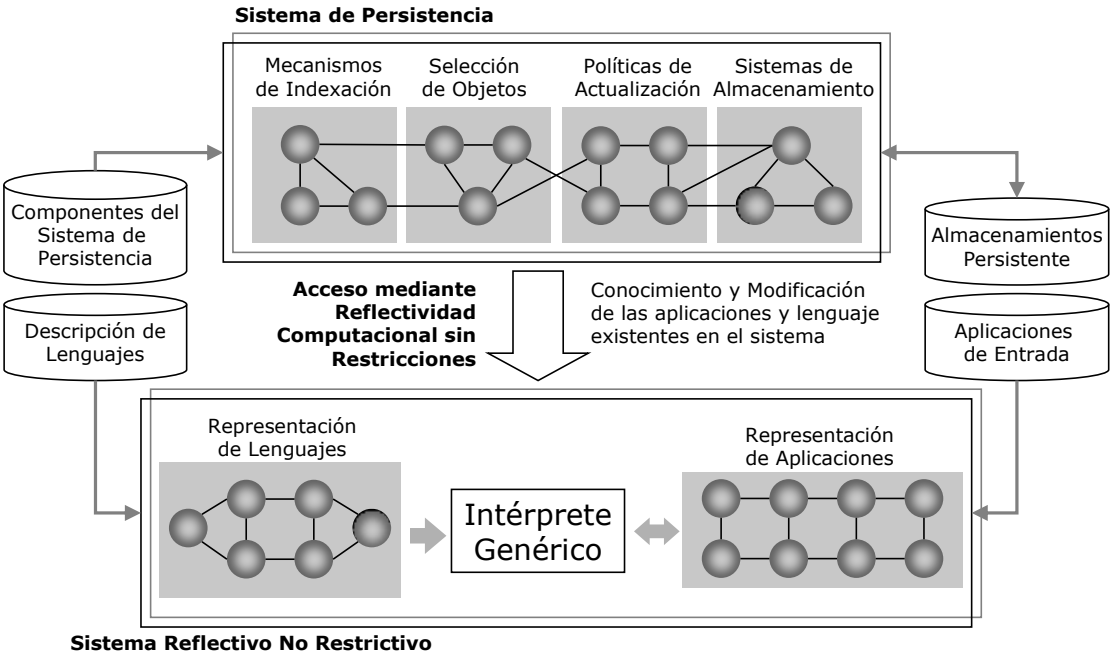


Figura 8.1: Arquitectura del sistema.

8.1.1 Sistema Reflectivo No Restrictivo

El sistema de persistencia propuesto ha de estar desarrollado sobre una plataforma que ofrezca reflectividad computacional no restrictiva en tiempo de ejecución. De esta forma, la programación de dicho sistema de persistencia podrá utilizar y modificar la representación de las aplicaciones en tiempo de ejecución así como su semántica –su lenguaje de programación. Así se ofrece a la implementación del sistema de persistencia una interacción con el motor computacional de las aplicaciones, en lugar de una transformación de código fuente, mucho más limitada computacionalmente, como sucede en los sistemas orientados a aspectos.

Un intérprete genérico toma la especificación de un lenguaje y ejecuta una aplicación codificada en éste, consiguiendo que todo el sistema sea independiente del lenguaje. La programación del sistema de persistencia se llevará a cabo en cualquier lenguaje de programación, pudiendo elegir varios en dicho desarrollo. La elección del modelo computacional por parte del intérprete genérico es una tarea crítica en el desarrollo del sistema puesto que éste será el modelo a utilizar por la totalidad de los lenguajes de programación. Adicionalmente se diseñará el sistema de persistencia acorde con dicho modelo, aunque las características inherentes al sistema de persistencia lo hacen adaptable sin restricciones establecidas a priori.

8.1.2 Sistema de Persistencia

El sistema reflectivo no restrictivo es el encargado de ejecutar los servicios del sistema de persistencia. Esta es una mera separación lógica de capas como un mecanismo de abstracción [Buschmann96], puesto que, en realidad, todo el sistema se ejecutará en el mismo nivel computacional. La capa de interpretación permite un "salto" computacional del sistema base al meta-sistema [Ortin2002], consiguiendo que el intérprete genérico, las aplicaciones de usuario, la representación de los lenguajes y el sistema de persistencia constituyan una misma aplicación en un mismo modelo computacional.

El resultado de esta arquitectura es que el sistema de persistencia permite añadir y modificar sus servicios en tiempo de ejecución así como la semántica y la estructura de las aplicaciones en función de los requisitos de persistencia impuestos en cada una de las aplicaciones. Puesto que el mecanismo primitivo de implementación de dichos requisitos es obtenido accediendo a la representación de las aplicaciones en tiempo de ejecución, el código fuente de las mismas se mantiene invariable independientemente de sus requisitos persistentes.

El sistema de persistencia está desarrollado por un conjunto de componentes adaptables dinámicamente. Éstos conforman cada uno de los parámetros referidos en los requisitos persistentes de una aplicación informática. Ejemplos son el mecanismo de indexación, el sistema de almacenamiento, la política de actualización de los objetos persistentes o la selección de elementos persistentes. Nuevos parámetros, o nuevas implementaciones de cada uno de ellos, podrán ser añadidos en tiempo de ejecución mediante el uso de reflectividad (siguiendo así una especie de mecanismo reflectivo de *plug-ins* [Biesack2005]) puesto que el motor computacional no ofrece restricciones de adaptabilidad.

8.2 Sistema Reflectivo No Restrictivo

En el análisis de cualquier sistema reflectivo, siempre se ha de tener en cuenta el debate "*Hamiltonians versus Jeffersonians"* [Foote92]. Por un lado está la flexibilidad otorgada al programador para hacer sus aplicaciones lo más adaptables y extensibles posible –*Jeffersonians*; por otro lado, la posibilidad de modificar indefinidamente el sistema puede conllevar a estados incoherentes de computación y semánticas ininteligibles [Foote90] –*Hamiltonians*.

Nuestro principal objetivo es desarrollar un estudio para la creación de un sistema flexible con un elevado grado de adaptabilidad. Por esta razón, nos centraremos en la vertiente *Jeffersoniana*, tratando de encontrar el mayor nivel de adaptabilidad posible. Una vez obtenido éste, un analizador semántico con un sistema de tipos robusto será el encargado de obtener una seguridad (*safety*) dinámica de acceso al meta-sistema desde el sistema base.

A continuación analizaremos los puntos más significativos de esta capa del sistema.

8.2.1 Características Adaptables

Basándose en la clasificación de adaptabilidad descrita en la taxonomía de reflectividad presentada en este documento, deberá otorgarse al sistema las siguientes características de adaptabilidad:

- Conocimiento dinámico del entorno. El conjunto del sistema ha de ser introspectivo, ofreciendo la posibilidad de conocer su estado y descripción dinámicamente: introspección.
- Acceso y modificación de su estructura. La estructura de los objetos existentes deberá ser manipulable en tiempo de ejecución, para conseguir adaptabilidad estructural dinámica: reflectividad estructural.
- Semántica computacional. La semántica computacional del sistema deberá poder conocerse, modificarse y ampliarse. De esta forma, una aplica-

ción en ejecución podrá ser adaptada sin necesidad de modificar su código fuente ni finalizar su ejecución: reflectividad computacional.

- Configuración del lenguaje de programación. Cualquier aplicación deberá ser capaz de modificar su propio lenguaje de programación para amoldarlo a sus necesidades específicas de expresividad: reflectividad de lenguaje.

Como hemos estudiado previamente, no existe sistema alguno dotado de la capacidad de ser adaptable en todas estas características. Por ello, propondremos la arquitectura de uno nuevo en el Capítulo 9.

8.2.2 Independencia del Lenguaje

Para esta capa del sistema, las aplicaciones deberán constituir procesos computacionales susceptibles de interactuar con el resto de aplicaciones existentes, sin que el lenguaje de programación sea una variable adicional del programa. Mediante algún mecanismo de implementación, el programador deberá ser capaz de desarrollar aplicaciones (o fracciones de aplicaciones) en el lenguaje que él desee, sin que ello restrinja el modo en el que se interactúe con el resto del sistema.

La programación de aplicaciones no deberá verse limitada a los lenguajes de programación más conocidos; el sistema deberá permitir constituir un entorno de desarrollo e interacción de lenguajes de propósito específico.

Deberá contemplarse la posibilidad de que la propia aplicación describa su lenguaje haciéndola autosuficiente para su ejecución, permitiendo el desarrollo de aplicaciones que realmente expresen computación por sí solas –no exista dependencia de la implementación de un intérprete del lenguaje empleado.

8.2.3 Único Modelo Computacional de Objetos

El sistema computacional reflectivo no restrictivo requiere la selección de un único modelo computacional de objetos, cualesquiera sean el lenguaje y plataforma seleccionados. La idea es que todo el código se ejecute sobre el modelo de objetos soportado por el sistema reflectivo, y que las aplicaciones puedan interactuar entre sí independientemente de su lenguaje de programación. Tanto las aplicaciones de usuario como los distintos componentes del sistema de persistencia deberán compartir un único modelo computacional.

Una de las tareas a llevar a cabo por el intérprete genérico del sistema reflectivo es traducir las aplicaciones codificadas mediante un lenguaje de programación, a su correspondencia en el modelo computacional del sistema. Una vez este proceso haya sido llevado a cabo, la aplicación podrá interaccionar con el resto del sistema como si hubiese sido codificada sobre su lenguaje nativo.

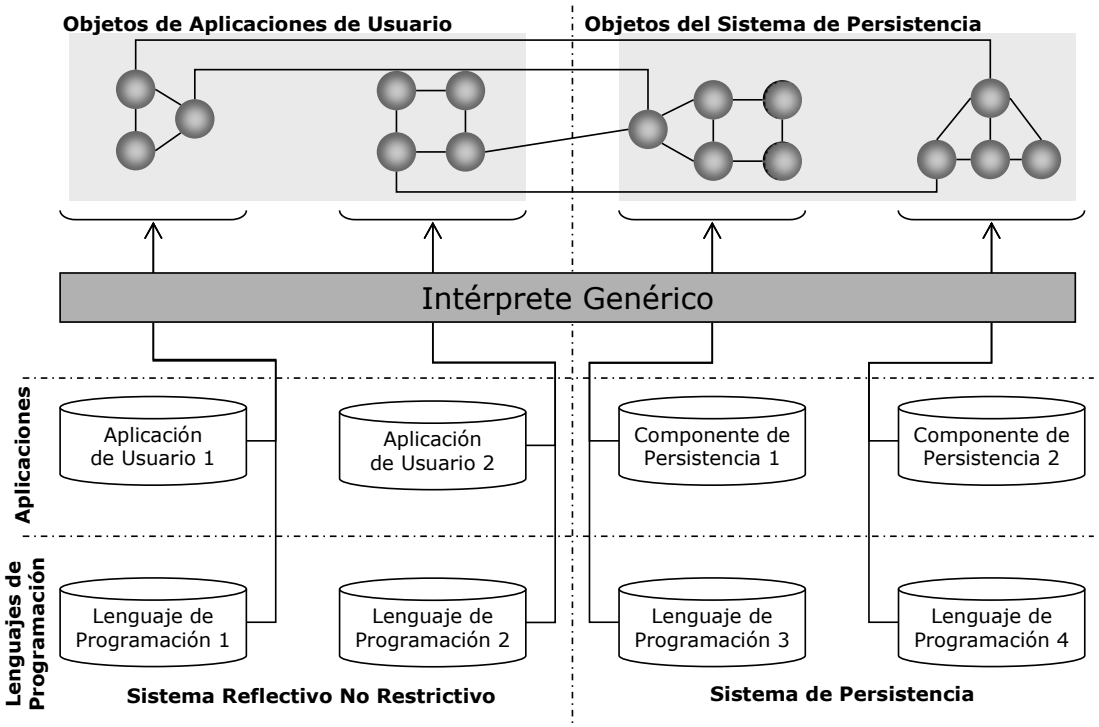


Figura 8.2: Interacción entre distintos objetos de un mismo espacio computacional.

Como se muestra en la figura anterior, tanto las aplicaciones de usuario como los diversos módulos del sistema de persistencia comparten el mismo modelo computacional. El programador elige un lenguaje de programación y, en la interpretación de éste, su modelo es traducido al propio del sistema reflectivo. Con el único modelo de objetos existente, la interacción de aplicaciones es directa, incluso con los componentes del sistema de persistencia, indistintamente del lenguaje de programación utilizado.

El resultado es un único espacio computacional de interacción de objetos que ofrecen sus servicios al resto del sistema. La procedencia de cada uno de ellos es intrascendente, puesto que la máquina computa éstos de modo uniforme.

8.2.4 Grado de Flexibilidad de la Semántica Computacional

Una de las características a hacer adaptable en nuestro sistema es su semántica computacional. Esta faceta implica la posibilidad de adaptar dinámicamente una aplicación, sin necesidad de modificar su código fuente, permitiendo así desarrollar código de persistencia más reutilizable que en los sistemas existentes estudiados a lo largo de este trabajo.

El modo en el que expresamos la semántica de un lenguaje es mediante otro lenguaje de especificación de semántica [Mosses92]. Si accedemos y modificamos la semántica del lenguaje de programación, lo haremos mediante el lenguaje de especificación de semántica. Podemos observar pues, como éste es un concepto recursivo: ¿cómo definimos la semántica del lenguaje de especificación de semánticas?

La flexibilidad de modificar la semántica de un lenguaje puede pasar por modificar, a su vez, la semántica del lenguaje de especificación de su semántica. La pregunta que debemos hacernos es si esta facultad es o no realmente necesaria para la consecución de los requisitos de nuestro sistema.

Tras el estudio realizado en § 7.5 de los sistemas que permiten modificar la semántica de cualquier lenguaje de un modo recursivo en infinitos niveles (intérpretes metacirculares), concluimos las siguientes afirmaciones:

- La posibilidad de modificar la semántica de semánticas en un grado indefinido puede llevar a la pérdida del conocimiento de la semántica real existente en el sistema –no conoceremos el comportamiento real de la ejecución de una instrucción.
- Los beneficios aportados por la utilización de elevar la flexibilidad a más de un nivel computacional son muy limitados [Foote90].
- La mayoría de los sistemas limitan el espectro de la semántica computacional a flexibilizar, es decir, no son capaces de modificar la totalidad de su comportamiento.

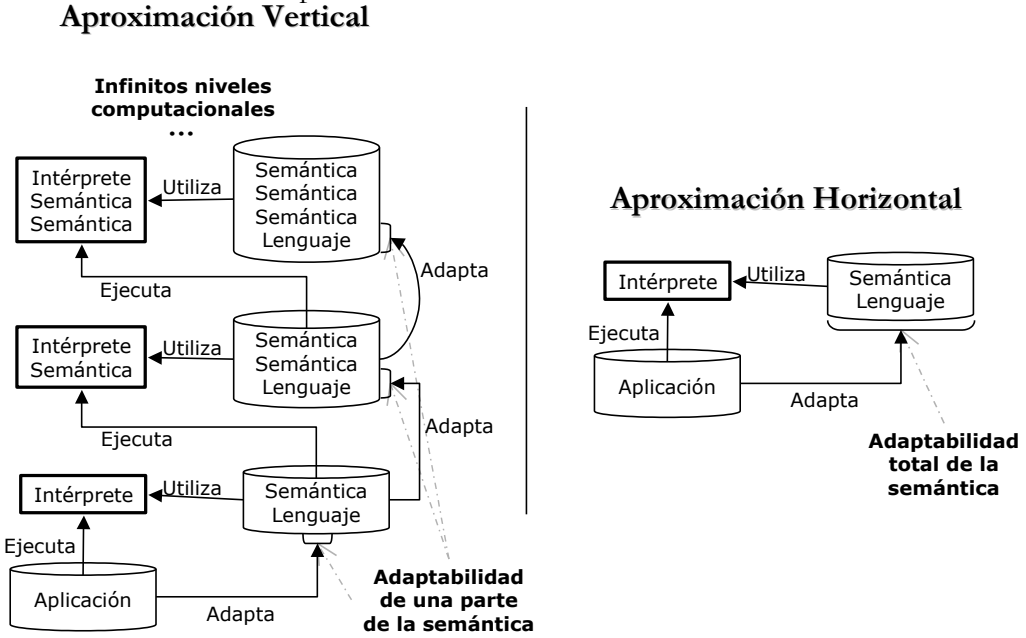


Figura 8.3: Distintas aproximaciones en la consecución de adaptabilidad en la semántica del sistema.

En función del resumen del estudio realizado en este punto, y buscando la mayor flexibilidad computacional de nuestro sistema, limitaremos la modificación de la semántica computacional a un nivel –dos grados de altura–, pero obligando a que ésta permita modificar la totalidad de sus computaciones –grado de anchura ilimitado, no especificado por un protocolo de acceso estático.

8.2.4.1 Adaptabilidad No Restrictiva

En el análisis efectuado en el punto anterior, requeríamos la necesidad de modificar la semántica de cualquier faceta computacional del sistema. Para este punto, se demanda que el mecanismo de modificación de la semántica no imponga restricciones previas la ejecución de dichas modificaciones.

Existen sistemas que permiten modificar cualquier semántica computacional, estableciendo previamente el protocolo de acceso a ellas –protocolo de meta-objetos (§ 7.4). Si para una aplicación, una vez codificada y ejecutada, se desea modificar algún aspecto no previsto con anterioridad, su adaptación dinámica no será factible. En este tipo de sistemas no es posible desarrollar aplicaciones adaptables a

contextos desconocidos en fase de desarrollo. Su flexibilidad está condicionada al conocimiento de a qué deberá adaptarse previamente a su ejecución.

La adaptabilidad de nuestro sistema no deberá necesitar el conocimiento previo de aquello que sea susceptible de ser modificado. Deberá desarrollar un mecanismo de flexibilidad no restrictivo, en el que cualquier característica podrá ser adaptada sin necesidad de estimarlo previamente.

8.2.5 Requisitos Computacionales de la Plataforma de Desarrollo

A la hora de desarrollar el sistema de reflectivo no restrictivo, es necesario llevar a cabo la selección de la plataforma de desarrollo de un modo adecuado. Deberemos tener en cuenta los requisitos impuestos a ésta, para que sea factible su implementación.

8.2.5.1 Portabilidad

La codificación del sistema reflectivo ha de ser independiente de toda plataforma física (§ 2.3). Para codificar una única vez éste e instalarlo en cualquier sistema, el código no deberá tener dependencia alguna de la plataforma física donde se implante (§ 2.3.3 y § 2.3.2). Del mismo modo, deberá ser un sistema abierto, para que cualquier aplicación desarrollada sobre cualquier lenguaje pueda utilizar los servicios ofertados por este código (§ 2.3.1); no deberá existir restricción alguna al respecto.

8.2.5.2 Introspección

La utilización de la plataforma de desarrollo en sistemas computacionales heterogéneos requiere el conocimiento dinámico del subconjunto de funcionalidades instaladas en cada plataforma –tal y como vimos en § 3.7. El conocimiento de la estructura de las aplicaciones y lenguajes en tiempo de ejecución es primordial a la hora de implementar un sistema de persistencia adaptable y adaptativo. Es necesario, pues, que la plataforma de desarrollo ofrezca un mecanismo dinámico de introspección.

8.2.5.3 Adaptabilidad

Puesto que el objetivo principal de este proyecto es el desarrollo de un sistema computacional flexible, la adaptabilidad de sus funcionalidades es una característica primordial. Para poder manipular, por parte del sistema de persistencia, de un modo sencillo en tiempo de ejecución la estructura de las aplicaciones y lenguajes, será necesario que la plataforma provea un sistema de adaptabilidad dinámica de las estructuras de los objetos: reflectividad estructural en tiempo de ejecución.

8.2.5.4 Generación de Código en Tiempo de Ejecución

Para que el sistema de persistencia pueda modificar la semántica de los lenguajes de programación, así como añadir nuevas funcionalidades propias de la incumbencia de la persistencia, es necesario que el sistema permita crear código dinámicamente y que éste se pueda añadir a las aplicaciones existentes en tiempo de ejecución como si se tratase de código creado estáticamente (programación generativa). Así, el sistema de persistencia podrá interactuar con el intérprete genérico, aumentando la semántica del lenguaje de las aplicaciones de usuario, como si de un lenguaje con persistencia transparente implícita se tratase.

8.3 Sistema de Persistencia

Como hemos comentado previamente, la separación en dos capas, el sistema reflectivo y de persistencia, es una separación lógica, no física. La idea general de la separación es que el sistema reflectivo no requiera de la existencia del sistema de persistencia, y que el segundo se apoye en las características del primero para ofrecer los objetivos descritos en este documento.

A continuación analizaremos los distintos elementos arquitectónicos del sistema de persistencia, así como el acoplamiento que posee con la capa del sistema reflectivo.

8.3.1 Componentes de Persistencia

El sistema de persistencia deberá estar compuesto por aquellos componentes funcionales que se encuentren dentro de los requisitos de algún sistema de persistencia –sea cual fuere. De esta forma, éstos podrían ser ofertados al desarrollador de aplicaciones como requisitos reutilizables de persistencia.

Adicionalmente, el sistema debería permitir la adición de nuevos componentes de persistencia dinámicamente sin necesidad de detener la ejecución del mismo. Mediante los requisitos de adaptabilidad impuestos a la otra capa (sistema reflectivo no restrictivo) éstos podrían formar parte de la nueva funcionalidad del sistema utilizando las técnicas reflectivas descritas.

Ejemplos de componentes de persistencia son la selección de objetos (qué objetos, en una determinada aplicación han de ser persistentes), la política de actualización (bajo qué circunstancias un objeto deberá ser actualizado en el sistema de almacenamiento), mecanismo de indexación (cómo se acceden a los objetos persistentes) y sistema de almacenamiento (dónde y cómo se almacenan los objetos persistentes). Tal y como hemos explicado, nuevos elementos deberían poder añadirse fácilmente –como, por ejemplo, sistemas implícitos de transacciones.

Dado un componente de persistencia, éste deberá ofrecer diversas alternativas de implementación. Si tomamos, por ejemplo, la política de actualización, debería permitirse la actualización de objetos persistentes en su sistema de almacenamiento cada vez que éstos vean modificado su estado o cada vez que transcurra un cierto intervalo de tiempo. Obviamente, podrán añadirse nuevas implementaciones (nuevas políticas de actualización en nuestro ejemplo) mientras el sistema se encuentre ejecutándose.

8.3.2 Sistemas de Almacenamiento

Un componente necesario del sistema de persistencia es la utilización de diversos sistemas de almacenamiento. Uno de los requisitos más típicos en lo referente a la persistencia de una aplicación es la utilización de uno, o varios, sistemas de almacenamiento específicos.

Los parámetros de persistencia descritos en el punto anterior no deben sufrir dependencia alguna del sistema de almacenamiento seleccionado. Deberán ser independientes unos de otros.

Una aplicación podrá elegir, y modificar dinámicamente, el sistema de almacenamiento que estime oportuno. Varios sistemas de almacenamiento pueden ser

utilizados para distintos elementos de un mismo programa, e incluso se debería permitir que un mismo objeto pudiese guardarse en varios sistemas de almacenamiento. Estos objetivos son propios de las funcionalidades de replicación y exportación de información ofrecidas por diversos sistemas gestores de bases de datos.

8.3.3 Interacción Directa con las Aplicaciones de Usuario

La implementación de un sistema de persistencia puede hacer uso de técnicas reflectivas para almacenar y restaurar los objetos persistentes de un sistema de almacenamiento. Dos de las ventajas obtenidas frente a los sistemas tradicionales es su transparencia (son independientes de la estructura del objeto) y su adaptabilidad (al estar codificadas en el mismo lenguaje, en lugar de formar parte de la semántica del sistema, su adaptabilidad pasa por modificar su implementación). Por estos dos objetivos, es necesario que el sistema posea características de introspección para el almacenamiento transparente y adaptable de los objetos, así como reflectividad estructural, para la reconstrucción de los objetos obtenidos del sistema de almacenamiento.

En la arquitectura propuesta de nuestro sistema, la capa del sistema de persistencia accede directamente a la representación de las aplicaciones del sistema en tiempo de ejecución. Puesto que uno de los requisitos de la plataforma de desarrollo del sistema reflectivo es que ofrezca reflectividad estructural, el conocimiento y modificación dinámicos de la estructura de las aplicaciones en tiempo de ejecución es directo. Los dos beneficios obtenidos son los descritos en el párrafo anterior.

8.3.4 Interacción Directa con los Lenguajes del Sistema

La característica de mayor innovación del sistema presentado en esta memoria es la característica de separar totalmente la incumbencia relativa a la persistencia de una aplicación del resto de sus requisitos, funcionales o no, permitiendo su adaptación dinámicamente.

Como hemos analizado en el Capítulo 3, dedicado al análisis de sistemas de persistencia existentes, los sistemas que más se acercan a la consecución de los objetivos impuestos son aquéllos que realizan una transformación del código de la aplicación, combinando la implementación de su lógica de negocio con sus competencias persistentes. Sin embargo, la transparencia, adaptabilidad y reutilización de su persistencia no se conseguía puesto que hay determinados servicios que dependen de la semántica del lenguaje –por ejemplo, es necesario saber cuándo se elimina un objeto de la memoria volátil.

Para poder obtener un sistema de persistencia totalmente transparente, es necesario añadir parte de su semántica al nivel del motor computacional (intérprete). Si queremos que éste sea además adaptable dinámicamente, deberá tener una representación externa y accesible programáticamente por el resto de las aplicaciones. Estos objetivos se obtienen gracias a que el sistema reflectivo ofrece la representación de todos los lenguajes existentes a las aplicaciones en ejecución, incluyendo al sistema de persistencia. El conocimiento y modificación dinámica de éstos permite obtener los requisitos impuestos.

8.3.5 Independencia del Lenguaje

Los diversos componentes del sistema de persistencia se podrán desarrollar en el lenguaje de programación que mejor conozca el desarrollador. Varios podrán coexistir al mismo tiempo, puesto que se desarrollan sobre el sistema reflectivo no restrictivo. Para que esto puedas ser llevado a cabo, la selección de un único modelo computacional de objetos, descrita en § 8.2.3, es decisiva.

8.3.6 Utilización del Único Modelo Computacional de Objetos

Uno de los requisitos de diseño que deberán tenerse en cuenta para poder desarrollar el sistema de persistencia de un modo adecuado es utilizar el único modelo computacional de objetos descrito en la capa del sistema reflectivo. Puesto que se ofrecerá todos los servicios de persistencia independientemente del lenguaje de programación elegido por el programador, el sistema deberá desarrollarse sobre entidades del modelo único, tal y como se busca en los sistemas dirigidos por modelos [OMG2001].

El modelo computacional orientado a objetos basado en prototipos ofrece una serie de ventajas para ser elegido:

- Este modelo se adapta de un modo coherente a la reflectividad estructural en tiempo de ejecución, puesto que no tiene que implementarse versiones distintas de esquemas [Ortín2005].
- No tiene pérdida de expresividad en comparación con el modelo orientado a objetos basado en clases [Borning86].
- Se ha identificado, precisamente, como el modelo universal de computación orientado a objetos [Wolczko96].

Puesto que los componentes software son desarrollados sobre el sistema reflectivo que utiliza un modelo computacional único y éste es el empleado para diseñar el sistema de persistencia, su implementación podrá llevarse a cabo siguiendo la técnica de definirse a sí mismo –*bootstrapping* [Aho90].

Capítulo 9
ARQUITECTURA DEL SISTEMA REFLECTIVO NO RESTRICTIVO

A raíz de la estructuración de nuestro sistema en capas, presentada en el Capítulo 8, estudiaremos en este título la arquitectura de la primera capa que representaba un sistema computacional adaptable sin restricción alguna, independiente del lenguaje.

La arquitectura presentada deberá cumplir todos los objetivos generales impuestos en el Capítulo 8, analizando y justificando la utilización de las técnicas existentes descritas en capítulos anteriores, y definiendo el esquema general del funcionamiento del sistema sin entrar en consideraciones propias de diseño (Capítulo 11).

El desarrollo de esta última capa del sistema se llevará a cabo sobre un motor computacional determinado, por lo que la arquitectura propuesta demandará unos requisitos computacionales a la plataforma base. Estos requisitos serán proporcionados en último lugar.

9.1 Análisis de Técnicas de Obtención de Sistemas Flexibles

En el Capítulo 4 estudiamos un conjunto de distintas técnicas utilizadas por sistemas para obtener distintos grados de flexibilidad computacional. Por el contrario, en el Capítulo 7 todos los casos analizados utilizaban la misma idea para conseguir adaptabilidad: reflectividad. Analizaremos las aportaciones de estos dos grupos de sistemas.

9.1.1 Sistemas Flexibles No Reflectivos

Exceptuando los *frameworks* de propósito específico, el conjunto de aproximaciones descritas en el Capítulo 4 poseen un tratamiento estático de las aplicaciones, buscando siempre la separación de incumbencias o aspectos. En el conjunto de métodos existentes, se desarrolla por un lado la funcionalidad central de una aplicación y por otro se especifican aspectos como por ejemplo persistencia, distribución o planificación de hilos. Se trata de separar el código funcional de cada aplicación de las incumbencias propias de múltiples aplicaciones, susceptibles de ser reutiliza-

das. Existe una variante de la programación orientada a aspectos que permite realizar un tejido dinámico de los aspectos (§ 5.3) pero, como se vio en el Capítulo 5, únicamente un sistema permitía generar los aspectos dinámicamente y dicho sistema imponía a su vez serias restricciones (§ 5.3.1). En el resto, los aspectos debían ser definidos en tiempo de diseño.

El resultado es el desarrollo de software reutilizable y flexible respecto a un conjunto de aspectos carentes de dinamicidad. Estas técnicas están demasiado enfocadas a la creación de software susceptible de ser adaptado por diversas incumbencias, careciendo de la posibilidad de crear aplicaciones flexibles en tiempo de ejecución.

La principal carencia de este enfoque reside en la imposibilidad de modificar dinámicamente los aspectos de una aplicación. De forma añadida, la aplicación no puede adaptar su funcionalidad principal en tiempo de ejecución. Estas limitaciones se deben a la orientación distinta de objetivos de este tipo de sistemas respecto buscado en esta Tesis: la separación de incumbencias está más enmarcada en campos de reutilización de código, ingeniería del software o creación de un paradigma de programación, que en la flexibilidad dinámica.

9.1.2 Sistemas Reflectivos

La reflectividad supone más una técnica de lenguajes de programación que de ingeniaría del software o desarrollo de aplicaciones. Los lenguajes de programación reflectivos permiten modificar o acceder, en un determinado grado, a un conjunto de sus propias características o de sus aplicaciones, utilizando para ello distintos mecanismos de implementación.

Nos centraremos en una de las clasificaciones de reflectividad identificadas en § 6.3: "Cuándo se produce el reflejo". Los sistemas reflectivos que son adaptables en tiempo de compilación generan las mismas carencias que los que acabamos de analizar (§ 9.1.1): limitan su adaptabilidad al momento en el que la aplicación es creada, careciendo de flexibilidad dinámica.

Es por tanto necesario centrarse en los sistemas reflectivos adaptables en tiempo de ejecución, capaces de ofertar adaptabilidad dinámica mediante el empleo de técnicas de procesamiento de lenguajes –en la mayoría de los casos, mediante intérpretes. Este tipo de sistemas se distingue de los estáticos fundamentalmente en:

- Ofrecen la adaptabilidad de las aplicaciones en tiempo de ejecución.
- La adaptabilidad dinámica supone una ralentización de la ejecución de las aplicaciones al no producirse ésta en fase de compilación.

De algún modo, la flexibilidad dinámica ganada se paga con tiempos de ejecución más elevados.

9.1.2.1 Reflectividad Dinámica

Siguiendo con el estudio desarrollado en el Capítulo 7 y centrándonos en los objetivos principales de este trabajo (§ 1.2), las técnicas estudiadas más cercanas a ofrecer la flexibilidad buscada son las reflectivas en tiempo de ejecución. Dentro de esta clasificación tenemos dos grupos: intérpretes metacirculares y sistemas basados en MOPs (*Meta-Object Protocols*).

Los intérpretes metacirculares ofrecen una torre infinita [Wand88] de intérpretes reflectivos, en el que sólo parte de la semántica del sistema es adaptable. Como hemos comentado y justificado en § 7.6 y § 8.2.4, estos sistemas no ofrecen un grado total de adaptación semántica y pueden conllevar a la pérdida del conocimiento real de la semántica del sistema, escapándose así de los objetivos marcados.

Los entornos computacionales basados en MOPs limitan el número de niveles computacionales a dos, ofreciendo un protocolo de acceso desde el nivel superior al subyacente. Esta idea es similar a la buscada, pero, como se muestra en la Figura 9.1, posee un conjunto de limitaciones:

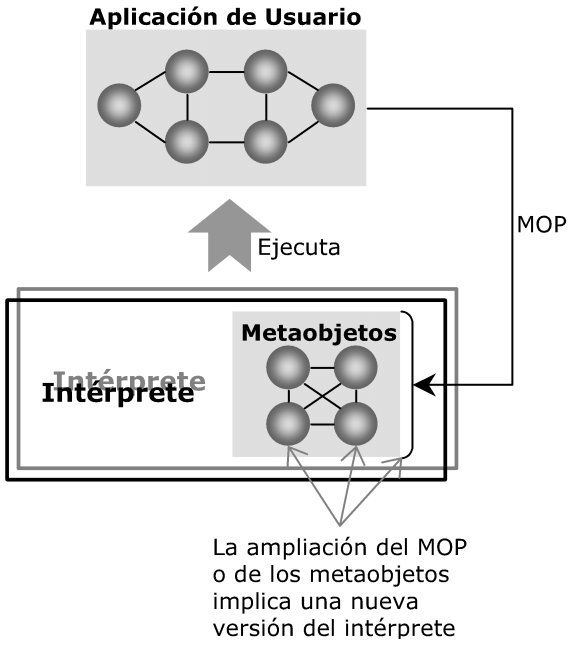


Figura 9.1. Estructura general de un MOP.

- La especificación del MOP supone una restricción a priori. El protocolo debe contemplar lo que será adaptable dinámicamente. Esta especificación a realizar previamente a la ejecución de una aplicación implica que ésta no podrá adaptarse a cualquier requisito surgido dinámicamente –ha de estar contemplado en el protocolo.

- La expresividad de los metaobjetos es restringida. El modo en el que una característica del sistema ofrece su adaptabilidad es mediante el concepto de metaobjeto [Kiczales91]: si algo es adaptable, se representará mediante un metaobjeto y sus métodos ofrecerán la forma en la que puede adaptarse.

- No existe una expresividad sin restricción de la adaptabilidad del sistema. Sólo podrá adaptarse dinámicamente aquello que posea un metaobjeto, y éste podrá adaptarse tan sólo en función de los métodos que ofrezca.

- Uno de los procesos de desarrollo de MOPs propuesto por Michael Golm [Golm98] se basa en la ampliación del protocolo y de los servicios de los metaobjetos bajo demanda: si algo no es adaptable, se modifica la especificación del protocolo o del metaobjeto y se vuelve a generar la aplicación.

- El criterio propuesto por Golm elimina la adaptabilidad dinámica sin restricciones buscada en esta Tesis y, adicionalmente, supone la generación de distintas versiones del intérprete, y por tanto del lenguaje, perdiendo así la portabilidad del código existente con anterioridad.

- La estructura establecida en los sistemas que utilizan MOPs es siempre dependiente de un único lenguaje de programación.

9.2 Sistema Computacional Reflectivo Sin Restricciones

Apoyándonos en la definición de un sistema reflectivo como aquél que puede acceder a niveles computacionales inferiores dentro de su torre de interpretación [Wand88] y buscando la mayor flexibilidad computacional de nuestro sistema, hemos limitado la modificación de la semántica computacional a un nivel (§ 8.2.4) –dos grados de altura–, pero obligando a que ésta permita modificar la totalidad de sus computaciones –grado de anchura ilimitado. Esto quiere decir que sólo son necesarios dos niveles de interpretación, pero que el nivel superior ha de poder modificar cualquier característica computacional del nivel subyacente que le da vida.

Recordando los objetivos marcados para esta capa del sistema (§ 8.2), podemos abreviar enunciando las principales características que ha de aportar frente a un sistema basado en un MOP:

- No debe ser necesario estipular lo que va a adaptarse previamente a la ejecución de la aplicación.

- Debe existir un mecanismo de expresividad en el que cualquier característica del sistema pueda adaptarse dinámicamente –horizontalidad total.

- El sistema debe ser totalmente independiente del lenguaje.

- Los grados de flexibilidad a conseguir son: introspección, estructural, semántica y lenguaje.

El esquema a seguir se muestra en la siguiente figura:

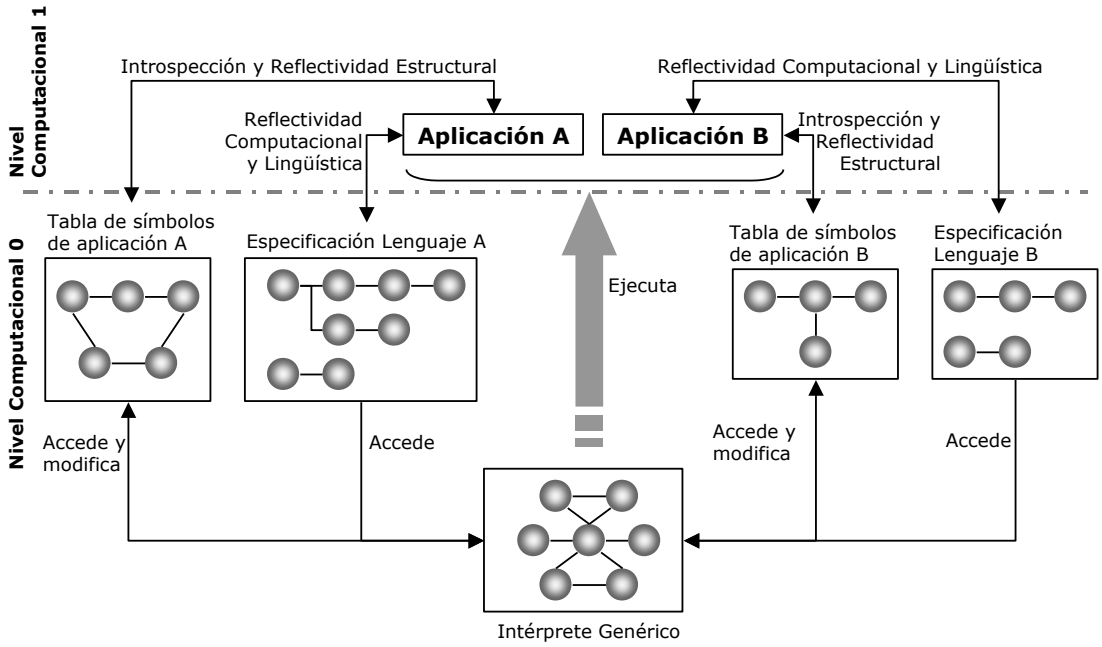


Figura 9.2. Esquema general del sistema computacional no restrictivo.

Una de las diferencias frente a los intérpretes convencionales es la creación de un intérprete genérico independiente del lenguaje. Este procesador ha de ser capaz de ejecutar cualquier aplicación escrita sobre cualquier lenguaje de programación. De este modo, la entrada a este programa no se limita, como en la mayoría de los intérpretes, a la aplicación a ejecutar; está parametrizado además con la especificación del lenguaje en el que dicha aplicación haya sido codificada.

Implementando un intérprete genérico en el que las dos entradas son la aplicación a procesar y la especificación del lenguaje en el que ésta haya sido escrita, conseguimos hacer que el sistema computacional sea independiente del lenguaje.

Para cada aplicación evaluada, el intérprete genérico deberá crear un contexto de ejecución o tabla de símbolos dinámica, en el que aparezcan todos los objetos[22] creados a raíz de ejecutar el programa. Se trata de ubicar todos los objetos creados en el contexto de ejecución de la aplicación en un único espacio de nombres.

La flexibilidad del sistema se obtiene cuando, en tiempo de ejecución, la aplicación accede a la especificación de su lenguaje, o bien a su tabla de símbolos existente. El resultado de estos accesos supone distintos grados de flexibilidad:

- Si analiza, sin modificar, su tabla de símbolos supondrá introspección.
- En el caso de modificar los elementos de su tabla de símbolos, la flexibilidad obtenida es reflectividad estructural.
- El acceso y modificación de la semántica propia de la especificación de su lenguaje de programación significará reflectividad computacional.

[22] El término objeto aquí denota cualquier elemento o símbolo existente en el contexto de ejecución de la aplicación. Un objeto puede ser, por tanto, una función, variable, clase u objeto propiamente dicho.

- La modificación de las características léxicas o sintácticas de su lenguaje producirán una adaptación lingüística.

El modo en el que las aplicaciones de usuario accedan a su tabla de símbolos y a la especificación de su lenguaje, no deberá poseer restricción alguna. La horizontalidad de este acceso ha de ser plena.

Para conseguir lo propuesto, la principal dificultad radica en la separación de los dos niveles computacionales mostrados en la Figura 9.2. El nivel computacional de aplicación poseerá su propio lenguaje y semántica, mientras que el nivel subyacente que le da vida, el intérprete, poseerá una semántica y lenguaje no necesariamente similar. El núcleo de nuestro sistema se centra en un salto computacional del nivel de aplicación al de interpretación; ¿cómo puede una aplicación modificar, sin restricción alguna, el intérprete que lo está ejecutando?

9.2.1 Salto Computacional

La raíz computacional de nuestro sistema está soportada por un motor computacional, que podrá ser una máquina abstracta, una máquina física o un lenguaje de programación dotado de una serie de requisitos computacionales. Sobre su dominio computacional, se desarrolla el intérprete genérico independiente del lenguaje de programación.

La especificación de cada lenguaje a interpretar se llevará a cabo mediante una estructura de objetos, representativa de su descripción léxica, sintáctica y semántica. Este grupo de objetos sigue el modelo computacional descrito por la máquina y, por tanto, pertenece también a su espacio computacional.

En la ejecución de una aplicación por un intérprete, éste siempre debe mantener dinámicamente una representación de sus símbolos en memoria. Como describíamos en el punto anterior, el intérprete genérico ofrecerá éstos al resto de aplicaciones existentes en el sistema. De esta forma, por cada aplicación ejecutada por el intérprete, se ofrecerá una lista de objetos representativos de su tabla de símbolos.

Siguiendo este esquema, y como se muestra en la Figura 9.3, el dominio computacional del motor computacional incluye el intérprete genérico y, por cada aplicación, el conjunto de objetos que representan la especificación de su lenguaje y la representación de sus símbolos existentes en tiempo de ejecución.

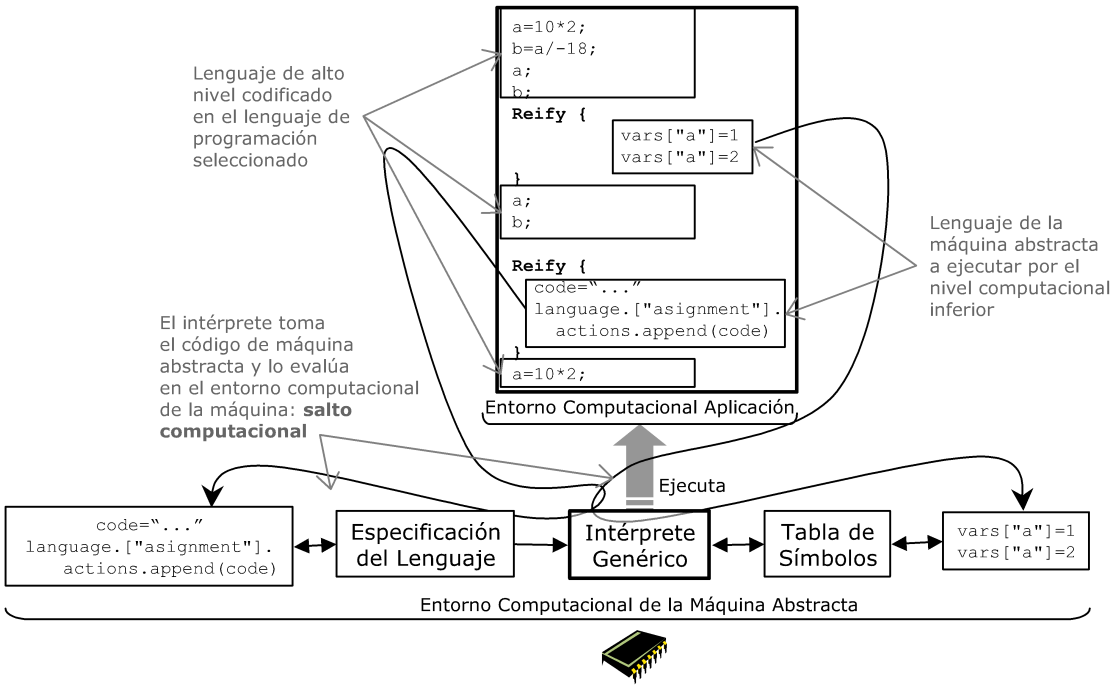


Figura 9.3. Salto computacional producido en la torre de intérpretes.

En el espacio computacional del intérprete se encuentran las aplicaciones de usuario codificadas en distintos lenguajes de programación. Cualquiera que sea el lenguaje de programación utilizado, una aplicación de usuario tiene siempre una instrucción de acceso al metasistema: `reify`. Dentro de esta instrucción (entre llaves) el programador podrá utilizar código de su nivel computacional inferior: código propio del motor computacional. Así, una aplicación poseerá la expresividad de su lenguaje más la propia del motor computacional.

Siempre el que intérprete genérico analice una instrucción `reify` en un programa de usuario, en lugar de evaluarla como una sentencia propia del lenguaje interpretado, seguirá los siguientes pasos:

- Tomará la consecución de instrucciones codificadas en el lenguaje del motor computacional, como una cadena de caracteres.
- Evaluará o descosificará los datos obtenidos, para que sean computados como instrucciones por el motor computacional.

El resultado es que el código de usuario ubicado dentro de la instrucción de cosificación es ejecutado por el nivel computacional inferior al resto de código de la aplicación. Es en este momento en el que se produce un salto computacional real en el sistema.

