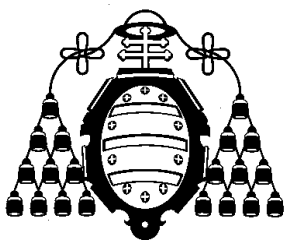
UNIVERSIDAD DE OVIEDO

Departamento de Informática



TESIS DOCTORAL

SISTEMA COMPUTACIONAL DE PROGRAMACIÓN FLEXIBLE DISEÑADO SOBRE UNA MÁQUINA ABSTRACTA REFLECTIVA NO RESTRICTIVA

Presentada por

Francisco Ortín Soler

para obtención del título de Doctor por la Universidad de Oviedo

Dirigida por el

Profesor Doctor D. Juan Manuel Cueva Lovelle

Oviedo, Diciembre de 2001

Universidad de Oviedo

Edita: Universidad de Oviedo
Biblioteca Universitaria, 2007
Colección Tesis Doctoral-TDR nº 3

ISBN: 978-84-690-6622-5
D.L.: AS.02274 -2007

RESUMEN

Esta tesis describe el modo en el que distintas técnicas de reflectividad pueden ser empleadas para el desarrollo de un sistema computacional de programación extensible y adaptable dinámicamente, sin sufrir dependencia alguna de un lenguaje de programación específico, y empleando para ello una plataforma virtual heterogénea.

Se diseña una máquina abstracta, de tamaño y complejidad semántica reducida, como la raíz computacional del sistema, que otorga primitivas básicas de reflectividad. Tanto su tamaño restringido como su capacidad introspectiva, la hacen susceptible de ser implantada en entornos computacionales heterogéneos, constituyendo adicionalmente un entorno computacional independiente de la plataforma.

Haciendo uso de las facultades reflectivas ofrecidas por la máquina abstracta su nivel de abstracción computacional podrá ser extendido dinámicamente, utilizando para ello su propio lenguaje de programación sin necesidad de modificar la implementación reducida de la máquina virtual, y, por tanto, sin perder portabilidad de su código. El empleo de su capacidad extensible se utilizará, a modo de ejemplo, en el diseño de abstracciones propias de persistencia, distribución, planificación controlada de hilos y recolección de basura. Todas las abstracciones ofrecidas mediante la extensión de la plataforma, utilizando su propio lenguaje, son adaptables para cualquier aplicación, en tiempo de ejecución.

Se diseña un sistema de procesamiento genérico de lenguajes disponiendo de las características reflectivas de la plataforma, dando lugar a una independencia global del lenguaje de programación seleccionado por el programador. Cualquier aplicación podrá interactuar con otra bajo el modelo computacional de objetos ofrecido por la máquina abstracta, independientemente del lenguaje en el que hayan sido creadas.

La flexibilidad dinámica del sistema creado es superior, tanto en expresividad como en el espectro de facetas computacionales adaptables, a los actualmente existentes. La estructura de las aplicaciones en ejecución, y la especificación léxica, sintáctica y semántica del lenguaje de programación, son parámetros configurables dinámicamente, tanto por la propia aplicación –en cualquiera que sea su lenguaje de programación–, como por cualquier otro programa. El sistema reflectivo ofrecido no posee restricción alguna respecto a las características computacionales a configurar, ni respecto al modo de expresar su adaptación. Toda la flexibilidad ofrecida es dinámica, no siendo necesario finalizar la ejecución de una aplicación para su configuración, pudiéndose adaptar ésta a requisitos surgidos dinámicamente, imprevisibles en tiempo de desarrollo.

PALABRAS CLAVE

Máquina abstracta, extensibilidad, adaptabilidad, introspección, reflectividad estructural, computacional y de lenguaje, modelo de objetos basado en prototipos, separación de aspectos e incumbencias, entorno de programación, intérprete genérico.

ABSTRACT

This thesis describes the way in which different reflective technologies can be used to develop a dynamically adaptable and extensible computational system, without any dependency of a concrete programming language, built over a heterogeneous computing platform.

An abstract machine with a reduced instruction set is built as the root computation system's engine; it offers the programmer basic reflection computation primitives. Its reduced size and its introspective capabilities, make it easy to be deployed in heterogeneous computational systems, becoming a platform-independent computational system.

By using the reflective features offered by the abstract machine, the programming abstraction level can be extended. This would be programmed on its own language, without needing to modify the virtual machine's source code and, therefore, without loosing code portability. As an example of its extensiveness, the programming environment is programmed (using the abstract machine programming language) achieving facilities such as persistence, distribution, thread scheduling or garbage collection. All this new abstractions are adaptable at runtime to any system application.

A generic processing language system is designed, making the whole system independent of the language the programmer may select. Any application can interact with each other, by using the abstract machine's computational model, whatever the programming language the application has been codified in.

The dynamic flexibility the system achieves is greater –both in expressiveness and in the adaptable computational-features set– than any other existing one. At runtime, the application structure and its programming language specification (lexical, syntactic and semantic features) are completely customisable; it can be adapted by the own application as well as by any other program –no matter the language they have been programmed in. The reflective system developed has no restriction at all: it can adapt any computational trait without any expressiveness restriction –it may use the whole abstract machine programming language. This flexibility offered is dynamic: there is no need to stop the application execution in order to adapt it to any runtime requirement, unexpected at design time.

KEYWORDS

Abstract machine, extensibility, adaptability, introspection, structural, computational and linguistic reflection, prototype-based object-oriented object model, separation of concerns, aspect oriented programming, programming environment, generic interpreter.

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 en la tesis enunciada en esta memoria 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

El modo en el que un ordenador es programado queda determinado, principalmente, por los requisitos que los futuros usuarios demandarán de dicha computadora. En el caso del entorno de computación de una oficina de ingeniería civil, se demandará la utilización de aplicaciones relacionadas con el diseño asistido por computador. Sin embargo, en el caso de entidades bancarias, el tratamiento de elevadas cantidades de información es el proceso más común. La elección del lenguaje de programación utilizado a la hora de implementar una aplicación, dentro de un amplio espectro de posibilidades, queda condicionada por el tipo de aplicación a desarrollar.

La semántica de un lenguaje de programación, así como el propósito para el que éste haya sido diseñado, restringe de algún modo la sencillez con la que pueda emplearse para resolver un determinado tipo de problema. Los lenguajes de propósito específico [Cueva98] están enfocados a desarrollar una clase concreta de aplicaciones del modo más sencillo posible, estando su semántica restringida al propósito para el que fueron creados. Por otro lado, los denominados lenguajes de propósito general tienen como objetivo el poder representar la solución de cualquier tipo de problema computacional. Sin embargo, dentro de esta clasificación existen diferencias significativas. Lenguajes como Java [Gosling96] y C++ [Stroustrup98], ambos de propósito general, orientados a objetos y de sintaxis análoga, pueden ser elegidos, por ejemplo, para desarrollar aplicaciones portables y distribuidas –en el primer caso–, o bien para crear aquéllas en las que prime su eficiencia en tiempo de ejecución –el caso del lenguaje C++.

La semántica computacional de un lenguaje de programación es algo fijo que se mantiene constante en las aplicaciones codificadas sobre él. Una aplicación carece de la posibilidad de modificar la semántica del lenguaje con el que fue creada, para poder amoldarse a nuevos requisitos que puedan surgir a lo largo de su existencia, sin necesidad de modificar su código existente –su funcionalidad principal.

Además de limitar las posibilidades computacionales de una aplicación a la hora de elegir el lenguaje de programación en el que ésta vaya a ser desarrollado, la interacción entre aplicaciones desarrolladas sobre distintos lenguajes se lleva a cabo comúnmente mediante la utilización de mecanismos adicionales. El uso de modelos binarios de componentes (como por ejemplo COM [Microsoft95]) o de arquitecturas de objetos distribuidos y *middlewares* (CORBA [OMG95]), son necesarios para permitir la interacción entre aplicaciones desarrolladas en distintos lenguajes de programación y, en ocasiones, sobre distintas plataformas.

Justificamos así la necesidad de estudiar las alternativas en la creación de un entorno computacional de programación, independiente del lenguaje y plataforma seleccionados, que permita desarrollar aplicaciones en cualquier lenguaje de programación, y que éstas puedan interactuar entre sí sin necesidad de utilizar ningún mecanismo intermedio. Del mismo modo, la semántica de un lenguaje de programación no deberá ser algo fijo e invariable a lo largo del ciclo de vida de las aplicaciones codificadas sobre él; su significado (semántica) podrá adaptarse a futuros contextos imprevisibles en tiempo de desarrollo.

A lo largo de esta tesis estudiaremos las distintas alternativas, y enunciaremos otras, para crear un entorno computacional de programación flexible, en el que se satisfagan todos los objetivos enunciados someramente en el párrafo anterior.

1.2 Objetivos

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

1.2.1 Independencia del Lenguaje de Programación y Plataforma

El sistema computacional podrá ser programado mediante cualquier lenguaje y sus aplicaciones podrán ser ejecutadas sobre cualquier plataforma. La dependencia actualmente existente en determinados sistemas computacionales, en el que el lenguaje de programación de aplicaciones y la plataforma de desarrollo imponen una determinada plataforma en su implantación, deberá ser suprimida. La elección del lenguaje de programación a utilizar estará únicamente determinada por el modo en el que su expresividad modela el problema, y por las preferencias del programador.

En nuestro sistema, una aplicación podrá desarrollarse utilizando múltiples lenguajes de programación, sin necesidad de utilizar una capa de intercomunicación entre sus distintos módulos. La programación del sistema no tiene por qué llevarse a cabo mediante un lenguaje conocido; en su código fuente podrá especificarse el lenguaje de codificación utilizado, capacitando al sistema para la ejecución de la aplicación sin conocimiento previo del lenguaje a emplear.

1.2.2 Interoperabilidad Directa

Actualmente, la utilización de una capa software intermedia es la solución más extendida para conseguir interoperabilidad entre aplicaciones desarrolladas en distintos lenguajes de programación o distintas plataformas físicas. La traducción entre modelos de componentes, *middlewares* de distribución o arquitecturas de objetos distribuidos, otorga la posibilidad de intercomunicar aplicaciones desarrolladas en distintos lenguajes de programación, sistemas operativos y plataformas. Sin embargo, la utilización de estas capas adicionales conlleva determinados inconvenientes: una mayor complejidad a la hora de des-

arrollar aplicaciones, múltiples traducciones entre distintos modelos computacionales, dependencia de los mecanismos utilizados (acoplamiento), y reducción de la mantenibilidad ante futuros cambios.

Para nuestro sistema, la interoperabilidad entre las distintas capas de una aplicación, así como entre distintas aplicaciones dentro del sistema, ha de llevarse a cabo sin necesidad de utilizar de un software intermedio de intercomunicación, de forma independiente al lenguaje de programación y plataforma utilizados.

El sistema ha de ofrecer un conocimiento dinámico del estado del mismo, ofreciendo los servicios de cualquier aplicación que esté ejecutándose.

1.2.3 Extensibilidad

Los sistemas computacionales comunes amplían su nivel de abstracción de diversas maneras; ejemplos son la ampliación del lenguaje de programación, la implementación de librerías o APIs, o el desarrollo de componentes software.

La modificación del lenguaje de programación supone nuevas versiones de los procesadores del lenguaje asociado, generando una pérdida de la portabilidad del código desarrollado para versiones anteriores. La utilización de componentes, APIs o librerías, no ofrece un conocimiento dinámico de los servicios ofrecidos y carece de un sistema genérico de ampliación de su funcionalidad para el resto del sistema –en ocasiones, existe una dependencia adicional de una plataforma o lenguaje de programación.

Nuestro sistema deberá soportar un mecanismo para poder extenderse: conocer dinámicamente los servicios existentes para poder ampliar éstos, si así es requerido, obteniendo un mayor nivel de abstracción para el conjunto del entorno de programación, de un modo independiente del lenguaje y plataforma.

Una aplicación podrá hacer uso de los servicios desarrollados que extiendan el nivel de abstracción global del sistema –cualquiera que sea su lenguaje de programación y plataforma.

1.2.4 Adaptabilidad no Restrictiva

La adaptabilidad de un sistema computacional es la capacidad para poder amoldarse a requisitos surgidos dinámicamente, desconocidos en fase de desarrollo. La mayoría de los sistemas existentes ofrecen mecanismos muy limitados para poder desarrollar aplicaciones dinámicamente adaptables.

Las aplicaciones desarrolladas en nuestro entorno de programación deberán poder adaptarse a contextos surgidos dinámicamente, sin haber predicho éstos en fase de desarrollo. En tiempo de ejecución, una aplicación podrá detectar nuevos requisitos y adaptarse a éstos sin necesidad de detener su ejecución, modificar su código fuente, y reanudarla.

Tanto la estructura de la aplicación en ejecución, como la semántica del lenguaje de programación sobre el que haya sido codificada, deberán poder adaptarse sin restricción alguna. Como estudiaremos en el capítulo 5 y en el capítulo 7, determinados sistemas sólo permiten adaptar dinámicamente una parte de la estructura de sus aplicaciones, o de su lenguaje de programación.

Bajo nuestro sistema computacional, una aplicación, cualquiera que sea su lenguaje de programación, podrá ser desarrollada separando su código funcional de determinados aspectos independientes de su funcionalidad y, por tanto, reutilizables (depuración, persis-

tencia, sincronización, distribución o monitorización). Dinámicamente, los distintos aspectos del programa podrán ser examinados y modificados, sin necesidad de finalizar su ejecución.

A lo largo de esta memoria, utilizaremos el concepto de **flexibilidad** como la unión de las características de extensibilidad y adaptabilidad, propias de un sistema computacional.

Cabe mencionar, como profundizaremos en el capítulo 6, que el objetivo principal del sistema buscado es la flexibilidad del mismo, siendo ésta comúnmente contraria a su eficiencia; cuanto más flexible es un entorno computacional, menor es su eficiencia. Queda por tanto 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 de rendimientos de ejecución, podrán ser aplicadas al sistema presentado.

1.3 Organización de la Tesis

A continuación estructuramos la tesis, 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 tesis. 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.
- Para fijar la arquitectura global de nuestro sistema, así como la arquitectura de cada una de las capas que éste posee.
- Para evaluar los resultados del sistema propuesto, comparándolos con otros sistemas existentes estudiados –capítulo 16.

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 introduce el concepto de máquina abstracta, así como las ventajas generales de su utilización. Un estudio y evaluación de distintos tipos de máquinas abstractas existentes es llevado a cabo en el capítulo 4.

En el capítulo 5, se detalla el análisis de cómo determinados sistemas ofrecen flexibilidad computacional sin utilizar técnicas de reflectividad. Esta técnica, sus conceptos principales y distintas clasificaciones, son introducidos en el capítulo 6. La evaluación de múltiples sistemas reflectivos, sus aportaciones y limitaciones frente a los requisitos impuestos, son presentados en el capítulo 7.

Finalmente, en el capítulo 8, se presentan y estudian los modelos computacionales orientados a objetos que, careciendo del concepto de clase, están basados en prototipos.

1.3.3 Diseño del Sistema

En esta sección se introduce el sistema 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 9; la arquitectura de la primera capa, máquina abstracta, en el capítulo 10; el sistema reflectivo sin restricciones e independiente del lenguaje es presentado en el capítulo 11.

Detallamos los capítulos anteriores presentando el diseño de la máquina abstracta en el capítulo 12. Sobre ella, desarrollamos un entorno de programación (capítulo 13) que amplia el nivel de abstracción del sistema sin necesidad de modificar la implementación de la máquina. En el capítulo 14 se diseña de la tercera capa del sistema que otorga la flexibilidad máxima, de un modo independiente del lenguaje de programación seleccionado.

1.3.4 Aplicaciones, Evaluación, Conclusiones y Trabajo Futuro

Un conjunto de posibles aplicaciones prácticas de nuestro sistema se presenta en el capítulo 15, 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 16 bajo los requisitos establecidos al comienzo de ésta. Finalmente, en el capítulo 17, se muestran las conclusiones globales de la tesis, las principales aportaciones realizadas frente a los sistemas estudiados, y las futuras líneas de investigación y trabajo a realizar.

1.3.5 Apéndices

Como apéndices de esta memoria se presentan:

- En el apéndice A se presenta un diseño de un prototipo, intérprete de la máquina abstracta diseñada en el capítulo 12.
- Manual de usuario del sistema reflectivo no restrictivo (apéndice B), cuyo diseño fue detallado en el capítulo 14.
- 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. Se identificarán los distintos requisitos y se describirán brevemente, englobando éstos dentro de distintas categorías funcionales. Los grupos de requisitos serán los propios de la plataforma básica de computación buscada (§ 2.1), los necesarios para diseñar un entorno de programación con las características perseguidas (§ 2.2), y los distintos niveles de flexibilidad requeridos para obtener los objetivos fijados (§ 2.3). Finalmente, identificaremos el conjunto de requisitos generales de nuestro sistema desde un punto de vista global (§ 2.4).

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 la Plataforma de Computación

Para desarrollar un sistema computacional flexible y adaptable, que se pueda ejecutar en cualquier plataforma de forma independiente al lenguaje y que sus aplicaciones sean distribuibles e interoperables, la capa base de ejecución –la plataforma virtual– ha de satisfacer un elevado número de requisitos. La selección e implantación de éstos será una tarea crítica, puesto que todo el sistema será desarrollado sobre esta plataforma; si ésta no ofrece la suficiente flexibilidad, el entorno de programación no podrá ser completamente adaptable.

Identificaremos brevemente los requisitos que deberá cumplir la plataforma virtual de nuestro sistema para satisfacer los objetivos descritos en § 1.2.

2.1.1 Sistema Computacional Multiplataforma

La plataforma base de computación para nuestro sistema deberá poder ser implantada en cualquier sistema existente. Deberá ser totalmente independiente de características

propias de un microprocesador, sistema operativo o cualquier particularidad propia de un determinado sistema.

La implantación de nuestra plataforma base de computación deberá poderse integrar en cualquier sistema computacional con unos requisitos mínimos de procesamiento[1].

2.1.2 Independencia del Lenguaje de Programación

La utilización de una plataforma virtual mediante la descripción de una máquina abstracta –capítulo 3– es común en entornos distribuidos y portables. Sin embargo, la definición de estas plataformas suele ir ligada a un determinado lenguaje de programación.

La base computacional de nuestro sistema deberá ser independiente de los lenguajes de programación utilizados para acceder a ella. Mediante un proceso de compilación deberá ser factible una traducción desde diversos lenguajes de programación al código nativo de la plataforma.

2.1.3 Independencia del Problema

La descripción de la plataforma no deberá estar enfocada a la resolución de un tipo de problema. Existen implementaciones que describen una plataforma para abordar problemas como la persistencia de sistemas orientados a objetos o la intercomunicación de aplicaciones distribuidas. Este tipo de plataforma es descrito de una forma específica para la resolución del problema planteado.

Nuestra plataforma computacional deberá ser descrita con flexibilidad frente a los problemas tratados. No se trata de que se implemente una solución para la mayoría de los problemas existentes, sino que permita adaptar de una forma genérica su semántica y modelo computacional para la resolución de cualquier problema que surgiere.

2.1.4 Tamaño y Semántica Computacional Reducida

El primer requisito identificado, “Sistema Computacional Multiplataforma”, obligaba a la plataforma a ser independiente del sistema en el que se implante. Identificábamos además la necesidad de que ésta se pudiese ejecutar sobre sistemas con capacidad computacional reducida. Para que este requisito se pueda llevar a cabo, la plataforma base deberá implementarse buscando un tamaño reducido.

La semántica computacional básica de la plataforma –primitivas de computación– deberá ser lo más sencilla posible. Esta semántica es la descripción del funcionamiento propio de la máquina abstracta. Sin embargo, las primitivas operacionales, o funcionalidades propias del lenguaje de la máquina, deberán ser externas, accesibles y ampliables, de forma concreta a cada implementación[2].

Dividiendo la semántica operacional y computacional de la plataforma, conseguimos reducir al máximo la plataforma raíz para que pueda ser implantada en cualquier entorno. Si el sistema posee capacidad computacional suficiente, podrá ampliar su número de operaciones computacionales. Esta división impulsa los dos siguientes requisitos.

[1] Veremos cómo estos mínimos serán restringidos en el requisito § 2.1.4.
[2] Esta diferenciación en la semántica de la plataforma queda patente en el capítulo 12, donde se especifica la plataforma computacional básica de nuestro sistema.

2.1.5 Flexibilidad Computacional

Como definíamos en el requisito “Independencia del Problema”, la plataforma deberá ser lo suficientemente flexible para adaptarse a la resolución de cualquier problema. No ser trata de prever todos las necesidades de un sistema y resolverlas, sino de diseñar una plataforma flexible que permita adaptarse a cualquier problema.

Para ello hemos dividido este requisito de adaptabilidad en tres partes:

2.1.5.1 Semántica Operacional Abierta

Las operaciones primitivas del lenguaje de la plataforma deberán ser modificables para cada implantación. En cada máquina abstracta podremos aumentar el número de primitivas sobre las existentes para añadir un mayor nivel de abstracción al entorno. Lo conseguido es una flexibilidad en la semántica operacional del lenguaje.

2.1.5.2 Introspección y Acceso al Entorno

Una aplicación portable deberá ser capaz de conocer las funcionalidades existentes en el entorno en el que se está ejecutando. Mediante introspección, una aplicación podrá saber si las operaciones que demanda han sido desarrolladas y obrar en consecuencia. De esta forma, con una única plataforma, ofrecemos diversos niveles de abstracción y la capacidad de conocer en cuál nos encontramos.

2.1.5.3 Semántica Computacional Abierta

El mayor grado de flexibilidad de la plataforma se obtendrá gracias un mecanismo de modificación de su semántica computacional, mediante el propio lenguaje de la máquina abstracta. Permitir modificar el funcionamiento propio de la máquina, facilitará al programador modificar la interpretación de un programa sin modificar una línea de código. Además, la implementación de la máquina nunca es alterada: se modifica su funcionamiento mediante su propio lenguaje.

2.1.6 Interfaz de Acceso y Localización del Código Nativo

Cumpliendo la especificación de requisitos descritos, la base de nuestro sistema deberá ser multiplataforma y autoprogramado –aumenta su nivel de abstracción desde un modelo básico, programándose sobre su propio lenguaje. Todo el código desarrollado sobre la propia plataforma es portable, sin embargo, la implementación del intérprete de la máquina abstracta tendrá código nativo dependiente de la plataforma física en la que se ejecuta –las operaciones primitivas.

La reducida[3] parte del sistema que sea dependiente de la plataforma deberá identificarse de forma separada al resto del código y tendrá una interfaz de acceso bien definida. La migración del sistema a una plataforma física distinta, se conseguirá únicamente mediante la implementación de esta parte en la nueva plataforma y la recompilación de la máquina virtual.

2.1.7 Nivel de Abstracción del Modelo de Computación

El nivel de abstracción proporcionado por la plataforma virtual será el modelo utilizado para interconectar las distintas aplicaciones del sistema. La selección del correcto nivel

[3] Cuanto más reducido sea el tamaño de código implementado dependiente de la plataforma, más sencilla será la implementación del sistema en una plataforma distinta.

de abstracción que ofrezca el modelo de computación es una tarea difícil: Un bajo nivel de abstracción hace más compleja la interoperabilidad de aplicaciones, mientras que un elevado nivel de abstracción puede generar una plataforma dependiente del lenguaje – incumpliendo así el requisito § 2.1.2.

Una mala elección del modelo de computación de la plataforma puede dificultar la obtención de muchos de los requisitos expuestos en este capítulo.

2.2 Requisitos del Entorno de Programación

El diseño de la plataforma de computación se centra en el desarrollo de una máquina abstracta (capítulo 12) flexible (§ 2.1.5), que posea un tamaño y semántica operacional reducidos (§ 2.1.4).

Para aumentar su semántica computacional y ofrecer un mayor nivel de abstracción al programador de aplicaciones, desarrollaremos un entorno de programación con servicios propios de un sistema operativo distribuido (sin que cobre carácter de sistema operativo[4]).

Para validar su condición de flexibilidad, el entorno de programación implementará las distintas funcionalidades de un modo adaptable, de forma que puedan ser seleccionadas y modificadas dinámicamente.

2.2.1 Desarrollo de Características de Computación (Extensibilidad)

La plataforma de computación deberá ser lo suficientemente extensible como para poder desarrollar sobre ella otras características computacionales. Dadas las primitivas de la plataforma (por ejemplo, paso de mensajes, creación de objetos o evaluación de métodos) el entorno de programación deberá ser capaz de construir sobre estas funcionalidades otras de mayor nivel de abstracción o substitutivas de las existentes.

2.2.2 Selección de Características de Computación (Adaptabilidad)

Las nuevas características de computación desarrolladas en el entorno de programación deberán ser reemplazables de forma dinámica, pudiendo modificar o elegir éstas en función de las necesidades del programador. Si, por ejemplo, el paso de mensajes ha sido modificado para ser distribuido, deberá poder seleccionarse dinámicamente el protocolo de comunicaciones utilizado.

2.2.3 Identificación del Entorno de Programación

La única condición para formar parte de la plataforma distribuida es estar ejecutando una máquina virtual. Como hemos mencionado, el tamaño de ésta y el conjunto de sus primitivas es reducido, y por ello ampliamos estos con el entorno de programación.

Si un sistema físico posee poca memoria o capacidad de procesamiento, tendrá una parte mínima del entorno de programación. Por esta razón, en un sistema heterogéneo de computación, es necesario tener un mecanismo para conocer o identificar el conjunto de servicios que en el entorno de programación hayan sido instalados.

[4] El entorno de programación desarrollado busca facilitar al programador la interacción con la plataforma base. En ningún momento se trata de desarrollar un sistema operativo sobre la plataforma creada [Álvarez96].

2.2.4 Autodocumentación Real

En el requisito anterior señalábamos la posibilidad de que existiesen infinitas versiones del entorno de programación –cada una en función de los intereses y limitaciones de la unidad de procesamiento. Para conocer exactamente el entorno existente es necesario un sistema de autodocumentación.

La plataforma deberá facilitar la información de todos los objetos, métodos y atributos reales existentes de forma real y directa. Si se modifica uno de éstos, el sistema deberá ser capaz de modificar dicha documentación automáticamente; ésta nunca podrá quedar desfasada respecto a lo que en el sistema realmente exista.

2.3 Requisitos Concernientes al Grado de Flexibilidad

El principal objetivo del sistema buscado en esta tesis, es obtener sistema con un elevado grado de flexibilidad que permita constituirse como una plataforma de ejecución universal, sin necesidad de modificar la implementación de la máquina abstracta.

Desde el punto de vista de la seguridad del sistema computacional, el acceso y modificación de las distintas aplicaciones representa una técnica a restringir mediante un mecanismo de seguridad. En el desarrollo de esta tesis nos centraremos en diseñar un sistema con el grado de flexibilidad fijado en este capítulo, dejando los aspectos de seguridad como una capa adicional que restrinja el acceso a estas funcionalidades.

2.3.1 Conocimiento Dinámico del Estado del Sistema

Cualquier aplicación deberá ser capaz de conocer el estado del sistema o de otra aplicación en tiempo de ejecución. Gracias a esta característica podremos desarrollar aplicaciones dinámicamente adaptables a contextos, puesto que podrán analizarse éstos en tiempo de ejecución.

2.3.2 Modificación Estructural Dinámica

La estructura de un objeto, aplicación o de todo el entorno de programación, podrá ser modificada dinámicamente por cualquier aplicación ejecutada sobre la plataforma. El hecho de poder modificar la estructura de cualquier entidad en tiempo de ejecución, permite construir sistemas extensibles dinámicamente.

2.3.3 Modificación Dinámica del Comportamiento

La modificación dinámica del comportamiento o semántica de una aplicación, implica la variación de su comportamiento sin necesidad de modificar su estructura ni su código fuente. Un programa se ejecutará sobre un entorno variable o adaptable de forma dinámica, sin modificar la aplicación.

En este punto, el grado de adaptabilidad del sistema es máximo: toda su semántica puede variar sin necesidad de modificar la aplicación adaptada.

2.3.4 Modificación Computacional sin Restricciones

El requisito anterior justifica la necesidad de tener un mecanismo de modificación del comportamiento del sistema. Muchos sistemas poseen esta capacidad pero con un con-

junto de restricciones. En ellos, la modificación de funciones del comportamiento se lleva a cabo mediante un protocolo que restringe el conjunto de características que podrán ser modificadas en un futuro: si un protocolo no acoge la posibilidad de modificar una función, ésta no podrá ser substituida en tiempo de ejecución.

Nuestro sistema deberá implantar un sistema de flexibilidad computacional en el que cualquier funcionalidad pueda ser modificada dinámicamente, sin necesidad de especificarlo a priori. La modificación de la máquina abstracta no deberá ser contemplada para modificar el entorno en el que se ejecute una aplicación.

2.3.5 Modificación y Selección Dinámica del Lenguaje

Nuestra plataforma computacional deberá ser independiente del lenguaje de programación seleccionado, de forma que no sólo se deberá poder seleccionar cualquier lenguaje, sino que se permitirá modificar la semántica de éste dinámicamente. Cualquier aspecto del lenguaje de programación podrá ser modificado, y nuevas características podrán ser añadidas.

Distintas aplicaciones en distintos lenguajes podrán interactuar entre sí, además de permitir desarrollar una aplicación en distintos lenguajes. De forma adicional, una aplicación podrá especificar dentro de su código el lenguaje de programación utilizado antes de su ejecución, permitiéndose así su computación sin necesidad de que la especificación del lenguaje tenga que existir previamente en el sistema.

2.3.6 Interoperabilidad Directa entre Aplicaciones

La flexibilidad ofrecida por nuestro sistema debe ser totalmente dinámica, de forma que la adaptabilidad y extensibilidad pueda producirse en tiempo de ejecución. Si una aplicación está ejecutándose, y queremos personalizarla sin detenerla, la interoperabilidad entre aplicaciones cobra una especial importancia.

Desde una aplicación que se ejecute sobre la misma máquina abstracta, se deberá tener un acceso directo –sin ningún artificio o *middleware*– a todas las aplicaciones que estén corriendo sobre dicha plataforma. De esta forma, el acceso y modificación dinámicos desde una aplicación a otra es directo, y no será necesario construir una aplicación de un modo especial para que pueda ser accesible desde otros procesos.

2.4 Requisitos Generales de Sistema

En este punto definiremos brevemente todos los requisitos que debe cumplir nuestro sistema desde un punto de vista global, y que no hayan sido contemplados en los puntos anteriores (§ 2.1, § 2.2 y § 2.3).

2.4.1 Independencia del Hardware

La interfaz de acceso a la plataforma no deberá ser dependiente del hardware en el que haya sino implantado. La plataforma, y en conjunto todo el sistema, deberá poder instalarse sobre distintos sistemas hardware.

2.4.2 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.

2.4.3 Interacción con Sistemas Externos

Las aplicaciones desarrolladas sobre el sistema deberán poder interactuar con aplicaciones nativas existentes en el sistema operativo utilizado. El acceso a la plataforma deberá tener una interfaz similar para cada una de los operativos utilizados, siguiendo en cada caso un método estándar documentado de interacción entre aplicaciones.

Mediante el mecanismo estándar seleccionado, cualquier aplicación existente en un sistema podrá interactuar con la plataforma diseñada, y el código portable de nuestra plataforma podrá acceder a los recursos propios del sistema real en ejecución.

2.4.4 Interoperabilidad Uniforme

Los distintos elementos de computación distribuidos que formen el sistema interactuarán entre sí de un modo uniforme, indiferentemente del hardware, sistema operativo o lenguaje utilizado. Esta uniformidad implica utilizar un único modelo de objetos y representación de los datos en entornos heterogéneos.

2.4.5 Heterogeneidad

El sistema deberá poder implantarse en entornos heterogéneos. Para cumplir esta faceta, la base de la plataforma deberá ser de un tamaño reducido –para poder implantarla en sistemas computacionales poco avanzados–, el sistema deberá ser extensible –para poder elevar el nivel de abstracción– e introspectivo –para conocer el entorno del sistema computacional.

Cualquier elemento de computación, indiferentemente de cuál sea su poder computacional, deberá poder formar parte de la plataforma diseñada, aportando en cada caso un distinto número de recursos.

2.4.6 Sistema de Programación Único

En la mayoría de sistemas distribuidos existentes, el arquitecto o ingeniero software debe dividir una aplicación en elementos ubicados en distintos nodos computacionales. De alguna forma se debe identificar dónde se va a ejecutar cada parte de la aplicación antes de que ésta sea implantada.

En el sistema computacional buscado, el programador tendrá la posibilidad de acceder a cualquier elemento de computación de forma dinámica, y la aplicación –o una parte de ella– podrá desplazarse en función de una serie de condiciones. Para el programador, el sistema computacional es una única entidad con distintos nodos computacionales, todos ellos accesibles en cualquier momento de la ejecución de la aplicación.

2.4.7 Independencia del Lenguaje

La programación de aplicaciones en el sistema deberá poder realizarse mediante cualquier lenguaje de programación; por lo tanto, 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.

2.4.8 Flexibilidad Dinámica No Restrictiva

El sistema debe aportar una flexibilidad dinámica en tiempo de ejecución total: se deberá poder acceder y modificar una aplicación en todas sus características, así como su entorno de ejecución (el sistema). El mecanismo utilizado para obtener esta flexibilidad no debe imponer ningún tipo de restricción previa.

Sobre el sistema deberán poder desarrollarse aplicaciones mediante separación de incumbencias; se define el aspecto funcional de ésta y, una vez finalizada (e incluso en tiempo de ejecución), se añadirán otros aspectos adicionales referentes a su distribución física, persistencia o planificación de procesos. Esta separación de incumbencias o aspectos deberá poder efectuarse sin necesidad de modificar la aplicación en su nivel funcional.

2.4.9 Interoperabilidad Directa

Cada aplicación deberá poder acceder directamente a cualquier otro programa u objeto, sin necesidad de establecer un mecanismo específico de interconexión. El sistema deberá verse como un conjunto único computacional (§ 2.4.6) con distintos procesos, todos ellos interactuando entre sí. El acceso, modificación y reutilización de aplicaciones serán tareas indiferentes del proceso en el que fueron creadas.

CAPÍTULO 3:
MÁQUINAS ABSTRACTAS

En este capítulo se define el concepto de máquina abstracta y máquina virtual que estarán presentes a lo largo de toda la memoria. Analizaremos la evolución histórica que han tenido así como el objetivo principal buscado en cada una de las etapas.

Aquellos sistemas que utilicen este concepto para obtener objetivos distintos a los expuestos en § 1.2 y § 2.1, serán descritos de forma superficial. En caso contrario, se explicará la funcionalidad general y el beneficio obtenido mediante la utilización de máquinas abstractas, sin detallar los casos reales existentes.

Al finalizar el capítulo se comentará, a modo de conclusión, qué ventajas aporta el concepto de máquina abstracta a esta tesis y se establecerá una clasificación funcional para el estudio de los sistemas existentes –que se llevará a cabo en el capítulo 4.

3.1 Procesadores Computacionales

Un programa es un conjunto ordenado de instrucciones que se dan al ordenador indicándole las operaciones o tareas que se desea que realice [Cueva94]. Estos programas van a ser ejecutados, animados o interpretados por un procesador computacional.

Un procesador computacional ejecuta las instrucciones propias de un programa que accederán, examinando o modificando, a los datos pertenecientes a dicho programa. La implementación de un procesador computacional puede ser física (*hardware*) o lógica (*software*) mediante el desarrollo de otro programa.

3.1.1 Procesadores Implementados Físicamente

Un procesador físico es un intérprete de programas que ha sido desarrollado de forma física (comúnmente como un circuito integrado). Los procesadores más extendidos son los procesadores digitales síncronos. Están formados por una unidad de control, una memoria y una unidad aritmético lógica, todas ellas interconectadas entre sí [Mandado73].

Un computador es un procesador digital síncrono cuya unidad de control es un sistema secuencial síncrono que recibe desde el exterior (el programador) una secuencia de instrucciones que le indican las microoperaciones que debe realizar [Mandado73]. La secuencia de ejecución de estas operaciones es definida en la memoria describiendo un pro-

grama, pero la semántica de cada instrucción y el conjunto global existente para el procesador es invariable.

La ventaja de este tipo de procesadores frente a los lógicos es su velocidad al haber sido desarrollados físicamente. Su principal inconveniente, como decíamos en el párrafo anterior, es su inflexibilidad.

3.1.2 Procesadores Implementados Lógicamente

Un procesador software es un programa que interpreta a su vez programas de otro procesador [Cueva98]. Es aquel programa capaz de interpretar o emular el funcionamiento de un determinado procesador.

La modificación de un programa que emule a un procesador es mucho más sencilla que la modificación física de un procesador *hardware*. Esto hace que los emuladores software se utilicen, entre otras cosas, como herramientas de simulación encaminadas a la implementación física del procesador que emulan.

La principal desventaja de este tipo de procesadores frente a los procesadores *hardware* o físicos es la velocidad de ejecución. Puesto que los procesadores lógicos establecen un nivel más de computación (son ejecutados o interpretados por otro procesador), es inevitable que requieran un mayor número de computaciones para interpretar un programa que su versión hardware. Esta sobrecarga computacional se aprecia gráficamente en la Figura 3.1:

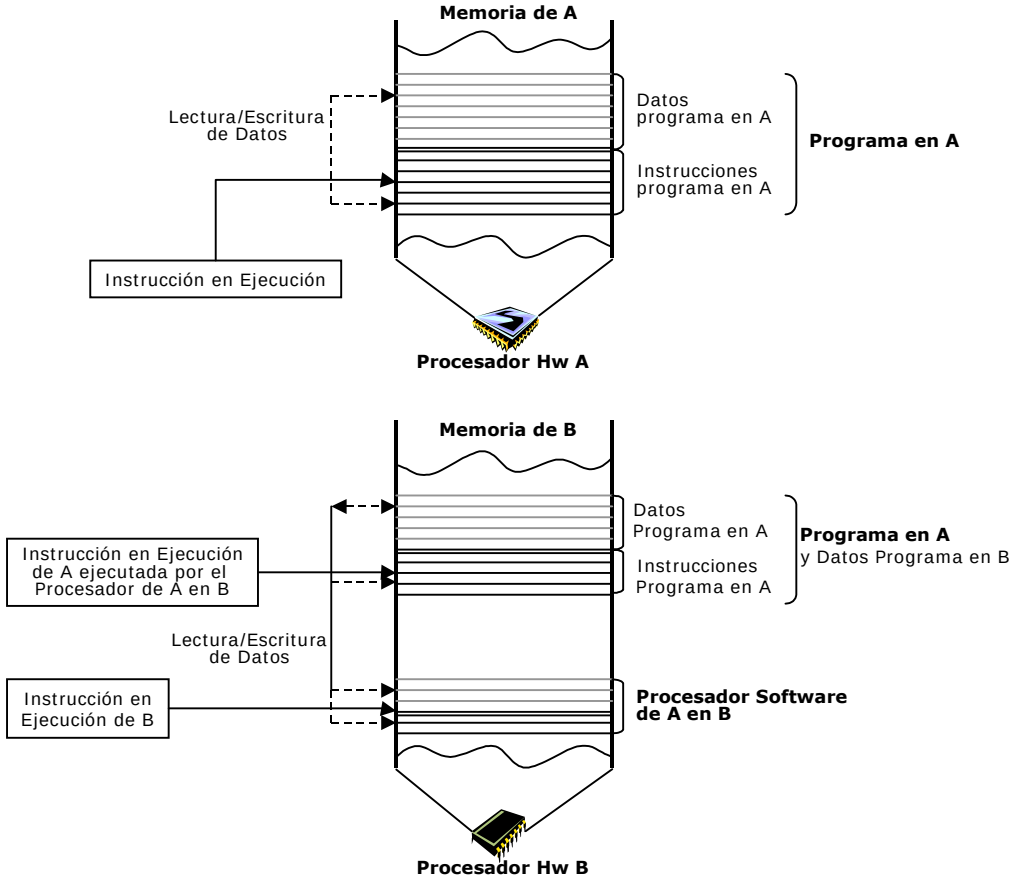


Figura 3.1: Ejecución de un procesador lógico frente a un procesador físico.

En la parte superior de la figura se muestra cómo el procesador físico **A** va interpretando las distintas instrucciones máquina. La interpretación de las instrucciones implica la lectura y/o escritura de los datos.

En el caso de interpretar a nivel *software* el programa, el procesador es a su vez un programa en otro procesador físico (procesador **B** en la Figura 3.1). Vemos cómo existe una nueva capa de computación frente al ejemplo anterior. Esto hace que se requieran más computaciones[5] o cálculos que en el primer caso.

En el segundo caso el procesador **A** es más flexible que en el primero. La modificación, eliminación o adición de una instrucción de dicho procesador, se consigue con la modificación del programa emulador. Sin embargo, el mismo proceso en el primer ejemplo, requiere la modificación del procesador a nivel físico.

3.2 Procesadores Lógicos y Máquinas Abstractas

Una máquina abstracta es el diseño de un procesador computacional sin intención de que éste sea desarrollado de forma física [Howe99]. Apoyándose en dicho diseño del procesador computacional, se especifica formalmente la semántica del juego de instrucciones de dicha máquina en función de la modificación del estado de la máquina abstracta.

Un procesador computacional desarrollado físicamente es también una máquina abstracta con una determinada implementación [Álvarez98]. Existen multitud de ejemplos de emuladores de microprocesadores físicos desarrollados como programas sobre otro procesador. Sin embargo, el nombre de máquina abstracta es empleado mayoritariamente para aquellos procesadores que no tienen una implementación física.

Un intérprete es un programa que ejecuta las instrucciones de un lenguaje que encuentra en una archivo fuente [Cueva98]. Su objetivo principal es animar la secuencia de operaciones que un programador ha especificado, en función de la descripción semántica de las instrucciones del lenguaje utilizado. Un procesador computacional –implementado física o lógicamente– es un intérprete del lenguaje que procese.

Se define máquina virtual como un intérprete de una máquina abstracta[6] [Howe99]. Una máquina virtual es por tanto un intérprete definido sobre una máquina abstracta. De esta forma, la máquina abstracta es utilizada en la descripción semántica de las instrucciones de dicho intérprete.

3.2.1 Máquina Abstracta de Estados

En teoría de lenguajes formales y autómatas, una máquina abstracta es un procedimiento para ejecutar un conjunto de instrucciones de un lenguaje formal [Howe99]. Una máquina abstracta define una estructura sobre la cuál es posible representar la semántica computacional del lenguaje formal asociado. Esta acepción es similar a la anterior, pero está más enfocada a estudios de computabilidad que a aplicaciones prácticas.

Ejemplos de máquinas abstractas son la máquina de Turing, autómatas lineales acotados, autómatas de pila y autómatas finitos, que soportan la computación de lenguajes de tipo 0, 1, 2 y 3 (lenguajes regulares) respectivamente [Cueva91].

[5] Mayor número de computaciones no implica siempre mayor tiempo de ejecución. Esta propiedad se cumpliría si la velocidad de procesamiento del computador **B** fuese igual a la del computador **A**.
[6] Aunque el concepto de máquina abstracta y máquina virtual no son exactamente idénticos, son comúnmente intercambiados.

Las máquinas abstractas de estados (*abstract state machines*) proporcionan el nexo entre los métodos formales de especificación y los modelos de computación [Huggins96]. Éstas amplían la tesis de Turing [Turing36] especificando máquinas más versátiles, capaces de simular cualquier algoritmo en su nivel de abstracción natural (no a bajo nivel) y de forma independiente al lenguaje de codificación utilizado. A partir de estos conceptos, los miembros de la comunidad ASM (*Abstract State Machine Community*) han desarrollado una metodología para describir máquinas abstractas de estados que modelen cualquier tipo de algoritmo [Huggins99].

Esta línea de investigación, que utiliza el concepto de máquina abstracta para especificar formalmente algoritmos computacionales en su nivel de abstracción natural, está fuera del estudio realizado en esta memoria. Utilizaremos el concepto de máquina abstracta para diseñar un sistema portable y flexible y no para profundizar en el campo de lenguajes formales.

3.3 Utilización del Concepto de Máquina Abstracta

En este punto realizaremos un estudio de las distintas aplicaciones prácticas encontradas al concepto de máquina abstracta –excluyendo la descrita en § 3.2.1. Especificaremos una clasificación por funcionalidades y una descripción colectiva de lo conseguido con cada utilización. En el siguiente capítulo (“Panorámica de Utilización de Máquinas Abstractas”) analizaremos cada caso particular, destacando sus aportaciones y carencias en función de los requisitos establecidos en el capítulo 2.

3.3.1 Procesadores de Lenguajes

En la implementación de compiladores ha utilizado el concepto de máquina abstracta para simplificar su diseño [Cueva94]. El proceso de compilación toma un lenguaje de alto nivel y genera un código intermedio. Este código intermedio es propio de una máquina abstracta.

Se diseña una máquina abstracta lo más general posible, de forma que se pueda traducir de ésta a cualquier máquina real existente. Para generar el código binario de una máquina real, tan sólo hay que traducir el código de la máquina abstracta a la máquina física elegida, independientemente del lenguaje de alto nivel que haya sido compilado previamente.

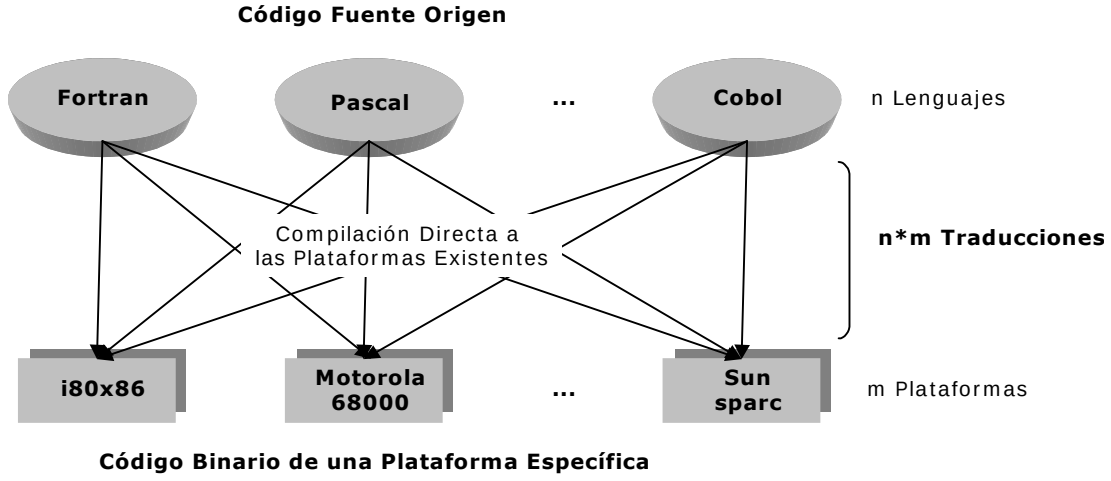


Figura 3.2: Compilación directa de n lenguajes a m plataformas.

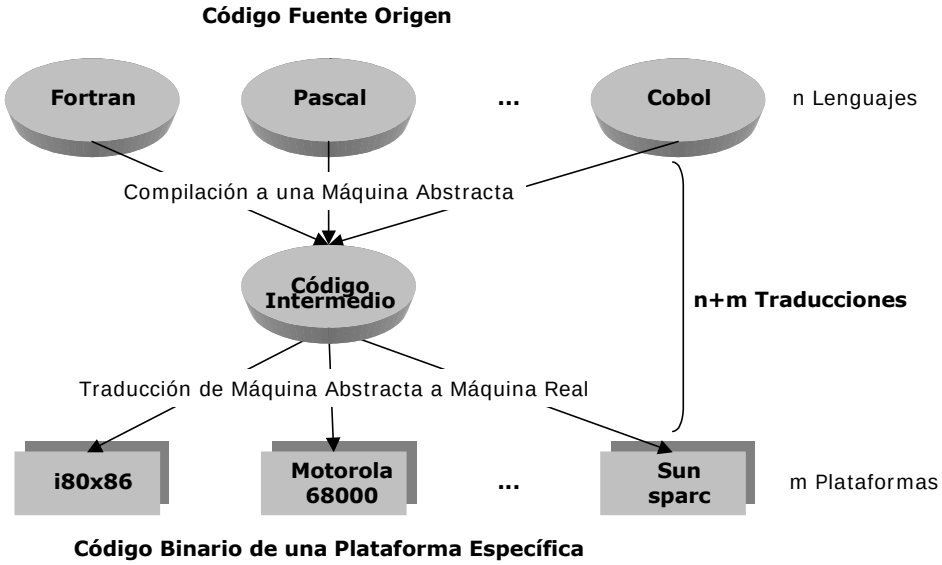


Figura 3.3: Compilación de lenguajes pasando por la generación de código intermedio.

En la Figura 3.2 y en la Figura 3.3 se observa cómo el número de traducciones y compilaciones se reduce cuando tenemos varios lenguajes fuente y varias máquinas destino existentes[7].

El proyecto UNCOL (*Universal Computer Oriented Language*) proponía un lenguaje intermedio universal para el diseño de compiladores [Steel60]. El objetivo de este proyecto era especificar una máquina abstracta universal para que los compiladores generasen código intermedio a una plataforma abierta.

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

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

Esta práctica también ha sido adoptada por varias compañías que desarrollan diversos tipos de compiladores. Un ejemplo la utilización de esta práctica son los productos de Borland/Inprise [Borland91]. Inicialmente esta compañía seleccionó un mismo "*back end*" para todos sus compiladores. Se especifica un formato binario para una máquina compartida por todas sus herramientas (archivos de extensión obj). Módulos de aplicaciones desarrolladas en distintos lenguajes como C++ y Delphi pueden enlazarse para generar una aplicación, siempre que hayan sido compiladas a una misma plataforma [Trados96].

El siguiente paso fue la especificación de una plataforma o máquina abstracta independiente del sistema operativo que permita desarrollar aplicaciones en distintos lenguajes y sistemas operativos. Este proyecto bautizado como "Proyecto Kylix" especifica un modelo de componentes CLX (*Component Library Cross-Platform*) que permite desarrollar aplicaciones en C++ Builder y Delphi para los sistemas operativos Win32 y Linux [Kozak2000].

[7] En concreto, para n lenguajes y m plataformas, se reduce el número de traducciones para n y m mayores que dos.

3.3.1.1 Entornos de Programación Multilenguaje

Apoyándose de forma directa en los conceptos de máquinas abstractas y máquinas virtuales, se han desarrollado entornos integrados de desarrollo de aplicaciones multilenguaje.

POPLOG es un entorno de programación multilenguaje enfocado al desarrollo de aplicaciones de inteligencia artificial [Smith92]. Utiliza compiladores incrementales de Common Lisp, Pop-11, Prolog y Standard ML. La capacidad de interacción entre los lenguajes reside en la traducción a una máquina abstracta de alto nivel (PVM, *Poplog Virtual Machine*) y la independencia de la plataforma física utilizada se obtiene gracias a la compilación a una máquina de bajo nivel (PIM, *Poplog Implementation Machine*) y su posterior conversión a una plataforma física –como veíamos en la Figura 3.3.

3.3.2 Portabilidad del Código

Aunque todos los procesadores *hardware* son realmente implementaciones físicas de una máquina abstracta, es común utilizar el concepto de máquina abstracta para designar la especificación de una plataforma cuyo objetivo final no es su implementación en silicio.

Probablemente la característica más explotada en la utilización de máquinas abstractas proviene de su ausencia de implementación física. El hecho de que sea un procesador *software* el que interpreta las instrucciones de la máquina, da lugar a una independencia de la plataforma física real utilizada. Una vez codificado un programa para una máquina abstracta, su ejecución podrá realizarse en cualquier plataforma[8] que posea un procesador lógico capaz de interpretar sus instrucciones, como se muestra en la siguiente figura:

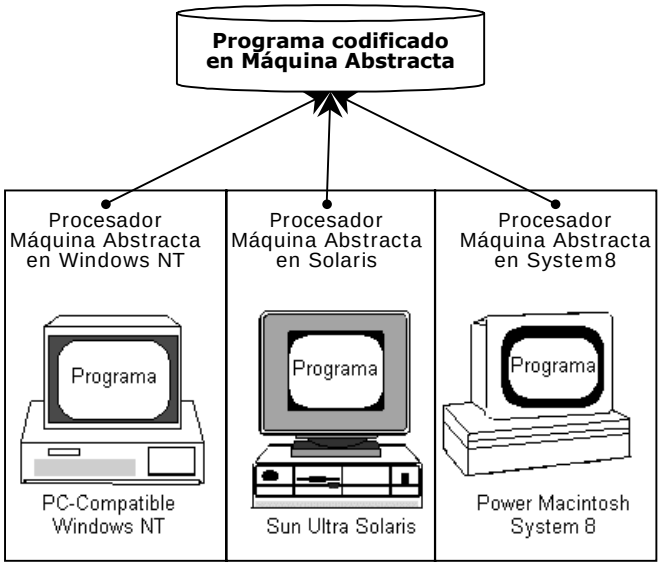


Figura 3.4: Ejecución de un programa portable sobre varias plataformas.

Al igual que un compilador de un lenguaje de alto nivel genera código para una máquina específica, la compilación a una máquina abstracta hace que ese programa generado sea independiente de la plataforma que lo ejecute. Dicho programa podrá ser ejecutado por cualquier procesador –ya sea hardware o software– que implemente la especificación computacional de un procesador de la máquina abstracta.

[8] Entendiendo por plataforma la combinación de microprocesador y sistema operativo.

Cada procesador computacional de la máquina abstracta podrá estar implementado acorde a las necesidades y características del sistema real. En la Figura 3.5 se identifican distintos grados de implementación del procesador de la máquina. En el primer ejemplo se implementa la máquina físicamente obteniendo velocidad de ejecución del programa al tener un único nivel de interpretación.

Si se implementa un procesador lógico sobre una plataforma distinta, obtenemos la portabilidad mencionada en este punto perdiendo velocidad de ejecución frente al caso anterior. Cada plataforma física que emule la máquina abstracta, implementará de forma distinta este procesador en función de sus recursos. En el ejemplo mostrado en la Figura 3.4, el procesador implementado sobre el PC con Windows NT podrá haber sido desarrollado de forma distinta a la implementación sobre el Macintosh con System8.

En la Figura 3.5 se muestra otro ejemplo en el que el emulador de la máquina abstracta es desarrollado sobre otro procesador lógico. Seguimos teniendo portabilidad del código y obtenemos una flexibilidad en el propio emulador de la máquina abstracta. Se puede ver este caso como un procesador (de **A** sobre **B**) de un procesador de la máquina abstracta (sobre **A**). Volvemos a ganar en flexibilidad a costa de aumentar en número de computaciones necesarias en la ejecución de un programa de la máquina abstracta[9].

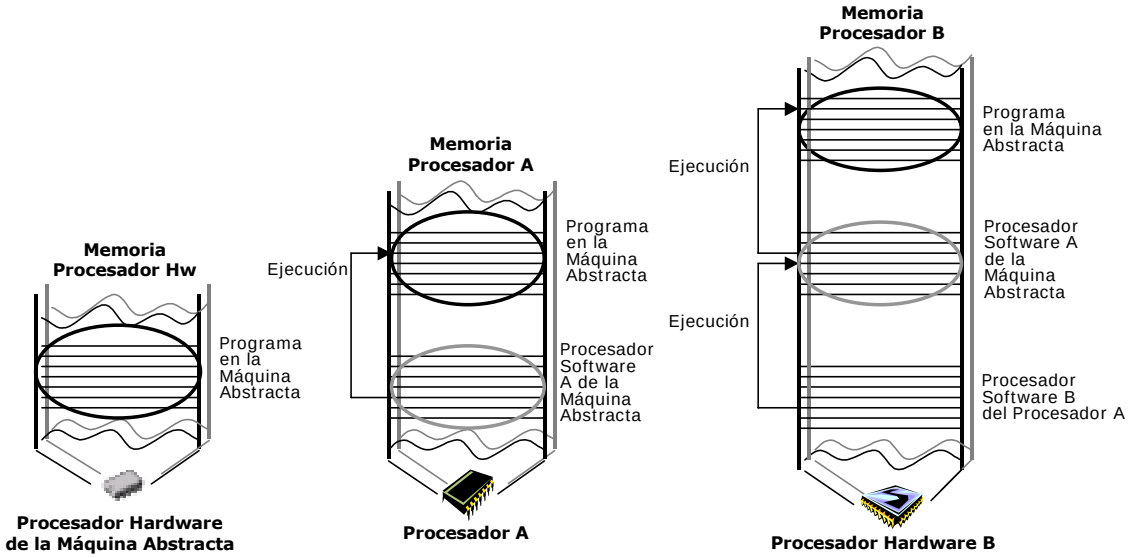


Figura 3.5: Distintos niveles de implementación de un procesador.

Existen multitud de casos prácticos que utilizan el concepto de máquina abstracta para conseguir programas portables a distintas plataformas. En el capítulo 4 estudiaremos un conjunto de éstos.

3.3.3 Sistemas Interactivos con Abstracciones Orientadas a Objetos

Con la utilización de los lenguajes orientados a objetos en la década de los 80, el nivel de abstracción en la programación de aplicaciones se elevó considerablemente [Booch94]. Sin embargo, los microprocesadores existentes se seguían basando en la ejecución de código no estructurado y los intentos de desarrollarlos con este nuevo paradigma finalizaban en fracaso por la falta de eficiencia obtenida causada por la complejidad de la implementación [Colwel88]. Posteriormente se llevaron a cabo estudios que concluyen que, haciendo uso de optimizaciones de compilación e interpretación se obtienen buenos ren-

[9] Esta estructura de procesamiento de procesamiento se utilizará posteriormente en el capítulo 11.

dimientos y, por tanto, no es rentable el desarrollo de plataformas físicas orientadas a objetos [Hölzle95].

Para desarrollar entornos de programación y sistemas que aportasen directamente la abstracción de orientación a objetos, y que fuesen totalmente interactivos, se utilizó el concepto de máquina abstracta –orientada a objetos. Los sistemas, al igual los lenguajes de programación, ofrecían la posibilidad de crear y manipular una serie de objetos. Estos objetos residían en la memoria de una máquina abstracta. Las diferencias de trabajar con una máquina abstracta en lugar de compilar a la plataforma nativa son las propias de la dualidad compilador/intérprete:

	Compilador	Intérprete
Tiempo de Ejecución	-	+
Tiempo de Compilación	+	-
Portabilidad de Código	-	+
Interacción entre Aplicaciones	-	+
Flexibilidad	-	+

Como vimos en los dos puntos anteriores, el utilizar un procesador lógico genera un mayor número de computaciones en su interpretación perdiendo tiempo de ejecución y ganando en portabilidad código. Si la plataforma que interpretamos es de un nivel más cercano al lenguaje (es orientada a objetos), los tiempos de compilación se reducirán al no producirse cambio de paradigma.

Como se muestra en Figura 3.6, el procesamiento lógico de los programas hace que todos ellos compartan una zona de memoria asociada a un proceso –el intérprete. La interacción entre aplicaciones es homogénea por tanto más sencilla de implementar que en el caso de que se cree un proceso distinto para la ejecución de cada aplicación.

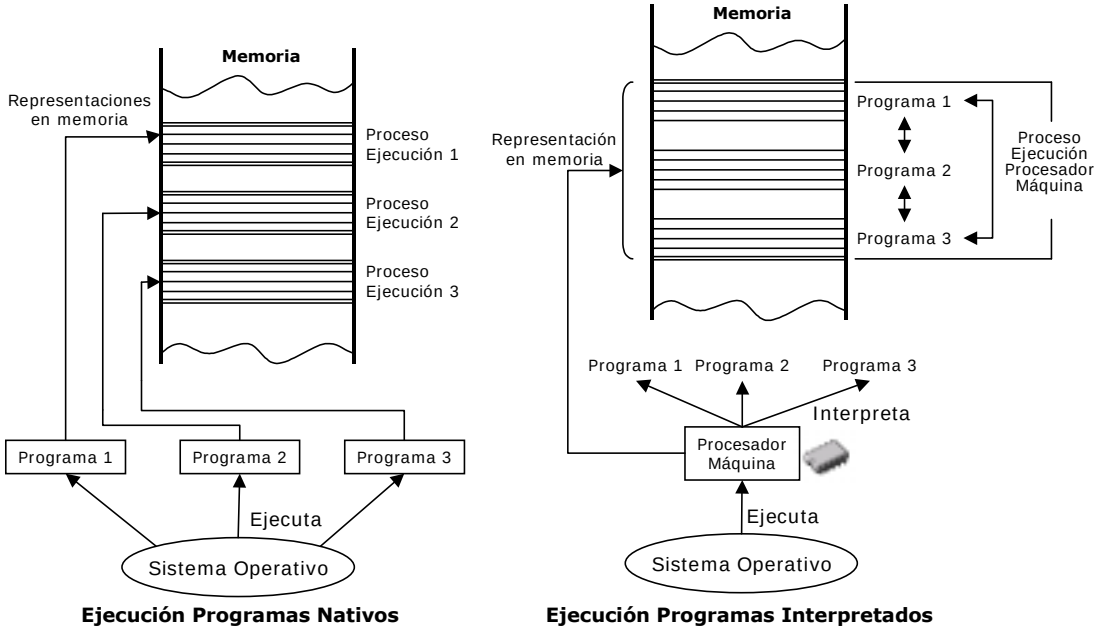


Figura 3.6: Diferencia entre la ejecución de programas nativos frente interpretados.

El hecho de que las aplicaciones puedan interactuar fácilmente y que se pueda acceder y modificar los distintos objetos existentes, aumenta la flexibilidad global del sistema pudiendo representar sus funcionalidades mediante objetos y métodos modificables.

Los sistemas desarrollados sobre una máquina abstracta orientada a objetos facilitan al usuario:

- Utilización del sistema con una abstracción más natural a la forma de pensar del ser humano [Booch94].
- Autodocumentación del sistema y de las aplicaciones. El acceso a cualquier objeto permite conocer la estructura y comportamiento de éste en cualquier momento.
- Programación interactiva y continua. Una vez que el usuario entra en el sistema, accede a un mundo interactivo de objetos [Smith95]. Puede hacer uso de cualquier objeto existente. Si necesita desarrollar una nueva funcionalidad va creando nuevos objetos, definiendo su estructura y comportamiento, comprobando su correcta funcionalidad y utilizando cualquier otro objeto existente de una forma totalmente interactiva. A partir de ese momento el sistema posee una nueva funcionalidad y un mayor número de objetos.

Dentro de este tipo de sistemas se pueden nombrar a los clásicos Smalltalk [Mevel87] y Self [Ungar87] que serán tratados con mayor profundidad en el punto § 4.3.1.

3.3.4 Distribución e Interoperabilidad de Aplicaciones

Esta ventaja en la utilización de una plataforma abstracta es una ampliación de la característica de portabilidad de código comentada previamente en § 3.3.2. El hecho de tener una aplicación codificada sobre una máquina abstracta implica que ésta podrá ejecutarse en cualquier plataforma que implemente esta máquina virtual. Además de esta portabilidad de código, la independencia de la plataforma física puede ofrecer dos ventajas adicionales:

- Una aplicación (su código) podrá ser distribuida a lo largo de una red de computadores. Los distintos módulos codificados para la máquina abstracta pueden descargarse y ejecutarse en cualquier plataforma física que implemente la máquina virtual.
- Las aplicaciones pueden interoperar entre sí (con envío y recepción de datos) de forma independiente a la plataforma física sobre la que estén ejecutándose. La representación de la información nativa de la máquina abstracta será interpretada por la máquina virtual de cada plataforma.

Aunque estos beneficios en la utilización de una máquina abstracta ya estaban presentes en Smalltalk [Goldberg83] y Self [Smith95], su mayor auge tuvo lugar con la aparición de la plataforma virtual de Java [Kramer96] –que estudiaremos en mayor profundidad en el § 4.3.1. Esta plataforma impulsó el desarrollo de aplicaciones distribuidas, especialmente a través de Internet.

3.3.4.1 Distribución de Aplicaciones

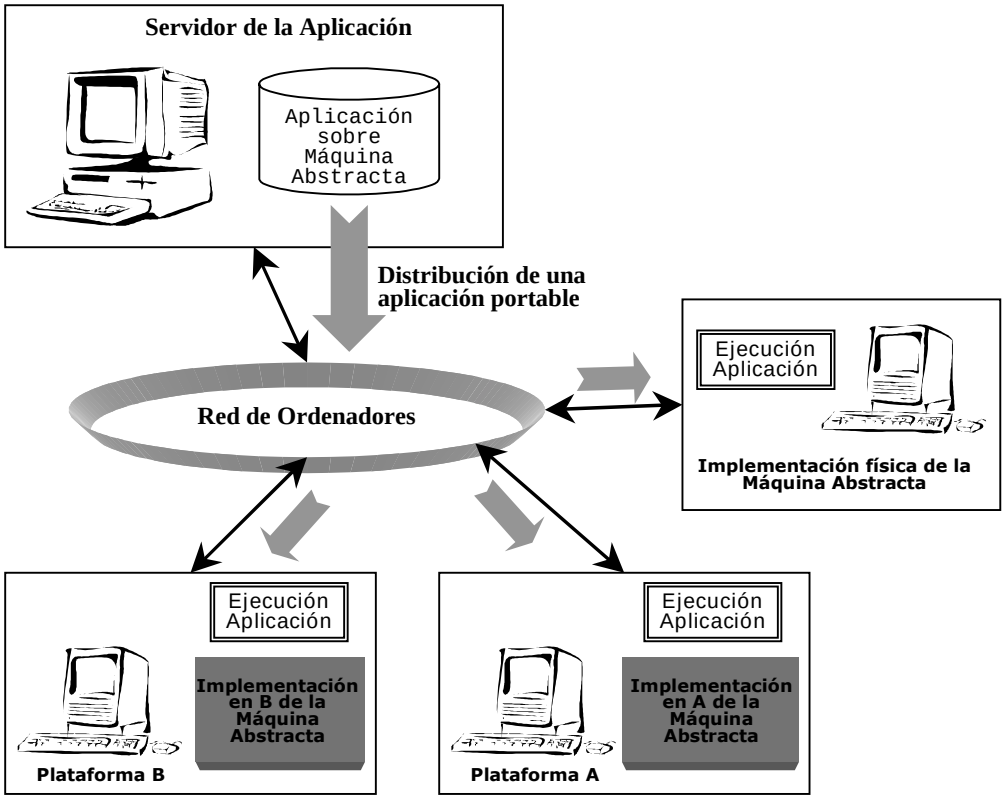


Figura 3.7: Distribución de aplicaciones portables.

En la Figura 3.7 se muestra un escenario de distribución de una aplicación desarrollada sobre una máquina abstracta. Un ordenador servidor de la aplicación, conectado mediante una red de ordenadores a un conjunto de clientes, posee el código del programa a distribuir. Mediante un determinado protocolo de comunicaciones, cada cliente demanda la aplicación del servidor, la obtiene en su ubicación y la ejecuta gracias su implementación de la máquina virtual. Esta ejecución será computacionalmente similar en todos los clientes, con el aspecto propio de la plataforma física en la que es interpretada –como mostrábamos en la Figura 3.4 de la página 20.

Un caso de uso típico del escenario mostrado es una descarga *applets* en Internet. El servidor de aplicaciones es un servidor Web. El código de la aplicación es código Java restringido denominado *applet* [Kramer96]. El cliente, mediante su navegador Web, se conecta al servidor utilizando el protocolo HTTP [Beners96] o HTTPS [Freier96] y descarga el código Java en su navegador. La aplicación es interpretada por la máquina virtual de Java [Sun95] implementada en el navegador de forma independiente a la plataforma y navegador que utilizados por el cliente.

3.3.4.2 Interoperabilidad de Aplicaciones

Uno de los mayores problemas en la intercomunicación de aplicaciones distribuidas físicamente es la representación de la información enviada. Si desarrollamos dos aplicaciones nativas sobre dos plataformas distintas y hacemos que intercambien información, deberemos establecer previamente la representación de la información utilizada. Por ejemplo, una variable entera en el lenguaje de programación C [Ritchie78] puede tener una longitud y representación binaria distinta en cada una de las plataformas.

A la complejidad de definir la representación de la información y traducir ésta a su representación nativa, se le une la tarea de definir el protocolo de comunicación: el modo en el que las aplicaciones deben dialogar para intercambiar dicha información.

Existen especificaciones estándar definidas para interconectar aplicaciones nativas sobre distintas plataformas. Un ejemplo es el complejo protocolo GIOP (*General Inter-ORB Protocol*) [OMG95] definido en CORBA [Baker97]. Establece el protocolo y representación de información necesarios para interconectar cualquier aplicación nativa a través de la arquitectura de objetos distribuidos CORBA. Este tipo de *middleware* proporciona un elevado nivel de abstracción, facilitando el desarrollo de aplicaciones distribuidas, pero conlleva una serie de inconvenientes:

- Requiere una elevada cantidad de código adicional para implementar el protocolo y la traducción de la información enviada por la red. Este código recibe el nombre de ORB (*Object Request Broker*).
- Aumenta el volumen de información enviada a través de la red de ordenadores al implementar un protocolo de propósito general que proporcione un mayor nivel de abstracción.

Si las aplicaciones se desarrollan sobre una misma máquina abstracta, el envío de la información se puede realizar directamente en el formato nativo de ésta puesto que existe una máquina virtual en toda plataforma. La traducción de la información de la máquina a la plataforma física la lleva a cabo en el intérprete de la máquina virtual. El resultado es poder interconectar aplicaciones codificadas en una máquina abstracta y ejecutadas en plataforma, y por lo tanto dispositivos, totalmente dispares.

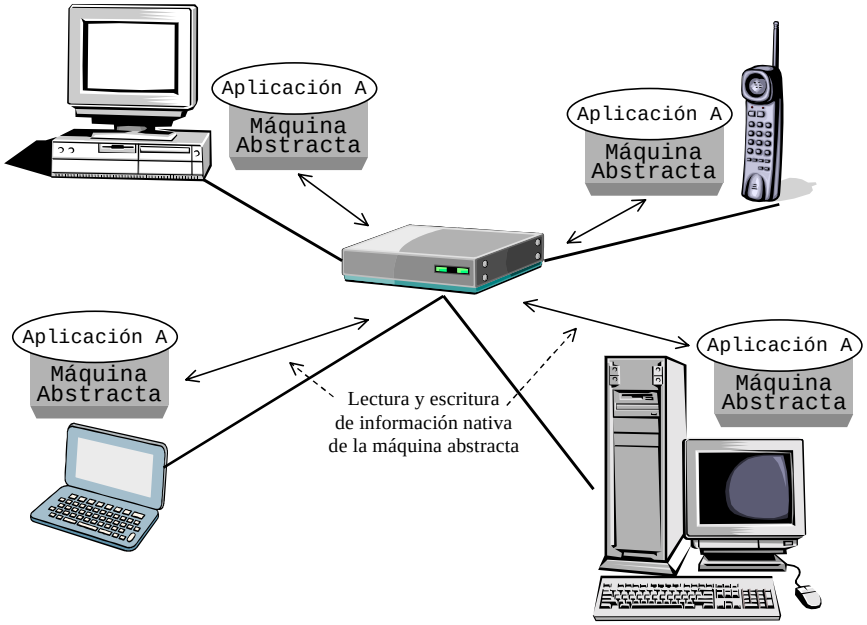


Figura 3.8: Interoperabilidad nativa de aplicaciones sobre distintas plataformas.

En el lenguaje de programación Java [Gosling96], podemos enviar los datos en su representación nativa. No estamos restringidos al envío y recepción de tipos simples de datos sino que es posible también enviar objetos, convirtiéndolos a una secuencia de bytes (*serialization*) [Campione99].

Mediante el paquete de clases ofrecidas en `java.net`, es posible implementar un protocolo propio de un tipo de aplicaciones sobre TCP/IP o UDP/IP [Raman98]. Si deseamos obtener un mayor nivel de abstracción para el desarrollo de aplicaciones distribuidas, al igual que teníamos con CORBA, podemos utilizar RMI (*Remote Method Invocation*) sin sobrecargar tanto la red de comunicaciones [Sun97e]. RMI desarrolla un protocolo de interconexión de aplicaciones Java –JRMP, *Java Remote Method Protocol*– que permite, entre otras cosas, invocar a métodos de objetos ubicados en otras máquinas virtuales.

La potencia de las aplicaciones distribuidas desarrolladas sobre la plataforma Java cobra significado cuando unimos las dos características comentadas en este punto: aplicaciones cuyo código es distribuido a través de la red, capaces de interoperar entre sí sin llevar a cabo una conversión de datos, de forma independiente a la plataforma física en la que se estén ejecutando.

3.3.5 Diseño y Coexistencia de Sistemas Operativos

La utilización del concepto de máquina abstracta ha estado presente también en el desarrollo de sistemas operativos. Podemos clasificar su utilización en función del objetivo buscado, de la siguiente forma:

- Desarrollo de sistemas operativos distribuidos y multiplataforma.
- Ejecución de aplicaciones desarrolladas sobre cualquier sistema operativo.

Existen sistemas operativos, como el VM/ESA de IBM [IBM2000], que utilizan la conjunción de ambas funcionalidades en la utilización de una máquina abstracta.

3.3.5.1 Diseño de Sistemas Operativos Distribuidos y Multiplataforma

Estos sistemas operativos aprovechan todas las ventajas de la utilización de máquinas abstractas comentadas en los puntos anteriores, para desarrollar un sistema operativo distribuido y multiplataforma. Sobre la descripción de una máquina abstracta, se implementa un intérprete de la máquina virtual en toda aquella plataforma en la que vaya a desarrollarse el sistema operativo. En el código de la máquina, se desarrollan servicios propios del sistema operativo que permitan interactuar con el sistema y elevar el nivel de abstracción; lo conseguido finalmente es:

- Una aplicación para este sistema operativo es portable a cualquier plataforma.
- Las aplicaciones no se limitan a utilizar los servicios de la máquina, sino que podrán codificarse en un mayor nivel de abstracción: el ofrecido por los servicios sistema operativo.
- El propio sistema operativo es portable, puesto que ha sido desarrollado sobre la máquina abstracta. No es necesario pues, codificar cada servicio para cada plataforma.
- La interoperabilidad de las aplicaciones es uniforme, al estar utilizando únicamente el modelo de computación de la máquina abstracta. En otros sistemas operativos es necesario especificar la interfaz exacta de acceso a sus servicios.
- Las aplicaciones desarrolladas sobre este sistema operativo pueden ser distribuidas físicamente por el sistema operativo, ya que todas serán ejecutadas por un intérprete de la misma máquina abstracta.

- En la comunicación de aplicaciones ejecutándose en computadores distribuidos físicamente, no es necesario establecer traducciones de datos. La interacción es directa, al ejecutarse todas las aplicaciones sobre la misma máquina abstracta – en distintas máquinas virtuales o intérpretes.

Existen distintos sistemas operativos desarrollados sobre una máquina abstracta ya sean comerciales, de investigación o didácticos. En el capítulo 4 analizaremos los sistemas existentes y lo que pueden aportar a esta tesis.

3.3.5.2 Coexistencia de Sistemas Operativos

En este apartado veremos la utilización del concepto de máquina virtual desde un punto de vista distinto. Este concepto puede definirse como un acceso uniforme a los recursos de una plataforma física (de una máquina real). Es una interfaz de interacción con una plataforma física que puede ser dividida en un conjunto de máquinas virtuales.

La partición de los recursos físicos de una plataforma, mediante un acceso independiente, permite la ejecución de distintos sistemas operativos inmersos en el operativo que se encuentre en ejecución. Dentro de un sistema operativo, se desarrollan tantas máquinas virtuales como sistemas inmersos deseemos tener. La ejecución de una aplicación desarrollada para un sistema operativo distinto al activo, se producirá sobre la máquina virtual implementada para el sistema operativo "huésped". Esta ejecución utilizará los recursos asignados a su máquina virtual.

La ventaja obtenida se resume en la posibilidad de ejecutar aplicaciones desarrolladas sobre cualquier sistema operativo sin tener que reiniciar el sistema, es decir, sin necesidad de cambiar el sistema operativo existente en memoria.

El principal inconveniente frente a los sistemas descritos en el punto anterior, "Diseño de Sistemas Operativos Distribuidos y Multiplataforma", es la carencia de interacción entre las aplicaciones ejecutadas sobre distintos sistemas operativos: no es posible comunicar las aplicaciones "huésped" entre sí, ni con las aplicaciones del propio operativo. Por este motivo, desecharemos la utilización de máquinas virtuales en este sentido.

El primer sistema operativo que utilizó de esta forma una máquina virtual fue el producto OS/2 de IBM. Se declaró como el sucesor de IBM del sistema operativo DOS, desarrollado para microprocesadores intel 80286. Este sistema era capaz de ejecutar en un microprocesador intel 80386 varias aplicaciones MS-DOS, Windows y propia aplicaciones gráficas nativas, haciendo uso de la implementación de distintas máquinas virtuales.

IBM abandonó el proyecto iniciado con su sistema operativo OS/2. Sin embargo, parte de sus características fueron adoptadas en el desarrollo de su operativo VM/ESA desarrollado para servidores IBM S/390 [IBM2000]. Este sistema permite ejecutar aplicaciones desarrolladas para otros sistemas operativos, utilizando una máquina virtual como modo de acceso a los recursos físicos. El empleo de una máquina virtual es aprovechado además para la portabilidad e interoperabilidad del código: cualquier grupo de aplicaciones desarrolladas sobre esta máquina abstracta puede interoperar entre sí, de forma independiente al tipo de servidor sobre el que se estén ejecutando.

Otro producto que utiliza el concepto de máquina virtual para multiplexar el acceso a una plataforma física es *VMware Virtual Platform* [Jones99]. Ejecutándose en Windows NT o en Linux permite lanzar aplicaciones codificadas para Windows 3.1, MS-DOS, Windows NT, Linux, FreeBDS y Solaris 7 para intel.

3.4 Aportación de la Utilización de Máquinas Abstractas

A lo largo de este capítulo hemos estudiado concepto de máquina abstracta y las ventajas que aporta la utilización de las mismas. De todas ellas podemos indicar que las aportaciones a nuestro sistema buscado, descrito en el capítulo 1, son:

1. Independencia del lenguaje de programación (§ 3.1.1).
2. Portabilidad de su código (§ 3.3.2).
3. Independencia de la plataforma (§ 3.3.2).
4. Elección del nivel de abstracción base para el sistema global (§ 3.3.3).
5. Interacción única entre aplicaciones, utilizando un único modelo de computación (§ 3.3.3).
6. Programación interactiva y continua (§ 3.3.3).
7. Distribución de aplicaciones (§ 3.3.4.1).
8. Interoperabilidad nativa de aplicaciones distribuidas (§ 3.3.4.2).
9. Desarrollo de servicios operativos mediante código único, portable y distribuible (§ 3.3.5.1).

Los sistemas existentes, que haciendo uso de máquinas abstractas consigan estos beneficios, serán estudiados en el próximo capítulo.

CAPÍTULO 4:
PANORÁMICA DE UTILIZACIÓN DE MÁQUINAS ABSTRACTAS

En el capítulo anterior introdujimos los conceptos de máquina abstracta y máquina virtual así como las distintas posibilidades prácticas que podían aportar a un sistema informático. Del abanico global de ventajas que ofrece su utilización, subrayamos un conjunto de ellas como adoptables al sistema propuesto en el capítulo 1.

En función de una clasificación práctica de sistemas existentes, estudiaremos las distintas implementaciones realizadas, siguiendo los requisitos establecidos en el capítulo 2, destacando los requisitos logrados y las carencias existentes.

Una vez descritos y analizados los distintos sistemas, identificaremos qué requisitos han sido cumplidos por los casos prácticos existentes y cómo adaptarlos a nuestros objetivos, así como sus insuficiencias y las necesidades surgidas para superarlas.

4.1 Portabilidad de Código

Como comentábamos en § 3.3.2, una de las ventajas más explotada en la utilización de máquinas abstractas es la portabilidad de su código. La ausencia de necesidad de una implementación física de su procesador computacional implica que el código desarrollado para esta plataforma puede ser ejecutado en cualquier sistema que implemente dicho procesador.

A continuación estudiaremos un conjunto de casos prácticos en los que se utilizaron máquinas abstractas para obtener, principalmente, portabilidad de un código generado.

4.1.1 Estudio de Sistemas Existentes

Máquina-p

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

La Universidad de California en San Diego (UCSD) desarrolló un procesador que ejecutaba código binario de la máquina-p (código-p). Adoptó la especificación de la máqui-

na abstracta para desarrollar así un proyecto de Pascal portable. Se llegó a disponer de soporte para multitarea y se desarrolló el p-System: un sistema operativo portable, codificado en Pascal y traducido a código-p [Campbell83]. La única parte del sistema que había que implementar en una plataforma concreta era el emulador de la máquina-p. El sistema se llegó a implantar en diversos microprocesadores como DEC LSI-11, Zilog Z80, Motorola 68000 e intel 8088 [Irvine99].

Al igual que los lenguajes C y Algol, el Pascal es un lenguaje orientado a bloques (orientado a marcos de pila[10], desde el punto de vista de implementación) [Jensen91] y esto hizo que el criterio fundamental en la especificación de la máquina-p fuese orientarla a una estructura de pila.

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

- El código (p-código) propio del sistema operativo (p-System).
- La pila del sistema (creciendo en sentido descendente).
- La memoria *heap* (creciendo en sentido ascendente).
- El conjunto de las pilas propias de hilos según se iban demandando en tiempo de ejecución.
- Los segmentos globales de datos (de constantes y variables).
- El intérprete o simulador de la máquina abstracta.

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

OCODE

OCODE [Richards71] fue el nombre asignado al lenguaje ensamblador de una máquina abstracta diseñada como una máquina de pila. Se utilizaba como código intermedio de un compilador de BCPL [Richards79], obteniendo portabilidad del código generado entre distintos sistemas.

BCPL (*Basic CPL*) es un lenguaje de sistemas desarrollado en 1969. Es un descendiente del lenguaje CPL (*Combined Programming Language*); se programa en un bajo nivel de abstracción, carece de tipos, está orientado a bloques y proporciona vectores de una dimensión y punteros. Es un lenguaje estructurado y posee procedimientos con paso por valor.

La ejecución de aplicaciones permite comunicarse mediante memoria compartida, donde se almacenan variables del sistema y de usuario. BCPL fue utilizado para desarrollar el sistema operativo TRIPOS, posteriormente renombrado a AmigaDOS.

Portable Scheme Interpreter

Es una implementación de un compilador del lenguaje Scheme [Abelson2000] una máquina virtual PSI (*Portable Scheme Interpreter*). La compilación del código Scheme a la máquina virtual permite ejecutar la aplicación en cualquier sistema que posea este intérprete.

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

El sistema permite añadir primitivas, depurar una aplicación sobre la máquina virtual (mediante el *Portable Scheme Debugger*) y la interpretación del código se considera "aceptable" en términos de eficiencia.

Forth

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

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

Sequential Parlog Machine

En este caso, además de la portabilidad del código, el concepto de máquina abstracta se utilizó para desarrollar un lenguaje multitarea. SPM (*Sequential Parlog Machine*) [Gregory87] es una máquina virtual del lenguaje de programación lógica Parlog [Clark83].

En § 3.3.3, estudiábamos cómo la interpretación del lenguaje para una plataforma virtual, facilita la intercomunicación de las aplicaciones en ejecución. En este caso, la máquina abstracta SPM permite implementar el lenguaje Parlog definido como "*Parallel-Prolog*" [Clark83].

Code War

La utilización de una máquina abstracta para que su código sea portable a distintas plataformas, cobra, en este caso, un carácter de originalidad respecto a los sistemas anteriores: La portabilidad del código es utilizada para crear programas que "luchen" entre sí, tratándose de eliminarse los unos a los otros.

Code War es un juego entre dos o más programas –no usuarios o jugadores– desarrollados en un lenguaje ensamblador denominado Redcode: código nativo de una máquina abstracta denominada MARS (*Memory Array Redcode Simulator*) [Dewdney88]. El objetivo del juego es desarrollar un programa que sea capaz de eliminar todos los procesos de los programas contrarios que estuvieren ejecutándose en la máquina virtual, quedando tan solo él en la memoria de la máquina.

Gracias a la utilización de una máquina virtual es posible desarrollar programas y jugar en Code War en multitud de plataformas: UNIX, IBM PC compatible, Macintosh y Amiga. Para tener una plataforma estándar de ejecución, se creo ICWS (*International Core War Society*), responsable de la creación y mantenimiento del estándar de la plataforma de Code War, así de cómo la organización de campeonatos –siendo "King of the Hill" uno de los más conocidos, accesible mediante Internet.