Para que la implementación del mecanismo descrito anteriormente sea factible, la posibilidad de evaluar o descosificar dinámicamente cadenas de caracteres como computación será uno de los requisitos impuestos al motor computacional.

La interacción directa entre aplicaciones es otro de los requisitos impuestos a la plataforma virtual en la descripción global de la arquitectura del sistema. Cualquier aplicación, desarrollada en cualquier lenguaje, podrá interaccionar directamente –sin necesidad de una capa software adicional– con el resto de aplicaciones existentes. De este modo, el código de la aplicación propio de la instrucción `reify`, al ser ejecutado en el entorno computacional de la máquina, podrá acceder a cualquier

aplicación dentro de este dominio, y en concreto a su tabla de símbolos y a la especificación de su lenguaje:

- Si analiza, mediante la introspección ofrecida por el motor computacional, su propia tabla de símbolos, estará obteniendo información acerca de su propia ejecución: introspección de su propio dominio computacional.
- Si modifica la estructura de alguno de sus símbolos, haciendo uso de la reflectividad estructural de la máquina, el resultado es reflectividad estructural de su nivel computacional.
- La modificación, mediante la utilización de la reflectividad estructural de la máquina, de las reglas semánticas del lenguaje de programación supone reflectividad computacional o de comportamiento.
- Si la parte a modificar de su lenguaje es su especificación léxica o sintáctica, el resultado obtenido es reflectividad lingüística del nivel computacional de usuario.

Vemos cómo otros dos requisitos necesarios en el motor computacional para llevar a cabo los procesos descritos son introspección y reflectividad estructural de su dominio computacional.

Finalmente comentaremos que la evaluación de una aplicación debe realizarse por el intérprete genérico analizando dinámicamente la especificación de su lenguaje, de forma que la modificación de ésta conlleve automáticamente al reflejo de los cambios realizados. De este modo, no es necesaria la implementación de un mecanismo de conexión causal (§ 6.1) ni la duplicación de información mediante metaobjetos, puesto que el intérprete ejecuta la aplicación derogando parte de su evaluación en la representación de su lenguaje.

9.2.2 Representación Estructural de Lenguajes y Aplicaciones

El salto computacional real ofrecido por nuestro sistema cobra importancia en el momento en el que la aplicación de usuario accede a las estructuras de objetos representantes de su lenguaje de programación o de su tabla de símbolos. Indicaremos brevemente cómo pueden representarse éstos para que su manipulación, gracias a la reflectividad estructural del motor computacional, sea posible. Una descripción más detallada puede obtenerse del diseño del prototipo realizado en el Capítulo 11.

La representación de los lenguajes de programación a utilizar en nuestro sistema se lleva a cabo mediante estructuras de objetos que representan gramáticas libres de contexto, para las descripciones léxicas y sintácticas, y rutinas semánticas expresadas mediante código de la plataforma virtual. Su expresividad es la propia de una definición dirigida por sintaxis [Aho90], utilizando el lenguaje del motor computacional como lenguaje de descripción semántica.

Para liberar al usuario de la necesidad de crear estas estructuras de objetos, se diseña un metalenguaje de descripción de lenguajes de programación con las características mencionadas. El procesamiento de código expresado mediante este metalenguaje supondrá la creación de la estructura de objetos representativa del lenguaje especificado.

A modo de ejemplo, mostraremos la traducción de la siguiente definición dirigida por sintaxis:

Producción	Rutina (regla) Semántica
`S ::= B S`	`r1: Regla en Código motor computacional`
	`r2: Regla en Código motor computacional`
`| C D`	`r3: Regla en Código motor computacional`
`B ::= "B"`	
`C ::= "C"`	
`D ::= "D"`	

Figura 9.4. Ejemplo de definición dirigida por sintaxis.

Los símbolos terminales se han mostrado entre comillas dobles. Las reglas semánticas `r1`, `r2` y `r3` se codificarán mediante el lenguaje del motor computacional. Las producciones cuyo símbolo gramatical a su izquierda es `S`, forman parte de la descripción sintáctica del lenguaje. Las tres últimas reglas representan su especificación léxica.

El procesamiento de la descripción del lenguaje anterior, crea su especificación mediante objetos siguiendo la estructura mostrada en la Figura 9.5. Cada una de las reglas posee un objeto que representa el símbolo gramatical no terminal de su parte izquierda. Éste está asociado a tantos objetos como partes derechas posea dicha producción. Cada una de las partes derechas hace referencia a una lista de símbolos gramaticales –parte derecha de la producción– y a una lista con todas las regla semánticas asociadas.

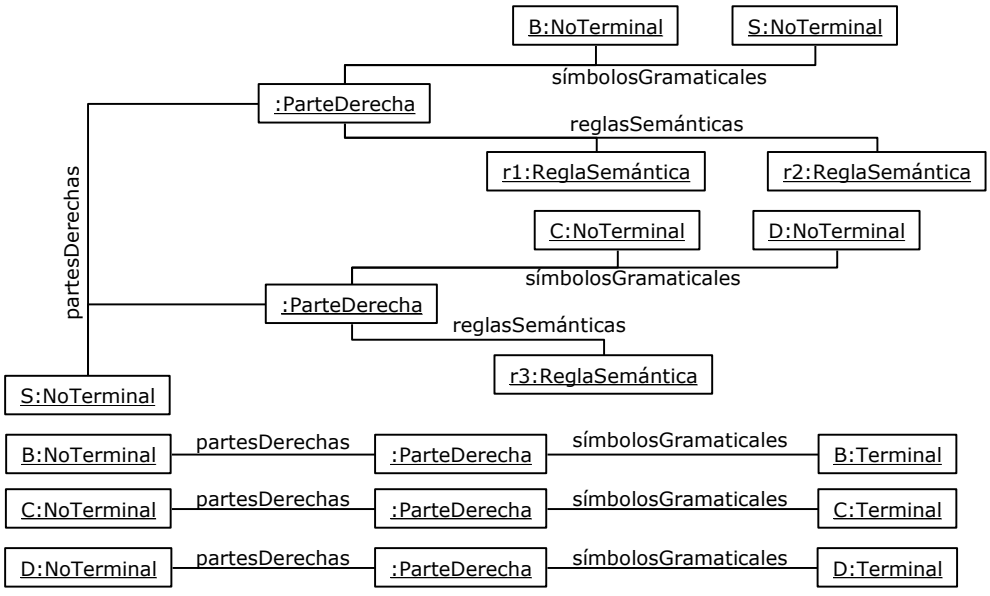


Figura 9.5. Especificación de un lenguaje de programación mediante una estructura de objetos.

Una vez creada esta estructura, se analizará una aplicación codificada mediante el lenguaje descrito, utilizando un algoritmo descendente de análisis sintáctico [Aho90]. Conforme se vaya examinando ésta, se irá creando su árbol sintáctico. Cada nodo de este árbol guardará una referencia a la representación de su lenguaje de programación. En concreto, para cada nodo propio de un símbolo no terminal,

se establecerá una relación con la parte derecha de la producción elegida en la creación del árbol –véase la Figura 9.6.

La evaluación de la aplicación supone el recorrido del árbol, ejecutando las reglas semánticas asociadas a la descripción de su lenguaje. La relación establecida entre el árbol, representante de la aplicación, y la estructura que especifica su lenguaje de programación, supone que la modificación del lenguaje implique automáticamente su actualización o reflejo en la aplicación de usuario; esta conexión es conocida como conexión causal [Maes87b].

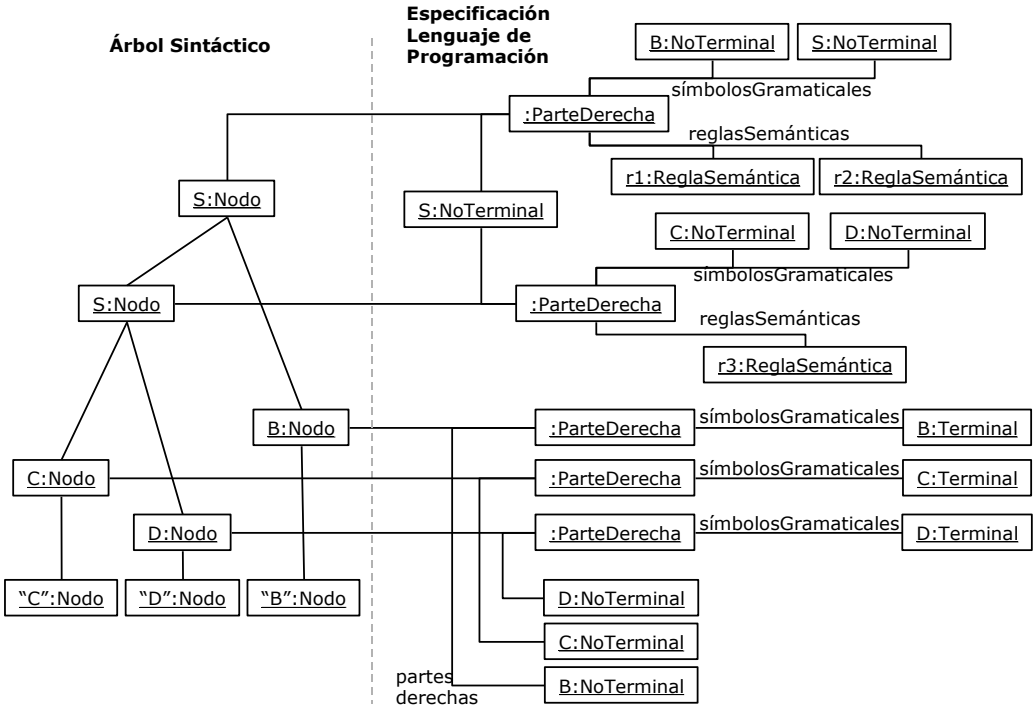


Figura 9.6. Creación del árbol sintáctico asociado a la especificación de un lenguaje.

El modo en el que se recorre el árbol no sigue un algoritmo predeterminado, como por ejemplo el propio de un esquema descendente de traducción [Aho90]. La regla semántica del símbolo inicial de la gramática, ha de indicar cómo se evalúa la ejecución del primer nodo del árbol; este proceso se extiende de un modo recursivo al resto de elementos del árbol.

Por cada aplicación en nuestro sistema existirá un objeto representante de ésta (Figura 9.7). Cada objeto de aplicación poseerá una referencia a su árbol sintáctico, a su lenguaje de programación, y a su tabla de símbolos dinámica –contexto de ejecución.

Accediendo a la lista de objetos representante de su tabla de símbolos, mediante la introspección y reflectividad estructural ofrecida por el motor computacional, se le brinda al programador de aplicaciones introspección y reflectividad estructural del lenguaje de programación que haya seleccionado.

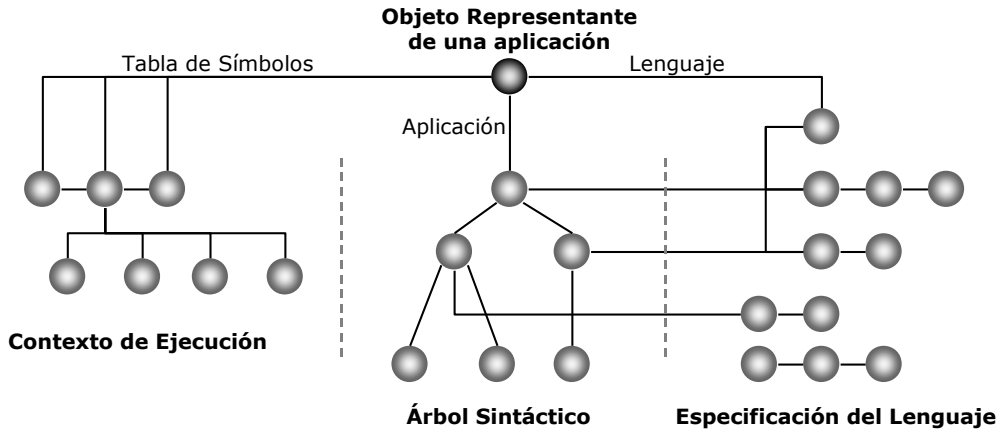


Figura 9.7. Principal información existente para cada aplicación en ejecución.

9.3 Beneficios del Sistema Presentado

Enlazando con el análisis realizado en § 9.1 de técnicas utilizadas para obtener sistemas computacionales flexibles, éste es el conjunto de beneficios aportados frente a los sistemas basados en MOPs:

No se establece un protocolo que restringe previamente las características susceptibles de ser adaptadas. Cualquier aplicación puede modificar cualquier característica de su lenguaje de programación y de su contexto de ejecución, sin restricción alguna, en tiempo de ejecución.

- No existe una expresividad limitada. El modo en el que podemos modificar una aplicación se expresa mediante un lenguaje de computación: el lenguaje del motor computacional. De este modo, la expresividad ofrecida es en sí un lenguaje de acceso y modificación de lenguajes, cuya evaluación supone un salto real en los dos niveles de computación existentes.

- Independencia del lenguaje. La separación de la especificación del lenguaje de programación a interpretar, del intérprete en sí, supone una independencia del lenguaje en el esquema ofrecido. Las características ofrecidas por el sistema no implican la utilización de un lenguaje de programación específico. Sin embargo, el lenguaje de adaptación dinámica siempre es el mismo: el lenguaje del motor computacional.

- Se ofrece cuatro niveles de adaptabilidad: introspección, estructura dinámica, comportamiento y lenguaje.

- Existe una conexión causal directa sin necesidad de duplicar información. La utilización de metaobjetos supone:

  - Duplicidad de información para ofrecer características del sistema. Cualquier característica adaptable es ofrecida al programador a través de un metaobjeto. Un objeto adaptable implica así la creación paralela de un metaobjeto.

  - Implementación de un mecanismo de conexión causal. Los reflejos de actualización de metaobjetos en el sistema base han de implementarse mediante un mecanismo adicional.

Ambas restricciones no aparecen en el sistema presentado.

- Adaptabilidad cruzada. Cualquier aplicación puede adaptar dinámicamente a otra existente en el sistema, sin necesidad de que ambas utilicen el mismo lenguaje de programación. Esta posibilidad atribuye a nuestro sistema la capacidad de constituirse como un entorno computacional de separación de incumbencias (*concerns*) en tiempo de ejecución y sin restricción alguna.

Como punto negativo, los sistemas basados en MOPs poseen una mayor eficiencia en tiempo de ejecución. El hecho de duplicar la información a reflejar mediante el uso de metaobjetos y la restricción impuesta por el uso de un MOP, se justifica mediante el diseño seguido en el que sólo se usa un nivel computacional – un único intérprete. Si el usuario de un MOP no identifica ningún elemento como adaptable, no existe casi penalización en los tiempos de ejecución. Sin embargo, nuestro esquema de dos niveles de computación hace que el sistema posea tiempos de ejecución más elevados, indistintamente de la utilización o no de las características reflectivas.

9.4 Requisitos Impuestos al Motor Computacional

Analizando los requisitos necesarios impuestos al motor computacional para poder desarrollarse el sistema reflectivo propuesto, podemos enumerarlos del siguiente modo:

- Introspección. Para que el programador pueda conocer la tabla de símbolos existente en la ejecución de su aplicación y la especificación del lenguaje de programación utilizado, la plataforma computacional deberá poseer características introspectivas.

- Evaluación o descosificación de datos (programación generativa). El intérprete genérico toma el código del motor computacional a evaluar como una cadena de caracteres. Éste deberá ser interpretado por la máquina como instrucciones, no como datos. Este mecanismo de conversión de datos a computación es necesario, pues, para desarrollar el sistema presentado.

- Reflectividad estructural. El código del motor computacional evaluado en el nivel computacional subyacente al propio de la aplicación, requiere esta característica para poder adaptar la especificación de su lenguaje y los objetos existentes en su contexto de ejecución.

- Interacción directa entre aplicaciones. Puesto que el código a descosificar por el motor computacional accede a otras aplicaciones desarrolladas sobre ésta (especificación del lenguaje y tabla de símbolos), la interacción entre aplicaciones computadas por la máquina es necesaria. Su uso también queda patente en la adaptación de una aplicación por otra existente en el sistema.

Un entorno computacional que ofrezca estas características será válido para desarrollar sobre él sistema reflectivo no restrictivo propuesto. El diseño de éste sistema será descrito en el Capítulo 11.

Capítulo 10
ARQUITECTURA DEL SISTEMA DE PERSISTENCIA

En el capítulo anterior hemos presentado la arquitectura del sistema reflectivo no restrictivo. En este capítulo estudiaremos la segunda capa de la arquitectura del sistema presentada en el capítulo 8, la correspondiente al sistema de persistencia. Esta capa es la encargada de, utilizando los servicios del sistema reflectivo no restrictivo, ofrecer persistencia a las aplicaciones de una manera totalmente transparente y adaptable.

La arquitectura del sistema de persistencia presenta una composición modular. En este capítulo se estudiarán las características de cada módulo. El funcionamiento global del sistema de persistencia se entenderá a partir las interacciones que se producen entre los diferentes módulos y entre éstos y el sistema reflectivo no restrictivo.

10.1 Módulos del Sistema de Persistencia

La arquitectura del sistema de persistencia comprende tres subsistemas:

- Aplicación. Este módulo ofrece la representación de la estructura de los programas en ejecución. Para ello, define un modelo de objetos único que represente cualquier aplicación en ejecución, independientemente del lenguaje en el que haya sido desarrollada.

- Intérprete. Es el responsable de llevar a cabo el análisis y ejecución de cada aplicación. Para ello, representa los programas en ejecución utilizando el modelo de objetos independiente definido por el módulo de Aplicación y ejecuta dicho modelo.

- Persistencia. Este es el módulo principal de la arquitectura del sistema de persistencia. Se encarga de adaptar dinámicamente las aplicaciones en ejecución para hacerlas persistentes de un modo transparente.

En la Figura 10.1 se representan estos subsistemas y sus relaciones. Se trata de una vista UML estática que representa estos módulos y sus relaciones. Es importante señalar que no existe dependencia entre las aplicaciones ejecutadas y el módu-

lo que les proporciona persistencia. Esta independencia se conseguirá modificando las aplicaciones y el intérprete en tiempo de ejecución utilizando las características reflectivas de la plataforma no restrictiva (§ 9.3).

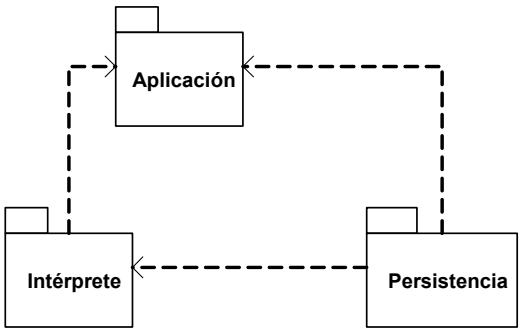


Figura 10.1. Módulos del sistema de persistencia.

10.2 Aplicación

El módulo de aplicación deberá ofrecer un modelo de objetos que permita representar el programa en ejecución. Este modelo de objetos constituirá el modelo computacional único y será independiente del lenguaje y la plataforma utilizados para desarrollar la aplicación (§ 8.2.3).

La elección de este modelo de objetos es una tarea crítica en el desarrollo del sistema puesto que éste será el modelo a utilizar por la totalidad de los lenguajes de programación. Dado el éxito de las tecnologías orientadas a objetos, se propondrá un modelo de computación que permita representar modelos orientados a objetos. El diseño de de este modelo no deberá estar restringido a las características particulares de ningún lenguaje de programación. Deberá ser lo suficientemente flexible como para adaptarse a diferentes aproximaciones a la orientación a objetos, por ejemplo, lenguajes con un sistema de tipos totalmente dinámico como Ruby, [Thomas2004], Python [Rossum2001] o lenguajes con comprobación estática de tipos como Java [Gosling96].

En la Figura 10.2 se muestra, a modo de ejemplo, un sencillo modelo computacional. Se muestran las clases de los objetos que permiten representar modelos de objetos y sus relaciones. Puede observarse cómo una `Clase` puede tener `Atributos` y `Métodos`, una `Instancia` tiene una `Clase` asociada y un `Constructor` será un `Método`. Podría desarrollarse un modelo computacional mucho más elaborado tomando como referencia el metamodelo de UML [OMG2003]. Aunque es habitual asociar UML únicamente con su notación, en la especificación de UML se define formalmente la semántica del lenguaje mediante la elaboración de un metamodelo[23]. Además esta definición se realiza utilizando UML, es decir, UML se utiliza a sí mismo para definirse.

[23] Los modelos utilizados para representar modelos se denominan habitualmente metamodelos. El metamodelo de UML hace referencia al modelo de un lenguaje que sirve para modelar. En el caso del modelo computacional único también se trata de un modelo de objetos que representa a su vez modelos de objetos. Por lo tanto también sería correcto denominarlo "metamodelo".

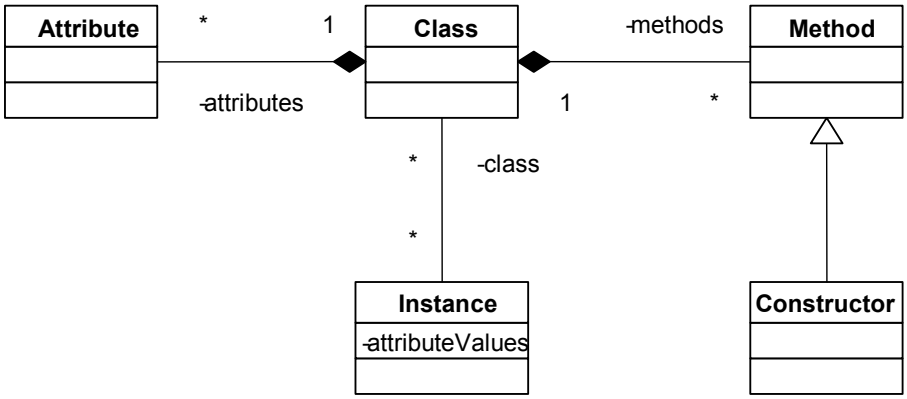


Figura 10.2. Diagrama de clases de un modelo computacional de ejemplo.

Tomando el modelo computacional de la Figura 10.2, se va a ilustrar cómo podrían representarse dos programas codificados con distintos lenguajes utilizando el modelo computacional único. En la Figura 10.3 se muestra el código de dos programas escritos respectivamente en Java y Python. Ambos programas definen una clase con un atributo y un método y posteriormente instancian un objeto de esta clase. El intérprete será el encargado de transformar estos programas utilizando el modelo computacional definido. Como resultado de esta traducción se obtienen los dos programas representados por el mismo modelo computacional. Los programas traducidos se representan en la figura por diagramas de objetos que muestran cómo los diferentes elementos del lenguaje se representan como instancias del mismo modelo de clases definido en la Figura 10.2.

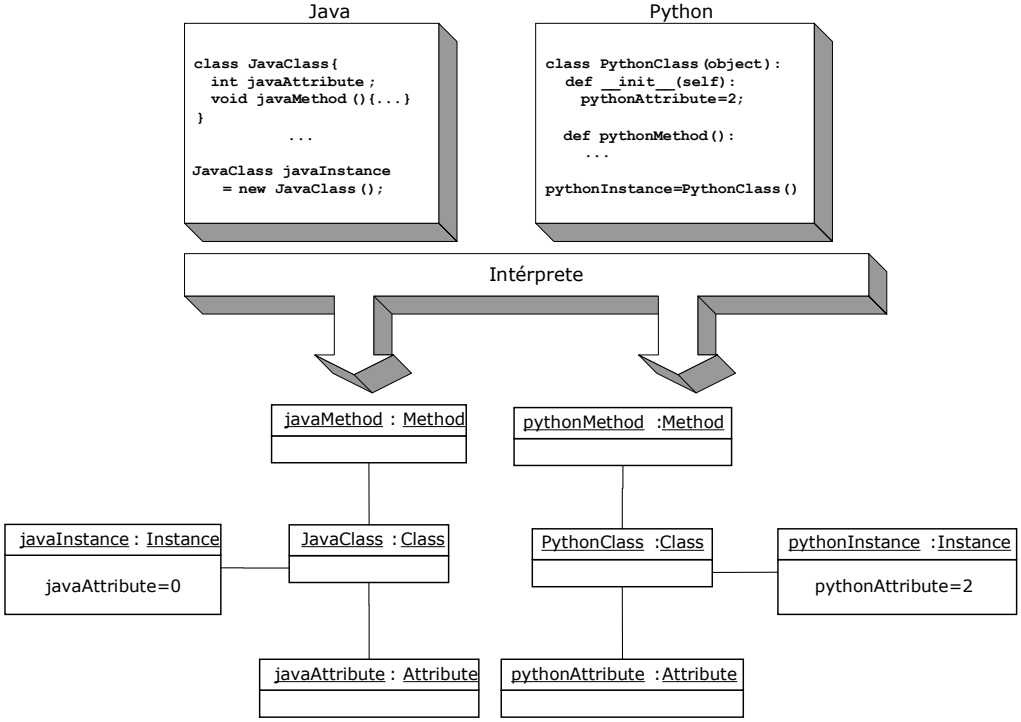


Figura 10.3. Representación de lenguajes utilizando un modelo computacional único.

El modelo de objetos que represente los programas en ejecución debe poseer las siguientes características:

- Introspección. Mediante el análisis de la estructura que representa a los programas se obtendrá introspección. El conocimiento de la estructura

de las aplicaciones y lenguajes en tiempo de ejecución es primordial a la hora de implementar un sistema de persistencia adaptable y adaptativo. Recordemos que en § 8.2.5 especificábamos los requisitos computacionales de la plataforma de desarrollo. En § 8.2.5.2 requeríamos introspección; aplicando ésta al sistema subsistema de aplicación, obtendremos introspección para cualquier lenguaje de programación gracias a la utilización de un modelo computacional único (§ 8.2.3).

- Adaptable. La modificación dinámica de los objetos producirá reflectividad estructural. Será necesaria para que el sistema de persistencia pueda manipular en tiempo de ejecución de un modo sencillo la estructura de las aplicaciones. Éste era, de hecho, el requisito impuesto a la plataforma de desarrollo en § 8.2.5.3.

- Ejecutable. Debe contener la información suficiente como para poder ser ejecutado por el Intérprete obteniendo los mismos resultados que si el programa hubiese sido ejecutado de forma nativa.

Estas características, junto con la posibilidad de obtener reflectividad computacional ofrecida por el subsistema intérprete (§ 10.3), permitirán implementar la funcionalidad de persistencia como una incumbencia totalmente separada de la aplicación ejecutada (§ 10.4.2). Además, la utilización de una única representación de los programas en ejecución permitirá que el sistema de persistencia sea independiente del lenguaje utilizado para codificar los programas.

10.3 Intérprete

El módulo Intérprete será el encargado de, dada una aplicación escrita en un lenguaje determinado:

- Construir el modelo de objetos que represente el programa utilizando el modelo computacional único.

- Ejecutar el modelo construido y permitir la modificación de su estructura y semántica.

10.3.1 Traducción del Programa al Modelo Computacional Único

Para la construcción del modelo de objetos independiente del lenguaje se hará uso del intérprete genérico del sistema reflectivo no restrictivo (§ 9.2). Este intérprete recibe la aplicación a ejecutar parametrizada con la especificación del lenguaje de programación en el que está escrita. Se extenderá la especificación del lenguaje utilizado para que, durante la interpretación de la aplicación, se genere el modelo de objetos independiente.

Para la generación del modelo de objetos independiente del programa se hará uso del módulo Aplicación (§ 10.2) que definirá el metamodelo del modelo computacional único. En la Figura 10.4 se ilustra este proceso tomando como entrada el programa Java utilizado en el ejemplo de la Figura 10.3. La especificación del modelo computacional único es una especificación de lenguaje para el intérprete genérico que, en lugar de indicar cómo se interpreta el programa Java, indica cómo se construye el modelo de objetos independiente que lo representa. La especificación del lenguaje se apoyará en estos servicios parar traducir cualquier programa al

modelo único computacional. Nótese que, aunque en este paso no se ejecute el programa, el modelo de objetos construido debe contener la información suficiente como para ser ejecutado posteriormente (§ 10.2).

Gracias al intérprete genérico[24] del sistema reflectivo no restrictivo, se puede construir el modelo que represente los programas con independencia del lenguaje utilizado para codificarlos. Para programas escritos en diferentes lenguajes, lo único que habrá que modificar es la especificación del lenguaje con la que se alimenta al intérprete genérico, con el fin de indicar cómo debe construirse el modelo de objetos haciendo uso del módulo Aplicación, descrito en el punto anterior. Las rutinas semánticas de la especificación de este nuevo lenguaje harán uso de los servicios de construcción del modelo computacional único, traduciendo sus programas al modelo computacional utilizado por el sistema de persistencia. Este proceso es similar al que se produce en los sistemas de desarrollo de software basados en modelos [OMG2001].

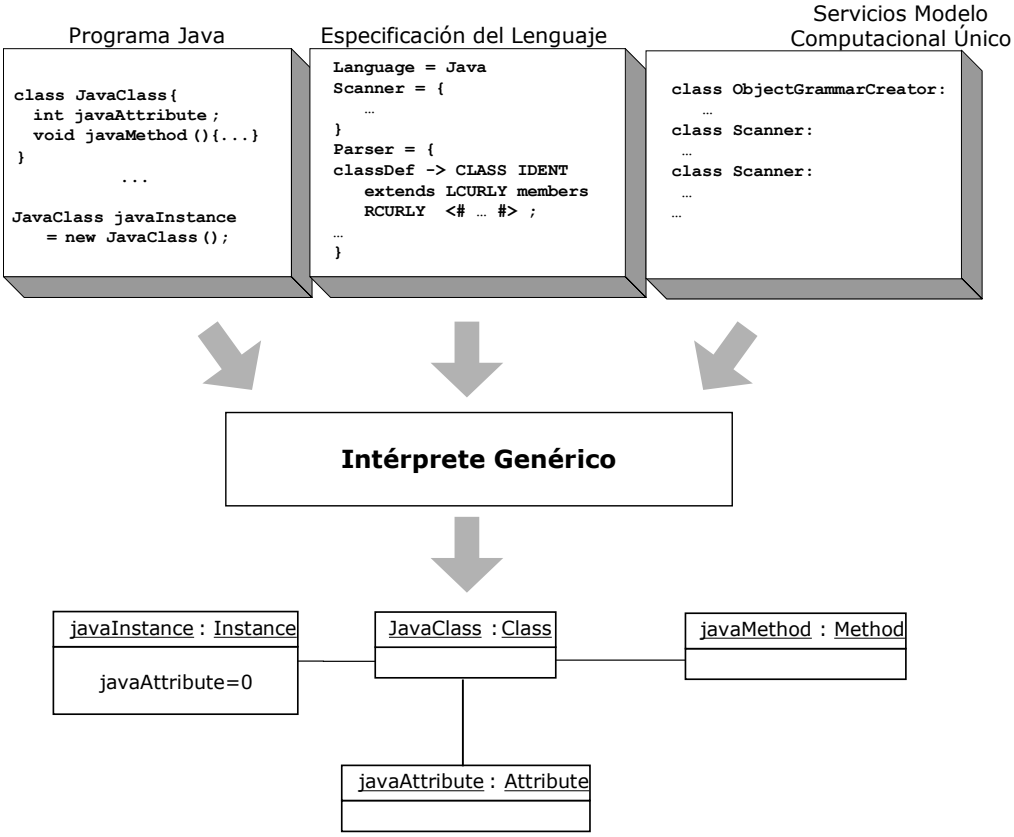


Figura 10.4. Traducción de un programa al modelo computacional único.

10.3.2 Ejecución del Modelo Independiente del Lenguaje

El intérprete será el encargado de ejecutar el modelo de objetos independiente que representa a los programas a ejecutar.

El mecanismo de interpretación utilizado debe ofrecer una característica fundamental: la posibilidad de modificar cómo se realiza la interpretación de los programas (reflectividad computacional). Sólo así podrá modificarse la semántica de determinadas facetas computacionales para hacer implícitas todas las rutinas de per-

[24] Puede obtenerse una descripción detallada de cómo funciona el intérprete genérico en el Capítulo 11.

sistencia (§ 7.6). Dado que el proceso de interpretación del programa implica procesar los objetos del modelo para ejecutar las acciones correspondientes, este requisito se traduce en la necesidad de modificar las acciones a realizar en cada procesamiento.

Un patrón de diseño habitualmente utilizado en procesadores de lenguaje es el *Visitor* [GOF94]. Este patrón está especialmente indicado para recorrer estructuras jerárquicas de objetos y realizar diferentes acciones con cada uno de ellos dependiendo de algún discriminante (habitualmente su tipo). Se explicará el funcionamiento y las ventajas de este patrón con un ejemplo.

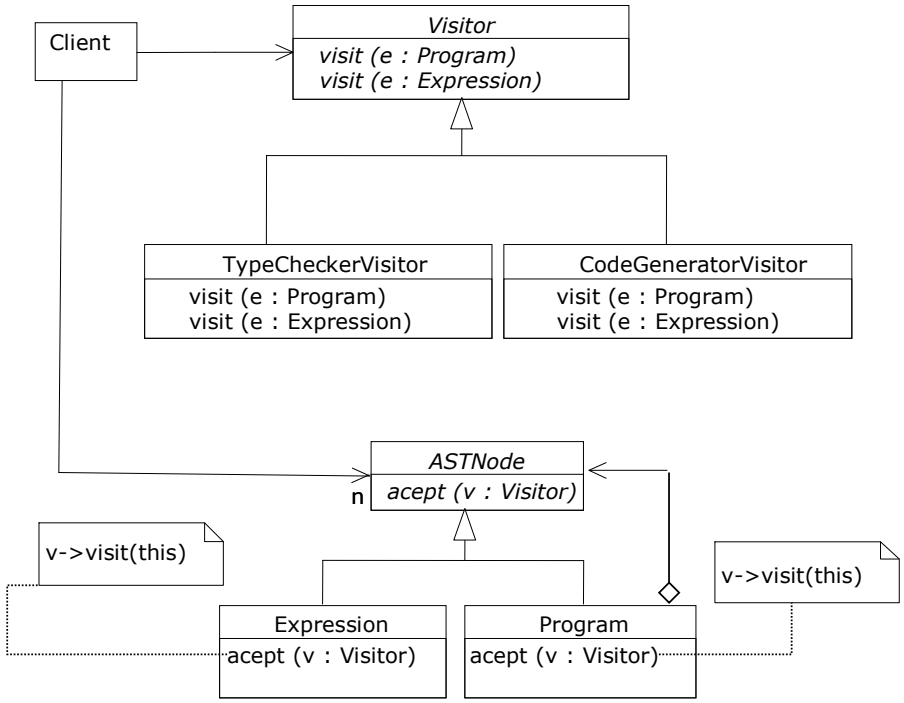


Figura 10.5. Estructura del patrón Visitor.

En el ejemplo de la Figura 10.5 se puede ver la estructura de este patrón [GOF94] aplicada sobre un árbol AST (*abstract syntax tree*) [Ortín2004b]. Dado un programa representado como un conjunto de nodos (objetos `Node`), propone preparar la estructura de nodos para aceptar el paso de un objeto `Visitor`. Las operaciones que puede realizar un *visitor* las define el interfaz `Visitor` donde se incluye un método `visit()` por cada tipo de nodo que vaya a ser tratado en la estructura. Los nodos simplemente incorporan un método `accept()` donde reciben un objeto `Visitor` sobre el que invocan el método de visita que corresponde con su tipo[25]. De este modo, una vez preparada la estructura para aceptar el paso del *visitor*, todas las operaciones a realizar se implementarán sobre las diferentes implementaciones del interfaz `Visitor`. Las acciones a realizar quedan separadas de los objetos sobre las que se realizan. El orden del recorrido de los objetos también será responsabilidad de los diferentes *visitors*[26].

[25] El ejemplo de la figura hace uso de la sobrecarga de métodos. Si el lenguaje de programación no soportase esta característica deberían diferenciarse los diferentes métodos de visita por su nombre. En el capítulo 12 estudiaremos una implementación del patrón simplificada gracias a las características dinámicas del lenguaje Python.

[26] La responsabilidad de establecer el orden del recorrido de los objetos podría extraerse de los *visitors* haciendo uso del patrón *Iterator* [GOF94].

Además de las ventajas señaladas, una característica que hace especialmente indicado este patrón en el contexto de nuestro sistema es que permite afrontar de una manera elegante el problema de cambiar cómo se interpretan los programas: con la modificación de los métodos de visita (`visit`) con los que el *visitor* realiza las acciones durante la interpretación. La modificación dinámica de estos métodos supondría reflectividad computacional (de comportamiento). Nótese cómo esta modificación es factible gracias al requisito § 8.2.5.4 que impone a la plataforma de desarrollo la funcionalidad de generar y ejecutar código en tiempo de ejecución.

10.4 Sistema de Persistencia

El sistema de persistencia es el encargado de ofrecer persistencia a las aplicaciones de una manera totalmente transparente. Este módulo presenta una serie de componentes funcionales que permitirán configurar su comportamiento dinámicamente. Los componentes de persistencia existentes, así como nuevos componentes de persistencia que no hayan sido tenidos en cuenta en tiempo de diseño, pueden ser añadidos, extraídos y modificados en tiempo de ejecución. Además, permite configurar un número arbitrario de sistemas y formatos de almacenamiento, incluso para permitir su uso simultáneamente.

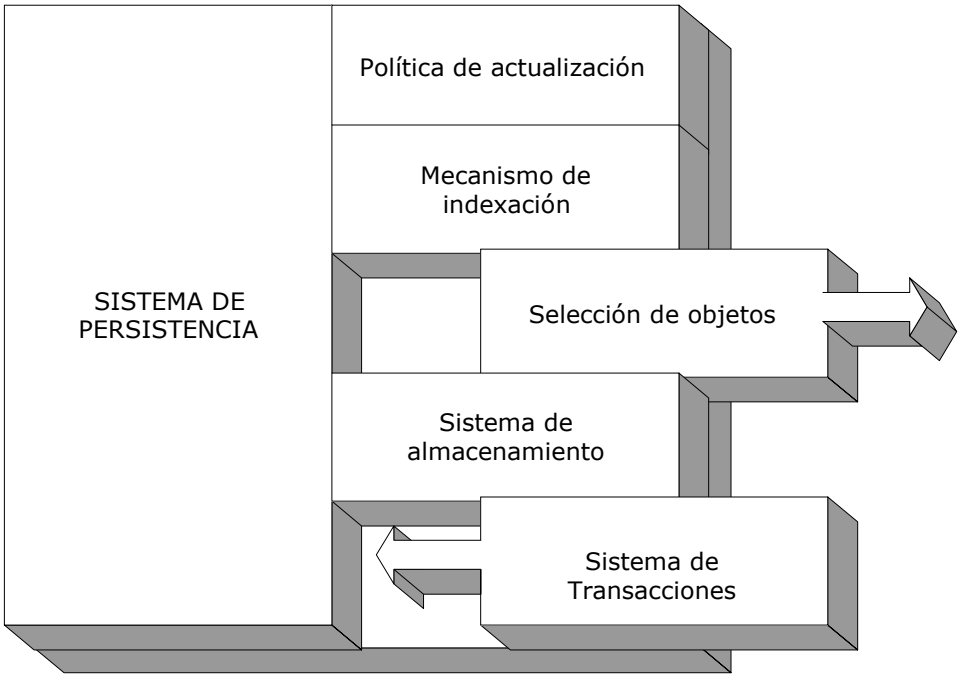


Figura 10.6. Arquitectura modular del sistema de persistencia.

10.4.1 Componentes de Persistencia

El diseño del sistema de persistencia permitirá configurar el funcionamiento de los diferentes módulos de una manera sencilla. En concreto, se ofrecerán diversas opciones dentro de cada componente de persistencia para facilitar la configuración por parte del usuario.

El diseño del sistema contemplará diferentes componentes de persistencia así como los mecanismos que permitan su configuración de una manera sencilla. Pero el sistema deberá además permitir, en tiempo de ejecución, la adición de nue-

vos componentes de persistencia que no hayan sido tenidos en cuenta en tiempo de diseño (como, por ejemplo, un sistema de transacciones implícito).

Dado que el sistema será desplegado sobre la plataforma reflectiva no restrictiva, podrán utilizarse sus características de reflectividad para adaptar el funcionamiento del sistema de persistencia.

El sistema ofrecerá, al menos, los siguientes componentes de persistencia:

10.4.1.1 Selección de objetos.

Módulo mediante el cual pueden seleccionarse qué objetos, dentro de una determinada aplicación, son persistentes. Como hemos estudiado a lo largo del Capítulo 3, existen múltiples alternativas empleadas para indicar qué objetos han de ser persistentes; ejemplos son:

- Cierre persistente transitivo o persistencia por alcance. Todos los objetos referenciados, directa o indirectamente, desde un objeto persistente son hechos persistentes [Atkinson96].
- Persistencia en cascada. Se basa en el mismo principio que la persistencia por alcance, pero permite especificar una estrategia de propagación en cascada para cada asociación, ofreciendo una mayor flexibilidad y control para las transiciones entre los estados de persistencia [Hibernate2005].
- Selección puntual de objetos. Este mecanismo es el más básico, requiriendo que el usuario, o la propia aplicación programáticamente, identifique todos y cada uno de los objetos que han de persistir. Este mecanismo se puede combinar con los anteriores; por ejemplo, el mecanismo de serialización de Java combina esta alternativa con la persistencia por alcance [Sun97c].
- Selección mediante estrategias. En [Martínez2001] se propone un diseño orientado a objetos, en le que se parametriza la selección de objetos y la selección de la técnica de indexación mediante la utilización del patrón de diseño *Strategy* [GOF94].

10.4.1.2 Política de actualización

Las políticas de actualización determinan cuándo deben actualizarse los objetos en el sistema de almacenamiento. A lo largo del Capítulo 3 hemos visto multitud de diagramas de estados en describiendo el ciclo de vida de un objeto. Los objetos podían estar en estado "sucio", "temporal", "persistente" o "libres" indicando si el sistema de persistencia debía, o no, actualizarlos en el almacén.

La actualización de frecuente de objetos supone una caída en el rendimiento del sistema, ofreciendo, sin embargo, una mayor seguridad a los posibles fallos del sistema. Estas dos variables son están comúnmente relacionadas de un modo inverso –a mayor seguridad en cuanto a la pérdida de datos, menor rendimiento del sistema. Por tanto, la selección dinámica, adaptable o adaptativa, es un requisito importante para desarrollar nuestro sistema de persistencia § 2.2.5.

Las políticas a desarrollar deberán tener en cuenta el estado de los objetos y requisitos temporales, así como permitir crear nuevas políticas de actualización y seleccionarlas dinámicamente.

10.4.1.3 Mecanismos de indexación

Un índice es una estructura de almacenamiento físico empleada para acelerar la velocidad de acceso a los objetos persistentes. Tal y como analizamos § 3.1.5, unas técnicas de indexación proporcionan mejores rendimientos que otras para determinadas características del modelo de objetos. Así por ejemplo, los árboles B+ se comportan bien con tipos de datos básicos, pero no se puede asegurar su comportamiento para otros tipos de datos definidos por el usuario.

El mecanismo de indexación utilizado por un sistema de persistencia debería, pues, ser adaptable, permitiendo al usuario del sistema de persistencia cambiar los parámetros relativos al sistema de indexación y adaptar de este modo su comportamiento.

Sería factible capacitar al sistema de persistencia para seleccionar la técnica de indexación idónea en unas circunstancias determinadas, sin necesidad de actuación por parte del usuario. Otra utilización sería el detectar cuál es la técnica de indexación que mejor se comporta para un tipo de datos dado –véase § 13.4. Estos ejemplos demandan un mecanismo de indexación adaptativo, es decir, capaz de adaptarse de una manera transparente a las necesidades del problema (§ 2.2.3).

10.4.1.4 Sistema de almacenamiento

Se encargará de determinar qué tecnología y formato de almacenamiento será utilizado para almacenar los objetos. Permitirá configurar el sistema de persistencia para que utilice un número no predeterminado de mecanismos de almacenamiento.

Será el intermediario entre el sistema de persistencia y los almacenes físicos de datos (Figura 10.7). Ningún otro componente del sistema de persistencia accederá directamente a los sistemas de almacenamiento. Además, el sistema permite configurar diferentes sistemas de almacenamiento simultáneamente, pudiendo así obtener requisitos de replicación y exportación de datos (§ 2.2.4).

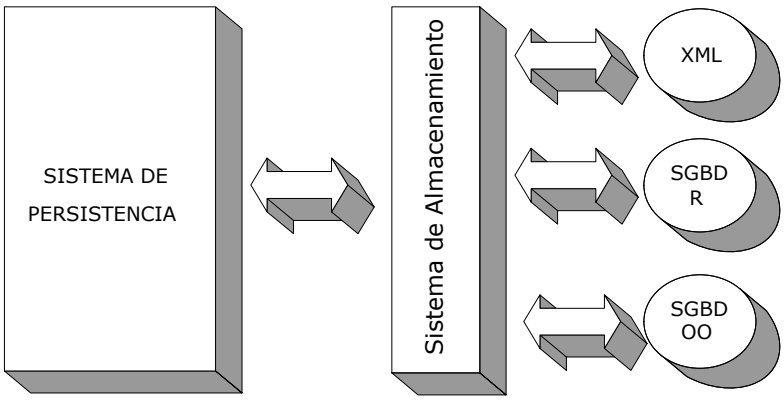


Figura 10.7. Sistemas de almacenamiento.

10.4.2 Funcionamiento del Módulo de Persistencia

El sistema de persistencia hará uso de los módulos Aplicación e Intérprete a la hora de proporcionar persistencia transparente a las aplicaciones en ejecución.

A través del módulo Aplicación (§ 10.2), el módulo de persistencia podrá trabajar directamente sobre la representación del programa en ejecución como un

modelo de objetos independiente. Gracias a que este modelo ofrece introspección, podrá analizar la estructura de los objetos para ofrecerles persistencia de una manera adaptable.

Además, haciendo uso de la reflectividad estructural, podrá modificar la estructura de éstos con el fin de prepararlos para ser utilizados por el sistema de persistencia. Por ejemplo, podría añadir a cada objeto, en tiempo de ejecución, métodos específicos del sistema de persistencia. Este mecanismo sería transparente al desarrollador, puesto que estos métodos no existen en el código fuente de la aplicación. A la hora de reconstruir los objetos a partir de su representación en el almacén también se hará uso de la reflectividad estructural: los objetos serán configurados dinámicamente a partir de su representación persistente.