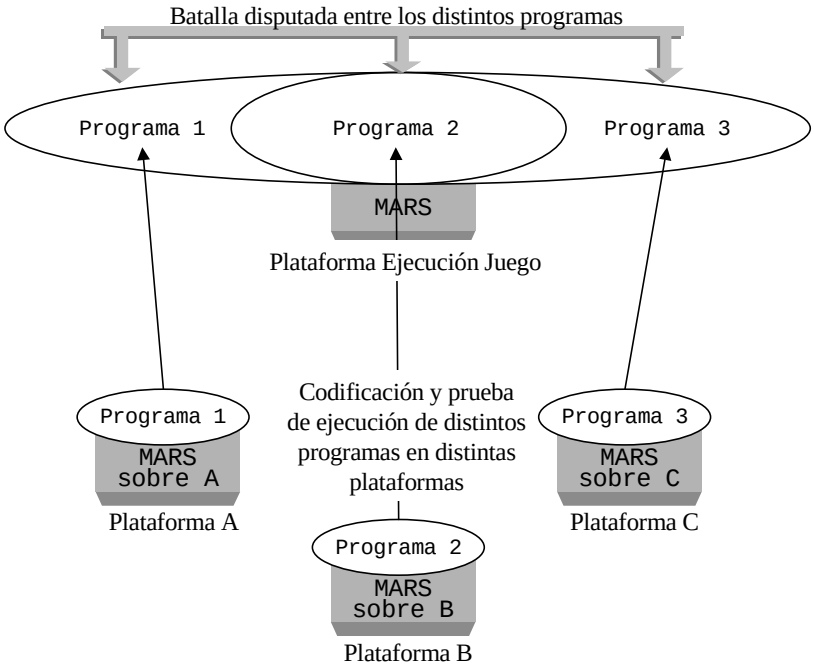


Figura 4.1: Desarrollo y ejecución de programas para Code War.

El sistema en el que los programas son ejecutados es realmente simple. El núcleo del sistema es su memoria: un vector de instrucciones inicialmente vació. El código de cada programa es interpretado de forma circular, de manera que cuando finaliza su última instrucción, se vuelve a ejecutar la primera.

La máquina virtual de MARS ejecuta una instrucción de cada programa en cada turno. Una vez evaluada la instrucción de un programa, toma otro código y ejecuta una instrucción de éste. La interpretación de cada instrucción lleva siempre el mismo tiempo, un ciclo de la máquina virtual, sea cual fuere su semántica. De esta forma, el tiempo de procesamiento es distribuido equitativamente a lo largo de todos los programas que estuvieren en memoria [Dewdney90].

Cada programa podrá tener un conjunto de procesos en ejecución. Estos procesos son almacenados por la máquina virtual en una pila de tareas. Cuando se produce el turno de ejecución de un programa, un proceso de éste será desapilado, y su siguiente instrucción será ejecutada. Los procesos que no sean destruidos durante la evaluación de su instrucción serán introducidos nuevamente en la pila de tareas.

4.1.2 Aportaciones y Carencias de los Sistemas Estudiados

Todos los sistemas estudiados en § 4.1.1 están basados principalmente en la portabilidad del código escrito para una plataforma virtual: requisito § 2.1.1. En la mayoría de los casos, los distintos lenguajes de programación son compilados a una plataforma intermedia, y ésta es interpretada en distintas plataformas, dando lugar a la portabilidad del código generado. En el caso de Code War, esta característica supone la posibilidad de jugar con un programa desarrollado y probado en cualquier entorno.

En la totalidad de los sistemas estudiados, se produce una dependencia del lenguaje de programación que se desea que sea portable. La máquina abstracta se diseña en función de un lenguaje de programación de alto nivel, restringiendo así la traducción a ella desde otros lenguajes de programación (requisito § 2.1.2).

Las plataformas son diseñadas para resolver únicamente el problema de hacer que el código de un lenguaje sea portable. En el caso de Code War, la plataforma es desarrollada para permitir establecer batallas entre programas. Esto rompe con el requisito § 2.1.3, que impone el diseño de una plataforma de forma independiente a la resolución de un único problema.

El resto de requisitos propios de la plataforma buscada, identificados en § 2.1, no han sido tenidos en cuenta en los sistemas estudiados.

4.2 Interoperabilidad de Aplicaciones

En este punto estudiaremos distintos casos prácticos en los que se establecen especificaciones computacionales, para permitir que distintas aplicaciones puedan interoperar entre sí. Si bien el concepto de máquina abstracta no es utilizado como una plataforma completa de computación, veremos cómo puede describirse como un pequeño *middleware* para interconectar distintas aplicaciones.

4.2.1 Estudio de Sistemas Existentes

Parallel Virtual Machine

Parallel Virtual Machine (PVM) es un sistema software, que permite la interacción de un conjunto heterogéneo de computadores UNIX, interconectados entre sí, pudiéndose abstraer el conjunto como un programa en una máquina multiprocesador [Geist94].

PVM fue diseñado para interconectar los recursos de distintos computadores, proporcionando al usuario una plataforma multiprocesador, capaz de ejecutar sus aplicaciones de forma independiente al número y ubicación de ordenadores utilizados.

El proyecto PVM comenzó en el verano de 1989, en el "Oak Ridge National Laboratory", donde se construyó el primer prototipo; éste fue interno al laboratorio y no se distribuyó públicamente. La segunda versión se implementó en la Universidad de Tennessee, finalizándose en marzo de 1991. A partir de esta versión, PVM comenzó a utilizarse en multitud de aplicaciones científicas. La versión 3 se desarrolló en febrero de 1993 y fue distribuida pública y gratuitamente por Internet.

PVM es un conjunto integrado de herramientas y librerías software, que emulan un entorno de programación de propósito general, flexible, homogéneo y concurrente, desarrollado sobre un conjunto heterogéneo de computadores interconectados. Los principios sobre los que se ha desarrollado, son:

- Acceso a múltiples computadores: Las tareas computacionales de una aplicación son ejecutadas en un conjunto de ordenadores. Este grupo de computadores es seleccionado por el usuario, de forma previa a la ejecución de un programa.
- Independencia del hardware: Los programas serán ejecutados sobre un entorno de procesamiento virtual. Si el usuario desea ejecutar una computación sobre una plataforma concreta, podrá asociar ésta a una máquina en particular.
- Independencia del lenguaje: Los servicios de PVM han sido desarrollados como librerías estáticas para diversos lenguajes como C [Ritchie78], C++ [Stroustrup98] o Fortran [Koelbel94].

- Computación basada en tareas: La unidad de paralelismo en PVM es una tarea: un hilo de ejecución secuencial e independiente. No existe una traducción directa de tarea a procesador: un procesador puede ejecutar diversas tareas.
- Modelo de paso de mensajes explícitos: Las distintas tareas en ejecución realizan una parte del trabajo global, y todas ellas se comunican entre sí mediante el envío explícito de mensajes.
- Entorno de computación heterogéneo: PVM soporta heterogeneidad respecto al hardware, redes de comunicación y aplicaciones.
- Soporte multiprocesador: Las plataformas multiprocesador puede desarrollar su propia implementación de la interfaz PVM, obteniendo mayor eficiencia de un modo estándar.

La arquitectura de PVM se muestra en la siguiente figura:

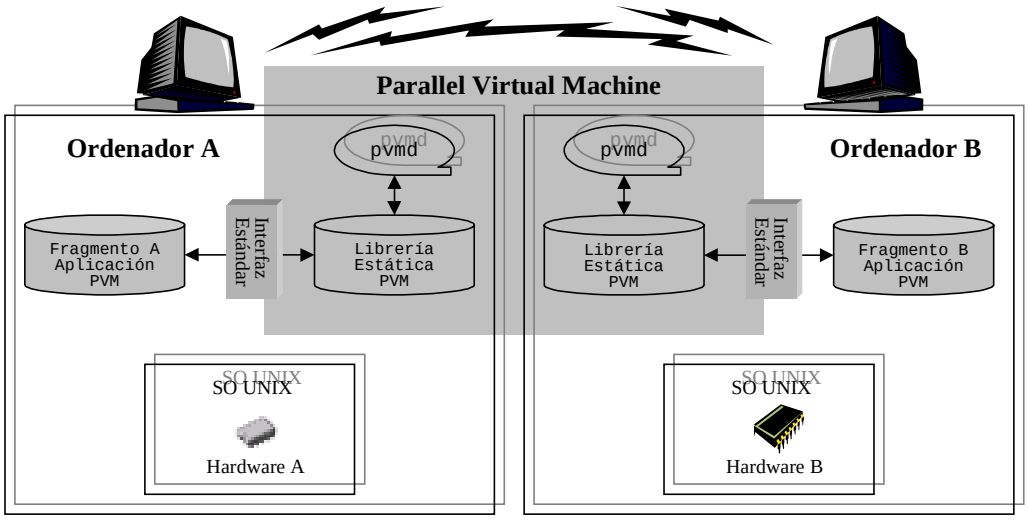


Figura 4.2: Arquitectura de PVM.

Cada plataforma UNIX que desee utilizar PVM deberá ejecutar el proceso demonio (*daemon*) de la máquina virtual. Eligiendo un lenguaje de programación, el acceso al sistema heterogéneo y distribuido se realiza mediante una interfaz de invocación a una librería estática. Esta librería ofrece un conjunto de rutinas, necesarias para la intercomunicación de las distintas tareas de una aplicación. La implementación de dichas rutinas se apoya en el proceso residente que intercomunica las distintas máquinas virtuales: pvmd (*Parallel Virtual Machine Daemon*).

Coherent Virtual Machine

Coherent Virtual Machine (CVM) es una librería de programación en C que permite al usuario acceder a un sistema de memoria compartida, para desarrollar aplicaciones distribuidas entre distintos procesadores [Keleher96]. La librería se ha desarrollado para sistemas UNIX y está enfocada principalmente para la intercomunicación de *workstations.*

CVM se ha implementado en C++ y consiste en un conjunto de clases que establecen una interfaz básica e invariable de funcionamiento; definen un protocolo de compartición de memoria flexible, un sistema sencillo de gestión de hilos y una comunicación entre aplicaciones distribuidas, desarrollado sobre UDP [Raman98].

Es posible implementar cualquier protocolo de sincronización de memoria compartida, derivando la implementación de las clases `Page` y `Protocol`. Todo funcionamiento que deseemos modificar respecto al comportamiento base, se especificará en los métodos derivados derogando su comportamiento [Booch94].

PerDiS

PerDis es un *middleware* que ofrece un entorno de programación de aplicaciones orientadas a objetos, distribuidas, persistentes y transaccionales, desarrolladas de una forma transparente, escalable y eficiente [Kloosterman98].

La creación de la plataforma PerDiS surge por la necesidad de desarrollar aplicaciones cooperativas de ingeniería en "empresas virtuales": un conjunto de compañías o departamentos que trabajan conjuntamente en la implementación de un proyecto. Las distintas partes de la empresa virtual cooperan y coordinan su trabajo mediante la compartición de datos a través de su red de ordenadores.

PerDiS ofrece un entorno distribuido de persistencia, combinando características de modelos de memoria compartida, sistemas de archivos distribuidos y bases de datos orientadas a objetos. PerDiS permite, al igual que en los entornos de programación de memoria compartida [Li89], desarrollar aplicaciones que accedan a objetos en memoria indistintamente de su ubicación. Los objetos son enviados a la memoria local de un modo transparente cuando son requeridos, haciendo que la programación distribuida sea menos explícita en cuanto a la ubicación de los objetos.

Al igual que un sistema de archivos distribuido, PerDiS almacena en el disco bloques de datos denominados *clusters*, almacenando en el cliente una caché de los mismos. Un *cluster* es un grupo de objetos que define una unidad de almacenamiento, nombrado, protección y compartición, del mismo modo que sucede en los sistemas de archivos [Sun89].

PerDiS facilita además un conjunto básico de funcionalidades propias de bases de datos orientadas a objetos [Bancilhon92] como el almacenamiento nativo y manipulación de objetos desde lenguajes orientados a objetos como C++, diferentes modelos de transacciones y tolerancia a fallos.

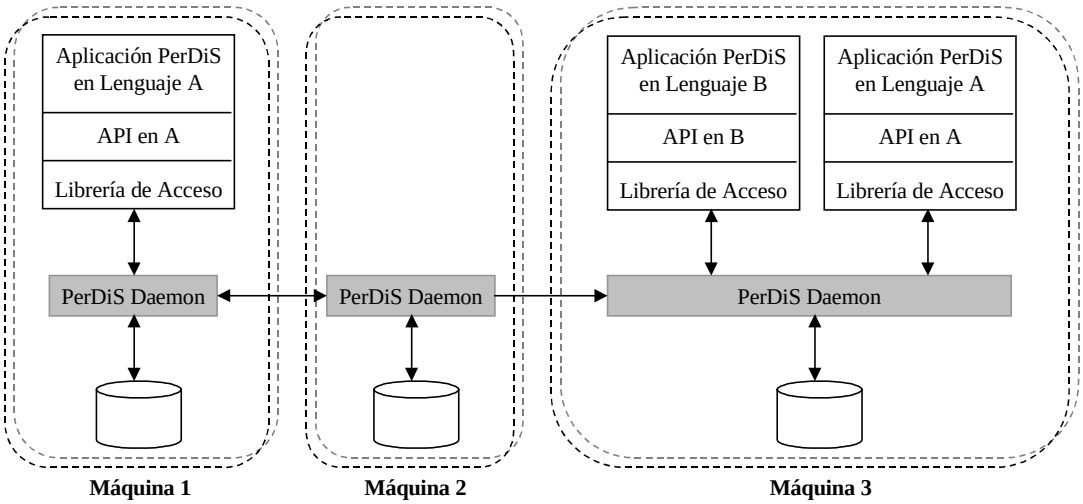


Figura 4.3: Arquitectura de la plataforma PerDiS.

La plataforma PerDiS se basa en una arquitectura cliente/servidor simétrica: cada nodo –ordenador del sistema– se comporta al mismo tiempo como cliente y servidor. La Figura 4.3 muestra los componentes propios de la arquitectura del sistema:

- Aplicación desarrollada sobre la plataforma virtual, en un lenguaje de programación.
- API (*Application Programming Interface*) de acceso a la plataforma, dependiente del lenguaje de programación.
- Una única librería, independiente del lenguaje, que sirve como nexo entre la plataforma virtual y el API utilizado para una determinada aplicación.
- El proceso demonio, "PerDiS Daemon", encargado de interconectar todos los ordenadores de la plataforma de forma independiente al lenguaje de programación, y gestor del sistema de persistencia distribuido.

Como se muestra en la Figura 4.3, el sistema es independiente del lenguaje, y en cada nodo existe parte del almacenamiento global del sistema –aunque no exista ninguna aplicación en ejecución.

4.2.2 Aportaciones y Carencias de los Sistemas Estudiados

Los tres sistemas estudiados en el punto anterior son ejemplos de sistemas que tratan de ofrecer una plataforma de computación virtual. Para ello, no definen una máquina abstracta con todos los requisitos identificados en § 2.1, sino que establecen una interfaz de acceso a cada máquina concreta.

La unión de todos los accesos desde las distintas plataformas existentes consigue la abstracción de una única plataforma virtual, susceptible de ser utilizada como una sola entidad para la resolución de un problema. Este requisito es definido como global del sistema final –en § 2.4.6; si bien busca una única entidad de programación, carece de determinadas características necesarias en ese tipo de sistema.

El principal inconveniente de los sistemas estudiados reside en que su diseño ha estado enfocado a la resolución de un determinado problema. No se trata de construir una plataforma flexible, requisito § 2.1.5, diseñada para la resolución de múltiples problemas, sino que su diseño se ha llevado a cabo bajo el establecimiento previo del problema a resolver. Van, por tanto, en contra del requisito § 2.1.3.

4.3 Plataformas Independientes

Dentro de esta clasificación, analizaremos casos prácticos de la utilización de máquinas abstractas como instrumento para conseguir plataformas independientes del lenguaje, sistema operativo y microprocesador. Los sistemas estudiados identifican una máquina abstracta como base computacional y, sobre ésta, desarrollan el conjunto de su plataforma, de modo independiente al entorno computacional real existente.

4.3.1 Estudio de Sistemas Existentes

Smalltalk-80

El sistema Smalltalk-80 tiene sus raíces en el centro de investigación de Xerox, Palo Alto. Empezó en la década de los setenta, pasando por la implementación de tres sistemas principales: Smalltalk 72, 76 y 80; el número corresponde al año en el que fueron diseñados [Krasner83].

Los esfuerzos del grupo investigador estaban orientados a la obtención de un sistema que fuese manejable por personas no informáticas. Para llegar a esto, se apostó por un sistema basado en gráficos, interfaces interactivas y visuales, y una mejora en la abstracción y flexibilidad a la hora de programar. La abstracción utilizada, más cercana al humano, era la orientación a objetos, y, respecto a la flexibilidad, se podía acceder cómodamente y en tiempo de ejecución a las clases y objetos que existían en el sistema [Mevel87].

El sistema Smalltalk-80 está dividido básicamente en dos grandes componentes [Goldberg83]:

1. La imagen virtual: Colección de objetos, instancias de clases, que proporcionan estructuras básicas de datos y control, primitivas de texto y gráficos, compiladores y manejo básico de la interfaz de usuario.
2. La máquina virtual: Intérprete de la imagen virtual y de cualquier aplicación del usuario. Dividida en:

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

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

Aunque exista una especificación formal de la máquina abstracta [Goldberg83], con la existencia de ésta no se buscó directamente una plataforma independiente, sino aprovechar determinadas características propias de un lenguaje interpretado[11]: flexibilidad del sistema y un nivel de abstracción base adecuado –orientación a objetos.

Todas las aplicaciones constituyentes del sistema de Smalltalk-80 están escritas en el propio lenguaje Smalltalk y, al ser éste interpretado, se puede acceder dinámicamente a todos los objetos existentes en tiempo de ejecución (como señalábamos en § 3.3.3). El mejor ejemplo es el *Browser* del sistema [Mevel87], mostrado en la Figura 4.4: aplicación que recorre todas las clases (los objetos derivados de `Class`) del sistema (del diccionario `Smalltalk`) y nos visualiza la información de ésta:

- Categoría a la que pertenece la clase.
- Un texto que describe la funcionalidad de ésta.
- Sus métodos.
- Sus atributos.
- Código que define el comportamiento de cada método.

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

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

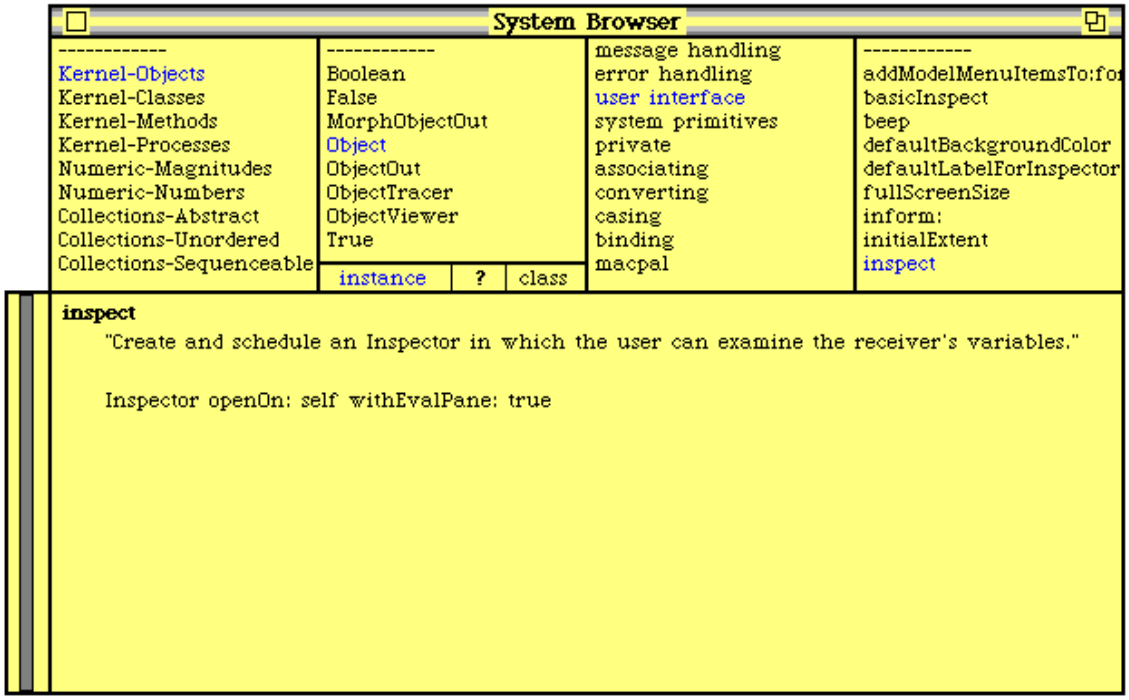


Figura 4.4: Acceso al método "inspect" del grupo "user interface", propio del objeto "Object" perteneciente al grupo "Kernel-Objects".

El aplicar estos criterios con todos los programas del sistema hace que la programación, depuración y análisis de estos sea muy sencilla. La consecución de dichos objetivos es obtenida gracias a la figura de una máquina abstracta, encargada de ejecutar el sistema.

Smalltalk-80 es pues, un sistema de computación que consigue, mediante una máquina virtual, una integración entre todas sus aplicaciones, una independencia de la plataforma utilizada y un nivel de abstracción orientado a objetos para su modelo de computación base.

Self

Tomando originalmente la plataforma y lenguaje de Smalltalk-80, se creó el proyecto Self [Ungar87]. Se especificó una máquina abstracta que reducía la representación en memoria del modelo de objetos utilizado por Smalltalk. El criterio principal de diseño fue representar el sistema sólo con objetos, eliminado el concepto de clase; este tipo de modelo de objetos se define como basado en prototipos [Evins94] –lo estudiaremos con mayor detenimiento en el capítulo 8.

La simplicidad y la pureza en los conceptos de esta máquina, hicieron que se realizasen múltiples optimizaciones en la compilación llegando a resultados interesantes –como la obtención tiempos de ejecución 50% superiores a código C compilado de forma optimizada en velocidad [Chambers91].

En la búsqueda de optimización de máquinas virtuales, se descubrieron nuevos métodos de compilación e interpretación como la compilación continua unida a la optimización adaptable [Hölze94]. Estas técnicas de compilación y optimización avanzadas han sido utilizadas en la mejora de máquinas abstractas comerciales como la de Java (The Java "HotSpot" Virtual Machine) [Sun98].

El modo de programar en Self varía respecto a los entornos clásicos. Self se basa en un sistema gráfico que permite crear aplicaciones de forma continua e interactiva. Posee un

conjunto de objetos gráficos denominados *morphs*, que facilitan la creación de interfaces, inspeccionándolos, modificándolos y volcándolos a disco. La programación se lleva a cabo accediendo, en tiempo de ejecución, a cualquier objeto existente en memoria –posea o no, representación gráfica; esta facilidad es definida como *"live editing"*. Los objetos existentes en el sistema son compartidos por todos los usuarios y aplicaciones.

Su implementación se llevó a cabo únicamente plataformas Solaris, y su utilización práctica se limitó a la investigación y análisis para la creación de otros sistemas más comerciales, como Java.

Uno de los proyectos para los que se utilizó fue el proyecto Merlin: un sistema que trataba de acercar los ordenadores a la forma de pensar de los humanos, pudiendo ser éstos fácilmente utilizables por cualquier persona [Assumpcao93]. Self fue ampliado para obtener un grado de flexibilidad mayor, ofrecido por la capacidad de modificar parte de su comportamiento mediante reflectividad computacional [Assumpcao95] –definiremos y estudiaremos este concepto en el capítulo 6. La plataforma Self modificada pasó de ser un sistema con un modelo muy básico, a ganar en flexibilidad y complejidad utilizando conceptos como "*mappings*" y "*reflectors*".

Java

En 1991, un grupo de ingenieros trabajaba en el proyecto Green: un sistema para interconectar cualquier aparato electrónico. Se buscaba poder programar cualquier aparato mediante un lenguaje sencillo, un intérprete reducido, y que el código fuese totalmente portable. Especificaron una máquina abstracta con un código binario en bytes y, como lenguaje de programación de ésta, intentaron utilizar C++ [Stroustrup98]. Se dieron cuenta de su complejidad y dependencia de la plataforma de ejecución y lo redujeron al lenguaje Oak, renombrándolo posteriormente a Java [Gosling96].

Ninguna empresa de electrodomésticos se interesó en el producto y en 1994 el proyecto había fracasado. En 1995, con el extensivo uso de Internet, desarrollaron en Java un navegador HTTP [Beners96] capaz de ejecutar aplicaciones Java en el cliente, descargadas previamente del servidor –denominándose éste HotJava [Kramer96]. A raíz de esta implementación, Netscape introdujo en su *Navigator* el emulador de la máquina abstracta permitiendo añadir computación, mediante *applets*, a las páginas estáticas HTML utilizadas en Internet [Beners93]; Java resultó mundialmente conocido.

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

- La máquina virtual de Java (*Java Virtual Machine*) [Sun95].
- La interfaz de programación de aplicaciones en Java o *core* API (*Application Programming Interface*).

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

La máquina virtual de Java es el procesador del código binario de esta plataforma. El soporte a la orientación a objetos está definido en su código binario aunque no se define

su arquitectura[12]. Por lo tanto, la implementación del intérprete y la representación de los objetos en memoria queda a disposición del diseñador del simulador, que podrá utilizar las ventajas propias de la plataforma real.

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

Una red de ordenadores tiene interconectados distintos tipos de dispositivos y ordenadores, con distintas arquitecturas hardware y sistemas operativos. Java define una máquina abstracta para conseguir implementar aplicaciones distribuidas que se ejecuten en todas las plataformas existentes en la red (plataforma independiente). De esta forma, cualquier elemento conectado a la red, que posea un procesador de la máquina virtual y el API correspondiente, será capaz de procesar una aplicación –o una parte de ésta– implementada en Java (característica estudiada previamente en § 3.3.4).

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

1. Búsqueda de una plataforma independiente.
2. Movilidad del código a lo largo de la red.
3. Especificación de un mecanismo de seguridad robusto.

La característica de definir una plataforma independiente está implícita en la especificación de una máquina abstracta. Sin embargo, para conseguir la movilidad del código a través de las redes de computadores, es necesario tener en cuenta otra cuestión: la especificación de la máquina ha de permitir obtener código de una aplicación a partir de una máquina remota (§ 3.3.4.1).

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

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

El primer módulo en la definición de la plataforma enfocado hacia su robustez, es el "*code verifier*" [Sun95]. Una vez que el código ha sido cargado, entra en fase de verificación: la máquina se asegura de que la clase esté correctamente codificada y de que no viole la integridad de la máquina abstracta [Venners98].

El sistema de seguridad en la ejecución de código, proporcionado por la máquina virtual, es ofrecido por el "*security manager*"; una aplicación cliente puede definir un *security manager* de forma que se limite, por la propia máquina virtual, el acceso a todos los recursos

[12] Si bien no define exactamente su arquitectura, supone la existencia de determinados elementos de ésta como una pila [Sun95].

que se estime oportuno[13]. Se pueden definir así los límites del software obtenido; verbigracia: la ejecución de un *applet* descargado de un servidor Web, no se le permitirá borrar archivos del disco duro.

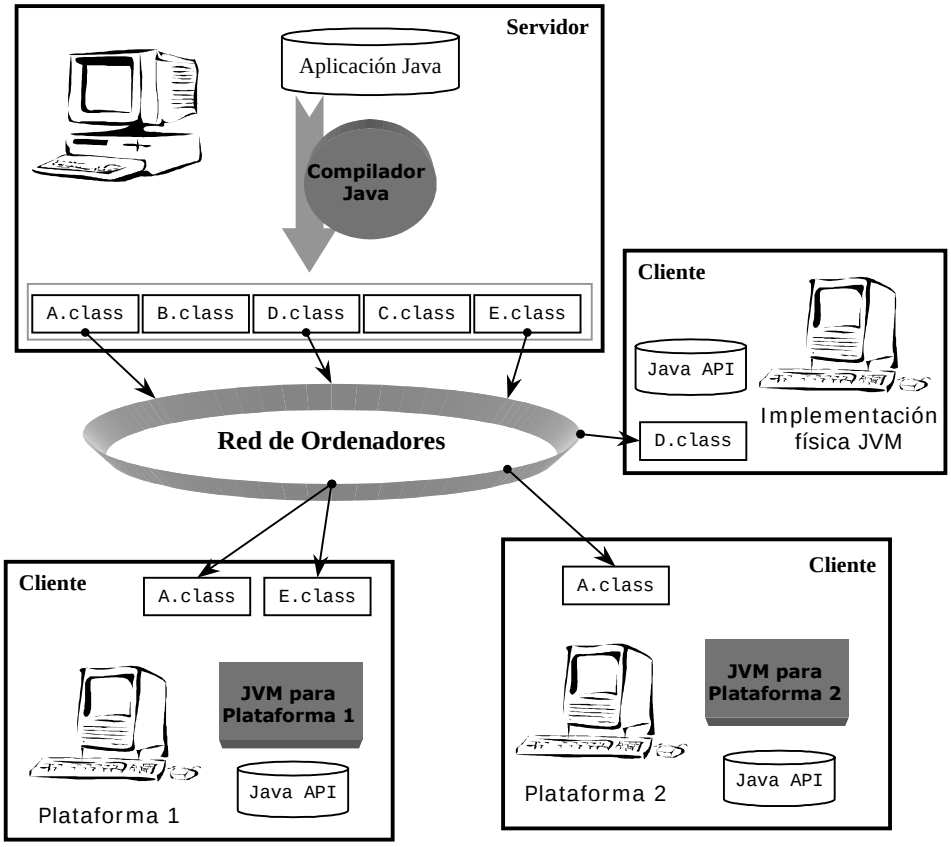


Figura 4.5: Ejemplo de entorno de programación distribuida en Java.

En la Figura 4.5 se aprecia cómo se enlazan los distintos conceptos mencionados. En un determinado equipo servidor, se diseña una aplicación y se compila al código especificado como binario de la máquina abstracta. Este código es independiente de la plataforma en la que fue compilado (característica de la utilización de máquinas abstractas, estudiada en § 3.3.2), pudiéndose interpretar en cualquier arquitectura que posea dicha máquina virtual.

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

El código obtenido puede ser "controlado" por un *security manager* definido en la aplicación cliente. De esta forma la máquina virtual comprueba los accesos a recursos no permitidos, lanzando una excepción en tiempo de ejecución si se produjese alguno [Gosling96]. El resultado es una plataforma independiente, capaz de distribuir de forma sencilla sus aplicaciones en una red de computadores y con un mecanismo de control de código correcto y seguro.

[13] Este concepto ha sido definido como caja de arena o "*sandbox*". Debido lo estricto que resultaba para el desarrollo de *applets*, en Java2 se ha hecho más flexible mediante la utilización de archivos de políticas en el "*Java Runtime Environment*" del cliente [Dageforde2000].

Aunque en la especificación de la máquina virtual de Java [Sun95] no se identifica una implementación, el comportamiento de ésta se describe en términos de subsistemas, zonas de memoria, tipos de datos e instrucciones. En la Figura 4.6 se muestran los subsistemas y zonas de memoria nombrados en la especificación.

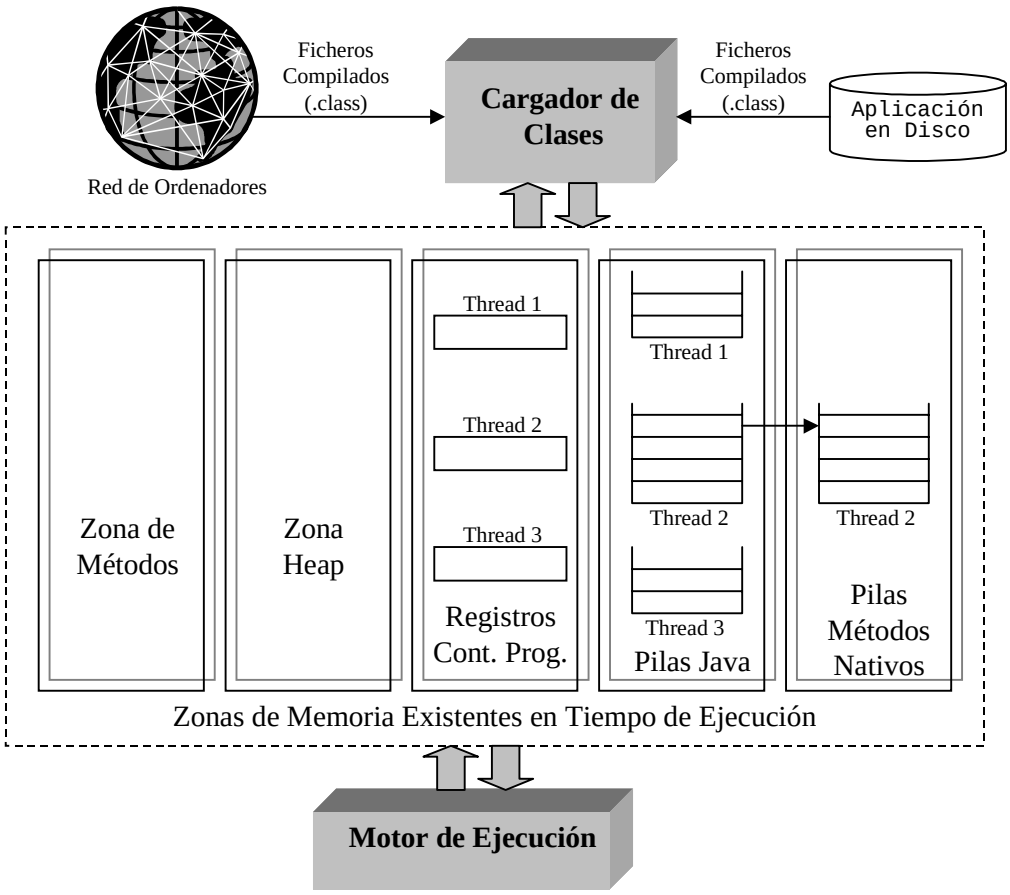


Figura 4.6: Arquitectura interna de la máquina virtual de Java.

La máquina virtual tiene un subsistema de carga de clases (*class loader*): mecanismo utilizado para cargar en memoria tipos –clases e *interfaces* [Gosling96]. La máquina también tiene un motor de ejecución (*execution engine*): mecanismo encargado de ejecutar las instrucciones existentes en los métodos de las clases cargadas [Venners98].

La máquina virtual identifica básicamente dos zonas de memoria necesarias para ejecutar un programa:

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

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

Por cada hilo en ejecución, se crea una zona de pila, un registro contador de programa y una pila de métodos nativos –métodos codificados en la plataforma real mediante el uso de una interfaz de invocaciones: JNI (*Java Native Interface*) [Sun97c]. En cada hilo se va incrementando el contador de programa por cada ejecución, se van apilando y desapi-

lando contextos o marcos en su pila, y se crea una pila de método nativo si se ejecuta un método de este tipo.

Con respecto a la implementación de la máquina virtual, comentaremos que Javasoft y Sun Microelectronics™ han desarrollado la familia de procesadores JavaChip™ [Kramer96]: picoJava™, microJava™ y ultraJava™ [Sun97]. Estos microprocesadores son implementaciones físicas de intérpretes de la máquina abstracta de Java, optimizados para las demandas de esta plataforma como son la multitarea y la recolección de basura.

The IBM J9 Virtual Machine

Como una implementación comercial de la máquina abstracta de Java, la máquina virtual J9 de IBM está enfocada al desarrollo de software empotrado [IBM2000b]. Mediante la utilización de esta máquina abstracta y el entorno de desarrollo "VisualAge Micro Edition", se puede implementar software estándar que sea independiente de las siguientes variables [IBM2000c]:

- El dispositivo hardware destino (microprocesador).
- El sistema operativo destino.
- Las herramientas utilizadas en el desarrollo de la aplicación.

"VisualAge Micro Edition" ofrece un amplio abanico de librerías de clases para ser implantadas: desde librerías de tamaño reducido para los dispositivos de capacidades reducidas, hasta amplias APIs para los entornos más potentes.

El desarrollo de una aplicación en este entorno de IBM se lleva a cabo siguiendo la arquitectura mostrada en la Figura 4.7:

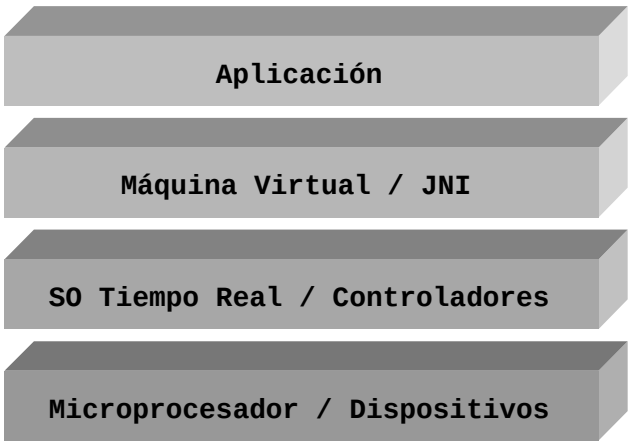


Figura 4.7: Arquitectura de aplicaciones en VisualAge Micro Edition.

Con esta arquitectura, el código de la aplicación está aislado de las peculiaridades existentes en la plataforma física destino. Mediante el "Java Native Interface" (JNI) [Sun97c], se podrá acceder directamente a los controladores de los dispositivos y a funcionalidades propias de sistemas operativos en tiempo real. Éste es el modo de asilar el código nativo del portable.

Este sistema de IBM es un ejemplo práctico de la portabilidad de código para la plataforma Java. Define un sistema estándar para la creación, ejecución y embebido de aplicaciones empotradas en hardware [IBM2000d].

.NET

.NET es la unión de una estrategia comercial de Microsoft y el diseño de una plataforma, ambos enfocados a introducir la computación de cualquier ordenador en Internet [Microsoft2000]. El objetivo principal es proporcionar una plataforma, en la que los usuarios individuales y las empresas, posean un entorno de interoperabilidad sin fisuras a través de Internet. Se podrán desarrollar y distribuir aplicaciones fácilmente mediante la red, de forma independiente a la plataforma física utilizada.

La versión final de .NET está anunciada para el año 2002. Una primera publicación de la herramienta de desarrollo "Visual Studio .NET" está disponible en el portal para desarrolladores de Microsoft, desde Julio de 2000. Toda la información relativa al desarrollo de esta plataforma es publicada en la página principal de .NET [Microsoft2000].

Con la plataforma .NET, Microsoft pasará de ser un proveedor de software a ser un proveedor de servicios [Johnston2000]; desarrollará funciones que los usuarios obtendrán mediante su conexión a Internet. Los consumidores de los servicios no seguirán el ciclo: compra, instalación y mantenimiento de cada aplicación; con .NET, comprarán una licencia de un servicio y éste se instalará y actualizará de forma automática en su máquina cliente.

Los tres servicios principales que la compañía ha identificado son: almacenamiento, autenticación y notificación. Con éstos, .NET proporcionará [Microsoft2000]:

- La facilidad de interconectar y hacer que se comuniquen entre sí distintos sistemas de computación, haciendo que la información de usuario sea actualizada y sincronizada de forma automática.
- Un mayor nivel de interacción para las aplicaciones Web, habilitando el uso de información en XML (*Extensible Markup Language*) [W3C98].
- Un servicio de suscripción en línea que permita acceder y obtener, de forma personalizada, productos y servicios del ordenador servidor.
- Un almacenamiento de datos centralizado, que aumentará la eficiencia y facilidad de acceso a la información, al mismo tiempo que los datos estarán sincronizados entre los distintos usuarios y dispositivos.
- La posibilidad de interconexión de equipos mediante distintos medios, como correo electrónico, faxes y teléfonos.
- Para los desarrolladores de software, la posibilidad de crear módulos reutilizables e independientes de la plataforma, que aumenten la productividad.

En la base de la arquitectura de la plataforma .NET (Figura 4.8), se encuentra una máquina virtual denominada "*Common Language Runtime*" que proporciona un motor de ejecución de código independiente del sistema en el que fue desarrollado, y compilado desde cualquier lenguaje de programación.

Sobre la raíz de la plataforma (*Common Runtime Language*), se ofrecerá un marco de trabajo base: código base, válido para cualquier plataforma, que puede ser utilizado desde cualquier lenguaje de programación.

El entorno de programación está basado en XML [W3C98]. A éste se podrá acceder desde una aplicación *web* –con un navegador de Internet– o bien desde una interfaz gráfica. La comunicación entre cualquier aplicación se llevará a cabo mediante mensajes codificados en XML, siendo así independiente del programa y sistema operativo emisor.

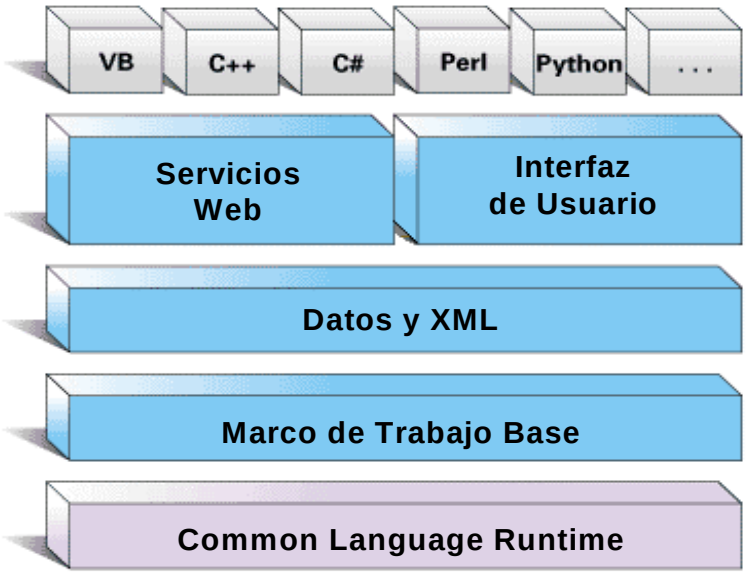


Figura 4.8: Arquitectura de la plataforma .NET.

Las ventajas propias de desarrollar aplicaciones en .NET, haciendo uso de la arquitectura mostrada en la Figura 4.8, son [Johnston2000]:

- Utilización del mismo código base en .NET, puesto que su codificación es independiente de la máquina hardware.
- La depuración de aplicaciones se realizará de forma indiferente al lenguaje de implementación utilizado. Se podrán depurar aplicaciones, incluso si están codificadas en distintos lenguajes de programación.
- Los desarrolladores podrán reutilizar cualquier clase, mediante herencia o composición, indistintamente del lenguaje empleado.
- Unificación de tipos de datos y manejo de errores (excepciones).
- La máquina virtual ejecutará código con un sistema propio de seguridad.
- Se puede examinar cualquier código con una clase, obtener sus métodos, mirar si el código está firmado, e incluso conocer su árbol de jerarquía[14].

4.3.2 Aportaciones y Carencias de los Sistemas Estudiados

En todos los sistemas estudiados, se ha obtenido un sistema de computación multiplataforma (requisito § 2.1.1). Si bien en Smalltalk no era el objetivo principal, la independencia del sistema operativo y procesador hardware ha sido buscada en el diseño de todas ellas.

Las máquinas abstractas de Smalltalk, Self y Java fueron desarrolladas para dar un soporte portable a un lenguaje de programación. Su arquitectura está pensada especialmente para trabajar con un determinado lenguaje. Sin embargo, aunque .NET está en fase de desarrollo, parece que su arquitectura no va a ser dependiente de un determinado lenguaje de programación; no obstante, parece que C# fue especificado especialmente para esta

[14] Estas facilidades se obtienen al tener una máquina abstracta dotada de introspección; estudiaremos este concepto en § 6.3.1.

plataforma [Microsoft2000b]. De esta forma, .NET es la única plataforma pensada de acuerdo con el requisito § 2.1.2.

Si bien, cuando se diseñaron las plataformas estudiadas, se trataba de obtener una serie de resultados con su utilización, ninguna se desarrolló para solucionar un tipo específico de problema. Al contrario de lo que ocurría en § 4.2, las máquinas virtuales no resuelven un problema *ad hoc*.

El tamaño de todas las plataformas estudiadas es realmente elevado si lo evaluamos con el requisito § 2.1.4. A modo de ejemplo, la máquina virtual más sencilla de las estudiadas, Self, posee más de 100.000 líneas de código C++ y 3.000 líneas de código ensamblador [Wolczko96]. Implementar una máquina virtual sobre una nueva plataforma, o en un sistema empotrado es costoso; una ratificación de esto es la distribución de la máquina virtual de Java por javasoft mediante un *plug-in*: las diversas modificaciones de la JVM y su complejidad hace que se ralentice la implementación de ésta en los navegadores de Internet, y por ello se distribuye con el sistema un *plug-in* [Javasoft99].

En cuanto a los grados de flexibilidad demandados en el requisito § 2.1.5, todas las máquinas gozan de introspección: se puede analizar cualquier objeto en tiempo de ejecución. Ninguna posee acceso al medio. Sólo en Smalltalk se pueden añadir primitivas, pero para ello hay que modificar la implementación de la máquina virtual. El mayor grado de flexibilidad lo consigue la modificación de la máquina Self, llevada a cabo en el proyecto Merlin [Assumpcao93]; mediante reflectividad computacional se pueden modificar partes del funcionamiento de la máquina (§ 2.1.5.3).

Otro de los requisitos necesarios en la búsqueda de una plataforma virtual era la identificación y existencia de un interfaz de invocaciones a codificación nativa (§ 2.1.6). La única máquina abstracta que lleva a cabo este requerimiento es Java, con su interfaz de invocaciones nativas JNI [Sun97c]. Como demostración de sus posibilidades, hemos estudiado en § 4.3 la plataforma J9 de IBM enfocada a implementar software empotrado. La interfaz empleada es un sistema de enlazado (*link*) con una librería de código C. Un inconveniente es que no se localizan los métodos de forma agrupada, sino que hay que analizar si poseen la palabra reservada `native` en su declaración.

Respecto al nivel de abstracción de su modelo de computación, exceptuando Self, todas poseen un modelo demasiado orientado al lenguaje de programación. Esto hace que la traducción desde otros sistemas sea compleja, si éstos poseen un modelo de computación diferente. Self se basa en un modelo basado en prototipos muy simple, uniforme y puede representar a multitud de lenguajes orientados a objetos [Wolczko96][15].

Finalmente, a modo de conclusión, comentar que la investigación en la creación de la plataforma .NET por Microsoft, indica, de alguna forma, que es necesario en el mundo de la computación, una solución global para el desarrollo de aplicaciones distribuidas, portables e interoperables. Actualmente existen diversas alternativas para crear este tipo de aplicaciones, pero no existe una solución global. Los objetivos buscados por Microsoft con .NET y los objetivos buscados en esta tesis, convergen en la necesidad desarrollar una nueva plataforma de computación virtual.

[15] Estudiaremos en profundidad este tipo de modelo de computación y sus posibilidades en el capítulo 8.

4.4 Desarrollo de Sistemas Operativos Distribuidos y Multiplataforma

La portabilidad del código generado para una máquina abstracta (§ 3.3.2) y la distribución e interoperabilidad de sus aplicaciones (§ 3.3.4) han sido los motivos principales para desarrollar sistemas operativos sobre máquinas abstractas. Las aplicaciones para este tipo de sistemas operativos adoptan todas las ventajas indicadas en § 3.4, y el código de las rutinas propias del sistema operativo es distribuible a lo largo de un entorno de computación heterogéneo e independientes de la plataforma.

En este apartado, estudiaremos un conjunto de sistemas operativos que hacen uso de máquinas abstractas. Nos centraremos en el diseño de las máquinas, sin entrar en profundidad en el desarrollo de los sistemas operativos. Finalmente, indicaremos las aportaciones y limitaciones de las distintas máquinas virtuales.

4.4.1 Estudio de Sistemas Existentes

Merlin

El proyecto Merlin buscaba un sistema en el que se acercase el modelo de computación de los ordenadores a la forma de pensar de los humanos, pudiendo ser éstos fácilmente utilizados por cualquier persona [Assumpcao93].

El proyecto Merlin se inició con la utilización de una simplificación de la máquina abstracta de Self [Ungar87] denominada “tinySelf” (las características de esta máquina virtual han sido descritas en § 4.3). Ante las limitaciones de la arquitectura monolítica propia de Self, desarrollaron un intérprete de este lenguaje sobre tinySelf añadiéndole reflectividad[16] [Assumpcao99]; el resultado fue el lenguaje Self/R que heredaba las ventajas de Self sumándole un grado de flexibilidad para obtener una semántica computacional abierta: es posible modificar parte de la semántica del lenguaje.

La flexibilidad obtenida para el sistema hace que esta máquina, al no ser monolítica, pueda clasificarse tanto en este apartado como en el § 4.5, “Máquinas Abstractas No Monolíticas”.

Inferno

El sistema operativo Inferno™ ha sido diseñado para ser implantado en elementos de computación heterogéneos así como en los sistemas computaciones tradicionales. Las ventajas que ofrece este sistema operativo son [Dorward97]:

- Portabilidad a distintos procesadores. Existen versiones para Intel, SPARC, MIPS, ARM, HP-PA, Power PC y AMD 29K.
- Portabilidad a distintos entornos. Puede ejecutarse tanto como un sistema operativo único para una máquina, como coexistir con otro sistema operativo como Windows NT, Windows 95, Irix, Solaris, Linux, AIX, HP/UX y NetBSD.
- Diseño distribuido. El mismo sistema es implantado tanto en la terminal del cliente como en el servidor, pudiéndose acceder siempre desde uno a los recursos del otro.

[16] Especificaremos este concepto en el capítulo 6.

- Requerimientos hardware reducidos. Inferno se ejecuta en máquinas con tan solo 1Mb de memoria y sin necesidad de que se implemente un sistema de traducción de direcciones de memoria en hardware.

- Aplicaciones portables. Las aplicaciones en Inferno se codifican en un lenguaje denominado Limbo™ [Dorward97b], cuya representación binaria es independiente de la plataforma.

- Adaptabilidad dinámica. En función del hardware existente, las aplicaciones podrán cargar un determinado módulo u otro. Se permite conocer las características de la plataforma real existente.

Todas las características que aporta Inferno se consiguen estableciendo una máquina abstracta como motor de ejecución: Dis, "*the Inferno Virtual Machine*" [Winterbottom97]. El diseño de esta máquina abstracta, al estar enfocado al desarrollo de un sistema operativo multiplataforma, se basa principalmente en dos pilares:

1. La eficiencia es un aspecto crítico para el posterior desarrollo del sistema operativo. El hecho de utilizar una máquina abstracta, ralentiza la ejecución de las rutinas del operativo; el diseño de la máquina ha de estar orientado a reducir los tiempos de ejecución al mínimo.
2. Su arquitectura debe cubrir las características generales de todas las plataformas físicas en las que va a ser implantado. Su definición no se ha llevado a cabo como un intérprete, sino como la raíz de todas las arquitecturas de los procesadores modernos.

La arquitectura de Dis no es una máquina de pila, sino que sus instrucciones son siempre de transferencia de memoria [Aho90]. La sintaxis de una instrucción para Dis es:

```
OP    src1, src2, dst
```

Donde `OP` es el código de la operación, `src1` y `src2` son operandos en memoria, y `dst` es el destino, también en memoria.

La memoria es un factor crítico al tratar de hacer el sistema lo más sencillo posible. La máquina virtual utiliza un sistema de recolección de basura mediante un simple contador de referencias [Cueva92]. Las referencias cíclicas son eliminadas con un algoritmo adicional. El resultado es un sistema de gestión de memoria sencillo y eficiente, lo suficientemente reducido como para poder implantarse en pequeños sistemas de computación.

Oviedo3

Oviedo3[17] [Cueva96] es un proyecto de investigación desarrollado por un grupo de investigación en Tecnologías Orientadas a Objetos de la Universidad de Oviedo. El objetivo del proyecto es construir un sistema computacional integral orientado a objetos. Las características que soporta son las propias de un sistema orientado a objetos [Booch94]:

- Abstracción e identidad de objetos.
- Encapsulamiento.
- Herencia.
- Polimorfismo.

[17] Oviedo Orientado a Objetos.

- Sistema de comprobación de tipos en tiempo de ejecución.
- Concurrencia.
- Persistencia.
- Distribución.

El motor computacional de todo el sistema es una máquina abstracta orientada a objetos denominada Carbayonia [Izquierdo96]. Ésta constituye el núcleo del sistema estableciendo el paradigma de la orientación a objetos desde la raíz del sistema.

El concepto básico en Oviedo3 es el objeto. Para manejar los objetos y alcanzar los requisitos inherentes al sistema, surgen otros conceptos como el paso de mensajes, la definición de métodos, la existencia de hilos, etc.

Sobre la plataforma virtual se desarrollan las distintas capas del sistema orientado a objetos (Figura 4.9). La primera capa es un conjunto de objetos que desarrollan las funciones propias de un sistema operativo: SO4[18] [Álvarez96, Álvarez97]; gestiona las tareas básicas del sistema, dando un soporte de nivel de abstracción mayor al resto de los módulos [Alvarez98].

Sobre ésta, se desarrollan compiladores y bases de datos orientadas a objetos –estos últimos se apoyan directamente en el sistema de persistencia ofrecido por la máquina virtual [Ortín97, Ortín97b] y en la noción de objeto persistente frente a tabla, cambiando la identificación de la información propia de las bases de datos relacionales [Martínez98].

Figura 4.9: Capas del Sistema Integral Orienta a Objetos Oviedo3.

El diseño de aplicaciones en esta plataforma utiliza de forma íntegra el paradigma de la orientación a objetos, reutilizando los objetos existentes en tiempo de ejecución –ya sean propios del sistema o creados por el programador.

La máquina abstracta Carbayonia está compuesta fundamentalmente por cuatro áreas mostradas en la Figura 4.10. En Carbayonia no se trabaja nunca con direcciones físicas de memoria, sino que cada área se puede considerar a su vez como un objeto: se encarga de la gestión de sus objetos, y se le envían mensajes para crearlos, obtenerlos y liberarlos. El direccionamiento de un objeto se realiza a través de una referencia.

[18] Sistema Operativo de Oviedo3.

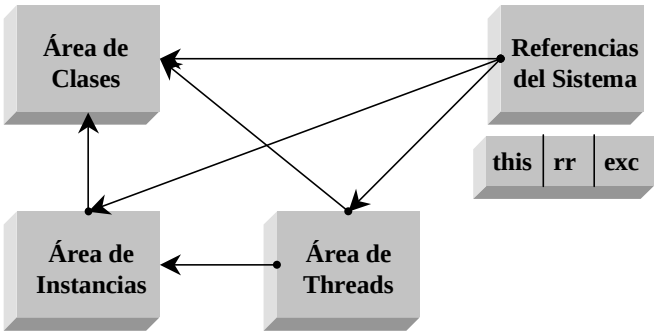


Figura 4.10: Áreas de la máquina abstracta Carbayonia.

- Área de Clases. En este área se guarda la descripción de cada clase. Esta información está compuesta por los métodos que tiene, las variables miembro que la componen y de quién deriva.
- Área de Instancias. Los objetos son ubicados en este área. Cuando se crea un objeto se deposita aquí, y cuando éste se destruye es eliminado. Se relaciona con el área de clases de manera que cada objeto puede acceder a la información de la clase a la que pertenece, conociéndose así su tipo en tiempo de ejecución.

  La única forma en la que se puede acceder a una instancia es mediante la utilización de una referencia. Mediante la referencia, podremos pasarle al objeto los mensajes pertinentes para que éste los interprete.
- Referencias del Sistema. Son una serie de referencias que posee la máquina virtual y tienen funciones específicas dentro el sistema:
  - `this`: dentro de un método, apunta al objeto implícito de la invocación.
  - `exc`: cuando se lanza una excepción, apunta al objeto que identifica la excepción lanzada.
  - `rr` (*return reference*): referencia donde los métodos asignan el valor de retorno.
- Área de hilos (*threads)*. En la ejecución de una aplicación se crea un hilo (*thread*) principal con un método inicial en la ejecución. En la ejecución de la aplicación se podrán crear más *threads*, coordinarlos y finalizarlos. Cada hilo posee una pila de contextos. El conjunto de *threads* existentes en una ejecución se almacena en este área.

4.4.2 Aportaciones y Carencias de los Sistemas Estudiados

Las plataformas estudiadas, orientadas a crear un sistema operativo, se construyen sobre una máquina abstracta multiplataforma (requisito § 2.1.1) y no dependen de un problema determinado a resolver (requisito § 2.1.3) –el diseño de un sistema operativo no está enfocado a la resolución de un tipo de problema, sino que ofrece unos servicios para que el sistema pueda ser fácilmente accesible. Exceptuando el caso de Self/R, los sistemas son también independientes del lenguaje: se puede acceder a sus servicios desde distintos lenguajes de programación, mediante una compilación previa.

En lo referente al tamaño y semántica de la máquina virtual (requisito § 2.1.4), la máquina virtual de Inferno™ es la única que posee un tamaño que permita implantarse en

pequeños sistemas de computación. El tamaño de las máquinas de Self y Oviedo3 hace, además, muy difícil su migración a distintas plataformas físicas. Ninguna separa la semántica computacional de la operacional.

Las máquinas virtuales Carbayonia, Dis y Self son monolíticas: definen un modelo estático de computación, careciendo de flexibilidad computacional (requisito § 2.1.5). No permiten modificar sus primitivas operacionales (§ 2.1.5.1) ni computacionales (§ 2.1.5.3). Dis facilita un modo de acceso al entorno (§ 2.1.5.2) y Self posee introspección para conocer el aspecto de sus objetos en tiempo de ejecución (§ 2.1.5.2.). Sin embargo, las carencias en flexibilidad de Self son eliminadas con la interpretación de este lenguaje por él mismo, consiguiendo un grado de reflectividad (ahondaremos en este concepto en el capítulo 6).

La separación del código dependiente de la plataforma, y la definición de una interfaz de acceso a éste (requisito § 2.1.6), no son definidas para ninguna máquina abstracta. A modo de ejemplo, Self/R entremezcla código Self y código ensamblador del microprocesador utilizado.

En Dis, el nivel de abstracción utilizado (requisito § **2.1.7**) es muy bajo puesto que no define un modelo computacional base (es otro tipo de código ensamblador). Carbayonia define un modelo basado en la orientación a objetos, pero aumenta demasiado la abstracción de la máquina con características como distribución, seguridad o persistencia: la demanda de una modificación en estas funcionalidades implica la modificación de la máquina virtual. Como hemos señalado en § 4.3.2, y como ampliaremos en el **capítulo 8**, el modelo computacional de Self, basado en prototipos, define una raíz computacional correcta en su abstracción, y es independiente del lenguaje.

4.5 Máquinas Abstractas No Monolíticas

Las diferentes máquinas abstractas estudiadas poseen una arquitectura monolítica: definen un sistema computacional invariable con una política y una semántica. El modo en el que llevan a cabo la computación de las aplicaciones y sus diferentes características son constantes y en ningún momento pueden modificarse. Este aspecto monolítico de las máquinas abstractas, limita la obtención del grado de flexibilidad computacional buscado (requisito § 2.1.5).

La máquina virtual de Java [Sun95], permite modificar sólo un conjunto limitado de sus características: el modo que carga las clases, mediante un `ClassLoader`, y el modo en el que restringe la ejecución de un código por motivos de seguridad, mediante un `SecurityManager` [Eckel2000]. Ésta es una solución de compromiso para determinados casos prácticos, pero no define una arquitectura flexible.

Para implementar el proyecto Merlin, se amplió la máquina virtual de Self para añadirle reflectividad [Assumpcao95]. Mediante la utilización de reflectividad podemos conseguir plataformas flexibles (como veremos en el capítulo 6). Sin embargo, en este punto estudiaremos las plataformas no monolíticas existentes sin que hagan uso de arquitecturas reflectivas.

4.5.1 Estudio de Sistemas Existentes

Virtual Virtual Machine

Actualmente existen aplicaciones con multitud de funcionalidades que llegan a implementar servicios propios de un sistema operativo; sin embargo, carecen de interoperabi-

lidad con otras aplicaciones. Un entorno computacional más orientado hacia la reutilización, debería dar soporte a un sistema que promueva la interacción entre aplicaciones (interoperabilidad interna) y que sea accesible desde cualquier lenguaje, código o representación de datos (interoperabilidad externa) [Folliot98]. Éstos fueron los objetivos buscados en el diseño de *virtual virtual machine* (VVM) [Folliot97], una plataforma multilenguaje, independiente del hardware y sistema operativo, y que es extensible y adaptable, de forma dinámica, a cualquier tipo de aplicación.

Una aplicación que se vaya a ejecutar sobre la VVM, tendrá asociado un "tipo" –el lenguaje en el que ha sido codificada. Cada "tipo" o lenguaje de programación tiene asociada una descripción para la máquina virtual denominada "VMlet". Esta descripción contiene un conjunto de reglas que describen cómo traducir la aplicación, codificada en su propio lenguaje, a la representación interna del modelo de computación de la VVM. La interoperabilidad interna del sistema (reutilización de código, independientemente del lenguaje en el que fue desarrollado) se consigue gracias a la traducción a un único modelo de computación. La arquitectura propuesta se muestra en la Figura 4.11:

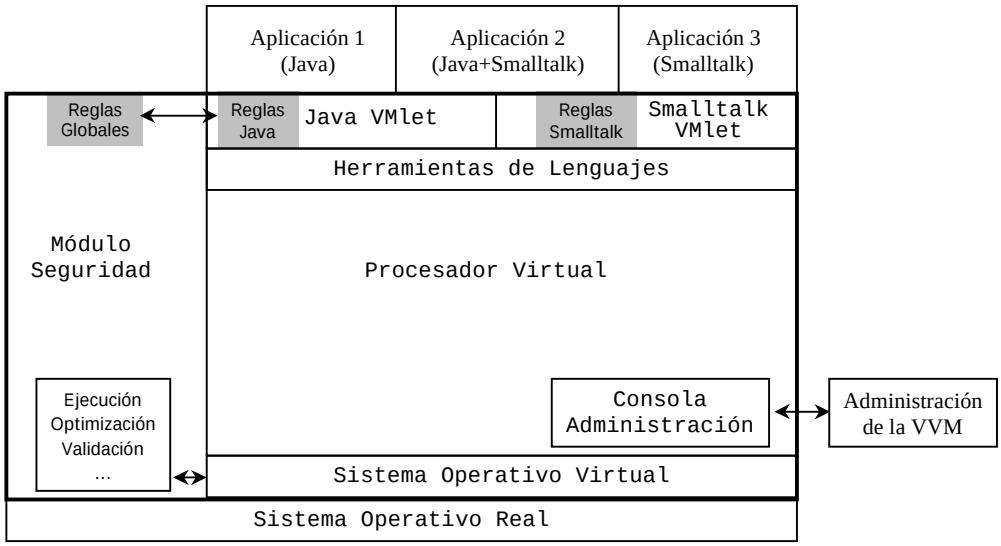


Figura 4.11: Arquitectura de Virtual Virtual Machine.

El procesador virtual es la raíz básica del modelo de computación del sistema. Ejecuta un lenguaje interno, al cuál se traduce cualquier lenguaje de programación utilizado para acceder a la plataforma. Sobre este procesador virtual se construye el sistema operativo virtual: implementa las rutinas propias de un sistema operativo, siendo éste independiente de la plataforma y del lenguaje.

Las herramientas de lenguajes son un conjunto de librerías de programación que proporcionan funcionalidades propias de un conjunto de lenguajes. Su representación también es independiente de la plataforma y lenguaje, al haber sido desarrolladas sobre el procesador virtual.

Las interoperabilidad interna se apoya en un módulo de seguridad, crítico para asegurar la integridad del sistema. Las reglas que definen la seguridad del sistema están dentro de este módulo. Las reglas propias de cada lenguaje, se encuentran en su VMlet.

Las aplicaciones codificadas para VVM son traducidas dinámicamente, utilizando su VMlet, a la representación interna de computación. Esto permite que aplicaciones desarrolladas en distintos lenguajes interactúen entre sí, y que una aplicación pueda desarrollarse en distintos lenguajes de programación.

VVM posee una consola para poder administrar y acceder a la plataforma virtual desde el sistema operativo real utilizado.

Adaptive Virtual Machine

El diseño de "*Adative Virtual Machine*" (AVM) está enfocado a encontrar una plataforma de ejecución dotada de movilidad, adaptabilidad, extensibilidad y dinamicidad. Sobre ella, las aplicaciones podrán distribuirse de forma portable, segura e interoperable. Las aplicaciones se podrán adaptar a cualquier plataforma física, y se podrá extender su funcionalidad de un modo sencillo. Además, el entorno de ejecución de una aplicación podrá ser modificado de forma dinámica [Baillarguet98].

Se identifica el desarrollo de una máquina abstracta como mecanismo para lograr los objetivos mencionados, puesto que éstas ofrecen una separación entre las aplicaciones ejecutables y el hardware sobre las que se ejecutan [Baillarguet98]. Se diseña pues una máquina abstracta adaptable (AVM) sobre la que se ofrece un entorno virtual de ejecución – *Virtual Execution Environment* (VEE). En la Figura 4.12 se muestra la arquitectura del entorno virtual de ejecución.

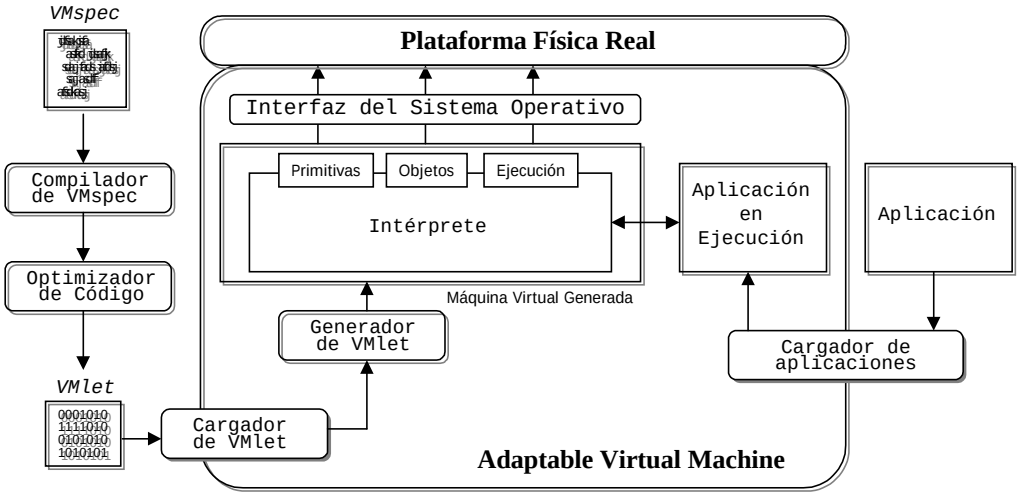


Figura 4.12: Arquitectura de la Adaptable Virtual Machine.

La AVM separa la parte estática que representa el motor de ejecución o intérprete, de la parte dinámica que define las características propias del entorno de computación en la interpretación –por ejemplo, el planificador de hilos. La unión de ambas partes supone el entorno virtual de computación (VEE): un sistema de computación adaptable.

La parte dinámica son especificaciones de máquina virtual (*VMspec*): constituyen especificaciones de alto nivel de componentes de la máquina virtual, como sistemas de persistencia o planificación de hilos. Cada *VMspec* incluye las definiciones de ejecución y un modelo de objetos, así como un conjunto de primitivas para acceder al sistema operativo. El compilador y optimizador de código transforman las *VMspecs* en una representación binaria denominada *VMlet*.

El cargador de *VMlets* carga en memoria la especificación de la máquina virtual definida, y el generador de la máquina virtual la introduce en el sistema de interpretación adaptable (AVM). Una vez concluido este proceso, la máquina virtual se comporta con unas determinadas características hasta que sean modificadas con la carga de una nueva especificación.

El intérprete de la máquina abstracta ha sido desarrollado como un *framework* orientado a objetos que produce un entorno de computación adaptable.

Extensible Virtual Machine

El hecho de implementar un motor mínimo de ejecución dotado de unas interfaces de acceso a bajo nivel, permite diseñar una plataforma de ejecución de un amplio espectro de aplicaciones, adaptable a cualquier tipo de problema. Esta es la idea básica sobre la que se diseña *Extensible Virtual Machine* (XVM) [Harris99].

La mayoría de las máquinas abstractas existentes siguen un esquema monolítico, en el que la arquitectura de éstas poseen un esquema fijo de abstracciones, funcionalidades y políticas de funcionamiento en la ejecución de sus aplicaciones. De forma contraria, un sistema extensible debería permitir a los programadores de aplicaciones, modificar el entorno de ejecución en función de las necesidades demandadas por del tipo de aplicación que se esté desarrollando. Podemos poner la gestión de memoria *heap* como ejemplo: Determinados tipos de aplicaciones hacen un uso extenso de un tamaño concreto de objetos y establecen una determinada dependencia entre éstos. Para este tipo de aplicaciones, un modo específico de creación y ubicación de objetos en la memoria *heap* implica una reducción significativa en los tiempos de ejecución [Vo96].

Sceptre es el primer prototipo desarrollado para obtener la XVM. Su diseño, orientado a la extensibilidad, utiliza una fina granularidad de operaciones primitivas, mediante las cuales se desarrollarán las aplicaciones. Su arquitectura se basa en la utilización de componentes; cada componente implementa un conjunto de operaciones. Una máquina virtual concreta se crea como composición de un conjunto de componentes. Al más bajo nivel, los componentes se implementan sobre la "*core-machine*" (motor base de ejecución), que proporciona acceso a la memoria y a la pila. Los servicios que ofrece un componente se especifican mediante una determinada interfaz denominada "*port*".

Distributed Virtual Machine

Distributed Virtual Machine (DVM) establece una arquitectura de servicios distribuidos que permite administrar y establecer un sistema global de seguridad, para un conjunto heterogéneo de sistemas de computación sobre una misma máquina virtual [Sirer99]. Con DVM, características como verificación del código, seguridad, compilación y optimización, son extraídas de los clientes (máquinas virtuales) y centralizadas en potentes servidores.

En las máquinas virtuales monolíticas, como Java [Sun95], no es posible establecer un sistema de seguridad y una administración para el conjunto de todas las máquinas virtuales en ejecución. DVM ha sido diseñada e implementada partiendo de Java, descentralizando todas aquellas partes de su arquitectura monolítica que deban estar distribuidas para un control global, al nivel de sistema.

La funcionalidad centralizada de DVM se realiza mediante un procesamiento estático (tiempo de compilación), y un análisis y gestión dinámica (en tiempo de ejecución). Este esquema es mostrado en la Figura 4.13.

Figura 4.13: Fases en la compilación y ejecución de una aplicación sobre DVM.

Los servicios estáticos son el verificador, compilador, auditor, perfilador y optimizador. Analizan la aplicación antes de su ejecución y aseguran que cumplan todas las restricciones del sistema. Los servicios dinámicos complementan la funcionalidad de los servicios estáticos, proporcionando servicios en tiempo de ejecución dependientes del contexto del cliente.

El enlace entre los servicios estáticos y dinámicos, se produce gracias a un código adicional que se inyecta en los archivos binarios, en la fase de procesamiento estático. Cuando, en el análisis estático, se encuentran operaciones que no pueden ser comprobadas en tiempo de compilación, se inserta en el código llamadas al correspondiente servicio dinámico.

Mientras que los problemas propios de una arquitectura monolítica son resueltos con DVM de una forma distribuida, ésta distribución puede llevar a costes de eficiencia en tiempo de ejecución; la transmisión de la información a través de la red, puede llegar a ser el cuello de botella de las aplicaciones en DVM. La resolución de este problema se puede conseguir mediante servidores que utilicen replicación de su información.

4.5.2 Aportaciones y Carencias de los Sistemas Estudiados

Las distintas plataformas estudiadas son multiplataforma (requisito § 2.1.1) y no dependen de forma directa de un lenguaje de programación (requisito § 2.1.2). Todas están enfocadas a desarrollar una plataforma computacional de forma independiente a un problema preestablecido (requisito § 2.1.3), aunque DVM está principalmente enfocado a la administración de una plataforma distribuida.

La flexibilidad y extensibilidad (requisito § 2.1.5) es la característica común que las agrupa. Rompen con las arquitecturas monolíticas típicas, y establecen mecanismos para modificar características del entorno de computación. Excepto XVM, todas ofrecen un mecanismo de extensibilidad basado en una arquitectura modular; se identifican las funcionalidades susceptibles de ser modificadas, y se establece una interfaz para extender éstas. XVM ofrece una expresividad de su flexibilidad basada en un lenguaje de acceso a su "*core-*

machine"; no se ofrece un conjunto de módulos variables, sino que el programador define una máquina virtual concreta, apoyándose la expresividad de un lenguaje.

El mismo diseño modular identificado en el párrafo anterior, es el culpable de los tamaños de las distintas plataformas (requisito § 2.1.4). VVM, AVM y DVM establecen un *framework* de desarrollo y ejecución flexible respecto a un conjunto de módulos. La flexibilidad modular hace que el tamaño del *framework* sea elevado, dificultando la implantación y migración de la plataforma a distintos sistemas. En XVM, el tamaño de la plataforma está en función de su nivel de abstracción, y la mayoría de su código se desarrolla sobre la máquina abstracta –siendo así totalmente portable.

En VVM y ADM existe una interfaz única de acceso al sistema operativo utilizado (requisito § 2.1.6).

4.6 Conclusiones

Hemos visto cómo inicialmente la utilización de una plataforma virtual, basada en una máquina abstracta, buscaba la portabilidad de su código binario. Los sistemas desarrollados conseguían este objetivo para un determinado lenguaje (§ 4.1.1); otras plataformas más modernas, que gozan de éxito comercial, también han sido desarrolladas para dar soporte principalmente a un lenguaje de programación (como es el caso de *Java Virtual Machine*). Uno de los objetivos a alcanzar en el diseño de una plataforma de computación flexible es que la construcción de ésta, no esté asociada a un determinado lenguaje de programación de alto nivel (requisito § 2.1.2).

La portabilidad del código ofrecida por una máquina virtual, es explotada en entornos distribuidos de computación. Si el código de una plataforma puede ser ejecutado en cualquier sistema, éste puede ser fácilmente distribuido a través de una red de ordenadores. Además, al existir un único modelo de computación y representación de datos, las aplicaciones distribuidas en un entorno heterogéneo de computación pueden intercambiar datos de forma nativa –sin necesidad de utilizar una interfaz de representación común.

Como hemos señalado en § 4.2.1, la creación de distintas máquinas abstractas ha sido utilizada para resolver problemas concretos. Sin embargo, el objetivo de esta tesis es encontrar una plataforma de computación independiente de un determinado problema (requisito § **2.1.3**).

Plataformas de diseño más avanzadas, con las ventajas ya mencionadas, han sido estudiadas en § 4.3. Estos sistemas ofrecen programación portable, distribución de código, interacción de aplicaciones distribuidas y un modelo de computación único, basado en el paradigma de la programación orientada a objetos. La implementación de aplicaciones es más homogénea y sencilla, para entornos heterogéneos distribuidos. Cabe destacar el proyecto ".NET" de Microsoft (§ 4.3.1), actualmente en fase de desarrollo, que unifica sus sistemas operativos en una única plataforma basada en una máquina abstracta.

En los sistemas analizados en el punto anterior, las distintas máquinas abstractas se han diseñado para obtener entornos de programación. Sin embargo, las características ofrecidas también se han utilizado para desarrollar sistemas operativos (§ 4.4.1). La única diferencia entre unos y otros se centra en la semántica de los servicios implementados.

El principal inconveniente de todos los sistemas mencionados es su arquitectura monolítica: se ofrece un modelo estático de computación, careciendo de flexibilidad computacional; las características computacionales de la plataforma no pueden modificarse. Un objetivo primordial en esta tesis es la obtención de flexibilidad del entorno computacional (requisito § 2.1.5). Las arquitecturas monolíticas no pueden ofrecer esta característica.

El desarrollo de plataformas no monolíticas se centra en la descomposición modular de sus características (§ 4.5). Mediante la selección dinámica de estas funcionalidades se pueden obtener distintas máquinas virtuales, específicas para un determinado tipo de problema. Sus limitaciones principales son el tamaño de los sistemas y el establecimiento a priori de las características modificables (§ 4.5.2) –algo no identificado en el diseño no se podrá modificar posteriormente.

A modo de conclusión, señalaremos la máquina virtual de Self como una plataforma con muchas características positivas: Representa un nivel de abstracción adecuado (basado únicamente en objetos), es independiente del lenguaje (en [Wolczko96] se utilizó a través distintos lenguajes, como Java y Smalltalk) y Self/R otorga una flexibilidad basada en reflectividad (desarrollada para el proyecto Merlin). Sin embargo, su desarrollo es complejo (más de 100.000 líneas de C++ y 3.000 de ensamblador), limitando su portabilidad; entrelaza código independiente de la plataforma con código ensamblador, dificultando su implementación en nuevas plataformas; finalmente, el sistema de reflectividad de Self/R es complejo y poco flexible.

CAPÍTULO 5:
SISTEMAS FLEXIBLES NO REFLECTIVOS

Buscando entornos computacionales de programación flexible, en este capítulo estudiaremos los trabajos relacionados existentes exceptuando una técnica: la reflectividad computacional –ésta será estudiada con detenimiento en el capítulo 6.

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.

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:

1. Elevada complejidad en el diseño y codificación de la aplicación.
2. 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.
3. 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.

Los sistemas a estudiar en este capítulo separan las distintas incumbencias o aspectos de una aplicación y, mediante distintos mecanismos de implementación, forman el sistema global a partir de las especificaciones realizadas. 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, para posteriormente analizar sus puntos positivos aportados y sus carencias, en función de los requisitos definidos en el capítulo 3.

5.1 Implementaciones Abiertas

La construcción de software se ha apoyado tradicionalmente en el concepto de módulo o caja negra que expone su funcionalidad ocultando su implementación. Este prin-

cipio de “caja negra” ha facilitado la división de problemas reduciendo su complejidad, y ha fomentado la portabilidad y reutilización de código. Sin embargo, el hecho de ocultar la implementación de las funcionalidades expuestas puede llevar a carencias de eficiencia en la reutilización del módulo.

La aproximación denominada “implementación abierta” (*open implementation)* trata de construir software reutilizable al mismo tiempo que eficiente [Kiczales96]. Esta técnica permite al cliente del módulo seleccionar distintas estrategias de implementación en función de sus necesidades de eficiencia. La implementación sigue estando separada de la interfaz del módulo, pero es posible parametrizarla en función de los contextos de utilización.

A modo de ejemplo, podemos pensar en la implementación de un módulo que ofrece la computación propia de un conjunto. Su interfaz poseerá operaciones como “insertar”, “eliminar”, “buscar” o “recorrer” sus elementos. Una vez implementado este módulo, lo reutilizamos en distintas aplicaciones bajo diferentes contextos, variando significativamente aspectos como:

- El número de elementos que contiene.
- La frecuencia con que se van a insertar dichos elementos.
- El número de elementos y frecuencia con que van a ser eliminados en tiempo de ejecución.

La implementación del contenedor se puede efectuar mediante distintas estructuras de datos como tablas hash, listas enlazadas o árboles. En función del contexto de utilización, la selección de una u otra estructura de datos producirá cambios en la eficiencia del sistema. De esta forma, las implementaciones abiertas describen una funcionalidad mediante una interfaz y la posibilidad de seleccionar una determinada estrategia de implementación (*ISC code*) [Kiczales96b]. Se puede expresar una implementación abierta como la función:

$$f : C \rightarrow S$$

Siendo `C` el conjunto de posibles perfiles de usuario, y `S` el conjunto de estrategias de implementación ofrecidas por el módulo. A lo largo del tiempo, el conjunto de contextos del cliente (`C`) podrá variar y, por lo tanto, deberá poder variar el abanico de estrategias de implementación (`S`). En una implementación abierta, deberemos permitir la modificación y ampliación de perfiles de usuario (`C'`), implementaciones (`S'`) y asociación entre éstas (`f'`). Un módulo abierto deberá pues proporcionar cuatro interfaces distintas:

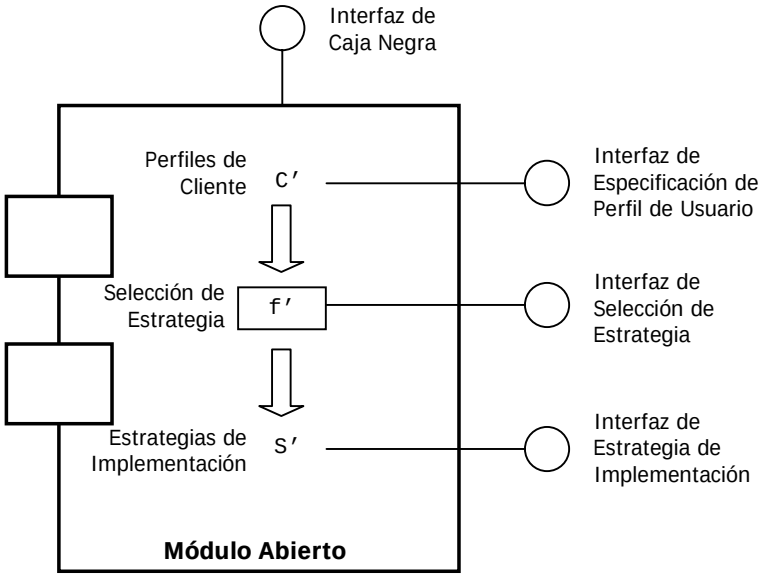


Figura 5.1: Distintas interfaces ofrecidos en una implementación abierta.

1. Interfaz de caja negra: ofrece la funcionalidad del módulo para una estrategia de implementación por defecto.
2. Interfaz de especificación de perfil de usuario: permite definir contextos de utilización futuros por parte del cliente.
3. Interfaz de estrategia de implementación: facilita la adición de nuevas implementaciones para conseguir los servicios ofrecidos por la interfaz de caja negra.
4. Interfaz de selección de estrategia: ofrece al cliente la posibilidad de elegir él mismo la estrategia de implementación oportuna –eliminando así la selección automática conseguida por `f`.

Sobre esta teoría se ha descrito un método para determinar el aspecto de las interfaces ofrecidas al usuario del módulo, a partir de la especificación de un problema. Este método ha sido bautizado con el nombre de "análisis y diseño de implementaciones abiertas" (*Open Implementation Analysis/Design*) [Maeda97].

5.1.1 Aportaciones y Carencias

Esta primera aproximación para la generación de software flexible, consigue ofrecer las ventajas de separar la implementación de la interfaz de un módulo, sin perder eficiencia en la reutilización de código. Es una instancia de la teoría de "separación de incumbencias" [Hürsch95]: se separa una funcionalidad de sus distintas implementaciones.

Este mecanismo ofrece una mera solución al problema de la eficiencia de un sistema construido a través de módulos reutilizables, sin ofrecer ninguno de los grados de flexibilidad indicados en los requisitos § 2.3. Como ejemplo, podemos indicar que el usuario podría conocer las distintas estrategias de implementación existentes en un módulo, si el sistema hubiese sido dotado de introspección (§ 2.3.1).

Este sistema fue la base para crear otros más avanzados como la programación orientada a aspectos (§ 5.3) o la programación adaptable (§ 5.4), que estudiaremos posteriormente en este capítulo.

5.2 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.

Los filtros de composición son objetos instanciados de una clase filtro. Su propósito es acceder y modificar la forma en la que se envían y reciben los mensajes, especificando condiciones para aceptarlo o rechazarlo, y determinando la acción a desencadenar en cada caso. El sistema se asegura que un mensaje sea procesado por los filtros especificados antes de que el método sea ejecutado (filtro de entrada) y antes de que un mensaje sea pasado (filtro de salida). Se identifican así dos niveles de abstracción distintos: los métodos –de mayor nivel de abstracción– y los filtros de composición –adaptables y dependientes del contexto, y por lo tanto de menor nivel de abstracción.

Las acciones a llevar a cabo en la recepción o envío de un mensaje dependen de la clase de la que el filtro es instancia. Existen filtros `Dispatch` (para implementar herencia dinámica o delegación), `RealTime` (restricciones de tiempo real), `Meta` (coordinación de comportamientos), `Error` (incorrección en el manejo de un mensaje) y `Wait` (sincronización de procesos).

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].

El primer lenguaje de programación de filtros de composición fue “Sina” [Koopmans95]. 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].

5.2.1 Aportaciones y Carencias

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). El paso de mensajes es el único comportamiento susceptible de modificación, sin poder rectificar cualquier otro aspecto computacional del sistema (§ 2.3.3).

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. 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 (§ 2.3.4).

El sistema final alcanza un compromiso flexibilidad-eficiencia: simplifica la adaptabilidad de una aplicación a la modificación del paso de mensajes, para conseguir desarrollar aplicaciones suficientemente eficientes en tiempo de ejecución.

5.3 Programación Orientada 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].

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

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

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

Una aplicación desarrollada mediante programación orientada a aspectos está compuesta por la programación de sus componentes funcionales en un determinado lenguaje, y por la programación de un conjunto de aspectos que especifican características adicionales del sistema. El “tejedor de aspectos” (*aspect weaver*) acepta la programación de aspectos y componentes, y genera la aplicación final en un determinado lenguaje de programación.

Como se muestra en la Figura 5.2, la generación de la aplicación final se produce en tres fases:

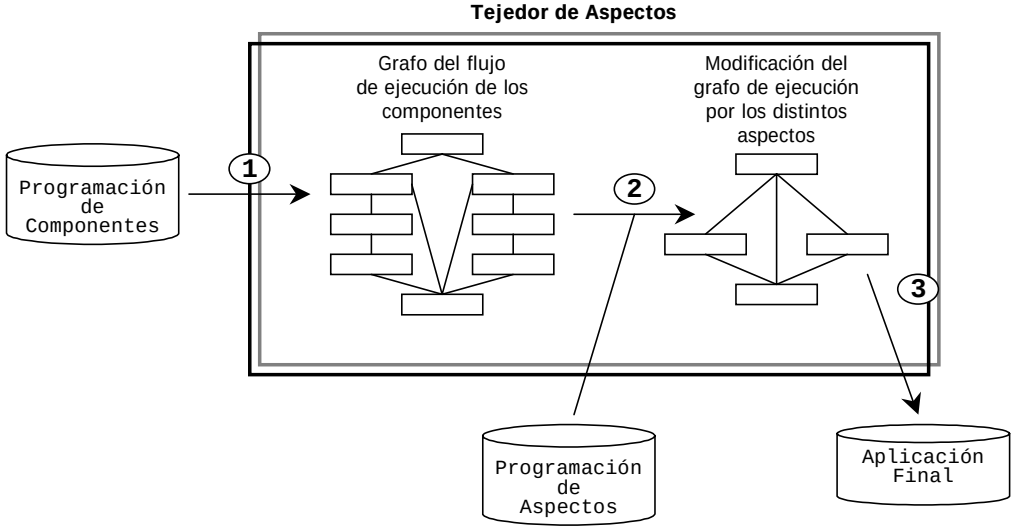


Figura 5.2: Fases en la generación de una aplicación orientada a aspectos.