Por otra parte, tal y como hemos estudiado en los capítulos 3 y 5, existen dos primitivas de persistencia básicas que no pueden hacerse transparentes al desarrollador desde una mera transformación de código: la obtención y borrado de objetos persistentes. En estos casos, el sistema de persistencia interactuará con el módulo Intérprete (§ 10.3) para modificar la semántica de las facetas computacionales que permiten hacer implícitas estas rutinas de persistencia. Gracias a las características reflectivas del sistema no restrictivo, el sistema de persistencia podrá modificar la estructura del intérprete. En concreto, podrá modificar los métodos de visita utilizados por el patrón *visitor* para realizar las acciones de interpretación del lenguaje necesarias (§ 10.3.2).

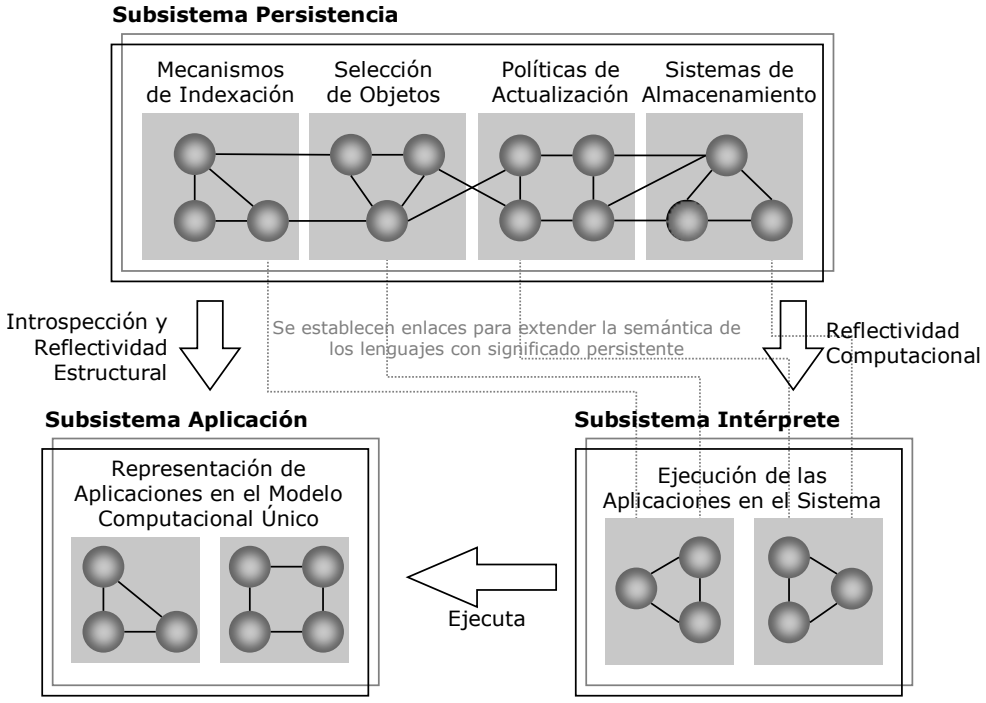


Figura 10.8. Adición de la semántica persistente a las aplicaciones en ejecución.

Finalmente debe señalarse que, gracias a que el sistema de persistencia trabaja directamente sobre el modelo de objetos independiente que representa la aplicación, el sistema de persistencia será reutilizable con cualquier programa escrito en cualquier lenguaje.

Capítulo 11
DISEÑO DEL SISTEMA REFLECTIVO NO RESTRICTIVO

En este capítulo se describirá el diseño de un prototipo que demuestre la posibilidad de construir el sistema reflectivo no restrictivo cuya arquitectura fue descrita en el Capítulo 9, a partir de las premisas o requisitos mínimos impuestos en § 9.4.

El motor computacional utilizado será un lenguaje de programación comercial que otorgue los requisitos mínimos necesarios para llevar a cabo su codificación. El objetivo de este prototipo es demostrar que la consecución de los objetivos y requisitos propios del sistema computacional reflectivo no restrictivo son factibles bajo las premisas mínimas expuestas en § 9.4.

11.1 Selección del Lenguaje de Programación

A raíz de las premisas mínimas impuestas al lenguaje de programación para el desarrollo de nuestro prototipo –expuestas en § 9.4–, hemos elegido el lenguaje de programación Python [Rossum2001]. Puesto que el lenguaje de programación Python tiene un elevado número de módulos software implementados para su reutilización por el programador, y dispone de un interfaz gráfico de programación (TK [Lundh99]), esto permitió la aceleración del desarrollo del prototipo.

A continuación enumeramos los requisitos impuestos al lenguaje, y justificaremos el cumplimiento de éstos por parte del lenguaje Python.

11.1.1 Introspección

El lenguaje de programación Python está caracterizado por su capacidad de ofrecer primitivas de introspección y reflectividad estructural, gracias a su naturaleza interpretada [Andersen98]. A continuación enunciamos un subconjunto de sus posibilidades introspectivas:

- Conocimiento dinámico del tipo de una variable. En Python no existe declaración de tipos estática; todos los tipos se infieren en tiempo de eje-

cución [Cardelli97]. La función `type` nos devuelve dinámicamente la descripción del tipo de un objeto o una variable de tipo simple.

- Conocimiento dinámico de la tabla de símbolos empleada en el contexto de ejecución. Es posible, mediante el uso de la función `dir`, el conocer la tabla de símbolos [Cueva92] utilizada por el intérprete en tiempo de ejecución; esto facilita el acceso a las variables, clases, objetos y módulos existentes dinámicamente.

- Conocimiento de los miembros de los objetos. Cada objeto (en Python las clases también se representan dinámicamente mediante objetos) posee un miembro `__dict__` que nos devuelve un diccionario de sus miembros –atributos para los objetos, métodos para las clases– con su nombre y valor [Rossum2001].

- Conocimiento del árbol de herencia. Todo objeto representativo de una clase posee un miembro `__bases__` que posee una lista de sus clases base.

11.1.2 Reflectividad Estructural

Python no sólo permite conocer dinámicamente partes de su contexto de ejecución (introspección), sino que facilita la modificación dinámica de su estructura (reflectividad estructural). Ejemplos de esta característica son:

- Creación y modificación dinámica de miembros. Python permite dinámicamente asignar atributos a cualquier objeto –métodos en el caso de que un objeto represente una clase. Si asignamos un valor a un miembro que no exista, dinámicamente se crea un atributo para el objeto implícito con el valor asignado.

- Modificación dinámica del tipo (clase) de un objeto. Todo objeto posee un atributo `__class__` que referencia a su clase. Modificar éste es posible, y el resultado producido es la alteración de su tipo.

- Modificación del árbol de herencia: herencia dinámica o delegación. Al representar el atributo `__bases__` de una clase una lista de sus superclases, la modificación de su valor implica un mecanismo de herencia dinámica o delegación, propio de lenguajes basados en prototipos como Self [Ungar87].

11.1.3 Creación, Manipulación y Evaluación Dinámica de Código

Python permite cosificar su comportamiento. Las cadenas de caracteres pueden representar código, además de datos, para poder evaluarse dinámicamente en un determinado contexto de ejecución. La función `exec` permite evaluar una cadena de caracteres, que pueda haber sido creada dinámicamente, como si de código se tratase. De forma adicional, es factible pasar como parámetros la tabla de símbolos local y global del contexto de ejecución –representada mediante un diccionario [Rossum2001].

Si lo que queremos es crear dinámicamente un método de una clase, codificamos mediante una cadena de caracteres una función y, como mostrábamos en el

primer punto de la característica anterior (reflectividad estructural), asignamos ésta a la clase adecuada. El resultado de este proceso implica la posibilidad de aumentar el número de mensajes que recibe un objeto en tiempo de ejecución, sin necesidad de parar la aplicación, y especificando la semántica de cada mensaje por el usuario de la aplicación –no el programador en fase de implementación.

11.1.4 Interacción Directa entre Aplicaciones

Como hemos comentado previamente, dentro de un mismo contexto de ejecución Python permite conocer y modificar la estructura de una aplicación dinámicamente. Sin embargo, la interacción entre aplicaciones se debe llevar a cabo ejecutándolas en instancias distintas del intérprete y comunicándolas mediante un middleware. Sobre esta capa intermedia, habría que implementar un módulo que facilitase la intercomunicación entre aplicaciones, con la consecuente complicación.

Una de las restricciones impuestas al motor computacional en 9.4 era ofrecer un entorno en el que las aplicaciones pudiesen interactuar entre sí, sin necesidad de una capa intermedia, y de forma independiente al lenguaje en el que éstas hubiesen sido desarrolladas.

La interacción entre aplicaciones Python es compleja al ejecutarse cada una en un proceso del sistema operativo distinto. Sin embargo, los distintos hilos (*threads*) que una aplicación Python cree pueden acceder a las variables globales del hilo padre que los creó. De esta forma, la solución al problema surgido pasó por asignar un hilo distinto a cada aplicación de nuestro sistema, ejecutándose éste como una única aplicación Python.

Como se muestra en la Figura 11.1, el hilo principal crea un objeto denominado nitrO que sigue el patrón de diseño Fachada (*Facade*) [GOF94]: todo el acceso al sistema se lleva a cabo a través de este objeto. Una vez que dentro del sistema se ejecute una nueva aplicación, se creará un hilo hijo del principal, pudiendo éste acceder al objeto nitrO propio del hilo padre. Al dar este objeto acceso a toda la funcionalidad del sistema, obtenemos una interacción directa entre aplicaciones de nuestro sistema, independientemente del lenguaje utilizado[27].

[27] La propiedad `apps` del objeto `nitrO` nos devuelve un diccionario con todas las aplicaciones activas dentro del sistema; el acceso a éstas supone, por tanto, la interacción entre aplicaciones.

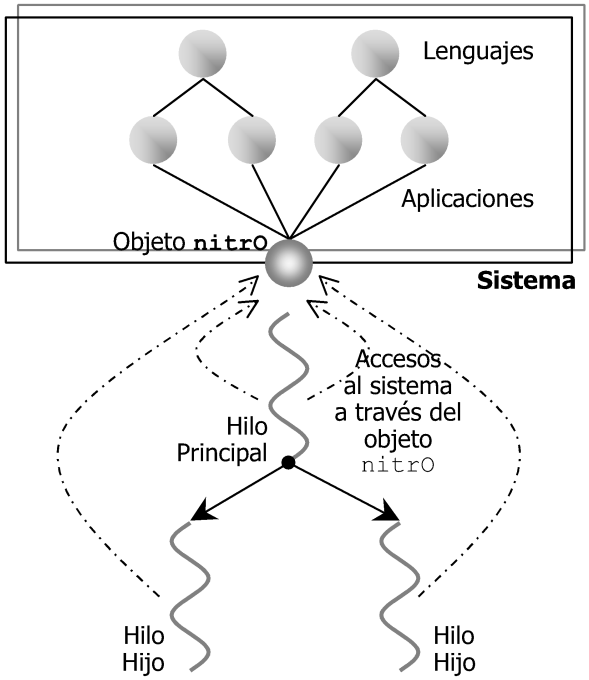


Figura 11.1. Acceso al objeto nitrO desde distintos hilos representativos de aplicaciones.

11.2 Diagrama de Subsistemas

El diseño del prototipo ha sido dividido en un conjunto de subsistemas, implementados como módulos Python [Ortín2003]. Éstos y sus dependencias se muestran en la siguiente vista estática, expresada mediante UML [Rumbaugh98]:

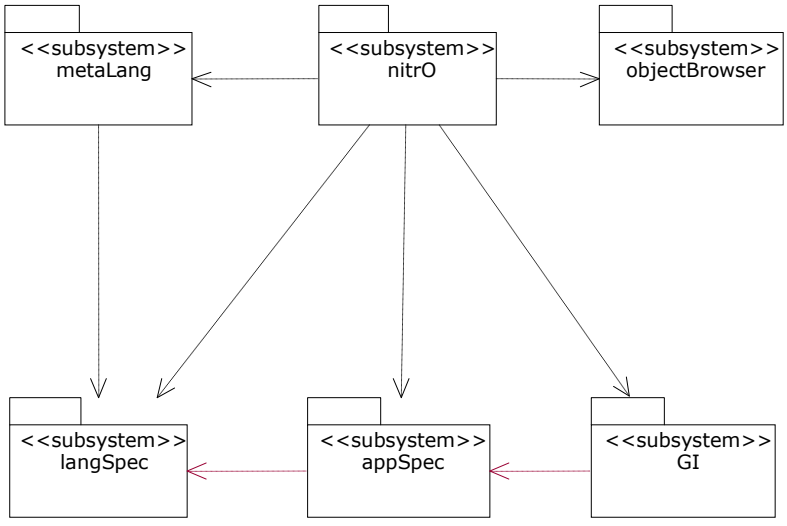


Figura 11.2. Diagrama de subsistemas del prototipo.

11.2.1 Subsistema nitrO

Como comentábamos en el punto anterior, la interacción directa entre las aplicaciones de nuestro sistema se consigue mediante la utilización del patrón de diseño Fachada [GOF94], en el que se establece un único punto de entrada al mismo.

De esta forma, el subsistema `nitrO` establece este punto central que permite acceder al resto de servicios ofrecidos en el sistema.

11.2.2 Subsistema objectBrowser

El subsistema `objectBrowser` hace un uso directo de las características introspectivas propias del lenguaje Python comentadas en el primer punto de este capítulo. A través del objeto `nitrO` existente dentro del subsistema con el mismo nombre, es ofrecida una colección de objetos y clases necesarias para obtener el entorno flexible de programación propuesto en esta Tesis. Este subsistema muestra al usuario esta información de forma dinámica.

Puesto que la cantidad de aplicaciones en ejecución puede ser elevada, el número de objetos y clases que representen a éstas, y a sus respectivas especificaciones de lenguajes de programación, representarán una información ingente para el usuario. Gracias a la introspección ofrecida por Python, este subsistema analiza dinámicamente las propiedades del objeto Fachada `nitrO`. El usuario podrá conocer toda la información ofrecida dinámicamente por el sistema, navegando por un árbol representativo de los objetos existentes, para posteriormente llevar a cabo la acción adecuada.

11.2.3 Subsistema langSpec

Puesto que nuestra herramienta es independiente del lenguaje de programación que el usuario quiera utilizar, este subsistema es el encargado de representar cada lenguaje utilizado por una aplicación.

Como expondremos de forma más detallada en el siguiente punto, es necesaria la representación del lenguaje de programación mediante una estructura de objetos. El conjunto de clases ofrecidas en este módulo, tiene por objetivo la representación de gramáticas libres de contexto [Cueva91] y su semántica asociada.

11.2.4 Subsistema metaLang

Dado que para que una aplicación sea ejecutada por nuestro entorno computacional flexible es necesario conocer el lenguaje en el que ésta haya sido programada, deberemos idear un modo de representar la totalidad de lenguajes de programación a emplear. El modo en el que permitimos expresar al usuario los distintos lenguajes de programación a utilizar –cualquiera libre de contexto [Cueva91]– es mediante otro lenguaje: un metalenguaje.

El propósito del conjunto de clases existentes en este subsistema es procesar la especificación del lenguaje de programación a utilizar –expresado en el metalenguaje– y, si fuere correcta, convertir su especificación en una representación de objetos, haciendo uso para ello de las clases del subsistema `langSpec` –de ahí la dependencia entre ambos.

11.2.5 Subsistema appSpec

Una vez que el lenguaje haya sido especificado por el usuario, reconocido por el subsistema `metaLang`, y convertido a su representación mediante objetos, la misión de éstos es tomar una aplicación codificada para este lenguaje y obtener su

árbol sintáctico [Ortín2004b] (AST, *Abstract Syntax Tree* [Aho90]). El conjunto de clases existentes en este subsistema permite representar el árbol sintáctico de una aplicación, con sus correspondientes enlaces a la representación de su lenguaje, para su posterior ejecución por el subsistema GI.

11.2.6 Subsistema GI

Tomando el árbol sintáctico creado por el subsistema `appSpec` este subsistema (GI, Generic Interpreter) es el encargado de ejecutar las acciones semánticas propias del lenguaje de programación asociado, es decir, de interpretar la aplicación.

11.3 Subsistema metaLang

El reconocimiento de un lenguaje de programación y la conversión de su especificación a una estructura de objetos, es el principal objetivo de este subsistema. La descripción del lenguaje deberá llevarse a cabo mediante un archivo con extensión `ml` o incluyendo ésta previamente a la codificación de una aplicación.

11.3.1 Diagrama de Clases

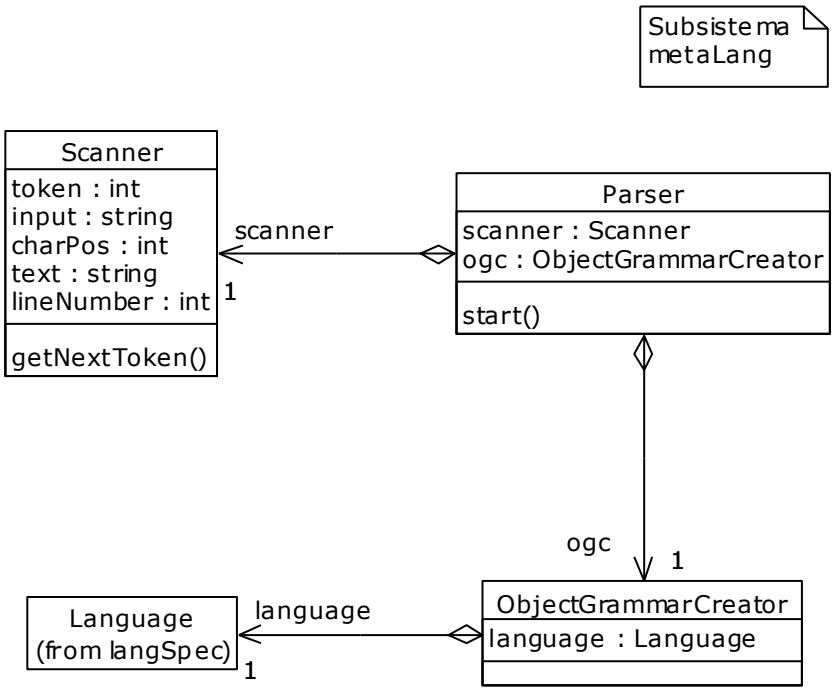


Figura 11.3. Diagrama de clases del subsistema metaLang.

El conjunto de clases sigue la estructura clásica de un procesador de lenguaje para llevar a cabo el análisis léxico y sintáctico del metalenguaje: la funcionalidad del módulo de análisis léxico es llevada a cabo por cada uno de los objetos instancia de la clase `Scanner`, mientras que el sintáctico es responsabilidad de los objetos `Parser`.

El analizador léxico [Cueva93] obtiene su código de entrada de su atributo `input` y analiza éste avanzando el contador de posición `charPos`. Para leer un nuevo componente léxico, invocamos el método `getNextToken`; el último token

leído puede ser obtenido de su propiedad token. La codificación de sus algoritmos ha sido llevada a cabo siguiendo las pautas propuestas por Holub [Holub90].

El analizador sintáctico se ha desarrollado mediante la técnica de traducción de una gramática descendente LL1 a subprogramas [Cueva95], obteniendo un analizador sintáctico descendente recursivo sin retroceso [Aho90]. Cada símbolo no terminal de la gramática constituye un método que desarrolla su análisis, siendo el método `start` el encargado de desencadenar el proceso asignado al símbolo inicial de la gramática [Cueva91] –el análisis de toda la aplicación.

El analizador sintáctico demanda componentes léxicos de su propiedad scanner y, conforme analiza el archivo fuente, va creando la representación mediante objetos de dicho lenguaje; su propiedad `ogc`, instancia de la clase `ObjectGrammarCreator`, posee métodos para facilitar dicha creación.

11.4 Subsistema langSpec

Representa mediante asociaciones entre objetos la especificación de un lenguaje de programación. El subsistema `metaLang` crea estas representaciones apoyándose en instancias de su clase `ObjectGrammarCreator`. Las aplicaciones accederán dinámicamente a esta representación de su lenguaje de programación para interpretar su semántica de evaluación.

Podremos cuestionarnos por qué es necesaria la representación de un lenguaje mediante estructuras de objetos. En el Capítulo 2 enunciábamos un conjunto de requisitos relativos a la adaptabilidad de nuestro sistema. Todos ellos se obtienen, como hemos indicado en el Capítulo 9, al permitir a una aplicación modificar dinámicamente la especificación de su lenguaje de programación. El hecho de que una aplicación acceda y modifique dinámicamente la estructura de los objetos que representan su lenguaje de programación es factible mediante el uso de reflectividad estructural. Es por ello por lo que es necesario que el lenguaje de programación seleccionado ofrezca esta característica.

11.4.1 Diagrama de Clases

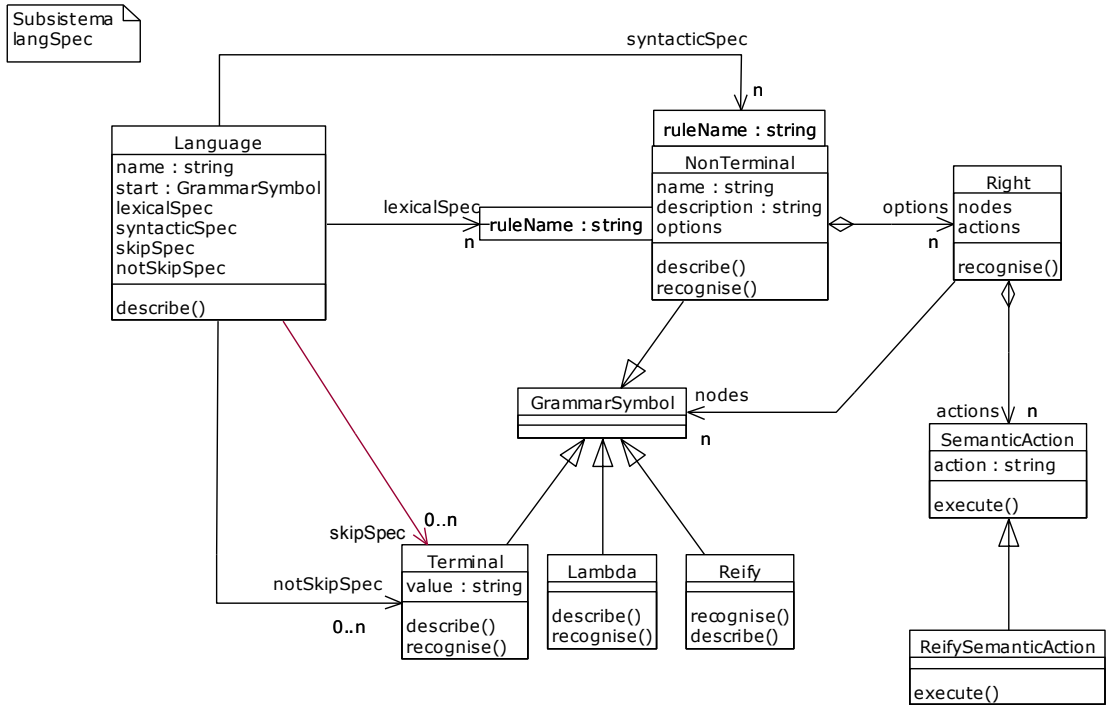


Figura 11.4. Diagrama de clases del subsistema langSpec.

Siguiendo con la descripción de lenguajes de tipo 2, los elementos de una gramática pueden ser terminales (clase `Terminal`), no terminales (`NonTerminal`) y el elemento vacío o lambda. Todos ellos son símbolos gramaticales, derivando pues de la clase `GrammaSymbol`[28]. El método `recognise` de todo símbolo gramatical está orientado a intentar reconocer su patrón ante una aplicación de entrada y, en el caso de que esta acción resulte satisfactoria, creará su árbol sintáctico –mediante las clases del subsistema `appSpec`. En la descripción del subsistema `appSpec` detallaremos el modo en el que este árbol es creado.

El método `describe` de todos los objetos, en cualquier subsistema, tiene la misma utilidad: ofrecer información descriptiva del objeto implícito.

Toda regla de una gramática libre de contexto se puede expresar como [Cueva91]:

$$A \rightarrow \alpha, A \in VN, \alpha \in (VN \cup VT)*.$$

La parte de la izquierda de una regla es siempre un no terminal, mientras que la parte derecha es una consecución de símbolos gramaticales. La clase `Right` representa cada una de las partes derechas asociadas a un símbolo no terminal (atributo `options`) mediante una regla. La colección de los símbolos gramaticales de la parte derecha de una regla se almacena en el atributo `nodes` de los objetos de tipo `Right`.

Las reglas semánticas asociadas a las distintas alternativas de una producción, que se ejecutan cuando ésta es reconocida, son instancias de la clase

[28] Esta clase no ha sido codificada realmente en nuestro prototipo por peculiaridades del lenguaje de programación utilizado. Python posee inferencia dinámica de tipos y por tanto sólo verifica si un objeto puede interpretar un mensaje cuando éste lo recibe -en tiempo de ejecución. De esta forma, la utilización de la herencia como la derogación de comportamiento polimórfico, no es necesaria en el lenguaje Python. Puesto que no hay validación estática de tipos, no tiene utilidad implementar una clase base con métodos a sobrescribir por las abstracciones derivadas.

`SemanticAction` y se accede a ellas a través del atributo `actions` de cada parte derecha. La semántica se representa mediante código Python, almacenándose en la propiedad `action` y evaluándose mediante el paso de mensaje `execute`.

La ejecución de una regla semántica no se limita a la evaluación de un código Python. Éste ha de evaluarse dentro de un contexto para cada aplicación: cada aplicación de nuestro sistema tendrá su propio espacio de direcciones con sus variables, objetos, clases y funciones, es decir, su contexto. Este contexto se almacena en la propiedad `applicationGlobalContext` de cada instancia de la clase `Application` del subsistema `appSpec`.

`Language` es la clase principal, cuyas instancias representarán cada uno de los lenguajes cargados en el sistema. Sus atributos `name` y `start` identifican respectivamente su identificador único y el símbolo no terminal inicial de la gramática. Los atributos `lexicalSpec` y `syntacticSpec` identifican dos diccionarios de símbolos no terminales ubicados en la parte izquierda de una regla; la clave de los diccionarios es el identificador único de los no terminales. Las reglas `lexicalSpec` representan componentes léxicos, mientras que `syntacticSpec` define la sintaxis del lenguaje.

Las propiedades `skipSpec` y `notSkipSpec` son una colección de terminales que especifican aquellos elementos que deben eliminarse automáticamente, o bien concatenarse a los tokens del lenguaje de forma implícita.

Como hemos explicado en § 9.2.1, nuestro entorno computacional de programación flexible permite realizar un salto computacional del espacio de aplicación al espacio de interpretación. Esta operación se lleva a cabo mediante la instrucción `reify`, que permite especificar al programador un código a ejecutar en el contexto de interpretación, indiferentemente del lenguaje utilizado. El reconocimiento de esta instrucción existente en todo lenguaje la realiza las instancias de la clase `Reify`.

Puesto que la ejecución de un código de programación se evalúa de modo distinto al código del nivel inferior (interpretación), ésta segunda semántica es la que deroga la redefinición del método `execute` de la clase `ReifySemanticAction`.

11.4.2 Especificación de Lenguajes mediante Objetos

La flexibilidad de nuestro sistema computacional viene otorgada por la posibilidad de ejecutar por el programador código de un nivel computacional menor. Este código es evaluado en el contexto del intérprete y podrá utilizarse para acceder y modificar el lenguaje de programación utilizados, consiguiendo así el grado de reflectividad deseado.

Puesto que la modificación de determinadas características del lenguaje de programación utilizado en una aplicación va a ser una tarea común, el programador de aplicaciones deberá conocer el estado dinámico de los objetos que constituyen la especificación de un lenguaje de programación. Para facilitar su comprensión, mostraremos un ejemplo de parte de los objetos que representan el siguiente lenguaje:

```
Language = Print
Scanner = {
"Digit Token"
digit -> "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9" ;
"Number Token"
NUMBER -> digit moreDigits ;
```

```
"Zero or more digits token"
moreDigits -> digit moreDigits
| ;
"PRINT Key Word"
PRINT -> "PRINT" ;
"SEMICOLON Token"
SEMICOLON -> ";" ;
}
Parser = {
"Initial Free-Context Rule"
S -> statement moreStatements SEMICOLON <#
nodes[1].execute()
nodes[2].execute()
#> ;
"Zero or more Statements"
moreStatements -> SEMICOLON statement moreStatements <#
nodes[2].execute()
nodes[3].execute()
#>
| ;
"Statement Rule"
statement -> _REIFY_ <#
nodes[1].execute()
#>
| printStatement <#
nodes[1].execute()
#> ;
"Print Statement"
printStatement -> PRINT NUMBER <#
write( "Integer constant: "+nodes[2].text+".\n")
#> ;
}
Skip={ "\t"; "\n"; }
NotSkip = { " "; }
```

El subsistema `metaLang` toma esta especificación y, haciendo uso de la clase `ObjectGrammarCreator`, crea un conjunto de objetos que especifiquen este lenguaje "Print". Un subconjunto de éstos es:

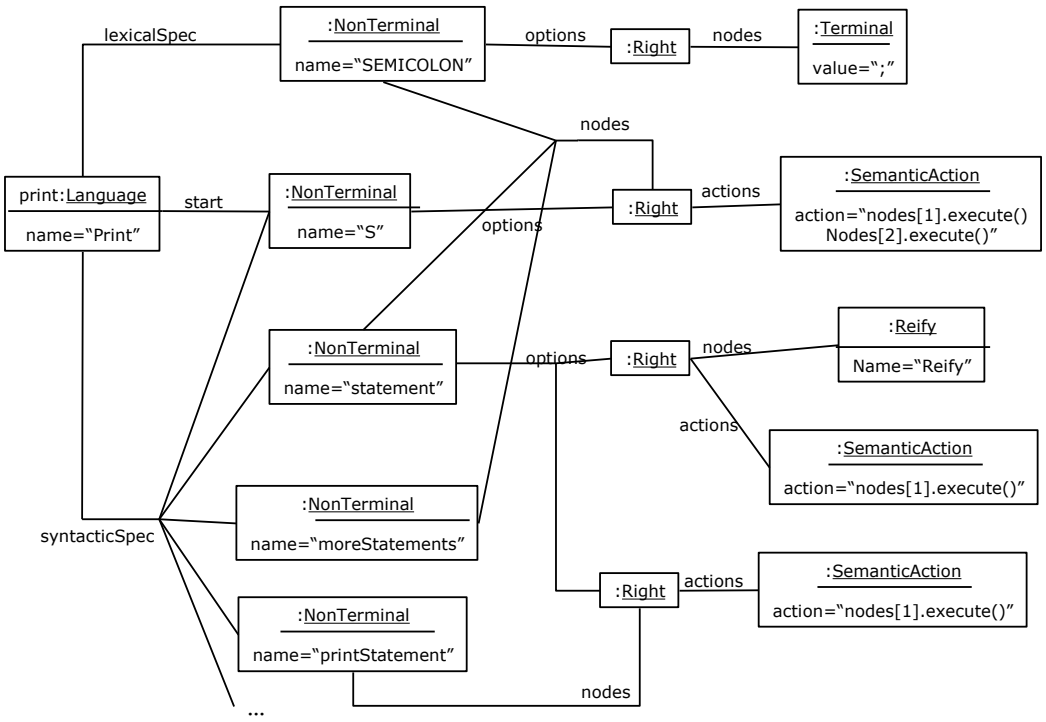


Figura 11.5. Diagrama de objetos de la especificación del lenguaje "Print".

Como se observa en la Figura 11.5, los objetos de tipo `Language` ofrecen la toda especificación cada lenguaje de programación utilizado. Éste posee una referencia `start` al símbolo no terminal inicial de la gramática y un par de diccionarios, `lexicalSpec` y `syntacticSpec`, para representar las reglas léxicas y sintácticas respectivamente.

Cada símbolo no terminal tiene un conjunto de objetos `Right`, identificados mediante su propiedad `options`, que representan la parte derecha de cada producción asociada. Las partes derechas de las producciones poseen una colección de símbolos gramaticales –atributo `nodes`– y un conjunto de acciones semánticas –atributo `actions`.

El resultado es un grafo de objetos que representan la especificación del lenguaje mediante su estructura. Utilizando reflectividad estructural y la cosificación ofrecida por la instrucción `reify`, el programador puede modificar la especificación de un lenguaje haciendo variar así todas las aplicaciones codificadas sobre éste.

11.5 Subsistema appSpec

Una vez que el sistema ha creado la representación en objetos del lenguaje de programación a utilizar, la funcionalidad de éstos es llevar a cabo el análisis sintáctico del código fuente y, en el caso de que la aplicación sea reconocida por la gramática, obtener el árbol sintáctico representante de ésta. Las clases cuyos objetos representarán el árbol sintáctico se muestran en el siguiente diagrama de clases:

11.5.1 Diagrama de clases

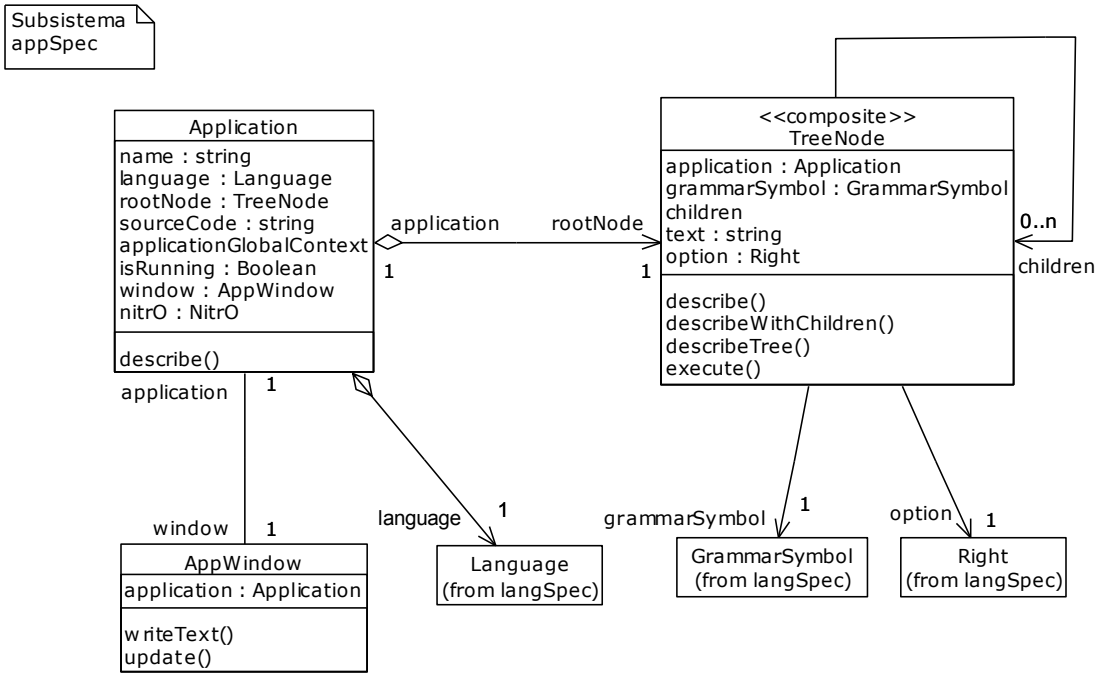


Figura 11.6. Diagrama de clases del subsistema appSpec.

Una aplicación en nuestro sistema viene identificada por una instancia de la clase `Application`. Sus atributos representan su identificación (`name`), lenguaje asociado (`language`), código fuente (`sourceCode`), contexto de ejecución o tabla

de símbolos (`applicationGlobalContext`), estado de ejecución (`isRunning`) y su vista mediante una ventana gráfica (`window`), entre otros.

La relación entre la clase `Application` y `Language` es de composición, es decir, cada vez que se ejecuta una aplicación en un determinado lenguaje de programación, aunque éste sea el mismo, se crea una nueva instancia del lenguaje. Como comentábamos en el Capítulo 6, la reflectividad computacional pura debe permitir modificar todas las aplicaciones del sistema que posean una misma semántica computacional (lenguaje de programación). Puesto que nuestro prototipo ha sido diseñado para justificar y evaluar un trabajo de investigación, hemos seguido el criterio de utilizar un grano más fino: cada aplicación está asociada a una única representación de un lenguaje[29]. De esta forma podremos modificar la semántica de una aplicación desde otra aplicación codificada en el mismo lenguaje, sin que varíe el comportamiento de la segunda aplicación.

Cada aplicación posee una visualización gráfica de su ejecución mediante una ventana –objeto de la clase `AppWindow`. El método `writeText` muestra en ella una línea de texto y su contenido puede actualizarse con el paso del mensaje `update`.

Los nodos del árbol sintáctico son instancias de la clase `TreeNode`, diseñada mediante el patrón de diseño Composición [GOF94]: cada nodo almacena una lista de sus nodos descendientes, de forma recursiva, mediante su atributo `children`. El texto reconocido por la gramática para el símbolo gramatical asociado ante una aplicación de entrada, es almacenado en su atributo `text`.

Los nodos del árbol semántico guardan la asociación con su símbolo gramatical del lenguaje asociado (`grammarSymbol`) y, en el caso de que éste no sea un terminal, la parte derecha de la producción (`option`) elegida para obtener sus hijos.

El modo en el que se evalúa el árbol dando lugar a la ejecución de la aplicación se lleva a cabo mediante envíos de mensajes `execute`. Analizaremos esta funcionalidad en el estudio del subsistema GI.

11.5.2 Creación del Árbol Sintáctico

Una vez creada la especificación del lenguaje de programación e identificado un programa a procesar, la invocación al método `recognise` del símbolo inicial de la gramática, deberá desencadenar el siguiente proceso:

- Validar si la aplicación pertenece al lenguaje.
- Obtener el árbol sintáctico de la aplicación de entrada.
- Asignar el texto reconocido a la propiedad `text` de cada nodo del árbol sintáctico creado.

El algoritmo utilizado es el de reconocimiento descendente con retroceso o *backtracking* [Cueva95], poco eficiente pero sencillo de implementar. La invocación del método afirmará que la sentencia pertenece al lenguaje si encuentra una parte derecha de su producción cuyos símbolos gramaticales validen la pertenencia al len-

[29] La modificación de todas las instancias de un lenguaje es posible y por tanto podrá modificarse la semántica de las aplicaciones que utilicen un mismo lenguaje de programación. Sin embargo, aunque no se pierda esta posibilidad, el tiempo de computación requerido para este proceso es mayor que el necesario en el caso de haber seguido el otro criterio de diseño.

guaje, invocando a su vez a sus métodos `recognise`. Este algoritmo es de naturaleza recursiva y polimórfico respecto al comportamiento del mensaje `recognise`.

Si la ejecución del método `recognise` es satisfactoria, se indicará la pertenencia al lenguaje con la devolución del árbol sintáctico asociado a esa producción. De forma paralela, se irá asignando a cada nodo la fracción de código reconocida por él, en su atributo `text`.

Supongamos que se va a analizar, para el lenguaje Print presentado en § 11.4.2, el siguiente programa:

```
PRINT 45;
```

Cuando se le envíe el mensaje `recognise` al símbolo gramatical "statement", un subconjunto de las operaciones desencadenadas entre los objetos pertinentes, es el mostrado en el siguiente diagrama de colaboración:

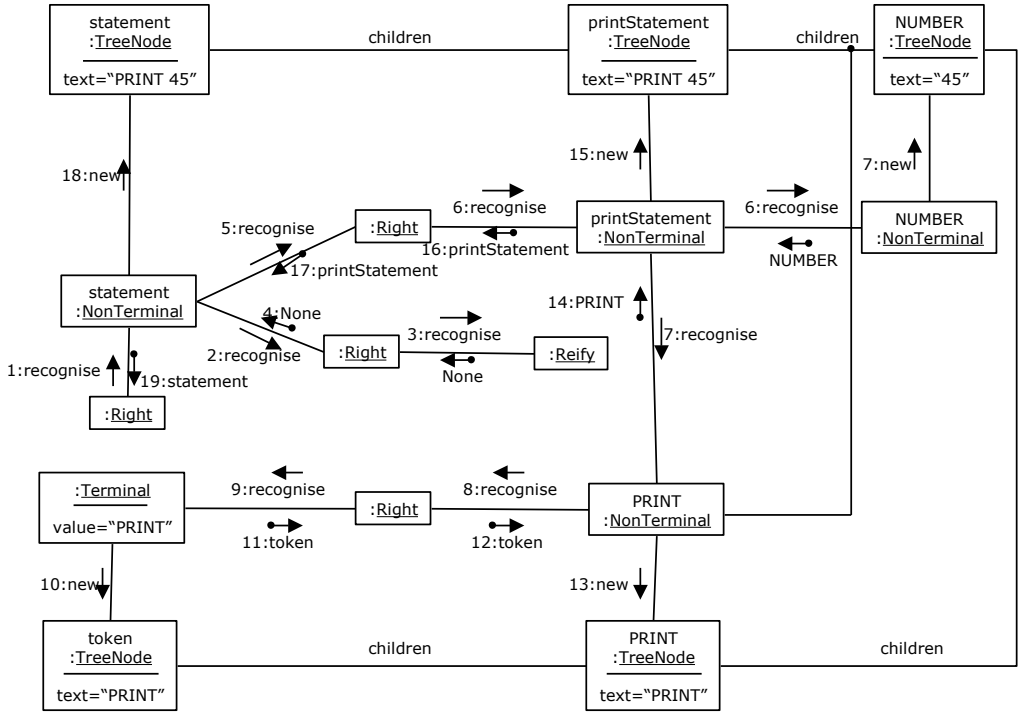


Figura 11.7. Creación del subárbol sintáctico propio de una sentencia.

Como se muestra en la figura, la invocación del método `recognise` de un no terminal implica la búsqueda de una parte derecha que valide la producción. Si una es encontrada, el valor devuelto es el subárbol creado. En el ejemplo vemos cómo una parte derecha "statement" no consigue validar la producción –al no aparecer en el código fuente una instrucción `reify`. A la parte derecha restante es enviado entonces el mensaje, resultando positivo y obteniendo el subárbol sintáctico apropiado.

11.6 Subsistema GI

Los objetivos principales de este subsistema son:

- Invocación del algoritmo de reconocimiento sintáctico con retroceso, aportan do un buffer en el que se encuentra el código fuente a analizar.

- Ejecutar la aplicación, recorriendo el árbol sintáctico y evaluando los nodos como se haya descrito en la especificación del lenguaje.
- Eliminar de la entrada los componentes léxicos descritos en la sección `skipSpec`.
- Concatenar automáticamente los tokens especificados como `notSkipSpec` en el metalenguaje.

11.6.1 Diagrama de Clases

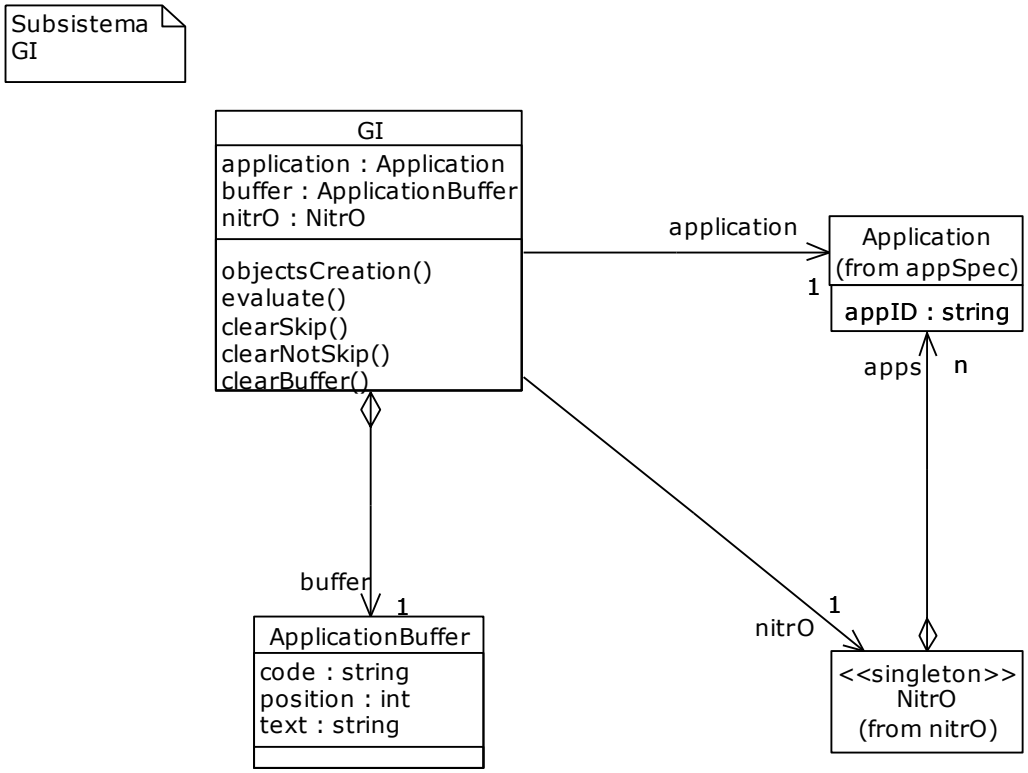


Figura 11.8. Diagrama de clases del subsistema GI.

Las instancias de la clase `ApplicationBuffer` llevan el control de la posición en el que el código fuente está siendo analizado –`code` almacena el código y `position` la posición–, así como la traducción de éste –atributo `text`. El archivo fuente va siendo procesado eliminando automáticamente los tokens de `skipSpec` y concatenando, sin necesidad de explicitarlo en la gramática, los indicados en `notSkipSpec`; el resultado es almacenado en text.

Como vimos en el algoritmo expuesto en § 11.5.2, cada nodo del árbol sintáctico posee el texto reconocido excluyendo e incluyendo automáticamente los componentes léxicos especificados en `skipSpec` y `notSkipSpec`, respectivamente. Este proceso se apoya directamente en el atributo `text` de las instancias de `applicationBuffer`: antes de invocar al método `recognise` de sus hijos, almacena la posición del buffer y, una vez reconocidos éstos, obtiene el incremento sufrido en el atributo `text`.

Los objetos GI llevan a cabo la traducción de código mediante los métodos `clearSkip` y `clearNotSkip`. El método `clearBuffer` invoca a estos dos sucesivamente hasta que no quede ningún token por limpiar en la posición actual del código fuente. En el ejemplo del lenguaje "Print", los caracteres tabulador y salto de

línea son eliminados automáticamente, mientras que los espacios en blanco se concatenan a los componentes léxicos reconocidos.

Cada instancia de GI posee una referencia al buffer utilizado (`buffer`), a la aplicación a ejecutar (`application`) y al sistema (`nitrO`). El método `objectsCreation` es el encargado de procesar la aplicación de entrada y obtener el árbol sintáctico tal y como se explicó en § 11.5.2.

La evaluación del árbol sintáctico es llevada a cabo mediante el método evaluate de las instancias de GI. Mostraremos a continuación cómo se coordinan los nodos del árbol y las reglas semánticas para producir la ejecución de una aplicación.

11.6.2 Evaluación del Árbol Sintáctico

Una invocación al método `evaluate` de una instancia de GI, hace que ésta localice el nodo inicial –atributo `rootNode` de `application`– y le pase el mensaje `execute`. En la aplicación y lenguaje de entrada de ejemplo mostrado en este capítulo, la ejecución del método asociado produce la evaluación del árbol siguiendo el siguiente diagrama de colaboración[30]:

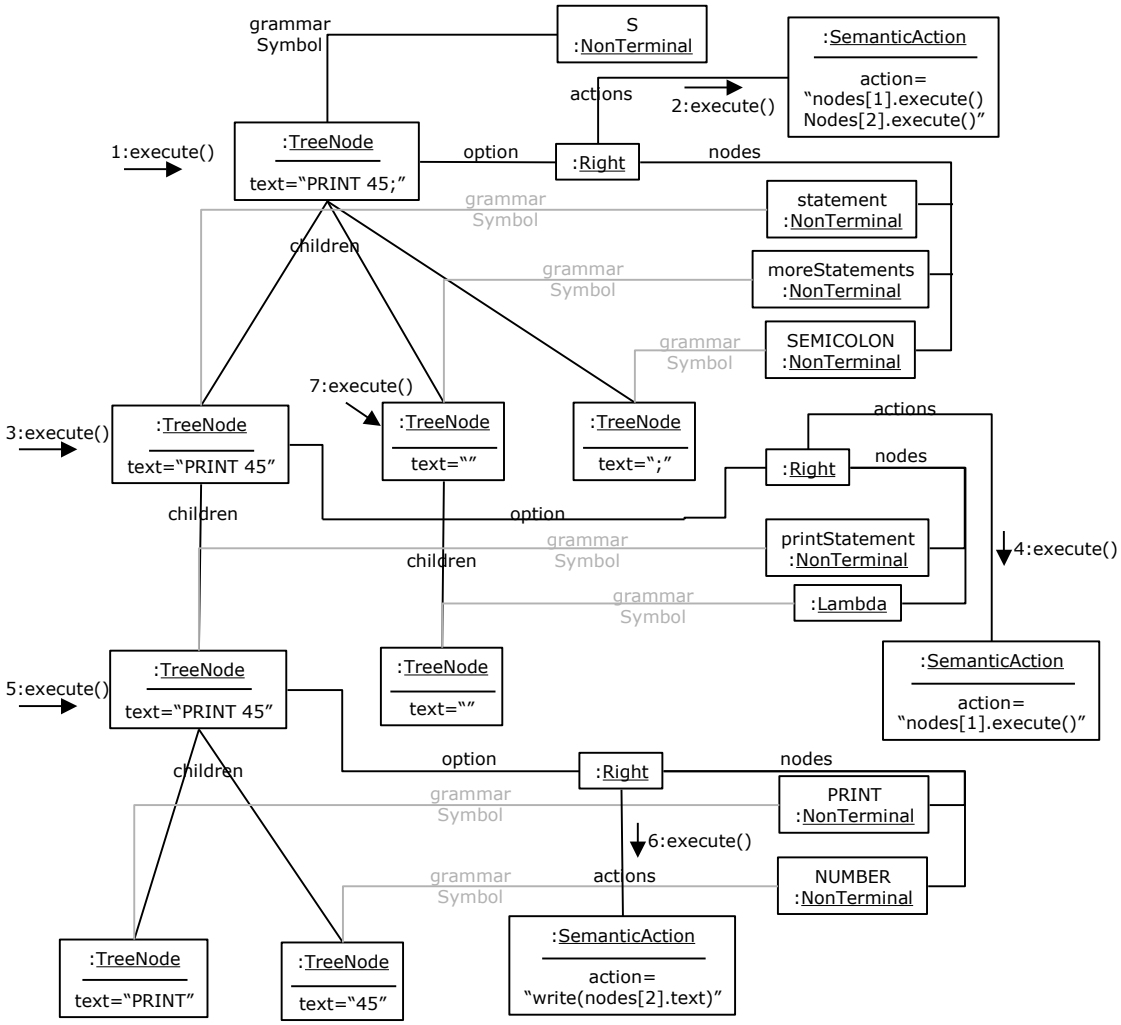


Figura 11.9. Evaluación de un árbol sintáctico.

[30] Los objetos descendientes de los no terminales que forman parte de la especificación léxica han sido omitidos para simplificar la figura. Su evaluación sigue los mismos pasos que los descritos para el resto de nodos.

Este diagrama muestra cómo los objetos del subsistema `langSpec` –más a la derecha en la figura– colaboran con los de `appSpec` para evaluar el árbol. Una vez que se le pasa el mensaje `execute` al nodo inicial, éste obtiene las acciones de su parte derecha (`option`) y les pasa el mismo mensaje. Ejecutadas las acciones asociadas, el proceso concluye.

Para nuestro ejemplo, la acción semántica del símbolo inicial conlleva la ejecución de las acciones semánticas de los nodos "statement" y "moreStatements", puesto que así se especificó en la descripción del lenguaje. La ejecución del nodo "statement" desencadena la evaluación del nodo "printStatement" y éste visualiza en su ventana la constante numérica 45, mediante la sentencia "`write(nodes[2].text)`".

11.7 Subsistema nitrO

El subsistema nitrO define el objeto `nitrO` que sigue el patrón de diseño Fachada [GOF94], ofreciendo todas las posibilidades del sistema.

11.7.1 Diagrama de Clases

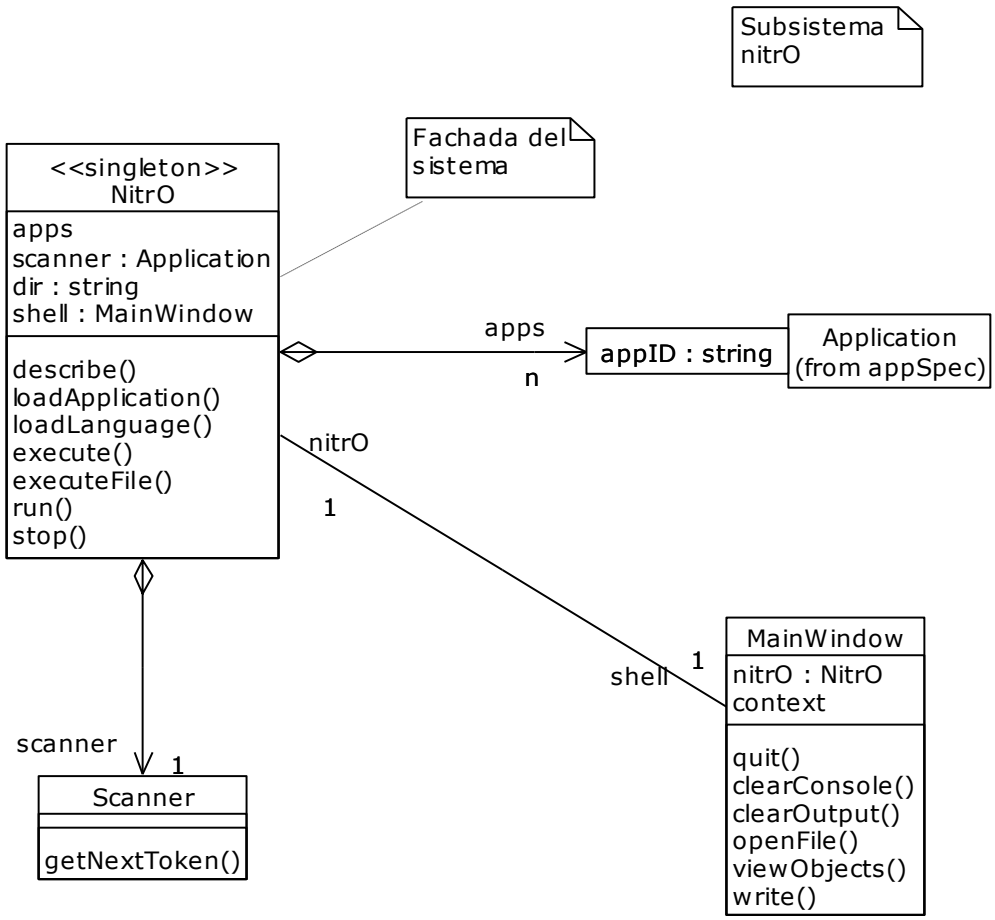


Figura 11.10. Diagrama de clases del subsistema nitrO.

La clase `NitrO` sigue el patrón de diseño Singleton [GOF94], prohibiendo la creación de más de una instancia de dicha clase. El único objeto creado (`nitrO`) ofrece el siguiente conjunto de servicios:

- Aplicaciones del sistema y sus respectivos lenguajes, accediendo a su atributo `apps`.
- Un pequeño intérprete de comandos (`shell`) que ofrece la evaluación de código en el sistema, una consola de salida (utilizando la función `write`) y la carga y edición de archivos.
- El atributo `dir` posee la ubicación del sistema dentro del árbol de directorios.
- Permite ejecutar cualquier sentencia de código dentro del sistema –utilizando el método `execute`– así como la ejecución de archivos que tengan un conjunto de sentencias –método `executeFile`.
- Si se desea finalizar la ejecución de una aplicación dentro del sistema, podemos enviarle el mensaje stop, pasando como parámetro el identificador de la aplicación.

Las instancias de la clase `Scanner` son las encargadas de obtener los componentes léxicos de los archivos de aplicación. Cada vez que se desee obtener un nuevo token, se le enviará el mensaje `getNextToken`.

Finalmente, la clase `MainWindow` supone una ventana gráfica con un menú, editor y consola de salida, para ofrecer un pequeño intérprete de comandos. El código Python que se escriba será evaluado dentro del sistema. Podremos utilizar el objeto `nitrO` para acceder al sistema, y la función `write` para visualizar información en la consola de salida.

Capítulo 12

DISEÑO DE UN PROTOTIPO DEL SISTEMA DE PERSISTENCIA

En este capítulo se presenta el diseño de un prototipo que demuestra la viabilidad de la arquitectura del sistema de persistencia presentada en el Capítulo 10. El prototipo del sistema de persistencia se desarrollará sobre el prototipo del sistema reflectivo no restrictivo cuyo diseño se ha presentado en el capítulo anterior. El objetivo de este prototipo es demostrar la satisfacción de los requisitos enunciados en esta Tesis Doctoral.

En primer lugar se ofrecerá una visión general de los diferentes subsistemas que componen el prototipo del sistema de persistencia. A continuación, se analizará con detalle el diseño de cada uno de estos subsistemas, los componentes que abarcan y cual es su funcionamiento interno.

12.1 Diagrama de Subsistemas

El diseño del prototipo ha sido dividido en un conjunto de subsistemas, implementados como módulos Python. En la el diagrama UML de la se muestran estos módulos y sus dependencias.

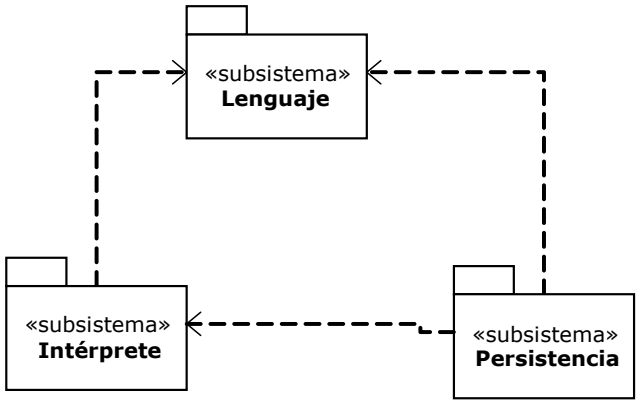


Figura 12.1. Diagrama de subsistemas del prototipo.

12.1.1 Subsistema Lenguaje

En el subsistema lenguaje se definen las clases que permitirán representar cualquier programa utilizando un modelo orientado a objetos independiente del lenguaje de programación en el que fue desarrollado.

12.1.2 Subsistema Intérprete

El subsistema intérprete se encarga de ejecutar el modelo de objetos independiente del lenguaje. Presenta un *Singleton* [GOF94] `Interpreter` que permite ejecutar dicho modelo.

Para construir el modelo de la aplicación a ejecutar se utilizará el intérprete genérico del sistema reflectivo no restrictivo cuyo diseño se abordó en el Capítulo 11. Este modelo se definirá en base al modelo de objetos presentado por el subsistema Lenguaje.

12.1.3 Subsistema Persistencia

El subsistema de persistencia ofrece los servicios de persistencia a la aplicación a través de un objeto Fachada (*Facade*) [GOF94]. Ofrece un conjunto de componentes de persistencia con los cuales puede personalizarse su comportamiento.

12.2 Subsistema Lenguaje

12.2.1 Diagrama de Clases

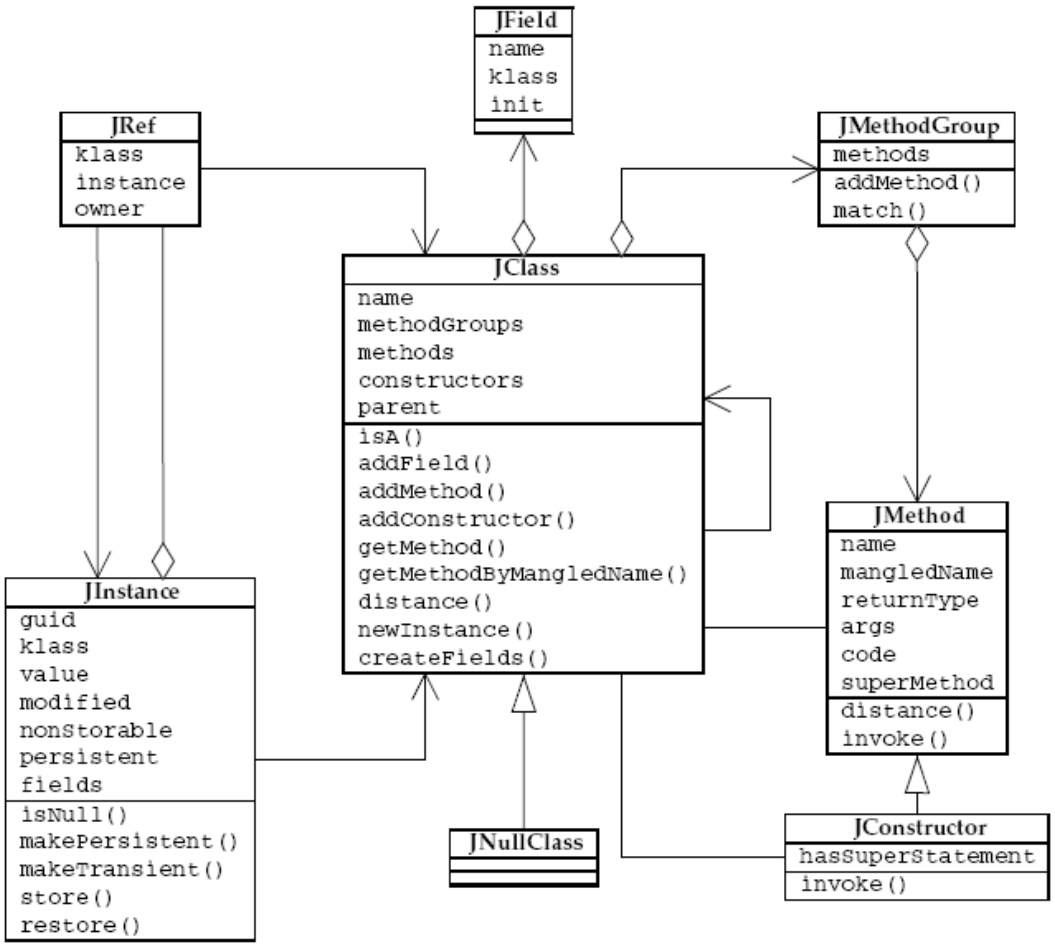


Figura 12.2. Diagrama de clases del modelo de objetos independiente del programa.

La Figura 12.2 muestra la estructura de clases de los elementos que componen el modelo de objetos independiente del lenguaje que representará las aplicaciones en ejecución. Las clases poseen métodos (`JMethod`) y constructores (`JConstructor`), ambos agrupados con la ayuda de instancias de `JMethodGroup`, pero manteniendo accesos directos para los casos en que se sabe exactamente qué método o constructor debe invocarse y no se necesita buscar el más apto dentro del grupo. Las clases también contienen campos, que servirán como modelo para la creación de los atributos al inicializar una nueva instancia.

Las instancias de objetos vienen representadas por la clase `JInstance`. En el diagrama se muestran los métodos que ofrece esta clase al sistema de persistencia. Debe señalarse que estos métodos no tienen porqué existir en tiempo de diseño. El sistema de persistencia podría añadirlos dinámicamente haciendo uso de la reflectividad estructural que ofrece el sistema reflectivo no restrictivo sobre el que se despliega el sistema de persistencia.

Las referencias son punteros a las instancias que vendrán representadas por la clase `JRef`, pudiendo ser variables del usuario o bien atributos de una instancia, en cuyo caso sabrán también de quién forman parte, para notificarle cuándo son

modificadas y que de esta forma una instancia persistente pueda avisar al gestor de persistencia de que ha cambiado de estado y es necesario que se vuelva a almacenar.

Existe una clase especial, `JNullClass`, pensada para ser la clase de una instancia especial que represente el valor nulo (patrón *NullValue*). Su único cambio es el comportamiento de su método `isA()`, que devolverá verdadero sea cual sea el tipo por el que se le pregunte (pues efectivamente un valor nulo puede asignarse a una referencia de cualquier tipo).

12.2.2 Identificador de Persistencia

La creación de un identificador (ID) único para todo objeto persistente es uno de las tareas propias de todo sistema persistente. Puesto que los objetos persistentes de una aplicación van a sobrevivir a su ejecución, las referencias a éstos (sus direcciones de memoria) no serán válidas en posteriores ejecuciones. Por tanto, deberemos asignar un identificar único y global a todo objeto persistente.

El diseño del sistema de persistencia obliga a que toda instancia de objeto persistente deba retornar su ID en la invocación de su método `getID()`. La implementación de dicho método en la clase `JInstance` devuelve la concatenación de los siguientes valores separadas por el carácter "`.`":

- La dirección IP de la máquina
- El PID del proceso, o el carácter '1' si no está disponible.
- El UID del usuario, o el carácter '1' si no está disponible.
- El TID del hilo activo, o el carácter '1' si no está disponible.
- Los milisegundos transcurridos desde la medianoche del 1 de Enero de 1970 (si el reloj del sistema lo soporta se devuelve un resultado en coma flotante, en cuyo caso se mostrarán 3 dígitos decimales).
- Un número aleatorio en el rango [0, 32767].

Se ha elegido un formato tan complejo para el identificador con el fin de evitar en la medida de lo posible la colisión, teniendo en cuenta que actualmente se soportan múltiples aplicaciones y almacenamientos. Además, en un futuro podría extenderse el sistema para soportar concurrencia o incluso distribución en múltiples máquinas de una forma transparente, y en ese caso este sistema para generar identificadores de objetos podría seguir utilizándose sin cambios.

12.2.3 Subconjunto del Lenguaje Java Implementado

El lenguaje de referencia implementado se ha denominado Java--. Su nombre ya indica claramente que se ha creado tomando como modelo el lenguaje Java, concretamente la versión 1.0 del mismo [Gosling96], con algunas simplificaciones y modificaciones que se detallarán en los siguientes puntos –para una especificación más detallada, consúltese el Apéndice B.

12.2.3.1 Tipos primitivos

A diferencia de Java, en el lenguaje Java-- todas las variables son referencias a objetos, no existiendo tipos primitivos. Esta decisión fue motivada por el diseño

del sistema de persistencia, al requerir que todo objeto almacenado tuviese un GUID asignado (§ 12.2.2). De permitir tipos primitivos se dificultaría la implementación del sistema al tener que considerar en varios puntos del mismo casos especiales. Por otra parte, la supresión de tipos primitivos no afecta a la consecución del principal objetivo del prototipo: demostrar la viabilidad del sistema de persistencia presentado en este Trabajo.

12.2.3.2 Valores Lógicos

Una consecuencia de la decisión de no tener tipos primitivos es que es necesario cambiar el sistema de representación de los valores lógicos en el lenguaje. En Java existe un tipo básico `boolean` para representar el resultado de las operaciones lógicas. Sin embargo, Java-- carece de tipos primitivos, por lo que hubo que buscar otra forma de representar esos resultados. La elección fue emplear el mismo sistema que en Lisp: una referencia nula tiene valor lógico falso, mientras que una no nula tiene valor lógico verdadero. El problema que aparece ahora es el siguiente: ¿qué tipo devuelven los operadores lógicos? El tipo `boolean` en Java tiene una serie de propiedades apetecibles: no se puede ahormar a otro tipo, y con él sólo es posible efectuar operaciones lógicas. Finalmente se optó por crear una clase primitiva `Bool`, la única del lenguaje con constructor privado (se pretendía que sólo se pudiesen obtener como resultado de una operación lógica) y que sólo contaría con una única instancia llamada `true` (nuevamente a semejanza de Lisp con su valor `T`).

12.2.3.3 Operadores Internos y Externos

En el lenguaje Java-- existen dos tipos de operadores, que hemos llamado internos y externos. Los primeros reciben su nombre por estar implementados como métodos de las instancias (por lo que podrían ser modificables, aunque actualmente el lenguaje no lo permite –sin embargo la estructura necesaria ya está implementada). Los operadores externos, en cambio, trabajan sobre las referencias, y por tanto son ajenos a las instancias. La Figura 12.3 muestra qué operadores pertenecen a cada categoría.

	Internos	Externos
Aritméticos	+, -, *, /, %, ++, –	
Lógicos	<, <=, >, >=, ==, !=	&&, ||, !, is, ?:

Figura 12.3. Clasificación de operadores internos y externos.

Como se puede observar en la tabla, todos los operadores aritméticos son internos (como es lógico, puesto que necesitan el valor asociado a la instancia), mientras que los lógicos están repartidos entre aquellos que emplean el valor (y por tanto son internos) y los que se limitan a consultar si sus argumentos son o no nulos, o si dos referencias apuntan a la misma instancia (operador `is`), por lo que son externos.

12.2.3.4 Sobrecarga de métodos

Dado que se permite la sobrecarga de métodos, al encontrarnos con la invocación de un método debemos elegir uno de entre todos aquellos que la clase tenga definidos con ese nombre. El escogido será aquel en el que los tipos de los parámetros que deseamos pasarle sean lo más cercanos a los que se le indicaron en su definición.

Antes de explicar el algoritmo, detallaremos el concepto de distancia entre dos clases: dadas las clases X e Y, la función $dist\,ancia_c(X,Y)$ se define de la siguiente manera:

- Si $X \equiv Y$, esto es, son la misma clase, entonces $dist\,ancia_c(X,Y) = 0$.
- Si X no es descendiente directa o indirectamente de Y, entonces $dist\,ancia_c(X,Y) = -1$. Matemáticamente sería más correcto que la función no estuviese definida, pero el darle un valor que no se puede alcanzar de forma normal hará que se pueda operar con ella de una forma más sencilla, sin preocuparse de posibles casos de indefinición).
- En otro caso, $dist\,ancia_c(X,Y) = 1 + dist\,ancia_c(padre(X),Y)$.

Apoyándonos en la función $dist\,ancia_c(X,Y)$ definiremos ahora $dist\,ancia_M(def,act)$, que nos dará la distancia entre los parámetros de un método y sus argumentos actuales. Dado un método con n_{def} parámetros de tipos $tipodef_i$, y una llamada con n_{act} argumentos de tipos $tipoact_i$, y refiriéndonos como *def* y *act* a los argumentos y parámetros en su conjunto respectivamente, $dist\,ancia_M$ se define de la siguiente forma:

- Si $n_{def} \neq n_{act}$ (el número de parámetros y el de argumentos no coinciden), entonces $dist\,ancia_M(def,act) = -1$ (nuevamente sería un caso de indefinición, pero optamos por el valor □ 1 para simplificar el tratamiento).
- Si $dist\,ancia_c(tipoact_i, tipodef_i) = -1$ para algún valor de $i \in \{1, n_{def}\}$ (algún argumento no es de un tipo adaptable a los parámetros), entonces $dist\,ancia_M(def,act) = -1$.
- En otro caso,

$$dist\,ancia_M(def,act) = \max\{dist\,ancia_c\}(tipoact_i, tipodef_i) \forall i \in \{1, n_{def}\}\}$$

Una vez definidas estas funciones, podemos explicar qué método se elige al encontrarnos con una invocación. Si tenemos una llamada a un método de nombre M, seguiremos los siguientes pasos para elegir aquel que se ejecutará:

- Desechamos los métodos cuyo nombre base no sea M.
- Calculamos $d_m = dist\,ancia_M(def,act)$ para todos los métodos que nos queden.
- Desechamos aquellos métodos en que $d_m = -1$.
- Los métodos aptos serán aquellos en que $d_m = \min\{d_m\}$

Si no nos quedase ningún método apto no habría candidatos válidos, y si hubiera más de uno que fuese apto estaríamos ante un caso de ambigüedad que el usuario debería resolver con un *cast* (o varios). Si por el contrario sólo nos queda un método apto, ése será el que se ejecute.

12.3 Subsistema Intérprete

12.3.1 Diagrama de Clases

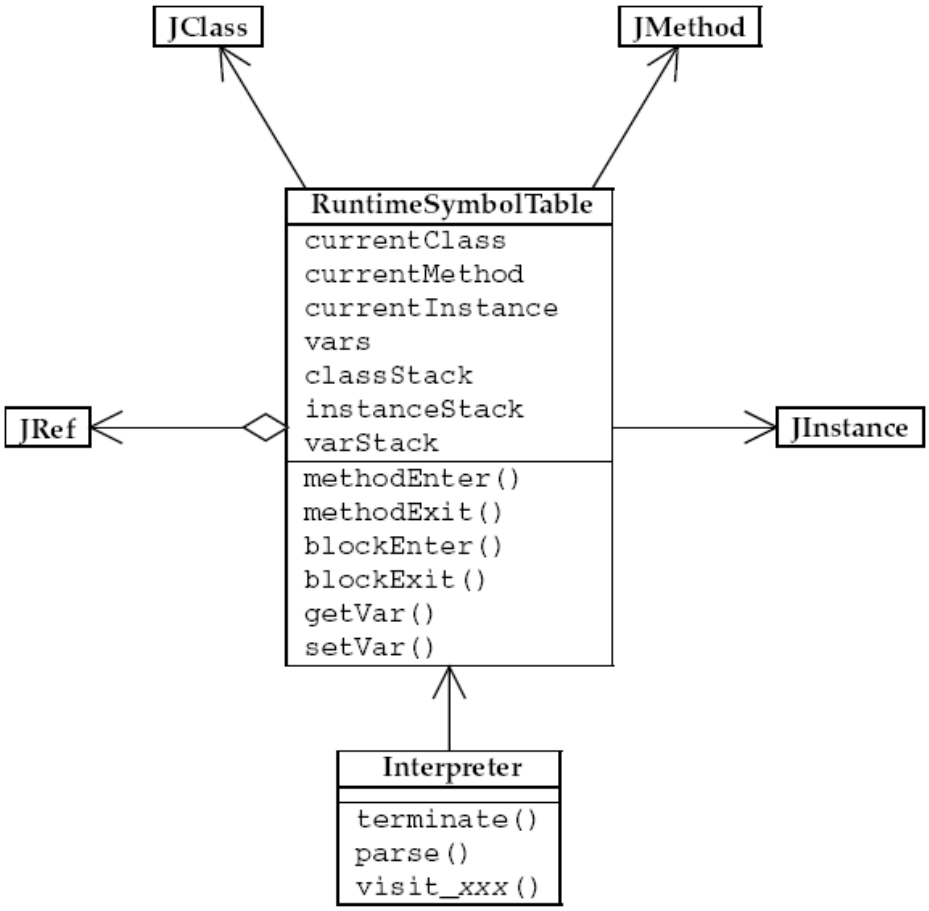


Figura 12.4. Diagrama de Clases del subsistema Intérprete.

El intérprete desarrollado se muestra en la Figura 12.4. La clase `Interpreter` será un *Singleton* [GOF94], es decir, existirá una única instancia del intérprete. El objeto intérprete trabajará sobre el modelo de objetos independiente del programa que represente la aplicación en ejecución. Dicho modelo será creado utilizando el intérprete genérico desarrollado sobre el prototipo de la plataforma reflectiva no restrictiva (§ 11.6), accesible a través del objeto fachada [GOF94] `nitrO`. El intérprete genérico recibirá de entrada:

- La aplicación a ejecutar.
- La especificación del lenguaje en la que está escrita la aplicación. Esta especificación incluirá la gramática libre de contexto tal y como se explicó en § 11.4. En las reglas de producción de la gramática deben incluirse las acciones necesarias para construir el modelo de objetos independiente que representa la aplicación a ejecutar. Dicho modelo se estructurará como un árbol de objetos utilizando las clases definidas en el subsistema Lenguaje (§ 12.2).

Con estos parámetros de entrada, nitrO generará en primer lugar el árbol sintáctico de la aplicación a ser ejecutada. Entonces, se ejecutará (siguiendo el pa-

trón de diseño *Command* [GOF94]) la rutina semántica especificada al final de cada producción. Este proceso devuelve el árbol sintáctico abstracto (AST, *Abstract Syntax Tree*) del programa de entrada, siguiendo el proceso descrito en § 11.5.

El intérprete toma el AST y lleva a cabo su interpretación. Este mecanismo está basado en realizar distintos recorridos del AST (patrón *Composite*), decorándolo siguiendo el patrón *Visitor* [GOF94] cuyos fundamentos fueron explicados en § 10.3.2. Como resultado de este proceso se obtiene un AST que representa el programa en ejecución utilizando el modelo de objetos definido por el subsistema Lenguaje.

Gracias a las características dinámicas de Python se ha podido simplificar la estructura del patrón. Se representado el AST correspondiente al modelo de objetos independiente utilizando listas anidadas, indicándose en su primer elemento el nombre del elemento `xxx` correspondiente a cada nodo. El intérprete presenta un método `parse()` que procesa estos nodos invocando al método `visit_xxx` que realizará el tratamiento adecuado de cada elemento. Una ventaja sustancial de esta implementación es que no necesita crearse una clase por cada tipo de nodo.

Sobre este modelo computacional único construido se realizarán las fases de análisis semántico y ejecución de la aplicación. La modificación de los métodos de visita utilizados para la ejecución de la aplicación conlleva la reflectividad computacional de la aplicación ejecutada.

La tabla de símbolos es otro *Singleton* [GOF94] que contiene un diccionario de referencias, `vars`, con las variables accesibles desde el punto de la ejecución en el que se encuentre el intérprete en cada momento. También conoce el método que se está ejecutando, así como la instancia sobre la que se está ejecutando (el argumento `this`), y emplea un par de pilas para almacenar ambos datos. Además, existe una tercera pila en la que se guarda el diccionario de variables cada vez que se entra a un nuevo bloque de código.

12.4 Subsistema Persistencia

12.4.1 Diagrama de Clases

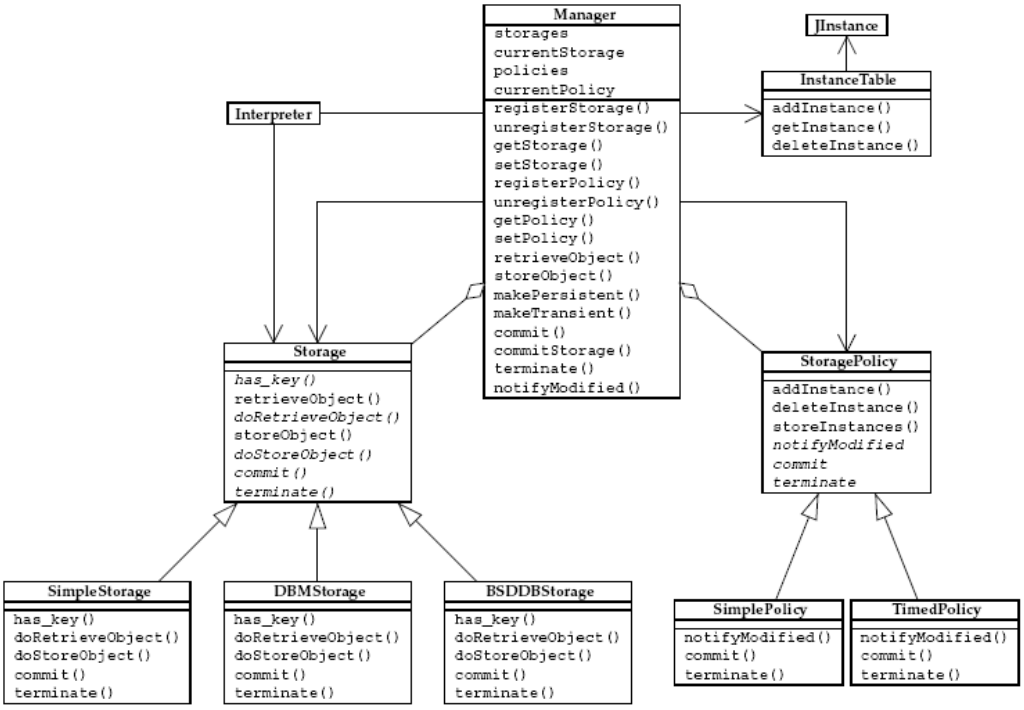


Figura 12.5. Diagrama de clases del subsistema de persistencia.

La Figura 12.5 muestra el modulo de persistencia. La clase `Manager` es la Fachada (*Facade*) [GOF94] y ha sido implementada mediante un *Singleton* [GOF94]. Ofrece servicios de persistencia al programador, permitiendo modificar el comportamiento de éste dinámicamente mediante la selección de objetos derivados de `Storage` y `StoragePolicy`. Estos objetos encapsulan las estrategias (patrón *Strategy* [GOF94]) correspondientes, respectivamente, al almacenamiento de los objetos en los sistemas físicos y a la política de actualización de objetos.

De este modo, nuestro sistema permite utilizar distintos sistemas de almacenamiento y de políticas de actualización de objetos. Las clases abstractas `Storage` y `StoragePolicy` ofrecen implementaciones parciales de tales funcionalidades (patrón *Template*), facilitando así la adición de nuevas clases derivadas. Las clases derivadas de `Storage` representan distintas formas de almacenar información persistencia, así como distintos sistemas de indexación. En el caso de `StoragePolicy`, se especifica la frecuencia con la que los objetos tengan que ser actualizados en el almacenamiento seleccionado. La selección, intercambio y modificación dinámica de estas dos variables puede ser llevada a cabo de un modo programático.

Hemos desarrollado tres almacenamientos de referencia:

- `SimpleStorage`: Es un sencillo diccionario que es salvado y cargado de disco. Es el almacenamiento por omisión seleccionado por el gestor de persistencia.

- `BSDDBStorage`: Proporciona el acceso a una *Berkeley DB Library*. El usuario puede crear sistemas de almacenamiento basados en técnicas de *hashing* extendido lineal, árboles B+ o registros de longitud variables, en función de los parámetros pasados a su constructor. Este sistema de almacenamiento puede ser usado para hacer cambiar dinámicamente el mecanismo de indexación, en función de contextos y condiciones surgidos en tiempo de ejecución.

- `DBMStorage`: Esta clase ofrece una librería del tipo *Unix (n)dbm*. El sistema de almacenamiento se comporta como una memoria asociativa persistente, teniendo en cuenta que tanto la clave como el contenido han de ser cadenas de caracteres.

También hemos desarrollado dos políticas de referencia distintas para la actualización de objetos:

- `SimplePolicy`: La actualización de los objetos persistentes en el sistema de almacenamiento (implementada por el método `commit`) se producirá siempre que el estado de un objeto haya sido modificado un número especificado de veces. Ésta es la política por omisión, con una única modificación necesaria para actualizar el objeto.

- `TimedPolicy`: Se emplea un temporizador parametrizado por un número determinado de milisegundos. Cuando el temporizador alcanza el número de milisegundos seleccionado, se invoca al método `commit` haciendo que todo objeto que se haya modificado desde la última actualización sea volcado a disco.

Cada uno de los parámetros mencionados en las políticas y almacenamientos anteriores puede ser modificado dinámicamente, en función de requisitos surgidos durante la ejecución de la aplicación.

12.4.2 Almacenamiento de Objetos

El los sistemas de almacenamiento implementados de referencia, hemos utilizado el módulo `pickle` de Python capaz de serializar objetos –convertirlos a una secuencia de bytes y viceversa. Aunque este módulo puede ser utilizar para almacenar objetos, no aborda el problema de asignar IDs a los objetos persistentes. Es por ello por lo que hemos implementado nuestro propio sistema de identificadores, descrito en § 12.2.2 . El proceso de convertir los IDs de objetos persistentes a referencias en memoria se denomina *swizzling*; el proceso contrario es llamado *unswizzling*.

El gestor de persistencia implementa un sistema de (*un*)*swizzling* perezoso (*lazy*). El en el caso de la traducción de referencias a IDs persistentes, *unswizzling*, es realizado justo antes de almacenar los objetos en el sistema de persistencia. Si el objeto tiene referencias a otros objetos persistentes, este proceso de traducción es efectuado de un modo recursivo de forma paralela a la serialización de los objetos.

El proceso inverso (*swizzling*) de desarrolla en dos fases. El objeto demandado es buscado inicialmente en el sistema de almacenamiento a partir de su ID. En este paso, la secuencia de bytes que representa a este objeto es extraída del sistema de almacenamiento y convertida en un objeto. Posteriormente se lleva a cabo el *swizzling*, recuperando los enlaces originales entre los objetos.

El proceso descrito se consigue empleando la clase `InstanceTable` representada en la Figura 12.5. Esta tabla es un diccionario de referencias débiles. Cada vez que un objeto se hace persistente, se añade una entrada a éste en la tabla. De este modo, si se necesita un objeto que tiene una entrada en la tabla se obtendrá de ésta su referencia, actuando así como una caché.

Es importante resaltar que la tabla usa referencias débiles: si el objeto persistente deja de ser referenciado en la aplicación, el recolector de basura podrá eliminarlo. Cuando un objeto persistente es requerido y no está en la tabla de instancias, el gestor de persistencia lo obtendrá del sistema de almacenamiento seleccionado.

12.4.3 Modificación de un Objeto Persistente

Cuando una instancia marcada como persistente se modifica (se llama a su método `setValue()` o se llama al método `setInstance()` de alguna de las referencias que componen sus miembros), la instancia avisa al gestor de persistencia llamando a su método `notifyModified()`, quien a su vez llama al método homónimo de la política de persistencia activa, tal y como se muestra en la Figura 12.6. De esta forma, la política de actualización podrá coleccionar las instancias modificadas para, en un futuro, poder almacenarlas de un modo adecuado. El sistema sabe cuándo un objeto ha sido modificado aumentando la semántica de asignación de atributos (reflectividad computacional).

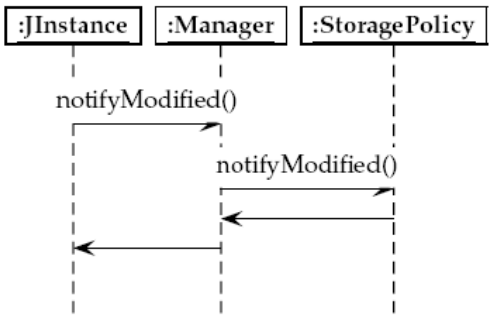


Figura 12.6. Diagrama de secuencia de la modificación de una instancia persistente.

Cuando la política elegida decide llevar a cabo la actualización de objetos modificados, invocará a su método `storeInstances`. Esta llamada generará la invocación de `storeObject` en el gestor, para cada una de las instancias que hayan sido modificadas. Entonces cada objeto se preparará para ser volcado a disco mediante el algoritmo de serialización y *unswizzling* mencionado previamente. Finalmente, una vez que los objetos se hayan preparado para actualizarse, se sincronizará la tabla de instancias con el sistema de almacenamiento realizando la transacción oportuna.

La Figura 12.7 muestra el proceso que se sigue una vez que se entra en el método `storeInstances()` de la política (pensado para ejecutarse cuando la política decida hacer un `commit` o el usuario ordene hacerlo explícitamente). La política va indicando al gestor todas las instancias que tiene que almacenar. El gestor, por su parte, le indica a la instancia que debe guardarse en el almacenamiento que se tenga activo en esos momentos.

El proceso de almacenar una instancia es un recorrido en profundidad del grafo formado por cada instancia y sus campos. La instancia le indica primero a ca-

da uno de sus campos que debe almacenarse, y cuando todos lo han hecho ella misma se guarda en el almacenamiento. Cuando se ha terminado con todas las instancias, el último paso es que la política le pida al gestor que ordene al almacenamiento hacer un `commit`, de forma que los cambios se hagan permanentes.

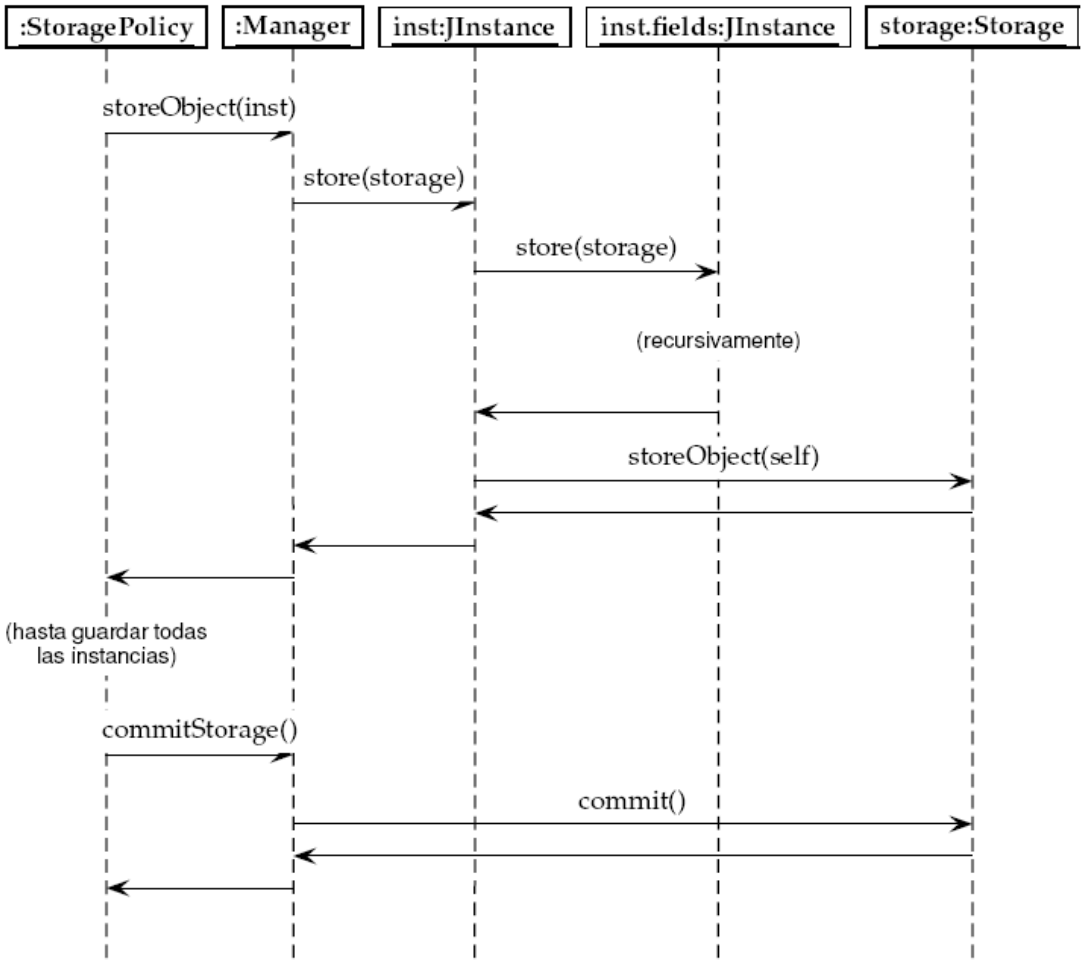


Figura 12.7. Diagrama de secuencia de la ejecución de una acometida por la política.

Capítulo 13
ÁMBITOS DE APLICACIÓN DEL SISTEMA

En este capítulo se analizarán una serie de áreas donde la utilización del sistema de persistencia presentado en esta memoria está especialmente indicada. En algunos casos, la viabilidad de la aplicación propuesta se demuestra con el desarrollo de un prototipo.

13.1 Separación del Aspecto de Persistencia

Hemos visto cómo con ninguno de los sistemas ni técnicas estudiadas, incluida la programación orientada a aspectos, es imposible una separación total de la incumbencia de persistencia.

Gracias a la utilización de la reflectividad computacional ofrecida por el sistema reflectivo no restrictivo, nuestro sistema sí consigue que los mecanismos de persistencia estén totalmente separados de la aplicación en ejecución [Ortin2004]. El código de la aplicación no necesita ningún cambio para poder hacer uso del sistema de persistencia, o lo que es lo mismo, el desarrollador puede centrarse en programar la aplicación sin tener en cuenta ningún detalle relativo a la persistencia de los objetos del dominio de su aplicación. Esto supone una serie de ventajas de aplicación inmediata al desarrollo de software:

- Se reduce la complejidad de desarrollo. Esta complejidad no radica en la configuración de los mecanismos de persistencia, sino en la interacción entre la aplicación y los sistemas de almacenamiento [Atkinson95], puesto que pertenecen a ámbitos de ingeniería radicalmente distintos. Nuestro sistema permite que esta interacción desaparezca de cara al desarrollador.