1. En la primera fase, el tejedor construye un grafo del flujo de ejecución del programa de componentes.
2. El segundo paso consiste en ejecutar la programación de cada uno de los aspectos. Éstos actúan sobre el grafo que define la funcionalidad del sistema (punto anterior) realizando las modificaciones o añadiduras oportunas.
3. La última fase consiste en recorrer el grafo modificado y generar el código final de la aplicación; ésta cumple con la funcionalidad descrita por sus componentes y ofrece los aspectos definidos por el programador.

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

- Puntos de unión (*join points*): Describen puntos de ejecución de una aplicación. AspectJ proporciona un elevado grupo de puntos de unión, siendo un ejemplo de éstos la invocación de métodos.
- Designación de puntos de ruptura (*pointcut designators*): Identifican colecciones de puntos de unión en el flujo de ejecución de un programa, para posteriormente poder modificar su semántica.
- Consejo (*advice*): Acciones que se ejecutan previamente, en el momento, o posteriormente a que se alcance la ejecución de una designación de puntos de ruptura.
- Aspecto (*aspect*): Módulo de implementación, definido como una clase, que agrupa acciones en puntos de unión.

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

5.3.1 Aportaciones y Carencias

La programación orientada a aspectos permite separar la parte funcional de una aplicación de otros aspectos propios de su implantación o despliegue. Esta facilidad otorga al sistema cierta flexibilidad con ciertas limitaciones.

El proceso de especificación de aspectos para modificar la semántica de una aplicación es un proceso estático (§ 2.3.3); una aplicación no puede modificar sus distintos aspectos en tiempo de ejecución.

El modo en el que se crea, y posteriormente se trata, el grafo que define el flujo de la ejecución de la aplicación, supone una elevada complejidad para el programador, e implica la definición de un lenguaje distinto para la descripción de aspectos. La implementación de AspectJ ha sido más sencilla, al hacer uso de la característica introspectiva de la plataforma Java [Sun97d] (§ 2.3.1). De la misma forma, podría facilitarse el tratamiento del grafo añadiendo un sistema de modificación estructural dinámica del sistema (§ 2.3.2).

Otra limitación en la flexibilidad de la programación orienta a aspectos es la restricción a priori impuesta por la definición de puntos de unión (*join points*): si una característica del lenguaje de programación de componentes no se identifica como un punto de unión, no podrá ser modificada posteriormente (§ 2.3.4). Todos los sistemas desarrollados son también dependientes del lenguaje utilizado (§ 2.3.5).

5.4 Programación Adaptable

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

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

Un caso práctico de programación adaptable es el método Demeter (*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; indicando la dirección en el grafo de relaciones entre clases que va a seguir el cálculo de la operación.

- Fragmento de código; especificando algún tipo de restricción, o la realización de la operación desde un elevado nivel de abstracción.

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

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

5.4.1 Aportaciones y Carencias

La flexibilidad conseguida en este tipo de sistemas es similar a la otorgada por la programación orientada a aspectos: es posible separar la funcionalidad central de la aplicación de otras incumbencias no funcionales. Sin embargo, la principal diferencia entre ambos es que los aspectos son una técnica de implementación, y la programación adaptable está más enfocada a la ingeniería del software. Los programas adaptables tratan de representar de algún modo la funcionalidad captada en tiempo de análisis y diseño.

La programación adaptable, más que buscar entornos de computación flexible (como los requeridos para esta tesis), está enfocada a minimizar el impacto en los cambios producidos en entornos de desarrollo evolutivos. Es por esta razón, por la que este tipo de sistemas no cumple los requisitos impuestos de flexibilidad (§ 2.3).

5.5 Separación Multidimensional de Incumbencias

En la introducción de este capítulo “Sistemas Flexibles No Reflectivos”, 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 descomposición 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 5.3): 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.

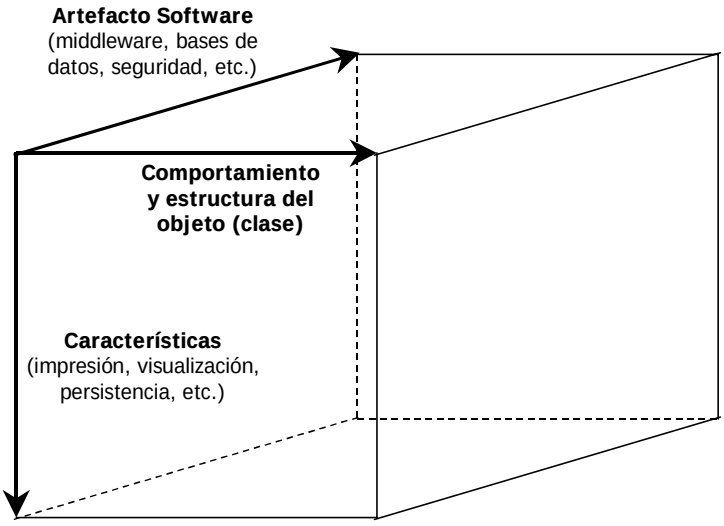


Figura 5.3: Ejemplo de sistema con tres dimensiones de incumbencias.

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 [IBM2000e]. Los principales objetivos de estos sistemas son:

1. 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.
2. 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.
3. 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 [IBM2000f] que da soporte a *Hyperspaces* en el lenguaje de programación Java [Gosling96].

5.5.1 Aportaciones y Carencias

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) descrito en los requisitos de § 2.3.

5.6 Sistemas Operativos Basados en Micronúcleo

A la hora de diseñar sistemas operativos distribuidos, existen dos grandes tipos de arquitecturas distintas [Tanenbaum95]:

- Una se basa en que cada máquina debe ejecutar un núcleo tradicional (núcleo monolítico) que proporcione la mayoría de los servicios.
- Otra diseña el núcleo del sistema operativo de forma que sea lo más reducido posible (micronúcleo), y el grueso de los servicios del sistema operativo se obtiene a partir de los servidores al nivel de usuario.

El objetivo de los sistemas no monolíticos es mantener lo más reducido posible la funcionalidad del micronúcleo. En términos básicos, suelen proporcionar primitivas de comunicación entre procesos, administración de memoria y entrada/salida a bajo nivel. Los servicios se construyen sobre estas primitivas, estableciéndose previamente una interfaz bien definida de acceso al micronúcleo.

El carácter modular de estos sistemas facilita la implantación, eliminación y modificación de los distintos servidores, obteniendo así mayor flexibilidad que los operativos monolíticos. Además, los usuarios que necesiten un determinado servicio específico, podrán codificarlo e implantarlo en el sistema ellos mismos.

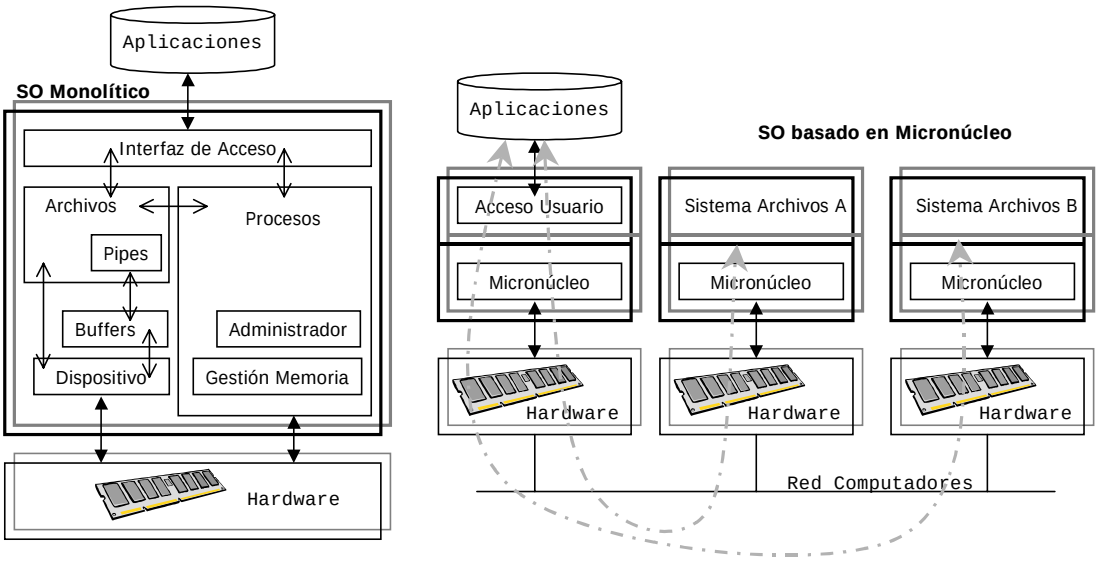


Figura 5.4: Comparación de acceso a sistemas de archivos de sistemas operativos monolíticos y basados en micronúcleo.

En la figura anterior se muestra un ejemplo de utilización de un sistema basado en micronúcleo en la que se implementan distintos sistemas de archivos. Se puede desarrollar un servidor de archivos DOS, otro UNIX y un tercero propietario implementado por el usuario. Los distintos programas desarrollados sobre este operativo, podrán seleccionar cualquiera de los tres sistemas de archivos a utilizar, e incluso utilizar los tres simultáneamente.

La mayor desventaja de estos sistemas respecto a los monolíticos es el rendimiento. El hecho de centralizar las operaciones sobre las primitivas del micronúcleo, supone un mayor acoplamiento, y por lo tanto una pérdida de rendimiento. Sin embargo, mediante distintas técnicas de implementación, se han realizado estudios en los que el rendimiento no baja substancialmente y es aceptable gracias a la flexibilidad obtenida [Liedtke95].

La arquitectura de sistemas operativos basados en micronúcleo ha sido motivo de investigación en los últimos años, existiendo multitud de ellos: Amoeba [Mullender87], CHORUS [Rozier88], MOS [Barak85], Topaz [McJones88], V-System [Cheriton88] o Mach [Rashid86]. Como caso práctico, estudiaremos brevemente dos de ellos.

Amoeba

Ameba es un sistema operativo distribuido que permite agrupar CPUs y equipos de entrada/salida, haciendo que todo el conjunto se comporte como una única computadora [Mullender90]. Facilita también elementos de programación paralela.

Se originó en Vrije Universiteit (Amsterdam) en 1981, como un proyecto de investigación en el cómputo distribuido y paralelo [Tanenbaum95]. Fue diseñado por Andrew Tanenbaum y tres de sus estudiantes de doctorado. En el año 1983, Amoeba 1.0 tenía un primer nivel operacional. El sistema evolucionó durante algunos años, adquiriendo características como emulación parcial de UNIX, comunicación en grupo y un nuevo protocolo de bajo nivel.

Una de las mayores diferencias entre Amoeba y el resto de sistemas operativos distribuidos es que Amoeba carece del concepto de máquina “origen”. Cuando un usuario entra en el sistema, entra a éste como un todo y no a una máquina específica. Las máquinas

no tienen propietarios. El shell ofrecido al usuario busca de manera automática la máquina con la menor carga, para ejecutar cada en ella comando solicitado.

Amoeba consta de dos partes fundamentales: el micronúcleo que se ejecuta en cada procesador, y una colección de servidores que proporcionan la mayor parte de la funcionalidad global de un sistema operativo tradicional. La estructura general se muestra en la siguiente figura:

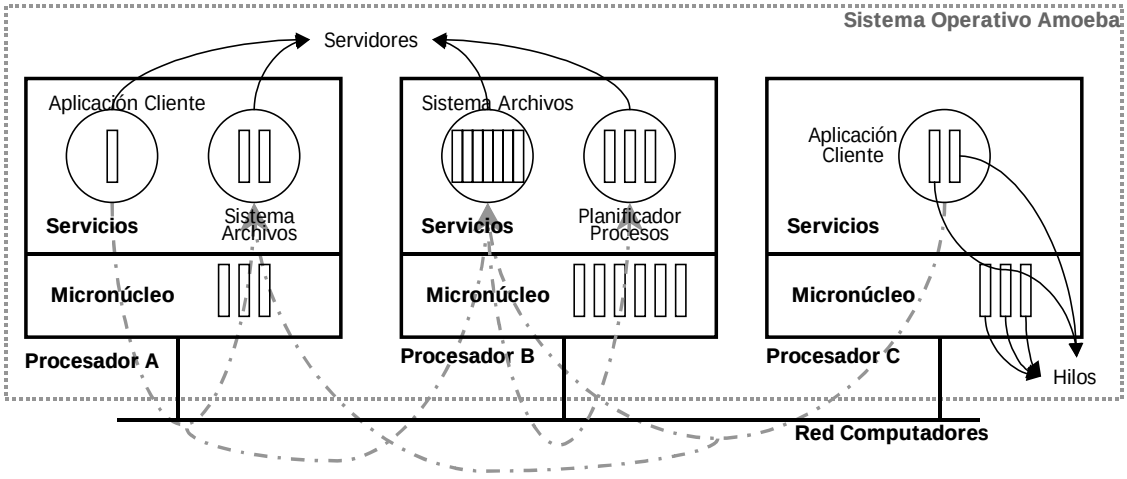


Figura 5.5: Estructura de aplicaciones sobre el sistema operativo Amoeba.

El micronúcleo se ejecuta en todas las máquinas del sistema. Éste tiene cuatro funciones básicas:

1. Manejo de procesos e hilos.
2. Soporte del manejo de memoria a bajo nivel.
3. Soporte de la comunicación.
4. Manejo de entrada/salida mediante primitivas de bajo nivel.

Mediante esta arquitectura se consiguen dos grandes objetivos: distribuir los servicios a lo largo de una red heterogénea de computadores, y separar la interfaz e implementación de un servicio para que éste pueda adaptarse a las distintas necesidades.

Mach

El principal objetivo del sistema operativo Mach era demostrar que se podían estructurar los sistemas operativos de manera modular, como una colección de procesos que se comuniquen entre sí mediante la transferencia de mensajes, incluso a través de una red [Babao89].

La primera versión de Mach apareció en 1986 para un VAX 11/784, un multiprocesador con cuatro CPUs. Poco después se instituyó la *Open Software Foundation*, un consocio de vendedores de computadores, con el fin de arrebatarle el control del UNIX a su creador AT&T. OSF eligió a Mach 2.5 como la base para su primer sistema operativo.

En 1988, el núcleo de Mach era grande y monolítico, debido a la presencia de una gran parte del código UNIX de Berkeley. En 1989, se eliminó dicho código del núcleo y se ubicó en el espacio de usuario, constituyendo Mach 3.0 como un sistema basado en micronúcleo [Tanenbaum95]. Éste manejaba cinco abstracciones principales:

1. Procesos.

2. Hilos.
3. Objetos de memoria.
4. Puertos.
5. Mensajes.

El micronúcleo de Mach se diseñó como una base para emular cualquier tipo de sistema operativo. La emulación se lleva a cabo mediante una capa software ejecutada fuera del núcleo, en el espacio de usuario –como se muestra en la Figura 5.6. Con este sistema, es posible ejecutar aplicaciones sobre distintos sistemas operativos, al mismo tiempo y sobre una misma máquina.

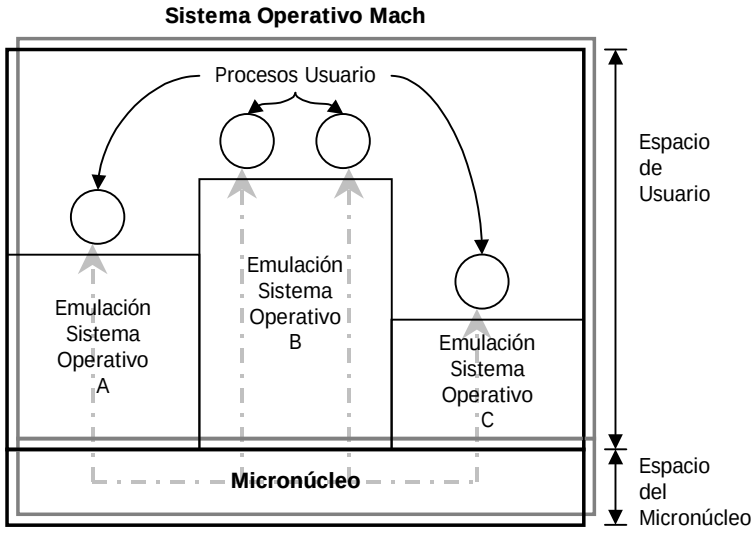


Figura 5.6: Ejecución de aplicaciones sobre distintos sistemas operativos emulados en Mach.

La idea subyacente en el núcleo de Mach es proporcionar los mecanismos necesarios para que el sistema funcione, dejando las distintas políticas para los procesos de usuario. El sistema de micronúcleo no sólo permite flexibilizar los distintos servicios del operativo –como veíamos en el caso de Amoeba–, sino que puede utilizarse para flexibilizar el propio sistema operativo, implementando simultáneamente distintas emulaciones.

5.6.1 Aportaciones y Carencias

La idea principal de los sistemas operativos basados en micronúcleo se centra en codificar la parte computacional mínima de forma monolítica, para identificar posteriormente los servicios de un mayor nivel de abstracción como unidades modulares reemplazables. Este tipo de sistemas aporta un modo sencillo de diseñar un primer mecanismo de flexibilidad para una plataforma de computación (§ 2.1.4) –sistemas estructuralmente reflectivos, como la máquina abstracta de Smalltalk [Goldberg83], utilizan esta técnica para poder adaptar y extender su núcleo computacional [Krasner83].

Las primitivas ofrecidas por el micronúcleo pueden ser ampliadas a un mayor nivel de abstracción implementándolas como servicios, y flexibilizando así los elementos básicos del sistema computacional (§ 2.2.1). El usuario puede modificar, elegir y ampliar cualquier característica del sistema (§ 2.2.2), siempre que haya sido desarrollada como un servicio.

Otra de las aportaciones positivas derivadas de la reducción del núcleo computacional es la posibilidad de implantar éste en entornos heterogéneos (§ 2.4.5). Al haberse

especificado previamente una interfaz de acceso, la selección de los distintos servicios que se desarrollan sobre el micro núcleo puede ser llevada a cabo de forma dinámica (§ 2.2.2).

Una de sus mayores carencias para convertirse en entornos computacionales flexibles es la falta de un sistema introspectivo que permita conocer (§ 2.2.3) y autodocumentar (§ 2.2.4) dinámicamente todas las características de los servicios implantados en el sistema.

5.7 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. 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. Estos trabajos de investigación identifican, de algún modo, la necesidad de desarrollar entornos de programación flexibles.

La principal diferencia entre los distintos sistemas estudiados reside en el punto de vista que adoptan para conseguir la separación de incumbencias: desarrollo de componentes (§ 5.1), modificación de parte de la semántica de un lenguaje de programación (§ 5.2), separación de funcionalidades (§ 5.3), ciclo de vida de un sistema informático (§ 5.4) y computaciones susceptibles de ser reemplazadas (§ 5.6).

A excepción de los sistemas operativos basados en micronúcleo, las herramientas analizadas otorgan un reducido nivel de flexibilidad para resolver un problema concreto, sin incurrir en elevadas carencias de eficiencia computacional.

En lo referente a los sistemas operativos basados en micronúcleo, el criterio de diseño centrado en separar la computación primitiva de los servicios supone un mecanismo para desarrollar un sistema con un elevado grado de flexibilidad: todo lo desarrollado como un servicio es reemplazable.

La simplificación del micronúcleo facilita su implantación en entornos heterogéneos y simplifica la recodificación de sus primitivas como servicios –para su posterior modificación.

Sus principales carencias son acarreadas por la ausencia de un mecanismo de acceso dinámico[19].

- No es posible conocer de forma exacta y en tiempo de ejecución los servicios existentes en el sistema.
- La selección dinámica de distintos servicios no está permitida.
- No es posible modificar un servicio de forma dinámica, sin necesidad de finalizar su ejecución.
- La utilización de un servicio por una aplicación es explícita, y por lo tanto es necesario modificar la aplicación si queremos modificar algún aspecto de los servicios que utiliza.

[19] Veremos en el capítulo 10 cómo estos problemas se pueden solventar con la incorporación de características reflectivas al sistema computacional base.

CAPÍTULO 6:
REFLECTIVIDAD COMPUTACIONAL

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[20] 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.

[20] 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[21] 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.

[21] Cosificar o concretizar: transformar una representación, idea o pensamiento en cosa.

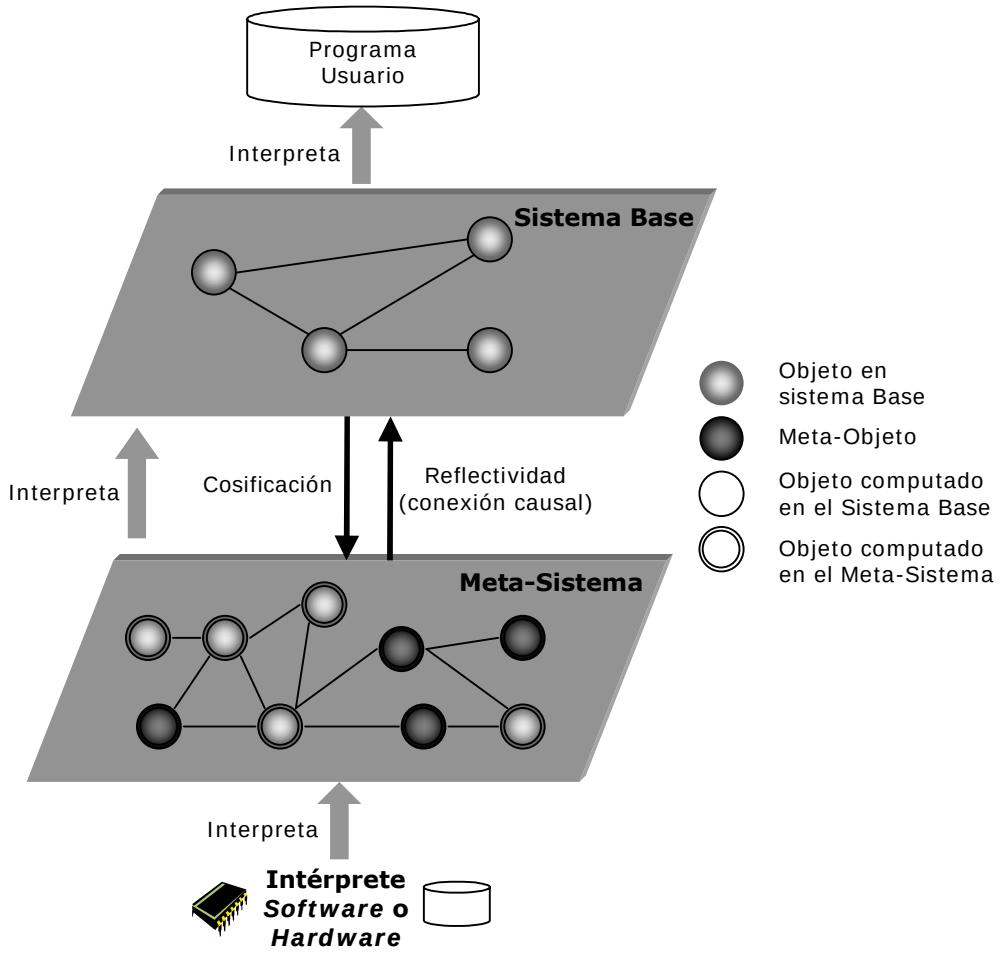


Figura 6.1: Entorno de computación reflectivo.

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 propia 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.

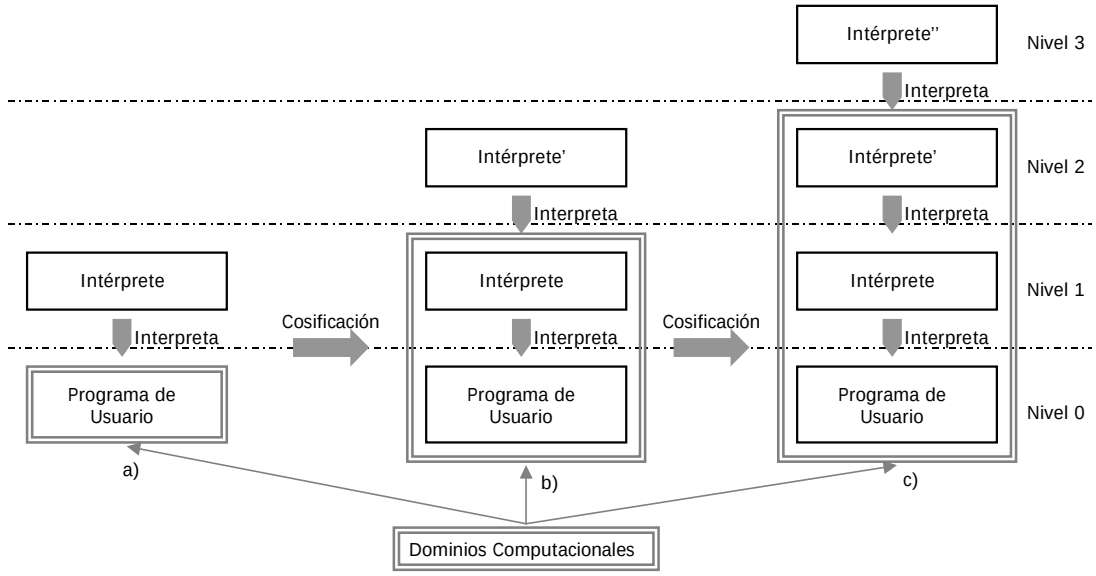


Figura 6.2: Torre de intérpretes definida por Smith.

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].

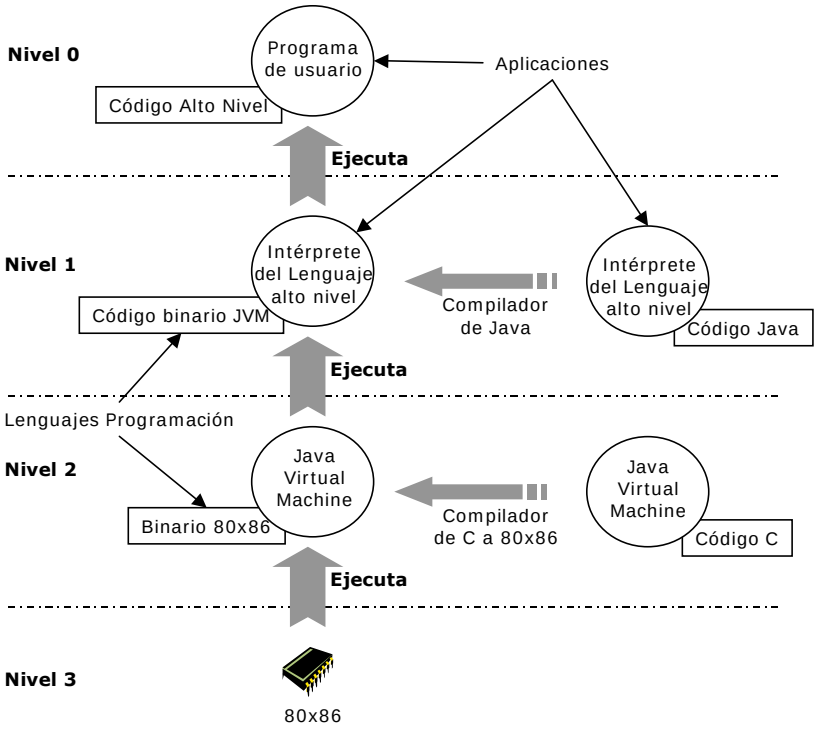


Figura 6.3: Ejemplo de torre de intérpretes en la implementación de un intérprete en Java.

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 mi-

croprocesador 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[22] (*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 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` [Sun97d]. 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 convierte en una secuencia de bytes para su posterior envío a un flujo[23] [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, *Run-Time Type Information*) [Kalev98]. Este mecanismo introspectivo permite conocer la clase de

[22] Ciertamente hacemos datos (o cosa) un comportamiento.
[23] Este proceso se conoce como “serialización” (*serialization*).

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 (§ 4.3).

6.3.1.3 Reflectividad Computacional

También denominada reflectividad de comportamiento (*behavioural 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[24]. 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 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 [Cueva95b]. 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.

[24] Estudiaremos este mecanismo, en mayor profundidad, en el siguiente capítulo.

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[25] 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].

[25] No previstos en tiempo de compilación –cuando la aplicación se esté siendo implementada.

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:

1. La totalidad del sistema es accesible y por lo tanto cualquier parte de éste podrá ser manipulada. No existen restricciones en el acceso.
2. 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.

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 obte-

nida; el principal inconveniente, la necesidad de establecer un mecanismo de seguridad en el acceso entre procesos (§ 6.4).

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 (§ 3.3.3).

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 sistema 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.

6.4 Hamiltonians versus Jeffersonians

Durante el desarrollo del sistema democrático americano, existían dos escuelas de pensamiento distintas: los *Hamiltonians* y los *Jeffersonians*. Los *Jeffersonians* pensaban que cualquier persona debería tener poder para llevar a cabo decisiones como la elección del presidente de los Estados Unidos. Por el contrario, los *Hamiltonians* eran de la opinión de que sólo una elite educada para ello, y no todo el pueblo, debería influir en tales decisiones. Esta diferencia de pensamiento ha sido utilizada para implementar distintas políticas en sistemas computacionales reflectivos [Foote92].

La vertiente Jeffersoniana otorga al programador de un sistema reflectivo la posibilidad de poder modificar cualquier elemento de éste. No restringe el abanico de posibilidades en el acceso al metasistema. El hecho de permitir modificar cualquier característica del sistema, en cualquier momento y por cualquier usuario, puede desembocar en una semántica del sistema prácticamente incomprensible. Un ejemplo de esto, es un programa escrito

en C++ con uso intensivo de sobrecarga de operadores (reflectividad del lenguaje): el resultado puede ser un código fuente prácticamente ininteligible.

Muchos autores abogan por el establecimiento de un mecanismo de seguridad que restrinja el modo en el una aplicación pueda modificar el sistema [Foote90]. Un ejemplo clásico es establecer una lista de control de acceso en el que sólo los administradores del sistema puedan cambiar la semántica global. Por el contrario, el resto de usuarios podrán modificar únicamente la semántica de sus aplicaciones.

Como mencionamos en el capítulo 1, esta tesis trata de buscar un entorno computacional con un elevado grado de flexibilidad. No es nuestro objetivo la implementación de un sistema de seguridad encargado de controlar el nivel de reflexión permitido –esto formaría parte del desarrollo de una plataforma comercial. Por esta razón, nos centraremos en la vertiente Jeffersoniana, teniendo siempre en cuenta los riesgos que ésta puede llegar a suponer.

6.5 Formalización

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

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

En esta tesis, el estudio y diseño de las distintas técnicas para obtener reflectividad computacional serán abordadas desde el punto de vista pragmático y empírico; no formal.

CAPÍTULO 7:
PANORÁMICA DE UTILIZACIÓN 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, especificaremos brevemente su funcionamiento, y, en el ámbito de grupo, detallaremos los beneficios aportados y carencias existentes de los sistemas estudiados. A modo de conclusión y apoyándonos en los requisitos definidos en el capítulo 2, analizaremos las principales carencias del conjunto total de sistemas existentes.

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.

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 ejecución, 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.

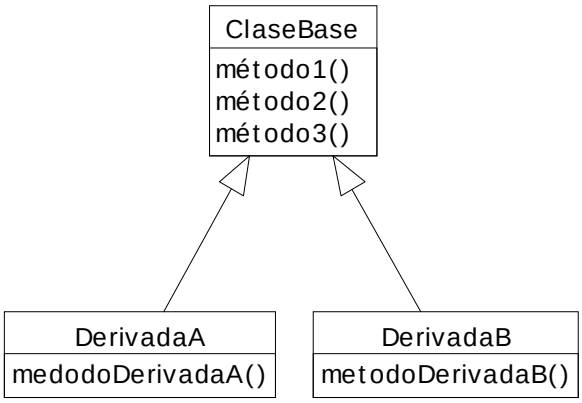


Figura 7.1: Punteros o referencias "Base" permiten la utilización genérica de objetos derivados.

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.

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* [Sun97d]. 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*[26] 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.

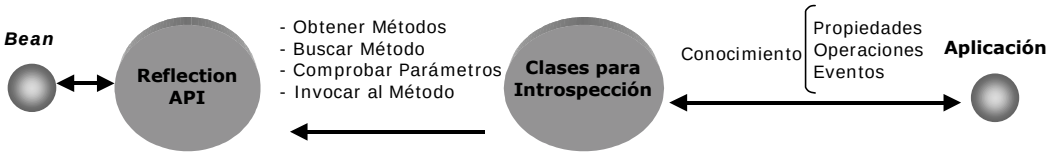


Figura 7.2: Utilización de introspección en el desarrollo de un sistema de componentes.

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[27], para posteriormente 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].

[26] En Java los *arrays* son objetos primitivos.
[27] También denominado "serialización" (*serialization*).

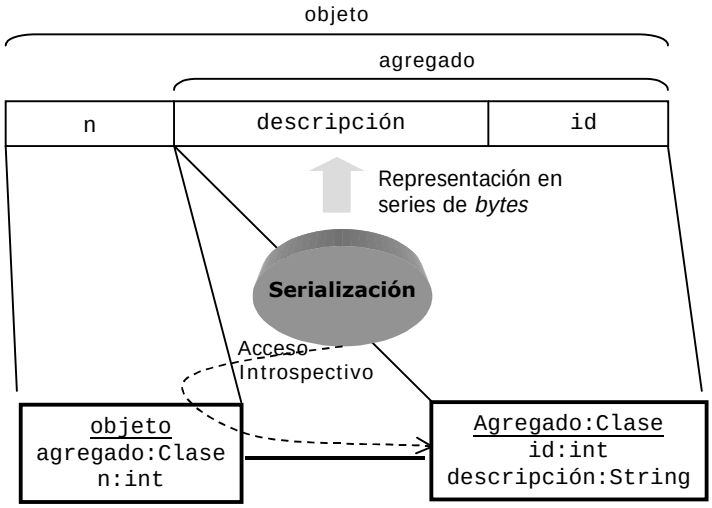


Figura 7.3: Conversión de un objeto a una secuencia de bytes, mediante un acceso introspectivo.

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.

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.

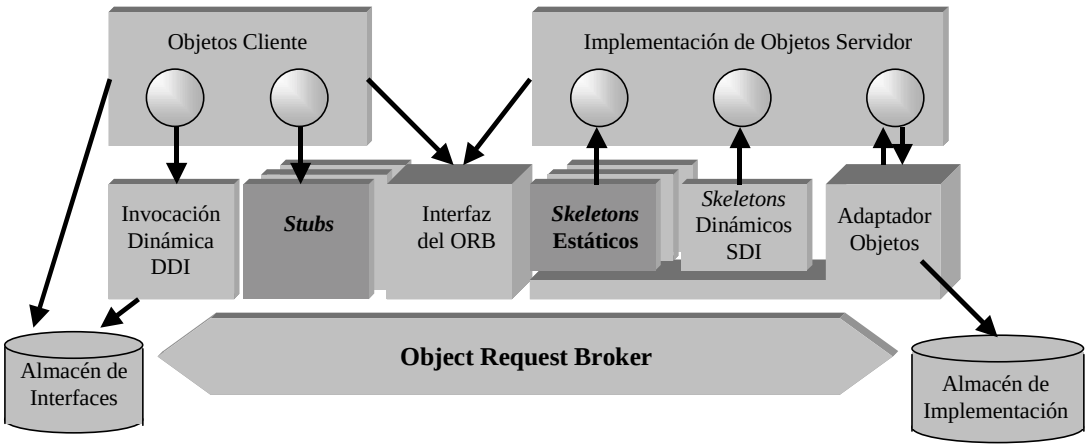


Figura 7.4: Elementos de la arquitectura de objetos distribuidos CORBA.

Como se muestra en la Figura 7.4[28], 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].

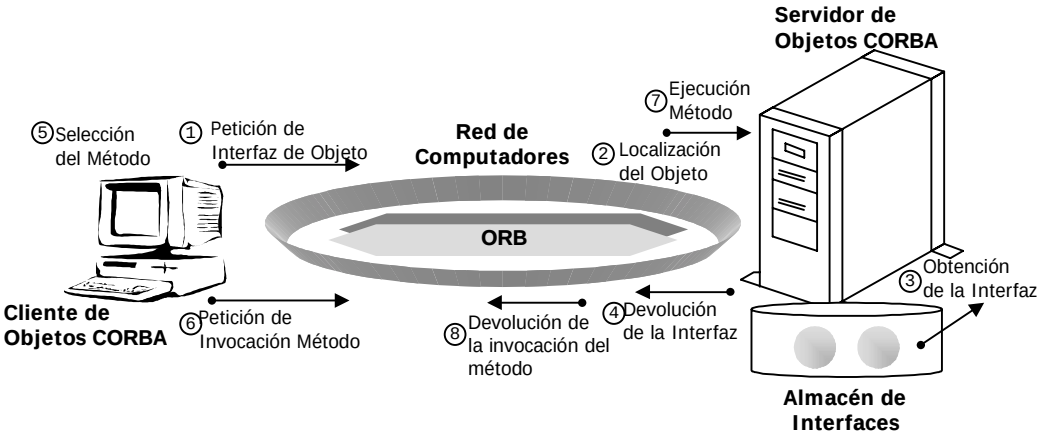


Figura 7.5: Invocación dinámica de un método en CORBA utilizando DII.

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

[28] Para obtener una descripción más detallada del funcionamiento de cada uno de los módulos de la arquitectura CORBA, consúltese [OMG95] o [Orfali98].

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.

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.

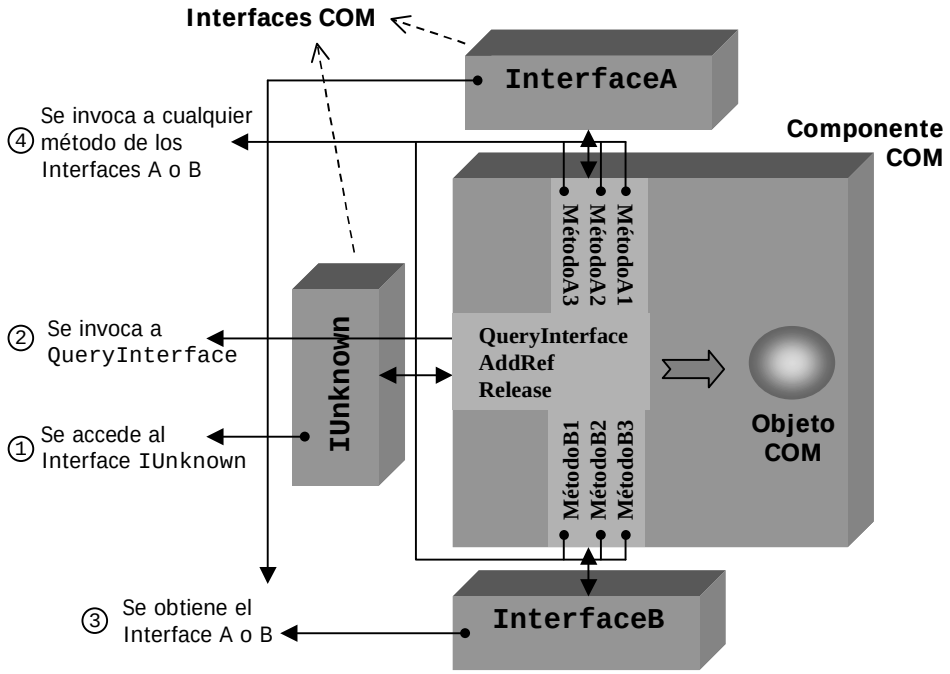


Figura 7.6: Sistema de introspección en el modelo de componentes COM.

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].

7.1.1 Aportaciones y Carencias de los Sistemas Estudiados

En este punto hemos visto cómo es importante la característica introspectiva en el desarrollo de una plataforma. Desde el punto de vista del programador, es necesaria para realizar operaciones seguras respecto al tipo (RTTI de C++) o para almacenar objetos en disco ("Serialización" de la plataforma Java). Desde el punto de vista de un sistema operativo, es necesaria para desarrollar un sistema de componentes (COM). Desde el punto de vista de entornos distribuidos, es necesaria para desarrollar aplicaciones flexibles (CORBA y Java).

La utilización de la introspección queda patente con numerosos ejemplos prácticos de sistemas actuales reales. En el desarrollo de aplicaciones de comercio electrónico, es común utilizar la transferencia de información independiente de la plataforma y autodescriptiva en formato XML [W3C98]. Mediante la utilización de una librería –DOM [W3C98b] o SAX [Megginson2000]– es posible conocer los datos en tiempo de ejecución (introspección), eliminando el acoplamiento que se produce entre el cliente y el servidor en muchos sistemas distribuidos. De esta forma, las aplicaciones ganan en flexibilidad pudiéndose adaptar a los futuros cambios.

En el desarrollo de una plataforma de computación flexible, la introspección es una característica fundamental a ser otorgada (§ 2.1.5.2) a su motor computacional, obteniéndose así un primer grado de flexibilidad gracias al conocimiento dinámico de la totalidad del sistema (§ 2.3.1).

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.

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[29] 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.

[29] 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*.

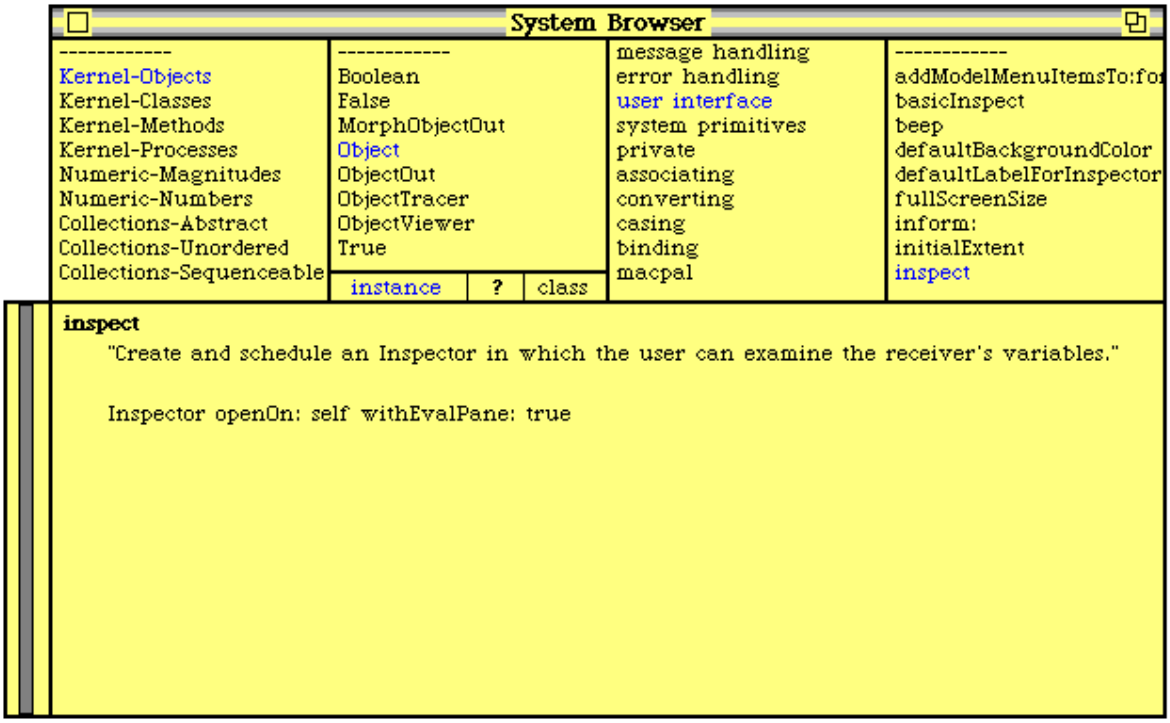


Figura 7.7: Análisis del método *inspect* del objeto de clase *Object*, con la aplicación *Browser* de Smalltalk-80.

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.

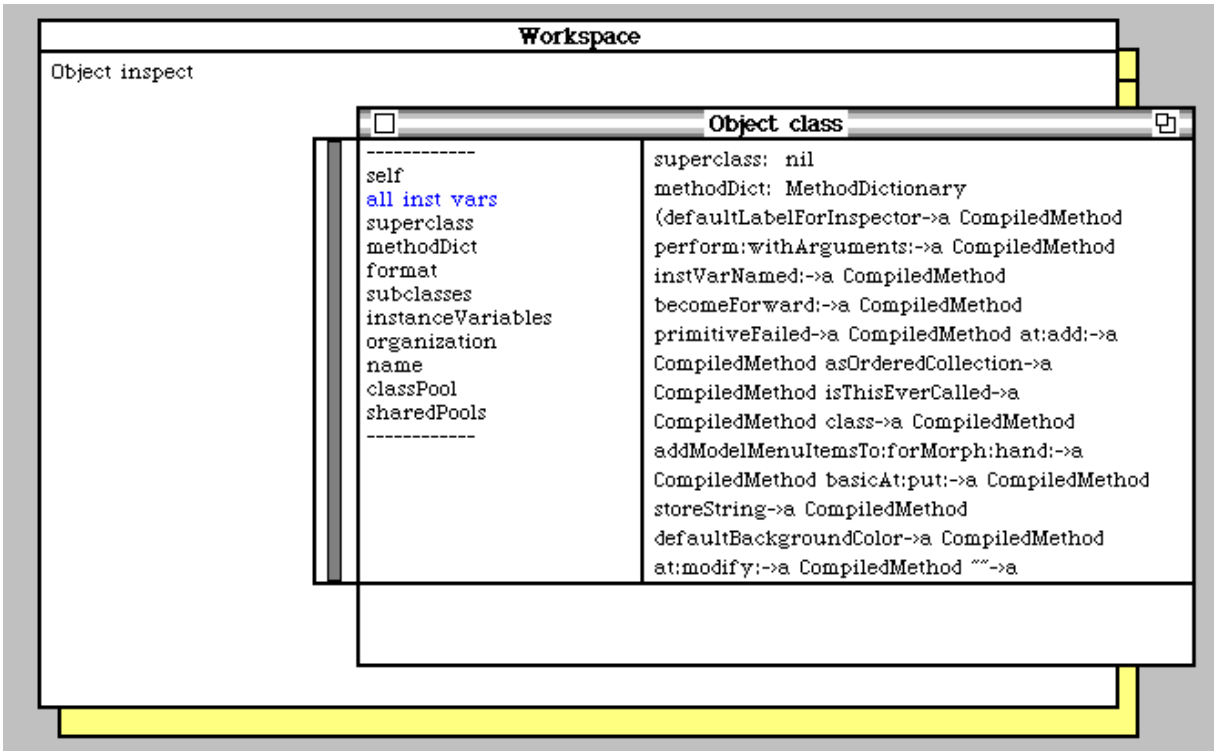


Figura 7.8: Invocando al método *inspect* del objeto *Object* desde un espacio de trabajo de Smalltalk-80, obtenemos un acceso a las distintas partes de la instancia.

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.

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” – profundizaremos en el estudio de éstos en el capítulo 8.

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:

1. Implementar, sobre cualquier lenguaje de programación, un pequeño intérprete de un subconjunto del lenguaje a interpretar. En su caso un intérprete de "tiny-Self". El requisito fundamental es que éste tenga reflectividad estructural.
2. Sobre lenguaje (tinySelf), se desarrolla un intérprete del lenguaje buscado (Self), con todas sus características.
3. 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.

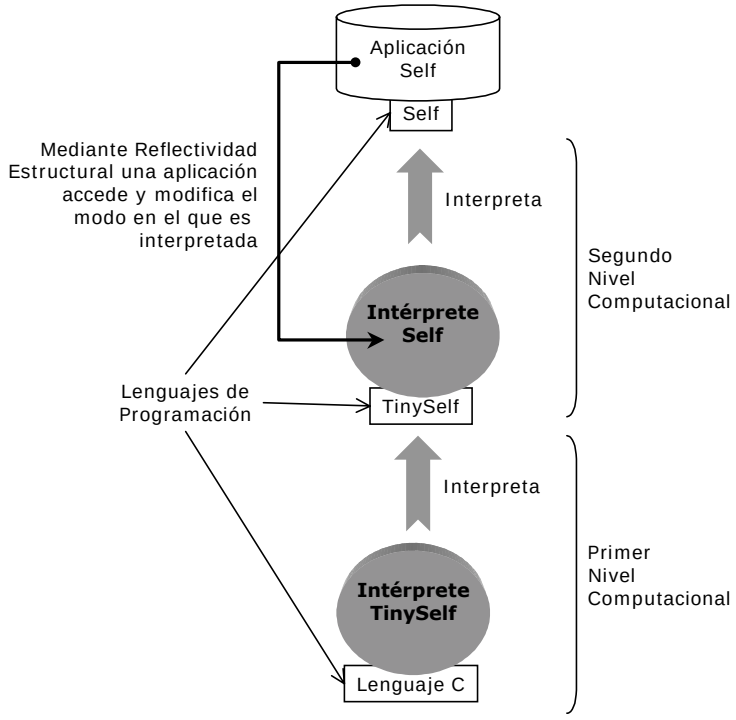


Figura 7.9: Consiguiendo reflectividad mediante la introducción de un nuevo intérprete.

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 des-

arrollado éste, traducir el código propio del intérprete de Self –codificado en el lenguaje tinySelf– a código nativo, reduciendo así un nivel computacional.

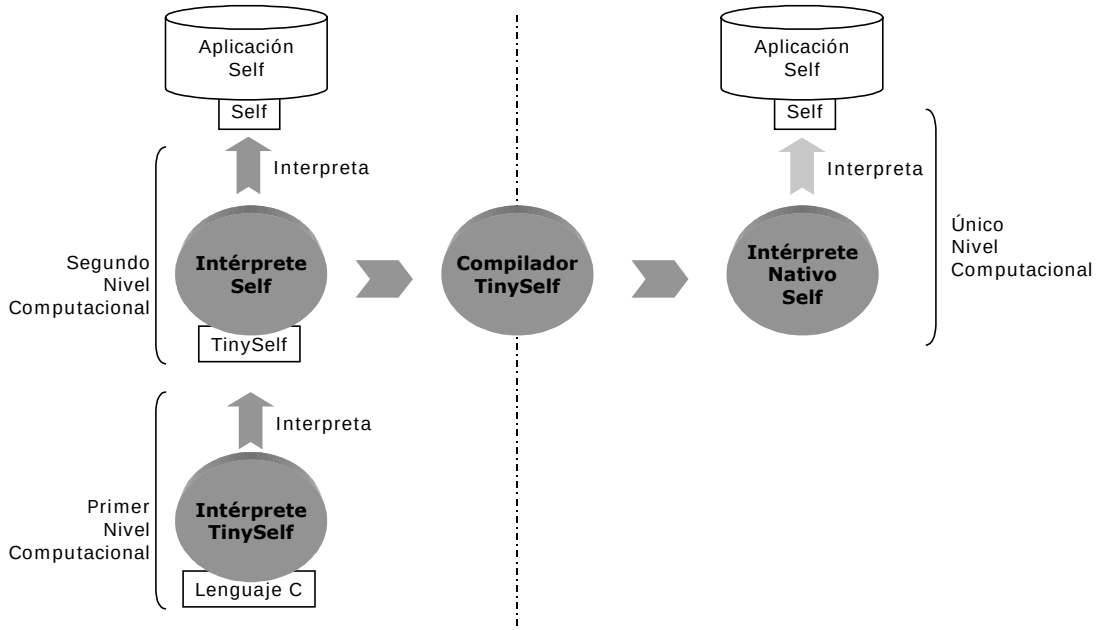


Figura 7.10: Reducción de un nivel de computación en la torre de intérpretes.

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:

1. 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.
2. 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.

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]:

1. 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 miem-

bros de un objeto pueden conocerse y modificarse dinámicamente (reflectividad estructural).

2. La única forma de llevar a cabo una computación sobre un objeto es enviándole un mensaje indicativo del método que queramos ejecutar.
3. Todo objeto ha de pertenecer a una clase que especifique su estructura y comportamiento. Los objetos se crearán dinámicamente como instancias de una clase.
4. 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.
5. 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`.
6. 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:

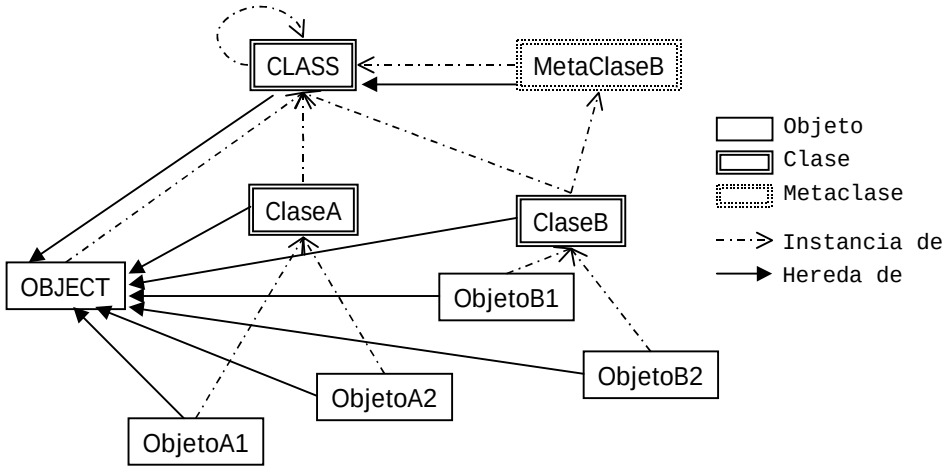


Figura 7.11: Doble grafo de objetos en el sistema ObjVlisp.

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.

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[30] [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:

1. La aplicación inicial es ejecutada por una implementación de la máquina virtual de Java.
2. La aplicación genera código dinámicamente en función de los requisitos propios de su contexto de ejecución.
3. Este nuevo fragmento de código es compilado a código binario de la máquina virtual, mediante un compilador de Java codificado en Java. Este proceso se realiza con la comprobación estática de tipos propia del lenguaje.
4. 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.

[30] El concepto definido como "*linguistic reflection*" no es el mismo que definíamos en § 6.3.1 como "reflectividad lingüística".

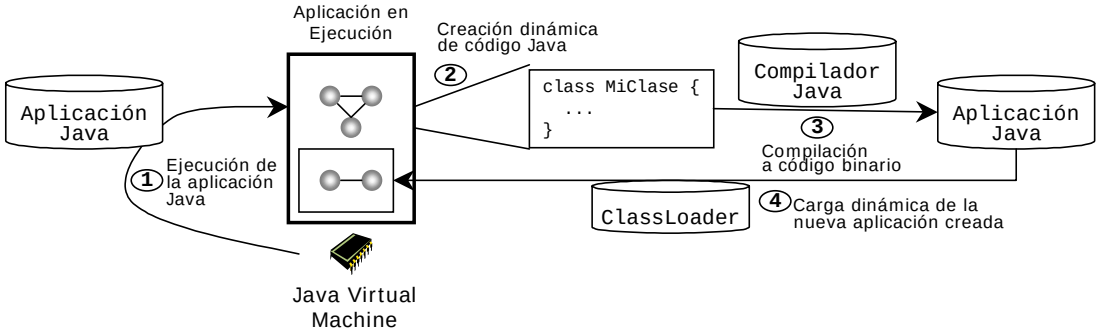


Figura 7.12: Evaluación dinámica de código creado en tiempo de ejecución.

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 [Sun97d]. 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.

Python

Python es un lenguaje de programación orientado a objetos, portable e interpretado [Rossum2001]. Su desarrollo comenzó en 1990 en el CWI de Amsterdam. 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 programa-

ción [Andersen98], ofreciendo un mayor nivel de abstracción basándose en el concepto de metaobjeto [Kiczales91].

7.2.1 Aportaciones y Carencias de los Sistemas Estudiados

Los sistemas dotados de reflectividad estructural ofrecen una mayor flexibilidad que los que únicamente ofrecen introspección. En Java, una aplicación puede modificar su flujo en función de su estado –conociéndolo mediante introspección–; en Smalltalk, además puede modificar su estructura dinámicamente –gracias a la reflectividad estructural. Si analizamos la dicotomía anterior desde el punto de vista de la seguridad, para una plataforma comercial sería peligroso ofrecer reflectividad estructural, puesto que cualquier aplicación podría acceder y modificar la estructura del sistema. Esta discusión se detalla en § 6.4.

Los sistemas estudiados ofrecen conocimiento automático de su estructura (§ 2.3.1) y su modificación (§ 2.3.2). Si se codifican todas las primitivas computacionales sobre el mismo lenguaje, al ofrecer éste reflectividad estructural, se podrán adaptar a cualquier contexto dinámicamente. Solo tendremos que acceder a éstas y modificar su implementación. Mediante este mecanismo implementado en el proyecto Merlin y en ObjVLisp, obtenemos la adaptabilidad del entorno de programación (§ 2.2).

Una de las características analizadas tanto en el lenguaje Python como en el desarrollo del sistema ObjVlisp, es la modificación de la semántica del lenguaje para un objeto o una clase. En estos ejemplos, mediante el uso de la reflectividad estructural, es posible modificar el mecanismo de interpretación de la recepción de un mensaje para una clase; modificando estructuralmente una metaclase en ObjVlisp se puede conseguir este objetivo. Sin embargo, no estamos obteniendo la modificación del comportamiento para el sistema global –característica propia de la reflectividad computacional. Estos sistemas carecen por tanto de la modificación dinámica de su semántica (§ 2.3.3).

Finalmente comentaremos cómo el proyecto Merlin propuso implementar un intérprete de Self codificado en tinySelf para ofrecer reflectividad computacional. Aunque ésta no es una tarea sencilla, sí que supone un mecanismo de implementación para obtener un sistema computacionalmente reflectivo (§ 2.3.3). Utilizaremos esta idea para llevar a cabo el diseño de nuestro entorno computacionalmente reflectivo sin restricciones (capítulo 11).

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.

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 ejecu-

ción. 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.

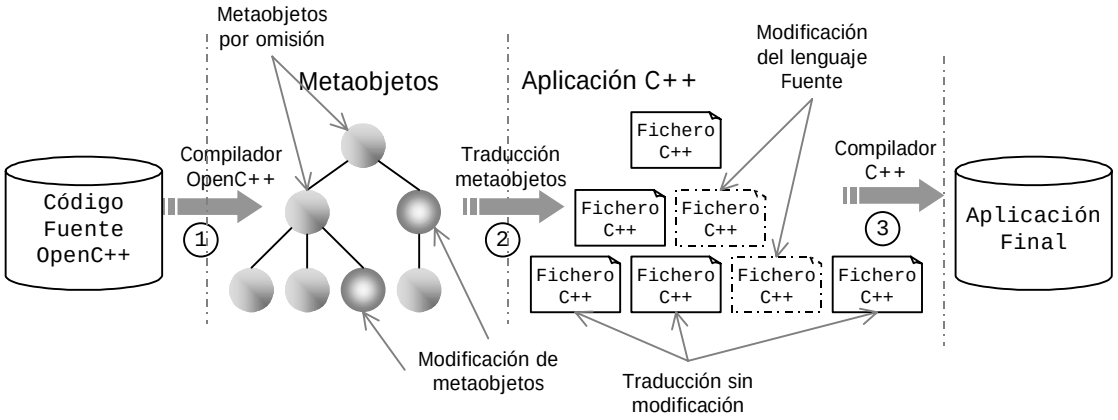


Figura 7.13: Fases de compilación de código fuente OpenC++.

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). Está enfocado a la creación de librerías para poder crear un sistema de separación de aspectos en programación (§ 5.3).

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` [Sun97d], 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 [Cueva95b].

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:

1. Al no modificarse la máquina virtual, no se generan distintas versiones de ésta con la consecuente pérdida de portabilidad de su código fuente [Ortin2001].
2. 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].

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:

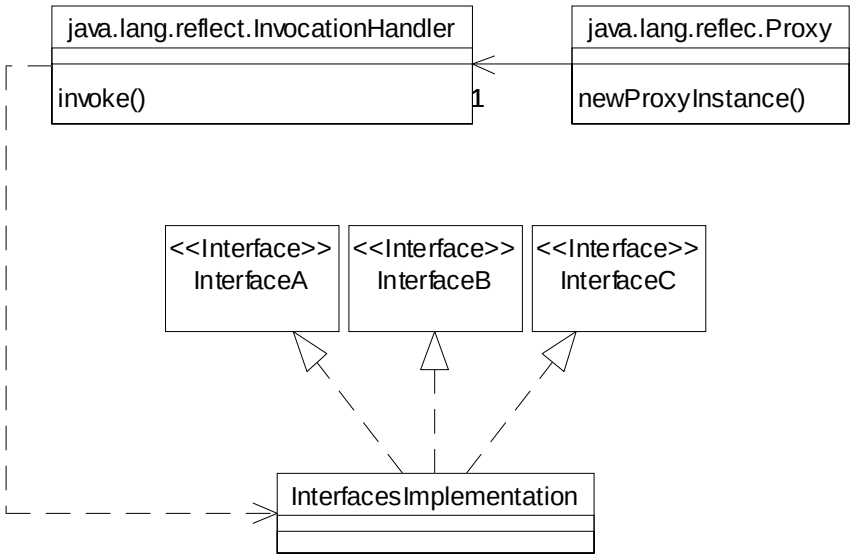


Figura 7.14: Diagrama de clases de la implementación de un manejador de invocaciones.

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 [Sun97d], 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, como indicábamos en § 7.2.1, 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.

7.3.1 Aportaciones y Carencias de los Sistemas Estudiados

La principal carencia de los sistemas reflectivos en tiempo de compilación es la incapacidad de poder amoldarse a situaciones no previstas en fase de desarrollo. Toda la adaptabilidad de un sistema, deberá definirse cuando éste es construido y, una vez que esté ejecutándose, no podrá adecuarse de otra manera (§ 2.3.3). Por otro lado, al determinarse el grado de flexibilidad en tiempo de compilación, estos sistemas son más eficientes que los adaptables dinámicamente.

Teniendo en cuenta el nivel de reflexión ofrecido, los sistemas estudiados presentan una mayor información a reflejar que los que estudiaremos en el siguiente punto. La reflec-

tividad del lenguaje –o modificación de éste– se ofrece comúnmente en sistemas reflectivos estáticos, cuando en los dinámicos supone una carencia (§ 2.3.5).

La compilación previa del código fuente, permite utilizar lenguajes seguros respecto al tipo [Cardelli97], eliminando una elevado número de errores en tiempo de ejecución propio de sistemas de naturaleza interpretada; por ejemplo Python [Rossum2001].

Como veremos en el estudio de los sistemas reflectivos dinámicos basados en *Meta-Object Protocols* (MOPs) (§ 7.4.1), existe una restricción previa de lo que puede ser modificado. En el estudio de los sistemas reflectivos en tiempo de compilación, vemos cómo también sucede lo mismo: el lenguaje define qué partes de él mismo pueden ser adaptadas (§ 2.3.4). La diferencia es que en los sistemas analizados, dicha restricción se produce en fase de compilación, frente a la restricción dinámica producida en el uso de MOPs.

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.

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 7.1:

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.

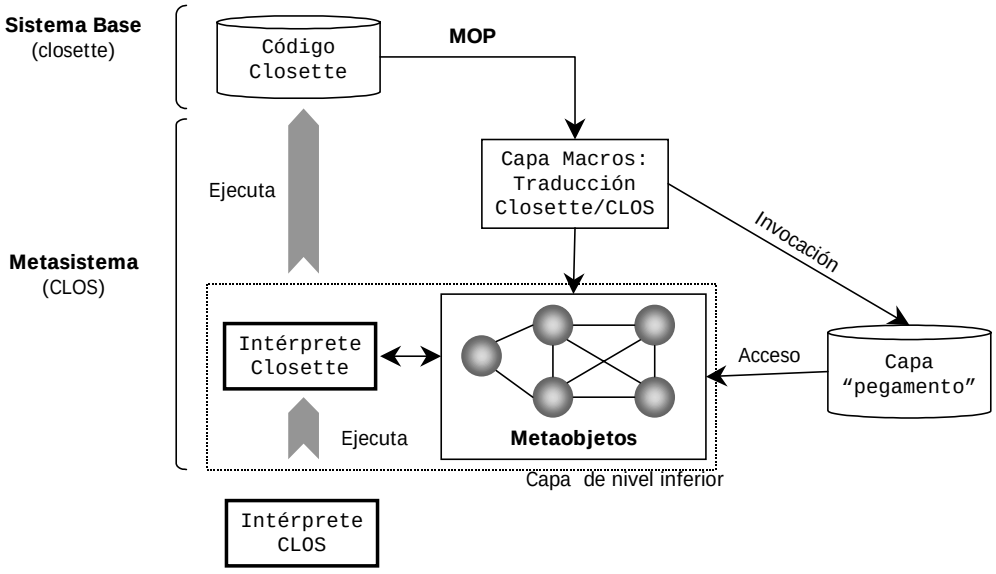


Figura 7.15: Arquitectura del MOP desarrollado para el lenguaje CLOS.

MetaXa[31]

MetaXa 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

[31] Anteriormente conocido como MetaJava.

del evento. La computación del sistema base es suspendida de forma síncrona, hasta que el metasistema finalice la interpretación del evento capturado.

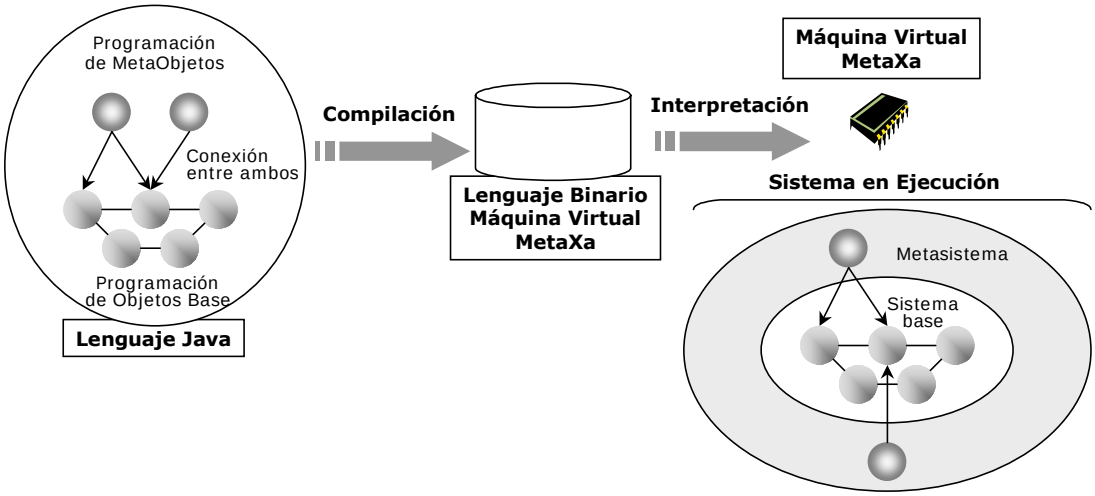


Figura 7.16: Fases en la creación de una aplicación en MetaXa.

Si un metaobjeto recibe el evento *method-enter*, la acción por omisión será ejecutar el método apropiado. Sin embargo, el metaobjeto puede sobreescribir 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 [Sun97c].

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.

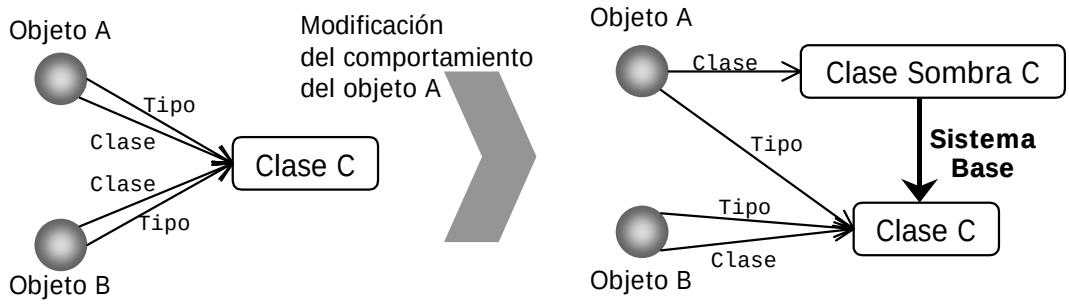


Figura 7.17: Creación dinámica de una clase sombra en MetaXa para modificar el comportamiento de una instancia de una clase.

Hay que solventar un conjunto de problemas para mantener coherente el sistema en la utilización de las clases sombra [Golm97c]:

1. Consistencia de atributos. La modificación de los atributos de una clase ha de mantenerse coherente con su representación sombra.
2. 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.
3. 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.
4. Recolector de basura. Las clases sombra deberá limpiarse cuando el objeto no tenga asociado un metaobjeto.
5. 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.
6. Consumo de memoria. La duplicidad de una clase produce elevados consumos de memoria; deberán idearse técnicas para reducir estos consumos.
7. 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].

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 [Cueva92b]– para llevar a cabo todo tipo de comprobación de tipos [Cueva95b] 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:

1. 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.
2. 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.

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:

1. 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).
2. 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.
3. 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.
4. 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.
5. 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 7.18. 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.

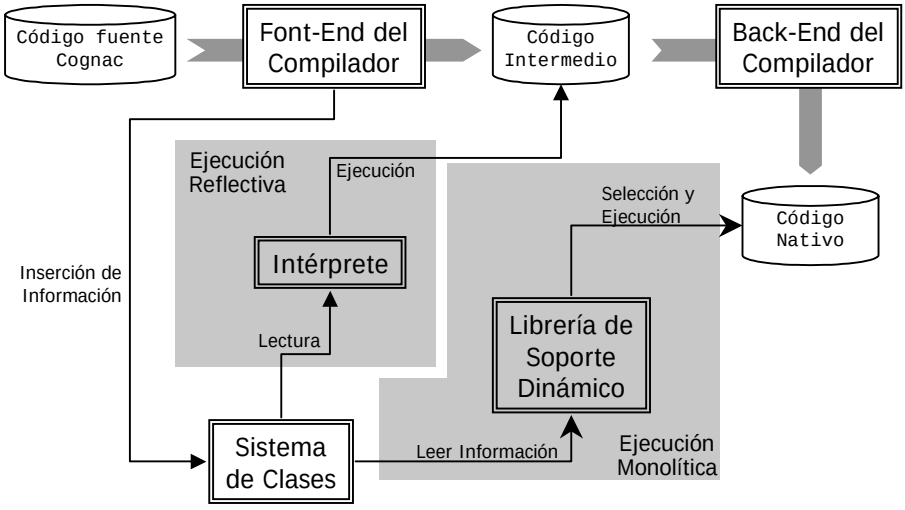


Figura 7.18: Arquitectura y ejecución de una aplicación en Cognac.

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 7.19).

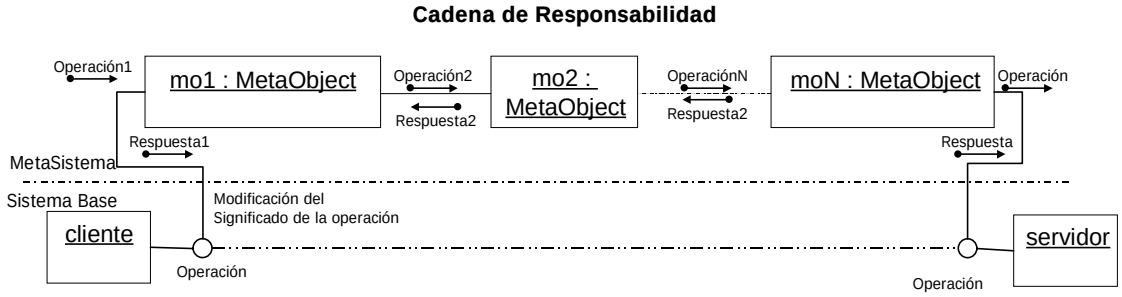


Figura 7.19 :Utilización del patrón *Chain of Responsibility* para asociar múltiples metaobjetos.

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 7.20, cada uno de los elementos de un compositor puede ser un metaobjeto u otro compositor.

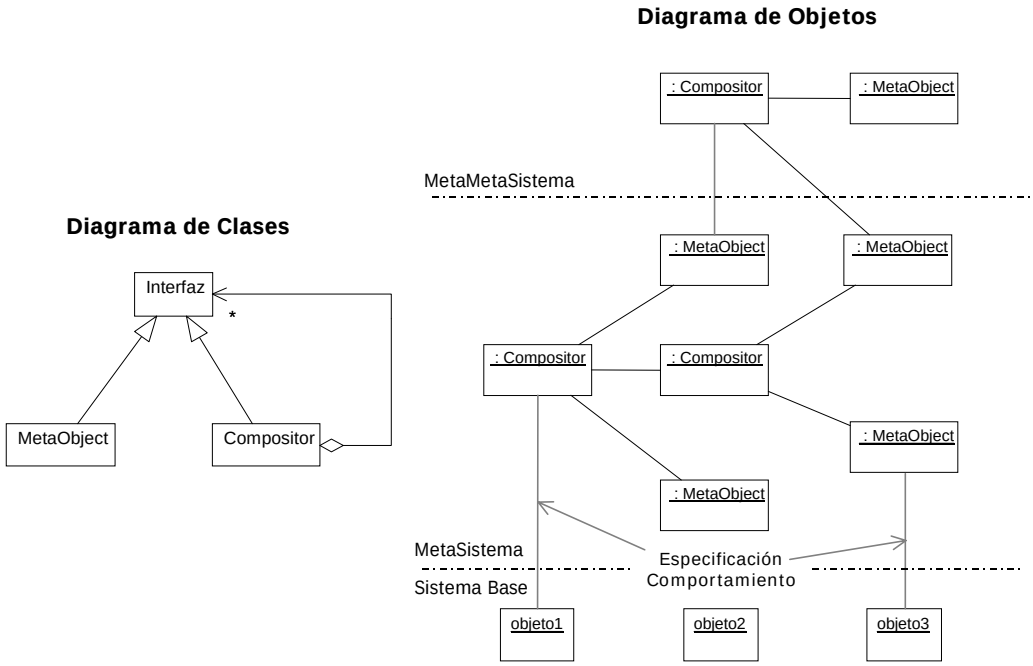


Figura 7.20: Utilización del patrón *Composite* para asociar múltiples metaobjetos.

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*.

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 7.21. 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 [Sun97d]. 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 7.21, este código reside en la implementación de los métodos `beforeMethod` y `afterMethod`).

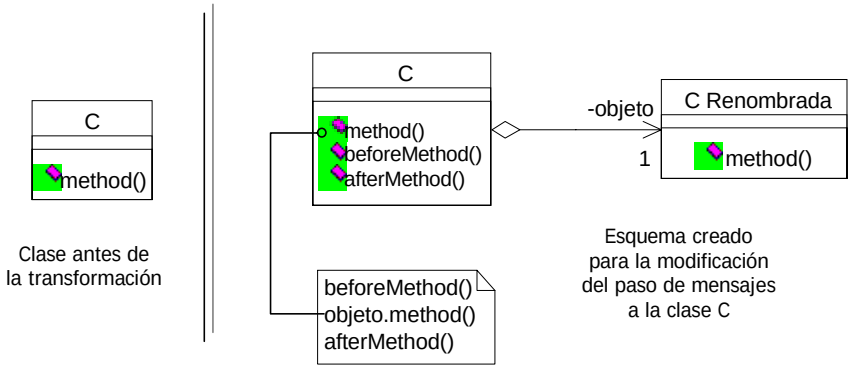


Figura 7.21: Transformación de una clase en Dalang para obtener reflectividad.

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.

NeoClasstalk

Tras los estudios de reflectividad estructural llevados a cabo con el sistema ObjV-lisp (analizado en § 7.2), 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[32], 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 metacla-

[32] Al igual que en los lenguajes orientados a objetos basados en clases, un objeto es una instancia de una clase, el concepto de metaclase define una clase como instancia de una metaclase –la cual define el comportamiento de todas sus clases asociadas.

ses 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 7.22, existe una metaclase por omisión denominada `StandardClass`; esta metaclase define el comportamiento general de una clase en el lenguaje de programación Smalltalk. El nuevo comportamiento deseado deberá implementarse en una nueva metaclase, derivada de `StandardClass`.

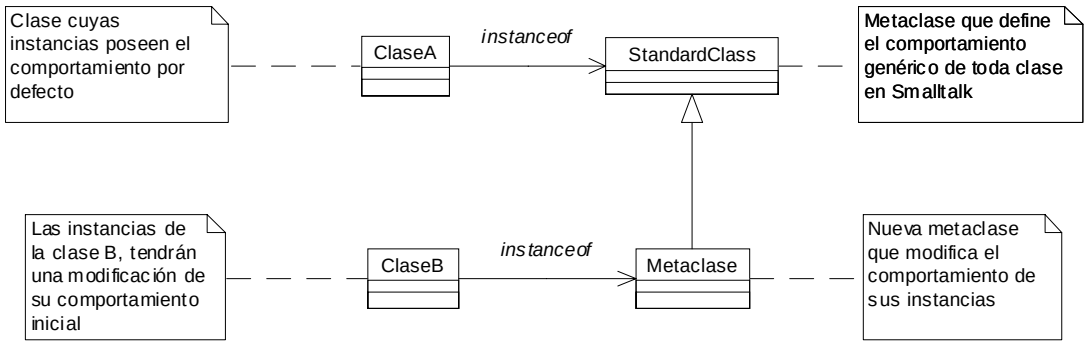


Figura 7.22: MOP de NeoClasstalk utilizando metaclases.

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.

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[33].

- 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].

7.4.1 Aportaciones y Carencias de los Sistemas Estudiados

En la totalidad de los MOPs estudiados, se permite la modificación dinámica de parte del comportamiento del mismo (§ 2.3.3), y, en casos como Dalang, dicha modificación se puede realizar en fase de compilación para obtener una mayor eficiencia en tiempo de ejecución.

El concepto de MOP establece un modo de acceso del sistema base al metasistema, identifica el comportamiento que puede ser modificado. El establecimiento de este protocolo previamente a la ejecución de la aplicación supone una restricción a priori de la semántica que podrá ser modificada dinámicamente. De esta forma, un MOP implica una restricción en la que un sistema puede modificar su propio comportamiento (§ 2.3.4).

[33] Este proceso de convertir datos en computación, lo definimos “descosificación” al tratarse del proceso contrario de “cosificación” –definido en el capítulo 6.

Una proposición para resolver esta limitación de los MOPs pasa por ampliar éste cuando sea necesario [Golm98]; verbigracia, si un MOP no contempla la modificación de la semántica de la creación de objetos, podemos modificarlo para que sea adaptable. Sin embargo, como se muestra en la Figura 7.23, la modificación del MOP supone la modificación del intérprete, dando lugar a distintas versiones del mismo y a la pérdida de la portabilidad del código existente para las versiones anteriores [Ortín2001].

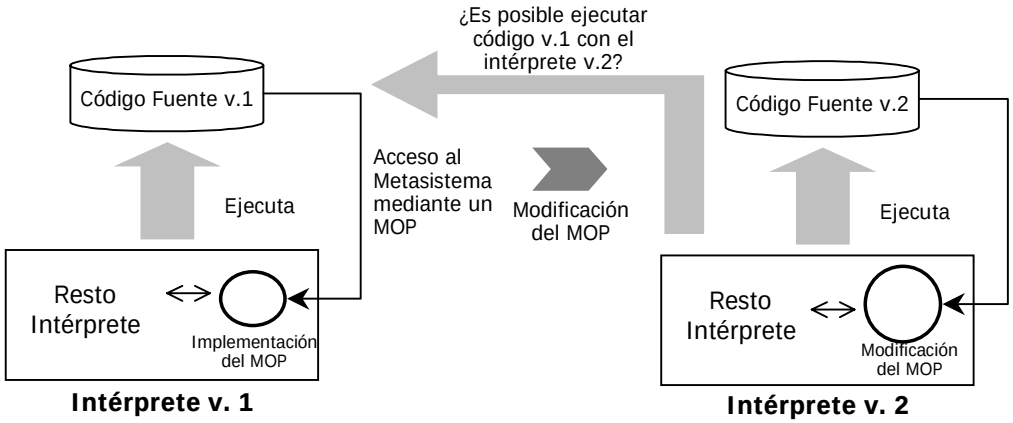


Figura 7.23: Pérdida de la portabilidad código de un intérprete, al modificar su MOP.

Los sistemas basados en MOPs otorgan una flexibilidad dinámica de su comportamiento pero, de forma contraria a las técnicas estudiadas en § 7.2, carecen de la posibilidad de modificar el lenguaje con el que son desarrolladas sus aplicaciones (§ 2.3.5) –se limitan a modificar la semántica de éstos.

La mayor carencia de los sistemas reflectivos es su eficiencia en ejecución. La utilización de intérpretes es más común, pero las aplicaciones finales poseen tiempos de ejecución más elevados que si hubieren sido compiladas a código nativo. A raíz de analizar los sistemas estudiados podemos decir:

1. La implementación de un MOP en un sistema interpretado (por ejemplo, MetaXa) es más sencilla y menos eficiente que el desarrollo de un traductor a un lenguaje compilable, como por ejemplo Iguana. En este caso, el sistema debe poseer información dinámica adicional para poder acceder y modificar ésta en tiempo de ejecución.
   Un ejemplo de la complejidad mencionada es la posibilidad de conocer dinámicamente el tipo de un objeto en C++ (RTTI, *RunTime Type Information*) [Stroustrup98]. Esta característica supone modificar la generación de código, para que todo objeto posea la información propia de su tipo –generalmente la ejecución de una aplicación nativa no necesita esta información y por tanto no se genera.

2. Puesto que los sistemas compilados poseen mayor eficiencia frente a los interpretados, que ofrecen una mayor sencillez a la hora de implementar sistemas flexibles, la unión de las dos técnicas de generación de aplicaciones puede dar lugar a un compromiso eficaz.
   En el caso de Cognac, todo el sistema se ejecuta en tiempo dinámico excepto aquella parte que se identifica como reflectiva; en este momento un intérprete ejecuta el código intermedio que define el nuevo comportamiento. Para el sistema Iguana todo el código es traducido sin información dinámica, salvo aquél que va a ser adaptado.

3. El desarrollo de un MOP en dos niveles de interpretación (Closette) es más sencillo que si sólo elegimos uno (MetaXa). Si necesitamos modificar el MOP

de Closette, deberemos hacerlo sobre el primer intérprete; en el caso de MetaXa, deberemos recodificar la máquina virtual.

Finalmente comentar, poniendo por ejemplo a Moostrap, que la reflectividad estructural y computacional son dos mecanismos útiles a la hora de desarrollar un sistema extensible y adaptable, a partir de un conjunto de primitivas reducidas (§ 2.2.1). Los modelos computacionales orientados a objetos basados en prototipos, facilitan su implementación.

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 implementaciones de intérpretes capaces de ofrecer esta abstracción se han denominado intérpretes metacirculares (*metacircular interpreters*) [Wand88].

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.

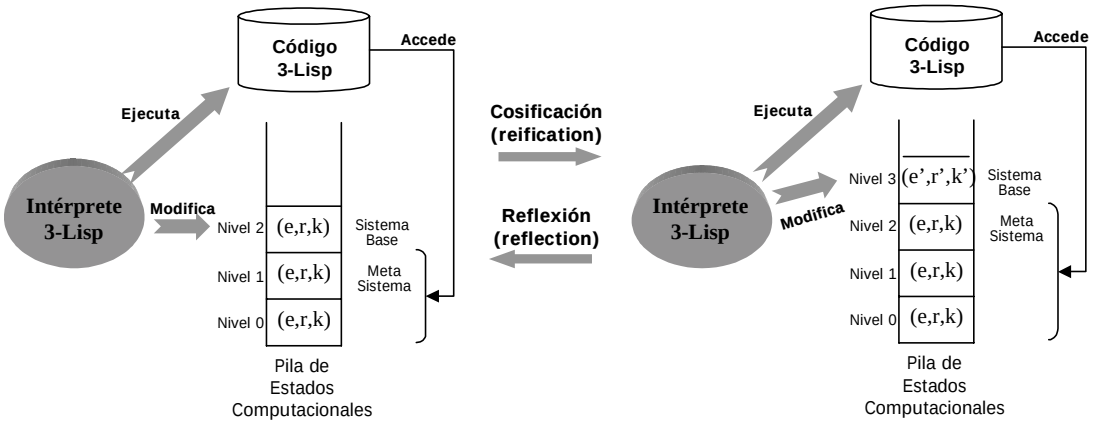


Figura 7.24: Implementación de un intérprete metacircular de 3-Lisp.

En la Figura 7.24 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:

1. Expresividad computacional mediante un único lenguaje. Puesto que realmente existe un único intérprete (ver Figura 7.24), éste estará obligado a animar un solo lenguaje de programación. En todos los niveles computacionales deberá utilizarse por tanto el mismo lenguaje de programación.

2. 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.