- Aumenta la mantenibilidad y legibilidad. La incumbencia de la persistencia se encuentra totalmente separada de la aplicación. Cambios en los requisitos del sistema de persistencia no afectan a la aplicación, y viceversa. Por tanto, la legibilidad de ambas incumbencias aumenta.

- Reutilización. Gracias al diseño del sistema de persistencia propuesto en esta Tesis, se pueden reutilizar todos los servicios del sistema de persis-

tencia, independientemente de la estructura y del lenguaje en el que esté codificada la aplicación que hace uso de dichos servicios.

13.2 Sistemas de Persistencia Adaptable

Las características de flexibilidad sin restricciones del sistema de persistencia lo hacen un candidato idóneo a la hora de implementar sistemas de persistencia adaptables, es decir, sistemas donde el usuario/administrador pueda adaptar el sistema de persistencia ante cambios que detecta durante su ejecución.

Hemos implementado un prototipo que demuestra esta adaptabilidad [López2004]. Se trata de un ejemplo de aplicación de autores, tomada de la información almacenada en el servidor Web DBLP [Ley2002]. El modelo de datos está compuesto de un conjunto de elementos bibliográficos (revistas, conferencias, libros, actas y artículos), editoriales, autores, localizaciones y editores (Figura 13.1).

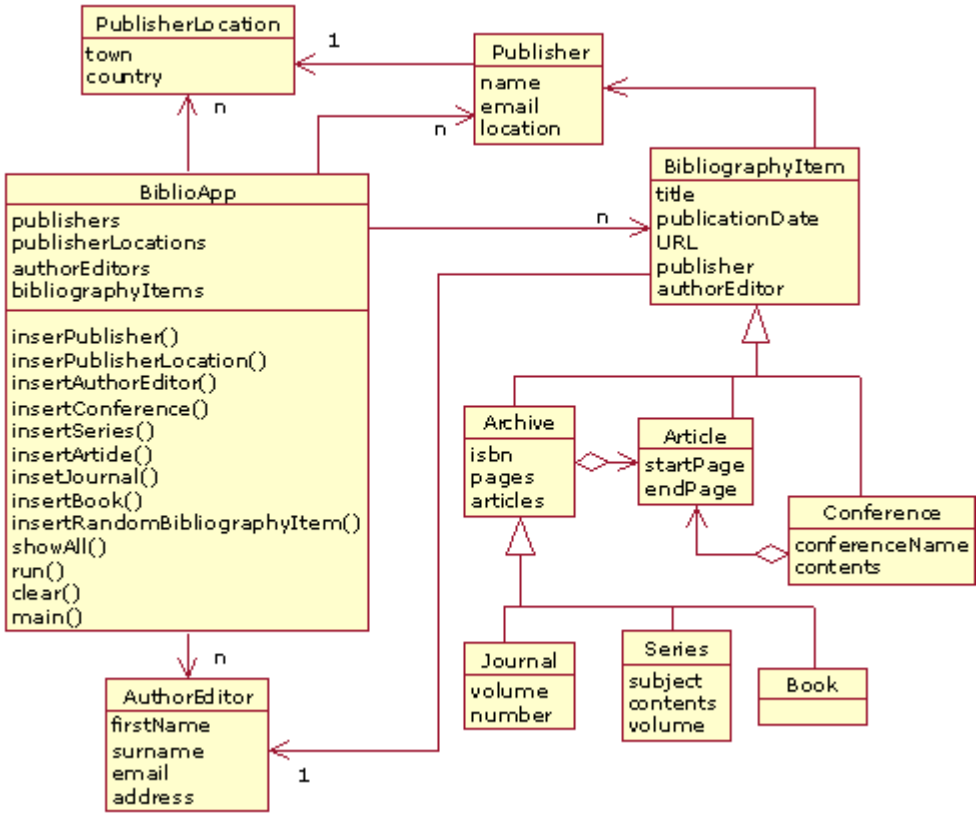


Figura 13.1. Modelo de objetos de la aplicación.

La aplicación ha sido desarrollada en su totalidad en el lenguaje de programación Java--, siendo ésta no persistente inicialmente. El programa se ejecutará permitiendo al usuario crear objetos y enlazarlos entre sí. Una instancia de la clase `AplicacionBiblografica` posee distintas colecciones de las editoriales, autores, editores, localizaciones y elementos bibliográficos. Al finalizar la ejecución de la aplicación, la memoria de estas colecciones es liberada –no son persistentes.

Aparte de la aplicación bibliográfica, desarrollamos otro programa que controla la incumbencia de la persistencia de primera aplicación, permitiendo asignar, eliminar y modificar dinámicamente los aspectos propios de persistencia del primer programa. Esta implementación será posible gracias a la utilización de reflectividad

computacional no restrictiva, utilizando código Python dentro de instrucciones `reify`.

Mediante un menú gráfico el controlador de persistencia permitirá al usuario hacer la aplicación bibliográfica, no persistente inicialmente, persistente. Por otra parte, permite cambiar en tiempo de ejecución el sistema de almacenamiento, política de actualización de objetos y el sistema de indexación utilizado. El código reflectivo accederá a la aplicación `Biblio`, permitiendo adaptar su persistencia mediante la invocación de métodos del gestor (`retrieveObject`, `makePersistent`, `makeTransient`, `setStorage` y `setPolicy`).

La Figura 13.2 muestra un escenario de ejemplo en el que los objetos de la aplicación son recuperados de una ejecución previa. Las dos ventanas superiores muestran el programa bibliográfico con su menú gráfico, estando la lista de elementos mostrada inicialmente vacía (ventana superior izquierda). Mediante el empleo del shell de `nitrO` lanzamos el controlador de persistencia (dos ventanas inferiores) permitiendo asignar y modificar la persistencia del primer programa. Seleccionando la opción `Recuperar Estado`, la aplicación bibliográfica recuperará indirectamente un conjunto de objetos persistentes procedentes de una ejecución previa. En ese momento, si se vuelven a mostrar las distintas colecciones de elementos, veremos la lista presentada en la Figura 13.2.

Es posible hacer la aplicación persistente mediante el controlador, obteniendo así una sincronización totalmente transparente entre los objetos del programa y el sistema de almacenamiento. Nótese que, aunque la aplicación bibliográfica es ahora persistente, no ha sido necesario escribir en ella ninguna sentencia de código para almacenar, obtener y actualizar objetos en disco. Lo único que gestiona es la lógica de la aplicación; el gestor de persistencia hace el resto. La Figura 13.2 también muestra cómo los sistemas de almacenamiento (ventana de abajo a la derecha) y las políticas de actualización de objetos pueden ser modificadas dinámicamente.

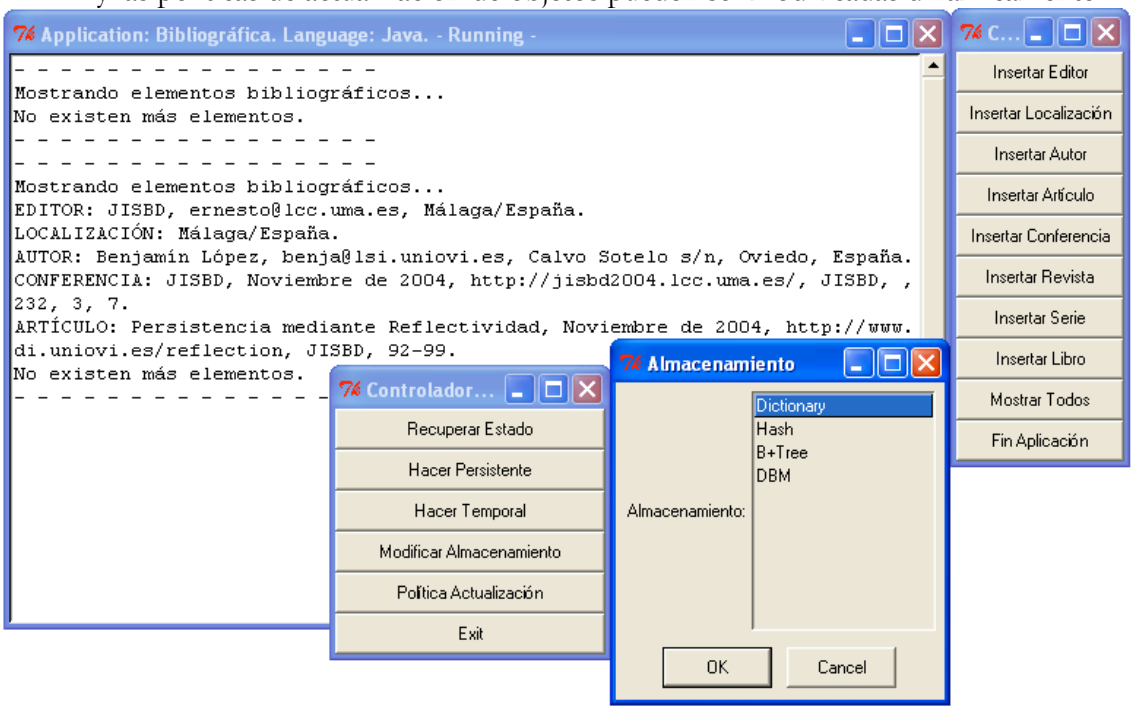


Figura 13.2. Ejecución de la aplicación bibliográfica y su controlador de persistencia.

El ejemplo presentado muestra cómo un usuario puede adaptar las características persistentes de una aplicación en tiempo de ejecución, no siendo necesario

modificar su código fuente. Mediante el empleo de reflectividad computacional, este sistema ofrece una separación total de la incumbencia (aspecto) de persistencia. El sistema de persistencia es, además, adaptativo: permite adaptar una aplicación de un modo programático, en función de variables tales como la carga del sistema, el nivel de persistencia, el número de usuarios conectados o incluso la estructura de las aplicaciones en ejecución. Permitirá, por tanto, hacer programas que adapten a otros programas en función de contextos surgidos dinámicamente.

Siguiendo el punto de vista de los sistemas de persistencia ortogonal, una alternativa completamente transparente puede ser llevada a cabo con nitrO. Adicionalmente a especificar al comienzo de un programa su identificador y su lenguaje, podría añadirse un elemento adicional para describir los atributos persistentes de la aplicación (almacenamiento, sistema de indexación y política de actualización). De este modo, se conseguiría una mayor transparencia que en los sistemas actuales.

13.2.1 Replicación del Sistema de Almacenamiento

La posibilidad de configurar diversos sistemas de almacenamiento simultáneamente permite replicar los mecanismos de almacenamiento utilizados. Un usuario administrador podría, sin necesidad de detener la aplicación que está ejecutándose, añadir nuevos sistemas de almacenamiento que repliquen el utilizado en un momento dado. La replicación de sistemas de almacenamiento puede responder a razones de disponibilidad (por redundancia o política copias de seguridad) y también a razones de eficiencia, para obtener un reparto de carga.

13.2.2 Modificación de la Actualización de la Información en Función de la Carga del Sistema

Dependiendo de la carga del sistema un administrador puede establecer que la actualización de objetos en memoria con su representación en el sistema de almacenamiento se realice con mayor o menor frecuencia. Cuando el sistema presenta una carga alta, puede establecer una frecuencia de actualización alta para que el sistema de persistencia no consuma los recursos que escasean. No obstante, a mayor frecuencia de actualización, mayor probabilidad de perder datos si se caen los sistemas (disminuye la seguridad).

13.2.3 Selección Explícita de Objetos Persistentes

El usuario del sistema puede especificar qué objetos son hechos persistentes con el objeto de tener un mayor control sobre el sistema de persistencia que cuando se utiliza una política de persistencia por alcance. Las razones pueden ser varias, por ejemplo, de seguridad, para que no se almacenen las contraseñas. También por razones de dominio, la persistencia de ciertos objetos no tiene sentido y se desea ahorrar recursos. En cualquier caso, gracias a la adaptabilidad, del sistema el usuario podrá utilizar la política de selección de objetos que considere adecuada o incluso incorporar nuevos mecanismos de selección si ninguno cubre sus requisitos.

13.2.4 Cambio de Formato en el Almacenamiento de un Conjunto de Objetos Existentes en Tiempo de Ejecución

En determinadas circunstancias puede ser interesante configurar diferentes sistemas de almacenamiento para los objetos de una aplicación. Por ejemplo, puede desearse almacenar detalles de configuración como ficheros XML para su manipulación por el usuario, y los objetos de dominio en bases de datos relacionales por razones de eficiencia. Otro escenario puede ser el de una aplicación que maneje ficheros de datos de carácter personal de diferentes niveles, a los que se les aplican diferentes medidas del reglamento de seguridad y protección de datos. A los datos de nivel alto se les configuraría un formato de almacenamiento remoto y encriptado, mientras que al resto no sería necesario. La ausencia de restricciones en la adaptabilidad del sistema de persistencia permitiría afrontar requisitos de este tipo en tiempo de ejecución sin necesidad de modificar el código de la aplicación.

13.2.5 Generación de Informes de depuración y *Logs*

Los *logs* e informes de depuración son una herramienta habitual de los administradores de sistemas de persistencia. Las características del sistema reflectivo no restrictivo sobre el que se ejecuta la aplicación permiten explorar cualquier detalle de la misma, añadir código de *logging* e, incluso, modificar su comportamiento para afrontar situaciones anormales. Del mismo modo, el sistema de persistencia también se ejecuta sobre el mismo sistema reflectivo, por lo que también pueden elaborarse detallados informes que expliquen su funcionamiento. De cara al administrador, el sistema de persistencia representa una plataforma que no impone restricciones a la hora de elaborar informes internos; éstos podrán tener la forma, contenido y detalle que se deseé.

13.3 Sistemas de Persistencia Adaptativo

Un sistema de persistencia adaptativo es aquel capaz de detectar cambios en el contexto de su ejecución y adaptarse a ellos modificando su comportamiento.

13.3.1 Mecanismos de Indexación

En § 3.1.5 se analizaban diferentes mecanismos de indexación y se concluía que un sistema de persistencia debería de ofrecer un mecanismo de indexación adaptativo, debido a que unas técnicas se comportaban mejor que otras según la estructura del modelo de objetos sobre el que se realiza la indexación. Si este modelo cambia dinámicamente, el sistema adaptativo detectaría el cambio y cambiaría con él.

13.3.2 Políticas de Actualización

Como ya se ha comentado en § 13.2.2, según la carga del sistema puede ser interesante modificar la frecuencia con la que se actualizan los objetos en el sistema de almacenamiento, es decir, cambiar la política de actualización de objetos. Además de un usuario administrador que detecte la necesidad del cambio, el propio sistema (u otra aplicación) podría automáticamente detectar el exceso (o defecto) de carga y adaptarse automáticamente.

13.3.3 Políticas de Clustering

En § 3.3 estudiamos una serie de sistemas que ofrecían cierta adaptabilidad sobre sus políticas de `clustering`. Vimos cómo la eficiencia del algoritmo utilizado para agrupar objetos dependía de la estructura de éstos, incluso en situaciones de carga fija [Manolis92]. El sistema de persistencia podría analizar la estructura de los objetos que almacena y decidir configurar dinámicamente la estrategia de *clustering* utilizada.

13.4 Plataforma de Ajuste de Parámetros de Persistencia

El sistema reflectivo no restrictivo propuesto y el sistema de persistencia desplegado sobre él permiten construir una plataforma que se encargue de monitorizar el rendimiento del sistema durante su ejecución y, de manera automática, ajuste los componentes de persistencia para afrontar los cambios en el entorno de una manera óptima.

En esta ocasión también hemos desarrollado un prototipo [López2004b] basado en la aplicación bibliográfica derivada de la información almacenada en el servidor Web DBLP [Ley2002] (§ 13.2).

Toda la aplicación ha sido desarrollada en Java y no es persistente en absoluto. Su clase principal es `BiblioApp`, la cual crea aleatoriamente diferentes elementos bibliográficos, los modifica (a través del método `run`) y, finalmente, los restaura a su estado inicial (con el método `clear`). El código recogido en la Figura 13.3 es el método `run` de la clase `BiblioApp`. Mide el tiempo empleado en insertar y modificar un número determinado de elementos bibliográficos. La referencia `cons` representa la consola de la ventana gráfica de la aplicación.

```
Application = "Biblio"
Language = "Java"
…
class BiblioApp {
…
  void run() {
    Integer insertions = 50;
    Timer timer = new Timer();
    cons.println('Measuring '+insertions.toString()+' insertions...');
    insertPublisherLocation();
    insertPublisher();
    timer.start();
    for(Integer i = 0; i < insertions; ++i) {
      insertRandomBibliographyItem();
    }
    timer.stop();
    cons.println(insertions.toString() + ' insertions: ' +
                 timer.getTime() + ' seconds');
    cons.println('Measuring '+data.numItems.toString()+' updates...');
    timer.start();
    for(Integer i = 1; i <= data.numItems; ++i) {
      BibliographyItem item = (BibliographyItem)data.
                  bibliographyItems.getItem('Title' + i.toString());
      item.setUrl('http://www.anotherURL.com');
    }
    timer.stop();
    cons.println(insertions.toString() + ' updates: ' +
                       timer.getTime() + ' seconds');
  }
}
```

Figura 13.3. Código de la clase BiblioApp.

13.4.1 Asignación de la Persistencia

Esta aplicación se ejecuta en el sistema nitrO con el identificador `Biblio`. Su ejecución muestra el tiempo empleado en insertar y modificar 5.000 instancias almacenadas en memoria RAM. Una vez esta aplicación ha empezado, se puede desarrollar otro programa que, usando el sistema reflectivo persistente, establecerá y modificará dinámicamente las características de persistencia de la aplicación `Biblio` aunque hayan sido escritos en diferentes lenguajes.

El nuevo programa (denominado *Benchmark)* va a ejecutar bucles anidados dentro de los cuales se asignarán a la aplicación `Biblio` diferentes almacenes, políticas de actualización y mecanismos de indexación. Durante cada iteración, el *benchmark* llama a los métodos `run` y `clear` de la aplicación bibliográfica causando el almacenamiento, modificación y borrado de objetos persistentes con diferentes configuraciones de persistencia. El código mostrado en la Figura 13.4 es un fragmento de la aplicación *benchmark*.

```
[1]  reify <#
[2]  import persistence

[3]  try:
[4]    biblioApp = nitrO.apps['Biblio']
[5]  except KeyError, e:
[6]    raise SystemExit('The application "Biblio" is not running.')

[7]  biblioInterpreter=biblioApp.applicationGlobalContext['theInterpreter']
[8]  biblioManager = biblioInterpreter.getPersistenceManager()
[9]  biblioPrint=nitrO.apps["Biblio"].window.writeText
[10] symtable = biblioInterpreter.getSymbolTable()

[11] def setStorage(storage):
[12]   biblioManager.setStorage(storage)
[13]   print("Setting "+storage+" persistence storage")
[14]   biblioPrint("Setting "+storage+" persistence storage\n")

[15] def setPolicy(policy):
[16]   biblioManager.setPolicy(policy)
[17]   print("Setting "+policy+" update policy")
[18]   biblioPrint("Setting "+policy+" update policy\n")

[19] appInstance = symtable.getVar('app').getInstance()
[20] appClass = appInstance.getClass()

[21] print("\nDYNAMIC PERSISTENCE BENCHMARK")
[22] biblioPrint("\nDYNAMIC PERSISTENCE BENCHMARK\n")
[23] for policy in application.policies:
[24]   print("\n------- "+policy+" Policy -------")
[25]   biblioPrint("\n------- "+policy+" Policy -------\n")
[26]   for storage in application.storages:
[27]     print("\n"+storage+" Storage:")
[28]     biblioPrint("\n"+storage+" Storage:\n")
[29]     makePersistent()
[30]     setPolicy(policy)
[31]     setStorage(storage)
[32]   appInstance.getClass().getMethod('run',()).invoke(
           biblioInterpreter,appInstance,())
[33]   appInstance.getClass().getMethod('clear',()).invoke(
           biblioInterpreter,appInstance,())
[34] #> }
```

Figura 13.4. Aplicación Benchmark.

El código mostrado es una única sentencia `reify`. Por consiguiente, se ejecuta al nivel del intérprete y tiene acceso a la aplicación `Biblio`. Este acceso es logrado utilizando el objeto fachada [GOF94] nitrO (línea 4), mediante el cual se accede al objeto que representa la aplicación bibliográfica. Si la aplicación está ejecutándose también se obtienen su intérprete (línea 7), gestor de persistencia (línea 8), consola (línea 9) y tabla de símbolos (línea 10).

Para cambiar dinámicamente los almacenes persistentes y las políticas de actualización, se definen las funciones `setStorage` (línea 11) y `setPolicy` (línea 15). Ellas indican al sistema de persistencia qué almacén y política deben establecerse, mostrando un mensaje tanto en la aplicación bibliográfica como en la aplicación *benchmark*. Las líneas 19 y 20 se encargan de recoger de la tabla de símbolos la referencia `app` del método principal de la aplicación `Biblio` y su clase `BiblioApp`.

De la línea 23 a la 33 pueden observarse los bucles anidados mencionados anteriormente. En cada iteración se selecciona un almacén y política y se llaman a los métodos `run` y `clear` de la instancia `app` (líneas 32 y 33). Hemos empleado un diccionario, DBM, *hashing* y árboles *B+Tree*. Las políticas de actualización utilizadas: cada vez que se modifica el estado de un objeto, cada 10 modificaciones, cada segundo y cada 30 segundos.

13.4.2 Ejecución del Benchmark

Las ventanas de cada aplicación en ejecución se muestran en la Figura 13.5. Puede observarse la consola de nitrO en la parte inferior de la figura: la ventana principal donde se le puede decir a nitrO qué aplicaciones deben ejecutarse. La aplicación a la izquierda es la aplicación bibliográfica mostrando los tiempos de inserción y modificación medidos. Finalmente, la ventana en la derecha es el programa *benchmark* que modifica las opciones de persistencia de la aplicación bibliográfica.

Debe señalarse que la primera información mostrada por la aplicación bibliográfica es la inserción y modificación en memoria. La razón es que el *benchmark* no había sido lanzado y la aplicación no era persistente inicialmente. Las posteriores medidas son todas persistentes y son recogidas por el *benchmark* en ejecución.

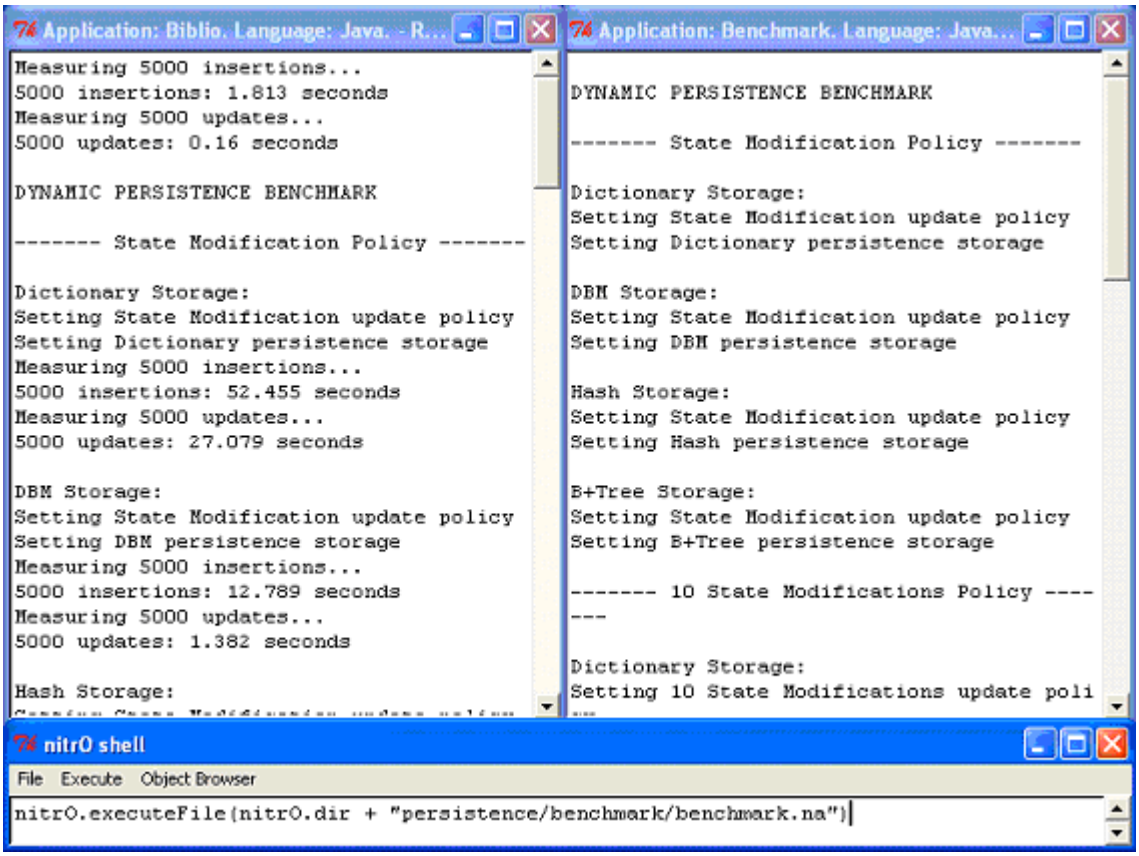


Figura 13.5. Ejecución de las aplicaciones en el sistema nitrO.

En este ejemplo se midieron 5 tipos diferentes de almacenamiento (incluida la memoria) con cuatro políticas. El número de inserciones y modificaciones fue 5.000. Todas las pruebas se ejecutaron sobre un sistema iPIII 1.0 GHz con 256 MBytes de RAM ejecutando Windows XP con una carga ligera. La métrica utilizada fue el tiempo de ejecución. La muestra las diferentes medidas de forma gráfica.

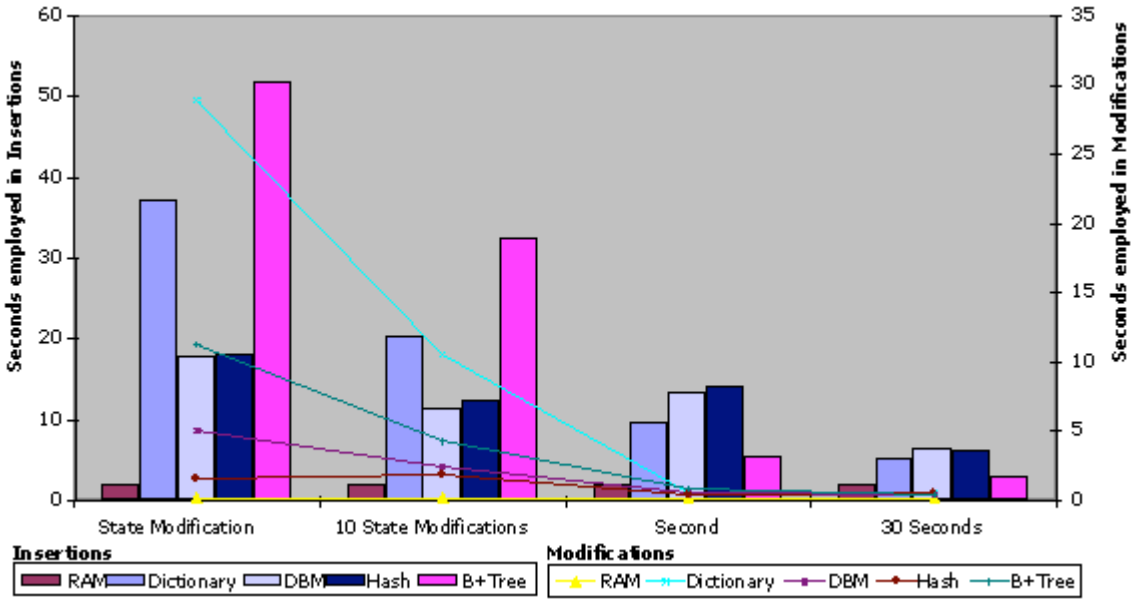


Figura 13.6. Ejecución del benchmark con 5.000elementos bibliográficos.

Para ilustrar cómo el sistema de persistencia puede ser utilizado para obtener un ajuste de los parámetros de persistencia, se realiza a continuación un primer análisis de los resultados obtenidos ejecutando el `benchmark` descrito:

- Se confirma la incompatibilidad existente entre seguridad y rendimiento: cuanto más alta es la frecuencia de actualización, peor es el rendimiento (la modificación del estado es la política más segura, pero también la más lenta). Esto se aprecia en la tendencia descendente de las mediciones.

- El almacenamiento utilizando árboles B+ es el más eficiente cuando se utilizan políticas periódicas, pero cuando se utilizan políticas relacionadas con la modificación de estado, *hash* y DBM son mejores.

- Cuando se utilizan las políticas de modificación de estado, las inserciones con los árboles B+ son las más lentas. Sin embargo, bajo las mismas condiciones, sus modificaciones son mucho más rápidas que las realizadas sobre diccionarios.

- Con un número pequeño de ítems bibliográficos (hasta 100), el mejor almacén, independientemente de la política utilizada, es el diccionario. Sin embargo, cuando existe un número grande de objetos, su rendimiento es desastroso.

Dependiendo de variables tales como la carga del sistema, el nivel de persistencia, el número de usuarios conectados, o incluso la estructura de la aplicación, cualquier aplicación (o fragmento de ella) podría seleccionar dinámicamente la configuración de persistencia más adecuada de un modo programático. Por lo tanto, nuestro sistema de persistencia es capaz de adaptarse a contextos únicamente conocidos durante el tiempo ejecución, ofreciendo un ajuste de parámetros óptimo.

Capítulo 14
EVALUACIÓN DEL SISTEMA

El sistema de persistencia presentado en esta Tesis será evaluado en función de los requisitos establecidos con anterioridad, en el Capítulo 2. Todos los sistemas presentados y estudiados en esta memoria han sido evaluados en función de estos criterios. Para llevar a cabo la valoración de un modo comparativo, evaluaremos de forma paralela un grupo de sistemas reales, adicionalmente al presentado. Los sistemas elegidos han sido aquéllos que obtuvieron una mejor valoración dentro de su clasificación.

Para evaluar el conjunto de sistemas, estableceremos una clasificación de los requisitos y estudiaremos éstos tanto en función en esta categorización, como de un modo global (absoluto). Una vez cuantificadas y representadas las distintas evaluaciones, analizaremos los beneficios aportados por nuestro sistema en comparación con el resto de los estudiados.

14.1 Comparativa de Sistemas

A la hora de comparar nuestro sistema con otros existentes, hemos seleccionado, del conjunto global de sistemas estudiados en este trabajo, los más próximos a satisfacer los requisitos estipulados. De esta forma, nos centraremos en:

- JDO. Sistema de persistencia propuesto por Sun para la persistencia de modelos de objetos sobre diferentes sistemas de almacenamiento [Sun2003].
- Hibernate. Herramienta de mapeo objeto/relacional para Java de código abierto [Bauer2004].
- PJama. Sistema de programación persistente ortogonal desarrollado sobre una modificación de la máquina virtual de Java [Atkinson96].
- ZODB. Base de datos orientada a objetos específicamente diseñada para el lenguaje Python [Rossum2001], cuya principal característica es su transparencia.

- AspectJ. Sistema que ofrece programación orientada a aspectos con tejido estático [Kiczales2001]. Tomaremos como referencia para la evaluación el sistema de persistencia construido con AspectJ presentado en [Rashid2003].
- Persistent nitrO. Este es el nombre con el que haremos referencia al sistema de persistencia presentado en este documento.

14.2 Criterios de Evaluación

I. Criterios de transparencia.

   A. Separación de la competencia de persistencia. La persistencia será identificada, configurada y añadida como cualquier componente reutilizable.

   B. Reutilización independiente de la funcionalidad. Sea cual fuere la funcionalidad de la aplicación el sistema de persistencia podrá adaptarse a ella.

   C. Reutilización independiente de la estructura. El sistema de persistencia será independiente de las estructuras de datos empleadas en las aplicaciones.

   D. Automatización integral de la plataforma. El sistema de persistencia deberá ser capaz de conocer, de un modo automático e implícito, el momento oportuno en el que se deban llevar a cabo las llamadas a las primitivas de persistencia.

   E. Ausencia de impacto en los procesos de desarrollo. La utilización del sistema de transparencia no afectará al ciclo de desarrollo utilizado por el programador de la aplicación que hace uso de sus servicios.

II. Criterios de adaptabilidad.

   A. Adaptabilidad dinámica. Los distintos parámetros del sistema de persistencia deberán ser configurables por un usuario administrador en tiempo de ejecución sin modificar el código fuente de la aplicación.

   B. Adaptabilidad programática. El propio sistema (u otra aplicación) podrá adaptarse en tiempo de ejecución.

   C. Mecanismos de indexación. Adaptación implícita de los mecanismos de indexación utilizados.

   D. Sistemas de almacenamiento. Número arbitrario de sistemas de almacenamiento que pueden funcionar simultáneamente y configurarse en tiempo de ejecución.

   E. Políticas de actualización. Deberá permitirse la adición, eliminación o modificación dinámica de las políticas de actualización que determinan cómo deben actualizarse los objetos en el sistema de persistencia.

F. Selección de objetos persistentes. Deberá ofrecerse la posibilidad de configurar un número arbitrario de mecanismos de selección de objetos persistentes.

III. Criterios de portabilidad.

A. Independencia del lenguaje. La programación de aplicaciones persistentes deberá poder realizarse mediante cualquier lenguaje de programación.

B. Independencia del sistema operativo. El sistema deberá poder implantarse en cualquier sistema operativo.

C. Independencia del hardware. La interfaz de acceso a la plataforma no deberá ser dependiente del hardware en el que haya sino implantado.

14.3 Evaluación

La evaluación del sistema será equiponderada respecto a los criterios descritos. La consecución plena de un objetivo será ponderada con la unidad; su carencia, con un valor nulo; los factores intermedios serán cuantificados en la mayoría de los casos cualitativamente –para conocer una justificación somera de la evaluación, consúltese § 14.4.

14.3.1 Criterios de Transparencia

Criterio	JDO	Hibernate	PJama	ZODB	AspectJ	Persistent nitrO
I.A	0,7	0,7	0,9	0,7	0,8	1
I.B	1	1	1	1	1	1
I.C	0,5	0,7	1	0,9	0,5	1
I.D	0,7	0,7	0,9	0,7	0,9	1
I.E	0	1	1	1	0	1
Total	**2,9**	**4,1**	**4,8**	**4,3**	**3,2**	**5**

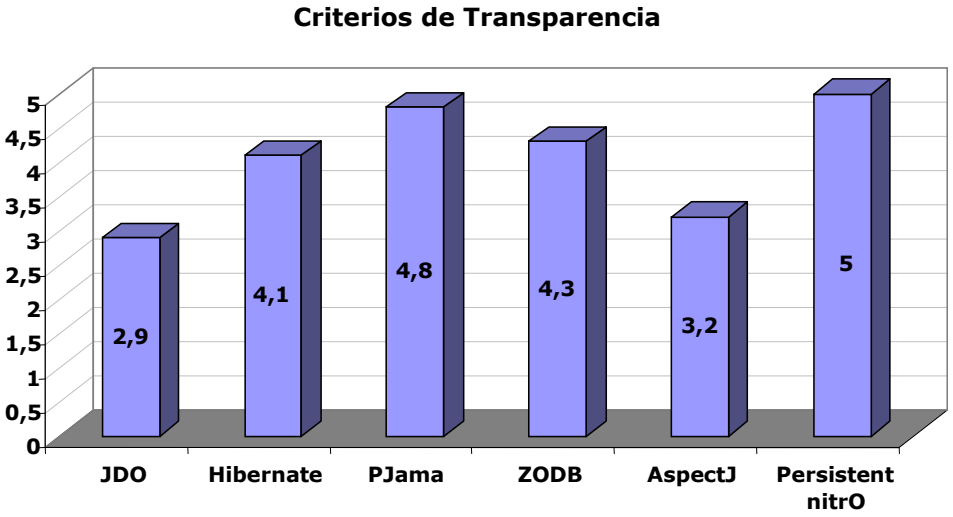


Figura 14.1. Evaluación de los Criterios de Transparencia.

14.3.2 Criterios de Adaptabilidad

Criterio	JDO	Hibernate	PJama	ZODB	AspectJ	Persistent nitrO
II.A	0	0	0	0	0	1
II.B	0,1	0,1	0	0,1	0	1
II.C	0	0	0	0	0	1
II.D	0,5	0	0	0	0	1
II.E	0	0,1	0	0	0	1
II.F	0,5	0,5	0	0	0	1
Total	**1,1**	**0,7**	**0**	**0,1**	**0**	**6**

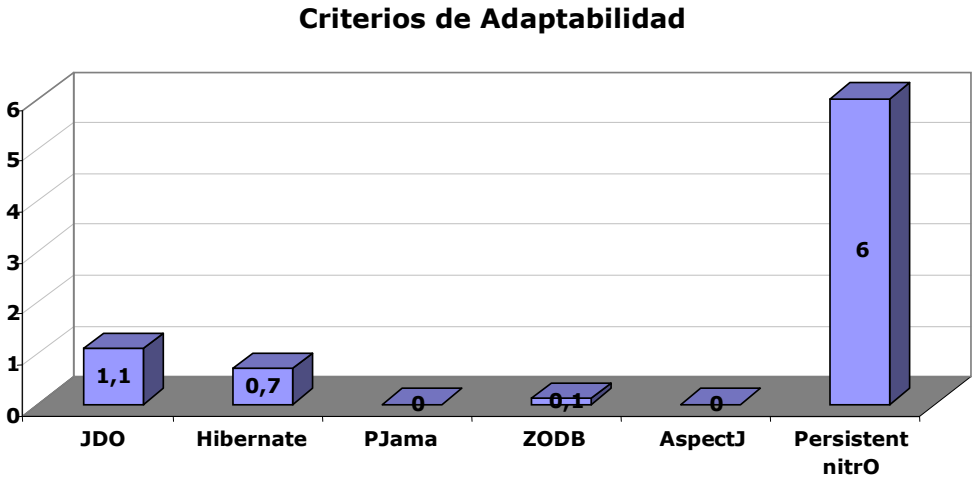


Figura 14.2. Evaluación de los Criterios de Adaptabilidad.

14.3.3 Criterios de Portabilidad

Criterio	JDO	Hibernate	PJama	ZODB	AspectJ	Persistent nitrO
III.A	0	0	0	0	0	1
III.B	1	1	1	1	1	1
III.C	1	1	1	1	1	1
Total	**2**	**2**	**2**	**2**	**2**	**3**

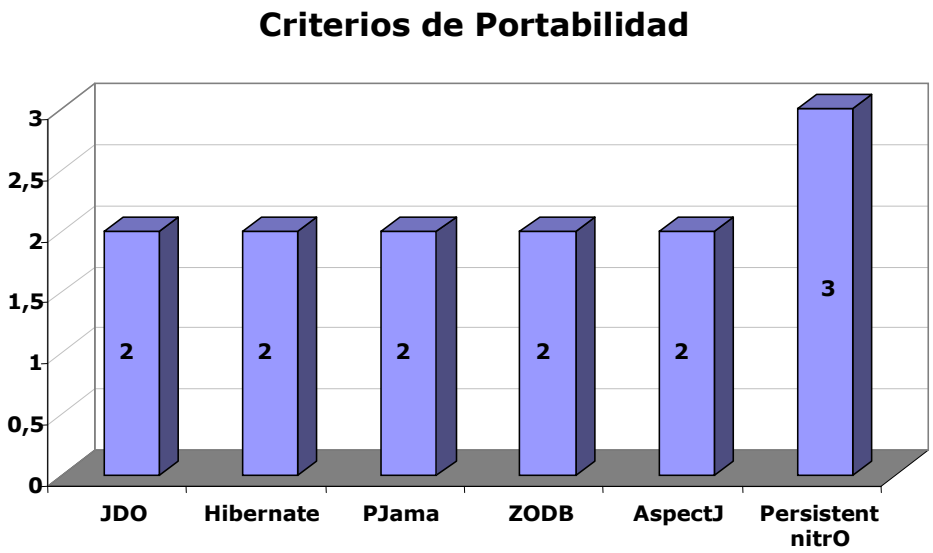


Figura 14.3. Evaluación de los Criterios de Portabilidad.

14.3.4 Evaluación Global del Sistema

Criterio	JDO	Hibernate	PJama	ZODB	AspectJ	Persistent nitrO
I	2,9	4,1	4,8	4,3	3,2	5
II	1,1	0,7	0	0,1	0	6
III	2	2	2	2	2	3
Total	**6**	**6,8**	**6,8**	**6,4**	**5,2**	**14**

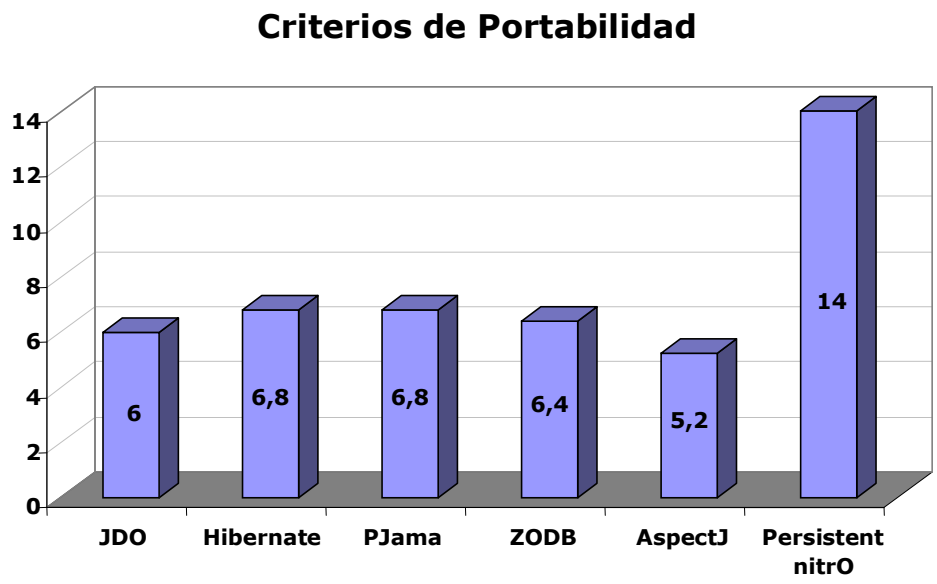


Figura 14.4. Evaluación global de todos los criterios.

14.4 Justificación de las Evaluaciones

A continuación describiremos, de forma somera, la evaluación de los distintos criterios seleccionados previamente para cada uno de los sistemas estudiados. Si se desea una ampliación de las justificaciones, puede consultarse el capítulo 3 los sistemas JDO, Hibernate, PJama, ZODB, el capítulo 5 para AspectJ, y el Capítulo 8 para las valoraciones de los las justificaciones relativas al sistema Persistent nitrO.

I. Criterios de transparencia.

   A. JDO, Hibernate y ZODB obligan a la demarcación explícita de transacciones. Salvo Persistent nitrO, ningún sistema permite eliminar las sentencias explícitas de obtención y borrado de objetos persistentes.

   B. Todos los sistemas ofrecen la suficiente flexibilidad como para adaptarse a cualquier tipo de aplicación.

   C. La modificación a nivel de bytecode en tiempo de diseño que realizan JDO y AspectJ impide ofrecer persistencia a *arrays* Java, con lo que se invalidan la mayor parte de contenedores desarrollados por los usuarios. Hibernate obliga a referenciar los contenedores en base a sus interfaces, aconseja no utilizar *arrays* por eficiencia y los desarrolladores deben prestar atención a la implementación de los métodos `equals()` y `hashcode()` heredados de `java.lang.Object`. ZODB únicamente permite almacenar los objetos que son serializables por el módulo `pickle`. Además, obliga a prestar una especial consideración a la hora de utilizar contenedores por el mecanismo de actualización implementado.

   D. JDO, Hibernate y ZODB obligan a la demarcación explícita de transacciones. Ningún sistema, salvo Persistent nitrO, permite hacer implícita la primitiva de borrado de objetos persistentes

E. JDO y AspectJ obligan a utilizar herramientas específicas que alteran el desarrollo habitual de aplicaciones.

II. Criterios de adaptabilidad.

A. Ninguno de los sistemas estudiados, salvo Persistent nitrO, permite su modificación en tiempo de ejecución por parte de un usuario administrador.

B. La única adaptación programática en tiempo de ejecución que permiten realizar JDO, Hibernate y ZODB es la relativa a su configuración. Esta configuración debe realizarse antes de ejecutar el sistema de persistencia, por lo que no puede hablarse de una adaptación programática real. Hibernate ofrece una API para manipular el metamodelo de configuración de la herramienta que ofrece mayor comodidad para manipular la configuración programáticamente pero no añade dinamismo al mecanismo.

C. Ningún sistema, salvo Persistent nitrO, implementa ningún tipo de adaptación implícita y dinámica de los mecanismos de indexación.

D. JDO permite configurar diferentes sistemas de almacenamiento, aunque no permite configurar la utilización de varios simultáneamente, ni tampoco permite su incorporación dinámica. El resto de sistemas, salvo Persistent nitrO, únicamente permiten utilizar una tecnología de almacenamiento predeterminada.

E. Únicamente Persistent nitrO permite configurar diferentes políticas de actualización de objetos. En JDO, Hibernate y ZODB la única política de actualización que existe es la acometida de transacciones explícita. PJama y AspectJ permiten realizar las actualizaciones de forma implícita pero no permiten su configuración. Hibernate permite configurar diferentes algoritmos de `fetching` (carga) de objetos, para controlar como se traen los objetos de memoria.

F. Hibernate y JDO permiten especificar cómo se propaga la persistencia transitivamente a través de los metadatos de configuración, aunque mediante un mecanismo predefinido que no se puede modificar.

III. Criterios de portabilidad.

A. El único sistema de persistencia independiente del lenguaje es Persistent nitrO. ZODB únicamente funciona con el lenguaje Python. El resto del sistema están restringidos a Java.

B. Todas las plataformas definidas son abstractas e independientes del sistema operativo sobre el que se ejecutan.

C. Todas las plataformas definidas son abstractas e independientes del hardware sobre el que se ejecutan.

14.5 Conclusiones

14.5.1 Evaluación de los Criterios de Transparencia

Analizando los resultados de la evaluación de los criterios de transparencia se observa que JDO y AspectJ son los sistemas más desfavorecidos. Ambos trabajan modificando el bytecode Java de forma estática lo que impide proporcionar persistencia para *arrays*. Además, ambos exigen que el desarrollador utilice herramientas específicas sobre las clases que van a ser hechas persistentes, lo que repercute en la transparencia del sistema de persistencia.

El resto de sistemas ofrecen un buen nivel de transparencia. El mejor nivel de transparencia, exceptuando a Persistent nitrO, es el ofrecido por PJama que obtiene todo el soporte que necesita para la persistencia de una máquina virtual Java modificada a tal efecto.

14.5.2 Evaluación de los Criterios de Adaptabilidad

En la evaluación de los criterios de adaptabilidad sí se observan notables diferencias. La adaptabilidad de dos aplicaciones comerciales de ámbito empresarial como son JDO e Hibernate se limita a un conjunto restringido de parámetros de configuración, en su mayor parte relacionados con la posibilidad de obtener configuraciones eficientes.

ZODB también presta mucha atención a la eficiencia, pero en su filosofía prima la idea de ofrecer un sistema de persistencia totalmente transparente limitando la configuración que tiene que realizar el usuario al mínimo. Lo mismo puede decirse de PJama, puesto que su fundamento es demostrar la viabilidad de desarrollar un sistema de persistencia ortogonal sobre una plataforma comercial.

No obstante, debe señalarse que no es incompatible ofrecer un alto grado de adaptabilidad a todos los niveles –por un usuario, programáticamente, por otras aplicaciones– con la oferta de un sistema de persistencia totalmente transparente. El sistema de persistencia presentado en este trabajo valida esta afirmación.

14.5.3 Evaluación de los Criterios de Portabilidad

Ninguno de los sistemas evaluados en la comparativa ofrecen independencia del lenguaje, exceptuando Persistent nitrO. ZODB únicamente ofrece persistencia para el lenguaje Python, y el resto de sistemas ofrecen persistencia sobre la plataforma Java. Otro hecho destacable es que estos sistemas presentan una gran dependencia del lenguaje para el que han sido desarrollados.

Con respecto a la independencia del hardware y del sistema operativo, todos los sistemas evaluados presentan esta propiedad por presentarla la plataforma sobre la que han sido desarrollados.

14.5.4 Evaluación Global del Sistema

En la evaluación global del sistema lo primero que se observa es una gran similitud entre los resultados globales obtenidos para JDO, Hibernate, PJama y

ZODB. Esto indica que no existe un sistema claramente superior al resto y que las carencias se ven compensadas con las virtudes según los criterios utilizados.

Destaca que la mejor puntuación, exceptuando a Persistent nitrO, se ha obtenido con una herramienta de mapeo objeto/relacional como Hibernate. Entre las virtudes de esta herramienta está el ofrecer un gran abanico de opciones de configuración que se añaden a un sistema de persistencia con un buen nivel de transparencia.

Finalmente debe decirse que la enorme diferencia existente entre las evaluaciones de los diferentes sistemas y la de Persistent nitrO radica en las diferencias en materia de adaptabilidad. Mientras que el soporte de los sistemas evaluados para su adaptación dinámica es muy pobre –en muchos casos inexistente– Persistent nitrO se despliega sobre una plataforma reflectiva no restrictiva donde la aplicación en ejecución, el intérprete que la ejecuta y el propio sistema de persistencia pueden ser accedidos y modificados en tiempo de ejecución. Esta característica tiene un coste en materia de eficiencia (ver siguiente apartado) pero proporciona una capacidad de adaptación dinámica cualitativamente superior al resto de los sistemas.

14.6 Eficiencia

El mayor inconveniente la adaptabilidad dinámica de una aplicación es su eficiencia en tiempo de ejecución [Böllert99]. El proceso de adaptar un programa, sumado al hecho de utilizar reflectividad, supone un consumo de recursos adicional en la ejecución de la aplicación [Popovici2001]. La adaptabilidad y la eficiencia son variables que normalmente son contrarias.

En nuestra primera implementación hemos tratado de obtener el mayor grado de adaptabilidad en tiempo de ejecución, siguiendo fielmente el principio de la separación de incumbencias. Por ello, no hemos evaluado la eficiencia del sistema de persistencia.

Una vez validada la Tesis aquí presentada, un trabajo futuro es mejorar el rendimiento del sistema. Para ello, al igual que ha sucedido en la utilización comercial de lenguajes con una naturaleza interpretada (Java o .Net), utilizaremos técnicas de generación dinámica de código nativo (*Just In Time*, JIT) y optimización dinámica (*HotSpot*), acelerando la ejecución del sistema en un número de veces comprendido entre 2 y 3 [Krall98].

Capítulo 15
CONCLUSIONES

A lo largo de esta Tesis Doctoral hemos analizado las distintas alternativas para crear un sistema de persistencia totalmente transparente, que permitiese realizar una separación efectiva de la incumbencia de persistencia y que ofreciese una alta flexibilidad dinámica que le permitiese adaptarse o ser adaptado a diferentes contextos en tiempo de ejecución.

Comenzamos estudiando en el Capítulo 3 diferentes sistemas de persistencia existentes, tanto en el ámbito comercial como en el académico. Concluimos que ningún sistema de persistencia existente ofrecía un grado total de transparencia y que el soporte para la adaptabilidad dinámica era muy escaso o inexistente en todos los casos (Capítulo 14).

Para solventar esta limitación se estudió, en el Capítulo 4, un conjunto de técnicas de construcción de sistemas flexibles. Una de estas técnicas, la programación orientada a aspectos, fue estudiada con mayor detalle en el Capítulo 5 y se concluyó que no ofrecía ventajas cualitativas sobre la mera transformación de código que ya se había mostrado insuficiente en Capítulo 3.

En el Capítulo 7 se estudió la aplicación de una de las técnicas más empleadas en la construcción de sistemas computacionales flexibles: la reflectividad. Tras el estudio de los sistemas reflectivos existentes, vemos cómo existen carencias de adaptación en tiempo de ejecución, de flexibilidad en su semántica, o de interacción entre la adaptación de distintas aplicaciones. Los sistemas que ofrecen modificación dinámica de su semántica, carecen principalmente de la posibilidad de modificar su lenguaje, de configurar entre sí las aplicaciones y, sobre todo, de utilizar un mecanismo de modificación de su semántica que no limite el número de primitivas semánticas a modificar y el modo en el que el sistema pueda ser adaptado (MOPs). Por ello, se observó que ninguno de los sistemas computacionales flexibles permitía alcanzar los requisitos impuestos a nuestro sistema de persistencia, por lo que se concluyó la necesidad de desarrollar un sistema reflectivo no restrictivo sobre el que desplegar el sistema de persistencia.

En este capítulo, estudiaremos cómo el diseño de nuestro sistema ha superado todas las limitaciones puntualizadas, satisfaciendo los requisitos preestablecidos a lo largo de este documento –Capítulo 2. Finalmente analizaremos las principales ventajas aportadas.

15.1 Sistema Diseñado

El sistema diseñado (Capítulo 8) se basa en la especificación de un sistema de persistencia que se despliega sobre un sistema reflectivo no restrictivo.

15.1.1 Sistema Reflectivo no Restrictivo

Para obtener independencia del lenguaje de programación y flexibilidad computacional dinámica sin restricciones, el sistema de persistencia se ejecutará sobre la capa que constituye el sistema reflectivo no restrictivo (Capítulo 9).

Inicialmente se desarrolla un intérprete genérico capaz de evaluar cualquier programa, independientemente de su lenguaje de programación. Este intérprete recibe la aplicación a ejecutar y la especificación del lenguaje en el que ésta haya sido codificada –puede recibir ambos parámetros en un único archivo. Su ejecución supone una traducción dinámica del modelo descrito por su lenguaje al modelo computacional del motor computacional.

La evaluación de una aplicación da lugar a la creación de dos estructuras de objetos ubicadas fuera del intérprete: la especificación de su lenguaje y la traducción de la ejecución de la aplicación en el modelo computacional de la máquina. Mediante la reflectividad estructural de la máquina, ambas representaciones pueden ser dinámicamente modificadas. El resultado es un sistema que, independientemente del lenguaje utilizado, permite modificar cualquier característica de su lenguaje y de la representación dinámica de su ejecución.

La flexibilidad obtenida no restringe el número de elementos adaptables –toda la especificación del lenguaje puede configurarse por la propia aplicación– ni la expresividad de cómo puede llevarse a cabo dicha adaptación –el propio lenguaje del motor computacional es empleado para llevar a cabo las modificaciones oportunas. Adicionalmente, este mecanismo es ofrecido por el intérprete genérico de forma independiente al lenguaje – aplicaciones desarrolladas en distintos lenguajes, pueden configurarse dinámicamente entre sí.

15.1.2 Sistema de Persistencia

El sistema de persistencia permitirá hacer persistente cualquier aplicación ejecutada sobre el sistema reflectivo no restrictivo, gracias a la utilización de los servicios de éste.

Dada una aplicación sobre la que se desean instalar los mecanismos de persistencia, en primer lugar se construye un modelo de objetos único que la representa y que es independiente del lenguaje de programación utilizado. Este modelo se basa en el concepto de metamodelo descrito por el OMG y utilizado en el desarrollo de software basado en modelos [OMG2001].

El sistema de persistencia manipula el modelo de objetos que representa la aplicación en ejecución: para instalar sobre ella los mecanismos de persistencia dinámicamente, para reconstruir los objetos leídos de los almacenes y para analizar la estructura de los objetos de cara a su persistencia. Por ello hará uso de la introspección y la reflectividad estructural ofrecida por el sistema reflectivo. En el mismo sentido, modifica el intérprete que ejecuta la aplicación para poder hacer implícitas determinadas primitivas de persistencia que no pueden obtenerse únicamente modificando la estructura de la aplicación; de este modo, está haciendo uso de reflectivi-

dad computacional para conseguir un sistema de persistencia totalmente transparente.

El sistema se configura en torno a una serie de componentes de persistencia que implementan las funcionalidades de persistencia más habituales. Estos componentes pueden ser instalados, desinstalados y modificados dinámicamente. El sistema puede además afrontar nuevos requisitos en tiempo de ejecución, por ejemplo, la adición de un nuevo componente, puesto que puede beneficiarse de todas las posibilidades de adaptabilidad del sistema reflectivo no restrictivo.

15.2 Principales Ventajas Aportadas

Detalladamente, el conjunto de ventajas aportadas por nuestro sistema es la consecución de los requisitos establecidos inicialmente en el Capítulo 2. Agrupando éstos y destacando los más representativos, podemos concluir las siguientes aportaciones principales.

15.2.1 Transparencia

Con el sistema presentado puede conseguirse un grado de transparencia total. Es decir, el código fuente de las aplicaciones no necesitará ningún cambio para incorporar el mecanismo de persistencia. La forma de las aplicaciones será la misma, independientemente de si sus objetos son persistentes o no.

En ninguno de los sistemas estudiados se alcanza este nivel de transparencia. La programación orientada a aspectos es habitualmente presentada como una candidata idónea para separar la incumbencia de la persistencia. Sin embargo, tal y como se vio en § 5.4 no puede conseguirse una separación total de la incumbencia de la persistencia utilizando programación orientada a aspectos.

La razón es que la programación orientada aspectos, así como los sistemas más avanzados estudiados en el Capítulo 3, basan su transparencia en la transformación del código de la aplicación que hace uso de los mecanismos de persistencia. Sin embargo, existen determinadas primitivas de persistencia que no pueden ser abordadas con una mera transformación de código (Figura 15.1). Por ejemplo, en un lenguaje con recolección de basura no existe en el código de la aplicación ninguna referencia al borrado de objetos. ¿Cómo puede entonces transformarse el código de la aplicación para que en el borrado del objeto se borre éste del almacén persistente?[31] En estos casos es necesario modificar las facetas computacionales del motor computacional que ejecuta la aplicación, es decir, su semántica. Esta modificación puede abordarse si el sistema presenta reflectividad computacional.

La diferencia entre la programación orientada a aspectos y la reflectividad computacional se ilustra con el ejemplo recogido en el esquema de la Figura 15.1. En el lado de la izquierda se observa cómo un programa P es traducido a un programa P' mediante un proceso de tejido[32] proporcionado por un sistema de POA. En negro se muestran las rutinas del lenguaje que, a través de una mera transformación de código, pueden tener su reflejo en el almacén persistente de una manera cuasi transparente. El código mostrado en gris representa el código concerniente a

[31] El borrado y la recuperación de objetos persistentes no pueden abordarse mediante la programación orientada a aspectos [Rashid2003] (§ 5.4).

[32] Desde el punto de vista de la transparencia el que el tejido sea estático o dinámico es irrelevante. Ambos enfoques se basan en la transformación del código.

la persistencia que se inserta durante el tejido. El código insertado se encarga de interactuar con el almacén persistente y, como resultado, el programa en ejecución presenta persistencia desde el punto de vista del desarrollador.

Nótese cómo la recuperación y borrado de objetos aparece en la aplicación cuando son competencias de persistencia [Rashid2003]. De este modo, el sistema de persistencia no es totalmente transparente desde el punto de vista del desarrollador.

En el caso de que estuviésemos desarrollando el código mediante el esquema tradicional (§ 3.2), el código desarrollado sería el propio del programa P', es decir, las competencias funcionales de la aplicación estarían entremezcladas con las rutinas del aspecto de la persistencia, diseminada a lo largo de todo el código fuente.

En el lado de la derecha de la Figura 15.1 se muestra el mismo proceso haciendo uso de reflectividad computacional. Como ésta permite cambiar la semántica del motor computacional que ejecuta la aplicación, se podría redefinir la acción a realizar al interpretar las operaciones de creación, modificación y borrado de objetos, así como el acceso a un objeto a través de una referencia, con el fin de que dichas operaciones tengan su reflejo en el sistema de almacenamiento.

La principal diferencia existente es, pues, que aunque la modificación de determinadas primitivas de computación pueden ser llevadas a cabo también mediante mecanismos de transformación de código, para otros casos, como el de borrado o recuperación de de objetos, sería necesario acceder a la semántica de recolección de basura o acceso a objetos a través de referencias para saber cuándo se eliminan los objetos de memoria volátil y se acceden a ellos respectivamente. Con reflectividad computacional se cambia la semántica del motor computacional para que, cuando un objeto sea eliminado de memoria (o accedido), dicha acción tenga su reflejo en el sistema de almacenamiento. De este modo, se hace implícitas operaciones que, por encima del motor computacional, siempre tendría que ser explícita.

En definitiva, la reflectividad computacional ofrece un nivel de modificación del motor computacional, superior a la mera transformación de programas [Ortín2004c], permitiendo que el nivel de transparencia (y adaptabilidad) alcanzado sea total.

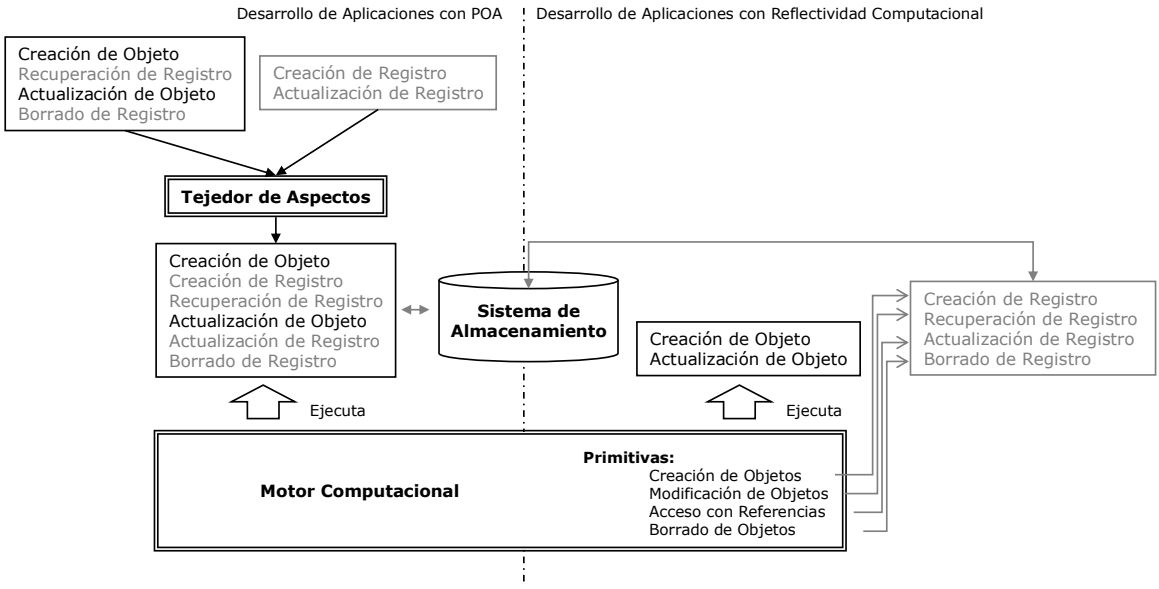


Figura 15.1. Diferencia entre la programación orientada a aspectos y la reflectividad computacional a la hora de proporcionar persistencia.

15.2.2 Adaptabilidad

El diseño sistema de persistencia ofrece un alto grado de adaptabilidad. Los posibles cambios a los que deba hacer frente el sistema se han considerado desde su diseño, elaborando mecanismos que faciliten la configuración de los distintos componentes de persistencia.

Para los cambios que no son tenidos en cuenta en el diseño, el salto computacional del sistema reflectivo no restrictivo ofrece la expresividad de un lenguaje para cambiar los parámetros de persistencia dinámicamente. Así, se podrá adaptar a contextos dinámicos, imprevistos en tiempo de ejecución.

Las ventajas referentes a la adaptabilidad son otra consecuencia de las diferencias existentes entre las alternativas estudiadas y la reflectividad computacional, comentadas en el punto anterior.

15.2.3 Independencia del lenguaje

Independientemente del lenguaje en el que estén codificadas, las aplicaciones serán representadas en el sistema con un modelo de objetos común (metamodelo). Como el sistema de persistencia trabajará sobre este modelo de objetos común, se obtiene un sistema de persistencia que puede ser utilizado por cualquier aplicación escrita en cualquier lenguaje, traduciendo las aplicaciones al modelo computacional único.

Gracias al intérprete genérico del sistema reflectivo no restrictivo, cualquier aplicación puede prepararse para ser utilizada por el sistema de persistencia. Para ello, debe elaborarse únicamente la especificación del lenguaje que toma el intérprete como parámetro junto con la aplicación propiamente dicha.

Capítulo 16
TRABAJO FUTURO

El trabajo de investigación realizado en esta Tesis abre nuevas líneas de investigación futuras, además de existir puntos en los que ampliar y mejorar las implementaciones asociadas al sistema presentado. Podemos identificar las siguientes líneas de trabajo inminente.

16.1 Eficiencia

El prototipo del sistema de persistencia fue diseñado sin tener en cuenta la eficiencia. Su objetivo era demostrar la transparencia y adaptabilidad que podían lograrse implementando el sistema de persistencia diseñado en este trabajo. En este apartado se plantean dos líneas de investigación relacionadas con la eficiencia del sistema de persistencia.

16.1.1 Utilización de Técnicas de Compilación Bajo Demanda

La razón principal de la caída de eficiencia de nuestra plataforma reflectiva es la interpretación de todos los lenguajes de programación. Hoy en día, es típico ver lenguajes interpretados empleados en la empresa (Java, C# o Python) debido a técnicas de optimización como compilación bajo demanda (JIT) o generación adaptativa de código nativo [Hölzle94]. En futuras versiones de nuestra plataforma, emplearemos estas técnicas para optimizar la implementación del intérprete genérico en un margen de 2 a 3 veces [Krall98].

Puesto que en el prototipo siempre traducimos cualquier lenguaje a Python, una forma de optimizar el sistema es utilizar una implementación de Python que haga uso de un de un compilador JIT, como la casi concluida IronPython [IronPython] para la plataforma .NET.

16.1.2 Modificación de Plataformas Nativas para Obtener Reflectividad

Esta línea de acción plantea implementar el sistema sobre la modificación realizada de la implementación SSCLI (Rotor) de la plataforma .NET: el proyecto rRotor [Ortín2005].

rRotor (*reflective Rotor*) es un proyecto que se centra en añadir reflexión estructural a la implementación SSCLI de Microsoft. Dado que el Rotor utiliza un compilador JIT para generar código nativo de forma dinámica, se ha obtenido un rendimiento de ejecución entre 5,74 y 9,01 veces superior a las implementaciones basadas en intérpretes puros [Ortín2005b].

16.2 Descripción de Lenguajes de Programación

Nuestro sistema es independiente del lenguaje de programación. Para utilizar un determinado lenguaje, hay que incluir en él su especificación. En el prototipo desarrollado han sido incluidas una especificación reducida de Python y una versión restringida de Java. Para facilitar la elección de múltiples lenguajes al programador de nuestro sistema, debería desarrollarse la especificación de los lenguajes más conocidos. Además, se deberán introducir características de Java que aún no se han implementado, tales como los tipos de datos simples o la implementación de su API.

16.3 Implementación de un Mayor Número de Elementos del Sistema de Persistencia

En el prototipo del sistema de persistencia, hemos desarrollado tres sistemas de almacenamiento de referencia:

- `SimpleStorage`: Es un sencillo diccionario que es salvado y cargado de disco. Es el almacenamiento por omisión seleccionado por el gestor de persistencia.
- `BSDDBStorage`: Proporciona el acceso a una *Berkeley DB Library*. El usuario puede crear sistemas de almacenamiento basados en técnicas de *hashing* extendido lineal, árboles B+ o registros de longitud variables, en función de los parámetros pasados a su constructor. Este sistema de almacenamiento puede ser usado para hacer cambiar dinámicamente el mecanismo de indexación, en función de contextos y condiciones surgidos en tiempo de ejecución.
- `DBMStorage`: Esta clase ofrece una librería del tipo *Unix (n)dbm*. El sistema de almacenamiento se comporta como una memoria asociativa persistente, teniendo en cuenta que tanto la clave como el contenido han de ser cadenas de caracteres.

También hemos desarrollado dos políticas distintas de actualización de objetos:

- `SimplePolicy`: La actualización de los objetos persistentes en el sistema de almacenamiento (implementada por el método `commit`) se producirá siempre que el estado de un objeto haya sido modificado un número especificado de veces. Ésta es la política por omisión, con una única modificación necesaria para actualizar el objeto.
- `TimedPolicy`: Se emplea un temporizador parametrizado por un número determinado de milisegundos. Cuando el temporizador alcanza el número de milisegundos seleccionado, se invoca al método `commit` ha-

ciendo que todo objeto que se haya modificado desde la última actualización sea volcado a disco.

En el futuro podrían implementarse nuevos componentes de persistencia para los mecanismos existentes: mecanismos de indexación, políticas de actualización y selección de objetos.

16.4 Ampliación de la Parametrización del Sistema de Persistencia

Una línea de ampliación del trabajo desarrollado muy interesante sería el desarrollo de nuevos parámetros del sistema de persistencia, no considerados durante su diseño.

Un ejemplo puede ser un sistema de transacciones. En la versión actual no existe ningún mecanismo de control transaccional. Se puede desarrollar un mecanismo de transacciones flexible, que permita configurar al usuario diferentes estrategias de control. Dos ejemplos de estrategias son un control implícito de las transacciones y una acometida explícita de transacciones.

Otro parámetro puede ser la configuración de diferentes estrategias de carga de objetos. Así se puede tener una estrategia de carga tardía (*lazy fetching*) que únicamente cargase un objeto cuando es referenciado, por necesidad de consumir pocos recursos. Otra estrategia puede basarse en cargar un determinado nivel del árbol de objetos por razones de eficiencia en ejecución.

Apéndice A

MANUAL DE USUARIO DEL SISTEMA REFLECTIVO NO RESTRICTIVO

Este documento narra cómo instalar, programar y utilizar el prototipo de computación reflectiva sin restricciones implementado en el lenguaje de programación Python [Rossum2001]. La programación de lenguajes para el sistema utiliza la semántica de Python, por lo que es necesario conocer éste si se desea diseñar uno.

El código fuente, diseño y distintas baterías de prueba pueden descargarse de la dirección:
`http://www.di.uniovi.es/reflection/lab/prototypes.html`.

A.1 Instalación y Configuración

El sistema reflectivo se ha codificado en Python 2.1, por lo que su descarga de `http://www.python.org` es necesaria previamente a su ejecución. Una vez descargado e instalado el intérprete del lenguaje Python, es necesario:

- Ubicar todos los archivos de extensiones `ml`, `py` y `pyw` en un directorio de nuestro sistema de archivos.
- Añadir a la variable de entorno PYTHONPATH la ruta en la que hayamos posicionado los archivos comentados.
- Ejecutar el archivo nitrO.pyw con el intérprete pythonw, para lanzar el sistema reflectivo.

La descripción detallada de cada uno de los archivos y directorios puede encontrarse dentro del archivo readme.txt que acompaña a la distribución de la aplicación.

A.2 Metalenguaje del Sistema

Una de las características del sistema desarrollado es su independencia del lenguaje de programación. Cualquier aplicación utilizando cualquier lenguaje, puede ser ejecutada en nuestro sistema, interactuando dinámicamente con otras aplicacio-

nes desarrolladas en otros lenguajes. Una aplicación podrá también construirse utilizando un conjunto de lenguajes.

Para que esta independencia del lenguaje sea posible en nuestro sistema de programación, es necesario idear un mecanismo de especificación de lenguajes que separe su descripción del sistema computacional. Para ello se ha descrito un lenguaje de especificación de lenguajes: un metalenguaje. Haciendo uso de éste, se puede describir un lenguaje en nuestro sistema de tres modos:

- Especificación del lenguaje mediante un archivo de extensión `ml`. Siguiendo la gramática § A.2.1, se describe el lenguaje y se almacena en el directorio del sistema con igual nombre que el lenguaje creado, y extensión `ml`.
- Especificándolo en el archivo de aplicación y anteponiéndolo a su código (como veremos en § A.3.2).
- Modificando un lenguaje ya existente previamente a la ejecución de una aplicación (detallado en § A.3.4).

A.2.1 Gramática del Metalenguaje

La siguiente gramática representa la especificación del metalenguaje utilizado en las especificaciones de lenguajes para nuestro sistema; los elementos en negrita representan componentes léxicos, para separarlos de los propios de la notación EBNF [Cueva98]:

```
<startLang> ::= LANGUAGE = ID <scan> <parser>
<skip> <notskip>
<scan> ::= SCANNER = { <scannerRules> }
<parser> ::= PARSER = { <parserRules> }
<skip> ::= SKIP = { <tokens> }
<notskip> ::= NOTSKIP = { <tokens> }
<scannerRules> ::= {<SR> ;}
<SR> ::= STRING ID -> <rightSR> {<moreRightSR>}
<rightSR> ::= {<rightSRItem>} <ruleCode>
<moreRightSR> ::= | <rightSR>
<rightSRItem> ::= STRING
| ID
<parserRules> ::= {<PR> ;}
<PR> ::= STRING ID -> <rightPR> {<moreRightPR>}
<rightPR> ::= <rightPRItems> <ruleCode>
<rightPRItems> ::= {ID}
| _REIFY_
<ruleCode> ::= CODE
| λ
<moreRightPR> ::= | <rightPR>
<tokens> ::= {STRING ;}
```

La primera parte es la identificación del lenguaje. Este nombre ha de ser único para el lenguaje y deberá coincidir con el nombre del archivo (si se está creando un archivo ml). A continuación se especifican las descripciones léxica, sintáctica, y los tokens de escape y reconocimiento automático.

A.2.2 Descripción Léxica

La descripción léxica se lleva a cabo dentro de la sección Scanner empleando reglas libres de contexto. Cada regla ha de estar precedida de una cadena de ca-

racteres –entre comillas dobles– descriptiva de su significado. La notación en la que puede expresarse cada regla es en BNF (*Backus Naur Form*) [Cueva98].

Los elementos no terminales son identificadores y los terminales cadenas de caracteres sensibles a mayúsculas / minúsculas. Toda producción ha de finalizar con un punto y coma, y la parte derecha de la producción se separa de la izquierda con la pareja de caracteres "->". Toda producción puede tener alternativas en su parte derecha, separadas con el carácter "|".

Puesto que el tratamiento de estas producciones está basado en un algoritmo descendente con retroceso (*backtracking*), si dos partes derechas pueden ser válidas para la entrada analizada, la primera será analizada y la segunda ignorada. Es buen criterio a seguir si esto se produce, el ubicar con anterioridad aquellas reglas que posean una parte derecha de mayor longitud que aquellas con las que pueda tener conflictos. Por lo tanto, la producción al vacío (□) deberá ubicarse como la última parte derecha de toda producción.

Las producciones pueden tener asociadas reglas semánticas para constituir definiciones dirigidas por sintaxis [Aho90]. El modo en que éstas son codificadas se describe en § A.2.5.

A.2.3 Descripción Sintáctica

Las reglas sintácticas de nuestro metalenguaje se ubican en la sección `Parser`. El modo en el que éstas son representadas coincide con las empleadas en la descripción léxica. La única diferencia es que la parte derecha de una regla sintáctica no puede poseer símbolos gramaticales terminales –éstos deben estar previamente especificados en la parte léxica.

El símbolo gramatical no terminal, ubicado en la parte izquierda de la primera producción, representa el símbolo inicial de la gramática; no es necesario que éste posea un identificador determinado.

La separación entre reglas léxicas y sintácticas supone básicamente una agrupación conceptual. Además, como se comentará en § A.2.4, en el reconocimiento de una producción léxica se ignora la detección de tokens de escape –no en el caso de las reglas sintácticas.

A.2.4 Tokens de Escape y Reconocimiento Automático

Las dos secciones restantes –`Skip` y `NotSkip`– facilitan la eliminación y reconocimiento automático de componentes léxicos en la aplicación a analizar. Si queremos que el analizador léxico del procesador de lenguaje elimine automáticamente un conjunto de tokens, debemos especificar éstos dentro de la sección `Skip`. El analizador sintáctico nunca tendrá noción de ellos.

La sección `NotSkip` produce el efecto contrario que `Skip`. Si queremos reconocer automáticamente un conjunto de tokens, sin necesidad de especificar su aparición sintácticamente, podremos hacerlo ubicándolos en esta sección. Un ejemplo de este tipo de tokens puede ser el tabulador en el lenguaje Python [Rossum2001]: el código ha de esta indentado (sangrado) mediante tabuladores para saber a qué estructura de control pertenece, pero, ¿cómo contemplar esto en su gramática?

Si un token `NotSkip` es reconocido automáticamente, su lexema aparecerá en el texto (atributo `text` del nodo –ver siguiente punto) asociado al no terminal de la parte izquierda de la producción analizada.

A.2.5 Especificación Semántica

Las producciones pueden especificar al final de ellas código Python representativo de su semántica. Este código ha de estar ubicado entre los pares de caracteres "<#" y "#>". Al codificar esta semántica, se ha de tener en cuenta que Python utiliza los tabuladores como indentadores obligatorios y el salto de línea como separador de instrucciones.

La ejecución de una acción semántica posee el siguiente contexto:

- Todos los símbolos gramaticales del árbol tienen asociado un objeto. Éstos forman parte de una lista denominada `nodes`. El primer elemento (`nodes[0]`) es el objeto asociado al no terminal de la izquierda; el resto representan los nodos de la parte derecha, enumerados de izquierda a derecha.

- Todo nodo del árbol posee un atributo `text` que representa el código reconocido, eliminando los tokens `Skip` y habiendo reconocido automáticamente los `NotSkip`.

- La función global `write` visualiza en la ventana gráfica de la aplicación la cadena de caracteres pasada como parámetro.

- El objeto global `nitrO`, nos brinda todos los servicios de nuestro sistema computacional –ver § A.4.3.

Puesto que todo nodo es un objeto Python, y este lenguaje posee reflectividad estructural, podemos asignarle cualquier atributo dinámicamente. Al no tener comprobación estática de tipos [Cardelli97], podemos crear en tiempo de ejecución nuevos atributos mediante el operador de asignación. Esto hace que la herramienta suponga un mecanismo de codificación de definiciones dirigidas por sintaxis [Aho90].

Una vez que el árbol sintáctico de una aplicación haya sido creado, se ejecutará únicamente el código asociado a la producción del símbolo inicial; ésta deberá encargarse de llamar al resto de las evaluaciones semánticas. La regla semántica de un nodo se ejecuta al pasarle a éste el mensaje `execute`. Por lo tanto, si queremos que se evalúe un nodo, debemos invocar su método `execute`.

Para ver ejemplos prácticos de la descripción de lenguajes, examínense los archivos de extensión `ml` de la distribución del prototipo.

A.2.6 Instrucción Reify

Aunque nuestro sistema sea independiente del lenguaje de programación, su flexibilidad se centra en la utilización de una instrucción que todo lenguaje posee: la instrucción `reify`. El programador de lenguajes sólo debe ubicar el terminal `_REIFY_` en aquella parte de la gramática donde pueda aparecer dicha instrucción.

La utilidad y funcionamiento de esta instrucción serán explicados en § A.3.3.

A.3 Aplicaciones del Sistema

La codificación de una aplicación en nuestro sistema, indistintamente del lenguaje de programación seleccionado, ha de seguir la siguiente gramática EBNF:

A.3.1 Gramática de Aplicaciones

```
<startApp> ::= APPLICATION = STRING
LANGUAGE = <langSpec> <langAlt> APPCODE
<langSpec> ::= STRING
| <startLang>
<langAlt> ::= + CODE
| λ
```

El identificador único de la aplicación es la cadena de caracteres entre comillas dobles que se asigna a la palabra reservada `Application`. La parte final de un archivo de aplicación siempre es el código de ésta (`APPCODE`), siguiendo la gramática del lenguaje utilizado.

A la hora de implementar una aplicación, o un módulo de aplicación, dentro de nuestro sistema, debemos tener en cuenta que el lenguaje a utilizar ha de haber sido especificado de alguna de las tres formas mencionadas en A.2.

Si optamos por ubicar la especificación del lenguaje de programación en un archivo de extensión `ml`, éste deberá llamarse igual que el identificador del lenguaje y será asignado a la palabra reservada Language. El sistema buscará su especificación en el directorio del sistema, empleando el nombre de archivo especificado.

A.3.2 Aplicaciones Autosuficientes

El segundo modo de especificar el lenguaje de programación a utilizar por una aplicación es incluyéndolo en la propia aplicación. Antes su codificación, puede especificarse el lenguaje a utilizar siguiendo la gramática descrita en A.2.1. Una vez descrito éste, la aplicación se codificará e interpretará en base a esta descripción.

La oportunidad de crear aplicaciones que puedan describir su propia sintaxis y semántica, hacen que éstas sean autosuficientes y directamente ejecutables. En cualquier plataforma en la que nuestro sistema esté instalado, este tipo de aplicaciones puede ejecutarse e interactuar con el resto de programas existentes en el sistema. Un ejemplo práctico de su utilización, haciendo uso de un paquete de distribución, es el desarrollo de un sistema de agentes móviles [Hohl96] independientes del lenguaje; las aplicaciones viajan por la red y se ejecutan en cualquier máquina con el lenguaje que ellas deseen, sin necesidad de que éste esté instalado en la plataforma de ejecución.

A.3.3 Reflectividad No Restrictiva

Cuando desarrollamos la especificación de un lenguaje para nuestro sistema, definimos sus aspectos léxicos, sintácticos y semánticos. Una vez descritos éstos, una aplicación se codifica, valida y ejecuta en base a esta especificación que se mantiene invariable a lo largo de su ciclo de vida. Uno de los objetivos principales de este prototipo es obtener un modo de flexibilizar la especificación de un lenguaje, desde sus propias aplicaciones.

Como mostrábamos en A.2.6, todo lenguaje de nuestro sistema dispone de una instrucción Reify con tan solo ubicar el terminal `_REIFY_` en aquella posición de la gramática donde pueda aparecer dicha sentencia. Esta instrucción está constituida por la palabra reservada `reify`, seguida de código Python entre las parejas de caracteres "<#" y "#>".

El código asociado a una instrucción `reify` será evaluado en el espacio computacional del intérprete en lugar ejecutarse en el contexto de la aplicación. Se produce un salto real en la torre de teórica de intérpretes propuesta por Smith [Smith82]. El resultado es que, en la codificación de aplicaciones, es posible especificar, aumentar o modificar las características del lenguaje de programación, como si nos encontrásemos diseñando éste.

El ejecutar un código en el contexto computacional de su intérprete nos permite:

- Conocer el estado del sistema (objetos, clases, variables. .): introspección.
- Acceder y modificar la estructura de sus objetos: reflectividad estructural.
- Modificar y aumentar la semántica de su lenguaje de programación: reflectividad computacional o de comportamiento.
- Modificar la sintaxis del lenguaje por la propia aplicación: reflectividad de lenguaje.

Un breve ejemplo de las tres primeras características se muestra en el archivo "`musimApp1.na`", entregado en el directorio de pruebas de la distribución del prototipo.

A.3.4 Reflectividad de Lenguaje

La tercera y última forma de especificar un lenguaje de programación en nuestro prototipo es mediante la modificación de un lenguaje existente, previamente definido por alguno de los dos mecanismos ya mencionados. Una vez identificado el lenguaje a utilizar, utilizando el lexema "+", el programador puede ubicar código Python a ejecutar en el contexto de intérprete, de igual que la instrucción `reify` descrita en el punto anterior.

El código descrito se evaluará antes de la ejecución de la aplicación, significando una personalización del lenguaje de programación para la ejecución de una determinada aplicación: el lenguaje no se modifica para todo el sistema, sino que es amoldado a la aplicación concreta.

El modo en el que este código actúa para modificar la especificación de un lenguaje es accediendo al conjunto de reglas que describen el lenguaje y, por medio de la utilización de reflectividad estructural, modificar éstas para obtener la personalización del lenguaje. Un ejemplo de esta posibilidad está codificado en el archivo "`printApp.na`" del prototipo distribuido.

A.4 Interfaz Gráfico

Para facilitar la programación de lenguajes y aplicaciones en nuestro sistema, hemos desarrollado un pequeño interfaz gráfico para nuestro prototipo. Este inter-

faz está construido sobre el estándar TK [Lundh99] independiente de la plataforma a utilizar.

A.4.1 Intérprete de Comandos

La ventana principal del sistema tiene el siguiente aspecto:

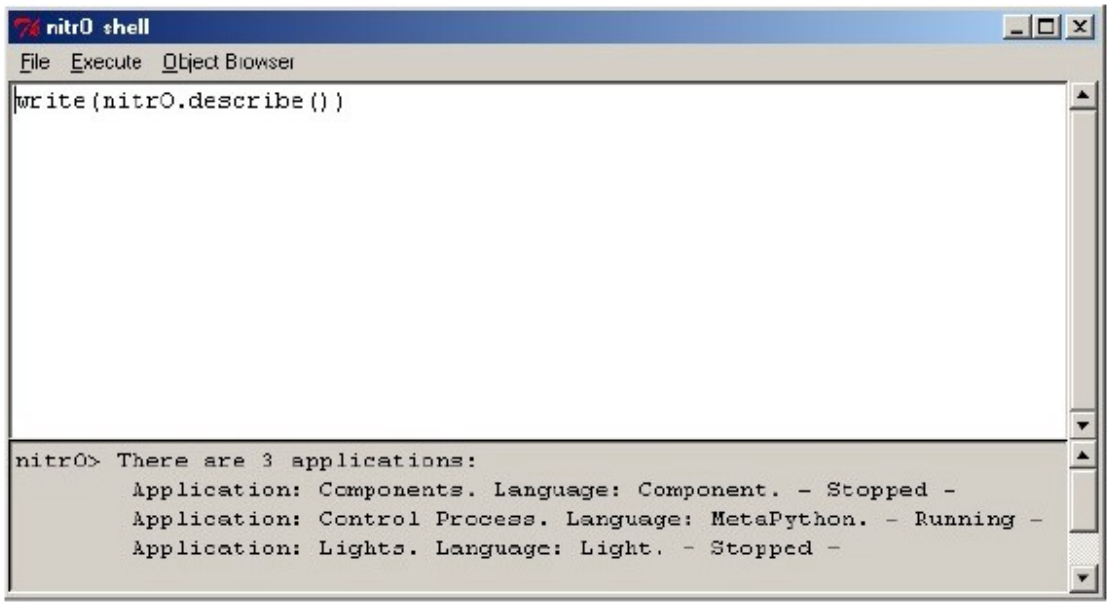


Figura 16.1. Ventana principal del prototipo.

La ventana está dividida en tres partes:

- Menú del sistema.
- Editor de código del intérprete de comandos.
- Ventana de salida, en la que se visualiza el resultado de ejecutar los comandos especificados en el editor.

El usuario del sistema describe los comandos deseados en el editor y evalúa éstos seleccionando la opción `Execute` del menú del sistema. La principal cuestión que puede preguntarse el usuario es ¿qué comandos poseo para acceder al sistema?

El código susceptible de ser ejecutado es cualquier programa Python; éste será evaluado en el contexto de ejecución de nuestro sistema. Además, este contexto tendrá dos elementos añadidos para acceder al sistema:

1. La función `write`, que recibe cualquier parámetro y lo muestra en la ventana de salida. Será utilizado cuando deseemos conocer algún valor. Por ejemplo, en la Figura 16.1 se muestra la cadena de caracteres que describe el sistema.

2. El objeto nitrO, que nos da acceso a todos los elementos del sistema. Ofrece un conjunto de objetos (lenguajes y aplicaciones) así como un conjunto de métodos para trabajar sobre éstos. El conjunto de objetos y métodos existentes podrá consultarse dinámicamente, gracias al carácter introspectivo del sistema, mediante el *Object Browser* (§ B.4.3).

A.4.2 Archivos Empleados

En la utilización nuestro sistema podemos diferenciar tres tipos de archivos:

- Archivos de especificación de lenguajes de programación codificados mediante nuestro metalenguaje. Estos archivos poseen la extensión `ml`, siguen el metalenguaje descrito en § A.2.1, y deben ubicarse en el directorio del sistema.

- Archivos de aplicación. Poseen la extensión `na` (*nitrO application*) y describen aplicaciones del sistema en un determinado lenguaje de programación.

- Archivos *script* o comandos. La extensión `ns` (nitrO script) identifica una secuencia de sentencias a ejecutar por el intérprete de comandos del sistema.

El tratamiento del último tipo de archivos se lleva a cabo mediante el menú de archivo (*File*) de nuestro prototipo.

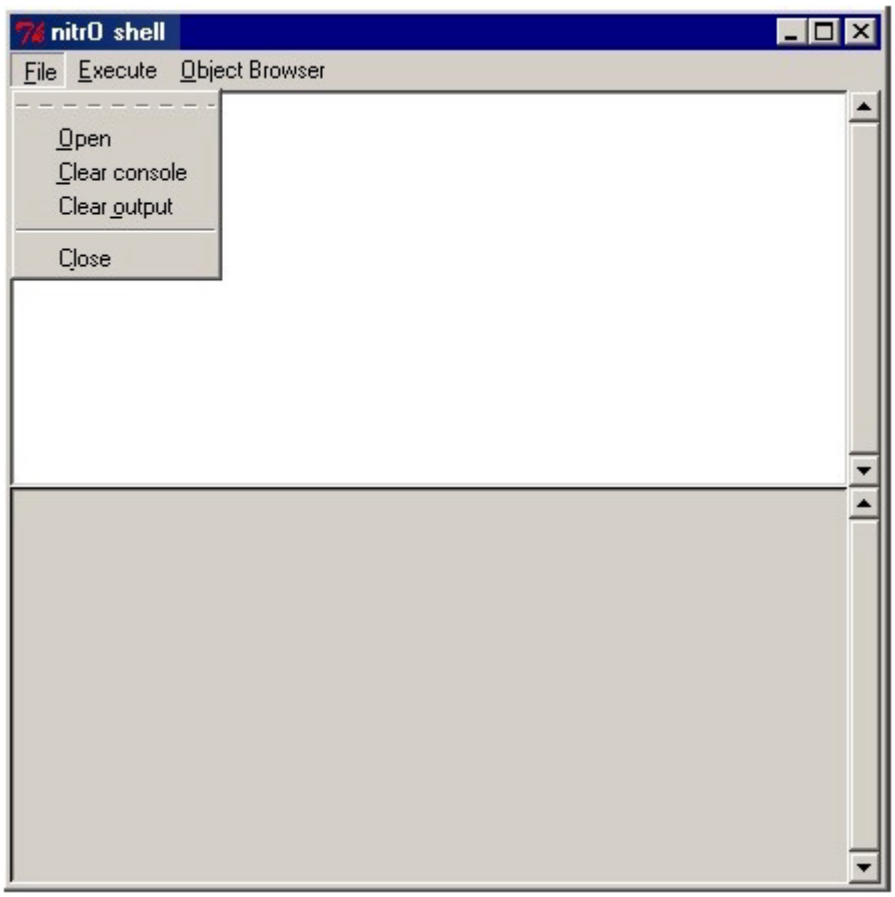


Figura 16.2. Menú archivo del prototipo.

Mediante este menú, pueden cargarse archivos de *script*, así como limpiar el editor y la ventana de salida y finalizar la ejecución del sistema.

A.4.3 Introspección del Sistema

Todo el acceso al sistema se obtiene a través del objeto `nitrO`, que ofrece un conjunto de atributos y métodos descriptores del sistema. Sin embargo, para utilizar éstos, debemos conocer dinámicamente su existencia y funcionalidad. Para ello tenemos dos herramientas en el sistema:

- Métodos `describe`: La mayor parte de los objetos del sistema implementan un método describe que nos muestra su descripción. Pasándoles este mensaje y escribiendo en la consola el resultado de su invocación –mediante la función `write`– obtendremos información dinámica de su estado.

Un ejemplo de esta utilización se muestra en la Figura 16.1.