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 define el comportamiento de un conjunto de objetos del sistema base. Para implementar la adición de metagrupos, surgen determinadas ampliaciones del sistema:

1. Nuevos objetos de núcleo (*kernel objects*) para gestionar los grupos: The Group Mangager, The Primary Metaobject Generator y The Primary Evaluator.
2. 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 "⇑").
3. 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.

MetaJ

MetaJ [Doudence99] 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:

1. 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.
2. 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 7.25 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.

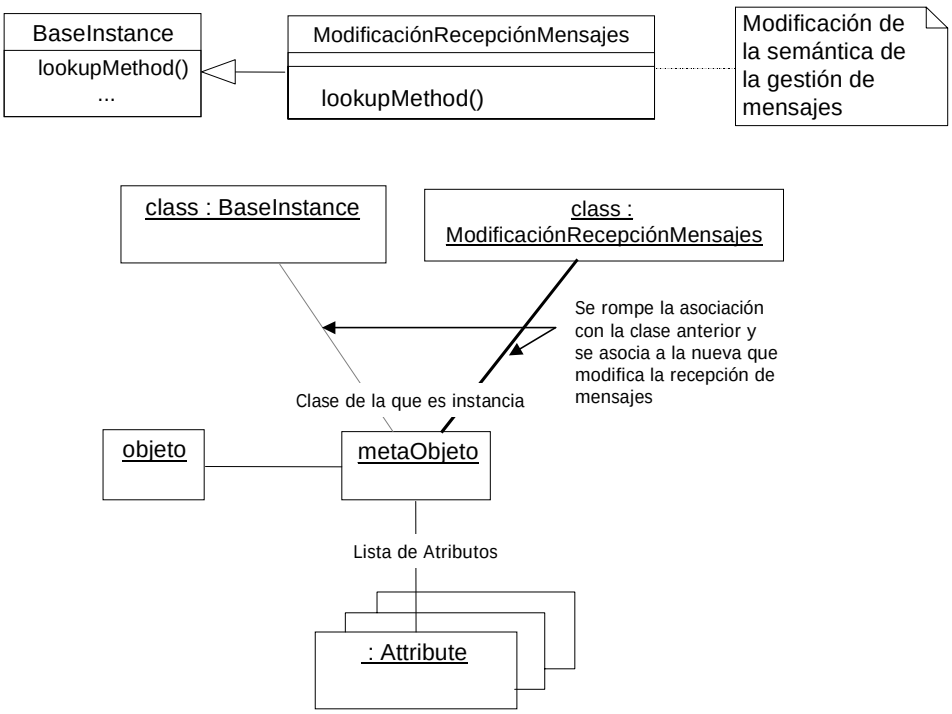


Figura 7.25: Modificación de la clase de un objeto, para obtener la modificación de la semántica de la recepción de mensajes.

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.

7.5.1 Aportaciones y Carencias de los Sistemas Estudiados

Los sistemas estudiados 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]. Un sistema de seguridad en la utilización de la reflectividad sería imprescindible en este caso –§ 6.4.

En los sistemas estudiados, 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 (§ 2.3.3); el intérprete es monolítico e invariable.

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. No existe pues, una flexibilidad no restringida a priori (§ 2.3.4). Además, la modificación a llevar a cabo en el comportamiento ha de especificarse en tiempo de compilación (§ 2.4.8).

Como conclusión, cabe mencionar que a la hora de desarrollar un sistema reflectivo, es más útil ahondar en la cantidad de información a cosificar y el modo en el que ésta pueda ser expresada, que aumentar el número de niveles computacionales cosificables.

Un punto adicional a destacar, propio del sistema ABCL/R2, es su definición de metagrupos. Vemos cómo se utiliza este concepto para agrupar el comportamiento de un conjunto de objetos en una sola abstracción. Para conseguirlo, se introducen un conjunto de entidades adicionales y dos torres de interpretación paralelas. Si bien la agrupación de metaobjetos puede ser atrayente para reducir la complejidad del metasistema, deberíamos buscar un mecanismo auxiliar más sencillo para conseguir dicha funcionalidad –los objetos rasgo o *trait* de los sistemas orientados a objetos basados en prototipos (capítulo 8), nos pueden ofrecer esta abstracción de un modo sencillo.

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 8 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.

Puesto que el concepto de sistema reflectivo puede ser catalogado de diversos modos (clasificación realizada en el capítulo anterior), analizaremos globalmente los sistemas reflectivos teniendo en cuenta tres criterios: cuándo se produce la reflexión, qué se refleja y el número de niveles computacionales utilizado.

7.6.1 Momento en el que se Produce el Reflejo

La reflectividad en tiempo de ejecución otorga un elevado grado de flexibilidad al sistema, puesto que éste puede adaptarse a contextos impredecibles en fase de diseño. Cuando una aplicación necesita poder especificar nuevos requisitos dinámicamente, la reflectividad en tiempo de compilación no es suficiente.

Sin embargo, la reflectividad estática posee una ventaja directa sobre la dinámica: la eficiencia de las aplicaciones en tiempo de ejecución. La adaptabilidad dinámica de un sistema produce una ralentización del mismo en su ejecución.

Desde el punto de vista empírico, podemos ver cómo los sistemas estáticos ofrecen reflectividad del lenguaje de programación, cuando esto no ocurre en ningún sistema dinámico; el lenguaje de programación en estos casos se mantiene inamovible.

7.6.2 Información Reflejada

El primer nivel de información a reflejar es la estructura del sistema en un modo de sólo lectura: introspección (§ 7.1). La característica práctica de este nivel de reflectividad queda patente en el número de sistemas comerciales que la utilizan. Permite desarrollar

fácilmente sistemas de componentes, de persistencia, comprobaciones de tipo dinámicas, o *middlewares* de distribución.

Para el sistema de computación flexible buscado en esta tesis, además de todas las utilidades prácticas estudiadas en § 7.1, ofrece un mecanismo de autodocumentación real y dinámica, y un conocimiento dinámico exacto del estado del sistema.

El segundo grado de información a reflejar es la reflectividad estructural, en la que se permite tanto el acceso como la modificación dinámica de la estructura del sistema. A nivel práctico existen muchas posibilidades para este tipo de sistemas, muchas de ellas todavía no explotadas. Ejemplos pueden ser interfaces gráficas adaptables mediante la incrustación, eliminación y modificación dinámica de la estructura de los objetos gráficos, aplicaciones de bases de datos que trabajen con un abanico de información adaptable en tiempo de ejecución, o la apertura a un nuevo modo de programación adaptable dinámicamente [Golm98] y creación de nuevos patrones de diseño [Ferreira98].

Para el desarrollo de nuestro sistema, al igual que fue utilizada en Smalltalk, Self y Moostrap, la reflectividad estructural puede emplearse para hacer un sistema extendido de un conjunto reducido de primitivas computacionales, programando la mayor parte de éste en su propio lenguaje y consiguiendo así adaptabilidad –el sistema posee la capacidad de modificarse a sí mismo, al estar escrito en su propio lenguaje– y portabilidad –cualquier intérprete de las primitivas computacionales básicas podrá ejecutar el sistema en su totalidad.

Cuando la semántica del sistema puede modificarse, nos encontramos en el tercer nivel de esta clasificación: reflectividad computacional. Éste ofrece una flexibilidad elevada para todo el sistema en su conjunto. Se ha utilizado en la mayoría de casos a nivel de prototipo, aplicándose a depuradores (*debuggers*), compilación dinámica (JIT, *Just In Time compilation*), desarrollo de aplicaciones en tiempo real, o creación de sistemas de persistencia y distribución.

En los sistemas estudiados aparecen limitaciones respecto al grado de modificación de la semántica adaptable, y a la imposibilidad de modificar el lenguaje de programación (§ 7.4.1). Ambas restricciones han sido especificadas como necesaria su superación en esta tesis –requisitos § 2.3.4 y § 2.3.5). Por lo tanto, nuestro sistema deberá tener un mecanismo no restrictivo de modificación de su semántica, así como la capacidad de modificar y seleccionar dinámicamente su lenguaje de programación (cuarto y último nivel de información a reflejar).

7.6.3 Niveles Computacionales Reflectivos

En la torre de intérpretes enunciada por Smith [Smith82], analizada en el capítulo anterior, el acceso de un sistema a su metasistema se representaba como el salto al intérprete que ejecutaba dicha aplicación. En un sistema reflectivo, podemos preguntarnos cuántos niveles computacionales necesitamos y qué ganaríamos introduciendo más.

Coincidiendo con la mayoría de los autores, un sistema altamente flexible y manejable es aquél que ofrece una elevada “anchura” y no “altura” de su torre de intérpretes. Esto quiere decir que es más útil obtener una forma sencilla y potente de modificación del metasistema desde el sistema base –anchura de la torre–, que la capacidad de modificar el comportamiento del lenguaje que especifica el comportamiento, denominado metacomportamiento –altura de la torre.

Como hemos especificado en § 7.5.1, la posibilidad de acceder un número infinito de niveles de computación, puede hacer al programador perder la semántica real del siste-

ma con el que está trabajando. Por esto, reduciremos el nivel computacional necesario para nuestro sistema a dos y trataremos de darle la mayor expresividad posible; sin restricciones.

CAPÍTULO 8:
LENGUAJES ORIENTADOS A OBJETOS BASADOS EN PROTOTIPOS

Según la noción utilizada para agrupar objetos de igual comportamiento en un modelo computacional orientado a objetos, es posible establecer la siguiente clasificación de lenguajes [Evins94]:

- Lenguajes orientados a objetos basados en clases.
- Lenguajes orientados a objetos basados en prototipos.

Estudiaremos las semejanzas y disimilitudes existentes entre ambos, y analizaremos las ventajas e inconvenientes que cada uno de los modelos aporta.

8.1 Clases y Prototipos

El modelo computacional de objetos basado en clases se apoya en la agrupación de objetos de igual estructura y comportamiento, como instancias de una misma clase [Booch94]. Se establece así una relación necesaria de instanciación entre objetos y clases; no es posible crear y utilizar un objeto, si no se ha definido previamente la clase a la que pertenece.

La estructura estática de un objeto y su comportamiento en función de su estado, son definidos por una clase mediante sus atributos y sus métodos respectivamente. Cuando se crea un objeto como instancia de una clase, su estado quedará definido por un conjunto dinámico de valores para cada uno de los atributos de la estructura definida por su clase, variando éste dinámicamente.

En el modelo computacional basado en objetos, 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 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 o delegación.

Vemos como, 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 8.1, el objeto *trait* `Object` define el comportamiento `toString` de todos los objetos. Del mismo modo, `Point` define el comportamiento de sus dos objetos derivados.

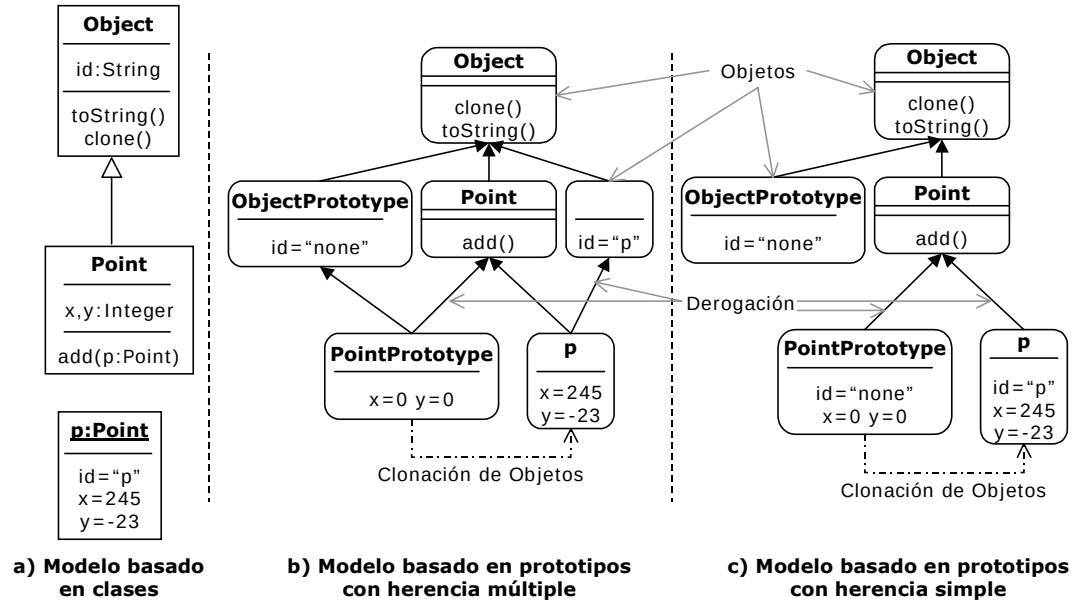


Figura 8.1: Representación de clases y objetos en los dos modelos.

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 8.1, 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.

Como ejemplos de lenguajes orientados a objetos basados en prototipos podemos mencionar a Self [Ungar87, Chambers89], Moostrap [Mulet93], ObjectLisp, Cecil [Chambers93], NewtonScript [Swaine94] o PROXY [Leavenworth93].

8.2 Utilización de Lenguajes Orientados a Objetos Basados en Prototipos

En este apartado evaluaremos las ventajas existentes en la utilización del modelo orientado a objetos basado en prototipos, dejando para el próximo sus inconvenientes frente al que utiliza clases.

8.2.1 Reducción Semántica

Una de las características destacables en el modelo de prototipos, al llevarse a cabo una comparación con su homólogo, reside en la simplicidad obtenida al:

1. Eliminar el concepto de clase. No es necesario crear una clase para todo objeto.
2. Suprimir la dependencia necesaria existente entre un objeto y una clase. Siempre ha de existir entre ellos una relación de instanciación, y el estado del objeto ha de ser coherente con la estructura defina por su clase.
3. Eliminar la comprobación estática de tipos. Un objeto sólo puede recibir los mensajes implementados por su clase y superclases. En un sistema basado en clases, esta comprobación es realizada en tiempo de compilación.

   En el caso del modelo basado en prototipos, al constituirse la herencia como un mecanismo dinámico (delegación), es imposible llevar a cabo esta validación estáticamente; el conocimiento de esta información sólo puede efectuarse dinámicamente[34].

8.2.2 Inexistencia de Pérdida de Expresividad

La sencillez del modelo computacional que se basa únicamente en objetos puede hacernos creer que supone una pérdida en la expresividad de los lenguajes que lo utilizan. Sin embargo, se han realizado estudios que demuestran que la utilización de prototipos no supone pérdida alguna de expresividad [Ungar91, Evins94]: toda semántica representable mediante el modelo de clases puede ser traducida al modelo de prototipos. Su demostración ha quedado patente en la implementación de compiladores de lenguajes basados en clases, como Self y Java, a la plataforma Self que utiliza prototipos [Wolczko96].

Sin ánimo de ahondar en la traducción de conceptos de un modelo a otro, mostraremos un ejemplo de traducción de la noción de miembro de clase, existente en la mayoría de los lenguajes de programación basados en clases.

[34] Es por esta razón por la que los sistemas basados en prototipos utilizan un sistema dinámico de tipos [Chambers89], o bien carecen de su inferencia [Cardelli97].

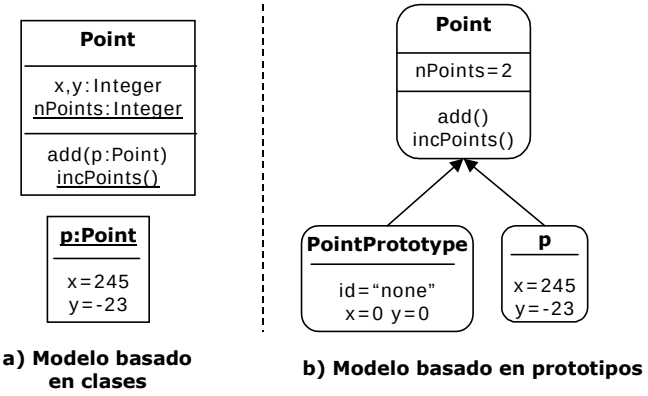


Figura 8.2: Miembros de clase en ambos modelos orientados a objetos.

Como se muestra en la figura anterior, la clase `Point` posee un atributo de clase que cuenta el número de instancias creadas, y un método de clase que permite incrementar éste. La ubicación de ambos en el modelo de prototipos se sitúa en el objeto *trait*. Éste, y no uno de sus derivados, será el que reciba los mensajes de clase y será su propio estado el que determine el resultado de su ejecución (puesto que el método `incPoints` incrementa el atributo `nPoints` que pertenece al objeto *trait*). La utilización de este servicio se demandará mediante la clase (objeto `Point`) y no mediante una de sus instancias.

8.2.3 Traducción Intuitiva de Modelos

La traducción del modelo basado en clases –de un mayor nivel de abstracción– al modelo que utiliza prototipos se produce de un modo intuitivo para el programador. Por ejemplo, el programador de aplicaciones que selecciona un lenguaje de programación como Python [Rossum2001] o Java [Gosling96], ha de tener en mente esta traducción.

Aunque los dos lenguajes mencionados en el párrafo anterior poseen un modelo computacional basado en clases, una vez compilado su código fuente, en fase de ejecución, el modelo computacional empleado está basado en prototipos. En tiempo de ejecución las abstracciones de clases son substituidas por objetos que representan éstas. Así, en Python, el atributo `__class__` de todo objeto nos devuelve su objeto-clase asociado; en Java, el método `getClass` nos devuelve un objeto del tipo `Class`.

Por esta traducción entre modelos intuitiva, la sencillez del modelo de prototipos y la carencia de pérdida en su expresividad, Wolczko propone los prototipos como modelo universal de computación de lenguajes orientados a objetos [Wolczko96]. Llevando a cabo la traducción de cualquier lenguaje a un único modelo, la interacción intuitiva entre aplicaciones se podría lograr independientemente del lenguaje que hay sido utilizado para su construcción.

8.2.4 Coherencia en Entornos Reflectivos

Dos problemas existentes en el campo de las bases de datos son la evolución y mantenimiento de versiones de esquemas. Pueden definirse de la siguiente forma [Roddick95]:

- Evolución de esquemas (*schema evolution*): Capacidad de una base de datos para permitir la modificación de su diseño, sin incurrir en pérdida de información.

- Mantenimiento de versiones de esquemas (*schema versioning*): Se produce cuando, tras haberse llevado a cabo un proceso de evolución del esquema, la base de datos permite acceder a la información tanto de un modo retrospectivo, como mediante la estructura actual, manteniendo así la información para cada versión de esquema existente.

Con el surgimiento de las bases de datos orientadas a objetos, este problema es trasladado al concepto de clase: el esquema –estructura y comportamiento– de un objeto queda determinado por la clase de la que es instancia. ¿Cómo se deberá adaptar un objeto si es modificada su clase?

La cuestión surgida en el punto anterior brota de igual modo en el caso de utilizar un modelo computacional basado en clases que esté dotado de reflectividad estructural (§ 6.3.1). ¿Qué sucede con los estados de los objetos si modifico la estructura de su clase? ¿Cómo puedo modificar la estructura de un único objeto? Existen distintas aproximaciones.

Una de las soluciones aportadas, sin necesidad de eliminar el concepto de clase, es la enfocada a obtener mantenimiento de versiones del esquema. El sistema MetaXa [Golm97], estudiado en § 7.4, implementa esta solución mediante la creación de las denominadas clases sombra (*shadow classes*). En este sistema, siempre que se modifique una clase se crea una copia (sombra) de la clase antigua, se modifica ésta, y se asocian los objetos nuevos a esta clase, manteniendo los existentes como instancias de la versión anterior.

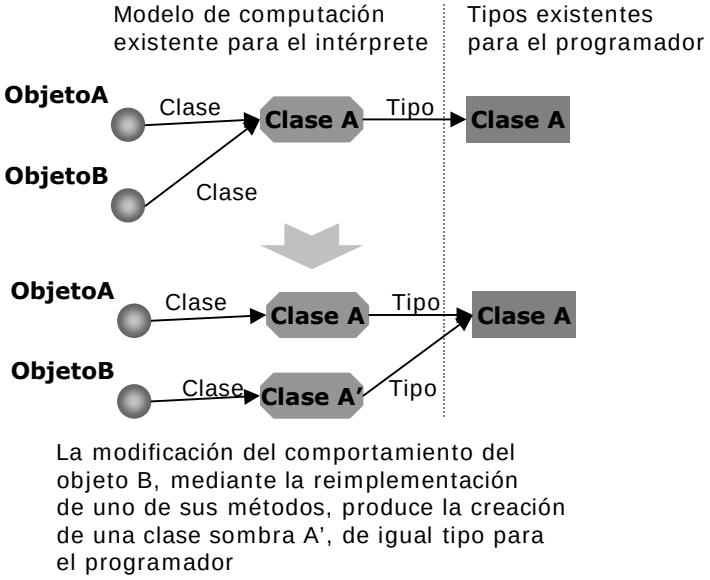


Figura 8.3: Creación de una clase sombra para la modificación del comportamiento de un objeto.

La implementación de este mecanismo ha de tener en cuenta que la clase original y la sombra, aun siendo distintas, deberán poseer la misma identidad, es decir, el programador de aplicaciones no deberá tener noción de la existencia de ambas. La implementación de este mantenimiento de versiones de clases es complejo y conlleva a un determinado número de incoherencias del modelo computacional [Golm97c]. Los propios autores del sistema, en [Golm97c], mencionan que sería más sencilla su implementación si se hiciese uso del modelo basado en prototipos.

La modificación de la estructura y comportamiento de objetos en un sistema basado en prototipos es más sencilla de implementar, y no genera incoherencias en dicho modelo –por ejemplo, el sistema Moostrap [Mulet93]. Distintos escenarios son:

1. La modificación de la estructura de un único objeto se obtiene al modificar directamente éste.
2. La modificación de la estructura de los nuevos objetos a crear se lleva a cabo mediante la modificación del prototipo utilizado.
3. La modificación del comportamiento de todos los objetos de un grupo, se obtiene mediante la manipulación del objeto *trait* correspondiente.
4. La modificación de los nuevos objetos a crear, o de únicamente uno de los existentes, se puede alcanzar mediante la clonación del objeto de comportamiento seleccionado y su posterior modificación.

En este modelo, es el programador el que lleva el peso de agrupar los objetos por comportamiento y estructura; de este modo también puede identificar lo que realmente desea modificar (el grupo, uno en concreto o los sucesivos).

8.3 Conclusiones

En función de la clasificación establecida al comienzo de este capítulo, veíamos cómo los modelos computacionales orientados a objetos podían utilizar clases o simplemente objetos. Como veíamos en el punto anterior, la utilización del modelo computacional basado en prototipos aporta un conjunto de ventajas. Sin embargo, los lenguajes que utilizan clases también poseen virtudes:

- El nivel de abstracción ofrecido por las clases es más elevado, permitiendo expresar al programador un modelo más cercano al problema a resolver.
- La agrupación obligada de objetos mediante la definición de clases exige al programador la división de las aplicaciones a desarrollar en abstracciones a modelar.
- La definición de los tipos de objetos mediante sus clases permite detectar en tiempo de compilación errores de tipo, reduciendo así del número de errores a producidos en tiempo de ejecución y facilitando la labor del desarrollador.

En función del estudio realizado de ambos modelos, podemos afirmar como conclusión que los sistemas basados en clases están más orientados a la programación, mientras que la utilización de prototipos está más acorde con el desarrollo de una plataforma computacional. Como ejemplo pragmático de esto, podemos mencionar cómo determinados lenguajes basados en clases, como Java o Python, traducen su semántica de programación basada en clases a un modelo computacional de objetos, en su fase de ejecución.

CAPÍTULO 9:
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 tesis, 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.

9.1 Capas del Sistema

El sistema está dividido en tres capas o elementos bien diferenciados, que pueden apreciarse en la siguiente figura:

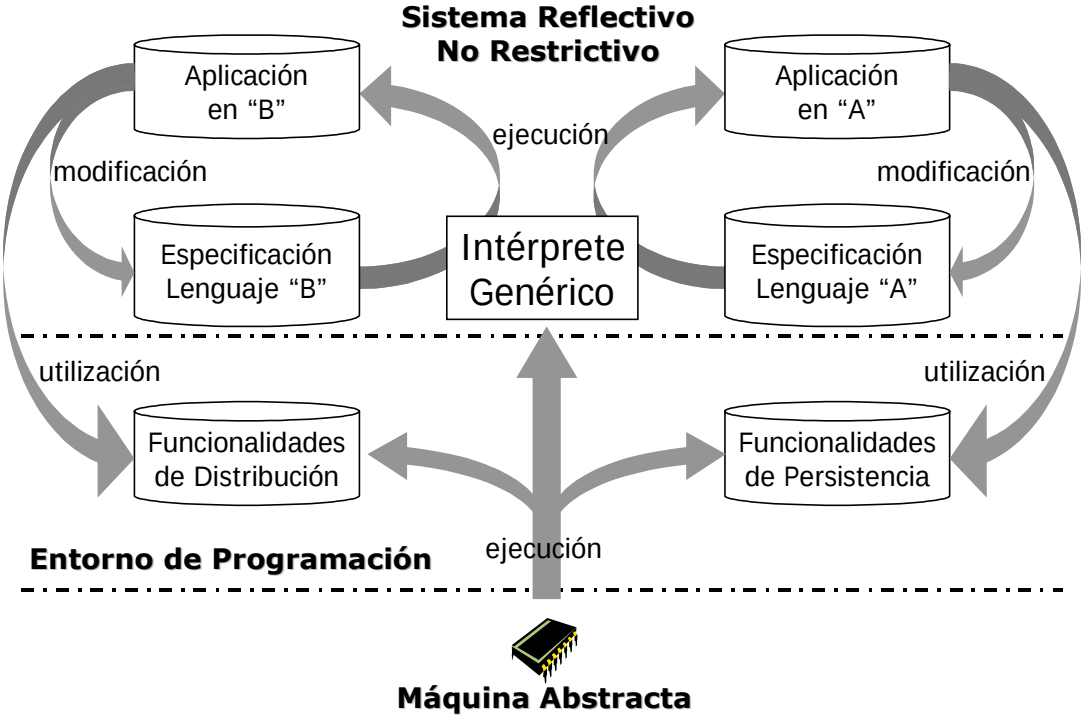


Figura 9.1: Arquitectura del sistema.

9.1.1 Máquina Abstracta

El motor computacional del sistema es la implementación de una máquina abstracta –máquina virtual. Todo el código ejecutado por ésta es portable y, por tanto, independiente de la plataforma física empleada.

Posteriormente ahondaremos en las ventajas de codificar el sistema sobre una máquina abstracta (§ 9.3) y en la arquitectura y diseño de ésta (capítulo 10 y capítulo 12), mas la ventaja a destacar en este momento es que todo el sistema comparte el mismo modelo computacional y éste es independiente de la plataforma.

9.1.2 Entorno de Programación

Sobre la máquina abstracta se desarrolla un código que facilite la labor del programador. Este código es portable, independiente del lenguaje (cualquier aplicación en el sistema, codificada en cualquier lenguaje, puede utilizarlo) e independiente de la plataforma.

La máquina abstracta ha de poseer la característica de ser extensible para, sin necesidad de modificar su implementación, pueda desarrollarse sobre ella un entorno de programación con funcionalidades de un mayor nivel de abstracción, como distribución o persistencia.

9.1.3 Sistema Reflectivo No Restrictivo

Hasta esta tercera y última capa, todas las aplicaciones del sistema se desarrollan sobre el lenguaje nativo de la máquina abstracta. Este lenguaje posee una semántica fija para sus aplicaciones. Mediante esta capa se otorga independencia del lenguaje al sistema y flexibilidad dinámica, sin restricciones previas, de los lenguajes a utilizar.

Un intérprete genérico toma la especificación de un lenguaje y ejecuta una aplicación codificada en éste. La aplicación puede hacer uso del entorno de programación e interactuar con otras aplicaciones codificadas en otros lenguajes. Adicionalmente podrá modificar la especificación del lenguaje utilizado, reflejándose estos cambios en la adaptación de la semántica de la propia aplicación, de forma instantánea.

9.2 Único Modelo Computacional de Objetos

Adicionalmente al conjunto de requisitos obtenidos por la utilización de una máquina abstracta –véase el capítulo 16–, la utilización de ésta está justificada por 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 la máquina virtual[35], y que las aplicaciones puedan interactuar entre sí independientemente de su lenguaje de programación. La codificación del entorno de programación sobre el lenguaje propio de la máquina, facilita la utilización de éste sin dependencia alguna del lenguaje a utilizar.

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 de la máquina abstracta. 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.

[35] Para saber más acerca del modelo computacional de objetos utilizado por la máquina abstracta, así como la justificación de su elección, consúltese el capítulo 10.

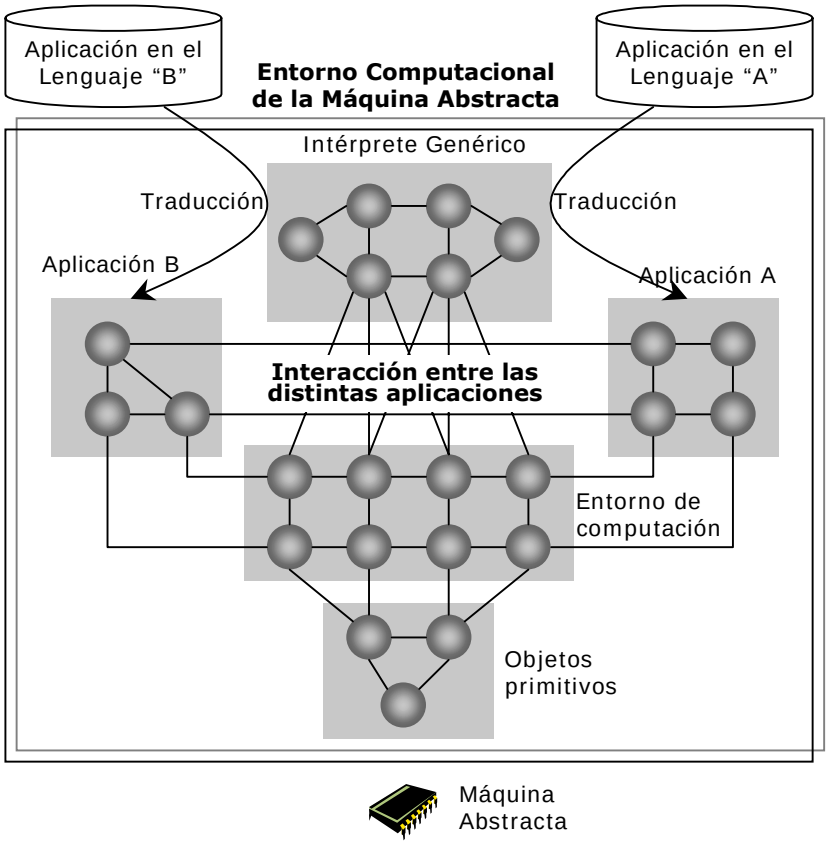


Figura 9.2: Interacción entre distintos objetos de un mismo espacio computacional.

Como se muestra en la figura anterior, la máquina abstracta parte de un conjunto mínimo de objetos primitivos –funcionalidad básica; la reducción del conjunto de éstos facilita su portabilidad. Haciendo uso de la extensibilidad de esta plataforma virtual, se desarrolla código que eleva el nivel de abstracción en la programación de la plataforma –entorno de programación–, facilitando la tarea del programador.

El programador elige un lenguaje de programación y, en la interpretación de éste, su modelo es traducido al propio de la plataforma abstracta. Con el único modelo de objetos existente, la interacción de aplicaciones es directa, reutilizando en todo momento las funcionalidades desarrolladas, 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.

9.3 Máquina Abstracta

La máquina abstracta supone el motor computacional del conjunto del sistema. La migración de éste a una plataforma pasa por la recompilación de su implementación para el nuevo sistema nativo. El modelo computacional de objetos definido por ésta representará el propio del sistema y la forma en la que las distintas aplicaciones interactúen entre sí.

Éstos son los objetivos generales a alcanzar:

9.3.1 Conjunto Reducido de Primitivas

La máquina abstracta ha de poder implantarse en entornos heterogéneos. Cualquier sistema computacional, por reducido que éste sea, deberá ser capaz de instalar una implementación. La reducción del número de primitivas computacionales facilita su implantación en plataformas heterogéneas.

El diseño de una plataforma de computación reducida facilita la migración del sistema. Si el código de la máquina es portable, la portabilidad total del sistema se reduce a la implantación de la máquina en distintas plataformas. Por lo tanto, la reducción de su tamaño minimiza el esfuerzo para llevar a cabo nuevas implantaciones.

9.3.2 Mecanismo de Extensibilidad

Dado que el número de primitivas computacionales de la máquina abstracta debe ser lo más reducido posible, el lenguaje de programación poseerá un bajo nivel de abstracción. Para ofrecer al programador de aplicaciones un mayor conjunto de funcionalidades y un mayor nivel de abstracción, es necesario que la máquina abstracta posea un mecanismo de extensibilidad.

Un problema típico en la creación de máquinas abstractas es la ampliación de sus funcionalidades mediante la inclusión de instrucciones, aumentando así la funcionalidad e implementación de ésta. Un ejemplo es el desarrollo de la máquina abstracta del sistema integral orientado a objetos Oviedo3 [Álvarez97]. Inicialmente ofrecía las características propias de una plataforma orientada a objetos pura. Posteriormente, distintas versiones fueron implementándose añadiendo características de persistencia [Ortin97], seguridad mediante capacidades [Díaz2000], planificadores genéricos de tareas [Tajes2000] y un sistema de distribución [Álvarez2000].

La existencia de múltiples versiones de implementaciones de la máquina abstracta, la pérdida de portabilidad de su código, la complejidad de implementar una única máquina virtual con el conjunto total de sus características, y la pérdida de sencillez en su implementación para poder implantarla en entornos heterogéneos, son los resultados de seguir este diseño erróneo.

La ampliación de la abstracción del sistema ha de codificarse en el propio lenguaje de programación de la máquina, a partir de sus primitivas computacionales. El hecho de que éste sea código propio de la máquina, hace que sea portable a cualquier plataforma. El resultado es una máquina virtual de sencilla implementación y, sobre ella, un código portable que amplia el nivel de abstracción para el programador, cualquiera que sea la plataforma existente.

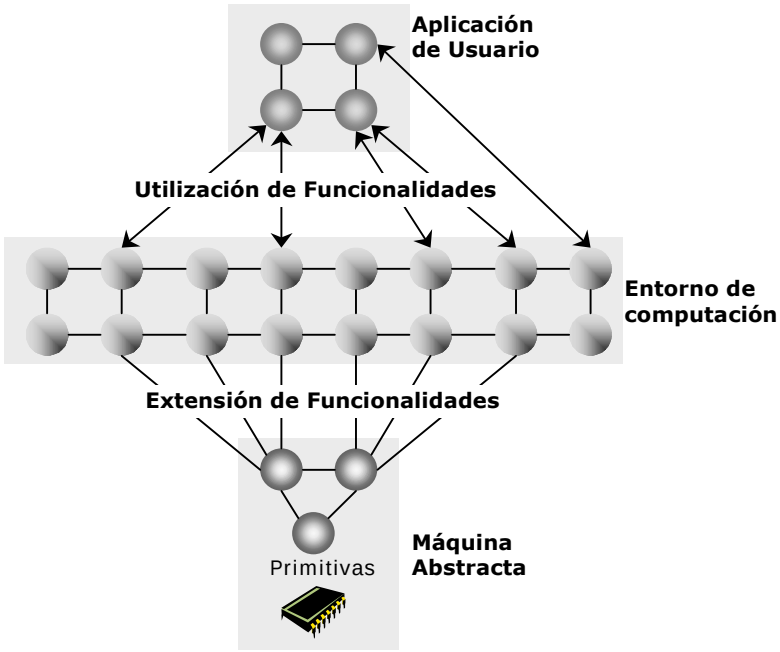


Figura 9.3: Extensibilidad de las funcionalidades primitivas de la máquina abstracta.

Para que lo propuesto pueda llevarse a cabo sin necesidad de modificar la implementación de la máquina abstracta, es necesario que ésta posea un mecanismo de extensibilidad. Un ejemplo de esta extensión del nivel de abstracción, es la codificación de la segunda capa de este sistema: el entorno de programación.

9.3.3 Selección del Modelo Computacional

El nivel de abstracción proporcionado por la máquina abstracta es propio del modelo computacional utilizado por todo el sistema. La selección del correcto nivel de abstracción que ofrezca el modelo de computación es una tarea difícil: un bajo nivel de abstracción hace más compleja la interoperabilidad de aplicaciones, mientras que si éste es demasiado elevado, podrá suponer que plataforma sea dependiente del lenguaje.

El modelo a elegir ha de permitir representar la computación de cualquier lenguaje de programación, ser sencillo para poder ser implementada su computación de un modo reducido, y no su suponer un cambio de expresividad elevado para que puedan interactuar entre sí las aplicaciones codificadas en distintos lenguajes de un modo natural.

9.3.4 Interacción Directa entre Aplicaciones

Puesto que el objetivo de esta tesis es conseguir un entorno de programación de aplicaciones flexible, que permita su interacción indistintamente del lenguaje de programación elegido para desarrollarlas, el motor computacional del mismo –la máquina abstracta– ha de facilitar la comunicación entre ellas.

Si tomamos por ejemplo la máquina virtual de Java [Sun95], cada aplicación que se ejecuta necesita un proceso con la ejecución de la máquina virtual que interprete este código, suponiendo así la interacción entre estas aplicaciones una comunicación entre dos procesos distintos dentro del sistema operativo existente. Mecanismos adicionales como el uso de *sockets* o RPCs (*Remote Procedure Calls*), son comúnmente utilizados en la intercomunicación de estos procesos.

Nuestra máquina abstracta deberá ser capaz de ejecutar aplicaciones en paralelo, facilitando la intercomunicación entre éstas, y estableciendo un espacio de nombres único en el que el paso de mensajes entre las distintas aplicaciones sea similar a la invocación de métodos de un objeto del mismo programa.

9.3.5 Evaluación Dinámica de Datos como Código

Esta característica es utilizada en la implementación de la tercera capa del sistema; supone la posibilidad de crear y modificar dinámicamente una información –datos–, y hacer posteriormente que la máquina abstracta los interprete como computación –código.

Si, como mencionábamos en § 6.1, el hecho de representar el comportamiento como datos manipulables por un proceso se define como "cosificación", el proceso contrario en el que éstos se evalúan, puede definirse como "descosificación". Ejemplos de esta característica en lenguajes de programación conocidos son la posibilidad de evaluar una lista como código mediante la función `eval` de Lisp [Steele90], o la función `exec` de Python [Rossum2001] tomando cadenas de caracteres.

Esta peculiaridad de la máquina es utilizada por el intérprete genérico de la capa del sistema reflectivo. Como se muestra en la Figura 9.1, una aplicación deberá ser capaz de modificar la especificación de su lenguaje. Para ello, generará dinámicamente un código evaluable por la máquina –no por el intérprete– que, al ejecutarse en el entorno de ésta, podrá modificar la especificación del lenguaje –existente en el mismo nivel computacional. Este sistema será detallado en el capítulo 11.

9.4 Entorno de Programación

Respecto al software flexible desarrollado sobre la plataforma abstracta, enfocado a elevar el nivel de abstracción del sistema, debe ser desarrollado siguiendo un conjunto de objetivos básicos. Enunciaremos éstos para posteriormente estudiar su diseño en el capítulo 10.

9.4.1 Portabilidad e Independencia del Lenguaje

La codificación del entorno de programación ha de ser independiente de toda plataforma física y lenguaje de programación. 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. Del mismo modo, cualquier aplicación desarrollada sobre cualquier lenguaje, deberá poder utilizar los servicios ofertados por este código; no deberá existir restricción alguna al respecto.

La única tarea a tener en cuenta a la hora de implantar el entorno de programación en una plataforma, es la selección del subconjunto de funcionalidades que deseemos instalar. En función de las necesidades del sistema, de su potencia de cómputo y de la memoria disponible, el sistema demandará una parte o la totalidad de las funcionalidades del entorno de programación.

9.4.2 Adaptabilidad

Puesto que el objetivo principal de esta tesis es el desarrollo de un sistema computacional flexible, la adaptabilidad de sus funcionalidades es una característica primordial. El

entorno de programación deberá aumentar el nivel de abstracción del sistema, ofreciendo nuevas funcionalidades; sin embargo, es importante que éstas sean adaptables.

Si el entorno de programación ofrece funcionalidades de persistencia, éstas deberán ser flexibles respecto al entorno físico utilizado para almacenar los objetos. Además, si el programador deseare introducir su propio sistema de persistencia, su diseño debería estar enfocado a minimizar el número de pasos necesarios para implantarlo.

La adaptabilidad también ha de aplicarse a las primitivas de la máquina abstracta. Si codificamos todo el software utilizando estas primitivas, las aplicaciones perderán adaptabilidad al ser variables de computaciones constantes –la semántica de las primitivas se mantiene invariable. Sin embargo, del mismo modo que fue diseñada la imagen computacional de Smalltalk [Ingalls78], la semántica de las primitivas pueden extenderse con nuevas rutinas adaptables[36]. Si se programa haciendo uso de las segundas, el código generado podrá adaptar su funcionalidad modificando la extensión de las primitivas.

El objetivo buscado es que la totalidad de las funcionalidades ofrecidas por esta capa del sistema sean adaptables a las distintas necesidades de los programadores.

9.4.3 Introspección

La implantación del entorno de programación en sistemas computacionales heterogéneos requiere el conocimiento dinámico del subconjunto de funcionalidades instaladas en cada plataforma. Si una plataforma no ofrece las características de persistencia por limitaciones de espacio, el código de usuario deberá tener la posibilidad de consultar si esta funcionalidad ha sido implantada, antes de hacer uso de ella.

El objetivo de desarrollo del sistema en entornos heterogéneos, y la posibilidad de crear aplicaciones en el conjunto de plataformas existentes como si de un único ordenador se tratase, requiere la existencia de un mecanismo que permita analizar y conocer un sistema desde sí mismo –introspección.

9.5 Sistema Reflectivo No Restrictivo

En el análisis de cualquier sistema reflectivo, siempre se ha de tener en cuenta el debate "*Hamiltonians versus Jeffersonians*" [Foote92] que expresábamos en § 6.4. 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, podría diseñarse un sistema de seguridad encargado de controlar el nivel de reflexión permitido a cada uno de los usuarios. Sin embargo, este propósito queda fuera de los objetivos marcados dentro de esta tesis.

A continuación analizaremos los puntos más significativos de esta capa del sistema.

[36] En Smalltalk, para acceder a un atributo indexado de un objeto la primitiva a utilizar es `basicAt`. Sin embargo, su utilización es desaconsejada frente al mensaje `at` [Mevel87]; este segundo método se implementa haciendo uso de la primitiva, teniendo en cuenta la jerarquía objetos establecida por la relación de herencia.

9.5.1 Características Adaptables

Basándose en la clasificación de adaptabilidad descrita en § 6.3, deberá otorgarse al sistema las siguientes características de adaptabilidad:

1. Conocimiento dinámico del entorno. El conjunto del sistema ha de ser introspectivo, ofreciendo la posibilidad de conocer su estado y descripción dinámicamente.
2. Acceso y modificación de su estructura. La estructura de los objetos existentes deberá ser manipulable en tiempo de ejecución, para conseguir adaptabilidad estructural dinámica.
3. Semántica computacional. La semántica computacional del sistema deberá poder conocerse, modificarse y ampliarse. De esta forma, una aplicación en ejecución podrá ser adaptada sin necesidad de modificar su código fuente ni finalizar su ejecución.
4. 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.

Como estudiamos en el capítulo 7, no existe sistema alguno dotado de la capacidad de ser adaptable en todas estas características.

9.5.2 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 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.

9.5.3 Grado de Flexibilidad de la Semántica Computacional

Como hemos definido en § 9.5.1, 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.

El modo en el que expresamos la semántica de un lenguaje es mediante otro lenguaje de especificación de semánticas [Mosses92]. Si accedemos y modificamos la semántica del lenguaje de programación, lo haremos mediante el lenguaje de especificación de semánticas. 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 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.

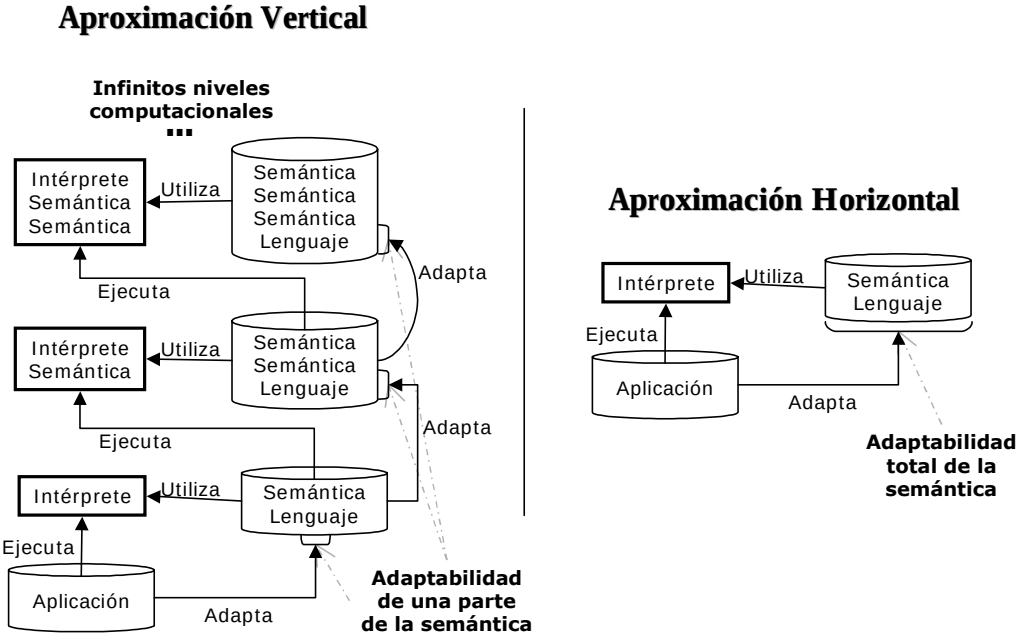


Figura 9.4: Distintas aproximaciones en la consecución de adaptabilidad en la semántica del sistema.

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.

9.5.4 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 (§ 7.4.1). 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.

CAPÍTULO 10:
ARQUITECTURA DE LA MÁQUINA ABSTRACTA

Partiendo de los requisitos impuestos en el capítulo 2 y los objetivos globales de esta tesis, analizaremos las alternativas estudiadas en la sección del estado del arte y, basándonos en su evaluación, adoptaremos los criterios de diseño generales (arquitectura) en la construcción de la máquina abstracta.

Fundamentalmente, los principales objetivos a obtener en el diseño de la plataforma, se pueden reducir a los siguientes: portabilidad, heterogeneidad, extensibilidad y adaptabilidad. Centrándonos en estos pilares, describiremos su arquitectura para posteriormente, en el capítulo 12, presentar su diseño.

En el apéndice A se describe finalmente un diseño para la implementación de una máquina virtual acorde a la máquina abstracta descrita. Se debe recordar la diferencia existente entre máquina abstracta y virtual, descrita en capítulo 3: una máquina virtual es un intérprete software de la especificación de una máquina abstracta.

10.1 Características Principales de la Máquina Abstracta

En este apartado, enunciaremos los requisitos de la máquina abstracta, evaluaremos posibles alternativas, y justificaremos y explicaremos la solución a emplear.

10.1.1 Reducción de Primitivas Computacionales

Buscando la portabilidad global del sistema y la implantación de éste en entornos computacionales heterogéneos, el número de primitivas de la plataforma deberá reducirse al mínimo. Seleccionando la cantidad mínima de primitivas a implementar por la máquina, obtenemos dos beneficios:

1. La implantación de ésta en múltiples plataformas, al ser reducida la implementación de su máquina virtual, se llevará a cabo de un modo sencillo, obteniendo así una elevada portabilidad.
2. Su tamaño reducido implica la posibilidad de implantación en sistemas computacionales de capacidad restringida; tanto en dispositivos físicos de capacidad

limitada de cómputo, como en otras aplicaciones que deseen interpretar su lenguaje[37].

La obtención de los dos beneficios anteriores producidos al reducir las operaciones primitivas de nuestro sistema de computación, quedó demostrada empíricamente en el desarrollo de los sistemas operativos basados en micronúcleo, estudiados en § 5.6.

En el diseño de la máquina abstracta distinguiremos entre dos tipos distintos de primitivas, siendo un conjunto de éstas variable mientras que el otro se deberá mantener inalterable:

1. Primitivas computacionales. Definen el sistema computacional de la máquina abstracta. Supone la semántica del lenguaje de la máquina que toda máquina virtual deberá implementar. El conjunto de estas primitivas es invariable. Ejemplos de estas operaciones pueden ser el modo en el que se crea un objeto o se interpreta el paso de un mensaje.

2. Primitivas operacionales. Suponen la semántica primitiva de una operación independiente del funcionamiento de la máquina. Un ejemplo puede ser la forma en la que se suman dos enteros o el almacenamiento de un *byte* en un sistema persistente.

   Este conjunto de primitivas operacionales ha de reducirse al mínimo; sin embargo, en ocasiones puede ser necesaria su ampliación. Un ejemplo puede ser la inclusión de números racionales en el álgebra operacional de la máquina. Esta ampliación debe llevarse a cabo sin necesidad de modificar la implementación de la máquina, de forma que el código existente para nuestra plataforma no pierda nunca su capacidad de ser ejecutado –la máquina virtual no deberá sufrir modificación alguna.

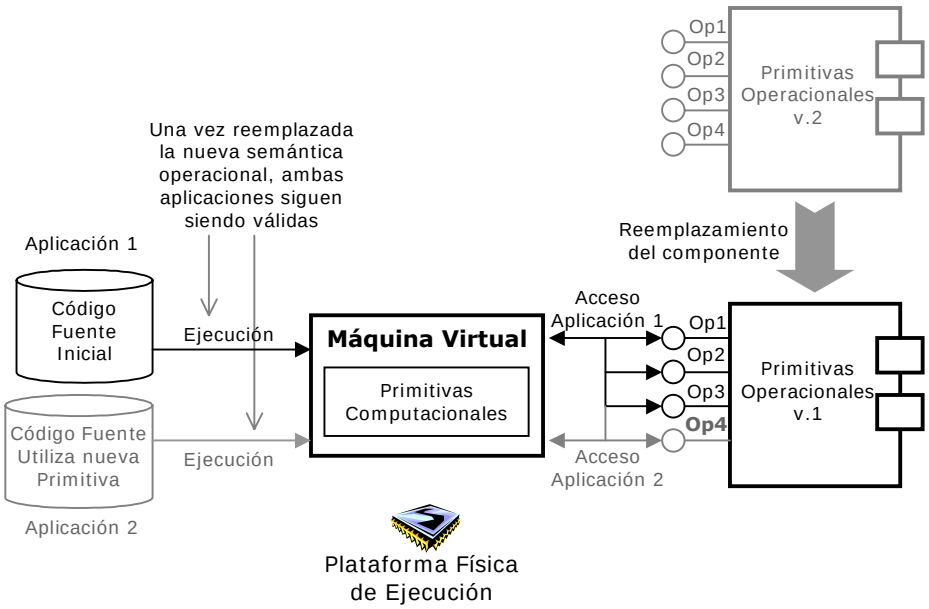


Figura 10.1: Ampliación de la semántica operacional de la máquina virtual sin perder la portabilidad de sus aplicaciones.

[37] Como ejemplo real podemos tomar la implementación de la máquina virtual de Java (JVM, *Java Virtual Machine*) [Lindholm96] como parte de un navegador Web, dedicada a ejecutar código Java descargado dinámicamente de la red.

La ampliación del sistema debe llevarse a cabo sin modificar la interfaz del módulo de primitivas operaciones existente con anterioridad. De este modo, la plataforma es adaptable a cualquier entorno, no existirán distintas versiones del intérprete, y las aplicaciones no quedarán desfasadas.

10.1.2 Extensibilidad

El hecho de diseñar la máquina abstracta con el menor número de primitivas posible ayuda a que ésta sea fácilmente implantada en diversas plataformas. Sin embargo, el bajo nivel de abstracción de su lenguaje hace complejo el desarrollo de aplicaciones sobre ésta. Un mecanismo de extensibilidad de la plataforma computacional será necesario para obtener un mayor nivel de abstracción.

En el estudio de los sistemas que hacen uso de la introspección (§ 7.1) y de la reflectividad estructural (§ 7.2), vimos cómo ambos conceptos pueden ser utilizados para implementar sistemas extensibles. En concreto, en el caso de los sistemas Smalltalk, Self y ObjVlisp, esta extensibilidad se ha empleado en la creación de entornos computacionales diseñados sobre máquinas abstractas.

Mediante introspección, podremos conocer dinámicamente la estructura dinámica de un objeto (sus métodos, atributos y el número de veces que es referenciado por otros objetos, por ejemplo). Haciendo uso de esta información, se podrá desarrollar un sistema de persistencia, un tratamiento de excepciones, o incluso un recolector de basura, gracias al conocimiento estructural de cada objeto.

En el caso de haber implementado un sistema de persistencia o un sistema de distribución con movilidad de objetos, la reflectividad estructural puede ser utilizada para crear dinámicamente un objeto y añadirle sus atributos y métodos oportunos. El desarrollo de estas funcionalidades eleva el nivel de abstracción computacional existente inicialmente significando la extensión del mismo.

El hecho de implementar un entorno de programación haciendo uso del propio lenguaje de la máquina y de su facultad extensiva, implica que la totalidad del código de dicho entorno será independiente de la plataforma, y que no será necesario modificar la implementación de la máquina virtual dando lugar a distintas versiones – se perdería así la posibilidad de ejecutar código de versiones anteriores.

10.1.3 Definición del Modelo Computacional

La selección correcta del modelo computacional a seguir por la máquina abstracta es una tarea compleja. Por un lado, ha de ser lo suficientemente sencilla como para satisfacer el criterio de reducir al mínimo su número de primitivas computacionales (§ 10.1.1); por el otro, deberá ofrecer un nivel de abstracción que permita la interacción entre aplicaciones de un modo sencillo, independientemente del lenguaje de programación seleccionado por el programador.

Los modelos computacionales orientados a objetos están siendo ampliamente utilizados en la actualidad. Sin embargo, la implementación del conjunto global de sus características [Booch94] supondría que la máquina virtual poseyese un tamaño demasiado elevado para implantarlo en plataformas con poca capacidad de procesamiento. Tomando el modelo de objetos de Smalltalk y reduciéndolo computacionalmente, se creó un modelo de objetos más limitado basado en el concepto de prototipo (capítulo 8). Este modelo de objetos era más sencillo de implementar y, lo más importante, no perdía expresividad frente al mo-

delo basado en clases: todo lo representable en un lenguaje orientado a objetos basado en clases, se puede expresar mediante el uso de prototipos [Ungar91].

Como explicábamos en el capítulo 8, el modelo orientado a objetos basado en prototipos elimina el concepto de clase. Además de reducirse la semántica de sus operaciones sin pérdida alguna de expresividad, la eliminación del concepto de clase es útil y coherente en entornos estructuralmente reflectivos. La utilización de reflectividad estructural con el modelo basado en clases produce la aparición de un problema denominado evolución de esquemas (*scheme evolution*) [Roddick95].

¿Qué sucede si queremos modificar la estructura de un único objeto de una clase sin alterar el resto de sus instancias? ¿Cómo se amoldan los objetos al eliminar, modificar o aumentar alguno de los atributos de su clase? Existen soluciones de compromiso como la creación de clases sombra (*shadow classes*), implementada por la plataforma reflectiva MetaXa (§ 7.4), de elevada complejidad de implementación y resultados poco satisfactorios. En la utilización del modelo de prototipos, estas modificaciones no conllevan a incoherencias en el lenguaje de programación –véase el capítulo 8.

Conocidos lenguajes de programación, como Smalltalk, Java y Python, utilizan este modelo computacional en tiempo de ejecución. Aunque en tiempo de diseño exista el concepto de clase, éste es traducido en la ejecución de la aplicación al de objeto, convirtiéndose así toda clase en un objeto.

La elección del modelo computacional orientado a objetos basado en prototipos para la máquina abstracta, y por tanto para el conjunto del sistema, se justifica por varias razones:

1. El modelo computacional es más sencillo que el basado en clases.
2. Existe una traducción directa y evidente del modelo de clases al de prototipos, utilizado por la mayoría de los lenguajes actuales –véase el capítulo 8.
3. La utilización del modelo no supone pérdida de expresividad alguna.
4. Se amolda mejor a un esquema computacional estructuralmente reflectivo.

10.1.4 Adaptabilidad

Los sistemas operativos basados en micronúcleo (§ 5.6) obtienen una adaptabilidad de sus servicios, que ofrecen mediante la definición de una interfaz para cada tipo de servicio. Cualquier rutina que implemente las operaciones definidas por la interfaz, puede formar parte del sistema operativo y sustituir a otro ya existente, obteniéndose así un grado de adaptabilidad respecto a sus servicios.

La idea de separar el núcleo computacional de los mayores niveles de abstracción es adoptada en nuestro sistema, separando la implementación de la máquina virtual de la codificación del entorno de programación. Sin embargo, la especificación previa de las interfaces de los distintos servicios, limita el número y tipo de servicios que pueden implantarse en nuestra plataforma –entorno de programación.

La introspección y reflectividad estructural de nuestra máquina, serán utilizadas también como mecanismos de adaptabilidad. En un entorno de computación heterogéneo, una aplicación puede hacer uso de la introspección para conocer los servicios existentes en la plataforma existente. En función de los servicios existentes, podrá llevar a cabo una funcionalidad u otra, adaptándose así al contexto de ejecución presente.

Haciendo uso de la reflectividad estructural ofrecida, la estructura de un objeto que ofrezca un determinado tipo de servicio podrá modificarse para derogar dicha funcionalidad en otros, sin necesidad de establecer previamente la interfaz de funcionamiento –como sucede en los sistemas basados en micronúcleo.

Siguiendo con los criterios reflectivos de diseño del proyecto Merlin [Assumpcao95] desarrollado sobre el lenguaje Self, los objetos primitivos se extienden con la implementación de nuevos objetos en el mismo lenguaje. La totalidad del sistema será codificado utilizando estos nuevos objetos. Si queremos modificar el significado de una de sus primitivas, modificamos la estructura del objeto –alguno de sus métodos–, o bien lo sustituimos por otro nuevo. De esta forma, todo el software es adaptable dinámicamente al no utilizar de forma directa las funcionalidades de los objetos primitivos.

Como ejemplo de lo comentado en el párrafo anterior, podemos codificar un objeto encargado de crear objetos haciendo uso de la primitiva existente para esto. Todo el entorno de programación se codificará haciendo uso de este nuevo objeto. Si se desea implementar un sistema de persistencia, podrá modificarse la funcionalidad de éste para hacer que la creación de dichos objetos implique su escritura en disco.

Otro modo de utilizar la reflectividad estructural para el desarrollo de aplicaciones adaptables es mediante la codificación de éstas en función de la estructura de sus objetos. La información de alta de elementos en una base de datos del menú de una aplicación, podrá adaptarse dinámicamente haciendo uso de esta capacidad.

10.1.5 Interacción Directa entre Aplicaciones

La adaptabilidad del sistema ha ofrecerse tanto para una misma aplicación, como entre distintas aplicaciones, de forma que la ejecución de un programa pueda adaptar el funcionamiento de otro. Del mismo modo, la interacción entre aplicaciones es requerida para que la intercomunicación entre ellas, indiferentemente del lenguaje de programación utilizado, sea factible –ya que el motor computacional base de todo el sistema es la máquina abstracta.

El acceso desde una aplicación a los objetos de otra ha de producirse de igual modo que el acceso a sus propios objetos, sin necesidad de utilizar una capa intermedia de software. En el caso de la máquina virtual de Java [Sun95], la ejecución de cada aplicación implica la creación de una nueva instancia de la máquina que interprete su código. Para intercomunicar estos dos procesos, es necesaria una capa software adicional como por ejemplo RMI (*Remote Method Invocation*) [Sun97b].

Para minimizar la complejidad en la interacción de aplicaciones, el diseño de la máquina abstracta ha de llevarse a cabo teniendo en cuenta que ésta supondrá un único proceso en el sistema operativo en el que esté hospedada. Las aplicaciones en una misma plataforma se ejecutarán en el mismo espacio de direcciones, interaccionando a través de un espacio de nombres que tenga en consideración, para cada objeto, la aplicación en la que fue creado.

10.1.6 Manipulación Dinámica del Comportamiento

La introspección y reflectividad estructural de la máquina nos permite conocer y modificar dinámicamente la estructura de un objeto. En el modelo computacional basado en prototipos, la estructura de un objeto comprende tanto sus propiedades (atributos) como sus métodos. Por lo tanto, para ofrecer conocimiento dinámico real de la estructura de un objeto, es necesario acceder y manipular su comportamiento.

La representación de la computación como datos manipulables recibe el nombre de cosificación (§ 6.1). El hecho de tratar dinámicamente instrucciones como datos, conlleva la necesidad de que la plataforma computacional implemente un mecanismo de evaluación o descosificación de dicha información. La posibilidad de manipular dinámicamente código como si de datos se tratase supone:

1. La viabilidad de modificación dinámica del comportamiento de los objetos.
2. El desarrollo de aplicaciones que se adapten a comportamientos imprevisibles en tiempo de diseño, surgidos cuando ésta ya esté desarrollada y en ejecución.
3. La factibilidad de depurar, modificar y ampliar una aplicación en ejecución, puesto que se puede especificar dinámicamente la creación y modificación de nuevos comportamientos.

Vemos cómo esta característica impuesta a la máquina abstracta está enfocada hacia la obtención de un sistema computacional más flexible.

10.2 Técnicas de Diseño Seleccionadas

Como hemos justificado en este capítulo, el diseño de la máquina abstracta, raíz computacional del conjunto del sistema, deberá diseñarse siguiendo los siguientes criterios:

1. Ha de reducirse al mínimo el número de primitivas computacionales y operacionales.
2. La implementación de las primitivas operacionales ha de separarse de las computacionales, siguiendo un mecanismo que permita al usuario extenderlas, sin modificar la implementación de la máquina virtual existente.
3. El sistema computacional deberá ofrecer funcionalidades introspectivas.
4. La reflectividad estructural permitirá al usuario modificar los atributos y métodos existentes para cada objeto.
5. El sistema computacional seguirá un modelo orientado a objetos, basado en prototipos.
6. La ejecución de cada una de las aplicaciones en una misma plataforma física se llevará a cabo en el mismo espacio de direcciones, y la interacción entre objetos de distintas aplicaciones se producirá sin ninguna capa software adicional.
7. Deberá permitirse el tratamiento del comportamiento de cada objeto como si de datos se tratase, implementándose un mecanismo de evaluación o descosificación de datos dinámico.

CAPÍTULO 11:
ARQUITECTURA DEL SISTEMA REFLECTIVO NO RESTRICTIVO

A raíz de la estructuración de nuestro sistema en capas, presentada en el capítulo 9, estudiaremos en este título la arquitectura de la tercera y última 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 9, 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 14).

Puesto que el desarrollo de esta última capa del sistema se llevará a cabo sobre el sistema computacional descrito por la máquina abstracta, la arquitectura propuesta demandará unos requisitos computacionales a la plataforma base. Finalmente, enunciaremos estos requisitos que deberán satisfacerse en el diseño de la primera capa.

11.1 Análisis de Técnicas de Obtención de Sistemas Flexibles

En el capítulo 5 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.

11.1.1 Sistemas Flexibles No Reflectivos

Exceptuando los sistemas operativos basados en micronúcleo, el conjunto de aproximaciones descritas en el capítulo 5 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 reutilizadas.

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 crea-

ción de software susceptible de ser adaptado por diversas incumbencias, careciendo de la posibilidad de crear aplicaciones flexibles en tiempo de ejecución.

Si tomamos, por ejemplo, la técnica de desarrollo de aplicaciones orientadas a aspectos, observamos que es necesario desarrollar la funcionalidad de la aplicación por un lado, y por otro definir los aspectos reutilizables a cualquier aplicación –por ejemplo, los aspectos "persistencia" y "optimización en velocidad". Una vez creadas las distintas partes de la aplicación, se crea el código final de ésta mediante el tejedor de aspectos (*aspect weaver*) –traductor de aplicaciones en función de los aspectos definidos.

La principal carencia del funcionamiento descrito 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.

Como comentaremos en el capítulo 15, el sistema reflectivo desarrollado puede ser utilizado como una plataforma en la que sea factible el conocimiento, selección y modificación dinámica de los distintos aspectos de una aplicación, sin necesidad de finalizar su ejecución.

11.1.2 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 (§ 11.1.1): 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.

11.1.2.1 Reflectividad Dinámica

Siguiendo con el estudio desarrollado en el capítulo 7 y centrándonos en los objetivos principales de esta tesis (§ 1.2), las técnicas estudiadas más cercanas a ofrecer la flexibilidad buscada son las reflectivas en tiempo de ejecución. Dentro de esta clasificación tene-

mos 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 § 9.5.3, 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 11.1, posee un conjunto de limitaciones:

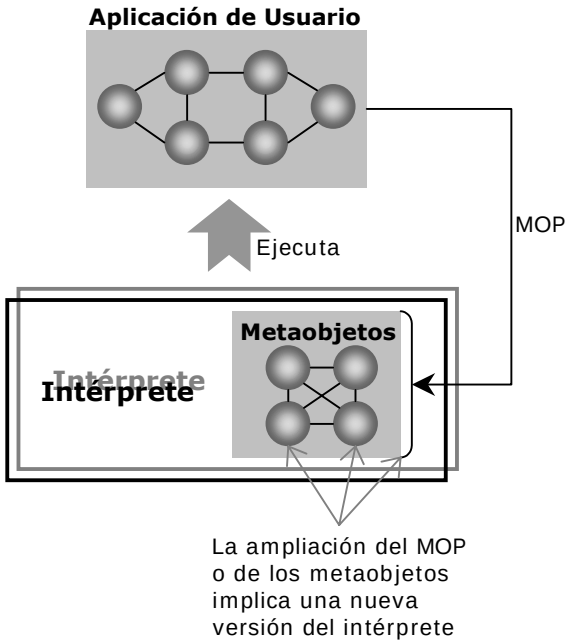


Figura 11.1: Estructura general de un MOP.

1. 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.
2. 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.
3. 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.
4. 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.

5. 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 (§ 7.4.1).
6. La estructura establecida en los sistemas que utilizan MOPs es siempre dependiente de un único lenguaje de programación.

11.2 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], hemos justificado en § 9.5.3 la necesidad de una altura de dos niveles –interpretación de un intérprete– y una anchura absoluta –acceso al metasistema no restrictivo. 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 última capa del sistema (§ 9.5), podemos abreviar enunciando las principales características que ha de aportar frente a un sistema basado en un MOP:

1. No debe ser necesario estipular lo que va a adaptarse previamente a la ejecución de la aplicación.
2. Debe existir un mecanismo de expresividad en el que cualquier característica del sistema pueda adaptarse dinámicamente –horizontalidad total.
3. El sistema debe ser totalmente independiente del lenguaje.
4. Los grados de flexibilidad a conseguir son: introspección, estructural, semántica y lenguaje.

El esquema a seguir se muestra en la siguiente figura[38]:

[38] En este apartado sólo estudiaremos la funcionalidad de cada una de las partes del esquema. En puntos sucesivos, profundizaremos acerca de cómo desarrollar éstas.

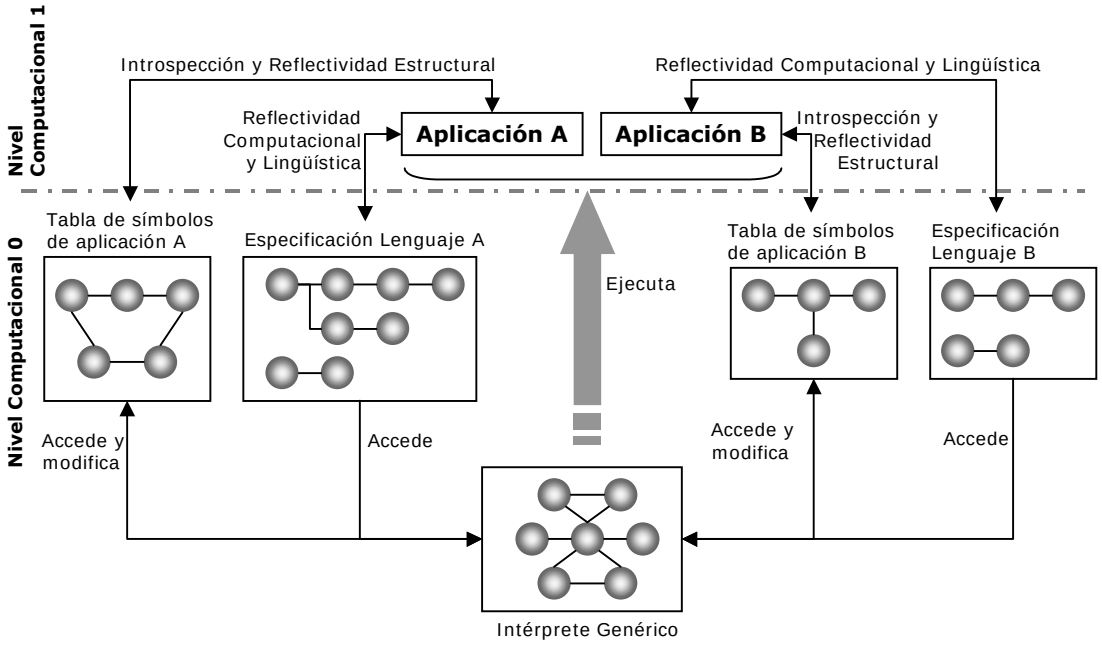


Figura 11.2: Esquema general del sistema computacional no restrictivo.

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[39] 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:

1. Si analiza, sin modificar, su tabla de símbolos supondrá introspección.
2. En el caso de modificar los elementos de su tabla de símbolos, la flexibilidad obtenida es reflectividad estructural.
3. El acceso y modificación de la semántica propia de la especificación de su lenguaje de programación significará reflectividad computacional.
4. La modificación de las características léxicas o sintácticas de su lenguaje producirán una adaptación lingüística.

[39] 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.

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 11.2. 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?[40]

11.2.1 Salto Computacional

La raíz computacional de nuestro sistema está soportada por una implementación, hardware o software, de nuestra máquina abstracta. 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 11.3, el dominio computacional de la máquina abstracta 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.

[40] Es importante darse cuenta cómo una primera aproximación para solucionar el problema planteado es la utilización de un MOP. Sin embargo, como hemos comentado en § 11.1.2.1, los MOPs establecen restricciones de acceso en el salto computacional.

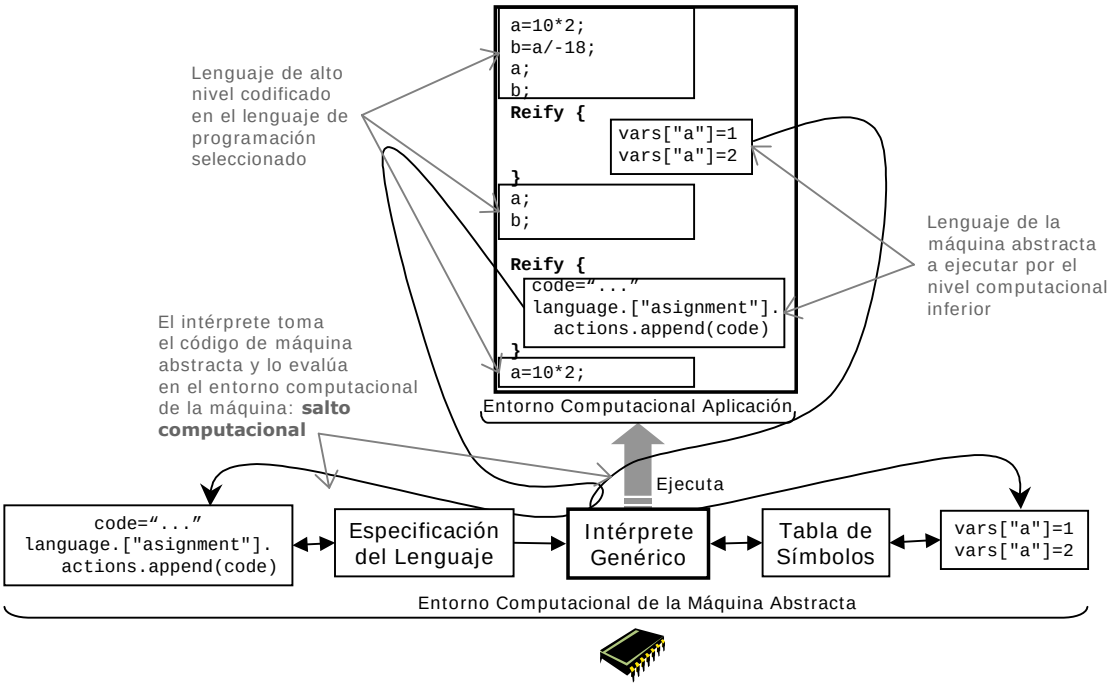


Figura 11.3: Salto computacional producido en la torre de intérpretes.

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 de la máquina abstracta. Así, una aplicación poseerá la expresividad de su lenguaje más la propia de la máquina abstracta.

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:

1. Tomará la consecución de instrucciones codificadas en el lenguaje de la máquina abstracta, como una cadena de caracteres.
2. Evaluará o descosificará los datos obtenidos, para que sean computados como instrucciones por la máquina abstracta.

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 a la máquina abstracta.

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:

1. Si analiza, mediante la introspección ofrecida por la máquina abstracta, su propia tabla de símbolos, estará obteniendo información acerca de su propia ejecución: introspección de su propio dominio computacional.
2. 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.
3. 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.
4. 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 como otros dos requisitos necesarios en la máquina abstracta 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.

11.2.2 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 de la máquina abstracta, sea posible. Una descripción más detallada puede obtenerse del diseño del prototipo realizado en el capítulo 14, “Diseño de un Prototipo de Computación Reflectiva Sin Restricciones”.

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 reglas semánticas expresadas mediante código de la plataforma virtual. Su expresividad es la propia de una definición dirigida por sintaxis [Aho90], utilizando el lenguaje de la máquina abstracta 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; la especificación de la gramática de una implementación de este metalenguaje se describe en el 283apéndice B. 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:

`S ::= B S`	`r1: Regla en Código Máquina Abstracta`

	r2: Regla en Código Máquina Abstracta
| C D	r3: Regla en Código Máquina Abstracta
B ::= “B”	
C ::= “C”	
D ::= “D”	

Los símbolos terminales se han mostrado entre comillas dobles. Las reglas semánticas `r1`, `r2` y `r3` se codificarán mediante el lenguaje de la máquina abstracta. 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 11.4. 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.

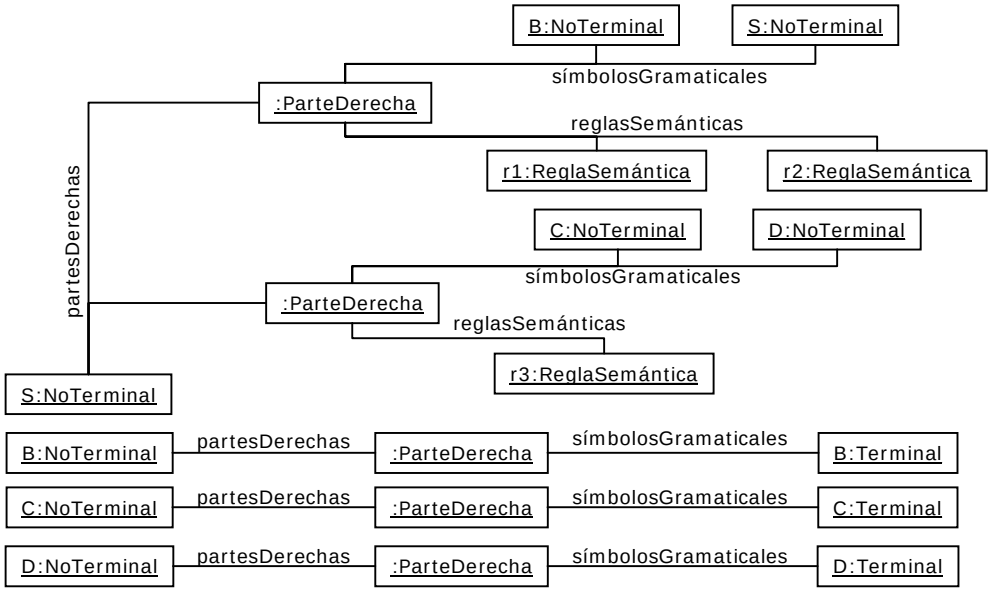


Figura 11.4: Especificación de un lenguaje de programación mediante una estructura de objetos.

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 11.5.

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].

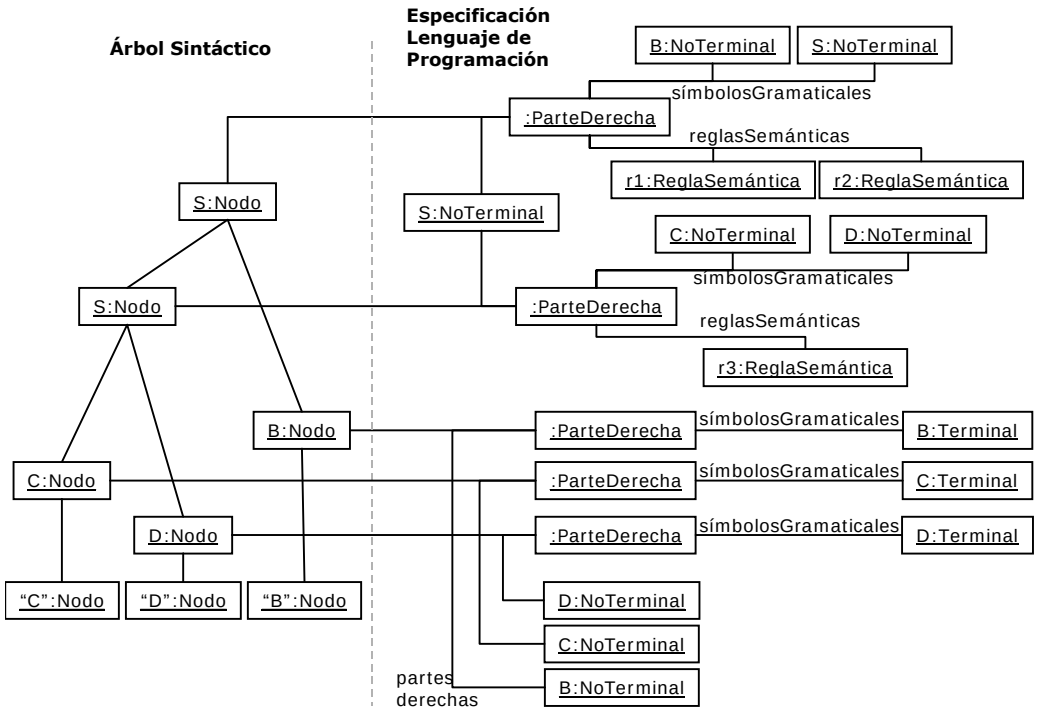


Figura 11.5: Creación del árbol sintáctico asociado a la especificación de un lenguaje.

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[41].

Por cada aplicación en nuestro sistema existirá un objeto representante de ésta. 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.

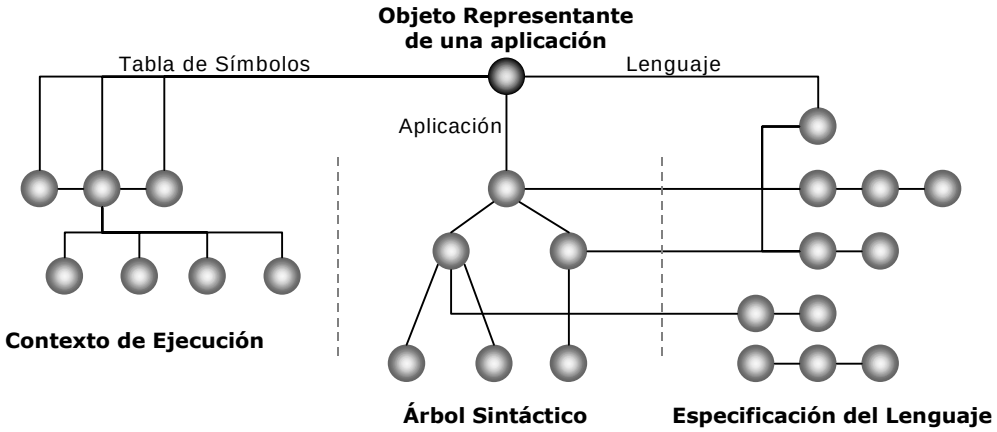


Figura 11.6: Principal información existente para cada aplicación en ejecución.

Accediendo a la lista de objetos representante de su tabla de símbolos –mediante la introspección y reflectividad estructural ofrecida por la máquina abstracta–, se le brinda al

[41] Para obtener más información acerca de cómo describir la evaluación del árbol, consúltese el diseño (capítulo 14) o manual de usuario (apéndice B) del prototipo implementado.

programador de aplicaciones introspección y reflectividad estructural del lenguaje de programación que haya seleccionado.

11.3 Beneficios del Sistema Presentado

Enlazando con el análisis realizado en § 11.1 de técnicas utilizadas para obtener sistemas computacionales flexibles, éste es el conjunto de beneficios aportados frente a los sistemas basados en MOPs:

1. 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.
2. No existe una expresividad limitada. El modo en el que podemos modificar una aplicación se expresa mediante un lenguaje de computación: el lenguaje de la máquina abstracta. 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.
3. 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 de la máquina abstracta.
4. Se ofrece cuatro niveles de adaptabilidad: introspección, estructura dinámica, comportamiento y lenguaje.
5. 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.
6. 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.

11.4 Requisitos Impuestos a la Máquina Abstracta

Analizando los requisitos necesarios impuestos a la máquina abstracta para poder desarrollarse el sistema reflectivo propuesto, podemos enumerarlos del siguiente modo:

1. 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.
2. Evaluación o descosificación de datos. El intérprete genérico toma el código de la máquina abstracta 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.
3. Reflectividad estructural. El código de la máquina abstracta 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.
4. Interacción directa entre aplicaciones. Puesto que el código a descosificar por la máquina abstracta 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.

Como tesis propuesta, afirmaremos que un entorno computacional que ofrezca estas características será válido para desarrollar esta capa del sistema reflectivo[42]. La demostración de esta proposición ha sido llevada a cabo mediante la implementación de un prototipo cuyo diseño es descrito en el capítulo 14, y su utilización en el apéndice B.

[42] Tan solo nos referimos al desarrollo de esta capa del sistema. Para obtener la totalidad de los beneficios, deberemos desarrollar ésta sobre una plataforma virtual poseedora de las características descritas en el capítulo 9 y en el capítulo 10.

CAPÍTULO 12:
DISEÑO DE LA MÁQUINA ABSTRACTA

Basándonos en los criterios de diseño especificados en la arquitectura de la máquina abstracta, capítulo 10, diseñaremos ésta mediante la especificación léxica y sintáctica de cada una de sus instrucciones, definiendo su semántica mediante los cambios producidos en su estado computacional.

Describiremos la plataforma virtual sin profundizar en la implementación de la misma. El diseño de un prototipo de su implementación –máquina virtual– será mostrado como un apéndice en el apéndice A.

Una vez descritas las funcionalidades básicas de la máquina, haciendo uso de su capacidad de extensibilidad, ampliaremos levemente su abstracción mediante la codificación de funcionalidades adicionales, como operaciones lógicas, aritméticas, de comparación, iteración, y creación de abstracciones y entidades.

12.1 Especificación de la Máquina Abstracta

La principal característica a definir dentro del modelo computacional ofrecido por la máquina abstracta, es su noción de objeto. Identificábamos como criterio de diseño a seguir en su arquitectura el modelo computacional de objetos basado en prototipos. De esta forma, el objeto supone la abstracción base de su modelo computacional.

12.1.1 Noción de Objeto

En tiempo de ejecución, la máquina alberga un conjunto de objetos. El modo de acceder y utilizar éstos, es mediante referencias: si queremos utilizar los servicios ofrecidos por un objeto o conocer su estado, debemos utilizar una referencia a éste.

Un objeto está constituido por un conjunto de referencias a otros objetos. Estas referencias entre objetos denotan distintas semánticas de relaciones como general-específico, asociación o todo-parte.

Dada una referencia a un objeto, podemos conocer el conjunto de referencias que éste posee, observando su estado, estructura y comportamiento –introspección. Del mismo modo, el conjunto de referencias que posee puede modificarse dinámicamente permitiendo la inserción, borrado y actualización de nuevas referencias en tiempo de ejecución –reflectividad estructural.

Un objeto puede denotar en sí información (datos) o comportamiento. La representación de ambos se produce del mismo modo, mas la representación del comportamiento puede evaluarse o descosificarse[43], es decir, traducir dinámicamente su información en una computación para la máquina abstracta. El comportamiento de un objeto viene denotado por sus objetos asociados que representan computación; su estado es descrito por aquellos asociados que simbolizan únicamente información.

12.1.2 Entorno de Programación

La máquina virtual interpretará el lenguaje de la máquina abstracta que definiremos en este documento. El estado computacional inicial de la máquina abstracta viene dado por un conjunto de objetos primitivos, descritos en § 12.1.3, que ofrecen su funcionalidad computacional básica. Uno de los criterios definidos en la arquitectura de la máquina, capítulo 10, requiere reducir al mínimo el número de objetos primitivos, para otorgar el mayor grado de heterogeneidad posible a la plataforma base, facilitando así la implantación de ésta en sistemas dotados de una capacidad computacional restringida.

Al igual que sucede en la plataforma de Smalltalk, al inicializar la máquina abstracta ésta interpreta una imagen del entorno de computación: codificación, en el propio lenguaje de la máquina, de un entorno de programación que eleve el nivel de abstracción inicial de la plataforma. Partiendo de la posibilidad de acceso a todos los objetos existentes en tiempo de ejecución, se irá creando nuevas abstracciones para conseguir un mayor nivel de abstracción –extensibilidad.

Introspección y reflectividad estructural son las técnicas ofrecidas para obtener el mecanismo de extensibilidad buscado –capítulo 10. El hecho de codificar la extensión computacional de la máquina en su propio lenguaje supone los siguientes beneficios:

1. El entorno de programación al completo es portable a cualquier plataforma.
2. No se crean distintas versiones de la máquina virtual, perdiendo la portabilidad de aplicaciones de versiones anteriores.
3. El tamaño de la implementación de la plataforma virtual no es excesivo y puede implantarse en un número elevado de entornos computacionales.

Para cada una de las funcionalidades desarrolladas en el entorno de programación (persistencia, distribución, planificación genérica de hilos y recolección de basura), se deberá proporcionar un grado de adaptabilidad que permita modificar y adaptar dinámicamente los mecanismos, técnicas y protocolos utilizados, en función de los requerimientos surgidos.

12.1.3 Objetos Primitivos

Inicialmente la máquina abstracta posee el siguiente conjunto de objetos primitivos, así como las referencias propias para acceder a ellos:

- Objeto `nil`. Objeto ancestro que ofrece las funcionalidades básicas de cualquier objeto de nuestra plataforma.
- Objetos cadenas de caracteres. Representan la información indivisible de nuestro sistema. Esta información puede representar simplemente datos, o computación a ser evaluada.

[43] Proceso inverso de cosificar: convertir en datos una computación.

- Objeto `Extern`. Identifica el mecanismo de adición de primitivas operacionales a nuestra plataforma, cuyo requerimiento fue impuesto en el capítulo 10.

- Objeto `System`. Representa la colección de todos los objetos existentes en la plataforma real utilizada. Implica un mecanismo introspectivo de conocimiento de los objetos existentes en la ejecución de cada máquina.

Referencias

El acceso a un objeto se ha de realizar siempre mediante una referencia. Un objeto como tal no tiene un nombre propio, pero sí una referencia a éste. Por abreviar, a lo largo de este documento expresaremos "el objeto `R`" en lugar de "el objeto referenciado por `R`", aunque realmente `R` sea una referencia al objeto.

Una referencia ofrece un mecanismo de acceso a los servicios propios de un objeto: su estado (datos) y comportamiento (computación). Mediante la utilización de una referencia a un objeto, podremos conocer y modificar su estado, así como solicitar el desencadenamiento de cualquiera de los comportamientos que éste desarrolle.

12.1.3.1 Objeto **nil**

Inicialmente existe el objeto `nil`, ancestro de todo objeto, al que se puede acceder mediante la referencia `nil`. Este objeto, al igual que cualquier otro, posee un conjunto de referencias a otros objetos. Todo objeto tendrá al menos dos referencias miembro:

- `id`: Referencia a un objeto cadena de caracteres, que identifica de forma única el objeto. Esta unicidad ha de ser universal, es decir, no puede existir colisión de identificadores con otro objeto, aunque se encuentre en otra plataforma

- `super`: Referencia al objeto padre o base del objeto actual.

El objeto al que se puede acceder a través de `nil`, posee como referencia `super` una referencia a él mismo, y como referencia `id` una referencia al objeto `<nil>`.

Pero, ¿qué es el objeto `<nil>`? Un objeto cadena de caracteres. Cada vez que se necesite un objeto cadena de caracteres, se podrá utilizar una referencia a él identificando la cadena de caracteres entre los delimitadores léxicos `<` y `>`. De esta forma, este tipo de objetos se van creando implícitamente por el programador, mediante la utilización de referencias nombradas entre `<` y `>`.

12.1.3.2 Objetos Cadenas de Caracteres

Para la máquina abstracta, un objeto cadena de caracteres significará la forma básica de representar cualquier dato. Además, la plataforma permite evaluar o descosificar[44] estos objetos, representando así su computación o comportamiento.

Hemos visto cómo todo objeto posee una referencia a otro objeto cadena de caracteres que identifica al primero de forma única. Para el objeto `nil`, su referencia `id` lo asocia al objeto `<nil>`. Este objeto, a su vez, posee como referencia `id` una referencia a sí mismo, y como referencia `super` una referencia al objeto de rasgo (*trait*) `String`.

En los lenguajes orientados a objetos basados en prototipos, el concepto de clase es sustituido por el concepto de objeto *trait* –véase el capítulo 8. Estos objetos no son tratados computacionalmente de un modo especial; son un objeto más. El programador es el encargado de utilizarlos para agrupar objetos con comportamiento y estructura similares.

[44] Al igual que sucede con la invocación de la función `eval` del LISP o `exec` de Python.

De este modo, el objeto *trait* `String` define el comportamiento de todos los objetos de tipo cadena de caracteres. Para obtener una mayor claridad de código, la plataforma y el entorno de computación han sido desarrollados siguiendo la notación de identificar los objetos de rasgo con su primera letra en mayúscula.

En la siguiente figura mostramos las asociaciones existentes entre los objetos primitivos mencionados.

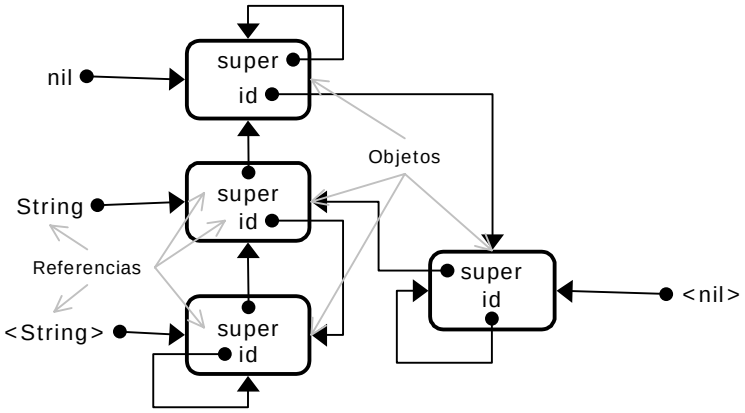


Figura 12.1: Objetos primitivos "nil" y "String".

Vemos cómo, siguiendo los criterios de diseño especificados, el identificador del objeto `String` es el objeto `<String>`, así como su objeto padre es el objeto `nil`.

Las cadenas de caracteres de este lenguaje han de poder anidarse. Existirán cadenas de caracteres dentro de otras cadenas de caracteres. El autómata que las reconoce y su gramática equivalente no puede ser de tipo 3 o regular, como las clásicas, comprendidas entre una pareja del mismo delimitador [Cueva91]. Por lo tanto, las cadenas de caracteres son declaradas como sentencias de un lenguaje de tipo 2, libre de contexto, y necesitan un elemento léxico de apertura y cierre distinto: < y > respectivamente.

Los objetos cadenas de caracteres son el elemento primitivo de representación de datos y computación para nuestra plataforma. Haciendo uso de la capacidad de reflectividad estructural, podremos acceder en tiempo de ejecución a la estructura y estado de cualquier objeto, así como a la implementación de todos los métodos no primitivos de la máquina y del entorno de computación[45].

Las cadenas de caracteres poseerán las siguientes de características:

- Podrán anidarse. Dentro de una cadena representativa de código, podremos necesitar representar a su vez otra cadena de caracteres. De este modo, deberá permitirse la introducción de cualquier conjunto de cadenas de caracteres, de forma recursiva.

- Podrán contener caracteres especiales sin necesidad de utilizar códigos de escape. Podremos utilizar cadenas de caracteres de más de una línea así como introducir tabuladores dentro de ésta.

- Representación de delimitadores. ¿Cómo se introducen los caracteres < y > dentro de la propia cadena, sin que indiquen una subcadena? Con el carácter de escape \, teniendo así dos caracteres distintos \> y \< que no indicarán anida-

[45] Esta capacidad hace que el sistema sea dinámicamente introspectivo.

miento de cadenas de caracteres, y cuya representación a la hora de visualizarlos será > y < respectivamente.

- Utilización de caracteres de escape. Se podrán utilizar, si se desea, los caracteres de escape `\n`, `\\` y `\t`, en lugar de su propia representación.

- Caracteres desoídos. Puesto que el sistema será autodocumentado mediante introspección, será necesario representar dicha documentación de forma elegante. Sin embargo, en la codificación de aplicaciones, puede surgir la necesidad de indentar (sangrar) el código. No obstante, no se desea que aparezcan los tabuladores de indentación en la documentación. Debemos identificar un modo de ignorar caracteres para su posterior documentación.

  Se establecen bloques de caracteres que ignorará el analizador léxico y no aparecerán al acceder posteriormente a ellos. Estos caracteres serán los comprendidos entre `\*`. Por ejemplo, podremos evaluar lo siguiente:

```
nil:<set>(<==>,<\*
            \* o <- params:<0>;
\*          \* sID <- o:<id>;
\*          \* return <- sID:<==>(<nil>);\*
            \* >);
```

Posteriormente, si accedemos al objeto asociado a `nil` mediante la referencia == tendremos:

```
< o <- params:<0>;
 sID <- o:<id>;
 return <- sID:<==>(<nil>); >
```

Para mostrar el código con mayor legibilidad, la indentación de cadenas de caracteres con los caracteres de escape se obviará a partir de aquí, a lo largo de este capítulo.

12.1.3.3 Objeto **Extern**

Como se identificó en el capítulo 10, separaremos las primitivas computacionales de las operacionales, permitiendo ampliar las segundas mediante un mecanismo de interconexión estándar con el sistema operativo hospedado.

La máquina abstracta posee las siguientes primitivas:

- Primitivas Computacionales. Propias de la semántica del funcionamiento de la máquina. Podemos mencionar el mecanismo de acceso a los miembros, la evaluación o descosificación de objetos, y el mecanismo de delegación o herencia dinámica. Estas primitivas son implementadas en la máquina virtual y no puede modificarse su semántica.

- Primitivas Operacionales. Se representan mediante objetos evaluables que especifican la semántica de operaciones, no representables mediante las existentes inicialmente en la plataforma. Este tipo de semántica podrá ampliarse en función de los requisitos de la plataforma.

  Ejemplos de este tipo de primitivas son: operaciones aritméticas, tratamiento de cadenas de caracteres, acceso a disco, primitivas de distribución o interfaces gráficas.

Para desarrollar un mecanismo de ampliación de primitivas operacionales, se necesita un protocolo de comunicación con las implementaciones externas de los objetos primi-

tivos operacionales, utilizando un mecanismo de interconexión estándar que sea coherente con el diseño de la máquina. En función del sistema operativo existente en la implantación de la plataforma virtual, se utilizará un mecanismo de interconexión que permita ampliar la semántica operacional, de un modo estándar –ejemplos de esto pueden ser sistemas de componentes o librerías de enlace dinámico.

Mediante la utilización de este sistema de ampliación de primitivas, la máquina no tendrá nunca que reemplazarse, no se perderá portabilidad de su código, y la plataforma computacional poseerá la posibilidad de ampliar sus operaciones concretas de una plataforma[46].

La referencia `Extern` apunta a un objeto primitivo que posee un miembro `id` con valor `<Extern>`, un miembro `super` haciendo referencia a `nil`, y un miembro `invoke`. Este último objeto es el encargado de representar todas las primitivas operacionales externas; mediante el paso parámetros, indicaremos qué primitiva externa debe evaluar.

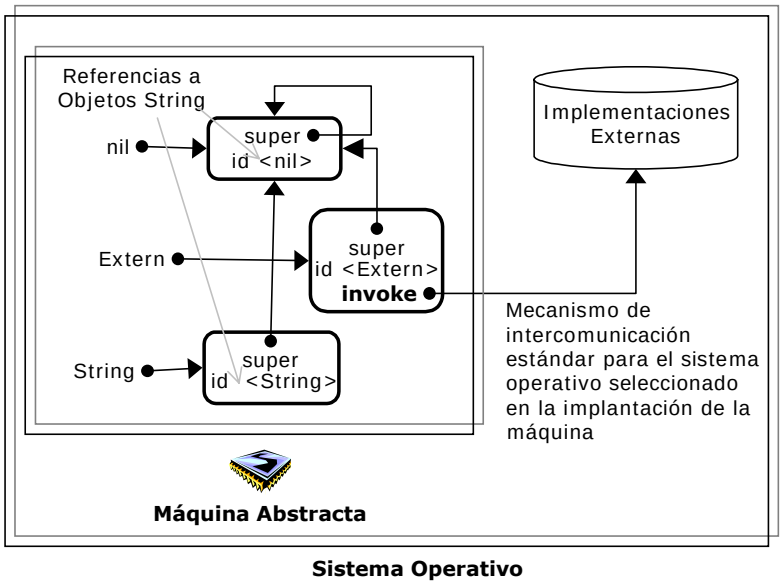


Figura 12.2: Acceso a las primitivas operacionales externas mediante el método "invoke" del objeto "Extern".

La sintaxis de invocaciones al método `invoke` tendrá como parámetros la referencia al objeto donde se ubica el miembro, la referencia al nombre del miembro, la referencia al parámetro implícito y los parámetros de la invocación –veremos en § 12.1.4.4 la sintaxis de invocación de métodos.

12.1.3.4 Objeto **System**

La reflectividad estructural de la plataforma nos permite acceder a los objetos y a la estructura de éstos, de forma dinámica. Para conocer y acceder a su estructura, veremos cómo los miembros de `nil` ofrecen las primitivas de contenedor. El objeto *trait* `System` será el encargado de ofrecer el conocimiento dinámico de los objetos existentes en la plataforma, así como una referencia para trabajar con ellos –introspección global del sistema.

El objeto `System` nos permite acceder al entorno de ejecución de la plataforma. Es un objeto que posee por padre a `nil`, y que tiene un miembro `objects` que hace

[46] Las distintas versiones de máquinas virtuales de la plataforma de Java [Kramer96], supone que el desarrollo de *applets* para las últimas versiones actuales de ésta, no puedan ser ejecutados por aquellos clientes que posean navegadores con versiones anteriores.

referencia a un objeto contenedor: posee, además de los miembros `super` e `id`, un miembro por cada objeto existente en tiempo de ejecución, siendo cada referencia miembro el `id` del objeto asociado.

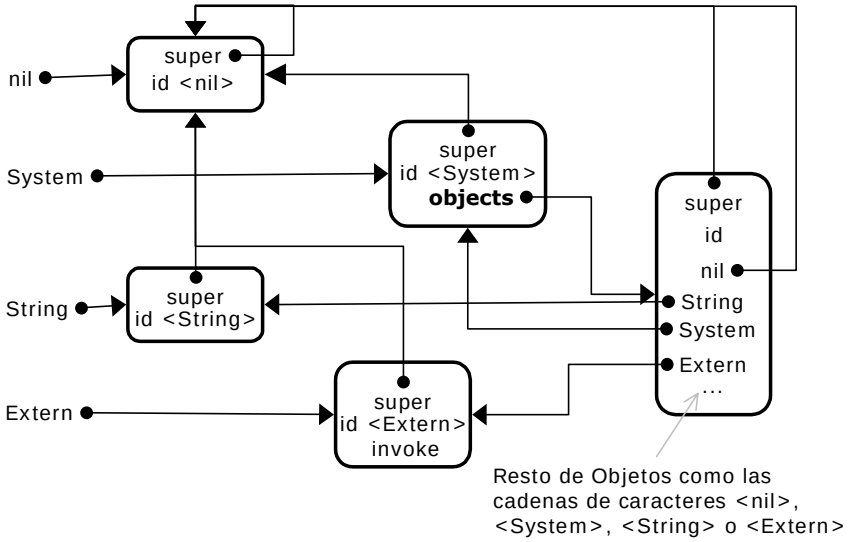


Figura 12.3: Acceso a los objetos de la máquina abstracta mediante el objeto "System".

Todas las primitivas de acceso a la plataforma física real o acceso al medio (la plataforma concreta en la que se esté ejecutando la máquina abstracta), se ubicarán como miembros de este objeto. Un ejemplo es el método `localhost` del sistema de distribución (§ 13.4) que nos devuelve el identificador de la máquina en la red de computadores.

12.1.4 Posibles Evaluaciones

Esta máquina abstracta posee un conjunto inicial de objetos con una estructura establecida. A raíz de éstos, crearemos la computación básica de la plataforma y posteriormente un entorno de computación. El modo de construir computaciones es llevado a cabo por el programador, mediante el siguiente conjunto de instrucciones:

1. Creación de referencias. Podremos declarar una referencia en el contexto actual de ejecución para su posterior utilización.
2. Asignación de referencias. Podremos hacer que una referencia apunte (se quede con la identidad) al objeto asociado a otra referencia existente.
3. Acceso a los miembros. Podremos acceder a cualquier miembro de un objeto en tiempo de ejecución.
4. Descosificación o evaluación de un miembro sobre un objeto. Podremos evaluar como comportamiento (instrucciones) un dato que sea miembro de un objeto. Este objeto será sobre el que actuará la computación: el objeto implícito.
5. Comentarios. Como hemos mencionado, la autodocumentación, y por tanto el acceso a los comentarios utilizados, es una característica propia de la plataforma. Los comentarios no se eliminarán por el analizador léxico, aunque su computación resulte nula.

6. Descosificación o evaluación de objetos cadena de caracteres. Cualquier secuencia de instrucciones de las anteriormente enumeradas, podrá tratarse como datos (objetos cadenas de caracteres) y evaluarse como computación.

Una instrucción finalizará siempre con el carácter punto y coma.

12.1.4.1 Creación de Referencias

Una referencia es el mecanismo de acceso a un objeto. La ejecución de la máquina parte de la existencia de un conjunto inicial de referencias. El usuario podrá utilizar éstas y crear otras para acceder a los distintos objetos del sistema.

Cada vez que se evalúa o descosifica un objeto, se crea un nuevo contexto de ejecución sobre el contexto existente –véase § 12.2.2. En éste contexto se pueden crear nuevas referencias, además de acceder a las referencias del contexto padre. Si aplicamos esta propiedad de modo transitivo, en realidad es factible acceder a cualquier contexto padre.

Una referencia ha de ser única en su contexto. Al destruirse un contexto (finalizar su ejecución) se liberan sus referencias, pero no los objetos a los que éstas acceden[47]. La creación de una nueva referencia en el contexto de ejecución se realiza mediante el componente léxico "`->`". Si la referencia en ese contexto ya existía, la evaluación es nula. Inicialmente una referencia posee la identidad –apunta al objeto– `nil`.

```
-> nuevaRef;
-> otraRef;
```

La creación de referencias en un contexto, unido a la posibilidad de acceder desde un contexto a otro padre, puede provocar conflictos de nombres. Si una referencia existe en un contexto y en alguno de los contextos padre, no se podrá acceder desde el contexto actual a las referencias de igual nombre en contextos padre. Esto representará un problema en la evaluación de miembros que toman un código como parámetro que accede a referencias del contexto padre; veremos posteriormente una solución a este problema.

Al tratar con un modelo de objetos basado en prototipos, no necesitamos declarar tipos para las referencias. Todo el proceso de validación semántica, así como la inferencia de tipos, se produce en tiempo de ejecución.

12.1.4.2 Asignación de Referencias

Hemos visto cómo crear nuevas referencias en un contexto. Una referencia es el mecanismo de acceso a un objeto. Podremos asignar referencias con el token `<-`

```
nuevaRef <- String;
otraRef <- <Hola a todos>;
```

La asignación de referencias produce un nuevo modo de acceso al objeto que estaba siendo utilizado por la referencia situada a la derecha de la flecha. El sentido de la asignación es el indicado por la flecha. La referencia que obtiene un nuevo valor, pierde la identidad del objeto que estaba utilizando.

12.1.4.3 Acceso a los Miembros

Hemos visto qué aspecto inicial tiene la máquina abstracta, y cómo se estructura y representa un objeto en memoria. Ahora mostraremos cómo se accede a los objetos existentes para su posterior manipulación.

[47] La liberación de las referencias de un contexto no implica la destrucción de los objetos referenciados, pero sí supone el decremento en una unidad de su miembro `refCount`.

Como hemos comentado en la introducción de este capítulo, hay que tener en cuenta que la abstracción base de la máquina es el objeto (orientación a objetos mediante prototipos), y que un objeto se define como un conjunto de referencias a otros objetos a los que podremos acceder en tiempo de ejecución. Un objeto es, por tanto, un contenedor indirecto de otros objetos[48].

Dado un objeto, accedemos a él mediante una referencia. Mediante ésta, accedemos al conjunto de objetos asociados al objeto referenciado. La sintaxis consiste en separar la referencia al objeto y la referencia al objeto que indica el nombre del miembro, mediante el componente léxico dos puntos:

```
nil:<id>;
String:<super>;
<id>:<id>;
```

Todas las expresiones anteriores devuelven referencias a objetos: el objeto `<nil>` en el primer caso, el objeto `nil` para el segundo ejemplo, y el objeto `<id>` en la tercera línea. La referencia devuelta puede asignarse a una referencia existente en el contexto, mediante la asignación de referencias, para la posterior utilización del objeto referenciado:

```
ancho <- rectangulo:<ancho>;
```

En el contexto de ejecución actual se pueden utilizar referencias locales a este contexto. En el ejemplo anterior, la referencia devuelta en el acceso al miembro se asigna a la referencia local `ancho`. Si ésta no existía previamente, se crea. Ahora existe otro medio (otra referencia), para acceder al objeto miembro `ancho`.

12.1.4.4 Evaluación o Descosificación de Objetos Miembro

Mediante la adición de miembros que denotan secuencias de instrucciones, puede añadirse comportamiento a los objetos del sistema. La invocación de un método se representará mediante la evaluación de un miembro descosificable, indicando el objeto o parámetro implícito sobre el que se actuará –al que se le enviará el mensaje.

	Referencia al objeto sobre el que se actúa		Nombre del miembro		Parámetros	
Invocación Síncrona	`RefObj`	`:`	`refNomMiembro`	`(`	`refParams`	`)`
Invocación Asíncrona	`RefObj`	`:`	`refNomMiembro`	`)`	`refParams`	`(`

Una vez accedido el objeto a evaluar, especificando siempre el parámetro implícito u objeto sobre el que se actúa, se podrá evaluar éste de modo síncrono (esperando el hilo actual a la finalización de su ejecución) o asíncrono (creándose un nuevo hilo sobre el contexto existente). Los parámetros se pasan en ambos casos mediante un conjunto de referencias, separadas por comas.

En la evaluación síncrona de un de un objeto podrá devolverse una referencia a un objeto. Sin embargo, puesto que en la evaluación asíncrona no existe una espera a la finalización de la invocación, no podrá asignarse ninguna referencia devuelta.

```
perimetro <- rectangulo:<perimetro>(<cm>);
```

[48] No contiene los propios objetos (composición) supeditando el ciclo de vida de éstos al suyo, sino que contiene referencias a otros objetos que siguen existiendo aunque finalice la existencia del contenedor.

```
        rectangulo:<dibujar>)(;
```

12.1.4.5 Comentarios

Una instrucción puede ser una referencia a un objeto cadena de caracteres. La utilización de esta posibilidad es de carácter documental para dicha instrucción. Si mantenemos la política común en procesadores de lenguajes de eliminar los comentarios en el análisis léxico, al acceder a la computación o instrucciones de la plataforma como datos, perderemos los comentarios. Para la máquina abstracta, un comentario posee entidad léxica y sintáctica, pero no semántica. La evaluación de un comentario es nula, pero no es eliminada por el analizador léxico.

Accediendo a los miembros de un objeto, podrán conocerse los comentarios que el programador haya identificado como necesarios. Si queremos dejar constancia de un comentario únicamente en el archivo de entrada, sin que aparezca posteriormente, podremos utilizar los caracteres a desoír en las cadenas de caracteres (§ 12.1.3.2):

```
< * Comentario para la posteridad * >;
...
< Comentario temporal que sólo queda en el archivo fuente >;
...
```

12.1.4.6 Evaluación de Objetos Cadena de Caracteres

No siempre es necesario evaluar un objeto como miembro. Puede interesar crear una instrucción o conjunto de instrucciones como datos, para posteriormente evaluarlas. La forma de representar instrucciones como datos, es utilizando objetos cadena de caracteres. Una vez creada la cadena, se podrá descosificar o evaluar ésta de forma síncrona o asíncrona.

```
        < perimetro <- rectangulo:<perimetro>(<cm>); > ();
        < rectangulo:<dibujar>)(; > )(;
```

La evaluación de un objeto cadena de caracteres se produce en un contexto nuevo creado sobre el contexto actual (el contexto en el que se evalúa dicho objeto). Veremos posteriormente cómo estas evaluaciones pueden devolver valores a los contextos padre, y cómo acceder a las referencias de éstos desde el contexto creado.

12.1.5 Delegación o Herencia Dinámica

El mecanismo de herencia ha sido utilizado en el paradigma de la orientación a objetos para reutilizar código e implementar aplicaciones adaptables. En nuestro lenguaje, la evaluación de un objeto indicando el parámetro implícito, no implica que el objeto miembro deba estar en ese objeto; puede formar parte de un objeto base –padre.

La especificación de una referencia a un objeto mediante el miembro `super`, indica dónde se deberá buscar el miembro no encontrado el paso de un mensaje. En el caso de que esto ocurra, se seguirá el proceso de búsqueda de modo recursivo en el objeto referenciado por `super`. Esta búsqueda finalizará cuando se cumpla que la referencia `super` del objeto examinado es una referencia a sí mismo, tratándose entonces del objeto ancestro `nil`.

Este mecanismo de herencia es dinámico, es decir, se puede modificar la referencia al objeto padre en tiempo de ejecución. La flexibilidad que esta capacidad otorga, hace que el mecanismo obtenido se defina como delegación y, en alguna bibliografía, como herencia dinámica.

12.1.6 Reflectividad Estructural y Computacional

La flexibilidad es el principal objetivo a obtener en el diseño de esta plataforma. Se trata de especificar una máquina abstracta computacionalmente simple y, apoyándonos en la extensibilidad y adaptabilidad otorgadas, construir un entorno de computación de un mayor nivel de abstracción. La consecución de este objetivo se debe llevar a cabo sin necesidad de modificar la máquina, aumentando su funcionalidad mediante su propio lenguaje.

Como hemos comentado a lo largo de esta tesis, la propiedad de reflectividad estructural es un mecanismo útil para conseguir los objetivos mencionados. Mediante un conjunto de primitivas, podremos acceder a la estructura y comportamiento de cualquier objeto existente en tiempo de ejecución, para conocer, modificar o ampliar dinámicamente cualquiera de sus miembros.

Como identificamos en el capítulo 6, la reflectividad computacional es un mecanismo utilizado para otorgar flexibilidad en la semántica computacional de la máquina abstracta. Al igual que multitud de sistemas (estudiados en el capítulo 7), para nuestra máquina utilizaremos un protocolo de metaobjetos (MOP) reducido, obteniendo así un mecanismo dinámico de flexibilidad computacional, con una expresividad restringida.

Determinadas primitivas computacionales de la máquina podrán modificarse mediante el propio lenguaje del sistema haciendo uso de un pequeño MOP. Mediante un mecanismo de reflectividad computacional restringido[49], la máquina facilitará al programador la posibilidad de modificar una serie de primitivas computacionales. De esta forma, no es necesario modificar la máquina para introducir nuevas características, pudiendo adaptar éstas mediante su propio lenguaje.

La máquina abstracta ofrece un MOP para modificar la semántica de las siguientes primitivas computacionales:

- Acceso a los miembros de un objeto, mediante la modificación del significado del operador dos puntos.
- Descosificación o evaluación síncrona y asíncrona de objetos, mediante la modificación de los operadores paréntesis; tanto para objetos miembro evaluables, como para objetos cadena de caracteres.

Veremos posteriormente en este documento la sintaxis del MOP que nos permite adaptar estas primitivas en el lenguaje –en § 13.3.3 introducimos la adaptación del acceso al miembro y en § 13.2.3 mostramos la modificación de la evaluación de código.

12.1.7 Objetos Miembro Primitivos

Existe un conjunto de objetos primitivos relativos al funcionamiento de la máquina abstracta. Estos objetos poseen una semántica definida por la propia máquina, y por tanto no están implementados en su propio lenguaje.

Podemos clasificarlos así:

- Creación, destrucción y referencias de objetos.

Objeto	Miembro	Parámetros	Descripción
`nil`	`new`	-	Crea un objeto nuevo con dos referen-

[49] Esta restricción supone establecer un protocolo a priori para identificar una sintaxis de modificación de la semántica de un conjunto reducido de primitivas.

Objeto	Miembro	Parámetros	Descripción
			cias miembro: "id" a un objeto derivado de `String` con su identificación única[50] y `super` a `nil`.
`nil`	`delete`	-	Libera la memoria del objeto implícito.
`nil`	`getRefCount`	-	Devuelve un objeto cadena de caracteres, indicando el número de referencias existentes dinámicamente a este objeto[51].

- Acceso a los objetos mediante reflectividad estructural.

Objeto	Miembro	Parámetros	Descripción
`nil`	`set`	1: Referencia a la clave 2: Referencia al contenido	Asigna un nuevo miembro con un nombre (clave) al objeto implícito. Si la referencia no existía, la crea.
`nil`	`has`	1: Referencia a la clave	Devuelve un objeto lógico indicando si existe o no la clave indicada.
`nil`	`firstKey`	-	Devuelve una referencia a un objeto cadena de caracteres que indica el nombre del primer miembro.
`nil`	`nextKey`	1:Referencia a la clave	Devuelve la referencia a la siguiente clave o `nil` si no hay más.
`nil`	`remove`	1: Referencia a la clave	Elimina la entrada con la clave pasada. El objeto al que apunta no es liberado.
`nil`	`get`	1: Referencia a la clave	Devuelve la referencia asociada al nombre de la clave pasada[52] o `nil` si no existe.

- Sincronización de hilos.

Objeto	Miembro	Parámetros	Descripción
`nil`	`enter`	-	Trata de acceder al monitor propio del objeto implícito. Si ya está utilizado, se bloquea en espera.

[50] El hecho de otorgar identificadores únicos no es una tarea sencilla. Hay que tener en cuenta que puede haber multitud de máquinas en distintas plataformas, y los objetos de cada una deberán tener identificadores globalmente únicos. Para ello, utilizaremos los identificadores universalmente únicos especificados por el *Open Software Foundation's Distributed Computing Environment* (OSF DCE) [Kirtland99], representando los IDs como cadenas de caracteres representativas de números de 32 bytes entre caracteres {} y en base 16.

[51] La utilidad principal de este miembro introspectivo, es facilitar al programador la implementación de distintos recolectores de basura en el propio lenguaje de la máquina, sin necesidad de modificar su implementación (§ 13.1).

[52] Esta primitiva tiene la misma función que el acceso a un miembro mediante el uso del operador dos puntos. Sin embargo, se utilizará una u otra en los sistemas de persistencia y distribución para diferenciar entre objetos locales y en RAM, frente a objetos remotos y persistentes –véase el diseño de los servicios de distribución y persistencia.

Objeto	Miembro	Parámetros	Descripción
`nil`	`exit`	`-`	Libera el recurso propio del objeto implícito. El siguiente hilo esperando, entrará a él. Si no existiese ninguno, la evaluación es nula.

A continuación se presentarán ejemplos del empleo de estos objetos miembro primitivos, para ampliar el nivel de abstracción de la plataforma, expresándolo en su propio lenguaje.

12.2 Extensión de la Máquina Abstracta

Hasta este punto, hemos descrito el funcionamiento primitivo de la máquina abstracta, mencionando las primitivas computacionales ofrecidas y los mecanismos a utilizar para extender sus abstracciones –mediante los miembros primitivos del objeto `nil`. A partir de este punto, extenderemos, haciendo uso del propio lenguaje de la máquina, su capacidad computacional, sin necesidad de modificar el código fuente de la máquina virtual.

Al igual que la plataforma de Smalltalk [Krasner83], la programación llevada a cabo será un conjunto de sentencias codificadas en un archivo imagen. Inicialmente la plataforma ejecutará el código de este archivo para extender su abstracción.

12.2.1 Creación de Nuevas Abstracciones

Vamos a ver cómo podemos crear nuevos objetos en el sistema. Inicialmente tenemos los objetos `String`, `nil`, `System` y `Extern`. Para poder añadir funcionalidades al conjunto de todos los objetos existentes en la plataforma, podremos crear un objeto ancestro `Object` que implemente estas funcionalidades. Este diseño ha sido adoptado por numerosos lenguajes como Smalltalk, Object Pascal o Java.

Podríamos introducir estas funcionalidades en el objeto ancestro existente `nil`. Sin embargo, dejaremos en `nil` aquellas capacidades semánticas primitivas que nos ofrece la plataforma[53].

Creamos inicialmente un objeto, almacenándolo en una referencia global (de contexto 0) `Object`:

```
-> Object;
Object <- nil:<new>();
Object:<set>(<id>,<Object>);
```

Un criterio seguido en la creación de esta plataforma, es separar los objetos que representan estados de una entidad de los objetos que representan comportamiento de un grupo. Este tipo de objetos, de comportamiento o rasgo, se suelen denominar objetos *trait*. Las referencias a este tipo de objetos las definiremos con la primera letra en mayúscula; el resto, con la primera en minúscula[54].

Otro convenio seguido es la definición de la referencia miembro `<id>` de cualquier objeto de la plataforma (no un objeto de usuario), y cambiarlo al nombre de su referencia

[53] En este punto en el que nos encontramos ampliando el nivel de abstracción de la plataforma mediante su propio lenguaje, pueden llevarse a cabo multitud de diseños para definir la estructura de su lenguaje. Esto es una muestra la flexibilidad otorgada.
[54] Simplemente es un convenio para poder leer el código de una forma más sencilla.

en el contexto global. Por ejemplo, al crear el objeto `Object` en el caso anterior, su identificador será un número hexadecimal comprendido entre `{` y `}` –funcionamiento de la primitiva `new`. Para identificarlo como objeto perteneciente a la plataforma, su miembro `<id>` se modifica al nombre de su referencia: `<Object>`.

Puesto que `Object` va a ser el objeto base –directa o indirectamente– de cualquier objeto de la plataforma, exceptuando `nil`, modificaremos el objeto base de `String`, `Extern` y `System`:

```
String:<set>(<super>,Object);
Extern:<set>(<super>,Object);
System:<set>(<super>,Object);
```

La plataforma va a tener una forma de operar basada en valores lógicos bivaluados. El objeto *trait* `Boolean` identificará el comportamiento de este tipo de objetos:

```
-> Boolean;
Boolean <- nil:<new>();
Boolean:<set>(<id>,<Boolean>);
Boolean:<set>(<super>,Object);
```

Aunque la plataforma no proporcione primitivas lógicas, definiremos dos objetos `true` y `false` que representarán respectivamente las constantes positiva y negativa en la lógica de la máquina.

```
-> true;
true <- nil:<new>();
true:<set>(<id>,<true>);
true:<set>(<super>,Boolean);
-> false;
false <- nil:<new>();
false:<set>(<id>,<false>);
false:<set>(<super>,Boolean);
```

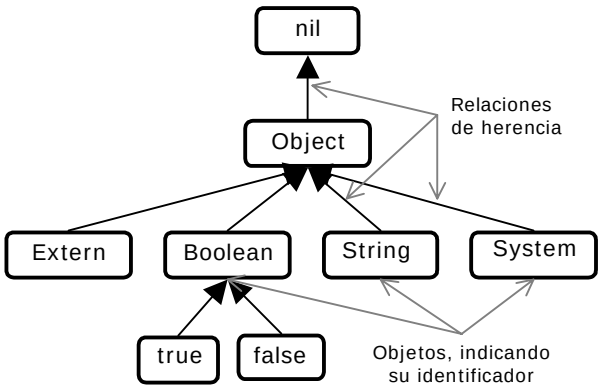


Figura 12.4: Objetos existentes en tiempo de ejecución.

12.2.2 Evaluación de Objetos

Hasta ahora hemos creado objetos y modificado su estructura, utilizando objetos evaluables primitivos como los miembros del objeto `nil`, `new` y `set`. Hemos comentado también, que el comportamiento de los objetos viene dado por la evaluación dinámica de objetos. Veremos a continuación la semántica propia de la evaluación de este tipo de objetos.

12.2.2.1 Evaluación de Objetos Miembro

Para añadir un objeto miembro evaluable a otro objeto, deberemos ejecutar el método `<set>` del objeto en cuestión, pasándole como parámetros el nombre del miembro a crear, y un objeto `String` representativo del código que se evaluará en la invocación del método.

Cuando se evalúe o ejecute el miembro creado, se construirá un nuevo contexto de ejecución en el hilo existente. En este nuevo contexto, se pueden crear nuevas referencias que se destruirán al finalizar el mismo. Un contexto podrá acceder a las referencias del contexto que lo creó –contexto padre. Una referencia de un contexto oculta las referencias de igual nombre, ubicadas en sus contextos padre.

```
devolución <- objeto:<miembro>(parámetros);
```

El nuevo contexto, creado en la invocación o evaluación de un objeto miembro, se caracteriza por tener las siguientes referencias adicionales:

- `sender`: Esta referencia posee la identidad del objeto utilizado para acceder al miembro. Es el parámetro u objeto implícito sobre el que actuará la computación del miembro al evaluarse. En el ejemplo anterior, es una referencia a `objeto`.

- `mySelf`: Referencia al objeto que posee el miembro que se está evaluando. Esta referencia no tiene por qué tener el mismo valor que `sender`, puesto que el miembro que se evalúa puede ser distinto al parámetro implícito, gracias al mecanismo de delegación. En el ejemplo anterior, `mySelf` puede ser una referencia a `objeto` o a alguno de sus objetos padre –directa o indirectamente.

- `params`: Esta referencia apunta a un objeto contenedor de los parámetros pasados. Si no se han pasado parámetros, apuntará a `nil`.

  El objeto contenedor de parámetros, `params`, tendrá los dos miembros de todo objeto, más un miembro por cada parámetro. Los miembros están etiquetados con números del cero al número de parámetros menos uno.

```
parámetroPrimero <- params:<0>;
parámetroSegundo <- params:<1>;
```

- `return`: Referencia que posee la identidad del objeto devuelto por la evaluación del método. Si la evaluación del miembro es asíncrona, la referencia `return` no será creada, puesto que este tipo de invocaciones no puede devolver ningún objeto.

  En nuestro ejemplo, el objeto que asociemos a la referencia `return` dentro del método será el objeto al que apunte la referencia `devolución`, después de su invocación.

Un ejemplo de implementación del miembro `suma` de un objeto `Entero` puede ser:

```
Entero:<set>(<suma>,<
        -> param;
        param <- params:<0>;
        return <- sender:<+>(parametro);
>);
```

En su evaluación, `param` es el único parámetro pasado, `sender` es el objeto implícito utilizado en su invocación (el objeto `Entero` o alguno de sus derivados), `mySelf` es una referencia a `Entero`, y la devolución del método es el resultado de otra evaluación.

12.2.2.2 Evaluación de Objetos Cadena de Caracteres

Además de poder evaluar objetos miembro que actúan sobre objetos implícitos, mediante la referencia `sender`, también podemos evaluar objetos cadena de caracteres, sin necesidad de utilizar un objeto implícito.

Este tipo de evaluación provoca la creación de un nuevo contexto, al igual que la evaluación de un miembro. El contexto creado puede también crear nuevas referencias, así como acceder a las referencias del contexto padre. De forma adicional, podrá devolver un valor –si no se evalúa de forma asíncrona– y recibir parámetros.

Posee, por tanto, las referencias `params` y `return`, y carece de `sender` y `mySelf`. De esta forma, si en algún contexto padre existieren estas referencias, se podría disponer de éstas desde el nuevo contexto creado, aunque no hayan sido definidas en el actual.

```
devolución <- <Instrucción>(parámetros);
```

12.2.3 Creación de Objetos Evaluables

A raíz de la semántica de evaluación de objetos, añadiremos funcionalidad computacional a los objetos creados en § 12.2.1.

12.2.3.1 Creación de Objetos con Identificador

El miembro `new` del objeto `nil` ofrece la posibilidad de crear objetos con un identificador único. Hemos visto cómo los identificadores de los objetos creados, constituyentes de la plataforma, son modificados por un identificador legible por el humano –`true`, `false`, `Boolean` u `Object`. A lo largo de la creación del entorno de programación, este criterio de modificación de identificadores seguirá llevándose a cabo. Para facilitar este proceso, se asignará a `Object` un miembro `newWithId` que implemente este algoritmo:

```
Object:<set>(<newWithId>,<
      -> ID;
      ID <- params:<0>;
      return <- self:<new>();
      return:<set>(<id>,ID);
>);
```

El parámetro pasado en la evaluación del miembro, es el identificador a asignar al objeto creado.

12.2.3.2 Igualdad de Objetos

La comparación de igualdad de objetos se apoya en la primitiva de comparación de igualdad de cadenas de caracteres. Este funcionamiento semántico de comparación de cadenas forma parte del protocolo de invocaciones externas –computación operacional básica, § 12.1.3.3.

```
String:<set>(<==>,<
            -> sParam;
            sParam <- params:<0>;
```

```
            return <- Extern:<invoke>(<String>,<==>,sender,sParam);
            >);
```

Para poder comparar cualquier objeto implícito, excepto `nil`, añadimos el miembro de igualdad al objeto *trait* `Object`, que compara dos objetos a partir de sus identificadores únicos.

```
Object:<set>(<==>,<
            -> oParam;
            -> sParamID;
            -> sSenderID;
            oParam <- params:<0>;
            sParamID <- oParam:<id>;
            sSenderID <- sender:<id>;
            return <- sParamID:<==>(sSenderID);
            >);
```

Añadimos a `nil` el método == para poder compararlo cualquier otro objeto:

```
nil:<set>(<==>,<
            -> oParam;
            -> sParamID;
            oParam <- params:<0>;
            sParamID <- oParam:<id>;
            return <- sParamID:<==>(<nil>);
            >);
```

12.2.3.3 Operaciones Lógicas

Definimos, sobre las dos constantes lógicas, dos métodos para la evaluación de un parámetro en función de su valor lógico. En la descosificación de los miembros `ifTrue` e `ifFalse`, se pasa como parámetro el código a evaluar, representado como un objeto cadena de caracteres.

```
true:<set>(<ifTrue>,<
            -> _sTrueCode;
            _sTrueCode <- params:<0>;
            return <- _sTrueCode();
            >);
true:<set>(<ifFalse>,<>);
false:<set>(<ifTrue>,<>);
false:<set>(<ifFalse>,<
            -> _sFalseCode;
            _sFalseCode <- params:<0>;
            return <- _sFalseCode();
            >);
```

En el objeto `Boolean` introducimos los métodos `if` e `ifElse`.

```
Boolean:<set>(<if>,<
            -> _sCode;
            _sCode <- params:<0>;
            return <- sender:<ifTrue>(_sCode);
            >);
Boolean:<set>(<ifElse>,<
            -> _sIfCode;
            -> _sElseCode;
            _sIfCode <- params:<0>;
            _sElseCode <- params:<1>;
            sender:<ifTrue>(_sIfCode);
            sender:<ifFalse>(_sElseCode);
            >);
```

Añadimos al objeto *trait* `Boolean` las posibilidades de negar un objeto lógico y de calcular las operaciones lógicas binarias `and` y `or`.

```
Boolean:<set>(<not>,<
            -> _booleanNot_result;
            sender:<ifElse>(
                  <_booleanNot_result <- false;>,
                  <_booleanNot_result <- true;>
                  );
            return <- _booleanNot_result;
            >);
Boolean:<set>(<and>,<
            -> _booleanAnd_result;
            -> _booleanAnd_bParam;
            _booleanAnd_bParam <- params:<0>;
            sender:<ifElse>(
                  < _booleanAnd_bParam:<ifElse>(
                        < _booleanAnd_result <- true; >,
                        < _booleanAnd_result <- false; >
                        );>,
                  < _booleanAnd_result <- false; >
                  )
            return <- _booleanAnd_result;
            >);
Boolean:<set>(<or>,<
            -> _booleanOr_result;
            -> _booleanOr_bParam;
            _booleanOr_bParam <- params:<0>;
            sender:<ifElse>(
                  < _booleanOr_result <- true; >,
                  < _booleanOr_bParam:<ifElse>(
                        < _booleanOr_result <- true; >,
                        < _booleanOr_result <- false; >
                        );>
                  );
            return <- _booleanOr_result;
            >);
```

En § 12.1.4.1, se anticipó el problema de ocultación de referencias de los contextos padre por la creación de referencias locales de igual nombre. Supongamos el siguiente ejemplo de programación:

```
      ...
      -> _sIfCode;
      <* Campo booleano que expresa una condición lógica *>;
      booleano:<ifElse>(
            < _sIfCode <- <Booleano es True> >; ,
            < _sIfCode <- <Booleano es False>; );
      <* ¿Qué vale _sIfCode aquí? *>;
      ...
```

Escribiendo este código, pretendemos que, en función de un valor, lógico la referencia `_sIfCode` posea un determinado valor; pero no lo conseguimos. La evaluación del miembro `ifElse` crea una referencia en su contexto (un nivel superior al actual) con nombre `_sIfCode`. Los dos parámetros pasados se evalúan en un contexto superior al último, accediéndose a la referencia `_sIfCode` del contexto del miembro `ifElse`, y no al deseado.

Como criterio general de programación, definiremos nombres absolutos siempre que queramos acceder a una referencia de un contexto padre desde código a evaluar. Los nombres absolutos podrán ser generados anteponiendo el nombre del objeto y del método,

al nombre de la referencia[55]. Esto sucede en la referencia `sCode` del miembro `while` de `String` que implementaremos posteriormente, así como en la referencia `result` de los métodos `and`, `or` y `not` de `Boolean`.

El criterio de utilizar referencias absolutas cuando accedamos a ellas desde contextos derivados es una tarea sencilla para el generador de código de un compilador. Sin embargo, cuando codifiquemos directamente el entorno de programación, el sistema no nos obliga a codificar con este criterio, y deberemos proteger la sobreescritura de referencias. De esta forma, siempre que un contexto cree referencias y posteriormente evalúe un código pasado como parámetro, las referencias creadas comenzarán con el carácter subrayado ("_"). Reservamos, pues, los identificadores que comiencen con este carácter para el desarrollo de la plataforma, y no generar así ocultación de referencias.

12.2.3.4 Iteraciones

Para programar de modo estructurado será necesario soportar algún tipo de instrucción iterativa. En nuestro caso crearemos esta funcionalidad basándonos en la evaluación de objetos miembro y en la recursividad[56]. Implementaremos un método `while` como miembro del objeto `String`, siguiendo el siguiente diagrama de flujo:

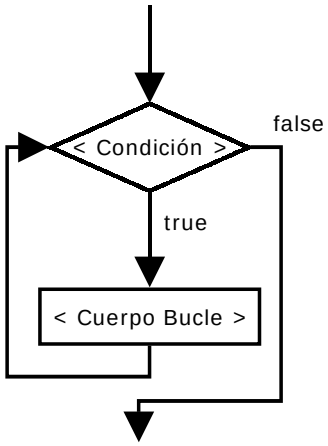


Figura 12.5: Organigrama de la evaluación del método "while".

La condición, objeto implícito de la invocación, será representada como una cadena de caracteres evaluable de forma síncrona y sin parámetros. En cada iteración se evaluará este objeto, que será el parámetro implícito de la invocación. La evaluación de este tipo de condición deberá devolver un valor lógico que, en caso de ser `true`, producirá la evaluación del cuerpo del bucle.

El cuerpo del bucle `while` es un objeto cadena de caracteres que será evaluado de forma síncrona. El cuerpo del bucle puede recibir parámetros que se le pasarán mediante un objeto contenedor. Este objeto coleccionará los n parámetros etiquetándolos desde 0 hasta n-1.

[55] Como técnica de programación, es una tarea tediosa puesto que ha de ser el programador el que lleve esta responsabilidad. Sin embargo, el lenguaje de la máquina abstracta es un lenguaje de bajo nivel y no está pensado para codificar directamente aplicaciones de usuario sobre él. Un compilador de un lenguaje de alto nivel, podrá generar automáticamente las referencias siguiendo el criterio definido.

[56] Obviamente la recursividad hará que las estructuras iterativas sean menos eficientes que las existentes en otras plataformas. Como se menciona en § 12.2.3.6, la plataforma requiere la implementación de la optimización de recursividad por la cola (*Tail Recursion Optimisation*) [Kessin99] que disminuye la diferencia de eficiencia. Sin embargo, queremos dejar constancia que la eficiencia no es un objetivo de nuestro diseño.

La evaluación del miembro `while` de un objeto cadena de caracteres –objeto implícito representante de la condición a evaluar–, recibirá como primer parámetro el objeto que representa el cuerpo del bucle y, como segundo, un objeto contenedor de los parámetros a pasar al cuerpo del `while`.

Como ejemplo, supongamos que nos encontramos en un contexto de ejecución en el que existen las referencias `ref`, `objeto`, `metodo`, `param0` y `param1` y deseamos:

- Que en cada iteración se compare la referencia `ref` con `nil`. Si la comparación es cierta, se sigue iterando.
- Que en cada iteración, se evalúe el método con nombre `metodo`, miembro de `objeto`, pasándole los parámetros `param0` y `param1`. El resultado de la evaluación deberá ser asignado a `ref`.

Una secuencia de instrucciones utilizadas en el contexto es la siguiente:

```
-> cParams
cParams <- nil:<new>()
cParams:<set>(<super>,Object)
cParams:<set>(<0>,objeto);
cParams:<set>(<1>,metodo);
cParams:<set>(<2>,param0);
cParams:<set>(<3>,param1);
< return <- ref:<==>(nil); >:<while> (
             <      -> cParams;
                    cParams <- params:<0>;
                    oObjeto <- cParams:<0>;
                    sMetodo <- cParams:<1>;
                    oParam0 <- cParams:<2>;
                    oParam1 <- cParams:<3>;
                    ref <- oObjeto:sMetodo(oParam0,oParam1);

             >,
                    cParams
             );
```

Realmente, el paso de todos los parámetros existentes en el contexto padre del nuevo contexto creado, para la evaluación del miembro `while` es innecesario, puesto que desde un contexto se pueden acceder a las referencias de sus contextos padre.

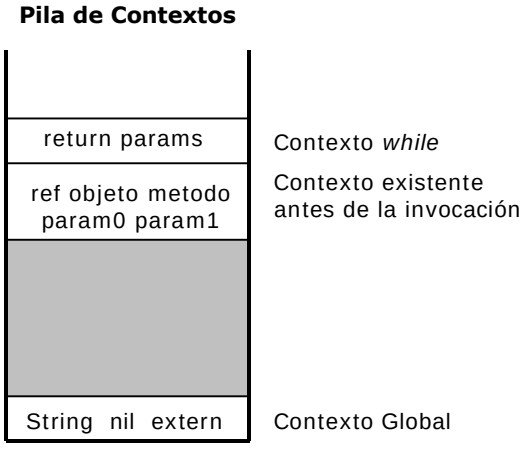


Figura 12.6: Referencias existentes en los contextos de ejecución del miembro “while”.

Si se ejecuta el cuerpo del bucle como un contexto hijo del contexto que posee las referencias `ref`, `objeto`, `metodo`, `param0` y `param1`, no es necesario pasar éstas como

parámetros al cuerpo. El paso de parámetros se utiliza precisamente cuando esto no ocurre –uno de los posibles usos, es el paso de parámetros de referencias a miembros. La implementación anterior podría reducirse al siguiente código:

```
< return <- ref:<==>(nil); >:<while> (
            < ref <- objeto:metodo(param0,param1); > ,
            nil
            );
```

La implementación del método `while` evalúa la condición –objeto implícito, `sender`– y, en el caso de que sea cierta, se ejecutará el cuerpo seguido de una llamada recursiva a la evaluación del objeto `while`.

```
String:<set>(<while>,<
            -> _stringWhile_sCode;
            -> _stringWhile_cParams;
            -> _bCond;
            -> _stringWhile_sender
            _stringWhile_sCode <- params:<0>;
            _stringWhile_cParams <- params:<1>;
            <* Condition is evaluated *>;
            _bCond <- sender();
            _stringWhile_sender <- sender;
            _bCond:<if>(<
                  <* The while-code is evaluated *>;
                  _stringWhile_sCode(_stringWhile_cParams);
                  _stringWhile_sender:<while>(_stringWhile_sCode,
                              _stringWhile_cParams);
                  >);
            >);
```

12.2.3.5 Recorrido de Miembros

El tratamiento genérico de todos los miembros de un objeto será un proceso útil en su tratamiento introspectivo. Su clonación, replicación o almacenamiento en disco, son ejemplos para los que la evaluación de una rutina sobre todos los miembros de un objeto, hace que se facilite su implementación.

Implementaremos un método `forEach` del objeto `Object` que permitirá evaluar, de forma iterativa, un código con un determinado parámetro, para todos los miembros del objeto implícito, excluyendo `id` y `super`. La invocación de este método será de la forma:

```
      objeto:<forEach>(<Código a Evaluar>,parámetro);
```

El primer parámetro será un código a evaluar de forma síncrona, recibiendo éste, en cada evaluación, tres parámetros:

- Como primer parámetro recibirá una referencia a la clave o nombre del miembro del objeto, propio de la iteración.
- El segundo parámetro será una referencia al miembro asociado al nombre representado por el primer parámetro.
- El último parámetro será el segundo argumento pasado en la evaluación del miembro `forEach`.

Implementaremos este miembro de la siguiente forma:

```
Object:<set>(<forEach>,<
      -> _objectForEach_sCode;
      -> _objectForEach_sKey;
```

```
        -> _objectForEach_cParams;
        -> _objectForEach_sender;
        _objectForEach_sCode <- params:<0>;
        _objectForEach_cParams <- params:<1>;
        _objectForEach_sKey <- sender:<firstKey>();
        _objectForEach_sender <- sender;
        <return <- _objectForEach_sKey:<!=>(nil);>:<while>(<
                -> _bNotIsId;
                -> _bNotIsSuper;
                -> _bIsMember;
                _bNotIsId <- _objectForEach_sKey:<!=>(<id>);
                _bNotIsSuper <- _objectForEach_sKey:<!=>(<super>);
                _bIsMember <- _bNotIsId:<and>(_bNotIsSuper);
                _bIsMember:<ifTrue>(<
                        ->_oContent;
                        _oContent <- _objectForEach_sender:
                                                _objectForEach_sKey;
                        _objectForEach_sCode(_objectForEach_sKey,
                                        _oContent,_objectForEach_cParams);
                        >);
                _objectForEach_sKey <- _objectForEach_sender:<nextKey>(
                                        _objectForEach_sKey);
                >, nil);
        >);
```

En la clonación de objetos implementada en § 12.2.7.2, se muestra un ejemplo de la utilización de este miembro.

12.2.3.6 Bucles Infinitos

Hemos implementado iteraciones mediante llamadas recursivas, siendo éstas menos eficientes que los saltos en la ejecución de código –más propios de las máquinas físicas existentes. Sin embargo, a la hora de implementar un bucle infinito, este inconveniente de ineficiencia se traduce en un error en tiempo de ejecución: la llamada recursiva sin condición de parada agotará los recursos de nuestro sistema, desbordando la pila de contextos.

Para que nuestra máquina abstracta permita evaluar bucles infinitos mediante recursividad, sin que se produzcan errores de ejecución, es necesario implementar la optimización de recursividad por la cola (*Tail Recursion Optimisation*) [Kessin99], implementada por otros lenguajes como Scheme [Abelson2000].

La optimización comentada, puede realizarse cuando la última instrucción de un método es una evaluación de un miembro o de una cadena de caracteres, y su resultado es lo que devuelve este método –se le asigna a `return`. En el caso de que el método se esté evaluando de forma asíncrona, la última premisa no es necesaria.

En el caso de que la condición mencionada en el párrafo anterior se cumpla, la invocación constituyente de la última instrucción no necesitará un nuevo contexto de ejecución. Pongamos un escenario como ejemplo:

```
Objeto:<set>(<metodo>,<
        -> referencia1;
        -> referencia2;
        -> referencia3;
        ...
        return <- referencia2:<otroMetodo>(referencia1,referencia3);
>);
```

El estado de la pila de contextos en la ejecución de la última sentencia es:

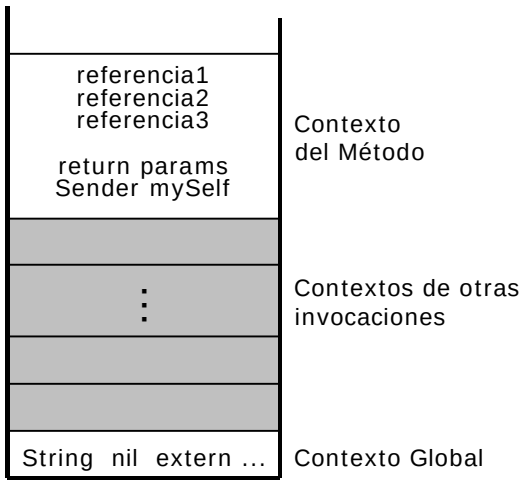


Figura 12.7: Estado de la pila de contextos en la evaluación del método.

Si no se crea un nuevo contexto en la invocación de `otroMetodo`, sucede lo siguiente:

1. Se sobreescribe el valor de `sender`, `mySelf` y `params` con los de la nueva invocación. No supone un problema puesto que, cuando finalice la ejecución del método, estos valores no se utilizarán, ya que la evaluación del método es la última sentencia.
2. La referencia al valor de devolución (`return`) se sobreescribe. Esto podría ser un problema, puesto que no tiene por qué coincidir la devolución de `otroMetodo` con la de `metodo`. Por lo tanto, para llevar a cabo esta optimización, es necesario que ambas coincidan, es decir, que la devolución de la invocación de `otroMetodo` se asigne a `return` –segunda premisa.
3. Las referencias locales existen en el contexto de la ejecución de `otroMétodo`. Esta característica la ofrece nuestra máquina, al permitir acceder a los contextos padre.

Si deseamos implementar un código que se ejecute repetitivamente de forma indefinida, deberemos situar como última línea de éste una invocación a sí mismo, y que la devolución de dicha invocación sea asignada a `return`. De esta forma, la máquina no generará un error de ejecución.

12.2.4 Aritmética de la Plataforma

En la mayoría de los lenguajes de programación existe el concepto de tipo, para reducir el número de errores de programación en tiempo de ejecución, y para obtener una mayor eficiencia en la representación de la información. Nuestra plataforma no posee proceso de compilación en su lenguaje nativo, por lo que no identifica tipos en su lenguaje de programación (un compilador de Java o C++ a esta plataforma, sí podría realizar una comprobación estática de tipos). Para la máquina abstracta, la comprobación de tipos se realiza en tiempo de ejecución, para conocer si un determinado objeto posee un miembro –comprobación dinámica de tipos [Cardelli97].

Aunque sea negativo respecto a la eficiencia de la plataforma, no identificamos tipos relacionados con valores numéricos, sino que manejamos una única constante: la cade-

na de caracteres. Determinadas evaluaciones sobrecargadas[57] sobre este tipo de datos, implementará dinámicamente una inferencia del tipo asociado, para conocer la semántica de la operación a realizar. Este proceso adicional conlleva una pérdida de eficiencia frente a la utilización de tipos estáticos, puesto que es el usuario quien identifica el tipo de cualquier objeto. Sin embargo, de este modo se reduce el número de primitivas operacionales de la plataforma.

Los objetos cadena de caracteres pueden identificar tres semánticas totalmente distintas:

- Semántica numérica. La representación de los datos se interpretará como un contenido numérico. Se podrán realizar operaciones numéricas típicas sobre estos datos, teniendo en cuenta que se puede establecer, a su vez, una clasificación numérica como[58]:
  1. Entera. Constantes enteras con signo opcional.
  2. Real. Representación real en formato de punto fijo y mantisa más exponente.
- Semántica computacional. Las cadenas de caracteres pueden poseer datos evaluables o descosificables, para producir computación de la plataforma.
- Semántica de datos. Las cadenas de caracteres podrán ser datos cuyo significado será interpretado por el humano.

En los rangos numéricos preestablecidos por los lenguajes de programación, se especifica el tamaño que necesitará una variable de dicho tipo. En nuestra plataforma no definimos tamaños máximos, sino que éstos se limitan al tamaño de la plataforma física de ejecución. Al representar los valores numéricos como cadenas de caracteres, no limitaremos su longitud[59].

Es importante observar que tanto la clasificación de tipos numéricos (enteros y reales), como la no-restricción de rangos en éstos, es posible modificarlos en cualquier momento, sin necesidad de modificar la máquina abstracta. Al separar la semántica de las operaciones de la propia plataforma, rompemos con las definiciones monolíticas clásicas de máquinas virtuales, y nos permite definir su lenguaje con diversas características –adaptabilidad. Un ejemplo sería primar la eficiencia optando por:

- Definir nuevos tipos numéricos basados en objetos *traits* –al igual que Smalltalk.
- Definir rangos máximos para los valores numéricos.
- No definir mensajes sobrecargados con distintas semánticas que necesiten analizar léxicamente el tipo de dato.

12.2.4.1 Miembros de Semántica Aritmética

Existe un conjunto de mensajes polimórficos respecto al tipo de dato numérico al que puede ser aplicado. Los siguientes objetos son miembros del objeto *trait* `String`, que define la semántica obvia para valores numéricos enteros y reales:

[57] Operación sobrecargada en relación con la teoría de tipos definida por Luca Cardelli [Cardelli97].
[58] Podemos insertar más grupos de representación numérica, como por ejemplo números racionales, sin necesidad de modificar la máquina, puesto que esta semántica estará implementada de forma externa.
[59] Java permite utilizar los tipos simples restringiendo su rango, pero también posee la clase `java.math.BigInteger` para representar enteros sin límites [Gosling96].

- Miembro +: Suma.
- Miembro -: Diferencia
- Miembro *: Producto.
- Miembro /: División.

Para todos estos miembros se define la semántica de tipos de la siguiente forma[60]:

- Si los dos operandos son ambos números enteros, el objeto devuelto es un número entero.
- Si los dos operandos son numéricos y uno de ellos es real, el resultado es numérico y real.
- Si alguno de los dos no posee semántica numérica, el resultado es `nil`, excepto en el caso del miembro +, en que se devuelve una concatenación de las cadenas de caracteres.

Además definimos:

- Miembro %: Semántica de resto sobre números enteros. Devuelve `nil` en el caso de que al menos uno de los dos operandos no sea un número entero.

A continuación añadimos esta computación a la plataforma:

```
String:<set>(<+>,<
             -> sParam;
             sParam <- params:<0>;
             return <- Extern:<invoke>(<String>,<+>,sender,sParam);
             >);
String:<set>(<->,<
             -> sParam;
             sParam <- params:<0>;
             return <- Extern:<invoke>(<String>,<->,sender,sParam);
             >);
String:<set>(<*>,<
             -> sParams;
             sParam <- params:<0>;
             return <- Extern:<invoke>(<String>,<*>,sender,sParam);
             >);
String:<set>(</>,<
             -> sParam;
             sParam <- params:<0>;
             return <- Extern:<invoke>(<String>,</>,sender,sParam);
             >);
String:<set>(<%>,<
             -> sParam;
             sParam <- params:<0>;
             return <- Extern:<invoke>(<String>,<%>,sender,sParam);
             >);
```

12.2.5 Miembros Polimórficos de Comparación

Definimos un conjunto de miembros polimórficos de comparación en el objeto `String`; todos ellos devuelven objetos lógicos. Su significado va a ser siempre el mismo, aunque trabaje con distintos tipos de datos inferidos dinámicamente:

[60] Estas expresiones de tipo podrían variar introduciendo por ejemplo el álgebra de números racionales.

- Miembro ==. Comparación de igualdad. Se podrá comparar valores numéricos y datos cadena de caracteres polimórficamente. Por ejemplo, la comparación `<3.2>:<==>(<32E-1>)` debe devolver `true`, aunque la comparación lexicográfica –como cadena de caracteres– resulte falsa.
- Miembro !=. Comparación de desigualdad.
- Miembro >. Comparación mayor.
- Miembro >=. Comparación mayor o igual.
- Miembro <. Comparación menor.
- Miembro <=. Comparación menor o igual.

Para implementar estas operaciones polimórficas de comparación, apoyándonos en primitivas externas, sólo necesitamos dos de ellas y las operaciones lógicas ya definidas. Vamos a definir como primitivas, las operaciones de igualdad y mayor:

Operaciones a implementar	**Implementación sobre primitivas Igual y Mayor**
Distinto	NOT(Igual)
Mayor o Igual	OR(Igual, Mayor)
Menor	AND(NOT(Mayor), Distinto)
Menor o Igual	NOT(Mayor)

Reducimos así el número de primitivas a implementar, codificando las operaciones de la siguiente forma (la operación de igualdad ha sido implementada previamente):

```
String:<set>(<\>>,<
             -> sParam;
             sParam <- params:<0>;
             return <- Extern:<invoke>(<String>,<\>>,sender,sParam);
             >);
String:<set>(<!=>,<
             -> sParam;
             -> bEqual;
             sParam <- params:<0>;
             bEqual <- sender:<==>(sParam);
             return <- bEqual:<not>();
             >);
String:<set>(<\>=>,<
             -> sParam;
             -> bEqual;
             -> bGreater;
             sParam <- params:<0>;
             bEqual <- sender:<==>(sParam);
             bGreater <- sender:<\>>(sParam);
             return <- bEqual:<or>(bGreater);
             >);
String:<set>(<\<>,<
             -> sParam;
             -> bGreater;
             -> bNotGreater;
             -> bNotEqual;
             sParam <- params:<0>;
             bGreater <- sender:<\>>(sParam);
             bNotGreater <- bGreater:<not>();
             bNotEqual <- sender:<!=>(sParam);
             return <- bNotGreater:<and>(bNotEqual);
             >);
```

```
String:<set>(<\<=>,<
             -> sParam;
             -> bGreater;
             sParam <- params:<0>;
             bGreater <- sender:<\>>(sParam);
             return <- bGreater:<not>();
             >);
```

12.2.6 Miembros de Objetos Cadena de Caracteres

Como miembros del objeto `String`, tratando los objetos como datos simples (no numéricos ni computación), definimos tres primitivas que nos facilitarán el manejo de estos datos:

- Miembro `append`. Este miembro siempre devolverá la concatenación del parámetro implícito y el parámetro pasado. Esta semántica se produce con el miembro `+`, siempre que alguno de los operandos no sea numérico.
- Miembro `length`. Devuelve un objeto cadena de caracteres con semántica entera, indicando el número de caracteres que posee el parámetro implícito.
- Miembro `subString`. Recibe dos parámetros cadena de caracteres de semántica numérica. El primero indica una posición; el segundo, el número de caracteres a raíz de los cuales se devolverá un objeto cadena de caracteres, que contenga el número indicado, a partir de la posición señalada por el primer parámetro, dentro del objeto implícito.

  Para pasar dos parámetros con una sola referencia, creamos un objeto contenedor con dos miembros etiquetados como `<0>` y `<1>`.

Estas operaciones son todas de semántica primitiva:

```
String:<set>(<append>,<
             -> sParam;
             sParam <- params:<0>;
             return <- Extern:<invoke>(<String>,<append>,sender,sParam);
             >);
String:<set>(<length>,<
             -> sParam;
             sParam <- params:<0>;
             return <- Extern:<invoke>(<String>,<length>,sender,sParam);
             >);
String:<set>(<subString>,<
             -> sp0;
             -> sp1;
             -> sPs;
             sp0 <- params:<0>;
             sp1 <- params:<1>;
             sPs <- nil:<new>();
             sPs:<set>(<super>,Object);
             sPs:<set>(<0>,sp0);
             sPs:<set>(<1>,sp1);
             return <- Extern:<invoke>(<String>,<subString>,sender,sPs);
             >);
```

12.2.7 Creación de Abstracciones e Instancias

El concepto de abstracción, en un lenguaje de programación orientado a objetos basado en clases, viene designado por las clases [Booch94]. Las instancias u objetos de éstas son entidades concretas, con igual estructura comportamiento definidos por su clase.

En lenguajes orientados a objetos basados en prototipos, no existe el concepto de clase ni el concepto de instancia de ésta; sólo existe el concepto de objeto. La computación o comportamiento genérico de un grupo de objetos, podemos identificarlo en objetos de rasgo (*trait*); las instancias o datos concretos, en copias –creadas mediante reflectividad estructural– de prototipos creados a priori. De esta forma, tendremos la misma expresividad que el modelo de clases, pero con tan sólo la abstracción del objeto [Evins94].

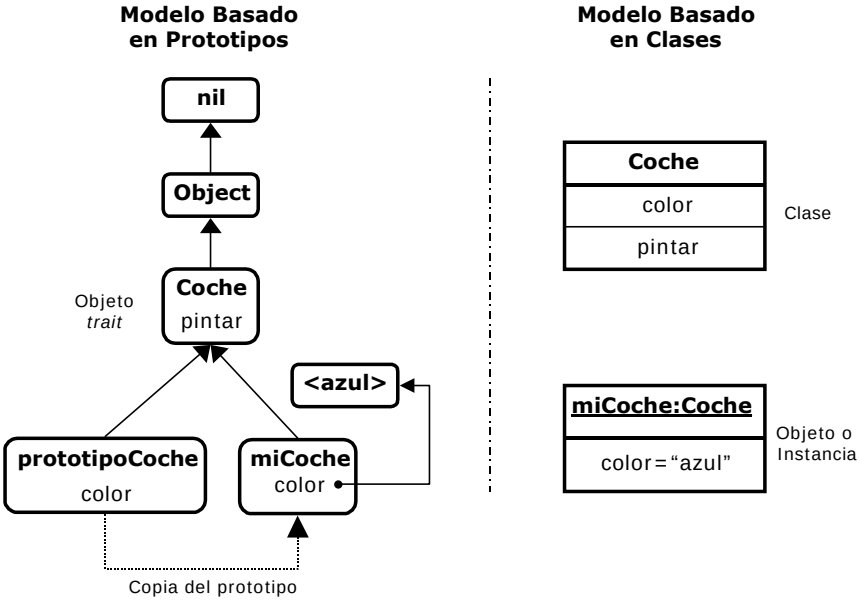


Figura 12.8: Modelo basado en clases frente al basado en prototipos.

La delegación es fundamental para especificar comportamientos y estructuras comunes entre objetos: si, al recibir un mensaje, un objeto no posee un miembro con igual nombre que el mensaje, delega la ejecución de este método en su objeto padre; si éste último especifica dicho comportamiento (objeto *trait*), estará definiendo el funcionamiento genérico de todos sus objetos derivados.

Siguiendo con este diseño de identificación de abstracciones y entidades, crearemos funcionalidades para ayudar al programador a realizar estas operaciones de forma automática, extendiendo así la abstracción de la plataforma.

12.2.7.1 Creación de Abstracciones

Para englobar el comportamiento y estructura de un conjunto de objetos, vamos a utilizar el concepto de objeto *trait* en el modelo de prototipos, de forma similar a la creación de clases en los modelos de objetos clásicos.

La creación de este tipo de objetos deberá realizarse de forma automática, mediante la invocación a un método. Para ello, implementaremos en `Object` un miembro `newChild`, que nos permita crear un descendiente directo del objeto utilizado como parámetro implícito.

```
Object:<set>(<newChild>,<
             -> newTrait;
             newTrait <- sender:<new>();
             newTrait:<set>(<super>,sender);
             return <- newTrait;
             >);
```

Si queremos crear un objeto *trait* `Figura` que derive de `Object`, podremos hacerlo mediante la utilización de este miembro. Volviendo a evaluar el mismo miembro, pero utilizando como parámetro implícito el objeto `Figura`, podremos crear un objeto derivado de este último.

```
figuraTrait <- Object:<newChild>();
elipseTrait <- figuraTrait:<newChild>();
```

En ocasiones, necesitaremos crear objetos de comportamiento y asignarle un identificador determinado. El miembro `newChildWithId` de `Object` implementa estos dos pasos en una sola abstracción.

```
Object:<set>(<newChildWithId>,<
            -> newTrait;
            -> ID;
            ID <- params:<0>;
            newTrait <- sender:<newChild>();
            newTrait:<set>(<id>,ID);
            return <- newTrait;
            >);
```

Ahora podremos hacer:

```
figuraTrait <- Object:<newChildWithId>(<{Figura}>);
elipseTrait <- figuraTrait:<newChildWithId>(<{Elipse}>);
```

La instanciación producida en los lenguajes basados en clases supone crear zonas de memoria, en las que una entidad concreta (objeto), de una representación genérica (clase), pueda almacenar sus propios datos.

En el modelo de prototipos, el proceso de instanciación puede traducirse a una copia de prototipos. En nuestro diseño de la imagen de la plataforma, todo objeto de rasgo poseerá una referencia a un objeto-instancia prototípico, que albergará los datos característicos de cualquier entidad definida por la abstracción representada: su estructura [Booch94].

Cuando queramos crear un objeto, copiaremos el prototipo y modificaremos los valores de los miembros de la copia, representando el estado de la nueva entidad. De este modo, los objetos *trait* expresan, no sólo agrupación de comportamientos, sino también estructuras comunes –al igual que las clases.

Puesto que la plataforma no separa los objetos *trait* de los representativos de entidades, debemos ser nosotros, en la creación de la extensión del lenguaje de la máquina, los que ofrezcamos esta distinción. Así, un objeto de rasgo debe poseer una referencia a su prototipo, ofreciendo un método de instanciación automático. Haciendo uso de la reflectividad estructural ofrecida por la máquina, implementaremos un miembro `newPrototype` del objeto `Object`, que permita crear prototipos, si no existiesen previamente, de un objeto de comportamiento.

```
Object:<set>(<newPrototype>,<
      -> bAlreadyExists;
      -> objectNewPrototype_result;
      -> objectNewPrototype_oTrait;
      bAlreadyExists <- sender:<has>(<prototype>);
      objectNewPrototype_oTrait <- sender;
      bAlreadyExists:<ifElse>(<
            -> oPrototype;
            oPrototype <- objectNewPrototype_oTrait:<newChild>();
            objectNewPrototype_oTrait:<set>(<prototype>,oPrototype);
            objectNewPrototype_result <- oPrototype; >
            ,<
```

```
            objectNewPrototype_result <-
                        objectNewPrototype_oTrait:<prototype>;
            >);
        return <- objectNewPrototype_result;
        >);
```

Pasándole a un objeto *trait* el mensaje `newPrototype`, se le asignará un miembro `prototype` que apunte a un nuevo objeto derivado de éste, devolviéndolo como resultado de la invocación.

12.2.7.2 Copia de Prototipos

Hemos desarrollado código para la creación automática de objetos derivados y de objetos prototipo. El siguiente paso es crear un mecanismo para la copia de prototipos de objetos *trait*. Para ello, analizaremos el siguiente ejemplo de herencia simple en dos niveles, tanto para el modelo basado en clases, como para su traducción a nuestro diseño.

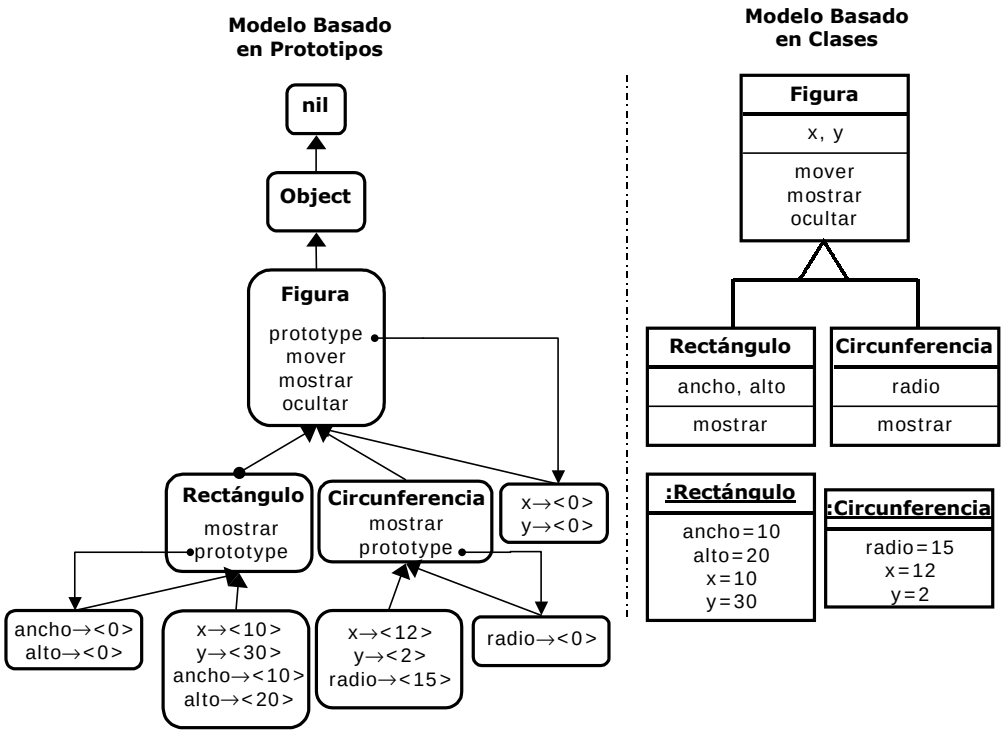


Figura 12.9: Abstracción e instanciación en modelos basados en clases y prototipos.

Como se muestra en la figura, la creación de una instancia no se limita simplemente a la copia de un objeto –aunque utilicemos la expresión "copia de prototipos". Al crear una instancia de una clase que posee otras clases heredadas, la representación en memoria del objeto creado constituye la unión de todas las estructuras representadas ascendentemente en el árbol. De esta forma, la creación de una instancia de `Rectángulo` supone crear una zona de memoria para albergar sus propiedades `alto` y `ancho` –propias del rectángulo –, así como las propiedades `x` e `y` –referentes a su condición de `Figura`.

De forma paralela, la construcción de objetos a raíz de la copia de prototipos, deberá llevarse a cabo de igual modo que su representación en el modelo de clases. El objeto `Rectángulo` posee un prototipo con las propiedades `ancho` y `alto`. La creación de un objeto `rectángulo` no se reduce a copiar este prototipo, sino que se deben añadir las propiedades de los objetos prototipos heredados –en nuestro caso los de `Figura` y `Object`, si los hubiere.

Antes de implementar el algoritmo de instanciación, definiremos la forma en la que vamos a copiar cada miembro de un objeto prototipo. Cada objeto definirá polimórficamente cómo se han de realizar sus copias, mediante la implementación del miembro `clone`.

Los objetos primitivos de tipo `String`, al ser constantes, implementan sus clonaciones como la devolución del propio objeto. Esto mismo sucede en el caso de los objetos constantes lógicos `true` y `false`.

```
String:<set>(<clone>,< return <- sender; >);
Boolean:<set>(<clone>,< return <- sender; >);
```

La clonación de cualquier objeto distinto de `String` consistirá en un algoritmo recursivo que se desglosa en los siguientes pasos:

- Creación de un objeto derivado del mismo objeto base del objeto a copiar.
- Creación de tantos miembros como miembros tiene el objeto a clonar.
- Cada miembro creado será la clonación de los miembros originales, excluyendo `id` y `super`.

```
Object:<set>(<clone>,<
      -> objectClone_oNewObject;
      -> oSenderSuper;
      -> sCode;
      oSenderSuper <- sender:<super>;
      objectClone_oNewObject <- sender:<new>();
      objectClone_oNewObject:<set>(<super>,oSenderSuper);
      sCode <- <
            -> sKey;
            -> oContent;
            -> bIsContentNil;
            -> oNewContent;
            sKey <- params:<0>;
            oContent <- params:<1>;
            bIsContentNil <- oContent:<==>(nil);
            bIsContentNil:<ifFalse>(< oNewContent <-
                                    oContent:<clone>(); >);
            objectClone_oNewObject:<set>(sKey,oNewContent);
            >;
      sender:<forEach>(sCode,nil);
      return <- objectClone_oNewObject);
      >);
```

La implementación de la instanciación de un objeto a través de la copia de prototipos se realizará mediante el miembro `copyPrototypes` del objeto `Object`, siguiendo los siguientes pasos:

- Creación de un objeto hermano (de igual padre) que el objeto prototipo del *trait* utilizado como parámetro implícito.
- Iterar con todos los prototipos partiendo del prototipo actual, en sentido ascendente, hasta el objeto `nil`, haciendo:
  1. Tomar el prototipo, si lo tiene definido, del objeto que se está analizando.
  2. Añadir al objeto creado una clonación de cada miembro del prototipo analizado, mediante la utilización de los métodos `forEach` y `clone`.

```
Object:<set>(<copyPrototypes>,<
      -> objectCopyPrototypes_oTrait;
```

```
	-> objectCopyPrototypes_oNewObject;
	objectCopyPrototypes_oNewObject <- sender:<newChild>();
	objectCopyPrototypes_oTrait <- sender;
	<return <- objectCopyPrototypes_oTrait:<!=>nil;>:<while>(<
		-> bHasPrototype;
		bHasPrototype <- objectCopyPrototypes_oTrait:<has>(
							<prototype>);
		bHasPrototype:<ifTrue>(<
			-> sCode;
			-> oPrototype;
			oPrototype <- objectCopyPrototypes_oTrait:
							<prototype>;
			sCode <- <
				-> sKey;
				-> objectCopyPrototypes_oContent;
				-> objectCopyPrototypes_oClone;
				-> bIsNotContentNil;
				sKey <- params:<0>;
				objectCopyPrototypes_oContent <- params:<1>;
				bIsNotContentNil <-
					objectCopyPrototypes_oContent:
								<!=>(nil);
				bIsNotContentNil:<ifTrue>(<
					objectCopyPrototypes_oClone <-
						objectCopyPrototypes_oContent:
								<clone>();>);
				objectCopyPrototypes_oNewObject:<set>(sKey,
					objectCopyPrototypes_oClone);
				>;
			oPrototype:<forEach>(sCode,nil);
			>);
		objectCopyPrototypes_oTrait <-
				objectCopyPrototypes_oTrait:<super>;
		>, nil);
	return <- objectCopyPrototypes_oNewObject;
	>);
```

12.2.8 Implementaciones Externas Requeridas

La funcionalidad ofrecida por la plataforma hasta este momento –definición de una lógica y sus operaciones (§ 12.2.3.3), comparación de objetos (§12.2.5), iteración (§12.2.3.4), tratamiento genérico de miembros (§12.2.3.5), aritmética numérica real y entera (§12.2.4), tratamiento de cadenas de caracteres (§12.2.6), creación de abstracciones (§12.2.7.1) e instanciación (§12.2.7.2)– ha sido desarrollada mediante su propio lenguaje de programación, extendiendo su nivel de abstracción. El conjunto de primitivas operacionales utilizadas externamente ha sido el siguiente:

Objeto	Miembro	Parámetros	Descripción
`String`	`==`	1: Objeto String	Compara dos objetos String y devuelve una referencia a un valor lógico. Esta operación posee semántica numérica y léxica.
`String`	`>`	1: Objeto String	Compara dos objetos String y devuelve una referencia a un valor lógico. Esta operación posee semántica numérica y léxica.
`String`	`+`	1: Objeto String	Realiza la operación suma de forma polimórfica: Semántica numérica y léxica si no representa números.

Objeto	Miembro	Parámetros	Descripción
`String`	`-`	1: Objeto String	Realiza la operación diferencia binaria de forma polimórfica: Semántica numérica en cualquier formato.
`String`	`/`	1: Objeto String	Realiza la operación división de forma polimórfica: Semántica numérica en cualquier formato.
`String`	`*`	1: Objeto String	Realiza la operación producto de forma polimórfica: Semántica numérica en cualquier formato.
`String`	`%`	1: Objeto String	Realiza la operación módulo de forma polimórfica: Semántica numérica tan solo para enteros.
`String`	`append`	1: Objeto String	Devuelve el resultado de concatenar el parámetro implícito y el parámetro pasado.
`String`	`length`	-	Devuelve la longitud en caracteres del parámetro implícito.
`String`	`subString`	1: Entero 2: Entero	Devuelve un objeto cadena de caracteres resultado de tomar el segundo parámetro de caracteres a partir del carácter con posición primer parámetro.

12.3 Implantación y Acceso a la Plataforma

La máquina abstracta del entorno de computación será implementada como un proceso dentro de un determinado sistema operativo[61]. La máquina deberá ser capaz de interactuar con otras aplicaciones del sistema operativo, así como con otras máquinas abstractas distribuidas a lo largo de una red de computadores.

La implantación de la máquina abstracta ha de proporcionar al usuario la ilusión de poseer otro microprocesador, capaz de computar código de esta plataforma, y que pueda interactuar a su vez con el sistema operativo existente. De este modo, la una aplicación de nuestra plataforma podrá intercomunicarse con otra aplicación –o con una vista gráfica de ésta– dentro del sistema operativo en el que se haya sido hospedada.