- *Object Browser*: Esta última opción del menú nos muestra el conjunto de todos los atributos y métodos del objeto nitrO, y por lo tanto de todo el sistema. Haciendo uso de esta herramienta, el programador podrá conocer el conjunto de aplicaciones y lenguajes existentes, su estructura y sus mensajes, y podrá programar y modificar el sistema en función de la información consultada.

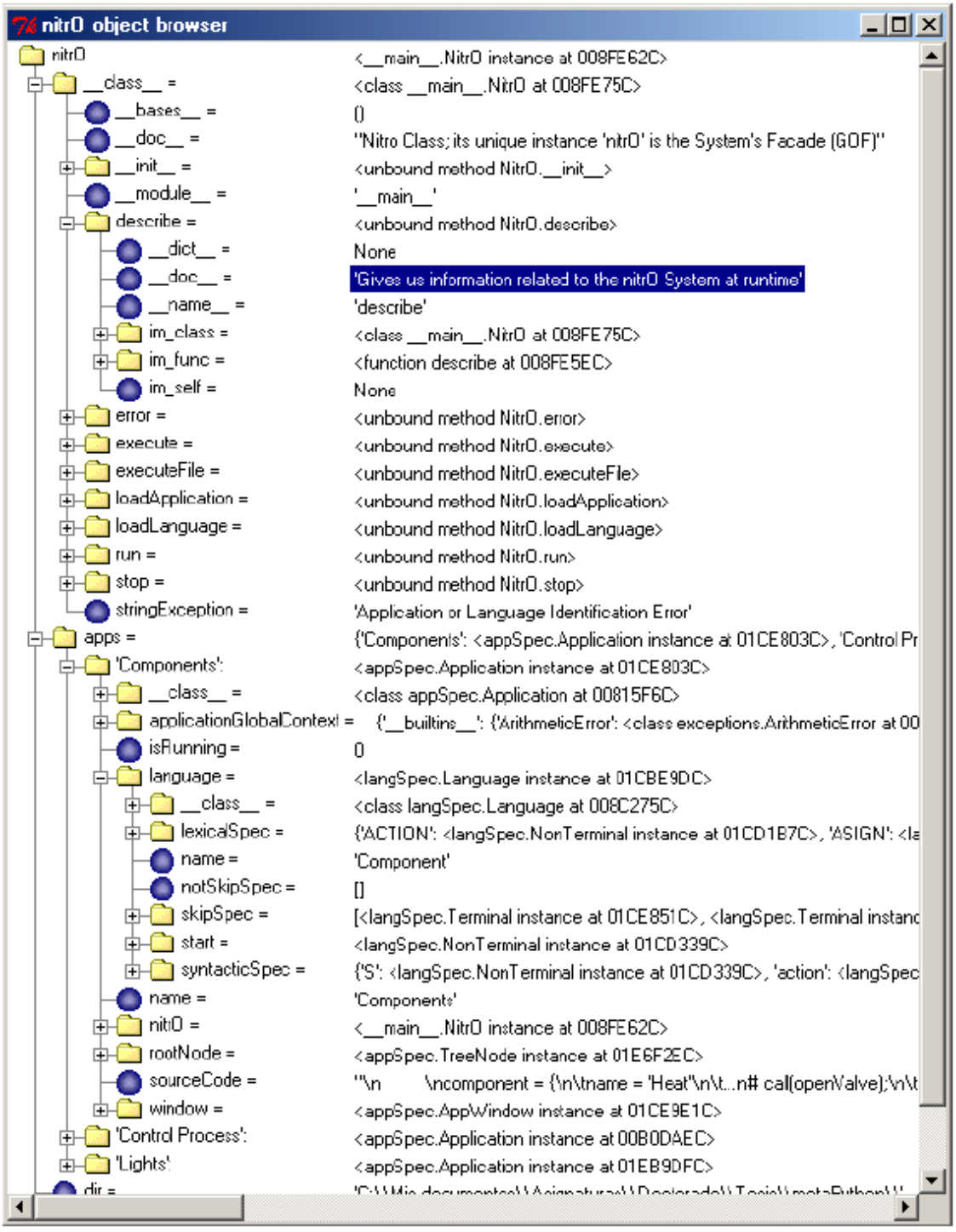


Figura 16.3. Estado del sistema en tiempo de ejecución, mostrado por el *Object Browser*.

En la Figura 16.3 se muestra la información dinámica del sistema. A modo de ejemplo, comentaremos parte de ésta:

- Accediendo al atributo `__class__` de nitrO, obtenemos todos los métodos de éste (`execute`, `executeFile`…) así como la descripción de la clase y cada uno de los mensajes, consultando sus atributos. `__doc__`
- El atributo `apps` es una lista de las aplicaciones existentes en el sistema (`Component`, `ControlProcess` y `Lights`).

- Cada aplicación posee un conjunto de atributos y métodos (su clase) que nos ofrecen información y funcionalidad de ésta. Uno de estos atributos es siempre la especificación de su lenguaje de programación mediante una estructura de objetos –atributo `language`.

- Por cada lenguaje de programación tenemos un conjunto de reglas léxicas y sintácticas libres de contexto (`lexicalSpec` y `syntacticSpec`), así como los componentes léxicos a descartar y a reconocer automáticamente (`skipSpec` y `notSkipSpec`).

Como se aprecia en este ejemplo, la información del sistema es elevada y compleja y, por tanto, una herramienta introspectiva como el *Object Browser* facilita el trabajo al programador.

A.4.4 Ejecución de Aplicaciones

Cada vez que se ejecuta una aplicación en nuestro sistema, se crea una ventana gráfica para ésta. La utilización de la función `write` por esta aplicación, supone la visualización de la información pasada como parámetro en su ventana asociada.

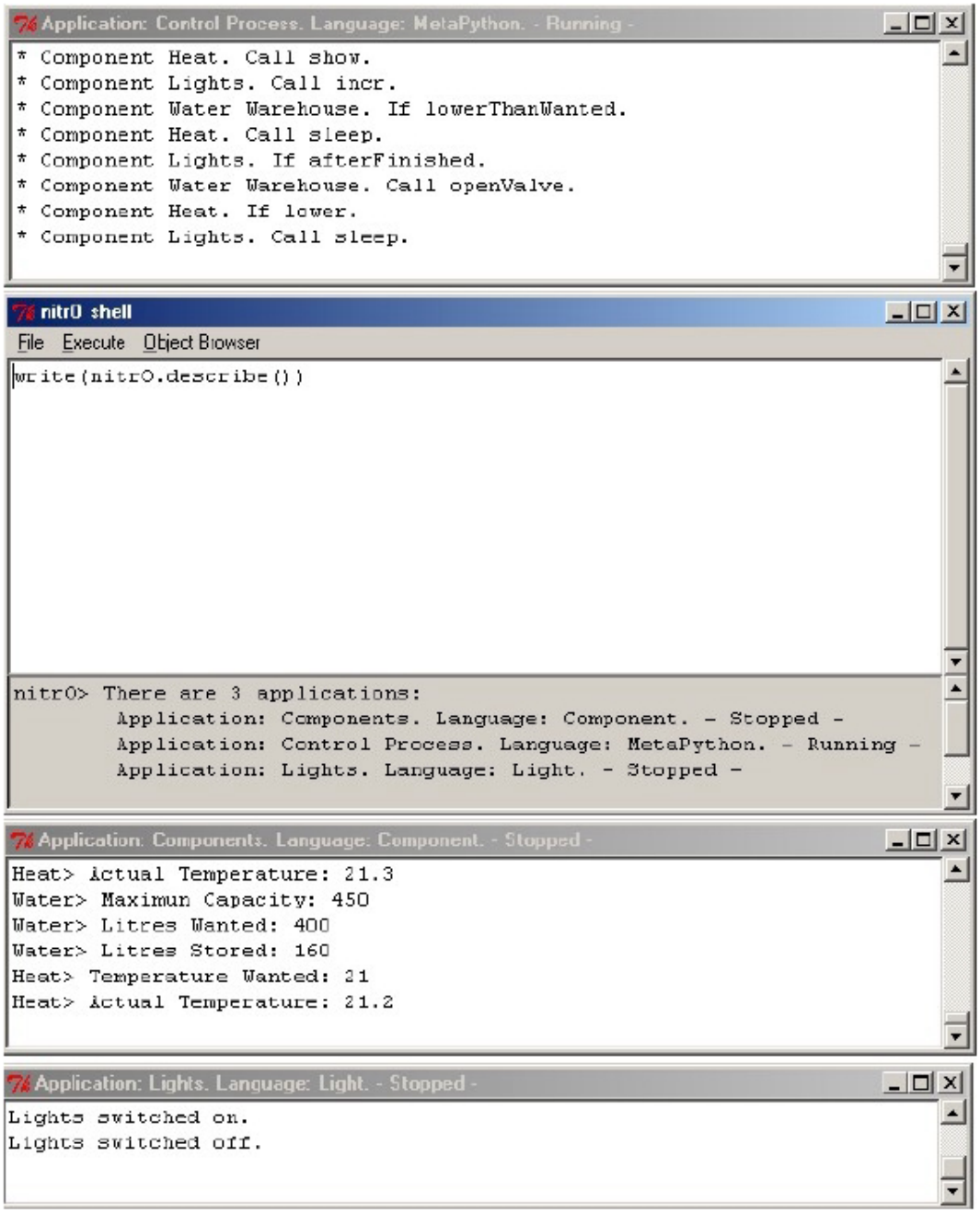


Figura 16.4. Ejecución de aplicaciones en el sistema.

En la figura anterior se muestra cómo el sistema está ejecutando 3 aplicaciones, cada una con su propia ventana de visualización. Una aplicación puede cerrarse cuando esté parada (al final de su título aparezca *Stopped*). El hecho de cerrar su ventana supone la eliminación de la aplicación asociada dentro del sistema.

Si deseamos finalizar una aplicación en ejecución, deberemos, desde el editor de comandos, enviar un mensaje *stop* al objeto `nitrO`, pasándole como parámetro el identificador de la aplicación. El título de ésta mostrará cuándo esté tratando de finalizar (*trying to stop*) y finalmente su estado de parada (*stopped*).

Cerrar la ventana principal del sistema supone cerrar el conjunto de aplicaciones existentes.

Apéndice B
DESCRIPCIÓN DEL LENGUAJE JAVA-- IMPLEMENTADO

B.1 Introducción al Lenguaje

Java--, como su nombre indica, está basado en una versión simplificada del lenguaje Java, con algunos cambios –el principal, que no existen tipos primitivos, sino que todo son objetos.

Algunas de las principales carencias de este lenguaje son:

- No soporta interfaces.
- No se pueden crear clases anidadas.
- El código del programa no se puede repartir entre varios archivos, y tampoco se pueden crear paquetes.
- No se soporta el control de acceso a los miembros de las clases –todos los atributos y métodos son públicos.
- No existen métodos ni atributos estáticos.

En las próximas secciones se mostrarán algunos ejemplos de uso del lenguaje, explicando sus capacidades y los puntos en los que difiere de Java. Se advierte que se supondrán conocimientos del lenguaje Java o alguno similar (C++, por ejemplo).

B.1.1 ¡Hola Mundo!

El siguiente ejemplo muestra la versión Java-- del clásico programa ¡Hola mundo!:

```
1  Application = "JavaApp"
2  Language = "Java"
3
4  {
5          Console cons = new Console();
6          cons.println('¡Hola mundo!');
7  }
```

Procederemos a explicar el programa paso a paso:

- En la línea 1 se le da nombre a la aplicación. Será válido cualquier nombre siempre que sea único dentro del conjunto de aplicaciones que se estén ejecutando en el sistema (de lo contrario se sobrescribiría la aplicación homónima al ejecutarlo).
- La línea 2 le indica al sistema el nombre del lenguaje empleado, que debe ser "Java".
- El bloque de código que aparece en las líneas 4–7 es el cuerpo del programa principal, nuestro punto de entrada.
- En la línea 5 se emplea el operador `new` para crear una nueva instancia de la clase `Console` y se asigna a la referencia llamada `cons`.
- Por último, en la línea 6 escribe la cadena "¡Hola mundo!" por la salida estándar.

Como se puede comprobar, la mayor diferencia respecto a Java se produce por no soportar métodos estáticos, lo que impide usar una clase con un método `main()` como en el lenguaje original. En su lugar, al final de todo programa Java--aparece un bloque de código situado fuera de cualquier clase, y ese bloque será nuestro programa principal.

B.1.2 Control de Flujo

Se han incluido las sentencias `if-then-else`, `for` y `while` para controlar el flujo de ejecución del programa. Un ejemplo de su uso aparece en el siguiente ejemplo, junto con algunas operaciones con enteros y cadenas.

```
1  Application = "JavaApp"
2  Language = "Java"
3
4  {
5          Console cons = new Console();
6          Integer a = 1, b = 2;
7
8          if(a < b)
9                  cons.println('Es menor');
10         else
11                 cons.println('No es menor');
12
13         for(Integer i = 0; i < 10; ++i)
14                 cons.println('i vale ' + i.toString());
15
16         Integer x = 3;
17         while(x > 0) {
18                 cons.println(x.toString());
19                 --x;
20         }
21 }
```

Se puede observar que no difiere mucho de un programa Java equivalente, salvo en detalles menores como el empleo de la comilla simple para las cadenas de texto en lugar de la comilla doble. Sin embargo, hay una diferencia importante, y es la interpretación de los valores lógicos. En Java existe un tipo básico `bool` para representar el resultado de las operaciones lógicas. Sin embargo, Java-- carece de tipos primitivos, por lo que hubo que buscar otra forma de representar esos resultados. La elección fue emplear el mismo sistema que en Lisp: una referencia nula tiene valor lógico falso, mientras que una no nula tiene valor lógico verdadero.

B.1.3 Operadores Internos y Externos

En el lenguaje Java-- existen dos tipos de operadores, que hemos llamado internos y externos. Los primeros reciben su nombre por estar implementados como métodos de las instancias (por lo que podrían ser modificables, aunque actualmente el lenguaje no lo permite—sin embargo la estructura necesaria ya está implementada). Los operadores externos, en cambio, trabajan sobre las referencias, y por tanto son ajenos a las instancias. La siguiente tabla muestra qué operadores pertenecen a cada categoría.

	Internos	Externos
Aritméticos	+, -, *, /, %, ++, –	
Lógicos	<, <=, >, >=, ==, !=	&&, ||, !, is, ?:

Como se puede observar en la tabla, todos los operadores aritméticos son internos (como es lógico, puesto que necesitan el valor asociado a la instancia), mientras que los lógicos están repartidos entre aquellos que emplean el valor (y por tanto son internos) y los que se limitan a consultar si sus argumentos son o no nulos, o si dos referencias apuntan a la misma instancia (operador is), por lo que son externos.

Tal y como muestra la tabla, todas las operaciones que tienen sentido con valores lógicos se corresponden con operadores externos, y por ese motivo la clase `Bool` no tiene ningún otro operador definido—simplemente no son necesarios.

B.1.4 Clases y Métodos

La siguiente figura muestra un ejemplo en el que se introducen las clases y los métodos. Nuevamente se comprueba que no hay grandes diferencias con lo que sería un programa Java equivalente. Este sencillo código también permite mostrar el funcionamiento de la herencia, los métodos virtuales y el acceso a los atributos. Se comprueba cómo un método accede únicamente a los atributos definidos en su clase o en sus clases base, aunque en clases derivadas se oculten con otros. También se ve cómo empleando `super` se puede acceder a la clase base.

```
1  Application = "JavaApp"
2  Language = "Java"
3
4  class A {
5          Integer i = 10;
6          Integer j = 20;
7
8          void metodo(Console cons) {
9                  cons.println('Estamos en la clase A');
10                 cons.println('func devuelve ' + func().toString());
11         }
12
13         Integer func() {
14                 return i + j;
15         }
16 }
17
18 class B extends A {
19         Integer i = 30;
20         Integer j = 40;
21
22         void metodo(Console cons) {
23                 cons.println('Estamos en la clase B');
24                 cons.println('func devuelve ' + func().toString());
25                 cons.println('Pero en el contexto de A devuelve '
26                         + super.func().toString());
27         }
28 }
29
30 {
31         Console cons = new Console();
32
33         A a = new A();
34         a.metodo(cons);
35
36         B b = new B();
37         b.metodo(cons);
38 }
```

B.1.5 Constructores

El ejemplo de la siguiente figura muestra cómo se define un constructor en el lenguaje Java--, como un método con el mismo nombre de la clase y sin tipo de retorno. También se muestra la sentencia `super` (no confundir con la expresión), que sirve para invocar al constructor de la clase padre y que, de aparecer, debe ser la primera sentencia en el cuerpo del constructor.

```
Application = "JavaApp"
Language = "Java"

class A {
        Integer i;

        A(Integer i) {
                this.i = i;
        }

        void metodo(Console cons) {
                cons.println('i = ' + i.toString());
        }
}

class B extends A {
        B() {
                super(5);
        }
}

{
        Console cons = new Console();

        A a = new A(3);
        a.metodo(cons);

        B b = new B();
        b.metodo(cons);
}
```

B.1.6 Sobrecarga de Métodos

El lenguaje Java-- también soporta la sobrecarga de métodos, tal y como se muestra en la figura que se muestra a continuación. Por supuesto, los constructores también pueden sobrecargarse, al no ser más que métodos con ciertos "privilegios" especiales.

```
Application = "JavaApp"
Language = "Java"

class A {
        void metodo(Console cons, Integer i) {
                cons.println('i = ' + i.toString());
        }

        void metodo(Console cons, String s) {
                cons.println('s = ' + s);
        }
}

{
        Console cons = new Console();

        A a = new A();
        a.metodo(cons, 1);
        a.metodo(cons, 'hola');
}
```

B.1.7 Reflectividad

La siguiente figura muestra cómo empleando la sentencia `reify` se puede emplear código Python dentro de un programa Java--, y acceder así a toda la estruc-

tura interna del sistema. En el ejemplo también se puede observar la forma de acceder a una variable local desde el código Python.

```
Application = "JavaApp"
Language = "Java"

class A {
        void metodo(Console cons) {
                reify
<#
print 'Hola desde Python'
cons = theInterpreter.getSymbolTable().getVar('cons').getInstance()
print 'La instancia de la consola:', cons
#>
        }
}

{
        Console cons = new Console();

        A a = new A();
        a.metodo();
}
```

B.2 Referencia

B.2.1 Operadores

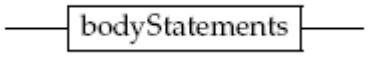
La siguiente tabla muestra los operadores existentes en el lenguaje, así como su asociatividad. Se encuentran ordenados de mayor a menor prioridad. Como se puede observar no se han implementado los operadores de asignación ampliados existentes en el lenguaje Java (`+=`, `*=`, etc.). Sin embargo añadirlos no sería problemático, pues podrían convertirse internamente en los operadores normales (ya que `a += b` es equivalente a `a = a + b`, por ejemplo).

Símbolo	Nombre o significado	Asociatividad
()	Llamada a función	Izquierda a derecha
.	Miembro de una instancia	
++	Incremento	Derecha a izquierda
–	Decremento	
!	*NO* lógico	Derecha a izquierda
+	Más unario	
-	Menos unario	
(*tipo*)	Ahormado de tipos	
*	Multiplicación	Izquierda a derecha
/	División	
%	Resto	
+	Suma	Izquierda a derecha
-	Resta	
==	Igualdad	Izquierda a derecha
!=	Desigualdad	
is	Identidad	
<	Menor que	Izquierda a derecha
<=	Menor o igual que	
>	Mayor que	
>=	Mayor o igual que	
&&	*Y* lógico	Izquierda a derecha
||	*O* lógico	Izquierda a derecha
?:	Condicional	Derecha a izquierda
=	Asignación	Derecha a izquierda
,	Coma	Izquierda a derecha

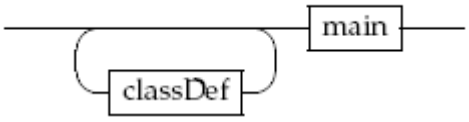
Figura 16.5. Operadores del lenguaje Java--.

B.2.2 Diagramas sintácticos

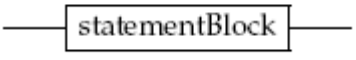
S



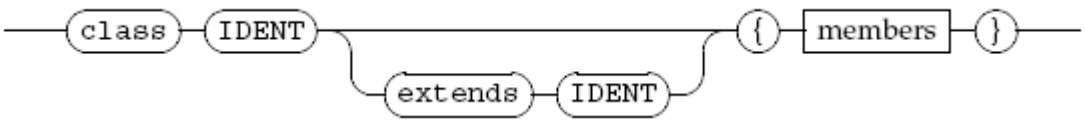
bodyStatements



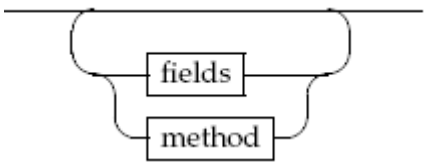
main



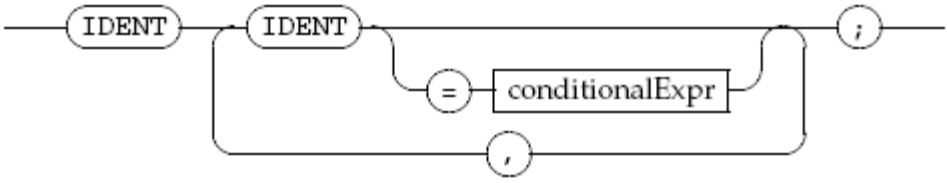
classdef



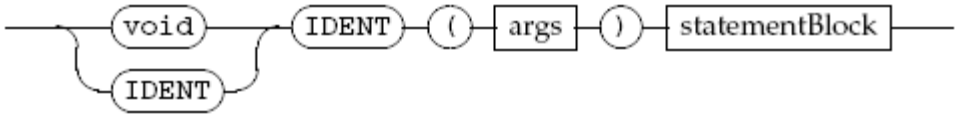
members



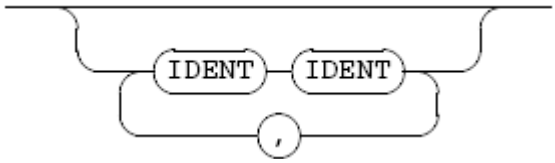
fields



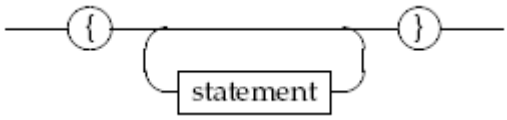
method



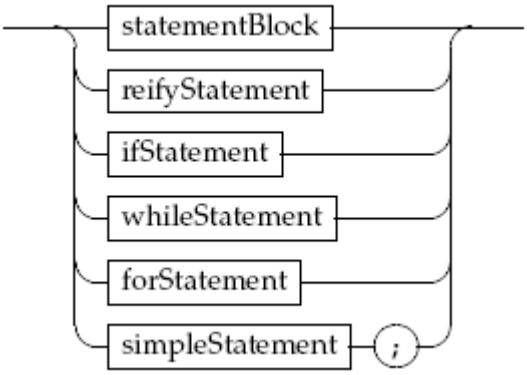
args



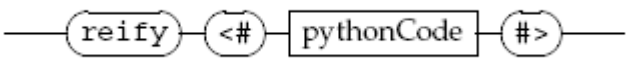
statementBlock



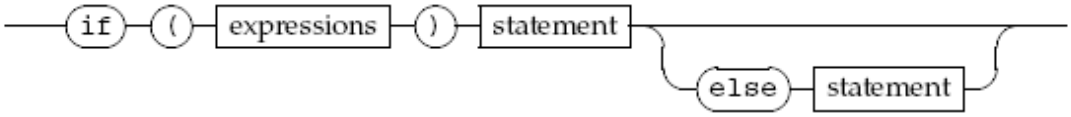
statement



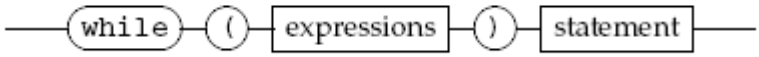
reifyStatement



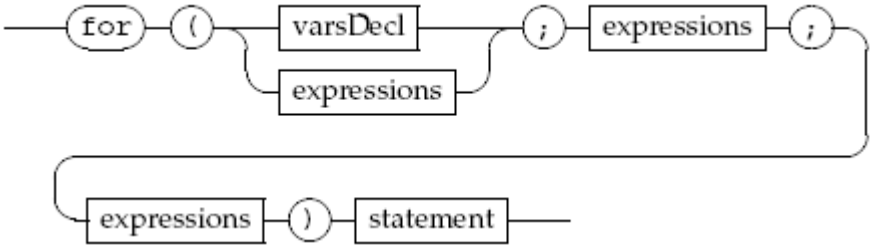
ifStatement



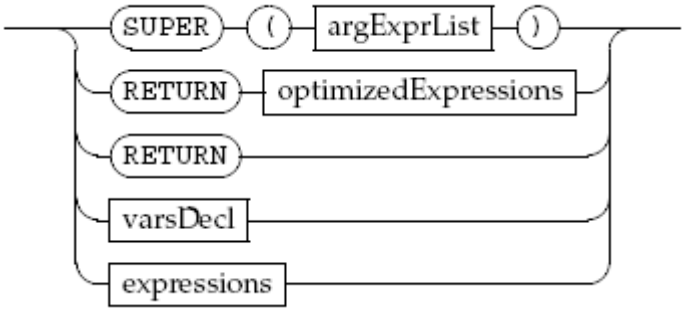
whileStatement



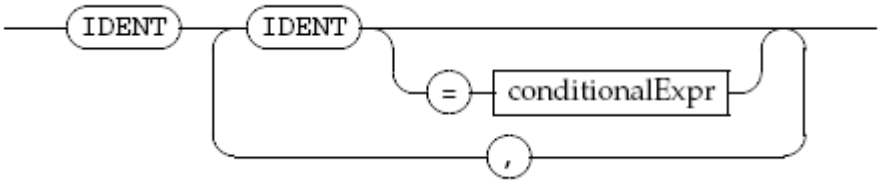
forStatement



simpleStatement

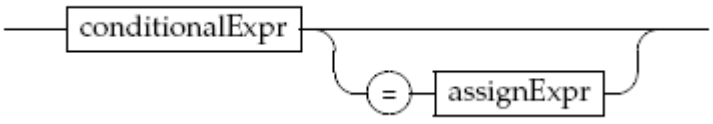


varsDecl

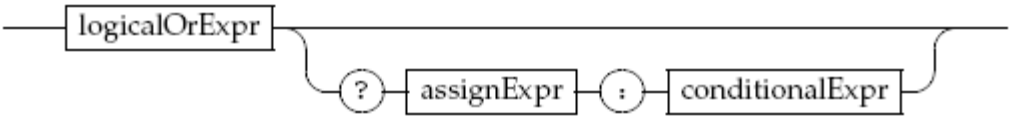


expressions

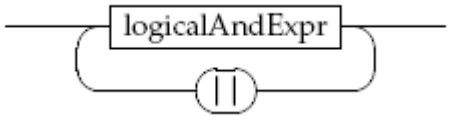
assignExpression



conditionalExpression

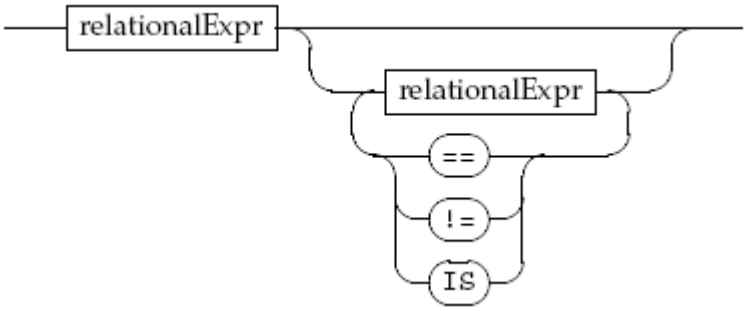


LogicalOrExpre

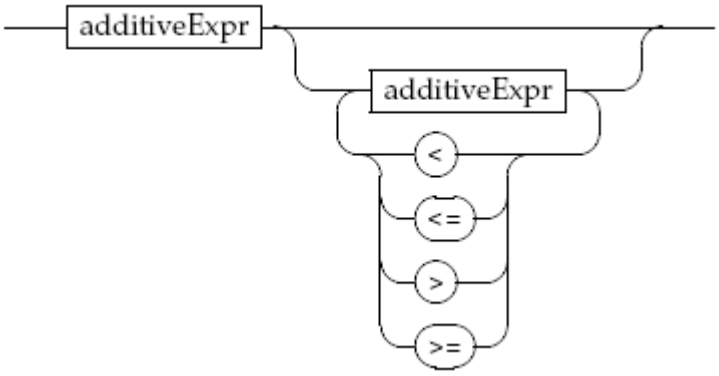


LogicalAndExpr

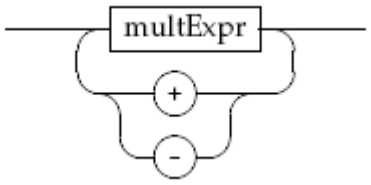
EqualityExpr



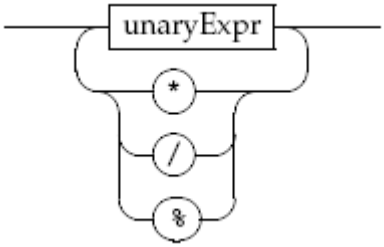
relationalExpr



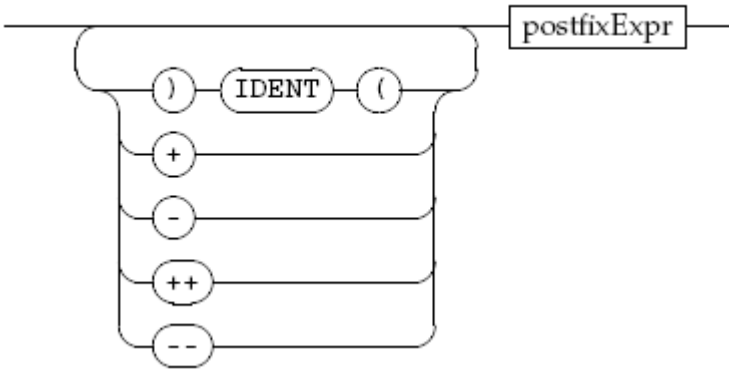
additiveEprs



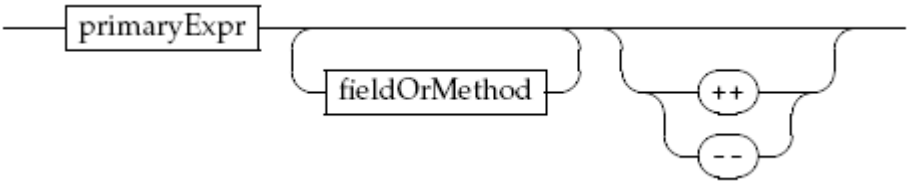
nullExpr



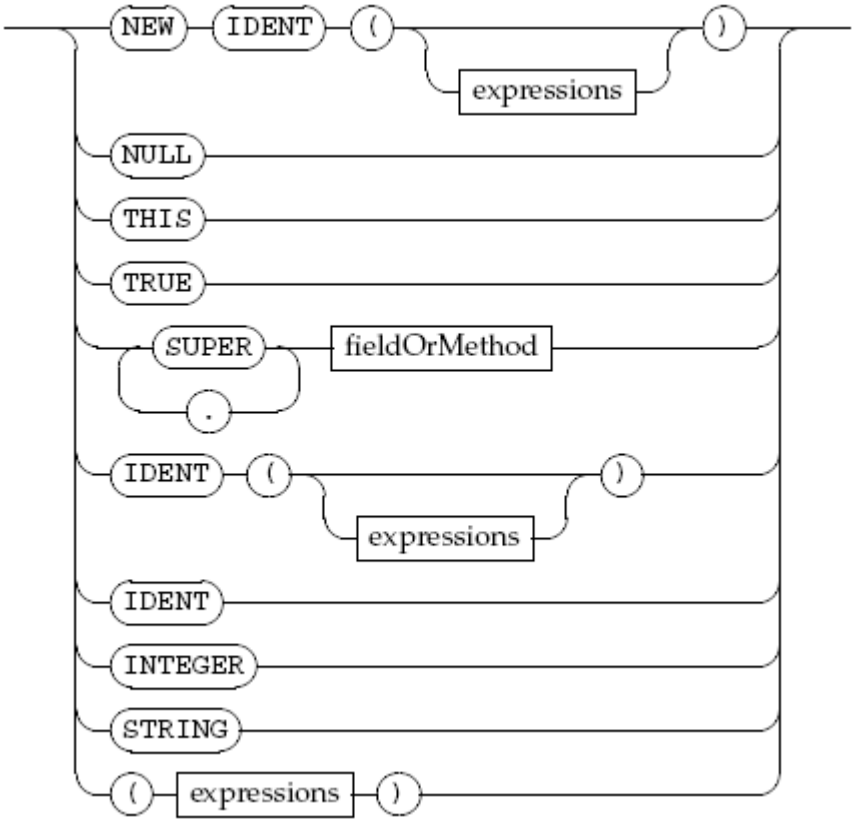
unaryExpr



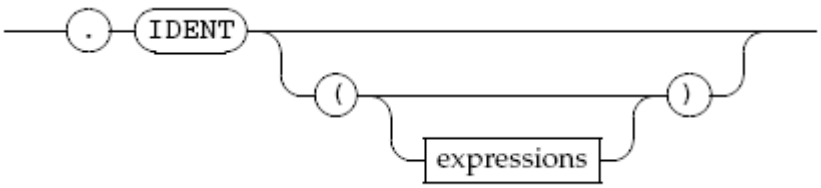
postfixExpr



primaryExpr



fieldOrMethod



B.3 API del Lenguaje Java Implementado

B.3.1 Diagrama de Clases

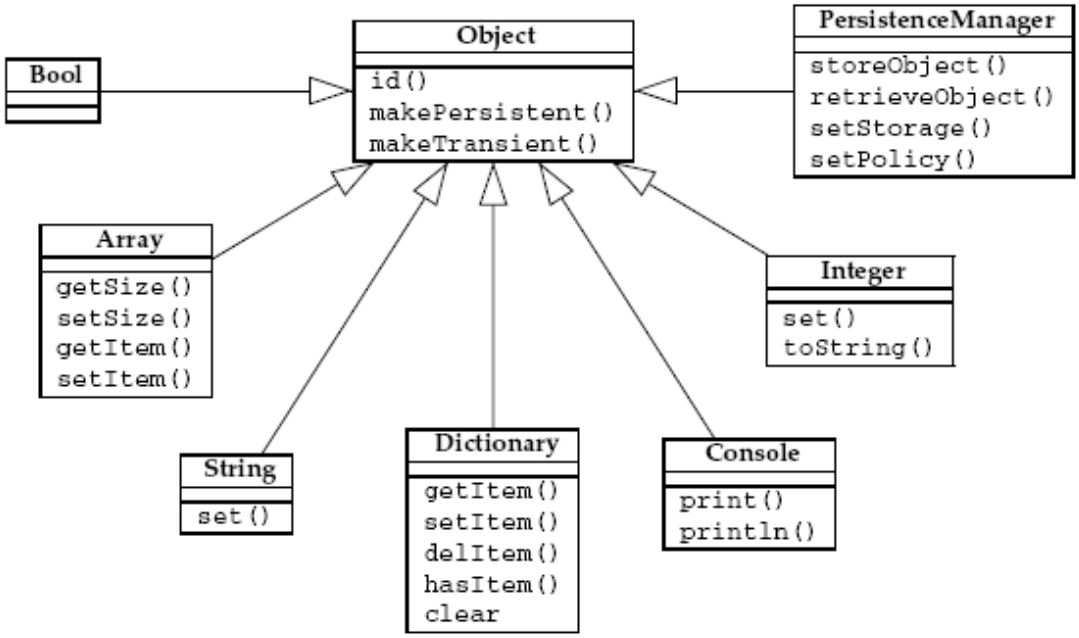


Figura 16.6. Diagrama de clases del API del lenguaje.

En la Figura 16.6 se muestra la jerarquía de las clases del API del lenguaje Java--. Se puede observar en particular que al derivar todas las clases de `Object`, tanto las primitivas como todas las clases definidas por el usuario, automáticamente toda instancia se podrá hacer persistente sin más que invocar un método de la misma. Además, para los casos en que simplemente se quiere guardar un *snapshot* de un objeto, se ha creado una clase `PersistenceManager` que expone a los programas de usuario la funcionalidad necesaria para cargar y almacenar objetos de forma puntual.

B.3.2 Descripción de las Clases

B.3.2.1 Clase Object

Clase base de todos los objetos del lenguaje Java

Método id()

- **Descripción**: Devuelve el GUID del objeto.
- Prototipo: `id()`
- **Argumentos**: Ninguno.
- **Valor de retorno**: Una cadena con el GUID del objeto.

Método makePersistent()

- **Descripción**: Hace que el objeto y sus miembros sean persistentes.
- **Prototipo**: `makePersistent()`
- **Argumentos**: Ninguno.
- Valor de retorno: Ninguno.

Método makeTransient()

- **Descripción**: Hace que el objeto y sus miembros dejen de ser persistentes.
- **Prototipo**: `makeTransient()`
- **Argumentos**: Ninguno.
- Valor de retorno: Ninguno.

B.3.2.2 Clase Bool

Clase de la única instancia que se utiliza para representar el valor lógico verdadero en los resultados de todas las operaciones y métodos cuyo resultado sea de tipo booleano.

B.3.2.3 Clase Integer

Un entero.

Operaciones soportadas

- Con parámetro `Integer`: `+, -, *, /, %, <, <=, >, >=, ==, !=`

Método set()

- **Descripción**: Cambia el valor del entero.
- **Prototipo**: `set(Integer newValue)`
- Argumentos:

`newValue`: Entero del que se copiará el nuevo valor.

- Valor de retorno: Ninguno.

Método toString()

- **Descripción**: Crea una representación textual del entero.
- **Prototipo**: `toString()`
- **Argumentos**: Ninguno.
- **Valor de retorno:** Una cadena con la representación textual del entero.

B.3.2.4 Clase String

Una cadena de texto.

Operaciones soportadas

- Con parámetro String: +

Método set()

Descripción: Cambia el valor del entero.

Prototipo: `set(Integer newValue)`

Argumentos: `newValue`: Entero del que se copiará el nuevo valor.

Valor de retorno: Ninguno.

B.3.2.5 Clase Console

Interfaz básico para escribir texto por la salida estándar (que por defecto se redirige a la ventana de la aplicación).

Método print()

- **Descripción**: Escribe una cadena por la salida estándar.
- **Prototipo**: `print(String str)`
- Argumentos:

`str`: Cadena a escribir.

- Valor de retorno: Ninguno.

Método println()

- **Descripción**: Escribe una cadena por la salida estándar, añadiendo un salto de línea al final.
- **Prototipo**: `println(String str)`
- Argumentos:

`str`: Cadena a escribir.

- Valor de retorno: Ninguno.

B.3.2.6 Clase Array

Vector de objetos.

Método getSize()

- **Descripción**: Devuelve el tamaño del *array*.
- **Prototipo**: `getSize()`
- **Argumentos**: Ninguno

- **Valor de retorno**: Un entero con el tamaño del *array*.

Método setSize()

- **Descripción**: Cambia el tamaño del *array*
- **Prototipo**: `setSize(Integer newSize)`
- Argumentos:
- `newSize`: Nuevo tamaño para el *array*.
- Valor de retorno: Ninguno.

Método getItem()

- **Descripción**: Devuelve uno de los objetos del *array*.
- **Prototipo**: `getItem(Integer index)`
- Argumentos:

`index`: Posición del objeto dentro del *array*.

- **Valor de retorno**: El objeto pedido.

Método setItem()

- **Descripción**: Cambia uno de los objetos del *array*.
- **Prototipo**: `setItem(Integer index, Object obj)`
- Argumentos:

`index`: Posición del objeto dentro del *array*.

`obj`: El nuevo objeto.

- Valor de retorno: Ninguno.

B.3.2.7 Clase Dictionary

Un diccionario que asocia cadenas a objetos.

Método getItem()

- **Descripción**: Devuelve uno de los objetos del diccionario.
- **Prototipo**: `getItem(String key)`
- Argumentos:

`key`: El nombre del objeto dentro del diccionario.

- **Valor de retorno**: El objeto pedido, o `null` si no existe un elemento con la clave dada.

Método setItem()

- **Descripción**: Inserta un objeto en el diccionario con un nombre dado, o lo sustituye si ya existía.
- **Prototipo**: `setItem(String key, Object value)`
- Argumentos:

`key`: El nombre del objeto dentro del diccionario.

`value`: El objeto a introducir.

- Valor de retorno: Ninguno.

Método delItem()

- **Descripción**: Elimina una entrada del diccionario.
- **Prototipo**: `delItem(String key)`
- Argumentos:
- `key`: El nombre del objeto dentro del diccionario.
- Valor de retorno: Ninguno.

Método hasItem()

- **Descripción**: Pregunta si el diccionario contiene algún objeto con un cierto nombre.
- **Prototipo**: `hasItem(String key)`
- Argumentos:

`key`: El nombre del objeto dentro del diccionario.

- **Valor de retorno**: Un booleano indicando si el objeto está en el diccionario o no.

Método clear()

- **Descripción**: Vacía el diccionario.
- **Prototipo**: `clear()`
- **Argumentos**: Ninguno.
- Valor de retorno: Ninguno.

B.3.2.8 Clase PersistenceManager

Interfaz básico con el sistema de persistencia.

Método storeObject()

- **Descripción**: Guarda un objeto en el almacenamiento actual.
- **Prototipo**: `storeObject(Object obj)`

- Argumentos:

`obj`: El objeto a guardar.

- **Valor de retorno**: Una cadena con el GUID del objeto almacenado.

Método retrieveObject()

- **Descripción**: Carga un objeto desde el almacenamiento actual.
- **Prototipo**: `retrieveObject(String guid)`
- Argumentos:

`guid`: GUID del objeto a recuperar.

- **Valor de retorno**: El objeto cargado.

B.4 Gramática del Lenguaje Java Implementado

La siguiente gramática es la finalmente implementada para la interpretación del lenguaje Java--. No es idéntica a la presentada en el diseño por las limitaciones de la herramienta, al no permitir ésta partes repetitivas de forma automática, por lo que hubo que modificar la gramática hasta obtener una equivalente y que nitrO aceptase.

```
<S>                 ::= <bodyStatements>
<bodyStatements>    ::= <classDef> <bodyStatements>
                    | <main>
<main>              ::= <statementBlock>
<classDef>          ::= CLASS IDENT <extends> LCURLY <members> RCURLY
<extends>           ::= EXTENDS IDENT
                    λ
<members>           ::= IDENT LPAREN <args> RPAREN <statementBlock>
<members>
                    | VOID IDENT LPAREN <args> RPAREN <statementBlock>
                    <members>
                    | IDENT IDENT LPAREN <args> RPAREN <statementBlock>
                    <members>
                    | IDENT IDENT ASSIGN <conditionalExpr>
<moreFieldDecls>
                    <members>
                    | IDENT IDENT <moreFieldDecls> <members>
                    λ
<args>              ::= IDENT IDENT <moreArgs>
                    λ
<moreArgs>          ::= COMMA IDENT IDENT <moreArgs>
                    λ
<moreFieldDecls>    ::= COMMA IDENT ASSIGN <conditionalExpr>
<moreFieldDecls>
                    | COMMA IDENT <moreFieldDecls>
                    | SEMI
<statements>              ::= <statement> <statements>
                    λ
<statement>         ::= <statementBlock>
                    | _REIFY_
                    | IF LPAREN <optimizedExpressions> RPAREN
<statement>
<elseBlock>
```

```
                    | WHILE LPAREN <optimizedExpressions> RPAREN
<statement>
                    | FOR LPAREN <forInit> SEMI <optimizedExpressions>
SEMI
<optimizedExpressions> RPAREN <statement>
                    | <simpleStatement> SEMI
<elseBlock>         ::= ELSE <statement>
                    λ
<forInit>           ::= IDENT IDENT ASSIGN <assignExpr> <moreVarDecls>
                    | IDENT IDENT <moreVarDecls>
                    | <optimizedExpressions>
                    λ
<statementBlock>    ::= LCURLY <statements> RCURLY
<simpleStatement>   ::= SUPER LPAREN <argExprList> RPAREN
                    | RETURN <optimizedExpressions>
                    | RETURN
                    | IDENT IDENT ASSIGN <assignExpr> <moreVarDecls>
                    | IDENT IDENT <moreVarDecls>
                    | <optimizedExpressions>
                    λ
<moreVarDecls>              ::= COMMA IDENT ASSIGN <assignExpr>
<moreVarDecls>
                    | COMMA IDENT <moreVarDecls>
                    λ
<optimizedExpressions>    ::= <expressions>
<expressions>             ::= <assignExpr> <moreExpressions>
<moreExpressions>   ::= COMMA <assignExpr> <moreExpressions>
                    λ
<assignExpr>              ::= <conditionalExpr> <moreAssignExpr>
<moreAssignExpr>    ::= ASSIGN <assignExpr>
                    λ
<conditionalExpr>   ::= <logicalOrExpr> <moreConditionalExpr>
<moreConditionalExpr>     ::= QUESTION <assignExpr> COLON
<conditionalExpr>
                    λ
<logicalOrExpr>     ::= <logicalAndExpr> <moreLogicalOrExpr>
<moreLogicalOrExpr>       ::= LOR logicalAndExpr
                    λ
<logicalAndExpr>    ::= <equalityExpr> <moreLogicalAndExpr>
<moreLogicalAndExpr>      ::= LAND <equalityExpr>
                    λ
<<equalityExpr>>    ::= <relationalExpr> <moreEqualityExpr>
<moreEqualityExpr>  ::= EQUAL <relationalExpr>
                    | NOTEQUAL <relationalExpr>
                    | IS <relationalExpr>
                    λ
<relationalExpr>    ::= <additiveExpr> <moreRelationalExpr>
<moreRelationalExpr>      ::= LT <additiveExpr>
                    | LTE <additiveExpr>
                    | GT <additiveExpr>
                    | GTE <additiveExpr>
                    λ
<additiveExpr>            ::= <multExpr> <moreAdditiveExpr>
<moreAdditiveExpr>  ::= PLUS <multExpr> <moreAdditiveExpr>
                    | MINUS <multExpr> <moreAdditiveExpr>
                    λ
<multExpr>          ::= <unaryExpr> <moreMultExpr>
<moreMultExpr>            ::= MUL <unaryExpr> <moreMultExpr>
                    | DIV <unaryExpr> <moreMultExpr>
CAPÍTULO 3. GRAMÁTICA DEL LENGUAJE JAVA-- 71
                    | MOD <unaryExpr> <moreMultExpr>
                    λ
<unaryExpr>         ::= LPAREN IDENT RPAREN <unaryExpr>
                    | PLUS <unaryExpr>
                    | MINUS <unaryExpr>
                    | LNOT <unaryExpr>
                    | INC <unaryExpr>
                    | DEC <unaryExpr>
```

```
                    | <postfixExpr>
<postfixExpr>             ::= <primaryExpr> <postfixSuffixes>
<postfixSuffixes>   ::= DOT IDENT LPAREN <argExprList> RPAREN
<postfixSuffixes>
                    | DOT IDENT <postfixSuffixes>
                    | INC
                    | DEC
                    λ
<argExprList>             ::= <expressions>
                    λ
<primaryExpr>             ::= NEW IDENT LPAREN <argExprList> RPAREN
                    | NULL
                    | THIS
                    | TRUE
                    | SUPER DOT <superAccess>
                    | IDENT LPAREN <argExprList> RPAREN
                    | IDENT
                    | INTEGER
                    | STRING
                    | LPAREN <optimizedExpressions> RPAREN
<superAccess>             ::= SUPER DOT <superAccess>
                    | IDENT LPAREN <argExprList> RPAREN
                    | IDENT
```

Apéndice C
REFERENCIAS BIBLIOGRÁFICAS

[Abelson2000] H. Abelson *et. al.* "Scheme. Revised Report on the Algorithmic Language Scheme". R. Kelsey, W. Clinger, and J. Rees Editors. Marzo 2000.