Sobre cada plataforma real, se elegirá un protocolo de interconexión de procesos, para poder abrir el proceso de la máquina abstracta a otras aplicaciones del sistema operativo. En cada sistema operativo podrá existir un protocolo de interconexión distinto –como COM, CORBA o *Sockets* TCP/IP. Un ejemplo de una aplicación nativa al operativo, puede ser un editor e intérprete de programación en el lenguaje de la máquina abstracta, que envíe las instrucciones a la plataforma, y los resultados de la interpretación sean visualizados gráficamente.

[61] Su implementación física también podrá llevarse a cabo, al igual que la plataforma Java ha implementado su picoJava [Sun97].

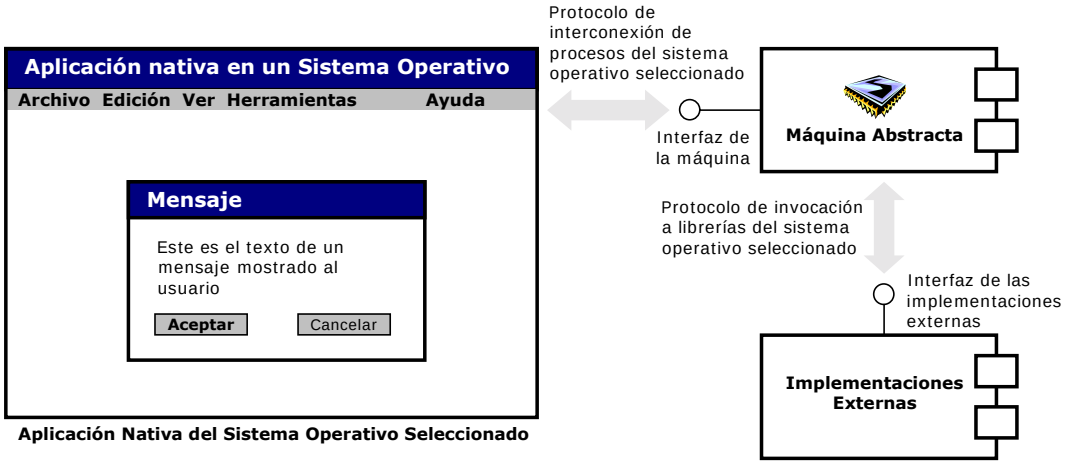


Figura 12.10: Acceso a la máquina abstracta desde aplicaciones del sistema operativo seleccionado.

En la figura anterior se muestra cómo una aplicación puede acceder a la plataforma, utilizando un mecanismo estándar de interconexión de procesos dentro del sistema operativo seleccionado en la implantación. Para computar instrucciones por la máquina, puede requerirse el acceso a implementaciones externas y, para acceder a éstas, la plataforma utilizará un mecanismo estándar de invocación a librerías extensibles.

Puesto que la interconexión entre las aplicaciones y la máquina abstracta será dependiente del sistema operativo utilizado, no podremos transferir entre ellos objetos de nuestra plataforma. Así, las primitivas de intercomunicación utilizaran únicamente cadenas de caracteres, eliminando el establecimiento previo de representación de la información[62].

La interfaz de llamadas que la máquina abstracta ofrece al sistema operativo utilizado es la siguiente:

1. `newReference`. Como único parámetro recibe el nombre de la referencia que se desea crear en el contexto inicial. Esta referencia se eliminará (finalizará su contexto) al finalizar la ejecución de la máquina abstracta.
2. `getMember`. Recibe como parámetros el identificador del objeto implícito y el nombre del miembro a acceder. Devuelve el identificador del miembro seleccionado.
3. `evaluateMemberSync`. Evalúa un miembro de forma síncrona. Recibe como parámetros el identificador del objeto implícito, el identificador del objeto a partir del cual se realiza la búsqueda del miembro (para encontrar a `mySelf`), el nombre del miembro, y los identificadores de los parámetros[63].

   La especificación del objeto a raíz del cual se va a realizar la búsqueda del miembro no tiene mucho sentido en la interfaz local de usuario –siempre coincidirán. Sin embargo, la separación de la referencia `sender` y `mySelf` es necesaria en la invocación entre máquinas abstractas distribuidas –en concreto, en el caso en el que el objeto *trait* se halle en una máquina distinta al referenciado por `sender`.

[62] Realmente, las cadenas de caracteres van a representar los identificadores únicos de los objetos utilizados por la máquina abstracta.
[63] Puesto que no podemos suponer que la interconexión entre procesos posea sobrecarga, los identificadores de los parámetros se pasarán como un único argumento cadena de caracteres, separando cada elemento con el carácter coma.

La devolución del método es el identificador del objeto referenciado por `return`, tras la ejecución del método.

4. `evaluateMemberAsync`. Evalúa un miembro de forma asíncrona. Los parámetros son los mismos que en la evaluación síncrona, salvo que no se obtiene un resultado.
5. `evaluateStringSync`. Evalúa un objeto cadena de caracteres de forma síncrona en el contexto cero. Recibe como parámetros el objeto y una cadena de caracteres con todos los identificadores de los parámetros. Devuelve el identificador del objeto devuelto.
6. `evaluateStringAsync`. Evalúa un objeto cadena de caracteres de forma asíncrona. Recibe los mismos parámetros que la evaluación síncrona y no devuelve ningún valor.

La interfaz de llamadas será siempre el mismo; el mecanismo de intercomunicación entre el proceso de la aplicación y el proceso de la máquina abstracta, será dependiente del sistema operativo utilizado.

CAPÍTULO 13:
DISEÑO DEL ENTORNO DE PROGRAMACIÓN

Una vez descrita la máquina abstracta, presentaremos la estructura global del desarrollo de funcionalidades del entorno de programación desarrollado, sin entrar en un nivel elevado de detalle[64], sirviendo como demostración de la extensibilidad y adaptabilidad ofrecidas por la plataforma.

Basándonos en la arquitectura presentada para el sistema (capítulo 9), el entorno de programación supone una ampliación de las funcionalidades computacionales ofrecidas inicialmente por la plataforma, utilizando para ello su propio lenguaje, sin necesidad de modificar la implementación de la máquina virtual.

Como ejemplo demostrativo de la flexibilidad de la plataforma diseñada, mostraremos cómo llevar a cabo la implementación de sistemas de persistencia, distribución, planificación de hilos y recolección de basura; todas ellas desarrolladas sobre la propia plataforma y diseñadas de forma adaptable –de forma que puedan ser introducidos nuevos mecanismos dinámicamente, sin necesidad de modificar la aplicación.

13.1 Recolección de Basura

Haciendo uso de:

- La introspección de la plataforma para descubrir sus objetos existentes en tiempo de ejecución, mediante la utilización del objeto `System`.
- La reflectividad estructural ofrecida por todo objeto para crear, modificar y conocer todos sus miembros.
- La introspección ofrecida en el conocimiento del número de referencias que apuntan dinámicamente a un objeto –miembro primitivo `getRefCount` del objeto `nil`.
- La primitiva de destrucción de objetos en superficie `delete`.

[64] Por motivos de espacio, reduciremos la presentación del entorno de computación a análisis y diseño, sin mostrar toda su codificación. Para acceder a ella, consúltese [Díez2001].

Se diseñará un sistema de recolección de basura que sea capaz de eliminar automáticamente los objetos no referenciados de nuestro sistema. Éste será codificado en el propio lenguaje de la máquina, demostrando su faceta extensible.

El diseño será adaptable a distintos algoritmos de recolección, así como a distintas políticas de invocación a éste. La configuración de ambos parámetros podrá llevarse a cabo dinámicamente, sin necesidad de finalizar la ejecución de ninguna aplicación –adaptabilidad.

13.1.1 Diseño General del Sistema

Mediante la extensibilidad ofrecida por la máquina abstracta, desarrollaremos en su propio lenguaje un sistema de recolección de basura, genérico respecto al algoritmo y a la política de ejecución del limpiado de objetos. El conjunto de objetos utilizados, se muestra en la siguiente figura:

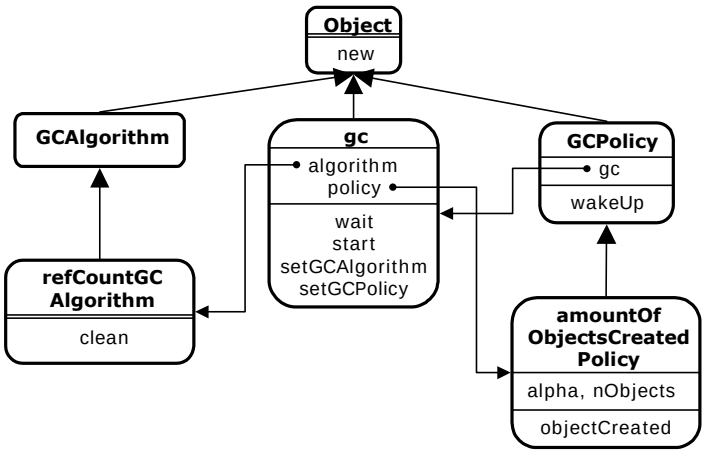


Figura 13.1: Objetos utilizados en el diseño del recolector de basura.

El objeto central es `gc`, recolector de basura (*Garbage Collector*). Éste posee una referencia miembro a su algoritmo de limpiado, `algorithm`, y a su política de activación, `policy` –cuándo salta el algoritmo de eliminación de objetos, como un nuevo hilo. Ambos parámetros son adaptables dinámicamente, siguiendo el patrón de diseño *Template Method* [GOF94].

Todo algoritmo de recolección deberá ser objeto derivado de `GCAlgorithm` e implementar únicamente un miembro evaluable `clean`. Cualquier política a utilizar activará la eliminación de objetos como un nuevo hilo del sistema –método `wakeUp`. Un objeto derivado de `GCPolicy`, simplemente deberá invocar a este método cuando surja el evento apropiado.

13.1.2 Recolección Genérica

Mediante la utilización del patrón *Template Method*, el sistema implementa el método `start`, derogando el limpiado de objetos en el método `clean` de su miembro `algorithm`. El método `clean` utiliza el propio objeto `gc` como monitor. El resultado es un bucle infinito que espera la apertura del monitor, para ejecutar el método `clean` de su asociado `algorithm`, y volver a dormirse.

Para la ejecución del recolector de basura, se ha de seleccionar la política y el algoritmo a utilizar, y se deja el sistema dormido a la espera de evaluar nuevas instrucciones

desde el sistema operativo[65] –invocando a su método `start`. Estos dos parámetros son totalmente adaptables en tiempo de ejecución, mediante la utilización de los métodos `setGCAlgorithm` y `setGCPolicy`.

```
gc:<setGCAlgorithm>(refCountGC);
gc:<setPolicy>(amountOfObjectsCreatedPolicy);
gc:<start>();
```

13.1.3 Algoritmos de Recolección

Como hemos visto en el punto anterior, el algoritmo genérico de activación del recolector de basura espera a que se active su invocación, para evaluar el miembro `clean` del algoritmo seleccionado. Cualquier objeto derivado de `GCAlgorithm` que implemente este método, podrá ser utilizado como recolector de basura. Su adaptación dinámica será llevada a cabo a través del método `setGCAlgorithm` del objeto `gc`. La implementación de este método deberá utilizar su objeto padre como monitor para sincronizarse con `setGCAlgorithm`, eliminando la posibilidad de modificar éste en plena ejecución del algoritmo.

Un algoritmo de ejemplo (`refCountGCAlgoritm`) puede ser aquél que recorre todos los objetos del sistema y, tras invocar a su método `getRefCount`, elimina aquéllos que posean únicamente una referencia a ellos –esta única referencia es la que posee todo objeto, como miembro del objeto `System`.

La existencia de referencias cíclicas entre objetos inaccesibles –y que por tanto deban eliminarse– puede detectarse haciendo uso de la reflectividad estructural. A partir de un objeto, podemos analizar todos sus miembros de forma recursiva, para detectar posibles ciclos.

Este algoritmo es un ejemplo del desarrollo de un recolector de basura en su propio lenguaje, sirviendo como ejemplo demostrativo de la extensibilidad del sistema. Otros algoritmos más sofisticados pueden implementarse y registrarse mediante `setGCAlgorithm`, demostrando así su adaptabilidad.

13.1.4 Políticas de Activación

El diseño del sistema se basa en activar la limpieza de objetos cuando se produzca un determinado evento. Tanto el algoritmo de recolección como el modo (política) de generar estos eventos, son configurables en el sistema. Cada invocación al método `wakeUp` –sincronizado con `setGCPolicy`, utilizando el monitor `GCPolicy`– provoca la ejecución del algoritmo de recolección –abriendo el monitor `gc`.

Una implementación de política de invocaciones al recolector de basura, puede ser la superación de un umbral de creación de objetos –`alpha`. Cada vez que se cree un objeto, se incrementa un contador y, si éste supera el umbral, se invoca al recolector de basura (`wakeUp`).

La implementación de esta política es desarrollada en el objeto `amountOfObjectsCreatedPolicy`. Para controlar la creación de objetos, derogamos el método `new` de `nil`, ubicando una nueva implementación en `Object`: cada vez que se

[65] Si se desea ejecutar más instrucciones de la imagen, la invocación al método `start` puede realizarse de forma asíncrona.

ejecute éste, se crea el objeto y se invoca al método `objectCreated` de `amountOfObjectsCreatedPolicy`.

13.2 Planificación de Hilos

La plataforma ha sido diseñada con primitivas de multitarea. La creación de hilos se puede realizar mediante la invocación asíncrona de métodos. La sincronización de los distintos hilos se puede obtener mediante el uso de los objetos como si fuesen monitores: los miembros primitivos `enter` y `exit` del objeto `nil`, sincronizan el hilo activo en función del recurso identificado por el objeto implícito. Las instrucciones primitivas de la plataforma han de ser consideradas de ejecución atómica.

Al igual que la máquina virtual de Java [Sun95], la especificación de nuestra plataforma ofrece un sistema computacional multihilo, pero deja el modo en el que se realice su planificación como dependiente de la implementación de la máquina virtual. La máquina abstracta no especifica cómo han de ejecutarse sus distintos hilos.

Siguiendo con los criterios de diseño de nuestro sistema, el planificador de hilos deberá construirse sobre su propio lenguaje (extensibilidad), y podrá adaptarse dinámicamente a las necesidades del usuario (adaptabilidad). Construiremos, pues, una jerarquía de objetos, cuyos servicios nos permitan ofrecer al usuario un sistema de control flexible del planificador de hilos del entorno de computación. Se podrá implementar cualquier tipo de planificador, se podrá seleccionar uno en tiempo de ejecución y modificar posteriormente, así como añadir planificadores nuevos.

13.2.1 Creación de Métodos

La forma de evaluar código primitivamente en nuestra plataforma es llevada a cabo por la máquina en la evaluación de objetos miembros o de cadenas de caracteres. La evaluación síncrona de este código se produce creando un nuevo contexto en el hilo activo; la evaluación asíncrona crea un contexto para un hilo nuevo.

La planificación de los hilos creados por invocaciones asíncronas no puede ser controlada, puesto que queda libre a la implementación de la máquina virtual. Para modificar este funcionamiento:

- Crearemos la noción del concepto de método. Un método será un objeto miembro que podrá evaluarse tanto síncrona, como asíncronamente. El modo en el que podemos definir cómo se evalúa un objeto –distinto a una cadena de caracteres– es gracias una primitiva de reflectividad computacional ofrecida por la máquina abstracta.

- Adición de código intruso de sincronización. A la cadena de caracteres que define el código de un método, se le añadirá, manipulando éste como si fuese información (cosificación), código adicional encargado de sincronizar su ejecución en función del planificador seleccionado.

Un método será un objeto derivado de un *trait* `Method`:

```
        -> Method;
        Method <- Object:<newChildWithId>(<Method>);
```

El objeto `Method` será el objeto base de todos los objetos miembro evaluables y planificables del entorno de computación. Cada objeto derivado de él, identificará un mé-

todo de un objeto. Cada uno de ellos –y por lo tanto su prototipo– deberá tener un miembro `code` que identifique el código del método[66]:

```
      <      -> prototype;
             prototype <- Method:<newPrototype>();
             prototype:<set>(<code>,nil);     >();
```

La creación de un objeto método se realizará a partir del miembro `newMethod` del objeto `Object`. Cuando una clase desee crear un método, invocará a su método `newMethod` (heredado de `Object`) para que añada el código planificable. Por ejemplo:

```
Circunferencia:<newMethod>(<perímetro>,<
             -> radio;
             radio <- sender:<radio>;
             return <- <2>;
             return <- return:<*>(<3.141592>);
             return <- return:<*>(radio);
             >);
```

La implementación del método `newMethod`, crea un objeto derivado de `Method` mediante la copia de prototipos, lo añade como miembro del objeto implícito, e inserta el código intruso de sincronización a su miembro `code` –mediante el método `setCode`.

```
Object:<set>(<newMethod>,<
             -> sName;
             -> sCode;
             -> method;
             sName <- params:<0>;
             sCode <- params:<1>;
             method <- Method:<copyPrototypes>();
             sender:<set>(sName,method);
             method:<setCode>(sCode);
             >);
```

13.2.2 Introducción de Código Intruso de Sincronización

La inclusión del código de sincronización en la cadena de caracteres referenciada por `code`, es llevada a cabo por la implementación del método `setCode`. Entre cada una de sus instrucciones se inserta un punto de espera de ejecución, mediante la utilización de un monitor: `_execMonitor`. La apertura de este monitor, para que pueda ser ejecutada una instrucción de código, es llevada a cabo mediante otro monitor: `_controlMonitor`.

Inicialmente se hace que los dos monitores estén ocupados:

```
             _controlMonitor:<enter>();
             _execMonitor:<enter>();
```

Antes de cada instrucción, se libera el monitor de control y se espera por el de ejecución. Esto hace que la ejecución de la instrucción quede a expensas de abrir el monitor de ejecución. Así, las dos líneas a anteponer a cada instrucción son:

```
             _controlMonitor:<exit>();
             _execMonitor:<enter>();
```

Para cada hilo de ejecución de un método, habrá otro hilo de control encargado de abrir el monitor de ejecución, para que se ejecute una única instrucción del método. Así,

[66] El código mostrado ha sido evaluado como una cadena de caracteres para no crear la referencia `prototype` en el contexto global.

este hilo de control ejecuta el siguiente código para que se pueda evaluar una instrucción del método:

```
_controlMonitor:<enter>();
_execMonitor:<exit>();
_controlMonitor:<enter>();
```

Inicialmente ocupa el monitor de control; posteriormente, libera el monitor de ejecución, permitiendo la evaluación de una instrucción; su finalización bloquea el hilo, a la espera de que se nos notifique la finalización de la ejecución de la instrucción.

El resultado, como se muestra en la Figura 13.2, es que:

- La ejecución de una instrucción espera a que se abra el monitor de control.
- Una vez ejecutada la instrucción del método, tras haber abierto `_execMonitor` por el hilo de control, se abre el monitor de control indicando que la sentencia ha sido evaluada.

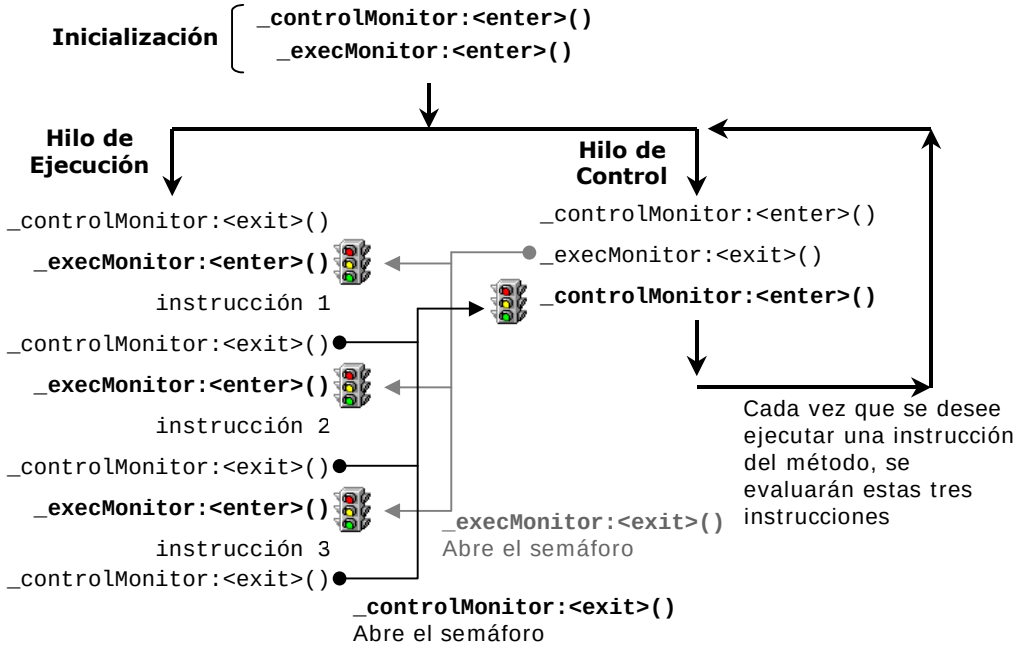


Figura 13.2: Inclusión de código intruso para la sincronización de hilos.

Tras la última instrucción de un método, simplemente se abre `_controlMonitor`, para liberar su hilo de control asociado.

13.2.3 MOP para la Evaluación de Código

La máquina abstracta, mediante reflectividad computacional basada en un MOP, ofrece la posibilidad de modificar el comportamiento de la evaluación de objetos. Al evaluar síncrona o asíncronamente un objeto miembro o un objeto cadena de caracteres, la semántica de la evaluación puede ser modificada.

El protocolo de modificación de la semántica (MOP), se basa en definir un miembro "`( )`" o ")(" del objeto a evaluar[67]. En el momento en que un objeto vaya a ser evalua-

[67] El objeto ha de poseer estos miembros si quiere modificar la semántica de evaluación. No es suficiente si posee dicho miembro uno de sus objetos padre.

do, si posee un miembro "`()`" o ")(", se evaluarán éstos en lugar del propio objeto –"()" si la invocación es síncrona, y ")(" si es asíncrona.

De este modo, todo objeto método –descendiente de `Method`– poseerá un miembro "()" y ")(" que evaluará el miembro `code` de un modo sincronizado. Para el ejemplo del método `perímetro` de `Circunferencia`, tendríamos la siguiente estructura:

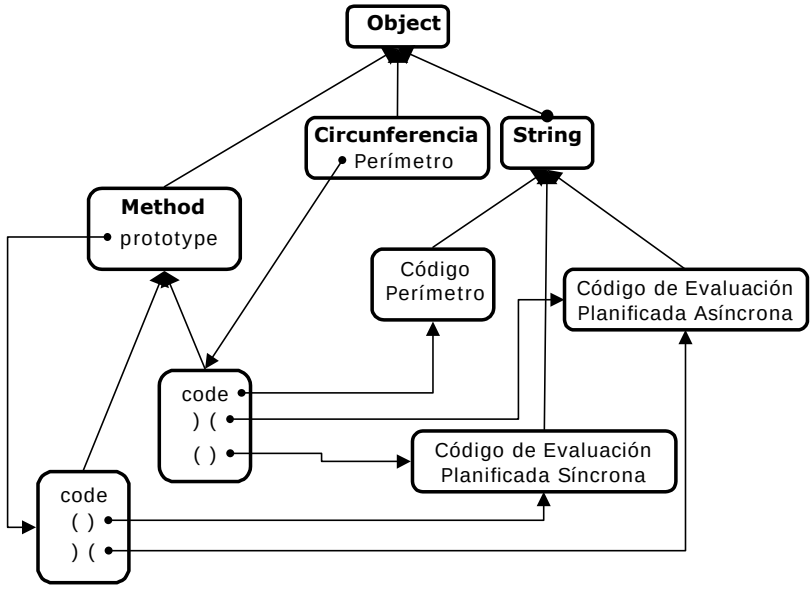


Figura 13.3: Representación de un método planificable.

La inclusión de los miembros "()" y ")(" en el prototipo de `Method`, hace que, en la creación de cualquier método, se derogue la evaluación de su código en las rutinas de evaluación planificada síncrona y asíncrona.

13.2.4 Creación de Hilos

La modificación de la semántica de la invocación asíncrona, supone la creación de un nuevo hilo planificable por el usuario. Cada hilo creado en la invocación asíncrona de un método, será representado por un objeto derivado del nuevo objeto *trait* `Thread`. Como se muestra en la Figura 13.4, cada objeto clonado del prototipo de éste posee:

- Un objeto monitor para controlar la finalización de la ejecución de una sentencia relativa a su hilo: `controlMonitor`.
- Un monitor para permitir ejecutar una nueva instrucción del hilo: `execMonitor`.
- Un valor lógico que le diga si ha finalizado la ejecución del hilo: `isThreadFinished`. Inicialmente tendrá un valor falso.
- Un objeto cadena de caracteres con semántica numérica (`number`) que va a ser su identificador de hilo en ejecución.

El objeto *trait* `Thread` implementará un método `evaluate`, cuya funcionalidad será la ejecución de las instrucciones asociadas al hilo que representa. En concreto, ejecutará las tres instrucciones del hilo de control, mostradas en la parte derecha de la Figura 13.2.

13.2.5 Planificación Genérica

El conjunto de objetos utilizados en el diseño de planificación genérica es el siguiente:

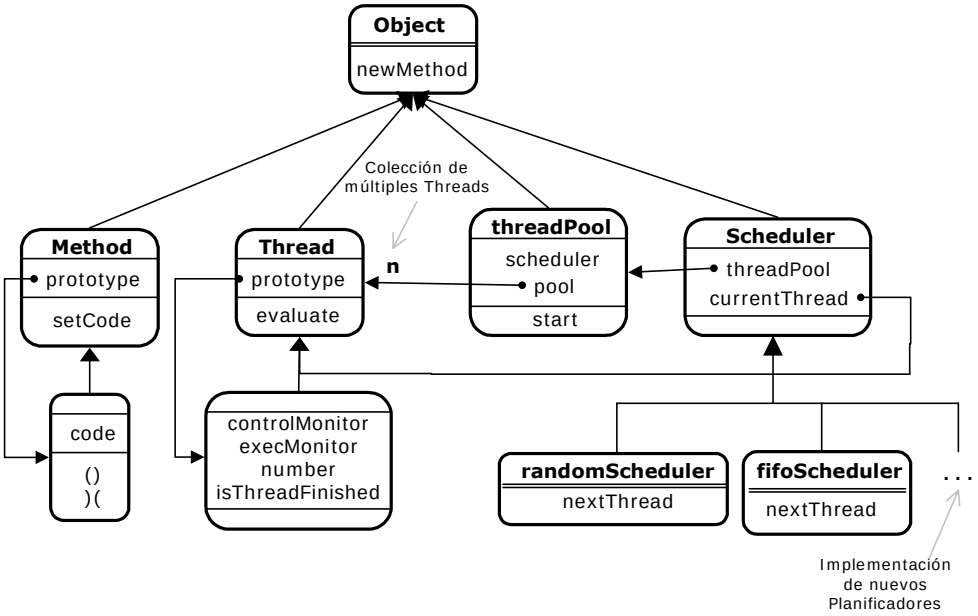


Figura 13.4: Diseño general del sistema genérico de planificación de hilos.

El conjunto global de hilos está en la abstracción `threadPool`: su miembro `pool` es un contenedor de todos los hilos planificables existentes en la máquina. La planificación genérica de hilos comienza cuando se invoca asíncronamente a su miembro `start`. Esta ejecución va tomando los distintos hilos del sistema en función del planificador seleccionado, y ejecuta una instrucción de cada uno de ellos –invocación del método `evaluate` de cada `Thread`.

La referencia al planificador seleccionado, adaptable dinámicamente, es aquélla asociada al miembro `scheduler` de `threadPool`. La modificación dinámica de este miembro, supone la adaptación instantánea al planificador seleccionado.

El diseño de la planificación genérica se ha llevado a cabo siguiendo el patrón de diseño “Método Plantilla” (*Template Method*) [GOF94]. El objeto genérico de planificación, `Scheduler`, define el algoritmo general de planificación, y cada objeto derivado, representante de un planificador determinado, implementa su funcionalidad específica a través del método `nextThread`. Este método ha de devolver, a raíz de conocer el hilo actual de ejecución –miembro `currentThread` de `Scheduler`–, el siguiente hilo a evaluar, siguiendo su política de planificación específica.

Ejemplos de implementaciones de planificadores genéricos son `fifoScheduler` y `randomScheduler`. El usuario podrá añadir cualquier otro, derivado de `Scheduler`, con tan sólo implementar el método `nextThread`. Una vez hecho esto, sin necesidad de haber parado la ejecución del sistema, podrá seleccionarse el nuevo planificador mediante la evaluación de la siguiente instrucción:

```
threadPool:<set>(<scheduler>,nuevoPlanificador);
```

13.3 Sistema de Persistencia

Dentro del entorno de programación, se diseñará un sistema de persistencia, en el que tanto los datos como la computación podrán persistir[68], siendo el mecanismo, formato y política, parámetros dinámicamente adaptables por el usuario.

Como se ha estudiado en el capítulo 8, el modelo computacional orientado a objetos basado en prototipos proporciona una mayor facilidad a la hora de implementar un sistema de persistencia: eliminamos la relación estrecha existente entre un objeto y su clase, substituyéndola por una relación de delegación. La modificación, eliminación o ampliación de comportamientos, manipulando los objetos de rasgo, no lleva al modelo de objetos a estados inconsistentes.

El desarrollo del sistema de persistencia hace uso intensivo de las características reflectivas ofrecidas por la plataforma:

- Mediante introspección, un objeto conoce su estado y comportamiento, y actualiza éstos en una memoria persistente.
- Mediante reflectividad estructural, se construye un objeto a raíz de una información obtenida de un dispositivo persistente.
- Mediante reflectividad computacional –un MOP–, se modifica el acceso a los miembros de un objeto para poder implementar un objeto delegado (*proxy*): objeto que no se encuentra realmente en la memoria, pero que es capaz de delegar el acceso y obtención de sus miembros, en un sistema de persistencia[69].

13.3.1 Diseño General del Sistema

La siguiente figura muestra el conjunto de objetos utilizados en el desarrollo del sistema de persistencia:

[68] Al tratar la computación como datos, la persistencia de computación es realmente persistencia de datos –cadenas de caracteres.

[69] Este concepto será también utilizado en el sistema de distribución: un objeto delegado es la representación local de un objeto que realmente está ubicado en otra máquina.

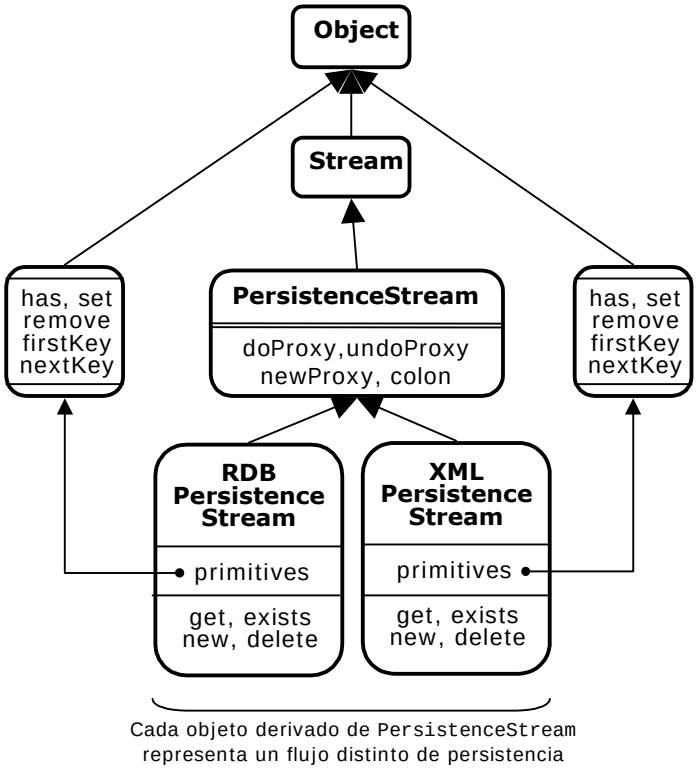


Figura 13.5: Conjunto de objetos utilizados en el diseño del sistema de persistencia.

El sistema de persistencia tiene ciertas similitudes con el de distribución (§ 13.4). En ambos casos, los objetos se obtienen de un flujo (*Stream*), ya sea de una memoria persistente, o bien de un dispositivo de comunicación. Las funcionalidades comunes a ambos sistemas han sido ubicadas como métodos del objeto *trait* `Stream`. Las funcionalidades propias de cualquier flujo de persistencia, de forma independiente al mecanismo seleccionado por el usuario, son representadas por los miembros del objeto `PersistenceStream`.

Al igual que el desarrollo del sistema de planificación genérica de hilos, el diseño del sistema de persistencia está centrado en la utilización del patrón de diseño "Método Plantilla" (*Template Method*) [GOF94]. Cada uno de los distintos sistemas de persistencia a implantar en la plataforma, son objetos derivados de `PersistenceStream`; la implementación de los miembros `get`, `exists`, `new`, `delete` y `primitive`, los hace susceptibles de ser utilizados como flujos de persistencia.

13.3.2 Implantación de un Flujo de Persistencia

Cada flujo de persistencia distinto ha de implementar un conjunto de operaciones básicas, sobre las que se crearán funcionalidades de mayor nivel de abstracción. En concreto, las primitivas básicas a desarrollar son los siguientes miembros:

- `exists`. Nos dice si un determinado objeto tiene una entrada en el sistema de persistencia, pasándole como único parámetro el identificador del objeto.
- `delete`. Elimina el espacio utilizado por un objeto en el sistema de persistencia; si tuviere un conjunto de miembros, eliminaría también las referencias a éstos.
- `new`. Tomando un parámetro representativo de un identificador de un objeto, produce la creación del espacio apropiado para albergar un objeto con dicho

`id`. Si este objeto existiere previamente, se eliminaría su espacio y todas sus referencias a sus miembros.

- `get`. Pasándole el identificador del objeto implícito y el nombre del miembro solicitado, devuelve el valor (el identificador[70]) almacenado para este par de valores.

Cuando un objeto se hace persistente, se creará un objeto delegado (*proxy*) de éste que interactúe con el sistema de persistencia. El nuevo objeto delegado no poseerá los miembros del objeto original, sino que los irá a buscar al flujo de persistencia asociado. De este modo, las funcionalidades básicas de reflectividad estructural deberán ser redefinidas. Por ejemplo, cuando preguntemos a un objeto *proxy* si un objeto persistente tiene un miembro (`has`), o queramos conocer la primera clave de sus miembros (`firstKey`), deberemos consultar esta información en el disco, en lugar de buscar en el propio objeto *proxy*.

Todas las operaciones primitivas que varíen su semántica en el uso de objetos persistentes, estarán situadas en un objeto contenedor asociado al miembro `primitives` de cada uno de los flujos de persistencia. Por cada operación primitiva, tendremos la implementación de su nueva semántica. Cuando se evalúe una de estas operaciones en un objeto *proxy*, ejecutaremos la nueva semántica.

Cada primitiva recibirá como primer parámetro el objeto implícito, o lo que es lo mismo, el objeto contenedor. El resto de los parámetros serán el conjunto de parámetros pasados a cada primitiva original. Las primitivas implementadas en cada sistema de persistencia son:

- `has`. Accede al sistema de persistencia y nos dice si el parámetro implícito posee el miembro pasado.
- `remove`. Elimina del parámetro implícito el miembro pasado.
- `set`. Asigna al parámetro implícito una referencia (una entrada en el sistema de persistencia) a un miembro pasado. Si ya existía un valor, lo sobreescribe.
- `firstKey`. Devuelve la primera clave o nombre del parámetro implícito.
- `nextKey`. Devuelve la siguiente clave o nombre del parámetro pasado asociado al parámetro implícito.

Estas primitivas estarán implementadas haciendo uso del protocolo de invocación externa (objeto `Extern`), puesto que todas dependen de la plataforma física –acceden a un sistema de persistencia. Sobre ellas, se diseña un sistema genérico de persistencia (patrón de diseño *Template Method*), haciendo que éste funcione de forma independiente al flujo específico seleccionado –adaptabilidad.

13.3.3 MOP para Acceso al Miembro

Cuando un objeto sea persistente, tendrá modificada la semántica de la primitiva de acceso a sus miembros –operador "`:`". Delegará su funcionalidad en un canal o flujo de datos (sistemas de persistencia y distribución); cada vez que queramos acceder o evaluar uno de sus miembros, la búsqueda se realizará en una base de datos o en una plataforma remota.

[70] Para cada objeto se guarda su identificador y una colección de pares formados por el nombre de cada miembro y su identificador asociado.

La modificación de la semántica de acceso a los miembros de un objeto, es llevada a cabo con la adición de un miembro "`:`" a ese objeto. Cada vez que se intente evaluar o acceder a un miembro de un objeto que tenga un método "`:`", la máquina invocará a éste en lugar de ofrecer su funcionamiento primitivo.

A la hora de implementar un miembro "`:`" que derogue la funcionalidad de acceso a los miembros, el protocolo semántico de la invocación (MOP) es descrito por las siguientes referencias –cuando se invoque a este miembro, éstas serán las referencias pasadas:

- `mySelf`: Referencia al objeto en el que se ha implementado el miembro "`:`". Será el objeto implícito o alguno de sus objetos padre. En cualquier caso, será siembre un objeto *proxy*, representante de un objeto persistente o remoto.
- `sender`: Referencia al objeto implícito.
- `return`: Referencia que apuntará al valor devuelto por el acceso a un miembro o por la invocación síncrona de un método.
- `params`: Referencia que apunta a un objeto contenedor. Cada miembro de este objeto está numerado consecutivamente a partir de cero. Sus valores son:
  - Desde 0 al número de parámetros pasados (n) menos uno, serán referencias a los parámetros pasados. En el caso de que se haya producido un acceso a un miembro (no haya evaluación), no existen estas referencias.
  - Miembro n: Es una referencia a un valor lógico. Indica si se ha producido una invocación (`true`) o un acceso (`false`).
  - Miembro n+1: Referencia al nombre del miembro –objeto cadena de caracteres.
  - Miembro n+2: Valor lógico que indica si la invocación a sido realizada síncrona (`true`) o asíncronamente (`false`).

El algoritmo de acceso a los miembros de un objeto persistente es siempre igual. Su implementación ha sido llevada a cabo de forma polimórfica respecto al de flujo de persistencia utilizado. Su código está ubicado en el miembro `colon` del objeto `PersistenceStream`, mostrado en la Figura 13.5.

13.3.4 Objetos Delegados

Como hemos comentado al principio de la presentación del sistema de persistencia, el diseño de este módulo está centrado en el concepto de objeto que delega, mediante reflectividad computacional, la funcionalidad de acceso a sus miembros en un flujo de persistencia.

El MOP ofrecido por la máquina abstracta para modificar la semántica del acceso a sus miembros, obliga a que todo objeto *proxy* posea un miembro "`:`". Este miembro será una referencia al método `colon` de `PersistenceStream`, que implementa el algoritmo de acceso a los miembros del objeto implícito sobre un flujo de persistencia; este flujo será definido por el miembro `_stream` que todo objeto *proxy* ha de tener.

Un objeto delegado ha de poseer, por tanto, además de los miembros `id` y `super`, las dos referencias:

- "`:`", que indica la nueva semántica de acceso a sus miembros, apuntando siempre al miembro `colon` de `PersistenceStream`.

- "`_stream`", que hace referencia al flujo físico seleccionado para dar persistencia al objeto real, representado por el objeto *proxy*.

Cuando queramos que un objeto pase a ser persistente –creándose un objeto *proxy*–, se almacenará su identificador único en el sistema de persistencia. Para cada uno de sus miembros, deberemos especificar si deseamos que se hagan o no persistentes. Esta determinación está en función de la semántica del problema. Normalmente se divide en relaciones de composición en las que los miembros se hacen persistentes, y en relaciones de asociación en las que el programador deberá decidir[71] [Ortín97b]. Nuestra plataforma no distingue entre tipos de asociaciones, así que debemos de proporcionar un mecanismo para discernir entre estas situaciones.

El grado de profundidad para la realización de los objetos *proxy*, será especificado por un objeto cadena de caracteres evaluable. Éste se ejecutará síncronamente para cada miembro, pasándole en cada evaluación tres parámetros: el objeto tratado, el nombre del miembro y el miembro. Si el resultado de la evaluación es `true`, el miembro se hará persistente. En caso contrario, el miembro seguirá siendo temporal.

Si queremos, por ejemplo, que todos los miembros sean persistentes, el siguiente objeto definirá la política de profundidad:

```
< return <- true; >
```

Si deseamos que todos los miembros de un objeto sean persistentes, excepto su asociado `autor`, el objeto indicativo de la política de profundidad sería:

```
<            -> miembro;
             miembro <- params:<1>;
             return <- miembro:<!=>(<autor>);       >
```

El miembro `doProxy` de `PesistenceStream` –utilizando como parámetro implícito en la invocación un flujo específico de persistencia– convierte el primer parámetro pasado a un objeto delegado, utilizando como política de profundidad el objeto cadena de caracteres pasado como segundo parámetro.

Si deseamos hacer que `miLibro` pase a ser persistente, al igual que todos sus miembros excepto `autor`, en un sistema basado en una base de datos relacional (`RDBPersistenceStream`), evaluaremos la siguiente instrucción:

```
RDBPersistenceStream:<doProxy>(miLibro,
                    <      -> miembro;
                           miembro <- params:<1>;
                           return <- miembro:<!=>(<autor>);
                    >);
```

La transformación llevada a cabo se muestra en la siguiente figura:

[71] De ahí la palabra reservada *transient* utilizada en el mecanismo de serialización de la plataforma Java.

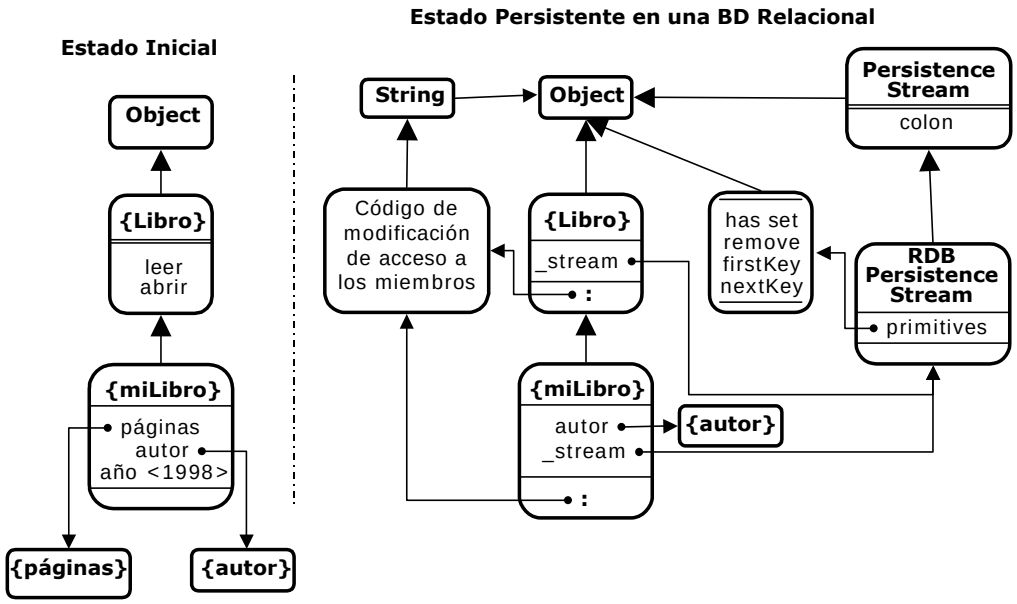


Figura 13.6: Transformación llevada a cabo al hacer persistente el objeto "miLibro".

Los objetos que pasan a ser delegados son `miLibro`, `Libro` y `páginas`. El objeto `Object` no pasa a ser persistente, porque forma parte de los objetos propios de la plataforma[72]. Si en el sistema no existiese ninguna referencia al objeto `páginas`, el recolector de basura podría eliminar éste, puesto que el objeto *proxy* `miLibro` ya no hace referencia a él –por esta razón no se muestra en la Figura 13.6. El objeto podrá recuperarse automáticamente del sistema de persistencia, cuando el usuario acceda a él a través del objeto `miLibro`.

13.3.5 Recuperación de Objetos Persistentes

El tratamiento de objetos persistentes es transparente al usuario [Ortín99c]. El programador no ha de notar diferencia alguna en el tratamiento y utilización de objetos persistentes, frente a los objetos temporales –residentes en la memoria principal.

Un objeto persistente puede poseer un ciclo de vida mayor al programa que lo creó [Booch94]. Así, el programador puede hacer un objeto persistente (`doProxy`) y trabajar con él en disco a través de un objeto delegado; una vez finalizado el programa, la eliminación de la referencia al *proxy* hará que, si no existe otra referencia al éste, el objeto se libere automáticamente por el recolector de basura, quedando constancia de éste únicamente en el dispositivo de persistencia.

En la ejecución de una aplicación, puede interesar recuperar un objeto persistente almacenado en otro momento por cualquier aplicación. Para seguir con el diseño establecido, permitiremos crear un nuevo objeto *proxy* de un objeto almacenado previamente en disco. Lo único que necesitamos conocer de él, es su identificador único.

El método `newProxy` de `PersistenceStream`, tomará un identificador de un objeto como único parámetro, y devolverá un objeto delegado del objeto persistente que posea dicho identificador. Como objeto implícito de esta invocación, deberemos utilizar un objeto representativo de un flujo de persistencia.

[72] La separación entre objetos de usuario y objetos de plataforma es llevada a cabo por un convenio de identificadores. Los identificadores que poseen los caracteres { y }, son propios de objetos de usuario.

Partiendo de un objeto delegado del sistema de persistencia, el método `undoProxy` de `PersistenceStream` permitirá recuperar a la memoria principal, a partir de un objeto delegado, el objeto inicial asociado. Este proceso no implica la eliminación del objeto del sistema de persistencia, sino la supresión de la relación existente entre éste y su *proxy* (si queremos eliminar el objeto deberemos invocar al miembro `delete` del sistema de persistencia).

A igual que a la hora hacer objetos persistentes (§ 13.3.4), se utilizarán objetos que definan política de profundidad: el segundo parámetro pasado a la invocación de `undoProxy`, es un objeto cadena de caracteres que define la profundidad de recuperación de sus miembros, siguiendo un proceso recursivo –véase § 13.3.4.

13.4 Sistema de Distribución

Dentro del entorno de programación, la capa de distribución permitirá acceder a objetos de máquinas remotas, moverlos entre distintas plataformas, y recuperarlos de otras máquinas. El sistema diseñado es adaptable dinámicamente a la utilización de distintos protocolos y representaciones, así como abierto a cualquier otro que el programador desee añadir.

Partiendo de un conjunto de primitivas básicas de distribución implementadas de forma externa (en el objeto *trait* `Extern`), y haciendo uso de la reflectividad estructural de la máquina abstracta, podremos construir un sistema de distribución, en el que los objetos puedan moverse entre las distintas máquinas virtuales interconectadas entre sí, extendiendo así el nivel de abstracción de la plataforma.

La funcionalidad central del diseño del sistema de distribución reside en el concepto de objeto delegado (*proxy*), utilizado también en el diseño del sistema de persistencia (§ 13.3). Los objetos *proxy* delegan sus peticiones de acceso a miembros y evaluación de éstos en objetos distribuidos en otras máquinas virtuales. Para implementar éstos, haremos uso de la reflectividad computacional ofrecida por la máquina, gracias a la inclusión del MOP que permite modificar la semántica de acceso al miembro –descrito en § 13.3.3.

13.4.1 Diseño General del Sistema

La siguiente figura muestra el conjunto de objetos utilizado en el diseño del sistema de distribución, desarrollando éste para dos protocolos de ejemplo, y tomando como muestra la implantación de la máquina virtual en dos plataformas remotas –máquina A y máquina B:

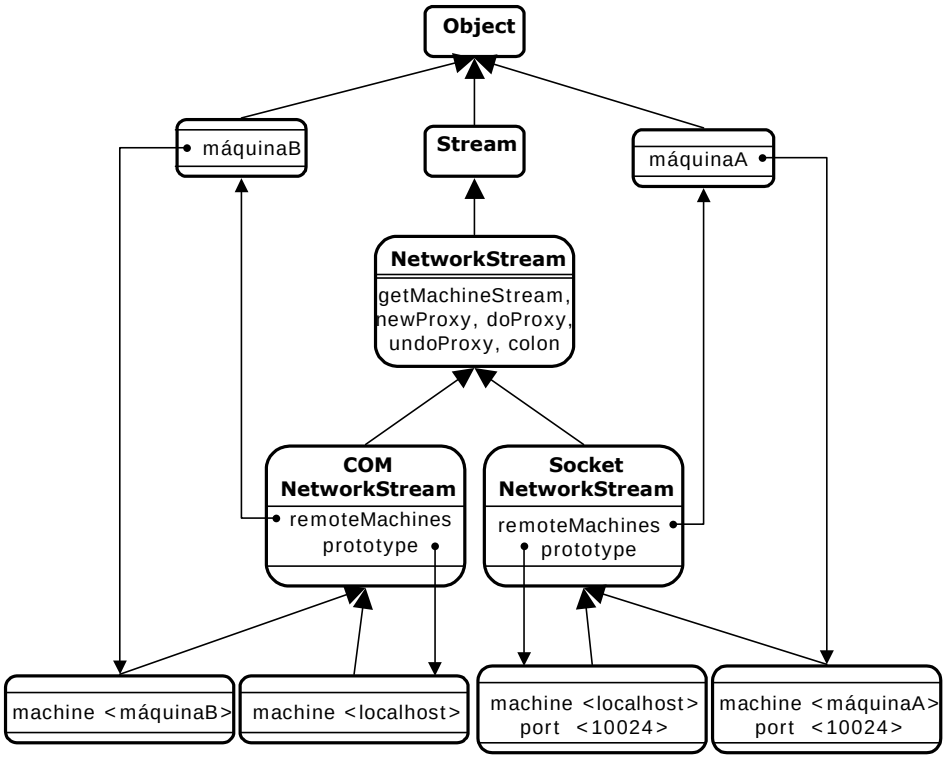


Figura 13.7: Objetos utilizados en el desarrollo del sistema de distribución.

El concepto de flujo de información fue utilizado en el diseño del sistema de persistencia. Como comentábamos entonces, tanto el sistema de persistencia como el de distribución, pueden plantearse como un modo de transferir objetos a través de un flujo. De esta forma, el objeto *trait* `Stream` representa un flujo genérico de información. La abstracción `NetworkStream`, simboliza de un modo genérico los flujos de distribución de objetos, sin tener en cuenta el protocolo utilizado.

Las operaciones dependientes de cada uno de los protocolos a utilizar para interconectar distintas máquinas abstractas, se situarán en objetos derivados de `NetworkStream`. Como ejemplo, se muestran los servicios sobre dos protocolos como *Sockets* TCP/IP y COM. La utilización de un nuevo protocolo se limitará a la adición de un objeto, derivado de `NetworkStream`, que implemente los miembros de estas clases derivadas. El sistema será capaz de amoldarse al nuevo protocolo, al haber sido diseñado utilizando el patrón de diseño *Template Method* [GOF94].

Cada uno de los objetos derivados de `NetworkStream` ofrece las primitivas de interconexión del protocolo que representa. Los objetos derivados a su vez de éstos identificarán la máquina y, si fuere necesario, cualquier otro parámetro de la conexión[73] que se desea establecer, teniendo en cuenta el protocolo seleccionado.

Cada protocolo poseerá un prototipo y, haciendo copias de éste, podremos crear flujos (*streams*) de conexión con una máquina determinada –en el caso de los *sockets*, también en un puerto determinado. Los servicios propios del protocolo se heredarán del objeto padre, al igual que los métodos propios de `NetworkStream`.

[73] Por ejemplo, haciendo uso de *sockets* TCP/IP, es necesario definir un puerto que no haya sido reservado por otra aplicación distribuida.

Una vez creada una copia de un prototipo, debe cambiarse el valor del atributo `machine` por el nombre de la máquina destino. Este objeto será el utilizado por el objeto *proxy* para delegar sus peticiones en un objeto remoto situado en la máquina indicada.

Para evitar que se cree un objeto nuevo derivado del protocolo cada vez que queramos conectarnos con una máquina remota, se lleva un control de los objetos creados, mediante la colección de éstos por el miembro `remoteMachines` del protocolo. El objeto creado contendrá un miembro por cada máquina abstracta remota utilizada previamente. El nombre de cada miembro será el nombre de la máquina remota; el miembro en sí será una copia del prototipo que identifique la conexión con la máquina, bajo el protocolo seleccionado.

El control de acceso y la creación de los flujos remotos, se llevarán a cabo a través del método `getMachineStream`. Recibe el nombre de la máquina remota y devuelve un objeto derivado del protocolo a utilizar (el objeto implícito), cuyo destino es la máquina virtual solicitada.

La primera vez que se solicite la creación de un flujo con una máquina a través de un protocolo, se creará un objeto nuevo derivado del protocolo en cuestión como copia de su prototipo, y se almacenará como miembro del objeto asociado al atributo `remoteMachines`.

13.4.2 Primitivas del Sistema de Distribución

Tal y como se ha diseñado el sistema de distribución, una máquina abstracta ha de permitir al usuario conocer su identificación dentro de la red de computadores. Este identificador se obtiene evaluando el miembro `localHost` del objeto `System` –implementado como primitiva operacional de invocación externa.

El resto de primitivas de distribución se basan en permitir acceder a la interfaz de intercomunicación de una máquina abstracta remota, definida por las seis operaciones descritas en § 12.3. En la siguiente figura, se muestra cómo la implementación externa de éstas ha de lograr acceder a las funcionalidades propias ofrecidas por una máquina remota.

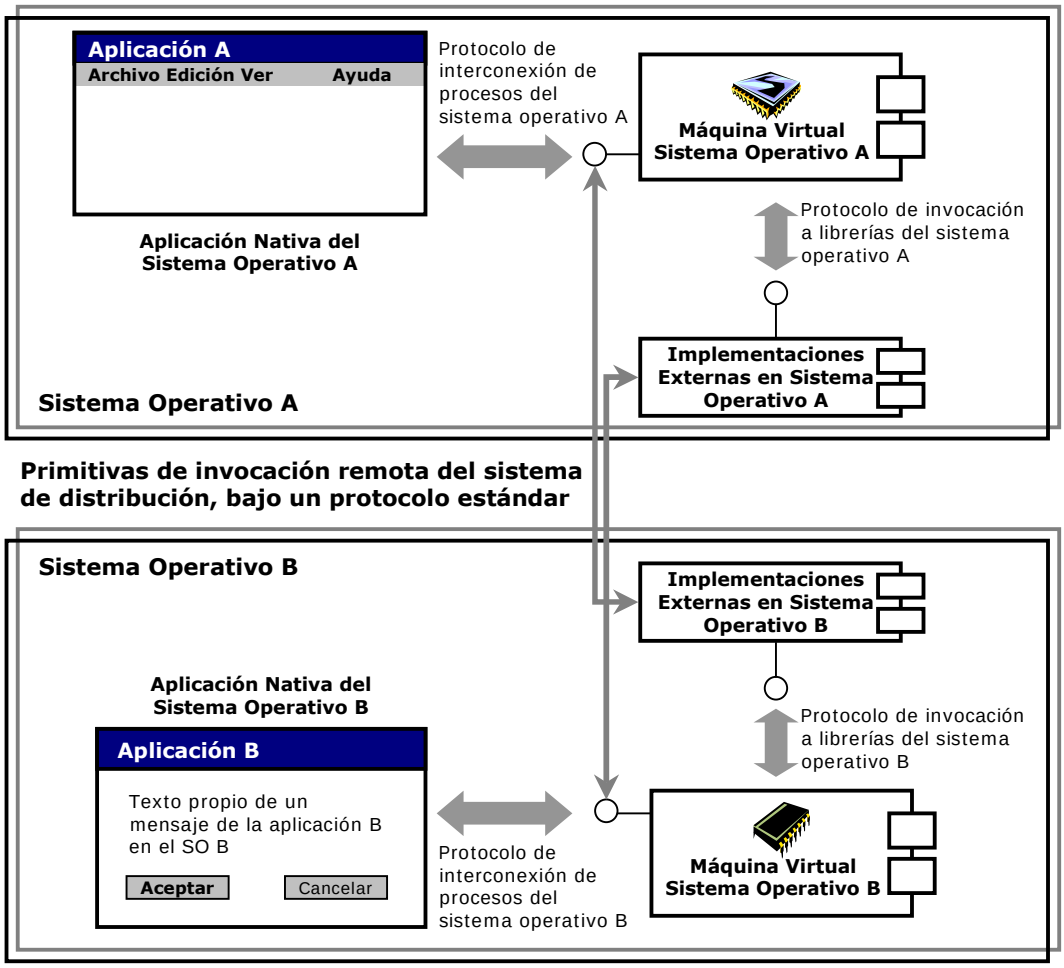


Figura 13.8: Interconexión de dos implementaciones distribuidas de la máquina abstracta.

Las primitivas que ha de implementar todo protocolo, para formar parte del sistema de distribución diseñado, son:

- `newReference`. Creación de una referencia en el contexto raíz.
- `getMember`. Acceso a un miembro de un objeto.
- `primitiveEvaluateString`. Evaluación un objeto cadena de caracteres de forma síncrona y asíncrona.
- `primitiveEvaluateMember`. Evaluación de un miembro de un objeto, de forma síncrona y asíncrona.

13.4.3 Objetos Delegados

La característica común a toda arquitectura de objetos distribuidos (COM, CORBA o Java RMI, por ejemplo) es el acceso a objetos remotos a través de una referencia. Obteniendo en una plataforma local una referencia a un objeto remoto, podremos pasarle a éste un mensaje y provocar así una ejecución remota del método asociado. En nuestro entorno de computación, es posible realizar este tipo de acceso remoto a través de la utilización de objetos delegados.

Como introdujimos en el sistema de persistencia (§ 13.3), un objeto delegado o *proxy* posee un miembro `_stream` que identifica el flujo de donde obtiene su información, y otro miembro "`:`" que sustituye la semántica de funcionamiento de acceso al miembro (§ 13.3.3), buscándolos en el flujo indicado por `_stream`.

Para crear un objeto delegado de otro existente en una máquina remota, es necesario obtener un flujo con la máquina y protocolo deseados (`getMachineStream`), y, utilizando éste como parámetro implícito, invocar a su método `newProxy` pasándole el identificador único del objeto deseado. El resultado es la devolución de un nuevo objeto *proxy* que delega todas las peticiones en el objeto remoto –cada vez que se evalúe un miembro de este objeto desde la plataforma local, se computará su código en la máquina destino.

El siguiente código devuelve un objeto delegado de `miObjeto`, ubicado en `máquinaA`, utilizando *sockets* TCP/IP:

```
<      -> stream;
       stream <- SocketNetworkStream:<getMachineSteam>(<máquinaA>);
       return <- stream:<newProxy>(<{miObjeto}>);
>();
```

Al igual que en el sistema de persistencia, el envío de un objeto a un flujo de información se hace mediante la invocación del método `doProxy` del flujo seleccionado. En el sistema de persistencia, esta operación se traducía al almacenamiento en disco de un objeto; en el sistema de distribución, implica el movimiento de un objeto a la plataforma destino. Para realizar esta operación el programador deberá:

1. Obtener una referencia al flujo deseado, indicando el tipo de protocolo a utilizar y la dirección de la máquina destino. Esta funcionalidad ha sido desarrollada en el miembro `getMachineStream` de cada uno de los protocolos derivados de `NetworkStream`.
2. Invocar al método `doProxy` del *stream* obtenido, pasándole el objeto que deseamos mover y la política que indica la profundidad del movimiento, es decir, el número de miembros del objeto a mover con él –véase 13.3.4.

El resultado es que el objeto local se mueve a la máquina destino quedando éste como objeto delegado del objeto remoto.

Finalmente, el método a utilizar para recuperar objetos delegados es `undoProxy`. Utilizando como objeto implícito el *trait* `NetworkStream`, se le pasa el objeto *proxy* a recuperar y el objeto política que indique la profundidad deseada. El objeto deja de ser delegado, para recuperar su representación en la máquina local.

13.4.4 Recolector de Basura Distribuido

Un objeto *proxy* es una representación virtual y local de su objeto real ubicado en otra máquina abstracta, a la cuál se tiene acceso mediante una red de computadores. Entre un objeto delegado y su objeto real existe una asociación virtual, es decir, una conexión semántica, sin que realmente se haya creado una referencia a éste. Sin embargo, al no existir realmente esta referencia, el recolector de basura podrá tratar de eliminar éste sin que sea correcto, puesto que el objeto delegado requiere su existencia.

Un objeto *proxy* delega toda su funcionalidad en su objeto asociado remoto. El tener una o más referencias a un objeto *proxy*, implica semánticamente que estas referencias sean al objeto real. Para que esta asociación semántica sea implementada de acuerdo con el sistema de recolección de basura de la plataforma, se deberá asegurar que:

1. Un objeto no sea destruido mientras existan objetos delegados de éste en otras máquinas abstractas del sistema de distribución.
2. Un objeto se elimine por su recolector de basura, cuando todos sus objetos delegados hayan sido liberados y no exista en la máquina abstracta referencia alguna a éste.

Nuestro diseño creará un conjunto de referencias a cada objeto local que tenga al menos un *proxy* en otra máquina abstracta, para que no sea destruido por el recolector de basura. De forma paralela, se llevará un contador del número de objetos remotos delegados de éste. Cuando se destruya un objeto *proxy*, se decrementará este contador; se eliminará la referencia, cuando su valor sea cero. A partir de este momento, el objeto local podrá ser limpiado por el recolector, si no existe ninguna referencia local a él.

El miembro `remoteProxies` del objeto `System` implementa la creación, recuento y eliminación de referencias, en función del número de objetos delegados de cada objeto local.

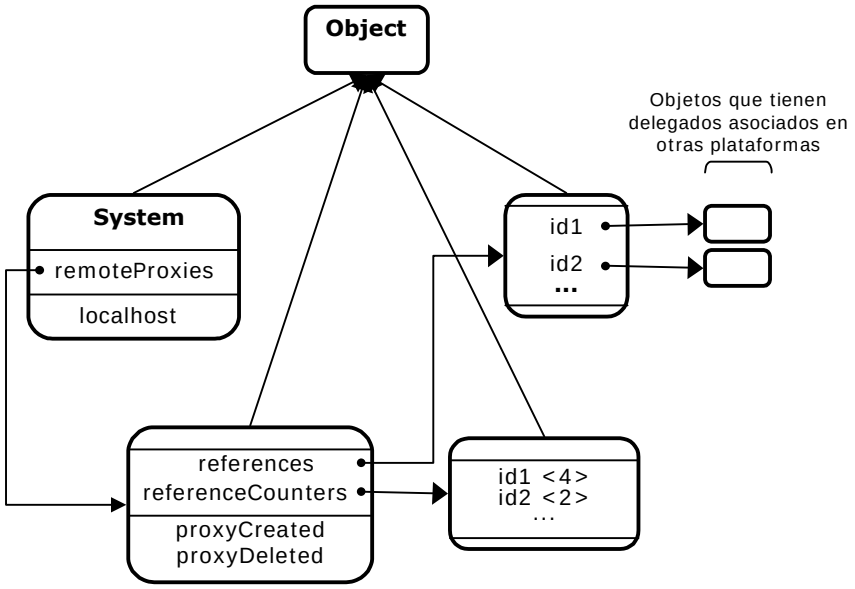


Figura 13.9: Objetos utilizados en el diseño del recolector de basura distribuido.

El objeto miembro `references` del objeto `remoteProxies` es un objeto contenedor, cuyos miembros tienen por nombres identificadores de objetos locales, y sus referencias apuntan a los propios objetos. De forma paralela, el objeto contenedor `referenceCounters` posee los mismos nombres con asociaciones a objetos cadena de caracteres, que contabilizan el número de objetos *proxies* remotos existentes para el objeto asociado.

Para controlar el recuento de referencias, el objeto `remoteProxies` ofrecerá dos métodos para registrar la creación y eliminación de objetos remotos delegados de un objeto local: `proxyCreated` y `proxyDestroyed`, respectivamente.

El miembro `proxyCreated` incrementa el número de referencias del objeto cuyo identificador se pase como parámetro. Si no existe una referencia al objeto en el objeto contenedor `references`, la crea. De la misma forma, `proxyDeleted` elimina los dos miembros del identificador asociado cuando el contador pasa a ser igual a cero. Estos dos métodos son invocados por el sistema de distribución, cada vez que se creen un destruyan objetos delegados.

13.5 Conclusiones

Los diseños llevados a cabo demuestran la capacidad de la plataforma a la hora de extender su nivel de abstracción, sin necesidad de modificar la implementación de la máquina virtual. Toda su codificación ha sido llevada a cabo utilizando su propio lenguaje de programación, evitando así la aparición de distintas versiones de la plataforma –con la consecuente pérdida de portabilidad de código. La introspección del sistema facilita, a una aplicación, el conocimiento de los distintos módulos del entorno de programación que hayan sido implantados –facilitando así su despliegue en sistemas computacionales heterogéneos.

Los diseños presentados para cada módulo, así como las funcionalidades computacionales ofrecidas, son meros ejemplos demostrativos de la extensibilidad y adaptabilidad ofrecidas por la máquina. Tanto otros diseños más sofisticados, como la implementación de otras capacidades computacionales, podrán ser desarrollados para ampliar el nivel de abstracción del sistema.

Inicialmente, la plataforma ofrece primitivas de creación y destrucción de objetos, reflectividad estructural de los mismos, y un pequeño MOP para modificar su semántica; además describe un mecanismo de delegación y otro de evaluación dinámica de objetos. Sobre estas primitivas computacionales básicas, a lo largo de este capítulo, se ha mostrado el diseño de:

- Un recolector genérico de basura. Flexible dinámicamente respecto al algoritmo y política de activación, se pueden implantar distintos recolectores de basura, haciendo uso, básicamente, de la introspección ofrecida por la plataforma.

- Un sistema de planificación flexible de hilos, en el que cualquier política de selección puede ser desarrollada e implantada en tiempo de ejecución. Este módulo se ha diseñado principalmente mediante la manipulación de computación como datos (cosificación), la modificación de la semántica de la evaluación de objetos (reflectividad computacional), y la sincronización de hilos por medio de objetos utilizados como monitores.

- El módulo de persistencia, que permite utilizar objetos que delegan su funcionalidad en un flujo asociado a un dispositivo persistente y representación determinados, siendo éstos parámetros totalmente adaptables. Para su desarrollo se ha utilizado: la reflectividad estructural, para almacenar, conocer y recuperar todos los miembros de un objeto; la cosificación, para representar computación como datos; y la reflectividad computacional de modificación de acceso al miembro, para implementar objetos delegados.

- El sistema de distribución, ha sido desarrollado de un modo similar al de persistencia, delegando su funcionamiento en un flujo de conexión entre ordenadores. Lo conseguido es invocación remota de métodos y movilidad de éstos entre distintas máquinas, de un modo adaptable al protocolo seleccionado.

CAPÍTULO 14:
DISEÑO DE UN PROTOTIPO DE COMPUTACIÓN REFLECTIVA SIN RESTRICCIONES

En este capítulo se describirá el diseño de un prototipo que demuestre la posibilidad de construir el sistema computacional de programación flexible descrito en el capítulo 9, a partir de las premisas o requisitos mínimos impuestos en § 11.4.

Por simplicidad de desarrollo, se ha implementado el prototipo sobre un lenguaje de programación comercial que otorgue los requisitos mínimos necesarios para llevar a cabo su codificación[74]. El objetivo de este prototipo es demostrar que la consecución de los objetivos y requisitos propios del sistema computacional reflectivo, enunciados en los dos primeros capítulos de esta tesis, son factibles bajo las premisas mínimas expuestas en § 11.4.

La implementación del prototipo presentado sobre la máquina abstracta descrita en el capítulo 10 es totalmente factible al cumplir ésta con las premisas anteriormente mencionadas. Sin embargo, el desarrollo de este prototipo sobre la arquitectura de nuestra máquina virtual, aportaría los beneficios propios de cumplir los requisitos expuestos en § **2.1** y pragmáticamente demostrados con los casos estudiados en el capítulo 4.

El código fuente del prototipo presentado, así como un diseño detallado y una batería de pruebas, pueden ser descargados en la URL:
`http://www.di.uniovi.es/reflection/lab/prototypes.html`.

14.1 Selección del Lenguaje de Programación

A raíz de las premisas mínimas impuestas al lenguaje de programación para el desarrollo de nuestro prototipo –expuestas en § 11.4–, 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 programa-

[74] Puesto que el objetivo de esta memoria es justificar, enunciar y evaluar una tesis, no consideramos necesaria la implementación de este segundo prototipo sobre el primero –nuestra máquina abstracta. Demostramos que su desarrollo es factible bajo unas premisas computacionales que cumple la arquitectura de nuestra máquina abstracta, e implementamos éste sobre un lenguaje de programación comercial que nos facilite considerablemente su desarrollo, orientándolo a la justificación y evaluación de nuestro estudio.

dor, 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.

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 ejecución [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 [Cueva92b] 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.

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].

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.

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 lenguaje de programación en § 11.4, y cumplidas por la máquina abstracta presentada en el capítulo 10, era crear 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 14.1, el hilo principal crea un objeto denominado `nitrO` que sigue el patrón de diseño Fachada [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[75].

[75] 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.

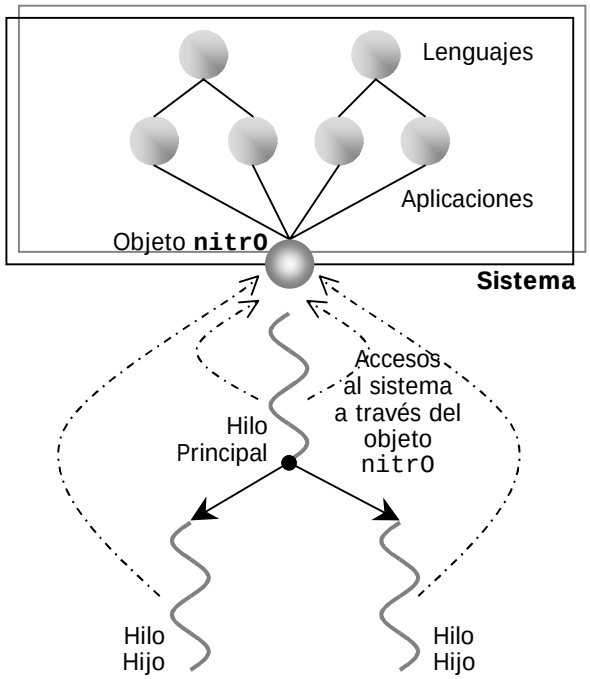


Figura 14.1: Acceso al objeto nitrO desde distintos hilos representativos de aplicaciones.

14.2 Diagrama de Subsistemas

El diseño del prototipo ha sido dividido en un conjunto de subsistemas, implementados como módulos Python. Éstos y sus dependencias se muestran en la siguiente vista estática, expresada mediante UML [Rumbaugh98]:

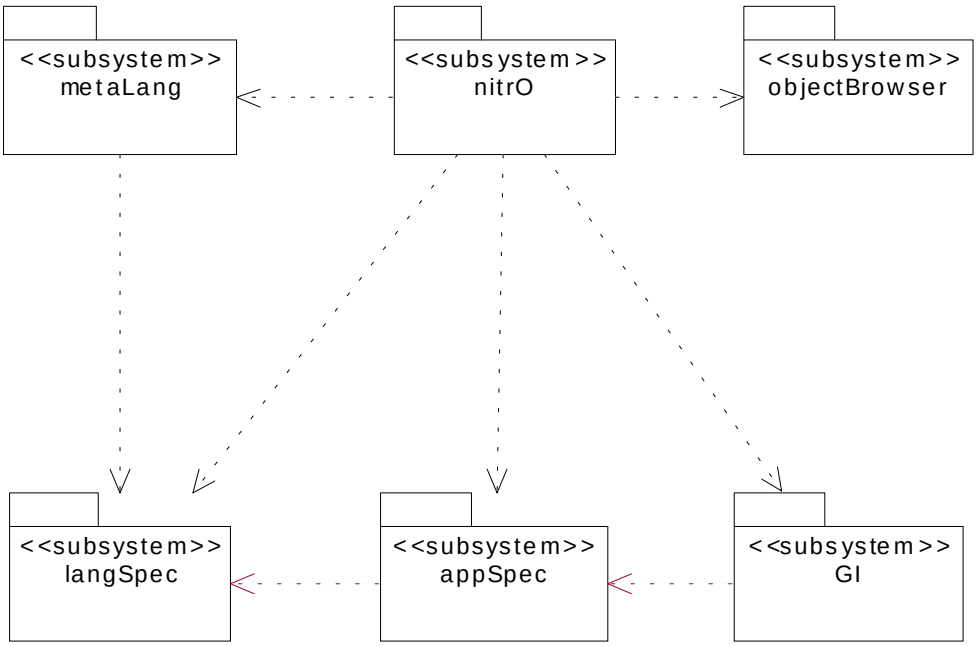


Figura 14.2: Diagrama de subsistemas del prototipo.

14.2.1 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.

14.2.2 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.

El aspecto de esta herramienta introspectiva se muestra en el manual de usuario de este prototipo (apéndice B).

14.2.3 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.

14.2.4 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[76].

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.

[76] Este lenguaje expresa lenguajes de programación de tipo 2. Su gramática y representación de semántica pueden consultarse en el apéndice B.

14.2.5 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 [Cueva95] (*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`.

14.2.6 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.

14.3 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. La especificación formal del metalenguaje, capaz de describir cualquier lenguaje libre de contexto, está descrita en el apéndice B.

14.3.1 Diagrama de Clases

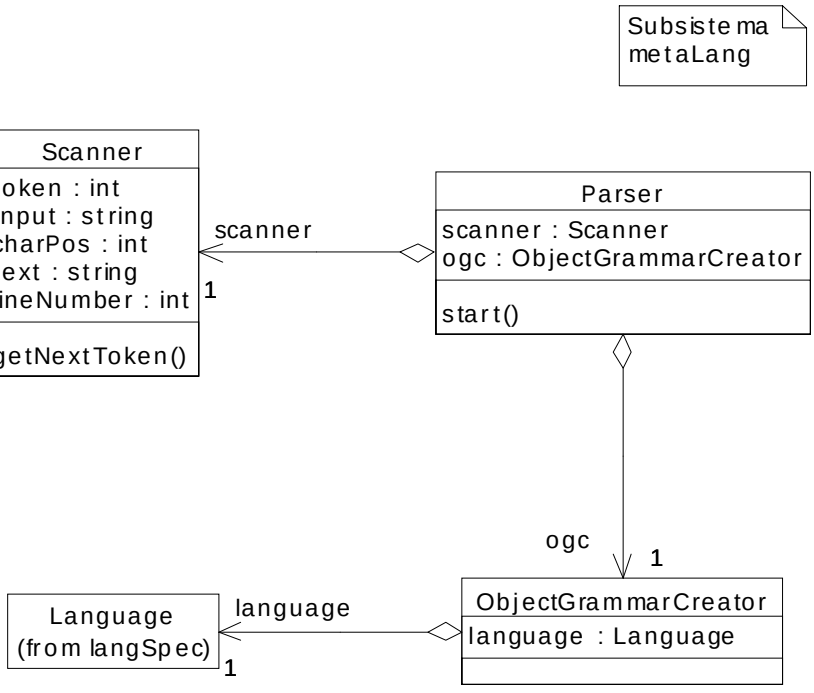


Figura 14.3: Diagrama de clases del subsistema metaLang.

El conjunto de clases sigue la estructura clásica de un procesador de lenguaje para llevar a cabo el análisis léxico y sintáctico del metalenguaje descrito en el apéndice B: 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.

14.4 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 flexibilidad de nuestro sistema. Todos ellos se obtienen, como hemos indicado en el capítulo 11, 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.

14.4.1 Diagrama de Clases

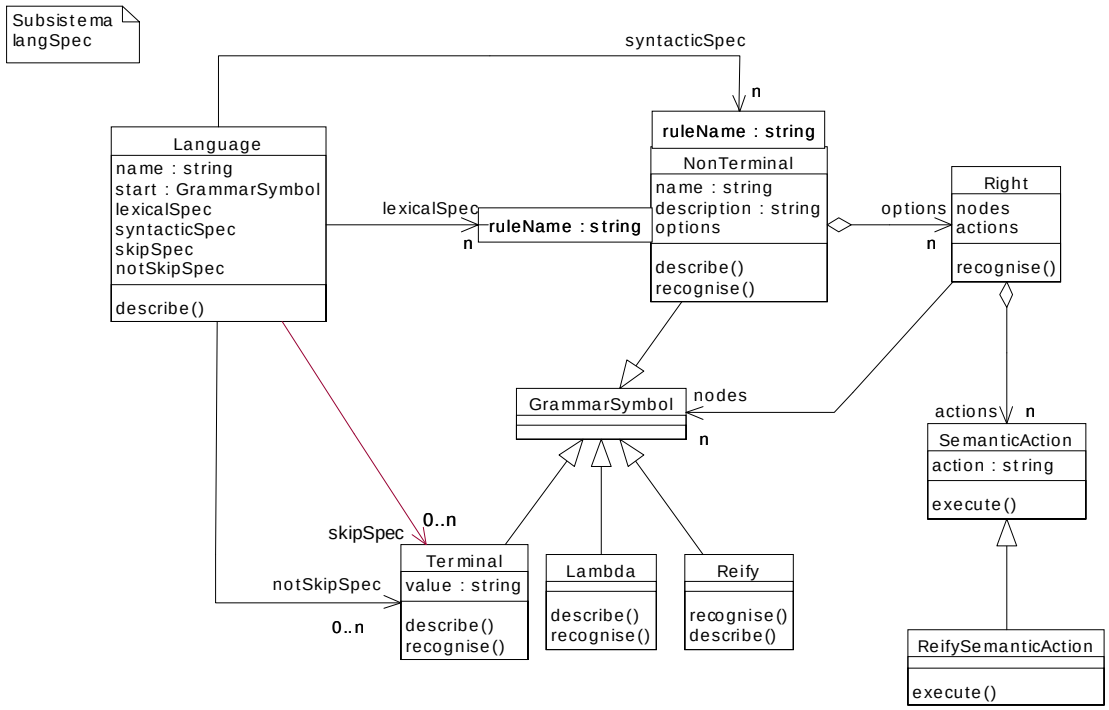


Figura 14.4: Diagrama de clases del subsistema langSpec.

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`[77]. 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 `SemanticAction` y se ac-

[77] 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.

cede 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[78].

Como hemos explicado en § 9.5, 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`.

14.4.2 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[79]:

```
Language = Print
Scanner = {
      "Digit Token"
      digit -> "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"    ;
      "Number Token"
```

[78] Para comprender mejor la utilidad de estos elementos consúltese el apéndice B.
[79] La descripción del lenguaje ha sido llevada a cabo mediante el metalenguaje de nuestro sistema, especificado en el apéndice B.

```
      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:

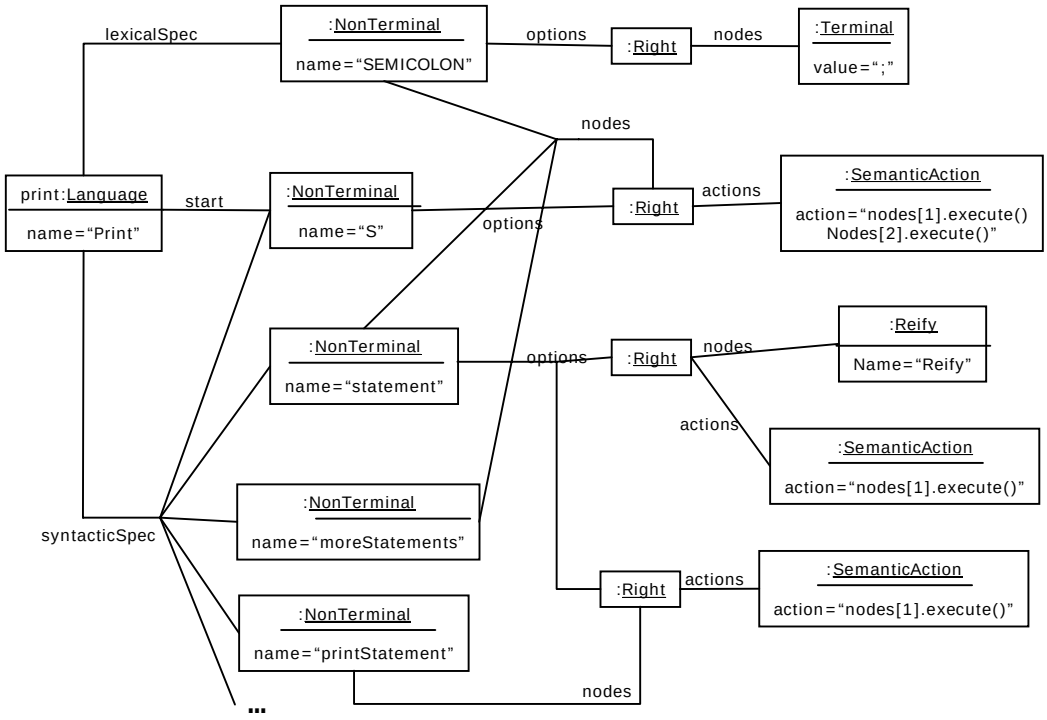


Figura 14.5: Diagrama de objetos de la especificación del lenguaje "Print".

Como se observa en la Figura 14.5, 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.

14.5 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:

14.5.1 Diagrama de Clases

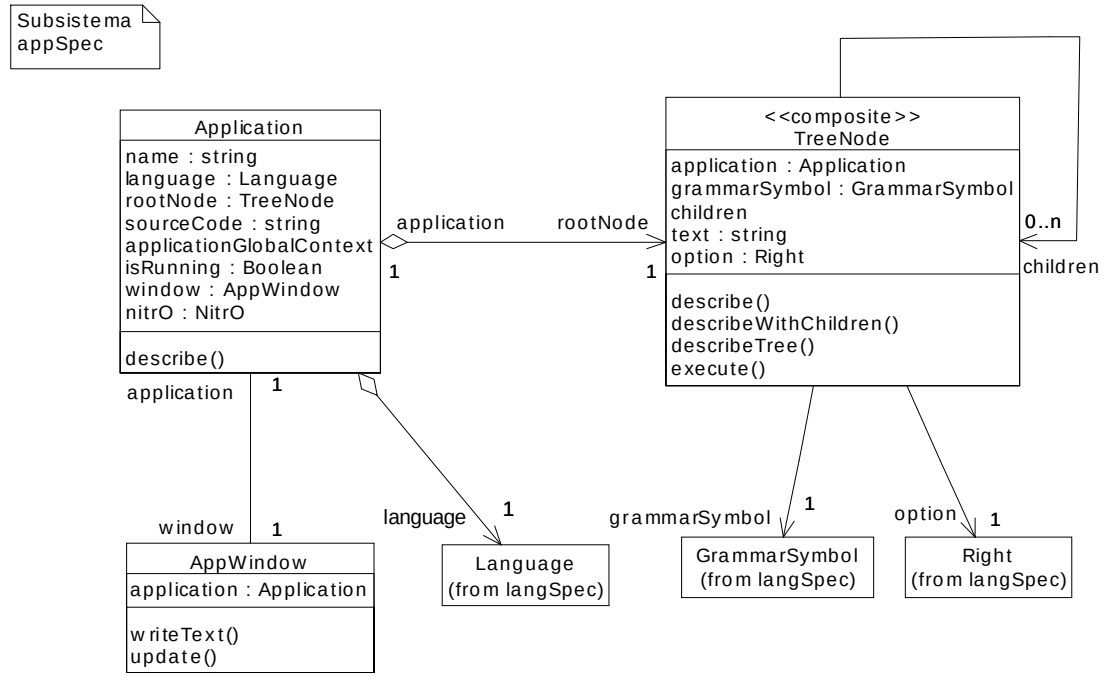


Figura 14.6: Diagrama de clases del subsistema appSpec.

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 una tesis, hemos seguido el criterio de utilizar un grano más fino: cada aplicación está asociada a una única representación de un lenguaje[80]. 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`.

[80] 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.

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`.

14.5.2 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 lenguaje, 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 § 14.4.2, 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:

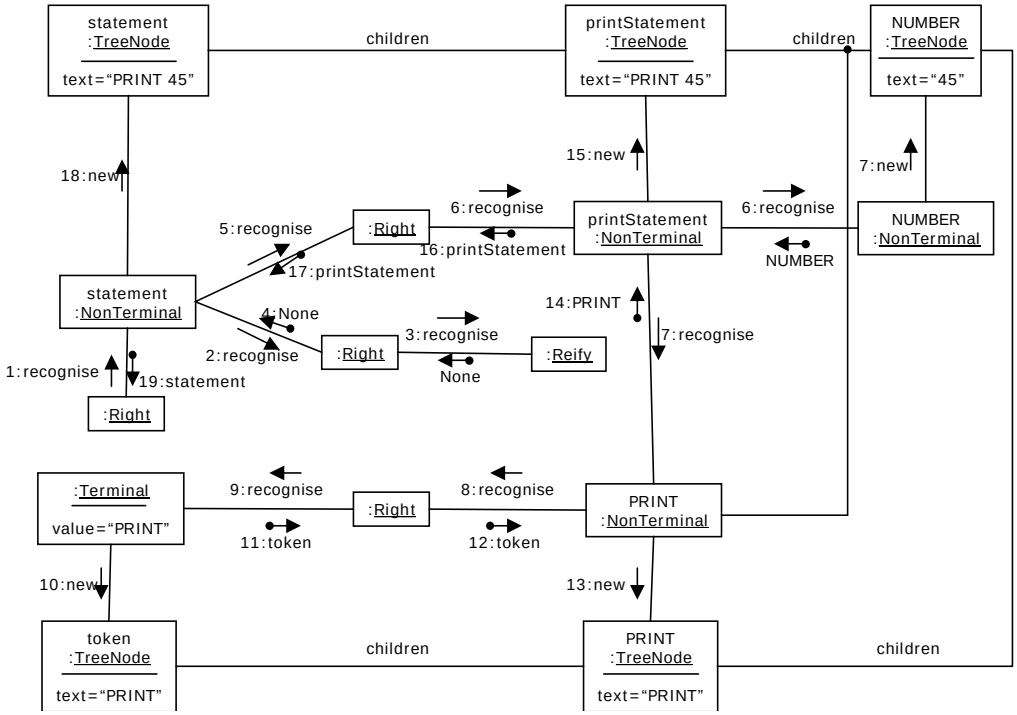


Figura 14.7: Creación del subárbol sintáctico propio de una sentencia.

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.

14.6 Subsistema GI

Los objetivos principales de este subsistema son:

1. Invocación del algoritmo de reconocimiento sintáctico con retroceso, aportando un buffer en el que se encuentra el código fuente a analizar.
2. Ejecutar la aplicación, recorriendo el árbol sintáctico y evaluando los nodos como se haya descrito en la especificación del lenguaje.
3. Eliminar de la entrada los componentes léxicos descritos en la sección `skipSpec`.
4. Concatenar automáticamente los tokens especificados como `notSkipSpec` en el metalenguaje.

14.6.1 Diagrama de Clases

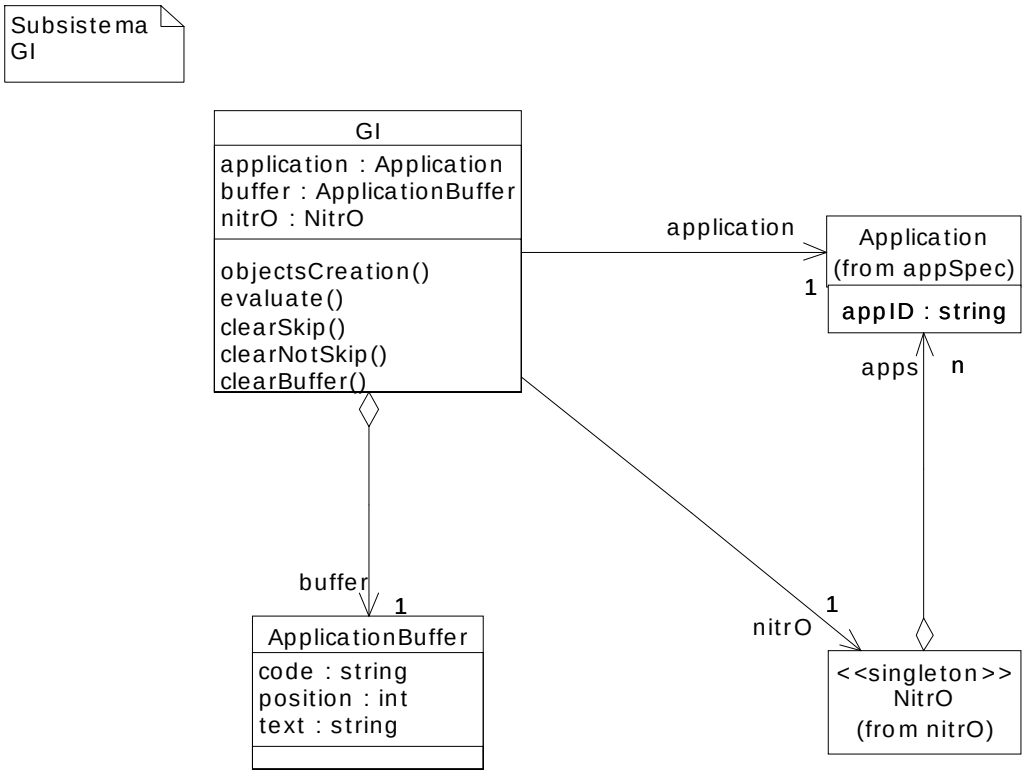


Figura 14.8: Diagrama de clases del subsistema GI.

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 § 14.5.2, 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 § 14.5.2.

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.

14.6.2 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 mostrados en este capítulo, la ejecución del método asociado produce la evaluación del árbol siguiendo el siguiente diagrama de colaboración[81]:

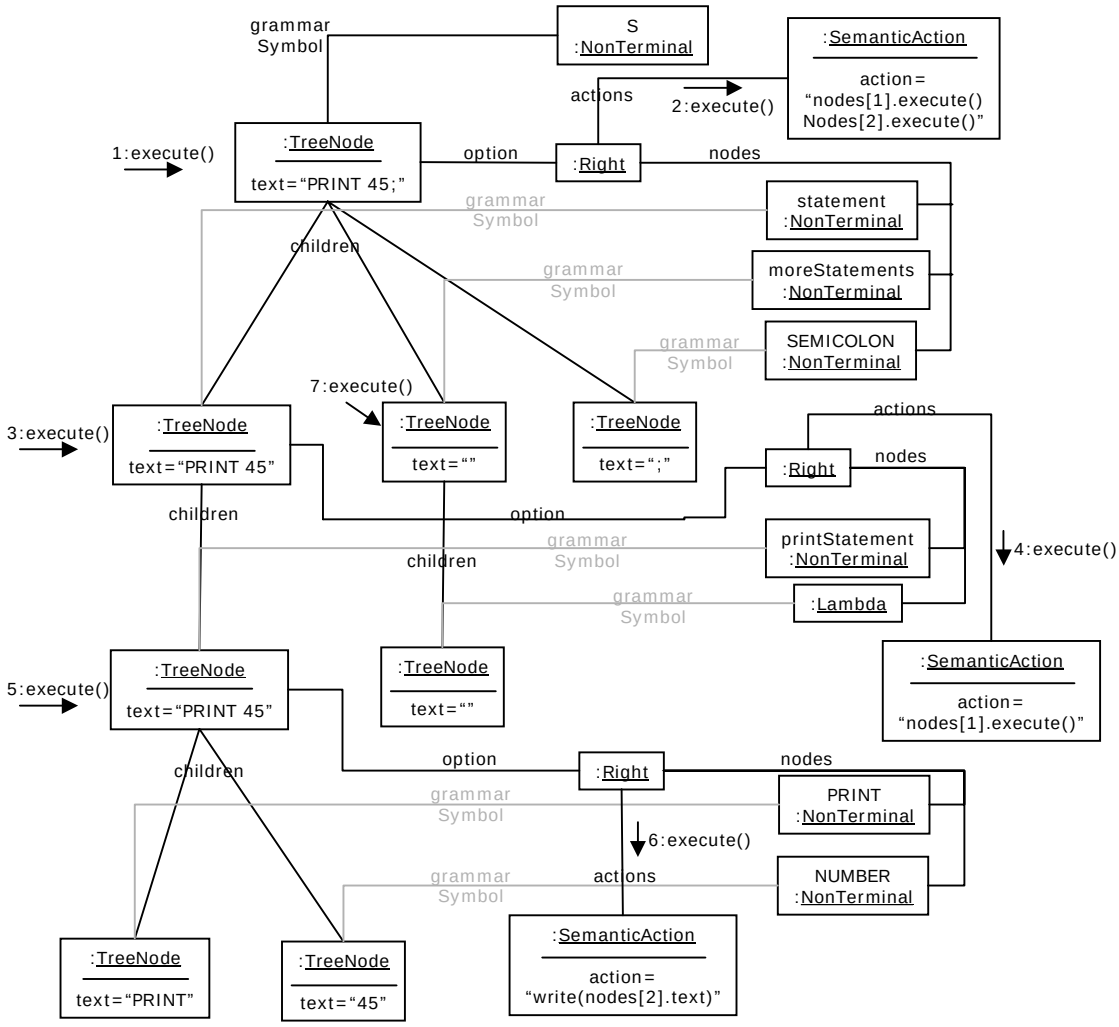


Figura 14.9: Evaluación de un árbol sintáctico.

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.

[81] 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.

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)”`.

14.7 Subsistema nitrO

El subsistema `nitrO` define el objeto `nitrO` que sigue el patrón de diseño Fachada [GOF94], ofreciendo todas las posibilidades del sistema.

14.7.1 Diagrama de Clases

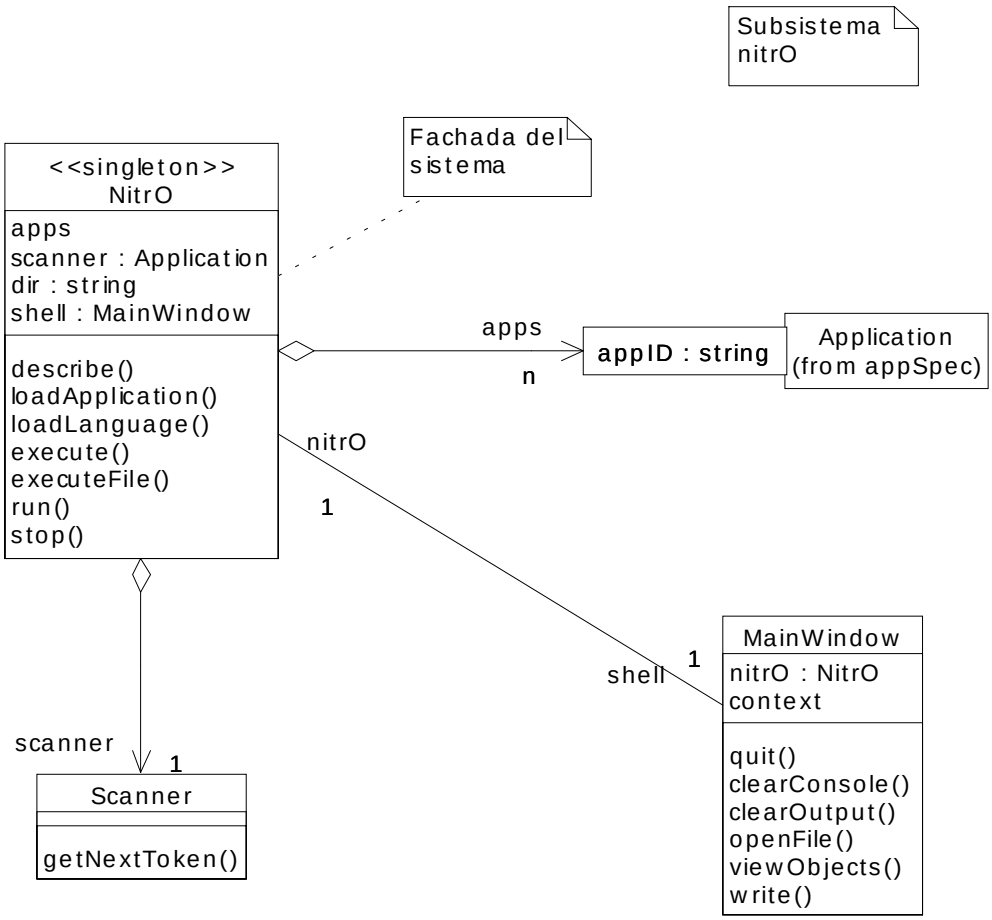


Figura 14.10: Diagrama de clases del subsistema nitrO.

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, siguiendo la gramática de aplicaciones descrita en el apéndice B. 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[82].

[82] Más información acerca de la utilización del intérprete de comandos puede obtenerse en el apéndice B.

CAPÍTULO 15:
ÁMBITOS DE APLICACIÓN DEL SISTEMA

Las características propias del sistema presentado en esta tesis doctoral pueden ser empleadas en distintos ámbitos de desarrollo de aplicaciones. A modo de ejemplo en la utilización de todas las ventajas aportadas por el sistema, pondremos el punto de atención en el análisis de grupos de aplicaciones para los que se pueda aplicar de un modo sencillo los beneficios aportados.

De un modo general, podemos decir que el tipo de aplicaciones que mayor beneficio puede obtener de la flexibilidad de nuestro sistema, son aquéllas que necesiten amoldarse dinámicamente a requisitos imprevisibles en tiempo de desarrollo, sin que sea posible modificar el código de la aplicación ni parar su ejecución. Un caso práctico de esta utilidad, es la adaptación dinámica de aspectos: una aplicación es definida por su código funcional y, en tiempo de ejecución, podrá inspeccionar y adaptar sus aspectos a nuevas necesidades –ejemplos de aspectos son persistencia, depuración, distribución, sincronización, monitorización, tolerancia a fallos o mediciones.

Una de las características propias de nuestro sistema es la independencia del lenguaje de programación a utilizar. Los distintos grados de flexibilidad aportados, de modo diferente a la mayoría de los sistemas existentes, no dependen del lenguaje que el programador haya deseado utilizar.

15.1 Sistemas de Persistencia y Distribución

Éste es uno de los campos en el que más se ha investigado la utilización de los distintos grados de flexibilidad ofertados por distintos sistemas [Foote90] –ya sean o no reflectivos.

Los sistemas de persistencia y distribución pueden plantearse mediante un criterio de diseño común: la utilización de información que realmente no se encuentra ubicada en el lugar donde es demandada; esta localización puede suponer un sistema de almacenamiento persistente, o bien otra computadora. Basándonos en esta similitud, vamos a analizar las posibilidades que oferta nuestro sistema para desarrollar ambos sistemas.

15.1.1 Adaptabilidad a los Cambios

Una vez desarrollada una aplicación que haga uso de sistemas de persistencia y distribución, puede suceder que, en el mantenimiento de ésta, surja la necesidad de realizar cambios. Siempre es deseable que la mayor parte de estos cambios supongan el menor impacto posible al sistema existente –minimizar el número de cambios a implementar.

Cambios clásicos en persistencia y distribución son la evolución de los esquemas de las bases de datos (clases, en orientación a objetos) [Roddick95], y la modificación del conjunto de servicios ofertados por el servidor –ya sean nuevos objetos, o bien nuevos métodos de los objetos existentes.

El hecho de que se pueda conocer dinámicamente la estructura de un objeto (introspección) y que se pueda modificar ésta sin restricción alguna (reflectividad estructural), hace que resulte factible la implementación de un mecanismo sencillo de modificación estructural de los objetos, en el que se pueden añadir, eliminar o modificar los distintos miembros que sea necesario adaptar –evolucionando así la estructura de un conjunto determinado de objetos.

En el desarrollo de aplicaciones distribuidas, cualquiera que sea la tecnología a utilizar, se suele identificar el concepto de servidor y cliente de un servicio [Orfali98]. Los pasos comunes a efectuar son:

1. Definir la interfaz del servicio.
2. Procesar la interfaz para conseguir el modo en el que se implementará por el servidor (comúnmente denominado *skeleton*) y se invocará por el cliente (*stub*).
3. Implementar, mediante el código obtenido en el punto anterior, los clientes y servidores de dicho servicio.

Una vez desplegado el sistema distribuido, puede surgir la necesidad de aumentar los servicios del servidor para implementar un nuevo cliente, sin necesidad de modificar los clientes ya existentes. Como mencionábamos en el segundo paso, la modificación del servicio generaría nuevo código de implementación e invocación, teniendo que modificar los clientes implantados.

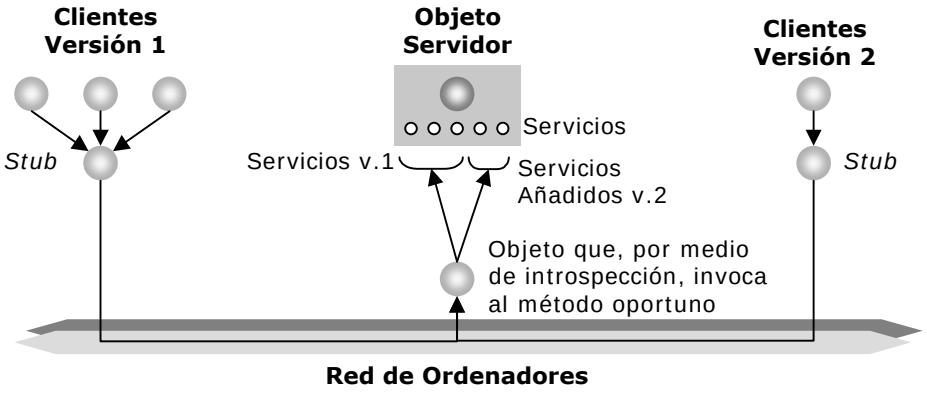


Figura 15.1: Adaptabilidad a distintas versiones de servidor y cliente mediante introspección.

Mediante el uso de introspección es posible eliminar la necesidad de generar los esqueletos del servidor: cuando el cliente invoca al objeto servidor, la recepción del mensaje remoto puede analizarse dinámicamente y comprobar si existe un método a invocar[83]. Del

[83] De hecho, esta implementación ha logrado eliminar en la plataforma de Java la necesidad de los *skeletons* del servidor en la versión 2 de RMI (*Remote Method Invocation*) [Allamaraju2000].

mismo modo, sería factible implementar un sistema de mantenimiento de versiones de servidor al igual que en los sistemas de persistencia [Roddick95].

15.1.2 Replicación y Movilidad de Objetos Flexible

La conversión de un objeto en un conjunto secuencial de bytes permite a la plataforma Java transferir dicha representación del objeto a un sistema de almacenamiento persistente, o a otra plataforma remota. El proceso anterior puede llevarse a cabo mediante introspección.

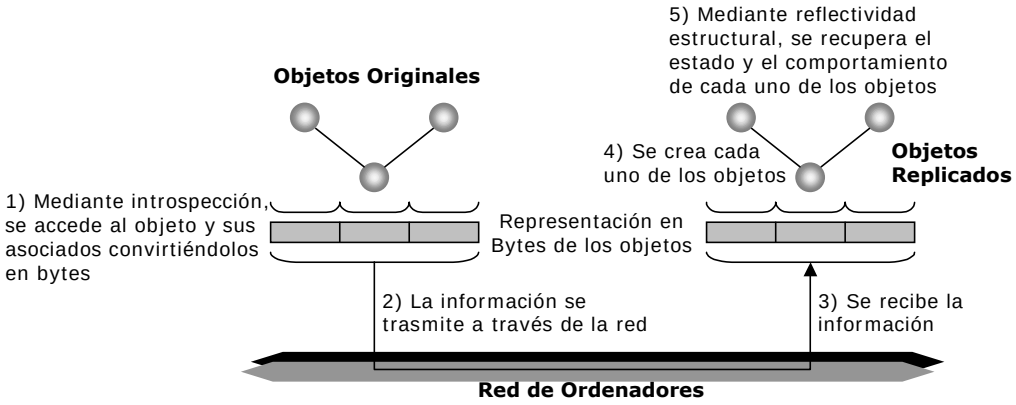


Figura 15.2: Replicación de objetos a través de una red de ordenadores.

Mediante reflectividad estructural, el programador podrá crear dinámicamente un objeto e ir asignándole sus distintos atributos y métodos –tanto la estructura como el comportamiento de los objetos es representado mediante datos (cosificación). En Java, el programador no puede realizar este proceso; es la máquina virtual quien ofrece la primitiva de obtención dinámica del objeto [Sun97e]. La posibilidad de codificar esta funcionalidad sobre el propio lenguaje flexibiliza tanto el modo en el que un objeto es representado por una información, como la forma de recuperarlo.

Gracias a la creación dinámica y modificación del estado de los objetos, el programador puede desarrollar sistemas de replicación de información y migración de objetos por las distintas plataformas del sistema, utilizando los algoritmos y estructuras de datos que él estime oportuno.

15.1.3 Independencia del Aspecto y Selección Dinámica

El mecanismo utilizado para almacenar y recuperar un objeto de disco (factores como frecuencia, compresión o sistemas de indexación), así como el mecanismo utilizado para interconectar aplicaciones (protocolo y representación utilizados), pueden desarrollarse de modo que sean adaptables en tiempo de ejecución.

Como se llevó a cabo en el diseño del prototipo de entorno de programación para la máquina abstracta (capítulo 13), se puede modificar –mediante reflectividad computacional– la semántica del paso de mensajes para que éste acceda a un objeto remoto o persistente. Una vez enviado un mensaje, se localiza el objeto y se ejecuta el método apropiado allá donde esté. De este modo el usuario tiene la ilusión de que es local, cuando no se encuentra realmente en memoria.

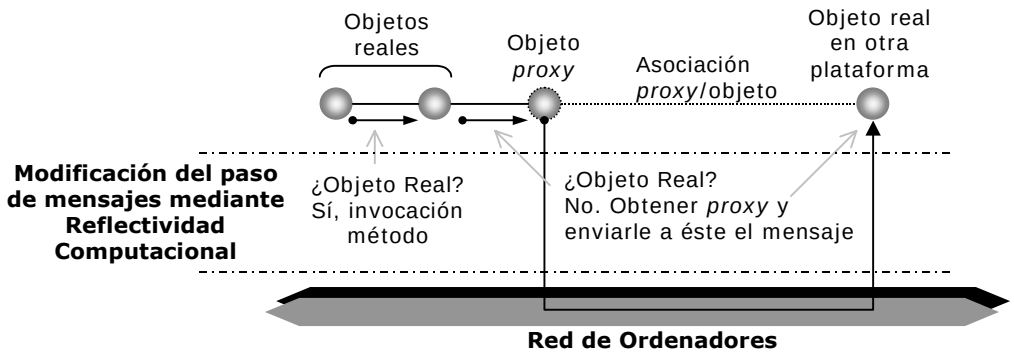


Figura 15.3: Modificación del paso de mensajes para desarrollar el sistema de distribución.

Además de la transparencia ofrecida, se permite utilizar una separación dinámica de aspectos. El programador no desarrollará la aplicación teniendo en cuenta que ésta será distribuida o persistente; tan sólo deberá dejar constancia de su especificación funcional. Dinámicamente se le atribuirán aspectos adicionales como su característica de persistencia o distribución, sin necesidad de modificar el código fuente de su especificación funcional.

Haciendo uso de la introspección del sistema se pueden desarrollar aplicaciones que modifiquen automáticamente sus aspectos –o los parámetros de cada uno de los aspectos. Por ejemplo, puede codificarse la modificación del mecanismo del sistema de indexación en función de la estructura de los objetos a consultar[84] [Ortin99]. Si la plataforma posee una elevada carga de procesamiento, podrá reducirse ésta automáticamente cambiando la frecuencia de actualizaciones a disco, o incluso enviando la computación de ciertas aplicaciones a otras plataformas físicas.

15.1.4 Agentes Móviles Inteligentes y Autodescriptivos

Haciendo uso del entorno de programación con características de distribución, es posible utilizar nuestra plataforma como una plataforma de agentes móviles. Como hemos visto, un objeto puede convertirse a una serie de bytes (mediante introspección), trasmitirse a través de una red, y recuperarse en la plataforma destino (mediante reflectividad estructural).

En la arquitectura del sistema reflectivo presentado (capítulo 11), vimos cómo una aplicación posee una asociación con la descripción de su lenguaje, estando ambos representados por una estructura de objetos. Mediante la utilización de los servicios descritos en el párrafo anterior, es posible mover una aplicación y la especificación de su lenguaje de programación a otra plataforma para que sea ejecutada sobre ésta, aunque no posea previamente la descripción de su lenguaje.

De forma adicional, un agente es capaz de analizar dinámicamente lo existente en su entorno (introspección) y tomar decisiones en función del contexto descubierto. Un ejemplo podría ser el desarrollo de un agente que sea capaz de encontrar un objeto en todas las máquinas del sistema, codificado en un lenguaje propio no instalado en el resto de plataformas.

Otra de las características de nuestro sistema radica en la posibilidad de crear aplicaciones que describan su propio lenguaje –capítulo 11. Si se desarrolla la aplicación con el

[84] Existen sistemas de indexación que ofrecen mejor resultados que otros en función del tipo de información a buscar u operación a realizar. Por ejemplo, "*nested index*" posee mejores resultados que "*multiindex*" para la consulta, pero peores para la actualización [Martínez2001].

lenguaje que ella misma describe, ésta será susceptible de ser descarga de cualquier servidor y ejecutada en una plataforma de nuestro sistema, sin necesidad de tener un conocimiento previo del lenguaje utilizado.

15.1.5 Distribución de Aplicaciones Independientemente de su Lenguaje

La raíz computacional de nuestro sistema se centra en la implementación de una máquina virtual capaz de ejecutar código de su plataforma abstracta. Su semántica es reducida para poder implantarse en cualquier hardware, sistema operativo o aplicación –por ejemplo, un navegador de Internet. Una vez implantada la máquina, el sistema computacional puede crecer con código reflectivo que extienda su nivel de abstracción, sin necesidad de modificar, en ningún momento, la implementación de la máquina virtual.

El diseño de la tercera capa de nuestro sistema consigue, mediante su intérprete genérico, una independencia total del lenguaje a interpretar. La ejecución de cualquier programa y la interacción entre aplicaciones es factible de un modo independiente al lenguaje de programación utilizado para su codificación. De forma adicional, como se muestra en § B.2, las aplicaciones pueden contener en su propio código fuente la especificación del lenguaje que empleen. Así, dichas aplicaciones podrán ser ejecutadas sin necesidad de instalar su lenguaje en la plataforma oportuna.

La utilización de la independencia del lenguaje por una aplicación navegador que implemente una máquina virtual de nuestra plataforma abstracta, permite ofrecer la ejecución de cualquier aplicación, codificada sobre cualquier lenguaje de programación, descargada de un servidor. Adicionalmente, esta aplicación podrá interactuar con cualquier otro programa desarrollado sobre nuestra plataforma, al utilizar un único modelo computacional de objetos.

15.2 Sistemas de Componentes

El desarrollo del sistema de componentes de Java, JavaBeans™ [Sun96], está directamente ligado con el concepto de introspección. Si se desarrolla una clase para la plataforma de Java, se puede utilizar ésta como componente que ofrece sus métodos públicos; las parejas de métodos cuyos nombres comiencen con `get` y `set`, seguidos del mismo identificador, describen la propiedad asociada a dicho identificador; las parejas de métodos que comiencen con `add` y `remove`, describen la posibilidad de gestionar eventos siguiendo el patrón “Observador” (*Observer*) [GOF94].

La carencia de introspección en la plataforma nativa de los sistemas operativos de Microsoft, obliga a su modelo de componentes COM [Microsoft95] a simular un entorno introspectivo. Cuando se desarrolla un componente para esta plataforma, es necesario implementar un método `QueryInterface` que devuelva una referencia al *interface* solicitado, en el caso de que éste sea implementado. La carencia en la utilización de introspección, puede llevar a incoherencias en el modelo cuando no coincida la implementación del componente con lo descrito ante la invocación de `QueryInterface`.

Siguiendo con el criterio de separación de incumbencias [Hürsch95] o aspectos [Kiczales97], se ha profundizado comercialmente en el desarrollo de sistemas transaccionales de componentes de aplicación, como EJB (*Enterprise Java Beans*) [Sun98b] o COM+ [Kirtland99b]. Un componente de aplicación describe únicamente su funcionalidad y se incrusta en un servidor de aplicaciones [Allamaraju2000]. Los distintos aspectos de este componente –tales como persistencia, transacciones, utilización de caché, seguridad o re-

cuperación ante los errores– pueden ser descritos por el programador de un modo declarativo, es decir, separado del código propio del componente.

Los sistemas de componentes transaccionales de aplicaciones otorgan al desarrollador de aplicaciones una determinada flexibilidad respecto al conjunto de aspectos que, tras un determinado análisis de aplicaciones reales, se han identificado como comunes a la mayoría de programas. El programador desarrolla la lógica de su problema (*business logic*) y, a la hora de implantar o desplegar su aplicación, deja constancia de la configuración de los aspectos necesarios.

El grado en el que se pueden definir los aspectos de un componente de aplicación estará siempre limitado por sus interfaces de acceso. Si queremos definir un sistema de componentes de aplicación ilimitadamente flexible, podemos hacer uso de las características de reflectividad computacional: la semántica del modo en el que se utilizan los componentes puede variarse con la expresividad propia de un lenguaje de meta-comportamiento, para separar realmente la funcionalidad de los aspectos, sin ningún tipo de restricción preestablecida.

Un ejemplo de la utilización de reflectividad computacional en el entorno descrito, es el propuesto por Zhixue Wu [Wu98]: toma la descripción del modelo de servicios de transacciones descrito por OMG [OMG94], y permite modificar su semántica, de un modo declarativo, mediante la utilización de metaobjetos.

15.3 Paradigma de Programación Dinámicamente Adaptable

La reflectividad otorga un grado elevado de flexibilidad en el desarrollo de aplicaciones. Igual que el polimorfismo permite especificar en tiempo de desarrollo distintos comportamientos de abstracciones genéricas, la reflectividad permite describir modificaciones dinámicas en la semántica y estructura de las aplicaciones. La reflectividad es considerada como un polimorfismo dotado de mayor flexibilidad [Golm98].

De modo distinto al polimorfismo y a la separación de aspectos, la flexibilidad que estudiaremos en este punto es obtenida dinámicamente por el sistema, sin necesidad de finalizar su ejecución, siendo adaptada por ella misma o cualquier otro proceso.

15.3.1 Polimorfismo Dinámico

Haciendo uso de las tecnologías orientadas a objetos, el polimorfismo posibilita la creación de aplicaciones que puedan ser ampliables respecto a un tipo de abstracción, facilitando así su mantenimiento.

Si deseamos hacer ampliable un proceso, de forma que pueda trabajar con un número indeterminado de abstracciones, podemos identificar la interfaz de tratamiento genérico de estas abstracciones mediante una clase base o un *interface*. Cada abstracción real derivada implementará el comportamiento genérico de un modo específico distinto. Como se muestra en la siguiente figura, el tratamiento genérico es programado mediante abstracciones base, pero recibiendo realmente instancias de las clases derivadas.

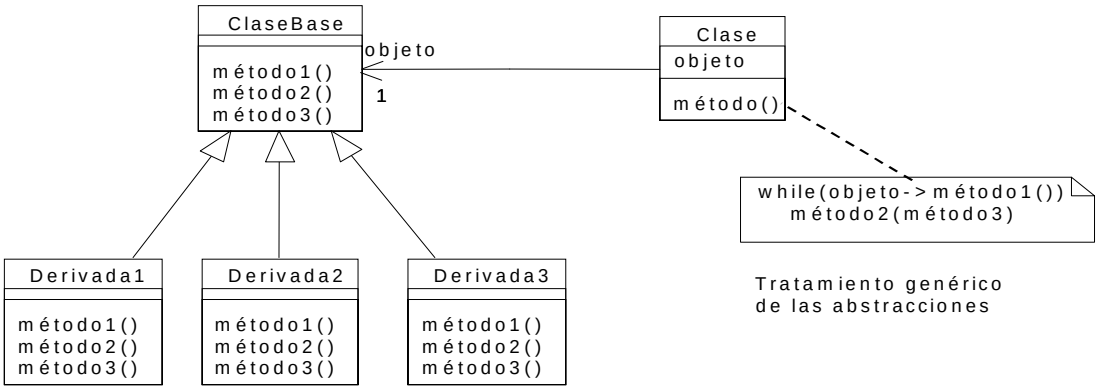


Figura 15.4: Tratamiento genérico de abstracciones mediante polimorfismo.

Para ampliar el funcionamiento de la aplicación mediante una nueva abstracción debemos:

1. Crear una nueva clase derivada de la abstracción genérica.
2. Implementar todos los métodos específicos de la nueva abstracción.
3. Crear explícitamente instancias de la nueva clase, cuando la aplicación lo requiera.

Ejemplos de este tipo de abstracciones, en el que una aplicación puede ser ampliada, son prácticamente infinitos; pueden variar desde elementos gráficos como menús, barras de herramientas, tipos de letra o cursores de ratón, hasta abstracciones propias del problema a resolver.

La limitación de la programación anterior reside en la necesidad de parar la ejecución de una aplicación, codificar la ampliación siguiendo los tres pasos descritos, compilarla y volver a ejecutarla. Mediante la reflectividad estructural y la interacción ofrecida entre aplicaciones, esto ya no es necesario.

Si hemos desarrollado una barra de herramientas implementando una interfaz bien definida que especifique su imagen, dimensiones y el comportamiento esperado a la hora de pulsar sobre ella, podemos ampliar dinámicamente la funcionalidad de esta aplicación –sin necesidad de parar su ejecución– siguiendo los siguientes pasos:

1. Implementamos la interfaz con el comportamiento y estructura del nuevo botón.
2. Mediante otra aplicación, lo insertamos en la lista de la barra de herramientas.

La aplicación será automáticamente adaptada en tiempo de ejecución. Este proceso puede aplicarse a cualquier abstracción, incluidas las propias del problema a resolver[85].

15.3.2 Gestión de Información Desconocida en Fase de Desarrollo

Las aplicaciones informáticas gestionan una información conocida en fase de desarrollo. Los sistemas gestores de bases de datos son aplicaciones encargadas de mantener

[85] Se ha desarrollado un ejemplo demostrativo en la distribución del prototipo descrito en el apéndice B. En el directorio “ControlProcess”, se encuentra un sistema de simulación de control de procesos automáticos en una planta. Cada vez que queramos añadir el control de un nuevo proceso a la planta, añadimos dinámicamente una implementación de la interfaz del proceso a la lista de procesos existente. El sistema reacciona automáticamente, distribuyendo su capacidad de cómputo entre todos los sistemas implantados.

esta información de un modo íntegro, seguro y eficiente. Sin embargo, el esquema de la información a almacenar ha de describirse previamente a la introducción de la misma.

La utilización de un sistema de persistencia de objetos dentro de nuestro entorno de programación [Ortín99b] y la característica de reflectividad estructural e introspección, hacen posible que nuestro sistema pueda adaptarse a nueva información aparecida en tiempo de ejecución.

A la hora de dar de alta un elemento de la base de datos, puede ser necesario, para ese objeto en particular, almacenar más información que para el resto sin que los nuevos campos hayan sido añadidos en tiempo de diseño. Mediante la utilización de un interfaz gráfico y haciendo uso de la reflectividad estructural, se podrá ampliar la información (atributos) para ese objeto en particular; el resto de los objetos no sufrirán este cambio [Yoder2001].

El posterior tratamiento de la información con determinadas modificaciones se puede llevar a cabo haciendo uso de la introspección de los objetos. Por ejemplo, las consultas se realizarán mediante los atributos comunes, pero la visualización y modificación se implementará sobre todos los atributos del objeto conocidos dinámicamente.

15.3.3 Patrones de Diseño Adaptables

Al tratarse nuestro sistema de un entorno computacional orientado a objetos, toma todos los conceptos propios de este tipo de sistemas añadiendo sus características específicas. Uno de los conceptos utilizados en el desarrollo de aplicaciones orientadas a objetos es el Patrón de Diseño [GOF94]: una solución arquitectónica de diseño para un determinado tipo de problema. Haciendo uso del concepto de reflectividad, pueden definirse patrones de diseño orientados a la flexibilidad dinámica de la solución ofertada.

Lejos de tratar de crear nuevos patrones de diseño en la utilización de las características reflectivas ofertadas por nuestro sistema, analizaremos posibles modificaciones a algún patrón conocido, que lo hagan más flexible. En concreto estudiaremos el caso del patón Decorador (*Decorator*) [GOF94].

El patrón decorador (*Decorator*) permite añadir nuevas responsabilidades o funcionalidades a abstracciones existentes. Si tenemos una clase `Archivo`, con sus primitivas de tratamiento (abrir, cerrar, leer y escribir), se puede derivar una clase de ésta que, haciendo uso de un agregado del tipo `Archivo`, pueda ofrecer adicionalmente servicios de compresión –manteniendo su interfaz.

Para ofrecer estos servicios, la aplicación deberá diseñarse con la siguiente jerarquía estática de clases:

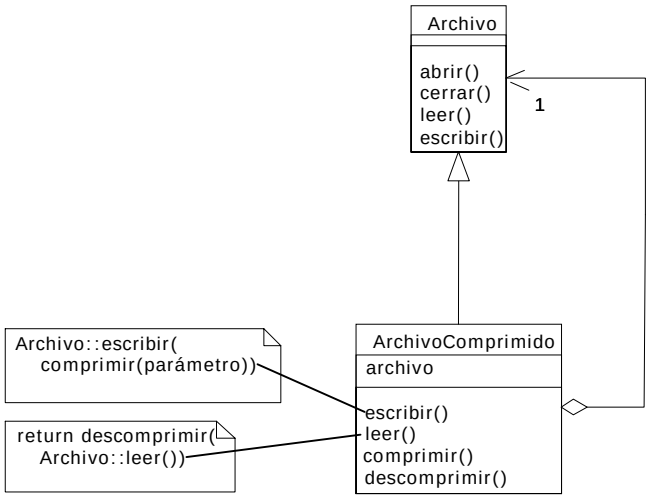


Figura 15.5: Utilización del patrón Decorador.

Una vez compilada y ejecutada esta aplicación, no se podrán añadir nuevas responsabilidades a la abstracción `Archivo` si no se modifica su código fuente. Mediante reflectividad estructural esta adaptación dinámica es posible substituyendo el objeto por otro que, haciendo uso del patrón Decorador, implemente su interfaz con responsabilidades añadidas –por ejemplo, el empleo de un buffer.

Haciendo uso de esta substitución dinámica, puede desarrollarse un decorador genérico que pueda añadir responsabilidades a cualquier abstracción, cualquiera que sea su interfaz. Por ejemplo, podemos diseñar un decorador que, mediante la invocación de un método `decorar` que reciba como parámetro el objeto a decorar, realice los siguientes pasos:

1. Cree un atributo con una referencia al parámetro pasado.
2. Vaya analizando todos los métodos del objeto asociado, creado en sí mismo tantos métodos como posea, obteniendo así la misma interfaz.
3. Utilice en cada uno de los métodos la funcionalidad propia del método homólogo de su asociado, añadiendo una determinada funcionalidad.

Un ejemplo puede ser la construcción de un objeto que añada funcionalidad de traza, mostrando un mensaje en cada invocación de los métodos del objeto asociado.

15.4 Aplicaciones Tolerantes a Fallos

Existen aplicaciones que son implantadas en plataformas susceptibles de poseer un elevado porcentaje de fallos (por ejemplo, las aplicaciones distribuidas por distintos nodos de Internet). Del mismo modo, existen aplicaciones críticas que tienen que dar una respuesta en tiempo real sin cometer jamás un error en dicha respuesta (como un sistema en tiempo real de tratamiento médico).

La validación y prueba de este tipo de sistemas puede pasar por una fase de ensayo basado en la simulación de entornos susceptibles de sufrir fallos. Las características reflectivas de nuestro sistema pueden ser empleadas para construir este tipo de entornos de simulación orientados a probar esta clasificación de aplicaciones. Por un lado se desarrolla la funcionalidad general de la aplicación y por otro, haciendo uso de la reflectividad dinámica,

se introducen situaciones computacionales simuladoras de fallos para observar la respuesta del sistema –separación dinámica de incumbencias.

Basándonos en sistemas reales, y a modo de ejemplo, mostraremos distintas aplicaciones de nuestro sistema en la simulación de fallos computacionales:

- Comunicaciones susceptibles a fallos. Uno de los ejemplos es el desarrollo de un entorno reflectivo basado en un MOP, que permite simular fallos en las comunicaciones mediante la modificación de la semántica del paso de mensajes [Killijian98] –reflectividad computacional. Permite analizar así, cómo se comporta una aplicación CORBA bajo los contextos simulados, sin necesidad de modificar el código propio de la aplicación.

- Recuperación de errores en sistemas concurrentes. Dada la resolución de un problema concurrente mediante un conjunto de hilos, se puede idear un sistema reflectivo tolerante a fallos que haga que cualquier hilo se recupere ante un error. El objetivo es que todos los hilos concluyan su ejecución satisfactoriamente para resolver el problema global.

  En [Kasbekar98], se define un *chekpoint* como un punto de estabilidad computacional de un hilo. Mediante reflectividad estructural, cuando un hilo alcanza un *checkpoint*, se puede salvar su estado en un almacenamiento persistente. El modo de computar un hilo (la semántica del lenguaje) se modifica mediante reflectividad computacional de modo que, si sucede un error en ejecución, se recupera el estado del *checkpoint* anterior para volver a intentar computar su código con éxito.

- Recuperación ante errores de sistemas complejos. Un sistema complejo que sea dividido en un conjunto de aplicaciones desplegadas sobre distintos entornos computacionales, debe analizarse de un modo global. Su tolerancia a los fallos debe estudiarse teniendo en cuenta cómo se comportaría ante el fallo de parte del sistema.

  Los estudios realizados en [Cibele98], añaden modificaciones aleatorias en el estado de los objetos (mediante reflectividad estructural), así como modificaciones aleatorias de su semántica (reflectividad computacional), simulando distintos tipos de errores en partes del sistema para analizar su comportamiento global.

Para los tres casos presentados, nuestro sistema puede emplearse como un entorno de simulación dinámica en el que pueden obtenerse empíricamente los grados de tolerancia a fallos de las aplicaciones, pudiendo incluso ofrecer informes de calidad de éstas, sin necesidad de incluir código adicional en la funcionalidad de las aplicaciones.

15.5 Procesamiento de Lenguajes

El hecho de poder conocer y modificar el estado de un sistema computacional, puede emplearse para facilitar la programación en sus lenguajes de programación; tanto para la depuración y mantenimiento de las aplicaciones, como para la generación dinámica de código y optimización en función de la propia aplicación.

15.5.1 Depuración y Programación Continua

A la hora de depurar una aplicación, muchos compiladores añaden código adicional a su código destino para que pueda ejecutarse por un depurador (*debugger*). Este proceso

permite ejecutar la aplicación paso a paso, conociendo los valores en memoria de las distintas variables. Sin embargo, la introducción de este código intruso de depuración hace que la aplicación pueda comportarse de distinto modo que aquélla no compilada en modo depuración –aplicaciones en tiempo real o multitarea pueden sufrir estas carencias.

La introspección es un mecanismo que es utilizado en entornos interpretados para conocer en todo momento el estado computacional de una aplicación, sin necesidad de inyectar en ésta código intruso de depuración. Dinámicamente, cualquier elemento de una aplicación puede analizarse y, mediante reflectividad estructural, modificarse su estructura, estado o comportamiento. Si queremos evaluar dinámicamente código, podemos crearlo, añadírselo a un objeto y evaluarlo (descosificarlo) para comprobar los resultados de su computación; si no es correcto, corregir los errores y repetir el proceso.

La principal ventaja es que estos pasos se pueden llevar a cabo dinámicamente, sin necesidad de separar dicotómicamente las fases de codificación (desarrollo) y ejecución (pruebas), propias del desarrollo de aplicaciones. Se analiza el funcionamiento de la aplicación, y se modifica y se examinan automáticamente la reacción a los cambios introducidos. Esta modificación de los objetos perdura en la aplicación, no es necesario volver a un entorno de desarrollo para salvar cambios y recompilar.

15.5.2 Generación de Código

Cuando se desarrolla una aplicación para compilar a distintas plataformas, el modo en el que se genera y optimiza el código destino es un factor crítico. Cada plataforma tiene su propio código nativo, y para cada tipo de entorno es necesario modificar el modo en el que se genera el código destino –optimización en velocidad, en tamaño o en utilización de recursos, son ejemplos reales. La mayoría de compiladores tiene un conjunto determinado de opciones para optimizar el código generado, pero carecen de un mecanismo genérico de especialización de éste.

Si se especifican y documentan todos los pasos de la generación de código de un compilador, mediante la utilización de reflectividad computacional estos pasos podrán modificarse, y será posible desarrollar un mecanismo de generación de código adaptable mediante un lenguaje, consiguiendo las siguientes funcionalidades:

1. Generación de código a cualquier plataforma.
2. Cualquier tipo de optimización que el usuario desee expresar mediante la modificación reflectiva de los pasos de invocación al generador de código.
3. Optimización en función de la aplicación. La codificación de la nueva semántica del generador de código puede utilizar introspección para analizar la estructura de la aplicación, llevando a cabo optimizaciones específicas para esa aplicación: optimización de código personalizada a la aplicación procesada.

Otra técnica genérica de compilación ampliamente utilizada en la ejecución de aplicaciones en entornos interpretados, es la compilación bajo demanda. Existen distintas variaciones para obtener mejores tiempos de ejecución de las aplicaciones generadas sobre plataformas virtuales:

- Compilación Dinámica (*Dynamic Compilation*): Se produce una compilación bajo demanda de una aplicación. En lugar de compilar todos los módulos de una aplicación, se compila el primero demandado en la ejecución del programa y, en función de las invocaciones de éste, se va demandando la generación de código de otros módulos [Chambers92]. El resultado obtenido es que la ejecución de la

aplicación comienza rápidamente, siendo ésta lenta en su evaluación, y acelerándose conforme pasa el tiempo.

- Compilación Dinámica Adaptable (*Adaptive Dynamic Compilation*) [Hölzle94]: Esta técnica es una modificación de la anterior. En tiempo de ejecución se realizan estudios probabilísticos de las fracciones de código más utilizadas (*Hot Spot*) y, una vez detectadas, se optimiza su código dinámicamente.

- Compiladores Nativos Bajo Demanda (JIT, *Just in Time Compilers*): Esta es una variación de los primeros en la que la generación de código se produce a la plataforma real específica utilizada en la ejecución de la aplicación [Yellin96]. Esta técnica ha sido utilizada en las plataformas comerciales de Java [Sun98] y .NET [Microsoft2000].

La reflectividad computacional es uno de los mecanismos más elegantes para implementar un compilador bajo demanda puesto que, sin necesidad de modificar la plataforma de interpretación, es posible compilar a la plataforma nativa los fragmentos de código que vayan siendo utilizados. La modificación de la semántica de creación de objetos, carga de clases y paso de mensajes, son computaciones que son adaptadas a la generación de código.

Si, utilizando reflectividad computacional, añadimos a la generación dinámica de código la posibilidad de modificar el módulo de generación de código, como mencionábamos al comienzo de este punto, obtendremos un compilador bajo demanda abierto (*OpenJIT*) [Matsuoka98]: compilador nativo bajo demanda, configurable para la generación de código específica de plataformas, recursos y aplicaciones, sin restricción alguna en su expresividad.

15.6 Adaptación de Lenguajes de Programación

La reflectividad de lenguaje ofrecida por nuestro sistema computacional reflectivo, permite adaptar cualquier lenguaje de programación, ya sea al problema a resolver o a características específicas de éste demandadas por el programador.

15.6.1 Adaptación al Problema

La capacidad de nuestro sistema de modificar las características de los lenguajes de programación seleccionados por el programador, hace que sea posible acercar el lenguaje al problema, eligiendo el nivel de abstracción adecuado para modelar un sistema de la forma más apropiada, y con una sintaxis y semántica fácilmente legibles.

Uno de los ejemplos típicos existentes representativos de la carencia de representatividad de lenguajes de programación, es la solución de problemas mediante patrones de diseño. La utilización de un patrón de diseño se traduce a múltiples líneas de código y clases que ocultan y dificultan el entendimiento de la solución, abordada mediante la utilización de un patrón. Esto suele dar lugar a un número de errores surgidos en el mantenimiento de la aplicación.

Sin perder la representatividad ni la sintaxis existentes del lenguaje de programación seleccionado, nuestro sistema permite amoldar éste a un determinado problema a resolver, previamente a la codificación de la aplicación existente.

Un ejemplo de esta utilidad queda patente en la utilización de OpenJava [Chiba98] –un sistema con reflectividad de lenguaje en tiempo de compilación– para adaptar el len-

guaje de programación Java a la resolución de problemas, mediante la utilización de los patrones de diseño Adaptador (*Adapter*) y Visitante (*Visitor*) [Tatsubori98]. En nuestro sistema, la aplicación elegiría su lenguaje y amoldaría éste al problema a resolver –véase el apéndice B.

15.6.2 Agregación de Características al Lenguaje

Nuestro sistema puede ser empleado para añadir de un modo sencillo nuevas características a lenguajes de programación conocidos. Si un determinado programador requiere facultades no ofrecidas por un lenguaje de programación, puede adaptar su especificación para que las contemple, aumentando así la productividad en el desarrollo de aplicaciones.

En [Kirby98] y [Chiba98b] se desarrollaron sistemas que permiten añadir a Java la posibilidad de desarrollar implementaciones genéricas respecto al tipo, de modo similar a las plantillas (*templates*) del lenguaje de programación C++.

15.7 Sistemas Operativos Adaptables

Del mismo modo que los sistemas operativos basados en micronúcleo estudiados en § 5.6, la separación del motor computacional (máquina abstracta) y los servicios a implementar (entorno de programación), permite adaptar los servicios del sistema operativo a los requisitos propios de la plataforma seleccionada.

Un paso adicional más cercano a nuestro sistema, es la implementación del sistema operativo Apertos [Yokote92b]: utiliza una máquina abstracta, y el concepto de reflectividad (un MOP) para conseguir flexibilidad, e identifica un metaobjeto por cada característica del operativo susceptible de ser adaptada.

Esta misma aproximación ha sido llevada a cabo en el desarrollo del sistema integral orientado a objetos Oviedo3 [Álvarez96]. Las distintas características pueden ser ofrecidas por un sistema operativo haciendo uso de reflectividad, siendo todas ellas adaptables dinámicamente:

- Planificación de Hilos: La modificación de la creación y sincronización de hilos mediante reflectividad computacional, permite adaptar los planificadores sin ningún tipo de restricción [Tajes2000].

- Seguridad: Distintos mecanismos y políticas de seguridad pueden implantarse mediante la adaptación de la semántica de los lenguajes de programación a distintos mecanismos de validación de usuarios [Díaz2000].

- Distribución: Modificando la creación y eliminación de objetos, así como el paso de mensajes, se puede desarrollar un sistema de distribución de objetos transparente al usuario y adaptable respecto a los distintos parámetros existentes en su intercomunicación [Álvarez2000].

- Persistencia: Al igual que para los lenguajes de programación (§ 15.1), esta faceta se puede ofrecer por el sistema operativo de un modo flexible [Martínez2001].

- Heterogeneidad y Acceso al Medio: El sistema operativo puede ofertar tanto características introspectivas, como acceso al medio –conocimiento de la plataforma física existente. De este modo, las aplicaciones podrán desplegarse en en-

tornos heterogéneos, pudiéndose conocer en todo momento las características de cada uno de ellos.

La consecución de estos objetivos en el sistema operativo de Oviedo3 es restringida por la utilización de un MOP. Sin embargo, la implantación de estos servicios en nuestro sistema se produce de un modo dinámico, no restrictivo (toda semántica puede ser modificada), con una expresividad elevada (un lenguaje –el de la máquina abstracta– es empleado para describir el meta-comportamiento) e independiente del lenguaje. Todo ello se consigue sin necesidad de modificar en ningún momento la implementación de la máquina virtual – la codificación se lleva a cabo en su propio lenguaje, sin dar lugar a distintas versiones del intérprete.

15.8 Adaptación y Configuración de Sistemas

La reflectividad computacional es utilizada genéricamente para modificar la semántica de un conjunto de operaciones. Si tomamos como ejemplo un lenguaje de programación, permite modificar cada una de las primitivas computacionales existentes para dicho lenguaje. Sin embargo, este proceso puede aplicarse de un modo genérico a cualquier tipo de aplicación o sistema computacional.

15.8.1 Gestión y Tratamiento del Conocimiento

Los motores de inferencia utilizados para la implementación de sistemas de ingeniería del conocimiento se basan en el concepto de regla. Pueden inferir o deducir el conocimiento a partir de unos hechos y unas reglas de inferencia. Sin embargo, la expresividad de la lógica obtenida puede en muchos casos no ser suficiente.

Nuestra plataforma puede utilizarse como mecanismo de tratamiento del conocimiento y simulación de la semántica del lenguaje natural. Mediante introspección se puede analizar dinámicamente las distintas propiedades de un objeto o entidad; utilizando reflectividad estructural se puede modificar su estructura y comportamiento, así como crear nuevas entidades del conocimiento; mediante reflectividad computacional puede modificarse la semántica de producción o reducción de reglas, teniendo en cuenta la utilización de una lógica más detallada y adaptable al contexto dinámico existente.

Como caso práctico, MetaPRL es un sistema de razonamiento que permite separar la representación del conocimiento, de la lógica empleada para inferir éste [Hickey2000]. La codificación de un módulo en MetaPRL viene dado por la representación de un conocimiento más la especificación de una lógica mediante la utilización de un metalenguaje.

15.8.2 Personalización de Aplicaciones

La mayoría de los sistemas informáticos que desean permitir su personalización a cada uno de los usuarios, permiten modificar de algún modo su funcionalidad. Así, es común que las barras de herramientas, menús o aceleradores de determinadas aplicaciones sean configurables. Elementos adicionales de configuración, pueden ser la utilización de un lenguaje de macros o *script* para automatizar un determinado número de tareas mediante la expresividad de un lenguaje de programación.

En ocasiones, la personalización y configuración de un sistema debe ser más flexible y otorgar una mayor expresividad en la modificación de su funcionalidad. Para este tipo

de entornos, la reflectividad otorgada por nuestro sistema puede convertirse en un mecanismo enfocado a lograr dicho objetivo.

Un sistema especificará de un modo abierto la semántica de las operaciones susceptibles de ser personalizadas. Mediante un lenguaje de programación, el usuario podrá adaptar éstas haciendo uso de la reflectividad computacional. La introspección ofrecerá un mecanismo para conocer dinámicamente la estructura del sistema. Poniendo como ejemplo un compilador, la derogación de las funcionalidades del generador de código puede significar la traducción a cualquier tipo de plataforma y la personalización en la optimización de código –como explicábamos en “Procesamiento de Lenguajes”, § 15.5.

Otro ejemplo es la simulación de máquinas de estados mediante el tratamiento de la información a través de eventos y acciones de un grafo representativo del flujo de la información [Sturman98]. Mediante un lenguaje se especifica el grafo y, haciendo uso de un lenguaje de meta-comportamiento, se expresan los eventos y las acciones relativos a cada arco del grafo en la simulación.

CAPÍTULO 16:
EVALUACIÓN DEL SISTEMA

El sistema presentado en esta tesis será evaluado en función de los requisitos establecidos con anterioridad, en el 2.1. 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.

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.

16.1 Comparativa de Sistemas

A la hora de comparar nuestro sistema con otros existentes, hemos seleccionado, del conjunto global de sistemas estudiados en esta tesis, los más próximos a satisfacer los requisitos estipulados. De esta forma, nos centraremos en:

- Smalltalk. Plataforma construida mediante el empleo de reflectividad estructural que define un lenguaje de programación orientada a objetos puro [Goldberg83].
- Self. Refinamiento y simplificación de la plataforma de Smalltalk, a la que se le añadió, en el proyecto Merlín [Assumpcao93], reflectividad computacional [Ungar87].
- Java. Plataforma y lenguaje orientado a objetos con características introspectivas y de modificación del paso de mensajes [Kramer96].
- MetaXa. Modificación de la máquina abstracta de Java para añadir un protocolo de metaobjetos en tiempo de ejecución [Kleinöder96] –con este sistema, evaluamos las características de la mayoría de los sistemas basados en MOPs, estudiados en § 7.4.1.
- nitrO. Este es el nombre con el que haremos referencia a nuestro sistema.

16.2 Criterios de Evaluación

La evaluación de los distintos sistemas es básicamente cualitativa. El objetivo principal de esta tesis es la consecución de una plataforma flexible. Distintos grados de flexibilidad han sido presentados y estudiados; en este momento será utilizados para evaluar los sistemas.

El conjunto de criterios es el descrito en el capítulo 2. Los clasificaremos y enumeraremos de la siguiente forma:

I. Criterios de la plataforma de computación.

   A. Sistema computacional multiplataforma. Independiente del hardware y del sistema operativo en el que se va a implantar.

   B. Independencia del lenguaje de programación. Su diseño no depende un lenguaje preestablecido.

   C. Independiente del problema. La plataforma ha de estar diseñada como una máquina computacional de propósito general.

   D. Tamaño y semántica computacional reducidos. La heterogeneidad en la implantación de la plataforma es más sencilla cuanto más reducidas sean las plataformas.

   E. Semántica operacional abierta. Existencia de un protocolo de ampliación de primitivas operacionales.

   F. Introspección y acceso al entorno. La plataforma deberá facilitar los parámetros propios del sistema físico en el que haya sido implantada.

   G. Semántica computacional abierta. Facilidad para modificar la semántica de la propia plataforma.

   H. Localización de las implementaciones nativas. La dependencia de la plataforma ha de estar separada de su implementación, y localizada en un único punto.

   I. Nivel de abstracción del modelo de computación. La elección de su modelo computacional requiere que éste sea sencillo, sin pérdida de expresividad, y con una traducción intuitiva desde la mayoría de los lenguajes existentes.

II. Criterios para la creación de un entorno de programación.

   A. Extensibilidad. La plataforma podrá extender su abstracción en su propio lenguaje de programación.

   B. Adaptabilidad. Las abstracciones creadas en el sistema, deberán poder ser adaptadas dinámicamente a requisitos surgidos en tiempo de ejecución.

   C. Identificación del entorno del entorno de programación. Cualquier aplicación deberá poseer la facultad de conocer los módulos software implantados localmente.

   D. Autodocumentación real. Todo el software deberá ser accesible dinámicamente para documentarse a sí mismo.

III. Criterios del grado de flexibilidad.

   A. Conocimiento dinámico del sistema. El sistema ha de ofrecer introspección en tiempo de ejecución.

B. Modificación estructural dinámica. Deberán existir primitivas de reflectividad estructural en tiempo de ejecución.

C. Modificación dinámica del comportamiento: reflectividad computacional en tiempo de ejecución.

D. Modificación computacional sin restricciones. Toda la semántica de los lenguajes de programación podrá ser adaptada, sin restricción alguna.

E. Modificación y selección dinámica del lenguaje. Reflectividad del lenguaje –aspectos léxicos y sintácticos.

F. Interoperabilidad directa entre aplicaciones. Cualquier aplicación podrá acceder a otra, independientemente del lenguaje de programación en el que haya sido codificada, y sin necesidad de un software intermedio.

IV. Criterios generales del conjunto del sistema.

A. Interacción con sistemas externos. Cualquier aplicación del sistema operativo hospedado, podrá acceder al sistema computacional, utilizando un mecanismo estándar.

B. Único sistema de programación. Debe ofrecerse transparencia entre los computadores que posean una máquina virtual, otorgando la ilusión de un supercomputador formado por ordenadores interconectados.

C. Flexibilidad dinámica no restrictiva. El modo en el que se pueda modificar el sistema deberá expresarse sin restricción alguna: un lenguaje propio, en lugar de un conjunto limitado de operaciones.

16.3 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 § 16.4.

Criterios de la Plataforma de Computación

Criterio	Smalltalk	Self	Java	MetaXa	nitrO
A.1	1	1	1	1	1
A.2	0	1	½	0	1
A.3	1	1	1	1	1
A.4	0,025	0,75	0,03	0,03	1
A.5	1	1	1	0	1
A.6	1	0	1	0	1
A.7	0	1	½	1	1
A.8	1	0	½	0	1
A.9	½	1	½	½	1
Total	5,525	6,75	6,03	3,53	9

Representación parcial de los criterios propios de la plataforma:

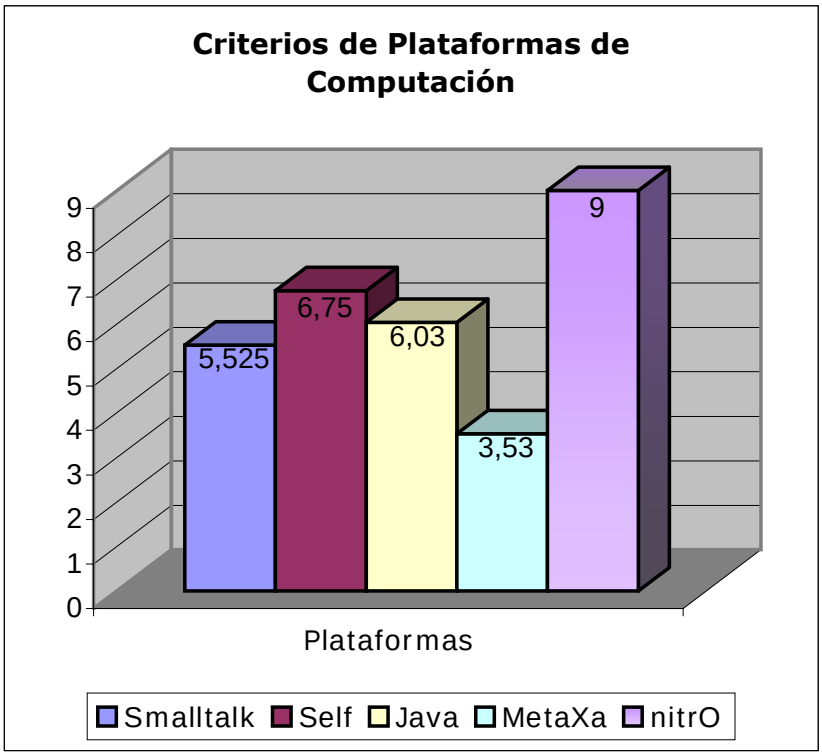


Figura 16.1: Evaluación de los criterios de la plataforma de computación.

Criterios requeridos para el desarrollo del entorno de programación

Criterio	Smalltalk	Self	Java	MetaXa	nitrO
B.1	1	1	½	1	1
B.2	1	1	0	1	1
B.3	1	1	1	1	1
B.4	1	1	½	0	1
Total	4	4	2	3	4

Representación parcial de los criterios necesarios para el desarrollo de un entorno de programación:

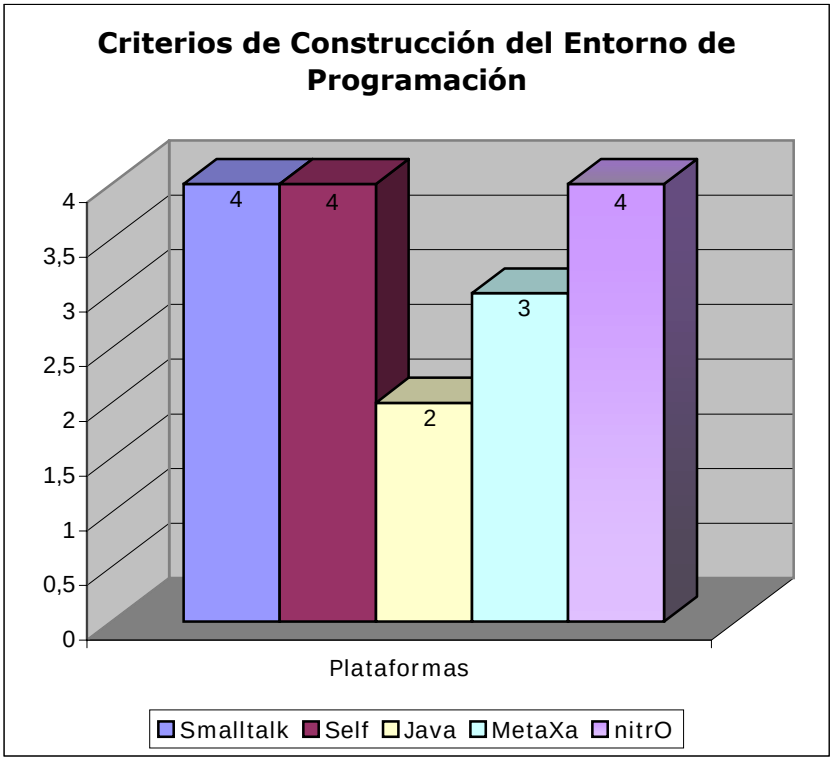


Figura 16.2: Evaluación de los criterios de construcción del entorno de programación.

Criterios concernientes al grado de flexibilidad

Criterio	Smalltalk	Self	Java	MetaXa	nitrO
C.1	1	1	1	1	1
C.2	1	1	0	1	1
C.3	0	1	½	1	1
C.4	0	0	0	0	1
C.5	0	0	0	0	1
C.6	1	1	0	0	1
Total	3	4	1,5	3	6

Representación parcial de los criterios que cuantifican el grado de flexibilidad del sistema:

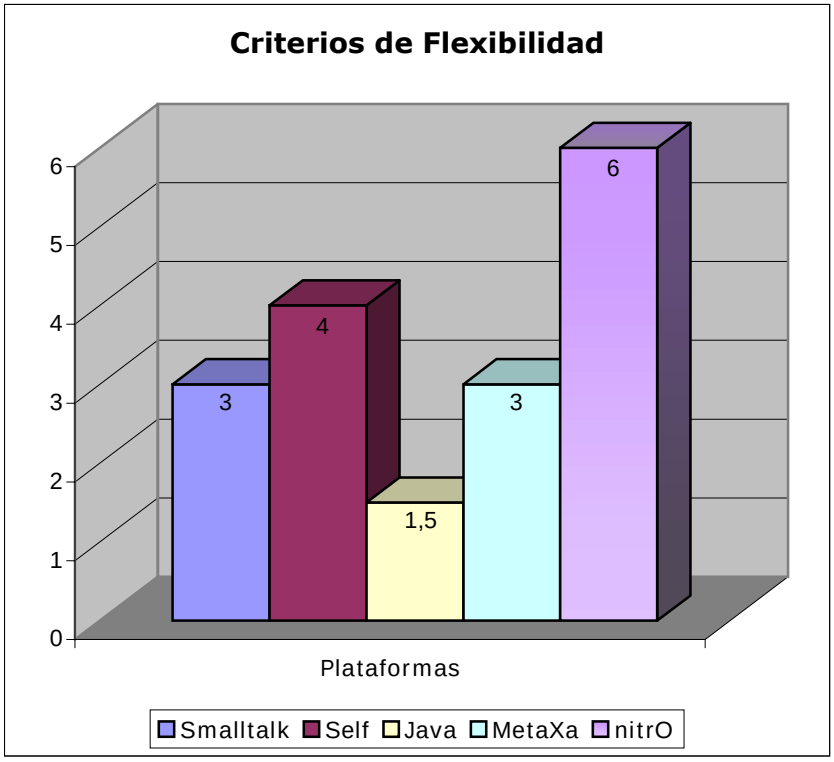


Figura 16.3: Evaluación de los criterios de flexibilidad.

Criterios de la Plataforma de Computación

Criterio	Smalltalk	Self	Java	MetaXa	nitrO
D.1	0	0	½	0	1
D.2	0	0	1	0	1
D.3	0	0	0	0	1
Total	0	0	1,5	0	3

Representación parcial de los criterios generales del sistema:

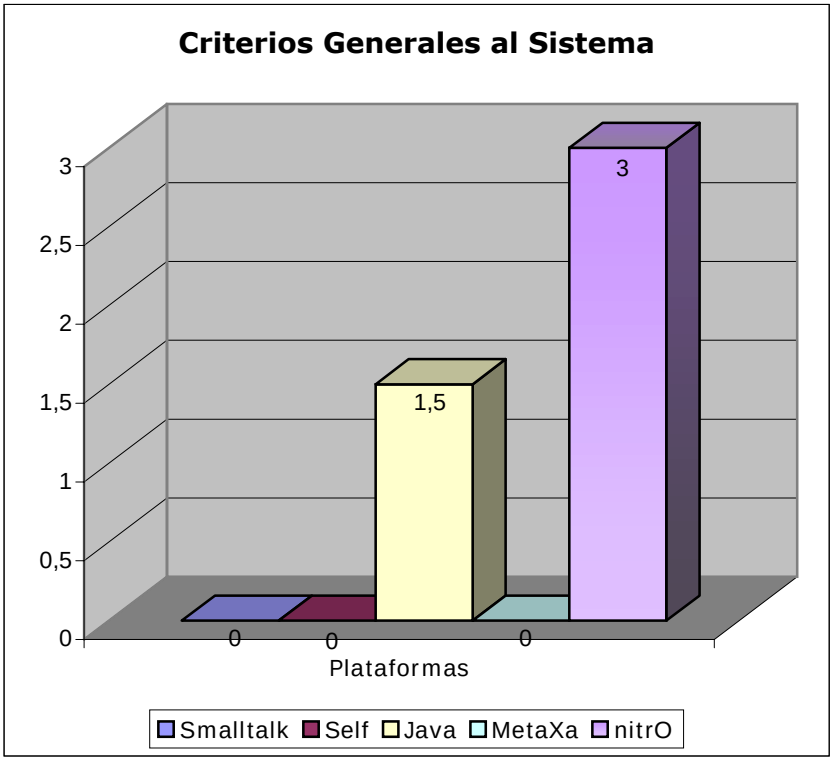


Figura 16.4: Evaluación de los criterios generales al sistema.

Evaluación global de todos los criterios

Grupo de Criterios	Smalltalk	Self	Java	MetaXa	nitrO
A	5,525	6,75	6,03	3,53	9
B	4	4	2	3	4
C	3	4	1,5	3	6
D	0	0	1,5	0	3
Total	12,525	14,75	11,03	9,53	22

Representación global de todos los requisitos del sistema, equiponderando todas las mediciones:

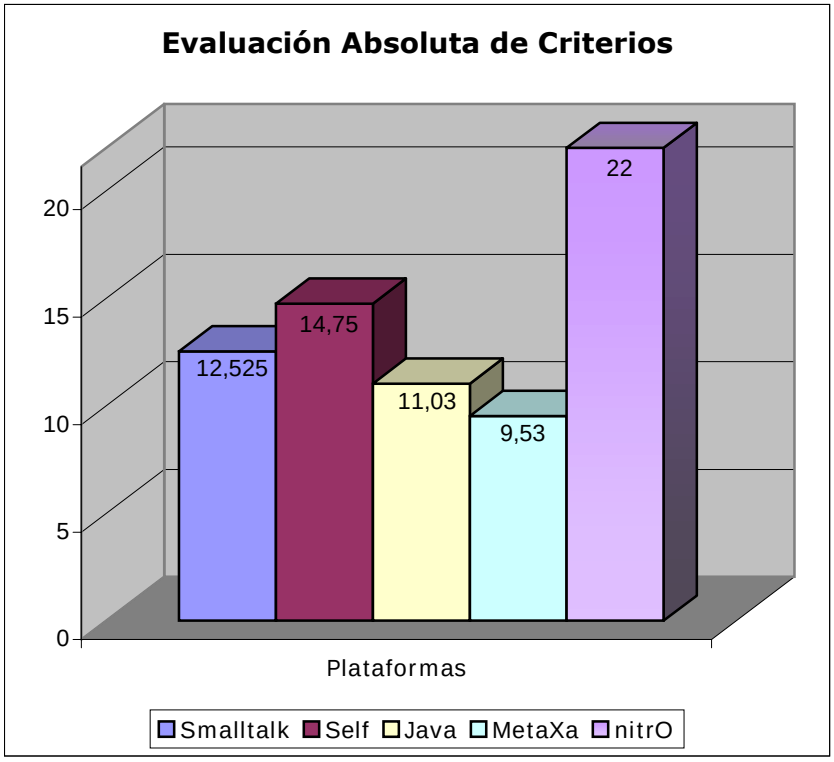


Figura 16.5: Evaluación de los criterios establecidos, de un modo absoluto.

16.4 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 4 para la evaluación de los criterios A y D, el capítulo 6 para las valoraciones de los puntos B y C, así como el capítulo 9 para las justificaciones relativas al sistema nitrO.

I. Evaluación de los criterios de la plataforma de computación.

A. Todas las plataformas definidas son abstractas e independientes de su implementación y plataforma en la que se implanten.

B. Tanto la máquina de Smalltalk como la modificación de la plataforma de Java para MetaXa, poseen fuertes dependencias del lenguaje de programación para el que fueron creadas –por ejemplo bloques (Smalltalk) o eventos de meta-comportamiento (MetaXa).

Java posee un modelo computacional basado en clases, con excepciones, miembros de clase y clases abstractas: un modelo computacional reutilizable pero diseñado para un determinado lenguaje de programación. Por otro lado, Self ha sido catalogado por determinados autores como la plataforma computacional base para representar cualquier lenguaje orientado a objetos [Wolczko96, Chambers91].

C. Aunque alguna de las plataformas haya sido creada para dar soporte computacional a un lenguaje de alto nivel, todos los lenguajes son de propósito general, sin estar orientados a un determinado tipo de problema predefinido.

D. El número de primitivas computacionales es 240 para Smalltalk, 8 para Self, 200 para Java y MetaXa y 6 para nitrO. Normalizando los valores entre cero y uno, obtenemos la evaluación mostrada en la tabla.

E. La plataforma de Smalltalk está definida sobre un conjunto de primitivas operacionales que el usuario puede ampliar; Self implementa sus primitivas operacionales mediante código ensamblador de la plataforma nativa; Java posee su interfaz de acceso nativo (JNI, Java Native Interface) [Sun97c]; MetaXa carece de implementación nativa; nitrO implementa estas operaciones con el protocolo de invocación del objeto `Extern`.

F. El conocimiento de acceso al sistema puede desarrollarse fácilmente en Smalltalk y nitrO con la adición de una primitiva operacional –en nitrO, ubicándola en el objeto `System`. Java ofrece la implementación de su clase `System`, con las propiedades de la plataforma real. Tanto MetaXa como Self carece de un mecanismo sencillo para obtener esta información.

G. Self (mediante sus *regions*, *reflectors*, *meta-spaces*), MetaXa y nitrO (mediante su MOPs) permiten modificar determinadas semánticas de su lenguaje como, por ejemplo, el paso de mensajes. Java, en su versión 2, permite modificar únicamente el paso de mensajes para cada clase [Sun99], pero no en su máquina abstracta –lo implementa en el lenguaje de alto nivel.

Smalltalk carece de esta posibilidad.

H. La plataforma Smalltalk ubica las operaciones nativas como implementaciones externas, asociadas a un número de primitiva. Este mismo proceso –mediante un identificador cadena de caracteres– lo desarrolla nuestra máquina en la invocación desde el objeto `Extern`.

La implementación de los métodos nativos de Java se enlaza con la carga de librerías de enlace dinámico, especificando el nombre de éstas y sin estar ubicadas todas en una misma posición predeterminada.

El código nativo de Self está desarrollado dentro de la implementación de la máquina –en código ensamblador. MetaXa no permite ampliar sus primitivas.

I. Smalltalk, Java y MetaXa utilizan un modelo de objetos orientado a representar, en bajo nivel, las abstracciones de su lenguaje de programación asociado. Sin embargo, nitrO y Self siguen el modelo basado en prototipos (capítulo 8), capaz de representar las mismas abstracciones, con la máxima sencillez computacional [Ungar91].

II. Evaluación de los criterios para la creación de un entorno de programación.

A. Todas las plataformas, excepto Java, ofrecen introspección y reflectividad computacional para extender dinámicamente las abstracciones del sistema. Java permite codificar, mediante introspección, de forma adaptable a requisitos previstos en tiempo de diseño, pero, al no poseer reflectividad estructural, no permite extenderse de un modo genérico sin finalizar su ejecución.

B. Para poder adaptarse dinámicamente es necesario que la plataforma permita modificar la estructura –reflectividad estructural– y añadir comportamiento –cosificación– en tiempo de ejecución. A excepción de Ja-

va, que carece de la segunda característica, el resto cumplen los requisitos.

C. Todas las plataformas poseen la introspección necesaria conocer los módulos software implantados localmente.

D. El tratamiento de la computación como datos y la reflectividad estructural, hacen que Smalltalk, Self y nitrO sean plataformas autodocumentadas dinámicamente.

Java, mediante su introspección y tratamiento de código fuente, ofrece la herramienta `javadoc` que, simulando el objetivo buscado, se limita a documentar a través del código compilado –no dinámicamente. MetaXa carece de esta posibilidad.

III. Evaluación de los criterios del grado de flexibilidad.

A. Todos los sistemas ofrecen introspección en tiempo de ejecución.

B. Todos, excepto Java, permiten modificar dinámicamente las estructuras de sus objetos.

C. Java ofrece un mecanismo limitado de modificación del paso de mensajes para cada clase. Smalltalk no ofrece esta posibilidad. El resto de sistemas permiten, mediante distintos mecanismos, modificar parte de su semántica.

D. Los sistemas que permiten modificar su semántica, ofrecen esta posibilidad mediante la utilización de un protocolo (MOP) que restringe el número de abstracciones a configurar. nitrO aporta un sistema reflectivo carente de estas restricciones –véase el capítulo 11.

E. Todos los sistemas, excepto nitrO, son dependientes de un lenguaje: no ofrecen un mecanismo de modificación dinámica de su propio lenguaje –reflectividad del lenguaje.

F. Las aplicaciones dentro del sistema Self, Smalltalk y nitrO, son ejecutadas por la misma máquina abstracta, interactuando entre sí como si se tratasen de la misma aplicación. En los sistemas basados en Java, cada aplicación utiliza la ejecución de una máquina distinta, requiriéndose una capa software adicional para su interconexión.

IV. Evaluación de los criterios generales del conjunto del sistema.

A. Self, Smalltalk y MetaXa son sistemas computacionales cerrados sin interacción con las aplicaciones del sistema operativo utilizado.

Java implementa un mecanismo de acceso (mediante RMI [Sun97b] y el protocolo IIOP de CORBA [OMG95]). Adicionalmente, existen implementaciones, como la de Visual Java de Microsoft, que ofrecen un puente con un sistema de componentes, como COM [Microsoft95].

nitrO especifica una interfaz de acceso, a implementar, utilizando un mecanismo estándar, en función del sistema operativo utilizado.

B. Ninguno de los sistemas ofrecen la posibilidad de ver distintas plataformas interconectadas como un único sistema de programación. Sin embargo, en la plataforma Java se han desarrollado sistemas de agentes móviles (los *aglets* [Mitsuru97]), al igual que el sistema de distribución de nitrO (**capítulo 13**).

C. nitrO es el único sistema que ofrece la expresividad propia de un lenguaje, para modificar cualquier característica del sistema. La utilización de un MOP, limita las operaciones a modificar por el conjunto de mensajes que implementa un metaobjeto [Ortín2000].

16.5 Conclusiones

16.5.1 Evaluación de las Plataformas

Comenzando con los resultados en la evaluación de las plataformas, observamos en la Figura 16.1 cómo MetaXa es el sistema más desfavorecido. El principal problema de esta plataforma es que ha sido diseñada para ofrecer reflectividad computacional en tiempo de ejecución, descuidando determinadas facetas propias de una máquina abstracta –como por ejemplo la posibilidad de incluir operaciones primitivas. El resultado es que MetaXa utiliza el concepto de máquina abstracta para simplificar el mecanismo reflectivo implementado, y no para definir un modelo computacional de un sistema.

El resto de las plataformas ofrecen un buen modelo computacional, válido para constituir la base de un sistema; nitrO ha sido diseñado reduciendo el número de primitivas computacionales y separando de forma clara la implementación de sus operaciones dependientes de la plataforma física.

16.5.2 Evaluación de los Criterios para Construir un Entorno de Programación

Para cumplir los objetivos de este grupo de requisitos, los dos sistemas que utilizan la plataforma Java son los más perjudicados. Su principal limitación es que no permiten obtener la computación como información, es decir, carecen de un mecanismo de cosificación. Hemos visto en el **capítulo 13**, cómo esta faceta puede utilizarse para desarrollar un entorno de programación en el propio lenguaje de la plataforma, sin necesidad de modificar la máquina virtual. De forma adicional, Java –no MetaXa– carece de reflectividad estructural, también empleada para obtener el objetivo que estamos evaluando.

La inclusión de introspección, reflectividad estructural y cosificación en las plataformas Self, Smalltalk y nitrO, las hacen óptimas para construir sobre ellas un entorno de programación que extienda, de modo adaptable, el nivel de abstracción inicial del sistema computacional.

16.5.3 Evaluación del Grado de Flexibilidad

En esta evaluación, quedan patentes las pretensiones comerciales de cada una de las plataformas. Como hemos comentado en el § 6.4, la flexibilidad de una plataforma puede conllevar a determinados problemas de seguridad en el sistema[86]. De este modo, la plataforma Java limita sus operaciones relativas a su adaptabilidad a aquéllas que sean consideradas como útiles y no inseguras.

El resto de plataformas ofrecen grados de flexibilidad más avanzados: Smalltalk ofrece interacción directa entre aplicaciones; MetaXa carece de esta posibilidad, pero ofrece

[86] Por ejemplo, el hecho de que una aplicación pueda cambiar la semántica de otra que se esté ejecutando en la misma plataforma, puede interpretarse como un problema de seguridad del sistema.

reflectividad de comportamiento en tiempo de ejecución –cuando ésta no la facilita Smalltalk; la plataforma Self ofrece ambas facultades.

En esta valoración, MetaXa no refleja fielmente –con su evaluación de tres puntos– su grado de flexibilidad. El hecho equiponderar cada criterio, hace que posea igual valoración que Smalltalk, cuando debiese superarla –Smalltalk no posee reflectividad computacional y MetaXa sí. El resultado de esta valoración ha de interpretarse como: "MetaXa posee mayor flexibilidad computacional que Smalltalk, pero carece de la posibilidad de que una aplicación modifique a otra distinta, cuando Smalltalk sí ofrece esta aptitud –aunque la limite a la estructura de las aplicaciones, sin poder adaptar su semántica".

nitrO ofrece un grado de flexibilidad no otorgado por ningún sistema reflectivo [Ortín2001] –véase el capítulo 11.

16.5.4 Evaluación Global del Sistema

En este punto se valora el sistema computacional desde el concepto de caja negra, requiriendo que sea abierto a aplicaciones nativas reales, existentes en cualquier plataforma, que ofrezca la abstracción de un único supercomputador, y que sea flexible.

Self, Smalltalk y MetaXa son sistemas totalmente cerrados, cuya interacción con el mundo exterior es compleja y dependiente del sistema. No ofrecen un mecanismo estándar de interconexión entre plataformas y no ofrecen al programador la ilusión de sistema único. Ninguno ofrece una flexibilidad expresada mediante un lenguaje.

Sobre Java se han desarrollado sistemas de agentes móviles [Mitsuru97] que ofrecen la ilusión de supercomputador distribuido, además de permitir la utilización de distintos protocolos estándar e implementaciones de la máquina virtual, para interconectarse con otro tipo de sistemas.

nitrO especifica una interfaz de interconexión con aplicaciones nativas –dependiente del sistema operativo utilizado–, implementa un sistema de distribución que permite el acceso a objetos remotos y su movimiento entre distintas máquinas, y ofrece una flexibilidad expresada en un lenguaje –el de su máquina abstracta– que permite modificar dinámicamente la estructura, comportamiento y lenguajes del sistema.

16.5.5 Evaluación Absoluta

En la evaluación absoluta vemos como MetaXa, ofreciendo un protocolo de modificación dinámica de su comportamiento, resulta la peor valorada. Dicha evaluación refleja las carencias de diseño relativas a establecerse como una plataforma base de un sistema computacional. El concepto de máquina abstracta se utiliza para implementar de un modo sencillo un MOP, y no para obtener las ventajas propias de la utilización de una plataforma abstracta (§ 3.4).

Java se ha constituido como la demostración empírica de que las plataformas virtuales pueden ser utilizadas en el desarrollo de aplicaciones –otros sistemas como .net han surgido posteriormente. Java diseña su plataforma de un modo abierto e independiente del sistema operativo y lenguaje. Posee determinadas facultades de flexibilidad, pero limitándolas por cuestiones de seguridad.

Smalltalk desarrolla una plataforma extensible y adaptable dinámicamente, al poseer primitivas de reflectividad computacional y tratamiento de computación como datos –mediante el concepto de bloque. Self toma estas características y las mejora, incluyendo

modificación de su comportamiento (reflectividad computacional), y reduciendo su modelo computacional (basándolo en prototipos).

La máquina abstracta de nitrO se ha diseñado con criterios similares a los de Self, reduciendo aún más su computación, permitiendo su ampliación operacional, y abriendo su computación a aplicaciones nativas. De forma adicional, el sistema nitrO ofrece un mecanismo de flexibilidad superior al de cualquier sistema, siendo éste totalmente independiente del lenguaje de programación a utilizar.

CAPÍTULO 17:
CONCLUSIONES

A lo largo de esta tesis hemos analizado las distintas alternativas para crear un sistema computacional de programación heterogéneo, en el que la flexibilidad dinámica constituye su primer objetivo, facilitando la interacción de todas sus aplicaciones –independientemente del lenguaje y plataforma en el que fueron creadas–, y extendiendo su modelo computacional de un modo adaptable sin restricción alguna.

Comenzamos estudiando en el capítulo 3 las ventajas obtenidas en la utilización de máquinas abstractas. Tras estudiar exhaustivamente un conjunto representativo de casos reales, concluimos cómo la mayor parte de ellas fueron diseñadas para establecerse como plataforma computacional de un determinado lenguaje de programación o sistema operativo, sin tratar de ofrecer un motor computacional genérico.

El tamaño reducido de la plataforma, así como un mecanismo de extensibilidad de su nivel de abstracción, son características fundamentales a la hora de implantarla en entornos heterogéneos–utilizando su propio lenguaje de programación, sin necesidad de modificar su implementación. La mayoría de los sistemas estudiados en el capítulo 4 no tienen en consideración estas restricciones.