[Aho90] A. V. Aho. "Compiladores: Principios, Técnicas y Herramientas". Addison-Wesley Iberoamericana. 1990.

[Aksit88] M. Aksit, A. Tripathi. "Data Abstraction Mechanism in Sina/ST". OOPSLA'88 Conference Proceedings, ACM SIGPLAN Notices, Vol. 23, nº 11. Noviembre de 1988.

[Aksit91] M. Aksit, J. W. Dijkstra, A. Tripathi. "Atomic Delegation: Object-Oriented Transactions". IEEE Software, Vol. 8, nº 2. Marzo de 1991.

[Aksit92] M. Aksit, L. Bergmans, S. Vural. "An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach". ECOOP'92, LNCS 615, Springer-Verlag. 1992.

[Aksit93] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa. "Abstracting Object-Interactions Using Composition-Filters". Object-based Distributed Processing". LNCS 791, Springer-Verlag. 1993.

[Aksit94] M. Aksit, J. Bosch, W. v.d. Sterren, L. Bergmans. "Real-Time Specification Inheritance Anomalies and Real-Time Filters". Proceedings of ECOOP'94, LNCS 821, Springer-Verlag. 1994.

[Alur2001] Alur, D., Crupi, J. and Malks, D. "Core J2EE Patterns – Best Practices and Design Strategies". Sun Microsystems Press, 2001, ISBN 0-13-064884-1.

[Alva2003] María E. Alva O., Ana Belén Martínez Prieto, Juan Manuel Cueva Lovelle, T. Hernán Sagástegui Ch., Benjamín López: Comparison of Methods and Existing Tools for the Measurement of Usability in the Web. International Conference in Web Engineering, Springer Verlag, Lecture Notes in Computer Science 2722. Junio de 2003.

[Álvarez96] Darío Álvarez Gutiérrez, Juan Manuel Cueva Lovelle, Benjamin López Pérez. Laboratorio de Bases de Datos: Realización con otras asignaturas. Jornadas de Investigación y Docencia en Bases de Datos.

Junio 1996.

[Andersen98] Anders Andersen. "A note on reflection in Python 1.5" Distributed Multimedia Research Group Report. MPG-98-05, Lancaster University (Reino Unido). Marzo de 1998.

[Aoki91] P.M. Aoki. "Implementation of Extended Indexes in POSTGRES." Special Interest Group on Information Retrieval Forum 25, 1, 1991.

[AspectJ] AspectJ homepage. http://eclipse.org/aspectj

[Assumpcao93] Jecel Assumpcao Jr. "O Sistema Orientado a Objetos Merlin em Máquinas Paralelas". Anais do V SBAC-PAD. Florianópolis (Brasil). Septiembre de 1993.

[Assumpcao95] Jecel Assumpcao Jr. y Sergio Takeo Kufuji. "Bootstrapping the Object-Oriented Operating System Merlin: Just add Reflection". Meta'95 Workshop on Advances in Metaobject Protocols and Reflection. ECOOP. 1995.

[Atkinson2000] Atkinson, M. P., Dmitriev, M., Printezis, T. "Scalable and Recoverable Implementation of Object Evolution for the PJama Platform". Proceedings of the 9º Intl. Workshop on Persistent Object Systems, Lillehammer, Norway, pp 255-268.

[Atkinson86] Atkinson, M.P., Morrison, R. & Pratten, G.D., 1986. "Designing a Persistent Information Space Architecture". En Proc. 10th IFIP World Congress, Dublin pp 115-120.

[Atkinson89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier y S. Zdonik. "The Object-Oriented Database System Manifesto". En Proc. of 1st International Conference on Deductive and Object-Oriented Databases (DOOD). Japón, 1989.

[Atkinson95] Malcolm Atkinson, Ronald Morrison. "Orthogonally persistent object systems The VLDB Journal". The International Journal on Very Large Data Bases archive Volume 4, Issue 3 (Julio 1995)

[Atkinson96] M. Atkinson, L. Daynès, M. Jordan, T. Printezis and S. Spence. "An Orthogonally Persistent Java", SIGMOD Record, vol 25 n. 4, 1996.

[Autonomic] Autonomic Computing. IBM. http://www-3.ibm.com/autonomic/index.shtml

[Baker2002] Jason Baker, Wilson Hsieh. "Runtime aspect weaving through metaprogramming". AOSD 2002 conference Proceedings, Pages: 86 – 95

[Bauer2004] Christian Bauer, Gavin King. "Hibernate in action". Manning publications. 2004.

[Beck96] Kent Beck. "Smalltalk Best Practice Patterns". Prentice Hall PTR; ISBN: 013476904X; 1st edition (October 3, 1996)

[Benzaken90] Veronique Benzaken y Claude Delobel. "Enhancing performance in a persistent object store: Clustering strategies in O2. Technical Report 50-90, Altair, Agosto 1990.

[Benzaken95] Benzaken, V., Delobel, C., Harrus, G. "Clustering Strategies in O_2: an overview". Capítulo de "Building an Object-Oriented System: the story of O_2"

[Bergmans94] L. Bergmans. "Composing Concurrent Objects: Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs". Ph. D. Dissertation. Universidad de Twente (Holanda). Junio de 1994.

[Bertino89] E. Bertino y W. Kim. "Indexing Techniques for Queries on Nested Objects". IEEE Transactions on Knowledge and Data Engineering. Vol 1 nº2, 1989.

[Bertino95] E. Bertino y P. Foscoli. "Index Organizations for Object-Oriented Database Systems". IEEE Transactions on Knowledge and Data Engineering. Vol.7, 1995.

[Bertino99] E. Bertino y B. Chin. "The Indispensability of Dispensable Indexes". IEEE Transactions on Knowledge and Data Engineering, vol. 11, nº 1, 1999.

[Biesack2005] David Biesack. "Plugging into SourceForge.net". Eclipse Corner Article. Octubre, 2005.

[Blair97] Gordon S. Blair, Geoff Coulson. "The Case for Reflective Middleware". Proceedings of the 3rd Cabernet Plenarg Workshop. Rennes (Francia). Abril de 1997.

[Böllert99] Böllert, K. "On Weaving Aspects". In: European Conference on Object-Oriented Programming (ECOOP) Workshop on Aspect Oriented Programming. 1999.

[Boner2003] Boner, Jonas. "AspectWerkz - dynamic AOP for Java". Technical Paper.
http://codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf. 2003

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

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

[Box99] Don Box. "Essential COM". Addison-Wesley. Reading, Massachusetts (EE.UU.) ISBN 0201634465. 1999.

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

[Buschmann96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad, Michael Stal . "Pattern-Oriented Software Architecture, Volume 1: A System of Patterns". John Wiley and Sons. 1996

[CAI98] CAI. Computer Associates International. Jasmine. Diponible en URL http://www.cai.com/products/jasmine, 1998.

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

[Carey86] M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E.

Richardson y E. J. Shekita. "The Architecture of the EXODUS Extensible DBMS". Proceedings of International Workshop on Object-oriented database system, 1986.

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

[Cattell94b] R. Cattell, T. Atwood, J. Duhl, G. Ferran, M. Loomis, D. Wade, D. Barry, J. Eastman y D. Jordan. "The Object Database Standard: ODMG-93. Morgan Kaufmann Publishers, 1994".

[Cattell96] Cattell, R. (ed). "The Object Database Standard: ODMG-93", Release 1.2. Morgan Kaufmann, 1996.

[Cattell99] R. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda y F. Velez. "The Object Data Standard: ODMG 3.0". Morgan Kaufmann Publishers, 1999.

[CGLIB] CGLIB Home Page. http://cglib.sourceforge.net/

[Chamberlin74] Donald D. Chamberlin and Raymond F. Boyce, 1974. "SEQUEL: A structured English query language". International Conference on Management of Data. Proceedings of the ACM SIGMOD workshop on Data description, access and control. http://portal.acm.org/citation.cfm?id=811515. 1974

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

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

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

[Chiba98] Shigeru Chiba, Michiaki Tatsubori. "Yet Another java.lang.Class". ECOOP'98 Workshop on Reflective Object-Oriented Programming Systems. 1998.

[Chin92] C. Chin, B. Chin, H. Lu. "H-trees: A Dinamic Associative Search Index for OODB". ACM SIGMOD, 1992.

[Clossman98] G. Clossman, P. Shaw, M. Hapner, J. Klein, R. Pledereder and B. Becker, "Java and Relational Databases: SQLJ". ACM SIGMOD Record, 27(2):500, June 1998.

[Codd70] E. F. Codd "A Relational Model of Data for Large Shared Data Banks". Communications of the ACM 13 (6)..Junio de 1970.

[Cointe88] Pierre Cointe. "The ObjVlisp Kernel: a Reflective Lisp Architecture to define a Uniform Object-Oriented System". Meta-Level Architectures and Reflection. P. Maes and D. Nardi Editors. North-Holland, 1988.

[Cointe92] Pierre Cointe, Jacques Malenfant, Chistophe Dony, Philippe Mulet.

"Etude de la réflexion de comportement dans le langage Self". Premières Journées Représentation par Objects. La Grande Motte (Francia). 19992.

[Cooper96] Tim Cooper. "Barbados: an Integrated Persistent Programming Environment". PhD Thesis. Basser Department of Computer Science. University of Sydney. 1996.

[Cueva91] Juan Manuel Cueva Lovelle. "Lenguajes, Gramáticas y Autómatas". ISBN: 84-600-7871-X. Diciembre de 1991.

[Cueva92] 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.

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

[Cueva98] Juan Manuel Cueva Lovelle. "Conceptos Básicos de Procesadores de Lenguaje". Cuaderno Didáctico número 10. Editorial Servitec.. Diciembre de 1998.

[DAJ] DAJ: Demeter in AspectJ. Home page. http://www.ccs.neu.edu/research/demeter/DAJ/

[Darmon2000] Darmon, J., Formantín, C., Reguier, S., Schneider, M. "Dinamic clustering in Object Oriented Databases". Symposium of Objects and Databases, ECOOP . 2000.

[Demeter96] Karl J. Lieberherr. "Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns". PWS Publishing Company. 1996.

[Deux91] O. Deux. "The O_2 System. Commun" ACM 34(10). pp 34-48. 1991

[Devillechaise2003] Marc Ségura-Devillechaise, Jean-Marc Menaud, Gilles Muller, Julia L. Lawall. "Web cache prefetching as an aspect: towards a dynamic-weaving based solution". AOSD 2003 Proceedings, Pages: 110 – 119.

[Devis97] Ricardo Devis Botella. "C++. STL-Plantillas-Excepciones-Roles y Objetos". Paraninfo, 1997.

[Dijk95] W. van Dijk, J. Mordhorst. "Composition Filters in Smalltalk". HIO Graduation Thesis. Universidad de Twente (Holanda). 1995.

[Douence99] Rémi Douence, Mario Südholt. "The next 700 Reflective Object-Oriented Languages". École des mines de Nantes. Dept. Informatique (Francia). Technical report no.: 99-1-INFO. 1999.

[Drew90] Drew, P., King. R. "The performance and utility of the CACTIS implementation algorithms". Proceedings of the 16º VLDB Conference, pp 135-147. Brisbane, Australia, 1990.

[Drye99] Stephen C. Drye and William C. Wake. "Java Foundation Classes Swing Reference". Manning Publications Co. 1999

[DynamicAspects] DynamicAspects Home Page: http://dynamicaspects.sourceforge.net/

[Eckel2000] Bruce Eckel. "Thinking in Java, second edition". Prentice Hall. ISBN 0-13-027363-5. 2000.

[Eckel2000b] Bruce Eckel. "Thinking in C++", second edition, volume 1. Prentice Hall, 2000.

[Excelon97] Excelon corp. "Comparing ODBMS and RDBMS Implementations of Qantum Objects". http://www.exceloncorp.com/products/objectstore/os_white_papers.html, 1997

[Ferber88] Jacques Ferber. "Conceptual Reflection and Actor Languages". Meta-Level Architectures and Reflection. P. Maes and D. Nardi Editors. North-Holland. 1988.

[Fisher2003] Fisher, M., Ellis, J., Bruce, J. "JDBC Tutorial and Reference, Third Edition", Addison Wesley, 2003.

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

[Foote92] Brian Foote. "Objects, Reflection, and Open Languages". Workshop on Object-Oriented Reflection and Metalevel Architectures. ECOOP'92. Utrect (Holanda). 1992.

[Foote98] Brian Foote, Joseph Yoder. "Metadata and Active Object-Models". Proceedings of Plo98. Technical Report #wucs-98-25. Octubre 1998. URL: http://jerry.cs.uiuc.edu/~plop/plop98

[Fowler2004] Martin Fowler. "Inversion of Control Containers and the Dependency Injection pattern". http://www.martinfowler.com /articles/injection.html. Enero de 2004.

[Fussel97] Fussel, Mark L. 1997. "Foundations of Object Relational Mapping". ChiMu Corporation. ww.chimu.com/publications/objectRelational

[Gamma95] Eric Gamma, Richard Helm, John Vlissides. "Design Patterns Applied", tutorial Oopsla 1995.

[Glandrup95] M. Glandrup. "Extending C++ Using the Concepts of Composition Filters". Master Science Thesis. Universidad de Twente (Holanda). 1995.

[GOF94] Eric Gamma, R. Helm, R. Johnson, J.O. Vlissides. "Design Patterns, Elements of Reusable Object-Oriented Software". Addison-Wesley Editorial. 1994.

[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 Implementation of a Meta Architecture for Java". Friedrich-Alexander-Universität. Computer Science Department. Erlangen-Nürnberg, Alemania. Enero de 1997.

[Golm97b] Michael Golm, Jürgen Kleinöder. "Implementing Real-Time Actors with MetaJava". ECOOP'97 Workshop on Reflective Real-time Object-Oriented Programming and Systems. Jyväskylä (Finlandia). Junio de 1997.

[Golm97c] Michael Golm, Jürgen Kleinöder. "MetaJava – A Platform for Adaptable Operating-System Mechanisms". ECOOP'97 Workshop on Reflective Real-time Object-Oriented Programming and Systems. Jyväskylä (Finlandia). Junio de 1997.

[Golm98] Michael Golm, Jügen Kleinöder. "metaXa and the Future of Reflection". OOPSLA'98 Workshop in Reflective Programming. Vancouver (Canadá). Octubre de 1998.

[González2002] Martín González Rodríguez, Benjamin López Pérez, María del Puerto Paule Ruíz, Juan Ramón Pérez Pérez. Dynamic Generation of Interactive Dialogs Based on Intelligent Agents. Second International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems. Lecture Notes in Computer Science 2347. Mayo 2002.

[Gosling2004] James Gosling, Bill Joy y Guy Seele. "The Java™ Language Specification" (Third Edition). Addison Wesley Publishing Company. 2004.

[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.

[Hibernate2005] Hibernate Reference Documentation. Versión 3.0.5. 2005. http://www.hibernate.org/

[Hohl96] Fritz Hohl, Joachin Baumann, Markus Straber. "Beyond Java Merging CORBA-based Mobile Agents and WWW". Joint W3C/OMG Workshop on Distributed Objects and Mobile Code. Boston, Massachusetts (EE.UU.) Junio de 1996.

[Holub90] A. I. Holub. "Compiler Design in C". Prentice Hall. 1990.

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

[Hölzle94] U. Hölzle and D. Ungar: "A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance", Proceedings of the Object-Oriented Programming Languages, Systems and Applications (OOPSLA), 1994.

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

[IBM2000] "Multi-Dimensional Separation of Concerns". International Business Machines Corporation, IBM Research. 2000.

[IBM2000b] "HyperJ™: Multi-Dimensional Separation of Concerns for Java™". International Business Machines Corporation, IBM Research. 2000.

[ICSE2000] Second Workshop on Multi-Dimensional Separation of Concerns in Software Engineering. ICSE'2000. Limerick (Irlanda). Junio de 2000.

[Ingalls78] D. H. Ingalls. "The Smalltalk-76 Programming System Design and Implementation"" Conference Record on the 5th Annual ACM Symposium on Principles of Programming Languages. Enero de 1978.

[IronPython] IronPython Home Page. http://www.ironpython.com/

[Ishikawa96] H. Ishikawa y Y. Izumida. "An Object-Oriented Database System Jasmine: Implementation, Application, and Extension". IEEE Transactions on Knowledge and Data Engineering, vol 8 nº 2, Abril, 1996.

[Izquierdo2002] Izquierdo Castanedo, Raúl. "RDM: Arquitectura software para el modelado de dominios en sistemas informáticos". Tesis doctoral Universidad de Oviedo, 2002.

[Izquierdo2003] Raúl Izquierdo Castanedo, Aquilino A. Juan Fuente, Benjamín López, Ricardo Davis, Juan Manuel Cueva Lovelle, César F. Acebal. Experiences in Web Site Development with Multidisciplinary Teams. From XML to JST. International Conference in Web Engineering, Springer Verlag, Lecture Notes in Computer Science 2722. Junio de 2003.

[Jikes] Jikes home page. http://jikes.sourceforge.net/

[Johnson2005] Rod Jonson et All. "Spring Framework Reference Documentation. Version 1.2.6". 2005

[Johnson98] Ralph E. Johnson and Jeff Oakes. "The User Defined Product Framework" 1998. URL http://st.cs.uiuc.edu/pub/papers/frameworks/udp

[Johnson98b] Ralph Johnson, Bobby Wolf. "Type Object". Pattern Languages of Program Design 3. Addison Wesley 1998

[Jordan2000] M.J. Jordan and M.P. Atkinson. "Orthogonal Persistence for the Java Platform". Specification. Sun Microsystems Laboratories, Palo Alto, CA 94303, USA, 2000.

[Jordan2004] Jordan, M. "A Comparative Study of Persistence Mechanisms for the Java Platform". SML Technical Report Series. Sun Microsystems Laboratories. Septiembre de 2004.

[Jordan96] M. Jordan. "Early Experiences with Persistent Java". In proceedings of the First International Workshop on Persistence and Java. Glasgow, Scotland, September 1996.

[Jordan98] M. J. Jorda and M. P. Atkinson. "Orthogonal Persistence for Java – A mid-term Report". Proceedings of the Third International Wokshop on Persistence and Java (PJW3). 1998

[Kalev98] Danny Kalev. "The ANSI/ISO C++ Professional Programmers Handbook". Que Editorial. 1998.

[Kaplan2000] Kaplan A., Ridgway, J. H. G., Schmerl, B. R. "Toward Polylingual Persistence". Proceedings of the Ninth International Workshop on Persistent Object Systems. pp 202-217. 2000.

[Kaplan96] Kaplan A., Myrestrand, G. Ridgway, J. V. E, Wileden, J.C. "Our Spin on Persistent Java: The JavaSPIN Approach". Proceedings of the

First International Workshop on Persistence and Java (PJW3). 1996.

[Kemper94] A. Kemper, C. Kilger y G. Moerkotte. "Function Materialization in Object Bases: Design, Realization and Evaluation". IEEE Transactions on Knowledge and Data Engineering, 1994.

[Kiczales2001] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G. Griswold. "Getting Started with AspectJ". CACM 2001. aspectj.org. 2001.

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

[Kiczales92] Kiczales, G., Rivieres, J., Bobrow, D.G. "The Art of Meta-object Protocol". MIT Press. 1992.

[Kiczales97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. "Aspect Oriented Programming". Proceedings of ECOOP'97 Conference. Finlandia. Junio de 1997.

[Kielze2002] J. Kielze and R. Guerraoui: "AOP: Does it Make Sense? The Case of Concurrency and Failures", European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 2374, 2002.

[Kim89] W. Kim, K.C. Kim, A. Dale. "Indexing Techniques for Object-Oriented Databases". En W. Kim y F.H. Lochovsky (ed): Object-Oriented Concepts, Databases, and Applications. Addison-Wesley, 1989.

[Kim91] Kim, W., Ballou, N., Chou, H., Garza, J., Eowlk, D. "Features of the ORION Object Oriented Database System". 1991.

[Kirby98] Graham Kirby, Ron Morrison, David Stemple. "Linguistic Reflection in Java". Software Practice & Experience, 28. 1998.

[Kirtland99] Mary Kirtland. "Designing Component-based Applications". Microsoft Press. ISBN 0-7356-0523-8. Redmond, Washington (EE.UU.) 1999.

[Kirtland99b] Mary Kirtland. "Object-Oriented Software Development Made Simple with COM+ Runtime Services". Microsoft Systems Journal. Noviembre de 1999.

[Kleinöder96] Jügen Kleinöder, Michael Golm. "MetaJava: An Efficient Run-Time Meta Architecture for Java™". Proceedings of the International Workshop on Object Orientation in Operating Systems (IWOOS'96). Seattle, Washington (EE.UU.). Octubre de 1996.

[Krall98] Andreas Krall. "Efficient JavaVM just-in-time compilation". International Conference on Parallel Architectures and Compilation Techniques, pages 205–212, Paris, 1998. North-Holland.

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

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

[Laddad2002] Ramnivas Laddad. "I want my AOP!". January 2002 issue of Java World http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html

[Lam2002] John Lam. "CLAW, Cross-Language Load-Time Aspect Weaving on Microsoft's Common Language Runtime". In AOSD 2002 conference. http://www.iunknown.com

[Ledoux96] T. Ledoux, P. Cointe. "Explicit Metaclasses as a Tool for improving the Design of Class Libraries". Proceedings of ISOTAS'96, Springer-Verlang, Kanazawa (Japón). Marzo de 1996.

[Ledoux99] Thomas Ledoux. "OpenCorba: a Reflective Open Broker". Lecture Notes in Computer Science, vol. 1616. 1999.

[Lee98] W. Lee y D.L. Lee. Path Dictionary: "A New Access Method for Query Processing in Object-Oriented Databases". IEEE Transactions on Knowledge and Data Engineering. Vol 10, nº3, 1998.

[Lewis2000] Brian Lewis, Bernd Mathiske, and Neal Gafter. "Architecture of the PEVM: A High-Performance Orthogonally Persistent Java Virtual Machine". SMLI TR-2000-93 October 2000.

[Ley2002] Michael Ley. "Dblp: Digital bibliography and library project". 2002. http://dblp.uni-trier.de/.

[Lieberherr96] Karl J. Lieberherr. "Adaptive Object Oriented Software: The Demeter Method". PWS Publishing Company. 1996.

[Lieberman86] H. Lieberman. "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems". In OOPSLA'86 Conference Proceedings. SIGPLAN Notices, 21, 11, 214-223. 1986.

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

[Lindsay87] B. Lindsay, J. McPherson y H. Pirahesh. "A Data Management Extension Architecture". Proceedings of the ACM SIGMOD Annual Conference on Management of data, pp 220-226, 1987.

[López2004] Benjamín López Pérez, Francisco Ortín Soler, Javier Noval Arango. "Reflection as the basis for Developing a Dynamic SoC Persistence System. Journal of Object Technology", Volume 3, Issue 8. Septiembre 2004.

[López2004b] Benjamín López Pérez, Francisco Ortín Soler, Juan Manuel Cueva Lovelle. "Separación Dinámica del Aspecto de Persistencia mediante Reflectividad Computacional". IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD). Málaga. Noviembre 2004.

[Lundh99] Fredrik Lundh. "An Introduction to Tkinter". Review Copy. Enero de 1999.

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

[Maes87b] Pattie Maes. "Issues in Computational Reflection. Meta-Level Architectures and Reflection". Pattie Maes and D. Nardi Editors. North-

Holland. Bruselas, Bélgica. Agosto de 1987.

[Maier89] David Maier: "Why Isn't There an Object-Oriented Data Model?" IFIP Wold Computer Congress, pp. 793-798. San Francisco, USA, Agosto 1989.

[Manolis92] Manolis M. Tsangaris, Jeffrey F. Naughton. "On the Performance of Object Clustering Techniques". SIGMOD Conference: 144-153. 1992

[Martínez2001] Ana Belén Martínez Prieto. "Un Sistema de Gestión de Bases de Datos Orientadas a Objetos sobre una Máquina Abstracta Persistente". Tesis Doctoral. Departamento de Informática. Universidad de Oviedo. Mayo de 2001.

[Martínez99] Ana Belén Martínez Prieto y Juan Manuel Cueva Lovelle. "Técnicas de Indexación para las Bases de Datos Orientadas a Objetos". Novática, Monográfico Bases de Datos Avanzadas, nº 140, julio-agosto 1999.

[Matsuoka91] Satoshi Matsuoka, Takuo Watanabe, Akinori Yonezawa. "Hubrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming". Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91), Ginebra (Suiza). 1991.

[Matthijs97] Matthijs, F., Joosen, W., Vanhaute, B., Robben, B., Verbaten, P., "Aspects should not die". In: European Conference on Object-Oriented Programming (ECOOP) Workshop on Aspect-Oriented Programming. 1997.

[Melton93] J. Melton and A. Simon. "Understanding the New SQL: A Complete Guide". Morgan Kaufmann, 1993, ISBN 1-55860-245-3.

[Melton99] A. Eisenberg y J. Melton. "SQL: 1999, formerly known as SQL 3". SIGMOD Record 28 (1), Marzo 1999.

[Mens97] K. Mens, C. Lopes, B. Tekinerdogan, and G. Kiczales: "Aspect Oriented Programming Workshop Report", ECOOP Workshop Reader, Springer-Verlag LNCS 1357, 1997.

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

[Microsoft95] "The Component Object Model Specification". Version 0.9. Microsoft Corporation. Octubre de 1995.

[Mosses92] Peter D. Mosses. "Action Semantics". Cambridge University Press. 1992.

[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 de 1993.

[Mulet94] P. Mulet, T. Ledoux, D. Barbaron, F. Rivard, P. Cointe. "Importing SOM Libraries into Classtalk". OOPSLA'94 Workshop on Experiences with CORBA: Is CORBA ready for duty? 1994.

[Murata94] Kenichi Murata, R. Nigel Horspool, Yasuhiko Yokote, Erig G. Mannig, Mario Tokoro. "Cognac: a Reflective Object-Oriented Programming System using Dynamic Compilation Techniques". Proceedings

of the annual Conference of Japan Society of Software Science and Technology (JSSS'94). Octubre de 1994.

[Nicoara2005] Angela Nicoara, Gustavo Alonso. "Dynamic AOP with PROSE". In: Proc. of International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA'05) in conjunction with CAISE'05, Porto, Portugal, June 2005.

[O'Brien2001] O'Brien, L., "The First Aspect-Oriented Compiler", Software Development Magazine, September. 2001

[Oliva98] Alexandre Oliva, Luiz Eduardo Buzato. "An overview of MOLDS: A Meta-Object Library for Distributed Systems". Segundo Workshop em Sistemas Distribuidos, Curitiba (Brasil). Junio de 1998.

[Oliva98b] Alexandre Oliva, Islene Calciolari Garcia, Luiz Eduardo Buzato. "The reflexive architecture of Guanará". Technical Report IC-98-14, Instituto de Compurtaçao, Universidad de Campinas. Abril de 1998.

[Oliva98c] Alexandre Oliva, Luiz Eduardo Buzato. "The implementation of Guanará on Java". Technical Report IC-98-32, Instituto de Compurtaçao, Universidad de Campinas. Septiembre de 1998.

[Oliva99] Alexandre Oliva, Luiz Eduardo Buzato. "The Design and Implementation of Guanará". 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99). San Diego (EE.UU.). 1999.

[OMG2001] Object Management Group. "A Proposal for an MDA Foundation Model". An ORMSC White Paper. V00-02. ormsc/05-04-01.

[OMG2003] [UML03] OMG Unified Modeling Languaje Specification. version 1.5. www.uml.org, 2003.

[OMG95] Object Management Group (OMG). "The Common Object Request Broker: Architecture and Specification". Julio de 1995.

[OMG96] Object Management Group (OMG). "Description of New OMA Reference Model. Draft 1". Mayo de 1996.

[OMG97] Object Management Group (OMG). "A Discussion of the Object Management Architecture". Enero de 1997.

[OMG98] Object Management Group (OMG). "CORBA Components. CORBA 3.0 Draft Specification". Noviembre de 1998.

[OOPSLA99] First Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems. OOPSLA'99. Denver (EE.UU.). Noviembre de 1999.

[Oracle99] Oracle Corporation. "Oracle 8iTM Objects and Extensibility Option. Features Overview". Disponible en URL http://www.oracle.com, Febrero 1999.

[Orfali98] Robert Orfali, Dan Harkey. "Client/Server Programming with Java and CORBA". 2ª Edición. Wiley Editorial. 1998.

[Orleans2001] Doug Orleans, Karl J. Lieberherr: DJ: Dynamic Adaptive Programming in Java. Reflection. 2001.

[Ortín2001] Francisco Ortín Soler, Juan Manuel Cueva Lovelle. "Building a Completely Adaptable Reflective System". European Conference on Ob-

ject Oriented Programming ECOOP'2001. Workshop on Adaptive Object-Models and Metamodeling Techniques. Budapest (Hungría). Junio de 2001.

[Ortín2002] Francisco Ortín Soler, Juan Manuel Cueva Lovelle. "Implementing a Real Computational-Environment Jump in order to Develop a Runtime-Adaptable Reflective Platform". ACM SIGPLAN NOTICES 37(8). Agosto 2002.

[Ortín2003] Francisco Ortín Soler, Juan Manuel Cueva Lovelle. Non-Restrictive Computational Reflection. Elsevier Computer Standards and Interfaces 25(3). Junio 2003.

[Ortín2004] Francisco Ortín Soler, Benjamín López Pérez, José Baltasar G. Pérez-Schofield. "Separating Adaptable Persistence Attributes through Computational Reflection.". IEEE Software 21(6). Noviembre 2004.

[Ortín2004b] Francisco Ortín, Juan Manuel Cueva, Raul Izquierdo, Aquilino Juan, Maria Cándida Luengo, José Emilio Labra. Análisis Semántico en Procesadores de Lenguaje. Servitec. Marzo 2004.

[Ortín2004c] Francisco Ortín Soler, Juan Manuel Cueva Lovelle. "Dynamic Adaptation of Application Aspects". Elsevier Journal of Systems and Software 71(3). Mayo 2004.

[Ortín2005] Francisco Ortín Soler, José M. Redondo, Luis Vinuesa, Juan M. Cueva. "Adding Structural Reflection to the SSCLI". Journal of .Net Technologies, Volume 3, Number 1-3, pp. 151-162. Mayo 2005.

[Ortín2005b] Francisco Ortín Soler. "Extending Rotor with Structural Reflection to Support Reflective Languages". Microsoft Research SSCLI RFP II Capstone Workshop. Redmond (Washington), USA. Septiembre 2005.

[Ossher99] H. Ossher, P. Tarr. "Multi-Dimensional Separation of Concerns using Hyperspaces". IBM Research Report 21452. Abril de 1999.

[PARC] Xerox PARC Research. http://www.parc.com/groups/csl/projects/aspectj/

[Parnas72] Parnas, D. "On the Criteria to be Used in Decomposing Systems into Modules." Communications of the ACM, Vol. 15, No. 12. 1972.

[Pérez2000] Jesús Arturo Pérez Díaz, Benjamín López Pérez, Bernardo Martín Glz., Darío Álvarez Gutiérrez. "An Overview of the Sahara Security Architecture for Mobile Agent Systems". World Multiconference on Systemics, Cybernetics and Informatics. Orlando, USA. Julio, 2000.

[Pérez98] Jesús Arturo Pérez Díaz, Juan Manuel Cueva Lovelle, Benjamín López y Sara Isabel García Barón. "Parallel Programming on the Internet using Mobile Agnets". Second IASTED International Conference European Parallel and Distributed Systems EURO-PDS'98. Viena, Austria. Julio de 1998.

[Pérez98b] Jesús Arturo Pérez Díaz, Juan Manuel Cueva Lovelle y Benjamín López. "Mobile Agents Development on the Internet, is Java or Corba the Dominant Trend?" The Third International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agent Technology, PAAM'98. Londres, Reino Unido, Marzo de

1998.

[Pérez98c] Jesús Arturo Pérez Díaz, Juan M. Cueva Lovelle, Benjamín López Pérez, Francisco Domínguez Mateos, Candida luengo Diez, Francisco Javier Pérez, Sara Isabel García Baron. "Design Patterns for Mobile Agents Applications". V Congreso Internacional de Investigación en Ciencias Computacionales CIICC'98. Aguascalientes, México. Noviembre de 1998.

[Pietrek2001] Pietrek, Matt. "The .NET Profiling API and the DNProfiler Tool". MSDN Magazine. Diciembre de 2001.

[Pinto2002] M. Pinto, L. Fuentes, M.E. Fayad, J.M. Troya. "Separation of Coordination in a Dynamic Aspect Oriented Framework". AOSD 2002 Proceedings, Pages 134-140

[Poet2000] "Poet Object Server Suite 6.0 User Guide". Disponible en URL http://www.poet.com/, 2000.

[Poet2004] FastObjects™ .NET Programmer's Guide. Poet Software. 2004

[Popovici2001] Popovici, A., Gross, T., Alonso, G., 2001. "Dynamic Homogenous AOP with PROSE", Technical Report, Department of Computer Science, ETH Zürich, Switzerland.

[Popovici2002] Popovici, A., Gross, T., Alonso, G. "Dynamic Weaving for Aspect Oriented Programming". In 1st Intl. Conf. on Aspect-Oriented Software Development, Enschede, The Netherlands, Aug. 2002

[Popovici2003] Popovici, A., Alonso, G., Gross, T. "Juis-In-Time Aspects: Efficient Dynamic Weaving for Java". In 2nd Intl. Conf. on Aspect-Oriented Software Development, Boston, USA, Apr. 2003.

[Pryor2002] Jane Pryor, Andres Diaz Pace and Marcelo Campo. Re-flecting on Separation of Concerns. Revista de Informática Teórica y Aplicada (RITA) Volume X, No. 2 (2002)

[Ramaswamy95] S. Ramaswamy y C. Kanellakis. "OODB Indexing by Class Division". ACM SIGMOD, 1995.

[Rashid2000] A. Rashid: "On to Aspect Persistence", GCSE Symposium, Springer-Verlag LNCS 2177, 2000.

[Rashid2003] A. Rashid, R. Chitchyan: "Persistence as an Aspect", International Conference on Aspect-Oriented Software Development, 2003.

[Reese2000] Reese, George. "Database Programming with JDBC and Java, Second Edition". O'Reilly & Associates. 2000.

[Ridgway2000] John V. E. Ridgway, Jack C. Wileden. "The JSPIN Experience: Exploring the Seamless Persistence Option for Java". L'OBJET 6(3). 2000.

[Ridgway97] John V. E. Ridgway, Craig Thrall, Jack C. Wileden. "Toward Assessing Approaches to Persistence for Java". CMPSCI Technical Report. Agosto de 1997.

[Riehle2000] Dirk Riehle, Michel Tilman, Ralph Johnson. "Dynamic Object Model". PLoP 2000

[Rivard96] F. Rivard. "A new Smalltalk kernel allowing both explicit and implicit

metaclasses programming". Workshop in Extending the Smalltalk Language, OOPSLA'96. San José (EE.UU.). Octubre de 1996.

[Rivières84] J. des Rivières, B. C. Smith. "The Implementation of Procedurally Reflective Languages". Proceedings of ACM Symposium on Lisp and Functional Programming. 1984.

[Roman2005] Ed Roman, Rima Patel Sriganesh, Gerald Brose "Mastering Enterprise JavaBeans Third Edition". Wiley Publishing. 2005.

[Roos2003] Roos R. M.. "Java Data Objects". Addison Wesley. ISBN 0-321-12380-8. 2003

[Roselló2001] Roselló E. G., Ayude J., García Perez-Schofield B., Perez M. "Towards orthogonal concurrency principles in object-oriented systems design". Journal of Object Technology (JOT). 2001.

[Rossum2001] Guido van Rossum. "Python Reference Manual". Fred L. Drake Jr. Editor. Relesase 2.1. Abril de 2001.

[Rumbaugh98] James Rumbaugh, Ivar Jacobson, Grady Booch. "The Unified Modeling Language Reference Manual". Addison-Wesley. Diciembre de 1998.

[Sanders98] Roger E. Sanders. "ODBC 3.5 Developers Guide". McGraw-Hill Companies. 1998

[Schofield2002] J. B. García Perez-Schofield, E. García Roselló, Tim B. Cooper, M. Pérez Cota. "Managing schema evolution in a container-based persistent system". Software, Practice & Expererience 32 (14). pp. 1395-1410, 2002.

[Schofield2002b] J. B. García Perez-Schofield, Tim B. Cooper, E. García Roselló, M. Pérez Cota. "First Impressions about executing real applications in Barbados". Fourth Workshop on Object-Oriented Operating Systems, 16th European Conference on Object Oriented Programming, Málaga, España. 2002.

[Schofield2002c] J.B. García Perez-Schofield. "Persistencia, Evolución del esquema y Rendimiento en el modelo basado en contenedores". Tesis Doctoral. Departamento de Informática, Universidad de Vigo. 2002.

[Schofield2003] J.B. García Perez-Schofield, Tim Cooper, Emilio García Roselló, Manuel Pérez Cota. "El modelo de persistencia basado en contenedores: conclusiones sobre una aproxmiación distinta hacia la persistencia". 8º Jornadas de Ingeniería del Software y Bases de Datos (JISBD'03). Alicante, España. 2003.

[Schofield2004] J. B. García Perez-Schofield, E. García Roselló, M. Pérez Cota. "A new experience in lecturing object-oriented programming". IADAT Journal of Advanced Technology 1(4). Noviembre 2004.

[Schult2002] W. Schult and A. Polze: "Aspect-Oriented Programming with C# and .NET", IEEE International Symposium on Object-oriented Real-time distributed Computing, Mayo 2002.

[Smith82] B. C. Smith. "Reflection and Semantics in a Procedural Language". MIT-LCS-TR-272. Massachusetts Institute of Technology. Cambridge (EE.UU.). 1982.

[Smith95] Randall B. Smith, David Ungar. "Programming as an Experience: The Inspiration for Self". Sun Microsystems Laboratories. 1995.

[Sousa94] Sousa, P., Alves Marques, J. "Object Clustering in Persistent and Distributed Systems". pp 11-13. 1994.

[Sreenath94] B. Sreenath y S. Seshadri. "The hcC-Tree: An Efficient Index Structure for Object Oriented Databases". International Conference on VLDB, 1994.

[Stamos84] James W. Stamos. "Static grouping of small objects to enhance performance of a paged virtual memory". ACM Transactions on Computer Systems. Mayo de 1984.

[Steele90] Jr. Steele "Common Lisp: The Language". Segunda Edición. Digital Press, 1990.

[Stonebraker86] M.R. Stonebraker. "Inclusion of New Types in Relational Data Bases Systems". Proc. 2nd IEEE Data Engineering Conf. Los Angeles, CA, 1986.

[Strachey67] Strachey, C. "Fundamental Concepts in Programming Languages". Oxford University Press, Oxford. 1967.

[Stroustrup98] Bjarne Stroustrup. "The C++ Programming Language". Third Edition. Addison-Wesley. October 1998.

[Sullivan2001] Gregory T. Sullivan. "Aspect-oriented programming using reflection and metaobject protocols". Communications of the ACM. October 2001/ Vol 44 No. 10 Pages: 95 – 97.

[Sun2003] Sun Microsystems, Inc. JSR-000012 Java Data Objects (JDO) Specification.

[Sun2003b] "Java Platform Debugger Architecture Documentation — SDK 21.4.0." http://java.sun.com/j2se/1.4.2/docs/guide/jpda/

[Sun2003c] "Long-Term Persistence for JavaBeans". http://java.sun.com/ products/jfc/tsc/articles/persistence. Sun Developer Network article. 2003

[Sun2003d] "Enterprise Java Beans 2.1 specification". Sun Mirosystems. 2003.

[Sun2004] JVM Tool Interface specification". version 1.0. http://java.sun.com/ j2se/ 1.5.0/ docs/guide/jvmti/jvmti.html

[Sun2004b] "JAR File Specification". J2SE 1.5. Sun Mirosystems. http://java.sun.com/j2se/1.5.0 /docs/guide/jar/jar.html

[Sun2005] JSR-000220 Enterprise JavaBeansTM 3.0 Draft Review. Sun Microsystems. 2005.

[Sun95] "The Java Virtual Machine Specification". Release 1.0 Beta Draft. Sun Microsystems Computer Corporation. Agosto de 1995.

[Sun96] "JavaBeans™ 1.0 API Specification". Sun Microsystems Computer Corporation. Diciembre de 1996.

[Sun97] "Java™ Native Interface Specification". JavaSoft. Sun Microsystems. Mayo de 1997.

[Sun97b] "Java Core Reflection. API and Specification". JavaSoft. Enero de 1997.

[Sun97c] "Java™ Object Serialization Specification". JavaSoft. Sun Microsystems. Febrero de 1997.

[Sun98] "The Java HotSpot Virtual Machine Architecture". White Paper. Sun Microsystems. 1998.

[Sun99] "Dynamic Proxy Classes". Sun Microsystems, Inc. 1999.

[Suzuki99] J. Suzuki and Y. Yamamoto: "Extending UML for Modelling Reflective Software Components", The Unified Modeling Language - Beyond the Standard, Second International Conference, Springer-Verlag LNCS 1723, 1999.

[Tarr99] Tarr, P., Ossher, H., Harrison, W., Sutton, S., N "Degrees of separation: Multi-Dimensional Separation of Concerns". In: Proceedings of the 1999 International Conference on Software Engineering.

[Tatsubori98] Michiaki Tatsubori, Shigeru Chiba. "Programming Support of Design Patterns with Compile-time Reflection". OOPSLA'98 Workshop in Reflective Programming. Vancouver (Candada). Octubre de 1998.

[Thai2002] Thuan L. Thai. Hoang Lam. ".NET Framework Essentials, 2nd Edition". O'Reilly. 2002

[Thomas2004] Dave Thomas, with Chad Fowler y Andy Hunt. "Programming Ruby. The Pragmatic Programmer's Guide, Second Edition". The Pragmatic Bookshelf. 2004

[Tsangaris91] Tsangaris M. M, Naugthon J. F. "A stochastic approach for clustering in object stores". Proceedings of the SIGMOD International Conference on Management of Data. pp 12-21. Mayo 1991.

[Ungar87] D. Ungary R. B. Smith. "SELF: The Power of Simplicity". In OOPSLA'87 Conference Proceedings. Published as SIGPLAN Notices, 22, 12, 227-241. 1987.

[Venners98] Bill Venners. "Inside the Java Virtual Machine". Java Masters. McGraw Hill. 1998.

[W3C98] "Extensible Markup Language (XML) 1.0". World Wide Web Consortium. Febrero de 1998.

[Walls2003] Craig Walls, Norman Richards. "XDoclet in Action". Manning Publications. Diciembre de 2003.

[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.

[Watanabe88] Takuo Watanabe, Akinori Yonezawa. "Reflection in an object-oriented concurrent language". Proceedings of OOPSLA'88, vol 23. SIGPLAN Notices, ACM Press. Septiembre de 1988.

[Welch98] Ian Welch, Robert Stroud. "Dalang – A Reflective Java Extension". OOPSLA'98 Workshop in Reflective Programming. Vancouver (Canadá). Octubre de 1998.

[Wells92] D. L. Wells, J. A. Blakely, and C. W. Thompson. "Architecture of an open object-oriented management system". IEEE Computer, 25(10):74–82, Oct. 1992.

[Wolczko96] Mario Wolczko, Ole Agesen y Davied Ungar. "Towards a Universal Implementation Substrate for Object-Oriented Languages". Sun Microsystems Laboratories, documento #96-0506. Diciembre de 1996.

[Yoder2001] J. Yoder, F. Balaguer, R. Johnson. "The Architectural Style of Adaptive Object-Models". Workshop on Adaptive Object-Models and Metamodeling Techniques. Budapest (Hungría). Junio de 2001.

[Yokote92] Y. Yokote. "The New Mechanism for Object-Oriented System Programming". Proceedings of IMSA'92 International Workshop on Reflection and Meta-level Architecture". 1992.

[Yonezawa90] Akinori Yonezawa. "ABCL: An Object-Oriented Concurrent System". Computer Science Series. The MIT Press, 1990.

[Yue73] Yue. P. C., Wong C. K. "On the optimality of the proability ranking scheme in storage applications". JACM. Octubre de 1973.

[Zinky97] Zinky, J.A., Bakken, D.E., Schantz, R.E., "Architectural Support for Quality of Service for CORBA Objects". Theory and Practice of Object Systems, 3 (1). 1997.