La principal carencia de las plataformas más conocidas es su flexibilidad para adaptar sus características dinámicamente: utilizan un lenguaje de semántica y especificación invariables. Para solventar esta limitación se estudió, en el capítulo 5, un conjunto de técnicas de construcción de sistemas flexibles, y, en el capítulo 7, 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 (MOPs) que no limite el número de primitivas semánticas a modificar y el modo en el que el sistema pueda ser adaptado.

En este capítulo, estudiaremos cómo el diseño de nuestro sistema ha superado todas las limitaciones puntualizadas, satisfaciendo los requisitos preestablecidos a lo largo de esta tesis –capítulo 2. Analizaremos las principales ventajas aportadas, y estableceremos las futuras líneas de investigación a seguir.

17.1 Sistema Diseñado

El sistema diseñado (capítulo 9) se basa en la especificación de una máquina abstracta como motor computacional de todo el sistema. Sobre ésta se desarrolla un entorno de programación que demuestre su capacidad de extenderse dinámicamente y de forma adaptable, utilizando únicamente su lenguaje de programación. Finalmente, se construye un sistema reflectivo sin restricciones, que ofrece al programador una flexibilidad no restrictiva, independientemente del lenguaje que desee utilizar.

17.1.1 Máquina Abstracta

La máquina abstracta (capítulo 12) ha sido diseñada siguiendo una serie de criterios, representado éstos las carencias de la mayoría de los sistemas estudiados.

El diseño de la máquina no ha sido llevado a cabo para ejecutar un determinado lenguaje de programación, ni para desarrollar un sistema operativo multiplataforma; no sufre estas dependencias y utiliza un modelo computacional adecuado (capítulo 8):

- Es sencillo, para que su implementación sea reducida y pueda implantarse en entornos computacionales heterogéneos.
- No sufre de pérdida de expresividad frente a otros modelos, como por ejemplo el basado en clases.
- Existe una traducción directa e intuitiva entre los distintos modelos computacionales y el utilizado.

Tanto su semántica operacional como las primitivas dependientes de la plataforma real, pueden ser implantadas sin necesidad de modificar la implementación de la máquina virtual. Su acceso se realiza mediante un mecanismo estándar de invocación a rutinas dependiente del sistema operativo utilizado, y están ubicadas en un único punto.

Su flexibilidad ofrece el conocimiento de la plataforma real utilizada, y la estructura de las aplicaciones y objetos existentes dinámicamente. También otorga la posibilidad de modificar dinámicamente la estructura, el estado y el comportamiento de todos sus objetos, así como configurar dinámicamente la semántica de un conjunto de sus primitivas computacionales.

La plataforma posee una interfaz de acceso, permitiendo a las aplicaciones nativas, codificadas en cualquier lenguaje y sobre cualquier sistema operativo, la utilización y programación del sistema computacional creado.

17.1.2 Entorno de Programación

Como demostración de la extensibilidad y adaptabilidad de la plataforma diseñada, se ha creado un entorno de programación que aumenta su nivel de abstracción básico (capítulo 13). El desarrollo de estas funcionalidades se ha llevado a cabo sobre el propio lenguaje de la máquina abstracta, eliminando la necesidad de modificar su implementación –y sin dar lugar así a la consecuente pérdida de portabilidad del código existente.

Mediante la utilización de introspección, reflectividad estructural y reflectividad computacional, se han construido sistemas de recolección de basura, persistencia, distribución de objetos y planificación de hilos. Todos ellos son adaptables dinámicamente: si se desea modificar su funcionamiento, éste puede ser llevado a cabo sin necesidad de parar la ejecución del sistema.

Las abstracciones ofrecidas suponen un conjunto de código reutilizable por cualquier aplicación del sistema, independientemente del lenguaje de programación en el que hayan sido codificadas –el único modelo computacional existente en el sistema, es el ofrecido por la máquina abstracta.

Todo el sistema desarrollado puede consultarse dinámicamente por cualquier aplicación, conociendo los distintos sistemas y módulos implantados en cada plataforma, así como la auto-documentación obtenida dinámicamente. Esto permite su implantación modular en entornos heterogéneos, dependiendo de las necesidades específicas de cada uno.

17.1.3 Sistema Reflectivo No Restrictivo

Para obtener independencia del lenguaje de programación y flexibilidad computacional dinámica sin restricciones, se desarrolla la tercera capa del sistema (capítulo 11).

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 de la máquina abstracta.

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 de la máquina abstracta 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.

17.2 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.

17.2.1 Heterogeneidad de la Plataforma

El diseño de la máquina abstracta (capítulo 12) ha sido llevado a cabo teniendo en cuenta que deberá poder ser implantada en el mayor número de sistemas computacionales posible. Para cumplir este objetivo, su arquitectura fue diseñada reduciendo al mínimo su semántica computacional, pero ofreciendo distintos mecanismos de extensibilidad.

La implementación de sus primitivas operacionales y de las funcionalidades de acceso a la plataforma utilizada, es ubicada de forma externa a la máquina virtual, y pueden ampliarse en función de las restricciones del sistema computacional, según sus requisitos específicos.

Mediante la introspección ofrecida, las aplicaciones serán capaces de conocer los distintos módulos existentes en cada plataforma y ejecutarse en función de esta información (por ejemplo, antes de utilizar el sistema de persistencia, consultar su existencia), permitiendo variar su evaluación en distintos entornos heterogéneos (no hacer objetos persistentes en la ejecución, o utilizar un sistema de persistencia remoto, apoyándose en el sistema de distribución).

17.2.2 Extensibilidad

La plataforma permite ampliar sus primitivas computacionales limitadas, ofreciendo un mayor nivel de abstracción en el conjunto del sistema. Esta característica es desarrollada haciendo uso de su propio lenguaje de programación, sin necesidad de modificar la máquina abstracta.

Si se modificase la versión de la máquina (como pasa con la especificación de la máquina de Java) se perdería la posibilidad de ejecutar código de versiones anteriores (código obsoleto, *deprecated*), no existiría compatibilidad con otras máquinas virtuales interconectadas entre sí (a la hora de mover código), y habría que actualizar todos los programas que pudiesen interpretar código de nuestra plataforma (como por ejemplo, un navegador de Internet).

Haciendo uso del mecanismo de extensibilidad ofrecido, el desarrollo de aplicaciones –por complejas que éstas sean– requiere únicamente su código fuente, cualquiera que sea su implementación y plataforma utilizada.

Ejemplos de abstracciones creadas con el mecanismo de extensibilidad ofrecido, expresadas en su propio lenguaje, son: definición de una lógica y sus operaciones, comparación de objetos, iteración y tratamiento genérico de miembros, aritmética numérica, creación de abstracciones e instanciación, recolección de basura, persistencia y distribución de objetos y planificación controlada de hilos.

17.2.3 Adaptabilidad

La adaptabilidad de nuestro sistema computacional y de las aplicaciones desarrolladas sobre éste queda patente en distintos niveles:

- Acceso al entorno. Como hemos comentado en § 17.2.1, las aplicaciones de nuestro sistema pueden analizar el entorno computacional existente (introspección), ejecutándose en función de la información obtenida (supone un marco ideal para desarrollar un sistema de agentes, § 15.1).

- Los distintos servicios ofrecidos por el entorno de programación, desarrollado sobre la plataforma base (capítulo 13), son adaptables dinámicamente, sin necesidad de finalizar la ejecución de ninguna aplicación –las existentes se amoldan a los cambios establecidos.

  Por ejemplo, si, en tiempo de ejecución, creamos un programa que modifique el algoritmo de recolección de basura y la política de planificación de hilos, todas las aplicaciones del sistema se adaptarán a la nueva configuración.

- El diseño de las características del lenguaje de programación, creadas extendiendo las primitivas computacionales de la máquina, es adaptable. Para un determinado entorno, puede interesar utilizar un lenguaje con características como herencia múltiple o tratamiento de excepciones. Mediante las técnicas de flexibi-

lidad ofrecidas por la plataforma, estas abstracciones pueden crearse en su lenguaje de programación, extendiendo las abstracciones iniciales [Kiczales91].

- Separación de incumbencias y aspectos. El sistema permite desarrollar una aplicación, separando su funcionalidad de los distintos aspectos ortogonales que sobre ella se puedan definir.

  Eligiendo el lenguaje de programación deseado, se desarrolla y ejecuta una aplicación. Haciendo uso de la reflectividad computacional (capítulo 11), se accede a ésta y se le asignan dinámicamente aspectos como el de persistencia. La adaptabilidad otorgada se consigue modificando la semántica del lenguaje utilizado.

17.2.4 Modelo Computacional Único e Independiente del Lenguaje

El diseño del sistema se ha llevado a cabo utilizando una traducción de cualquier lenguaje (únicamente ha de existir su especificación) al modelo computacional ofrecido por la máquina abstracta. La existencia de un único modelo computacional hace que todas las aplicaciones puedan interaccionar entre sí, independientemente del lenguaje utilizado para su desarrollo.

La interconexión de aplicaciones se produce de un modo directo. La evaluación de un programa no da lugar a una nueva ejecución de la máquina abstracta, sino que todas se sitúan en el mismo espacio de direcciones de la existente. De este modo, el acceso –y por tanto, utilizando reflectividad, la adaptación– de una aplicación a otra sucede como si fuese la misma, cualquiera que sea el lenguaje en que cada una de ellas haya sido implementada.

El desarrollo de un interprete genérico independiente del lenguaje, hace que el sistema esté preparado para ejecutar una aplicación, sin conocer previamente el lenguaje de programación que ésta utilice; la especificación del lenguaje puede anteponerse al código del propio programa (apéndice B).

De igual modo, mediante el uso del sistema de distribución, la especificación del lenguaje puede moverse junto a su programa –al tratarse de objetos– a cualquier plataforma, para que ésta pueda evaluar la aplicación sin conocer previamente su lenguaje.

17.2.5 Flexibilidad

En los sistemas reflectivos estudiados (capítulo 7), el mayor grado de flexibilidad dinámica ofrecido es la modificación de la estructura de las aplicaciones y parte de la semántica del sistema –comúnmente, mediante el empleo de MOPs. Existen sistemas que permiten adaptar características del lenguaje, pero únicamente en fase de compilación –no una vez ésta haya sido ejecutada.

El sistema presentado en esta tesis permite, en tiempo de ejecución, acceder y cambiar la estructura de toda aplicación, adaptar dinámicamente cualquier característica de los lenguajes de programación utilizados (aspectos léxicos, sintácticos y semánticos) y ofrece, además, un mecanismo de modificación semántica no restrictivo (capítulo 11).

En la utilización de MOPs, todo comportamiento susceptible de modificarse dinámicamente debe ofrecerse mediante la abstracción de metaobjeto [Kiczales91]; sus métodos representan las características y el modo en el que pueden adaptarse. Nuestro sistema no sufre estas limitaciones:

1. Cualquier característica semántica del lenguaje, al ser ésta totalmente accesible, puede ser modificada dinámicamente por el usuario. No existe, pues, la restricción previa de ofrecerla mediante un metaobjeto.
2. El modo en el que esta abstracción será adaptada no es restringido por la ejecución de un conjunto de métodos, sino que poseemos la expresividad completa del lenguaje de la máquina abstracta para configurar su funcionamiento [Ortín2001].

17.3 Futuras Líneas de Investigación y Trabajo

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:

17.3.1 Aplicación a Sistemas Computacionales

Como mencionamos en el capítulo 15, las aportaciones de esta tesis pueden ser aplicadas a numerosos casos dentro del mundo de la computación. De modo general, la elevada flexibilidad ofrecida es interesante en el desarrollo de sistemas que deban adaptarse a requisitos surgidos dinámicamente, sin que pueda finalizar su ejecución [Foote90].

En el capítulo 15 se muestra un estudio en profundidad de las posibilidades de nuestro sistema. Dentro de los casos descritos en dicho capítulo, los que aquí presentamos son los sistemas a desarrollar de un modo venidero, encontrándose actualmente alguno de ellos en desarrollo.

17.3.1.1 Sistema de Persistencia Implícita

La persistencia es uno de los campos en los que más se ha utilizado los conceptos de introspección y reflectividad estructural. Las plataformas de Java, Smalltalk y Python, por ejemplo, utilizan estas características para hacer que sus objetos persistan a la ejecución de la aplicación que los creó. El elevado grado de flexibilidad ofrecido por nuestro sistema, que no poseen otros, puede ser utilizado adicionalmente en el desarrollo de un sistema de persistencia implícito [Ortín99d].

- La introspección es utilizada para el conocimiento dinámico del estado y comportamiento del objeto a almacenar –mediante cosificación, el comportamiento es mera información.
- La reflectividad estructural es utilizada para crear dinámicamente un objeto a raíz de su representación en un sistema de persistencia.
- La reflectividad computacional es empleada para establecer dinámicamente los distintos aspectos relativos a la persistencia de las aplicaciones. Sin modificar el código de la aplicación, pueden adaptarse dinámicamente los mecanismos de indexación y la frecuencia de actualización a disco. Pueden desarrollarse rutinas que analicen la estructura de las aplicaciones, añadiendo el mecanismo de indexación más apropiado [Martínez98b], o que reduzcan la frecuencia de las actualizaciones cuando la carga computacional del sistema sea elevada.

  El sistema puede constituirse como un entorno de pruebas de análisis de los distintos parámetros propios de un sistema de persistencia –estudio de su compor-

tamiento dinámico, cambiando sus parámetros en función de los contextos de ejecución.

17.3.1.2 Entorno de Distribución de Agentes

La creación de un sistema de agentes distribuidos puede llevarse a cabo en nuestra plataforma ofreciendo características adicionales a determinados sistemas existentes:

- La utilización de una plataforma virtual hace que el código de las aplicaciones pueda ejecutarse en entornos computacionales heterogéneos.
- La introspección del sistema ofrece a los agentes la capacidad de consultar las abstracciones existentes en cada plataforma, así como analizar cada una de ellas.
- La reflectividad estructural facilita la construcción de rutinas que otorguen la migración y replicación de objetos a través de una red de ordenadores.
- La independencia del lenguaje implica la posibilidad de crear agentes en cualquier lenguaje de programación –incluso en uno inventando por el propio programador.

  Las rutinas de movilidad de objetos son desarrolladas y ofrecidas por el sistema en el modelo computacional de la máquina abstracta. Éstas pueden ser utilizadas por cualquier aplicación, independientemente del lenguaje utilizado –toda aplicación es traducida dinámicamente al modelo computacional de la máquina, y ésta permite la interacción directa entre aplicaciones.

  Si una vez que la aplicación haya sido movida a otra plataforma, se detecta –mediante el uso de introspección– que su lenguaje no está presente en la nueva máquina, la especificación del lenguaje –formada por una estructura de objetos– podrá replicarse al sistema remoto, para que pueda ejecutarse el agente codificado en el lenguaje anteriormente desconocido.
- La reflectividad computacional podrá emplearse para separar la funcionalidad de la aplicación de su aspecto relativo a su movilidad. Una aplicación, sin modificar su implementación, podrá ser llevada dinámicamente a otras plataformas sin que su código así lo especifique.

17.3.1.3 Semántica del Lenguaje Natural

Un amplio campo a investigar es la representación computacional del lenguaje natural. Una vez procesado éste, se trata de representar la semántica propia del humano, en un sistema computacional de conocimiento que represente la misma semántica [Link98].

- Mediante el uso de introspección, se puede analizar la estructura y comportamiento de las entidades existentes en la base de conocimiento, tomando decisiones en base a esta información.
- Utilizando la reflectividad computacional, pueden añadirse dinámicamente nuevas características al estado y estructura de las entidades, así como crearse otras no existentes previamente. También puede añadirse y modificarse el comportamiento de las mismas.
- Si la estructura sintáctica de una oración establece una semántica al respecto –sujeto, quien realiza la acción; verbo, la acción a realizar; objeto directo, quién o qué recibe la acción–, la reflectividad computacional puede ser empleada para

refinar cada uno de las semánticas –cada verbo representa una acción, cuya ejecución propia podrá ser definida mediante un meta-comportamiento.

17.3.2 Ampliación y Modificación de la Implementación

Siguiendo con el sistema presentado en esta tesis, es interesante emplear trabajo de implementación para obtener una única aplicación que ofrezca todas las ventajas presentadas, mejorando detalles como su eficiencia o el desarrollo de una interfaz gráfica de usuario.

17.3.2.1 Única Aplicación

Como hemos mencionado en el capítulo 14, para justificar la tesis enunciada en esta memoria, se ha desarrollado un prototipo de la tercera capa de la arquitectura de nuestro sistema (capítulo 11) en el lenguaje de programación Python. Para simplificar su desarrollo, se identificaron los requisitos necesarios en el desarrollo de dicha capa –todos ofertados por la especificación nuestra máquina abstracta– y, al cumplir Python con ellos, implementamos el prototipo presentado en el apéndice B como demostración empírica de la tesis enunciada.

Como trabajo de implementación, sería necesario desarrollar la tercera capa sobre el entorno computacional de programación, para obtener el sistema como una única aplicación de nuestra máquina abstracta.

17.3.2.2 Especificación de Lenguajes

Nuestro sistema es independiente del lenguaje de programación. Para utilizar un determinado lenguaje, hay que incluir en él su especificación (apéndice B). En el prototipo desarrollado han sido incluidas las especificaciones de lenguajes sencillos de prueba, así como una especificación reducida de Python. Para facilitar la elección de múltiples lenguajes al programador de nuestro sistema, debería desarrollarse la especificación de los lenguajes más conocidos.

17.3.2.3 Implantación en Distintos Sistemas

El diseño e implementación del prototipo de máquina virtual (apéndice A) han sido creados sin dependencias físicas de ninguna plataforma. Sin embargo, la interacción de acceso a la plataforma e invocación de primitivas operacionales, siguen un mecanismo de interconexión dependiente del sistema operativo –la interfaz de acceso, no obstante, es invariable. Trabajo futuro es la implantación de la máquina virtual en distintas plataformas físicas.

La realización de implementaciones directas sobre código binario de un microprocesador, sin necesidad de implantar un sistema operativo previo, es otro campo en el que se pueden enfocar futuros trabajos de implementación.

17.3.2.4 Ayuda al Desarrollo de Aplicaciones

El sistema propuesto en esta tesis está orientado a ofrecer un entorno de programación independiente del lenguaje, flexible y heterogéneo. La implantación de una máquina virtual ofrece al usuario la ilusión de poseer un nuevo procesador en su sistema. Mediante un protocolo estándar de interconexión de procesos, se accede a la plataforma enviándole el código a procesar y obteniendo los resultados de su ejecución. De esta forma, cualquier aplicación nativa puede interactuar con nuestro sistema.

Para desarrollar aplicaciones, al ubicarse la plataforma como un proceso adicional, será interesante desarrollar un entorno gráfico de programación capaz de visualizar el estado dinámico del entorno computacional –como el *Browser* de Smalltalk [Mevel87], que documenta la totalidad del sistema–, crear nuevos objetos, atributos y métodos, así como evaluar rutinas que desencadenen una computación.

17.3.2.5 Eficiencia

La flexibilidad constituye el principal objetivo de nuestro sistema. El diseño de la plataforma y la posterior construcción del entorno de programación y del sistema reflectivo no restrictivo, han sido llevados a cabo tratando en todo momento de obtener un grado máximo de flexibilidad. Sin embargo, esta característica suele estar reñida con la eficiencia en tiempo de ejecución.

Los sistemas de adaptación semántica mediante MOPs, utilizan el concepto de metaobjeto: "Objeto sobre el que se delega la semántica de parte del sistema". Éstos forman parte del nivel computacional de interpretación del lenguaje, ofreciendo la capacidad de derogar parte del funcionamiento del sistema. Constituyen sistemas computacionales con un único nivel computacional –dentro de la torre de intérpretes propuesta por Smith [Smith82]–, sufriendo restricciones en su adaptabilidad, pero ofreciendo mayor eficiencia que nuestro sistema.

Mejoras en el rendimiento del sistema pueden obtenerse de diversas maneras:

- Acelerando la implementación de la máquina. Ésta puede ser llevada a cabo de un modo más eficiente. Puede optarse por su desarrollo en lenguaje ensamblador de cada plataforma, en lugar de su codificación en C++. Del mismo modo, puede estudiarse el desarrollo físico (*hardware*) de la plataforma [Myers82].

- Acelerando la ejecución de su código. La máquina abstracta interpreta su código de un modo puro, puesto que debe adaptarse a los cambios que el usuario introduzca dinámicamente. Sin embargo, podría utilizar técnicas híbridas de interpretación y compilación.

  La utilización de técnicas de compilación bajo demanda (JIT) provoca, conforme se va ejecutando la aplicación, la traducción del código a su representación nativa [Yellin96]. Se produce así una ejecución en lugar de una interpretación, reduciéndose un nivel computacional. Adicionalmente, la compilación dinámica adaptable, optimiza dinámicamente el código más utilizado (*Hot Spots*) [Hölzle94]. Plataformas como Java o .net utilizan estas técnicas.

  Para no perder la flexibilidad de poder modificar la estructura y comportamiento dinámicamente, puede describirse un esquema computacional híbrido: ejecución nativa mientras no haya modificación, e interpretación con traducción dinámica cuando se produzca ésta [Murata94].

  La principal diferencia en el desarrollo de estas técnicas para nuestra plataforma, frente a los sistemas actuales como Java, es que el sistema de traducción adaptable bajo demanda se desarrolla fuera de la implementación de la máquina –gracias a la utilización de la reflectividad computacional (§ 15.5)–, pudiéndose adaptar a los requisitos del sistema y de la plataforma existente.

APÉNDICE A:
DISEÑO DE UN PROTOTIPO DE MÁQUINA VIRTUAL

Como ejemplo de implementación de una máquina virtual capaz de ejecutar las instrucciones propias de la máquina abstracta descrita en el capítulo 12, mostraremos el diseño de un prototipo desarrollado en el lenguaje de programación C++ [Stroustrup98].

La implementación, su código fuente, un diseño detallado y una batería de pruebas, pueden ser descargados en la URL:
`http://www.di.uniovi.es/reflection/lab/prototypes.html`.

A.1 Acceso a la Máquina Virtual

Dentro de lo que es la vista física de la implementación, la máquina virtual puede verse como un componente, `nitrOVM`, que ofrece una interfaz, `nitrOInterface`, con seis operaciones que permiten acceder a su sistema computacional. La implementación de la máquina virtual utiliza el acceso a un conjunto de primitivas operacionales básicas, ubicadas en un componente aparte –`nitrOExtern`.

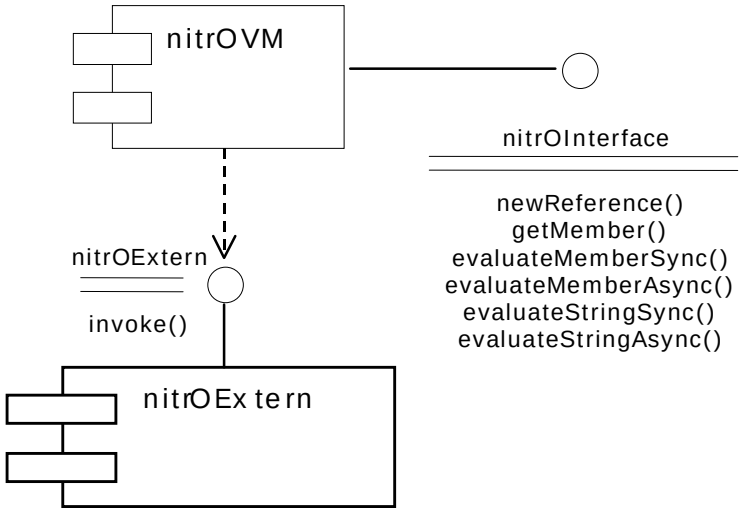


Figura A.1: Diagrama de componentes de la implementación de la máquina virtual.

Las operaciones primitivas de acceso a la máquina son:

1. `newReference`. Crea una referencia con el identificador pasado dentro del contexto de ejecución raíz (contexto cero).
2. `getMember`. Recibe como parámetros el identificador del objeto implícito y el nombre del miembro a acceder. Devuelve el identificador del miembro seleccionado.
3. `evaluateMemberSync`. Evalúa un miembro de forma síncrona. Recibe como parámetros el identificador del objeto implícito, el identificador del objeto a partir del cual se realiza la búsqueda del miembro (para encontrar a `mySelf`), el nombre del miembro, y los identificadores de los parámetros separados por comas. La devolución del método es el identificador del objeto referenciado por `return`, tras la ejecución del método.
4. `evaluateMemberAsync`. Evalúa un miembro de forma asíncrona. Los parámetros son los mismos que en la evaluación síncrona, salvo que no se obtiene un resultado.
5. `evaluateStringSync`. Evalúa un objeto cadena de caracteres de forma síncrona en el contexto cero. Recibe como parámetros el objeto y una cadena de caracteres con todos los identificadores de los parámetros. Devuelve el identificador del objeto devuelto.
6. `evaluateStringAsync`. Evalúa un objeto cadena de caracteres de forma asíncrona. Recibe los mismos parámetros que la evaluación síncrona y no devuelve ningún valor.

La única operación que define la interfaz de acceso al componente de implementaciones externas es el método `invoke`; sus parámetros son el objeto en el que se ubica dicho método primitivo, el nombre del método, el objeto implícito de la invocación y los parámetros pasados.

Como habíamos mencionado en el capítulo 12, la implantación del componente que ofrece el acceso a la máquina virtual ha de ser desplegado haciendo uso de un mecanismo estándar de intercomunicación de procesos, dentro del sistema operativo seleccionado; para las implementaciones externas se debe utilizar un mecanismo estándar de invocación a una librería. En la implementación de este primer prototipo de la máquina, hemos utilizado el sistema operativo WindowsNT/2000/XP, en el que ambos componentes han sido desarrollados mediante la utilización del modelo de objetos COM [Brown98], accesibles independientemente del lenguaje seleccionado por el programador, dentro de dichos sistemas operativos.

A.2 Diagrama de Subsistemas

La separación en subsistemas de la implementación presentada, así como las dependencias existentes entre éstos, se muestran en el siguiente diagrama de subsistemas UML [Rumbaugh98]:

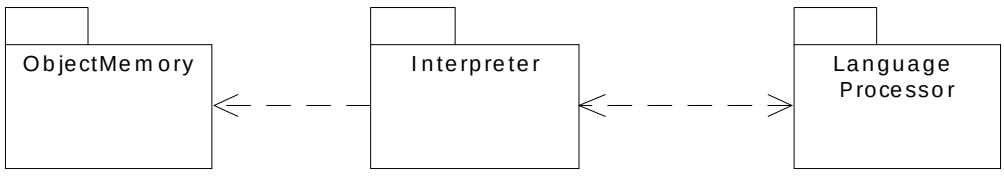


Figura A.2: Diagrama de subsistemas.

A.2.1 Subsistema LanguageProcessor

Subsistema encargado de ofrecer todos los servicios de análisis léxico y sintáctico del lenguaje de la máquina abstracta. El intérprete demanda instrucciones a este subsistema previamente a su ejecución. Éste devuelve objetos evaluables propios del subsistema `Interpreter`.

A.2.2 Subsistema Interpreter

Encargado del procesamiento computacional de cada una de las instrucciones de la máquina abstracta. Demanda al analizador sintáctico instrucciones y evalúa éstas sobre contextos asociados a un hilo de ejecución. Las evaluaciones actualizan la representación de los objetos en memoria –subsistema `ObjectMemory`.

A.2.3 Subsistema ObjectMemory

Subsistema que representa todos los objetos existentes en memoria, en tiempo de ejecución, dentro de la máquina abstracta. Objetos de usuario, primitivos, cadenas de caracteres y representación de contextos e hilos.

El intérprete implementa la semántica de las operaciones de la máquina abstracta en función de los estados de cada uno de los objetos de este subsistema.

A.3 Subsistema LanguageProcessor

Todas las operaciones relativas al procesamiento del lenguaje de la máquina abstracta, están ubicadas en este subsistema. Su vista estática viene dada por el siguiente diagrama de clases:

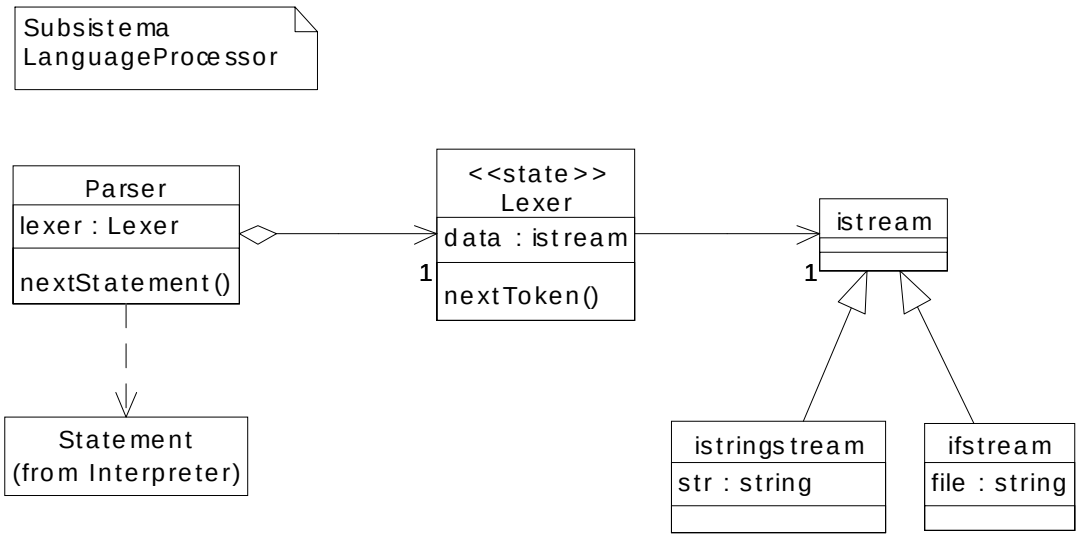


Figura A.3: Diagrama de clases del subsistema "LanguageProcessor".

La clase principal es `Parser`, que implementa el análisis sintáctico descendente asociado a la gramática del lenguaje de la máquina. Cada evaluación asociada a un hilo, tiene un objeto de este tipo, encargado de obtener instrucciones de un flujo de entrada –archivo, cadena de caracteres o miembro de un objeto. Cada vez que se desee una nueva

instrucción, se invocará al método `nextStatetement` y ésta devolverá un objeto derivado de la clase `Statement`, dentro del subsistema `Interpreter`.

El análisis léxico es llevado a cabo por objetos propios de la clase `Lexer`. Estos objetos serán utilizados por el analizador sintáctico que demandará los distintos componentes léxicos mediante el paso del mensaje `nextToken`. El analizador léxico utiliza, para implementar esta operación, el patrón de diseño Estado (*State*) [GOF94]: en función del flujo de entrada asociado a su atributo `data`, la obtención de los tokens se llevará a cabo desde una cadena de caracteres, o bien desde un archivo de entrada –el archivo de imagen que posee el entorno de computación del sistema.

A.4 Subsistema ObjectMemory

El subsistema `ObjectMemory` está formado por el siguiente conjunto de clases:

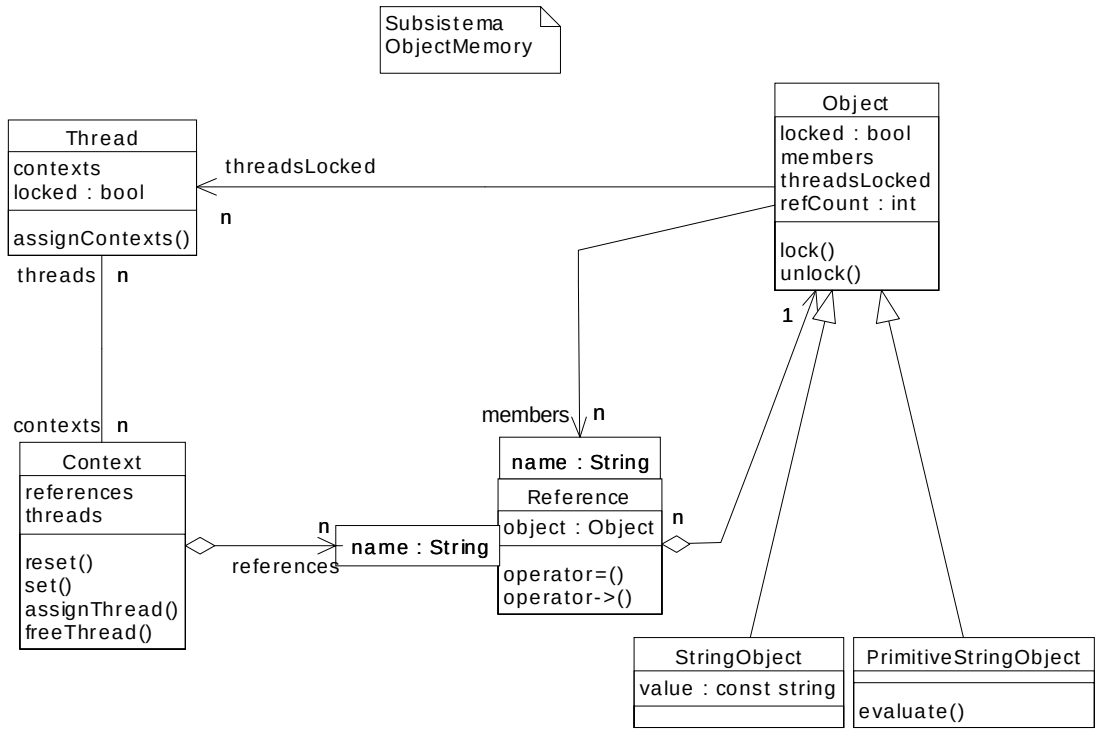


Figura A.4: Diagrama de clases del subsistema "ObjectMemory".

Las instancias de `Object` representan los objetos creados dinámicamente por la máquina virtual tras la interpretación de su código. Un objeto está compuesto por un conjunto de referencias miembro (`members`) a otros objetos, accediendo a cada de ellos por su nombre –el número de referencias que existen a un objeto es contabilizado por su atributo `refCount`. Esta descripción recursiva finaliza con la inclusión de dos objetos primitivos derivados:

- `StringObject`. Objeto que posee una cadena de caracteres invariable – `value`.
- `PrimitiveStringObject`. Son todos aquellos objetos primitivos evaluables, cuya semántica constituye funcionalidad primitiva de la máquina –

ejemplos son los miembros `new` o `delete` del objeto `nil`. Su semántica es ejecutada tras el paso del mensaje `evaluate`.

Un objeto posee la posibilidad de comportarse como un monitor en la sincronización de hilos. Cuando se ejecuta sobre éste la operación `enter` del lenguaje de la máquina (método `lock`), el objeto pasa a estar cerrado (atributo `locked`). Cuando, sobre él, se vuelva a invocar a la operación `lock`, el hilo que realizó esta solicitud pasará a bloquearse (atributo `locked` de la clase `Thread`), guardándose un puntero a éste en la pila de hilos a la espera de ejecutarse por el bloqueo de este objeto –`threadsLocked`.

La clase `Thread` representa todos los hilos en ejecución de la máquina. La evaluación de éstos se realiza de un modo secuencial, siempre que su atributo `locked` posea el valor falso. Todo hilo posee una referencia a sus contextos asociados, mediante su atributo `contexts`, indicando, de un modo ordenado, los contextos sobre los que las instrucciones asociadas podrán actuar –los contextos forman una estructura de pila.

Cada contexto de ejecución –instancias de `Context`– representa la invocación a un método o la evaluación de una cadena de caracteres, tras la que se requiere un nuevo espacio de nombres en la creación de referencias –operación `set`. Un contexto es una colección de referencias identificadas por una cadena de caracteres. Cada vez que se produce una invocación, en un determinado hilo, se crea un nuevo contexto y se apila éste. Cuando se acceda a una referencia, se buscará en el contexto existente en la cima de la pila del hilo asociado y, si no existiere, se examinan ordenadamente los contextos padre de un modo recursivo.

Las invocaciones asíncronas crean nuevos hilos a raíz del hilo padre que realizó la ejecución asíncrona. El nuevo hilo creado trabajará sobre un nuevo contexto, pero sus contextos padre serán los propios de los contextos del hilo inicial. De este modo, un hilo tiene que conocer sus contextos, y un contexto tiene que conocer sus hilos asociados (atributo `threads`). Esta situación se muestra en la siguiente figura:

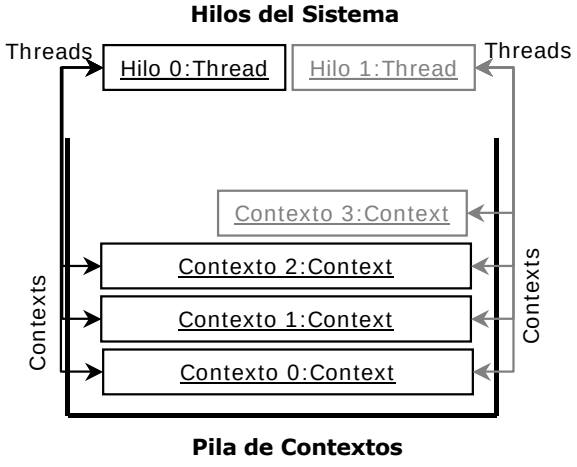


Figura A.5: Diagrama de objetos de la asociación existente entre contextos e hilos.

Inicialmente existe un único hilo (Hilo 0) que trabaja sobre los contextos 2, 1 y 0 –siguiendo dicho orden. Al realizar éste una invocación asíncrona, se crea un nuevo hilo (Hilo 1) y, para él, un nuevo contexto (Contexto 3). Los contextos 0, 1, y 2 son accesibles para ambos hilos –se les pasará el mensaje `assignThread` con el nuevo hilo.

Cada vez que finalice la ejecución de un hilo, se le notificará a todos sus contextos con el método `freeThread`. Un contexto podrá destruirse, y por tanto todas sus referencias, cuando no posea ningún hilo asociado.

A.5 Subsistema *Interpreter*

Este subsistema constituye en núcleo funcional de la máquina abstracta. Como se mostraba en el diagrama de subsistemas de la Figura A.2, este subsistema posee dependencias de los dos ya presentados. Básicamente su objetivo es ir evaluando cada hilo de la máquina virtual, asociado a una determinada entrada léxica, y ejecutando cada una de sus instrucciones, que actuarán sobre la representación dinámica del sistema ofrecida por el subsistema `ObjectMemory`.

Ésta es la vista estática del subsistema `Interpreter`:

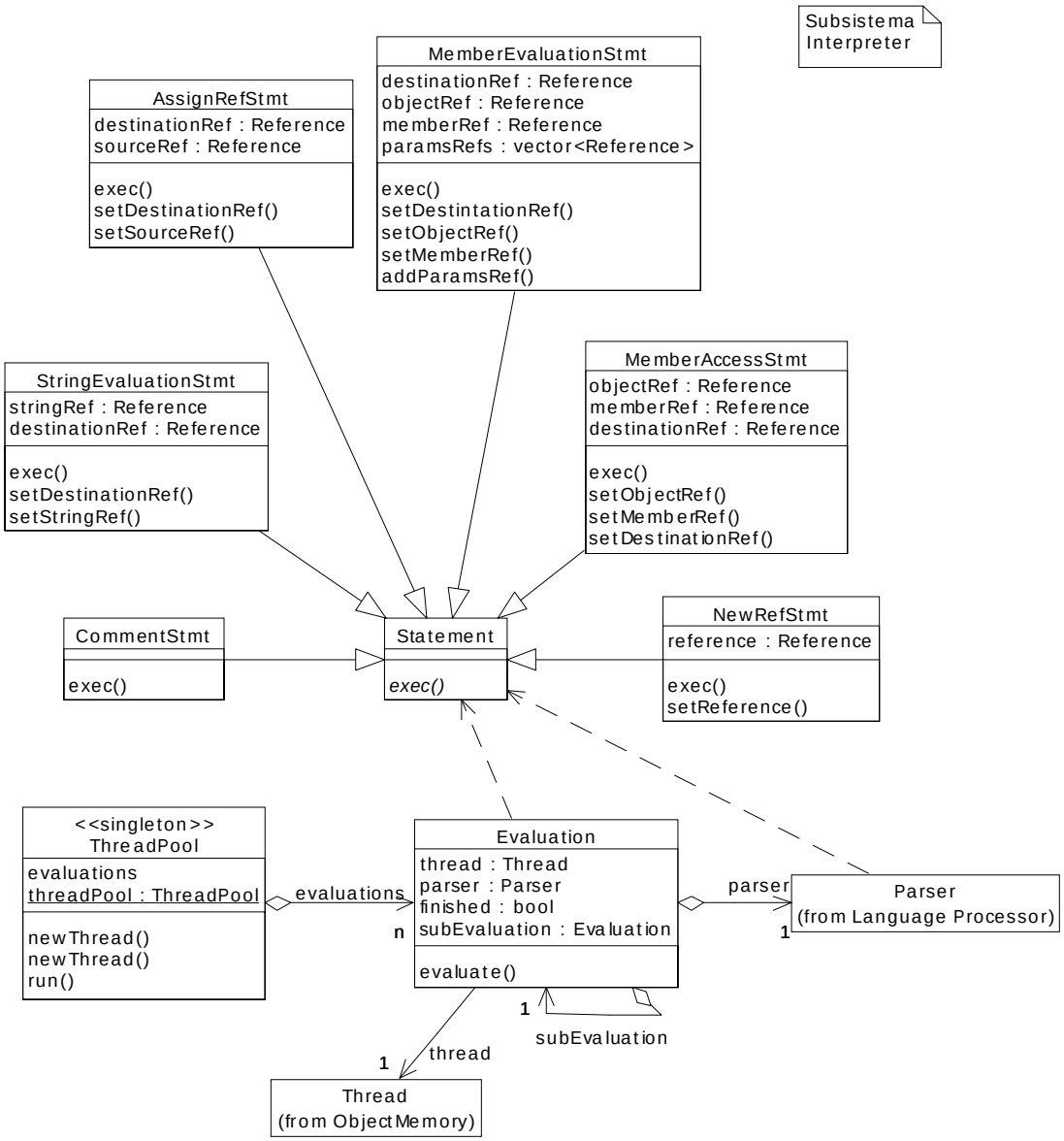


Figura A.6: Diagrama de clases del subsistema "Interpreter".

Toda instrucción tiene una representación genérica mediante la abstracción `Statement`. La operación principal a implementar por cada instrucción es `exec`: evalua-

ción de una instrucción sobre un hilo y, por tanto, sobre sus contextos. Cada clase derivada representa las distintas instrucciones de la plataforma:

- `NewRefStmt`. Creación de una referencia en el último contexto del hilo asociado.
- `AssignRefStmt`. Asignación de los objetos apuntados por dos referencias.
- `CommentStmt`. Instrucción comentario. La implementación de `exec` será nula, puesto que no tiene semántica asociada. Su utilización es únicamente para objetivos documentales.
- `MemberAccessStmt`. Acceso a un miembro de un objeto implícito, teniendo en cuenta el mecanismo de delegación ofrecido.
- `StringEvaluationStmt`. Evaluación del código asociado a un objeto cadena de caracteres.
- `MemberEvaluationStmt`. Instrucción de evaluación de un miembro de un objeto. Posee la referencia a asignar el resultado (`lvalue`), el objeto implícito (`objectRef`), el objeto miembro (`memberRef`) y un conjunto de referencias a cada parámetro pasado (`paramsRefs`). Deberá implementarse teniendo en cuenta el mecanismo de herencia.

El control de la evaluación de los distintos hilos del sistema es llevado a cabo por la única instancia existente de la clase `ThreadPool`, utilizando para ello el patrón de diseño Singleton [GOF94]. El método `run` lanza la ejecución de las instrucciones de la máquina virtual asociadas al archivo de imagen. El área de hilos guarda el conjunto de evaluaciones existentes, a través de su atributo `evaluations`.

Cada objeto de `Evaluation` representa un hilo del sistema (en ejecución o bloqueado) que trabaja sobre un flujo de entrada (atributo `parser`), evaluando, sobre el hilo asociado, las instrucciones que el analizador sintáctico le devuelva. Cada vez que se desee ejecutar una instrucción de un hilo, se le pasará a este objeto `Evaluation` el mensaje `evaluate`. La interpretación del lenguaje de la máquina se desarrolla mediante un bucle que recorre todas las evaluaciones del sistema, invocando a su método `evaluate` –esta funcionalidad la desarrolla el método `run` de `ThreadPool`.

La evaluación síncrona de una cadena de caracteres o miembro de un objeto, crea una subevaluación del objeto evaluación actual, utilizando el mismo hilo pero un distinto flujo de entrada –una cadena de caracteres o un objeto miembro respectivamente. El único hilo existente para ambas ejecuciones tendrá un nuevo contexto en la cima de su pila de contextos, correspondiente a la nueva evaluación.

La evaluación asíncrona de un objeto miembro o una cadena de caracteres, supone la creación de una nueva evaluación asociada al nuevo hilo creado, que utiliza como flujo de entrada el objeto cadena de caracteres o el miembro empleados en la invocación.

APÉNDICE B:
MANUAL DE USUARIO DEL PROTOTIPO DE COMPUTACIÓN REFLECTIVA SIN RESTRICCIONES

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`.

B.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.

B.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 aplicaciones 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:

1. Especificación del lenguaje mediante un archivo de extensión `ml`. Siguiendo la gramática § B.2.1, se describe el lenguaje y se almacena en el directorio del sistema con igual nombre que el lenguaje creado, y extensión `ml`.
2. Especificándolo en el archivo de aplicación y anteponiéndolo a su código (como veremos en § B.3.2).
3. Modificando un lenguaje ya existente previamente a la ejecución de una aplicación (detallado en § B.3.4).

B.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.

B.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 caracteres –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 (λ) 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 § B.2.5.

B.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 § B.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.

B.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.

B.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 § B.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.

B.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 § B.3.3.

B.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:

B.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 B.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.

B.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 B.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.

B.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 B.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.

B.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.

B.4 Interfaz Gráfico

Para facilitar la programación de lenguajes y aplicaciones en nuestro sistema, hemos desarrollado un pequeño interfaz gráfico para nuestro prototipo. Este interfaz está construido sobre el estándar TK [Lundh99] independiente de la plataforma a utilizar.

B.4.1 Intérprete de Comandos

La ventana principal del sistema tiene el siguiente aspecto:

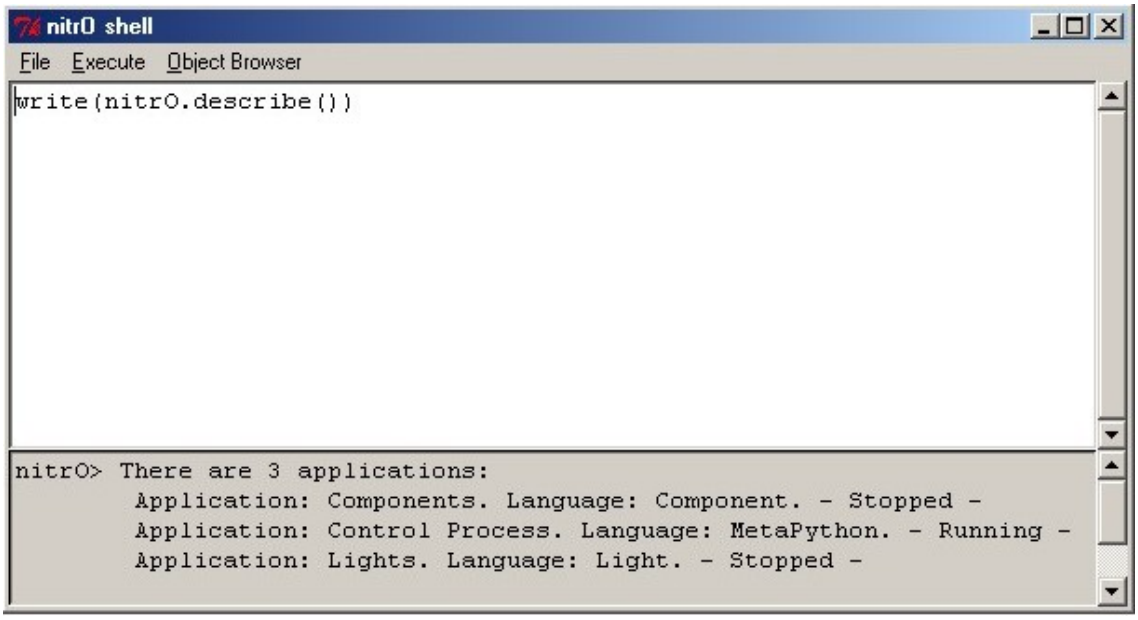


Figura B.1: Ventana principal del prototipo.

La ventana está dividida en tres partes:

1. Menú del sistema.
2. Editor de código del intérprete de comandos.
3. 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 A.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).

B.4.2 Archivos Empleados

En la utilización nuestro sistema podemos diferenciar tres tipos de archivos:

1. Archivos de especificación de lenguajes de programación codificados mediante nuestro metalenguaje. Estos archivos poseen la extensión `ml`, siguen el metalenguaje descrito en § B.2.1, y deben ubicarse en el directorio del sistema.
2. Archivos de aplicación. Poseen la extensión `na` (*nitrO application*) y describen aplicaciones del sistema en un determinado lenguaje de programación.

3. 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.

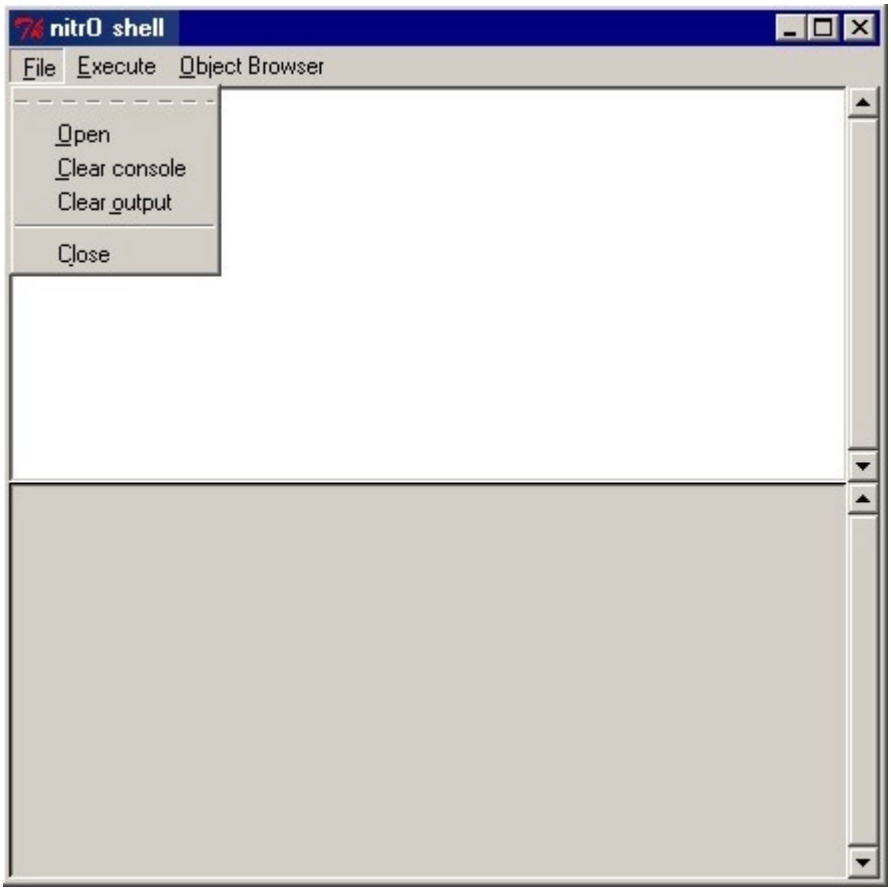


Figura B.2: Menú archivo del prototipo.

Mediante este menú, pueden cargarse archivos de *script*, así como limpiar el editor y la ventana de salida y finalizar la ejecución del sistema.

B.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:

1. 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 A.1.

2. *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.

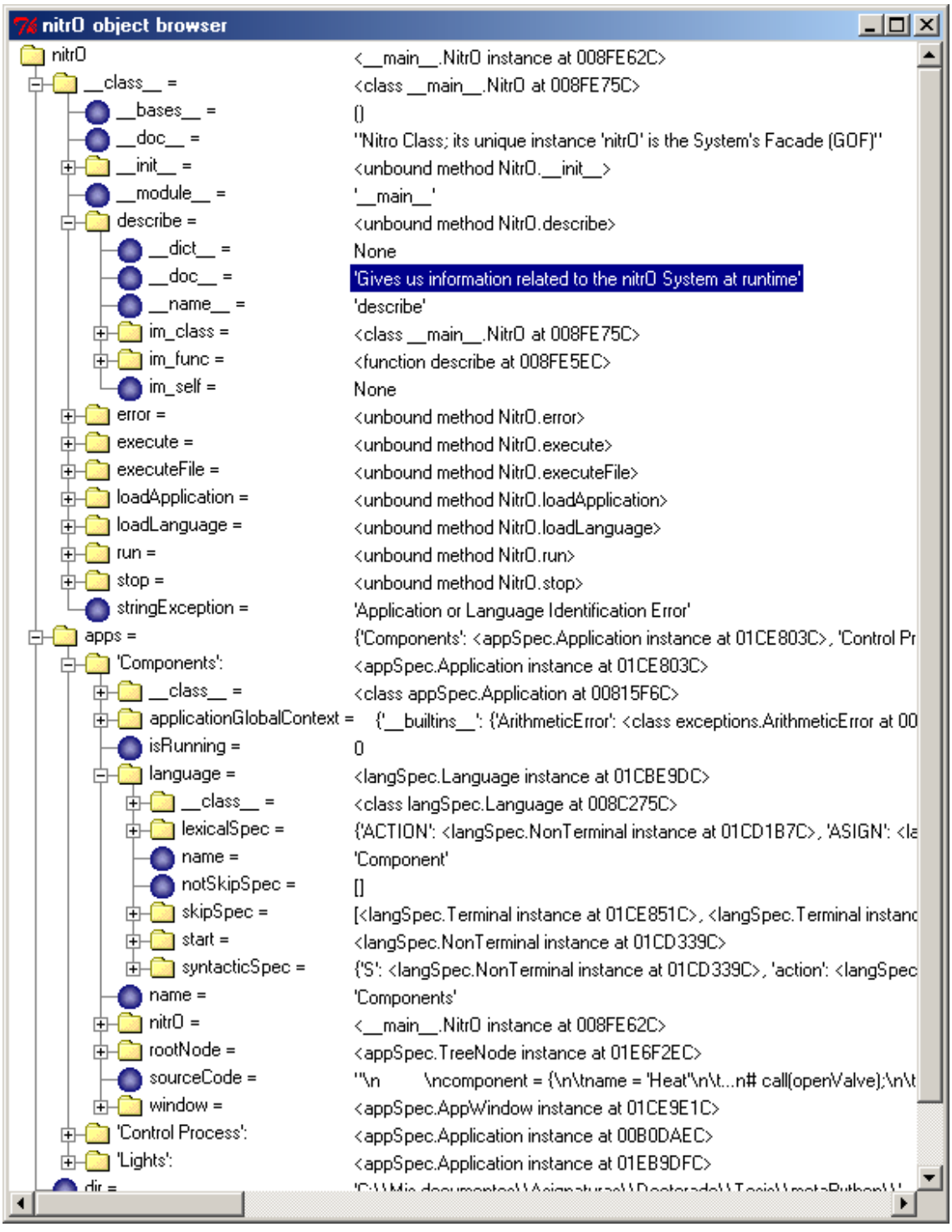


Figura B.3: Estado del sistema en tiempo de ejecución, mostrado por el *Object Browser*.

En la Figura B.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.

B.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.

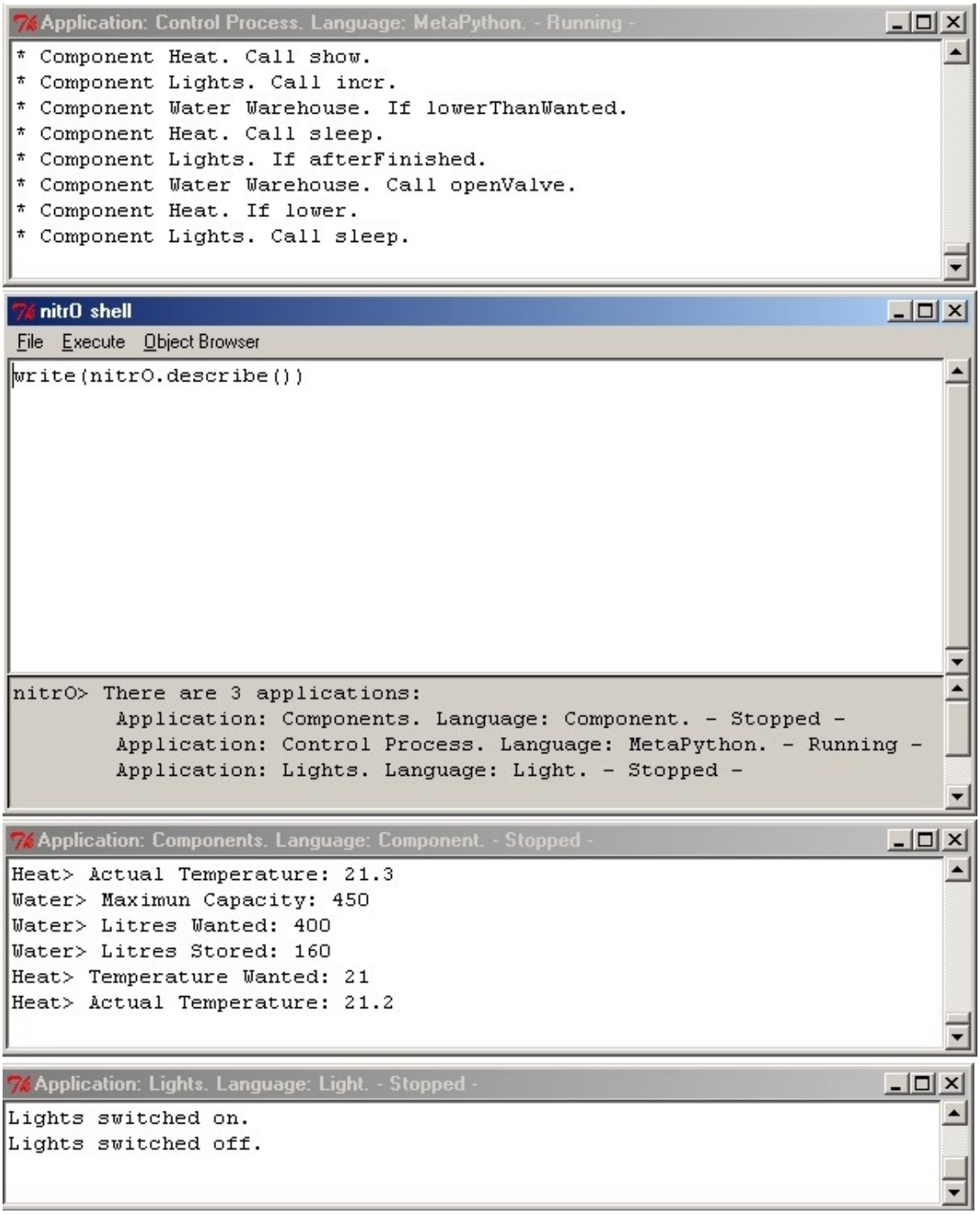


Figura B.4: Ejecución de aplicaciones en el sistema.

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 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.

[Allamaraju2000] Subrahmanyam Allamaraju et. al. "Professional Java Server Programming J2EE Edition". Wrox Press 2000.

[Álvarez96] Darío Álvarez Gutiérrez, Lourdes Tajes Martínez, Fernando Álvarez García, María Ángeles Díaz Fondón, Juan Manuel Cueva Lovelle y Raúl Izquierdo Castanedo. "Un Sistema Operativo para el Sistema Orientado a Objetos Integral Oviedo3". II Jornadas de Informática. Almuñécar, Granada. Julio de 1996.

[Álvarez97] Darío Álvarez Gutiérrez, Lourdes Tajes Martínez, Fernando Álvarez García, María Ángeles Díaz Fondón, Juan Manuel Cueva Lovelle y Raúl Izquierdo Castanedo. "An Object-Oriented Abstract Machine as the Substrate for an Object-Oriented Operating System". 11th Con-

ference in Object Oriented Programming (ECOOP). Workshop in Object Oriented Operating Systems. Jyväskylä, Finlandia. Junio de 1997.

[Álvarez98] Darío Álvarez Gutiérrez. "Persistencia Completa para un Sistema Operativo Orientado a Objetos usando una Máquina Abstracta con Arquitectura Reflectiva". Tesis Doctoral. Departamento de Informática. Universidad de Oviedo. Marzo de 1998.

[Álvarez2000] Fernando Álvarez García. "AGRA: Sistema de Distribución de Objetos para un Sistema Distribuido Orientado a Objetos soportado por una Máquina Abstracta". Tesis Doctoral. Departamento de Informática. Universidad de Oviedo. Septiembre de 2000.

[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.

[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.

[Assumpcao99] Jecel M. de Assumpcao Jr. "Incremental Porting of the Self/R Virtual Machine". OOPSLA'99 Virtual Machine Workshop. Denver, Colorado (EE.UU.). Noviembre de 1999.

[Babao89] Ozalp Babao. "Fault-Tolerant Computing based on Mach". Technical Report TR 89-1032. Cornell University, Dept. of Computer Science, Ithaca, New York (EE.UU.). Agosto de 1989.

[Baillarguet98] Carine Baillarguet y Ian Piumarta. "An Highly-Configurable, Modular System Architecture for Mobility, Interoperability, Specilisation and Reuse". INRIA, Rocquencourt, B.P. 105, 78153, Les Chesnay Cedex, Francia. 1998.

[Baker97] Seán Baker. CORBA Distributed Objects. Addison-Wesley, ACM Press. ISBN 0-201-92475-7. 1997.

[Bancilhon92] François Bancilhon, Claude Belobel y Pâris Kanellakis. "Building an Object-Oriented Database System – The store of O2". Morgan Kaufman. 1992.

[Barak85] Amon Barak, Ami Litman. "MOS: A Multicomputer Distributed Operating System". Software Practice and Experience, vol. 15, nº 8. Agosto de 1985.

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

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

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

[Bergmans94] L. Bergmans. "Composing Concurrent Objects: Applying Composi-

tion Filters for the Development and Reuse of Concurrent Object-Oriented Programs". Ph. D. Dissertation. Universidad de Twente (Holanda). Junio de 1994.

[BerkeleyCW] Original Code War Site. ftp://ftp.esua.berkeley.edu/pub/codewar. University of Berkeley. California, EE.UU.

[Blair97] Gordon S. Blair, Geoff Coulson. "The Case for Reflective Middleware". Proceedings of the 3rd Cabernet Plenarg Workshop. Rennes (Francia). Abril de 1997.

[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.

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

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

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

[Campione99] Mary Campione, Kathy Walrath. "The Java Tutorial. Second Edition. Object Oriented Programming for Internet". The Java Series. Sun Microsystems. 1999.

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

[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 Notices, 24, 10, 49-70. 1989.

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

[Chambers92] Craig Chambers. "The Design and Implementation of the Self Compiler, and Optimizing Compiler for Object-Oriented Programming Languages". PhD Thesis. Universidad de Stanford (EE.UU.). Marzo de 1992.

[Chambers93] Craig Chambers. "The Cecil Language: Specification and Rationale". Technical Report TR-93-03-05. Department of Computer Science and Engineering. University of Washington. Marzo de 1993.

[Cheriton88] David Cheriton. "The V Distributed System". Communications of the ACM, vol. 31, nº 3. Marzo de 1988.

[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.

[Chiba98b] Shigeru Chiba. “Javassist –a Reflection-Based Programming Wizard for Java”. OOPSLA’98 Workshop in Reflective Programming. Vancouver (Canadá). Octubre de 1998.

[Cibele98] Amanda Cibele, A. Rosa, Eliane Martins. “Using a Reflective Architecture to Validate Object-Oriented Applications by Fault Injection”. OOPSLA’98 Workshop in Reflective Programming. Vancouver (Canadá). Octubre de 1998.

[Clark83] K. L. Clark y S. Gregory. “Parlog: A Parallel Logic Programming Language”. Imperial College, London. Mayo de 1983.

[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.

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

[Cueva91] Juan Manuel Cueva Lovelle. “Lenguajes, Gramáticas y Autómatas”. ISBN: 84-600-7871-X. Diciembre de 1991.

[Cueva92] Juan Manuel Cueva Lovelle. “Organización de la Memoria en Tiempo de Ejecución en Procesadores de Lenguajes”. Cuaderno Didáctico número 55. Departamento de Matemáticas. Universidad de Oviedo. Abril de 1992.

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

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

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

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

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

[Cueva96] Juan Manuel Cueva Lovelle, Raúl Izquierdo Castanedo y Darío Álvarez Gutiérrez. “Oviedo3: Acercando las Tecnologías Orientadas a Objeto al Hardware”. I Jornadas de Trabajo en Ingeniería del Software. Sevilla. Noviembre de 1996.

[Cueva98] Juan Manuel Cueva Lovelle. “Conceptos Básicos de Procesadores de Lenguaje”. Cuaderno Didáctico número 10. Editorial Servitec.. Diciembre de 1998.

[Dageforde2000] Mary Dageforde. “Security in Java2 SDK 1.2. The Java™ Tutorial.” Javasoft, Sun Microsystems. Febrero de 2000.

[Dewdney88] A. K. Dewdney. “The Armchair Universe: An Exploration of Computer World”. W. H. Freeman, ISBN: 0-7167-1939-8. 1988.

[Dewdney90] A. K. Dewdney. “The Magic Machine: A Handbook on Computer Sorcery”. W. H. Freeman, ISBN: 0-7167-2125-2. 1990.

[Díaz2000] María Ángeles Díaz Fondón. “Núcleo de Seguridad para un Sistema Operativo Orientado a Objetos soportado por una Máquina Abstracta”. Tesis Doctoral. Departamento de Informática. Universidad de Oviedo. Enero de 2000.

[Díez2001] Diego Díez Redondo. “Implementación de un Entorno Flexible de Computación sobre una Máquina Abstracta Reflectiva”. Proyecto Final de Carrera 1012056. Escuela Técnica Superior de Ingenieros Industriales e Ingenieros Informáticos. Universidad de Oviedo. Propuesta aprobada en Septiembre de 2001 (por publicar).

[Dijk95] W. van Dijk, J. Mordhorst. “Composition Filters in Smalltalk”. HIO Graduation Thesis. Universidad de Twente (Holanda). 1995.

[Dorward97] Sean M. Dorward, Rob Pike, Davied Leo Presotto, Dennis M. Ritchie, Howard W. Trickey y Philip Winterbottom. “The Inferno™ Operating System”. Bell Labs Technical Journal. Invierno de 1997.

[Dorward97b] Sean M. Dorward, Rob Pike y Philip Winterbottom. “Programming in Limbo”. Proceedings of the IEEE Computer Conference (COMPCON). San Jose, California (EE.UU.). 1997.

[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.

[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.

[Evins94] M. Evins. “Objects Without Classes”. Computer IEEE. Vol. 27, N. 3, 104-109. 1994.

[Ferber88] Jacques Ferber. “Conceptual Reflection and Actor Languages”. Meta-Level Architectures and Reflection. P. Maes and D. Nardi Editors.

North-Holland. 1988.

[Ferreira98] Luciane Lamour Ferreira, Cicília M. F. Bubira. "The Reflective State Patern". OOPSLA'98 Workshop in Reflective Programming. Vancouver (Canadá). Octubre de 1998.

[Folliot97] Bertil Folliot, Ian Piumarta y Fabio Reccardi. "Virtual Virtual Machines". Cabernet Radicals Workshop, Creta. Septiembre de 1997.

[Folliot98] Bertil Folliot, Ian Piumarta y Fabio Reccardi. "A Dynamically Configurable, Multi-Language Execution Platform". 8th ACM SIGOPS European Workshop. Septiembre de 1998.

[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.

[Freier96] Alan O. Freier, Philip Karlton y Paul C. Kocker. "The SSL Protocol, version 3.0". Transport Layer Security Working Group.

[Geist94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck y Vaidy Sunderam. "PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing". The MIT Press. Cambridge, Massachusetts, EE.UU. 1994.

[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.

[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.

[Gregory87] S. Gregory. "Parallel Logic Programming in Parlog, The Language and its Implementation". Addison-Wesley. 1987.

[Harris99] Timothy L. Harris. "An Extensible Virtual Machine Architecture". Citrix Ssytems (Cambridge) Ltd., Computer Laboratory. Agosto de 1999.

[Hickey2000] Jason Hickey, Alexey Nogin. "Fast Tatic-Based Theorem Proving". TPHOLs 2000.

[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.

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

[Howe95] Denis Howe. The Free Online Dictionary. "Abstract Machine". www.foldoc.org. Marzo 1995.

[Howe99] Denis Howe. The Free Online Dictionary. "Virtual Machine". www.foldoc.org. Marzo 1999.

[Huggins96] Jim Huggins. "Abstract State Machines". www.eecs.umich.edu/gasm/cover.html. Septiembre 1996.

[Huggins99] Jim Huggins. "Abstract State Machines: Introduction". www.eecs.umich.edu/gasm/intro.html. Febrero 1999.

[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] IBM Corporation. "VM/ESA – An S/390 Platform for Business Solutions. VM/ESA Version 2 Release 4 Specification Sheet. 2000.

[IBM2000b] "The IBM J9 Virtual Machine". International Business Machines Corporation. Junio de 2000.

[IBM2000c] "Visual Age Micro Edition. Strategic Benefis of Virtual-Machine-Based Architecture. Core Technology for Embedded Systems". International Business Machines Corporation. 2000.

[IBM2000d] "Developing Embedded Applications". International Business Ma-

chines Corporation. Junio de 2000.

[IBM2000e] "Multi-Dimensional Separation of Concerns". International Business Machines Corporation, IBM Research. 2000.

[IBM2000f] "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.

[Irvine99] "p-System: Description, Background, Utilities". Educational Technology Center. Department of Information and Computer Science, University of California, Irvine. 1999.

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

[Javasoft99] "Java 2 SDK Standard Edition Documentation". Javasoft, Sun Microsystems. 1999.

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

[Johnston2000] Stuart J. Johnston. "The Future of COM+ –Microsoft's .NET Revealed". The XML Magazine. Octubre de 2000.

[Jones99] Paul Jones. "VMware Virtual Platform for Linux: Beyond Dual-Booting". Internet.com. Mayo de 1999.

[Kalev98] Danny Kalev. "The ANSI/ISO C++ Professional Programmers Handbook". Que Editorial. 1998.

[Kasbekar98] Mangesh Kasbekar, Chandramouli Narayanan, Chita R. Das. "Using Reflection for Checkpointing Object Oriented Programs". OOPSLA'98 Workshop in Reflective Programming. Vancouver (Canadá). Octubre de 1998.

[Keleher96] Pete Keleher. "CVM: The Coherent Virtual Machine". University of Maryland. Noviembre de 1996.

[Kessin99] Zachary Kessin. "Recursion and Loops in Scheme". www.script-fu.org. 1999.

[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.

[Kiczales96] Gregor Kiczales. "Beyond the Black Box: Open Implementations". IEEE Software. Vol. 13, nº 1. 1996.

[Kiczales96b] Gregor Kiczales, John Lamping, Cristina Videira Lopes, Chris Maeda,

Anurag Mendhekar, Gail Murphy. “Open Implementation Design Guidelines”. 19th International Conference on Software Engineering. 1996.

[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.

[Killijian98] Mark-Olivier Killijian, Jean-Charles Fabre, Juan Carlos Ruiz García, Shigeru Chiba. “Development of a MetaObject Protocol for Fault Tolerance using Compile-Time Reflection”. OOPSLA’98 Workshop in Reflective Programming. Vancouver (Canadá). Octubre de 1998.

[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.

[Kloosterman98] Sytse Kloosterman y Marc Shapiro. “Large Scale Distributed Object Programming Made Easy”. www.perdis.esprit.es.org. Marzo de 1998.

[Koelbel94] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele y Mary E. Zosel. “The High Performance Fortran Handbook”. Editorial Zosel. 1994.

[Kozak2000] Robert Kozak. “The Dish on Kylix. Cross-Platform Controls. From Windows to Linux, and Back”. DelphiZine.com. Mayo 2000.

[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.

[Leavenworth93] B. Leavenworth. “PROXY: a Scheme-Based Prototyping Language”. Dr. Dobb’s Journal, No. 198, 86-90. Marzo de 1993.

[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.

[Li89] Kai Li y Pal Hudak. “Memory coherence in shared virtual memory systems”. ACM Transactions on Computer Systems, 7. Noviembre de 1989.

[Lieberherr96] Karl J. Lieberherr. “Adaptive Object Oriented Software: The Demeter Method”. PWS Publishing Company. 1996.

[Lieberman86] H. Lieberman. “Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems”. In OOPSLA’86 Conference Proceedings. Published as SIGPLAN Notices, 21, 11, 214-223. 1986.

[Liedtke95] Jochen Liedtke. “On μ-Kernel Construction”. 15th Symposium on Operating Systems (SOSP). Colorado (EE.UU.). 1995.

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

[Link98] G. Link. “Algebraic Semantics in Language and Philosophy”. CSLI Publishers. 1998.

[Lundh99] Fredrik Lundh. “An Introduction to Tkinter”. Review Copy. Enero de 1999.

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

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

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

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

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

[Martínez98b] Ana Belén Martínez, Darío Álvarez, Juan Manuel Cueva, Francisco Ortín. “BDOviedo3, An Object-Oriented DBMS Incorporated to an Integral Object-Oriented System”. 27th Argentine Conference on Informatics and Operations Research. Symposium on Object-Orientation. Buenos Aires (Argentina). Septiembre de 1998.

[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.

[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.

[Matsuoka98] Satoshi Matsuoka, Hirotaka Ogawa, Kouya Shimura, Yasunori Kimura, Koichito Hotta, Hiromitsu Takagi. "OpenJIT –a Reflective Java JIT Compiler". OOPSLA'98 Workshop in Reflective Programming. Vancouver (Canadá). Octubre de 1998.

[McJones88] Paul R. McJones, Garret F. Swart. "Envolving the UNIX System Interface to Support Multithreaded Programs". Technical Report 21, DEC Systems Research Center, Palo Alto, California (EE.UU.). Septiembre de 1988.

[Megginson2000] David Megginson. "SAX 2.0: The Simple API for XML". http://www.meggison.com.

[Mendhekar92] Anurag Mendhekar, Danel P. Friedman. "Towards a Theory of Reflective Programming Languages". Department of Computer Science. Indiana University (EE.UU.). 1992.

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

[Meyer97] Bertrand Meyer. "Object-Oriented Software Construction". 2ª Edición. Prentice-Hall. 1997.

[Microsoft2000] "Microsoft .NET Home Page". www.microsoft.com/net. 2000.

[Microsoft2000b] "C# Language Specification v.0.22". Microsoft Corporation. Febrero de 2000.

[Microsoft95] "The Component Object Model Specification". Version 0.9. Microsoft Corporation. Octubre de 1995.

[Mitsuru97] Oshima Mitsuru, Karjoth Guenter. "Aglets Specification". Draft 2.0. IBM Japón. Septiembre de 1997.

[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.

[Mulet95] Philippe Mulet, Jacques Malenfant, Pierre Cointe. "Towards a Methodology for Explicit Composition of MetaObjects". OOPSLA'95 Conference Proceedings, ACM Sigplan Notices. Octubre de 1995.

[Mullender87] "The Amoeba Distributed Operating System: Selected Papers 1984-1987" Sape J. Mullender Editorial. Amsterdam (Holanda). 1987.

[Mullender90] S. J. Mullender, G. van Rossum, A. S. Tanenbaum. "Amoeba: A Distributed Operating System for the 1990s". IEEE Computer Society, vol. 23, nº 5. Washington D.C. (EE.UU.) 1990.

[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.

[Myers82] G. Myers. "Advances in Computer Architecture. Second Edition". New York, NY: John Weley and Sons. 1982.

[Nori76] K. V. Nori, U. Ammann, K. Jensen, H. H. Nageli y C. Jacobi. "The Pascal-P Compiler: Implementation Notes". Bericht 10, Eidgengössische Technische Hochschule. Zurich, Suiza. Julio 1976.

[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.

[OMG94] Object Management Group (OMG). "Object Transaction Service". Agosto de 1994.

[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.

[Orfali96] Robert Orfali, Dan Harkey y Jeri Edwards. "The Essential Client / Server Survival Guide". Second Edition. Wiley. 1996.

[Orfali98] Robert Orfali, Dan Harkey. "Client/Server Programming with Java and CORBA". 2ª Edición. Wiley Editorial. 1998.

[Ortín2000] Francisco Ortín Soler, Juan Manuel Cueva Lovelle. "A Flexible Integral Computing System based on a Structurally-Reflective Abstract Machine". Simposio Español de Informática Distribuida (SEID'2000). Ourense (España). Septiembre de 2000.

[Ortín2001] Francisco Ortín Soler, Juan Manuel Cueva Lovelle. "Building a Completely Adaptable Reflective System". European Conference on Object Oriented Programming ECOOP'2001. Workshop on Adaptive Object-Models and Metamodeling Techniques. Budapest (Hungría).

Junio de 2001.

[Ortín97] Francisco Ortín Soler. “Diseño y Construcción del Sistema de Persistencia en Oviedo3”. Proyecto Final de Carrera 972001. Escuela Técnica Superior de Ingenieros Industriales e Ingenieros Informáticos. Universidad de Oviedo. Septiembre de 1997.

[Ortín97b] Francisco Ortín Soler, Darío Álvarez, Raúl Izquierdo, Ana Belén Martínez, Juan Manuel Cueva. “El Sistema de Persistencia en Oviedo3”. III Jornadas de Tecnologías Orientadas a Objetos. Sevilla. 1997.

[Ortin99] Francisco Ortín Soler, Ana Belén Martínez Prieto, Darío Álvarez Gutiérrez, Juan Manuel Cueva Lovelle. “An Implicit Persistence System on a Object-Oriented Database Engine using Reflection”. International Conference on Information Systems, Analysis and Synthesis. Orlando (EE.UU.). Agosto de 1999.

[Ortín99b] Francisco Ortín Soler, Ana Belén Martínez Prieto, Darío Álvarez Gutiérrez, Juan Manuel Cueva Lovelle. “A Reflective Persistence Middleware over an Object-Oriented Database Engine”. XIV Brazilian Symposium on Databases (SBBD). Florianopolis (Brasil). 1999.

[Ortín99c] Francisco Ortín Soler, Ana Belén Martínez Prieto, Darío Álvarez Gutiérrez, Juan Manuel Cueva Lovelle. “Implicit Object Persistence on a Reflective Abstract Machine”. 28th Argentine Conference on Informatics and Operations Research. Symposium on Object Orientation (ASSO). Buenos Aires (Argentina). 1999.

[Ortín99d] Francisco Ortín Soler, Ana Belén Martínez Prieto, Darío Álvarez Gutiérrez, Juan Manuel Cueva Lovelle. “Diseño de un Sistema de Persistencia Implícita mediante Reflectividad Computacional”. IV Jornadas de Ingeniería del Software y Bases de Datos (JISBD). Cáceres (España). Noviembre de 1999.

[Ossher99] H. Ossher, P. Tarr. “Multi-Dimensional Separation of Concerns using Hyperspaces”. IBM Research Report 21452. Abril de 1999.

[Raman98] L. G. Raman. “OSI Systems and Network Management”. IEEE Communications Magazine. Marzo de 1998.

[Rashid86] Rick Rashid. “Threads of a New System”. Unix Review. Agosto de 1986.

[Richards71] Martin Richards. “The Portability of the BCPL Compiler”. Software Practice and Experience. 1971.

[Richards79] Martin Richards y Colin Whitby-Stevens. “BCPL – The Language and its Compiler”. Cambridge University Press. 1979.

[Ritchie78] D.M. Ritchie y B. W. Kerninghan. “The C Programming Language”. Prentice Hall. 1978.

[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.

[Roddick95] J. Roddick. “A Survey of Schema Versioning Issues for Database Systems”. Information and Software Technology, 37. 1995.

[Rossum2001] Guido van Rossum. “Python Reference Manual”. Fred L. Drake Jr. Editor. Relesase 2.1. Abril de 2001.

[Rozier88] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, Will Neuhauser. “CHORUS Distributed Operating System”. Computing Systems Journal, vol. 1, nº 4. The Usenix Association. Diciembre de 1988.

[Rumbaugh98] James Rumbaugh, Ivar Jacobson, Grady Booch. “The Unified Modeling Language Reference Manual”. Addison-Wesley. Diciembre de 1998.

[Sirer99] Emin Gün Sirer, Robert Grimm, Arthur J. Gregory y Brian N. Bershad. “Design and Implementation of a Distributed Virtual Machine for Network Computers”. 17th ACM Symposium on Operating Systems Principles (SOSP’99). Diciembre de 1999.

[Smith82] B. C. Smith. “Reflection and Semantics in a Procedural Language”. MIT-LCS-TR-272. Massachusetts Institute of Technology. Cambridge (EE.UU.). 1982.

[Smith92] Robert Smith, Aaron Sloman y John Gibson. “PROLOG’s Two-Level Virtual Machine Support for Interactive Languages”. Research Directions in Cognitive Science, v.5. 1992.

[Smith95] Randall B. Smith, David Ungar. “Programming as an Experience: The Inspiration for Self”. Sun Microsystems Laboratories. 1995.

[Steel60] T. B. Steel Jr. “UNCOL: Universal Computer Oriented Language Revisited”. Datamation. Enero-Febrero 1960.

[Steele90] Jr. Steele “Common Lisp: The Language”. Segunda Edición. Digital Press, 1990.

[Stroustrup98] Bjarne Stroustrup. “The C++ Programming Language”. Third Edition. Addison-Wesley. October 1998.

[Sturman98] Daniel Sturman, Guruduth Banavar, Robert Strom. “Reflection in the Gryphon Message Brokering System”. OOPSLA’98 Workshop in Reflective Programming. Vancouver (Canadá). Octubre de 1998.

[Sun89] Sun Microsystems, Inc. “NFS: The Network File System protocol specification”. RFC 1094, Network Information Center, SRI International. Marzo de 1989.

[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] “picoJava™ I Data Sheet. Java Processor Core”. Sun Microsystems. Diciembre de 1997.

[Sun97b] “Java™ Remote Method Invocation Specification (RMI)”. JavaSoft. Sun Microsystems. Febrero de 1997.

[Sun97c] "Java™ Native Interface Specification". JavaSoft. Sun Microsystems. Mayo de 1997.

[Sun97d] "Java Core Reflection. API and Specification". JavaSoft. Enero de 1997.

[Sun97e] "Java™ Object Serialization Specification". JavaSoft. Sun Microsystems. Febrero de 1997.

[Sun98] "The Java HotSpot Virtual Machine Architecture". White Paper. Sun Microsystems. 1998.

[Sun98b] "Enterprise JavaBeans™ v1.0". JavaSoft. Sun Microsystems. 1998.

[Sun99] "Dynamic Proxy Classes". Sun Microsystems, Inc. 1999.

[Swaine94] M. Swaine "Programming Paradigms. Developing for Newton". Dr. Dobb's Journal, No. 212, 115-118. Marzo de 1994.

[Tajes2000] Lourdes Tajes Martínez. "Modelos de Computación para un Sistemas Operativos Orientados a Objetos basados en una Máquina Abstracta Reflectiva". Tesis Doctoral. Departamento de Informática. Universidad de Oviedo. Marzo de 2000.

[Tanenbaum95] Andrew S. Tanenbaum. "Sistemas Operativos Modernos". Prentice-Hall. ISBN 968-880-323-5. 1995.

[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.

[Trados96] Alain Trados y Eric Uber. "Using Borland's Delphi and C++ Together". A technical Paper for Developers. Borland Online. Marzo de 1996.

[Turing36] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". Proceedings of The London Mathematical Society, Series 2, 42. 1936.

[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.

[Ungar91] David Ungar, Craig Chambers, Bay-wei Chang, and Urs Hölzle. "Organizing Programs Without Classes". Lisp and Symbolic Computation: An International Journal, 4, 3. 1991.

[Venners98] Bill Venners. "Inside the Java Virtual Machine". Java Masters. McGraw Hill. 1998.

[Vo96] Kiem-Phong Vo. "Vmalloc: A General and Efficient Memory Allocator". Software –Practice and Experience. Marzo de 1996.

[W3C98] "Extensible Markup Language (XML) 1.0". World Wide Web Consortium. Febrero de 1998.

[W3C98b] "Level 1 Document Object Model Specification". Version 1.0. W3C Working Draft, WD-DOM-19980720. Julio de 1998.

[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.

[Winterbottom97] Philip Winterbottom y Rob Pike. “The Design of the Inferno Virtual Machine”. Bell Labs, Luncent Technologies. 1997.

[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.

[Wu98] Zhixue Wu. “Reflective Java and A Reflective Component-Based Transaction Architecture”. OOPSLA’98 Workshop in Reflective Programming. Vancouver (Canadá). Octubre de 1998.

[Yellin96] Frank Yellin. “The JIT Compiler API”. Sun Microsystems. Octubre 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.

[Yonezawa90] Akinori Yonezawa. “ABCL: An Object-Oriented Concurrent System”. Computer Science Series. The MIT Press, 1990.

[Yokote92] Y. Yokote. “The New Mechanism for Object-Oriented System Programming”. Proceedings of IMSA’92 International Workshop on Reflection and Meta-level Architecture”. 1992.

[Yokote92b] Y. Yokote. “The Apertos Reflective Operating System: The Concept and its Implementation”. Proceedings of The Conference on Object-Oriented-Programming Systems, Languages and Applications (OOPSLA’92). Octubre de 1992.