SISTEMAS INTEGRALES ORIENTADOS A OBJETOS

Darío Álvarez Gutiérrez

Doctor en Informática
Departamento de Informática de la
Universidad de Oviedo

Oviedo, Marzo de 1999

Ed. Servitec / ISBN: 84-669-0721-2

RESUMEN

La adopción del paradigma de la orientación a objetos no se produce de manera integral dentro de todos los componentes de un sistema de computación. Existen lenguajes, bases de datos, interfaces de usuario y otros elementos que utilizan la orientación a objetos que deben cambiar a otro paradigma para interactuar con otros elementos del sistema como el sistema operativo. Incluso los modelos de objetos que usan son diferentes. Se produce un serio problema de desadaptación de impedancias e interoperabilidad debido a los cambios de paradigma y/o conversiones de objetos que se realizan dependiendo del elemento del sistema con el que se trabaja.

En los sistemas convencionales, no orientados a objetos, se intentan aliviar estos problemas de manera parcial y con soluciones específicas mediante la introducción de capas de adaptación. Sin solucionar la problemática general, estas capas de hecho introducen complejidad y problemas adicionales en el sistema.

Otra aproximación más interesante, que es la que se investiga en este trabajo, es crear un sistema que ofrezca un soporte directo y común para el paradigma de la orientación a objetos, para crear un sistema integral orientado a objetos. En este sistema todos los elementos, interfaces de usuario, aplicaciones, lenguajes, bases de datos, incluso el sistema operativo comparten el mismo paradigma orientado a objetos.

Para demostrar las ventajas de esta aproximación, se describe una arquitectura software de referencia para construir un sistema integral, denominado Oviedo3, que se usará como plataforma de investigación y docencia en tecnologías orientadas a objetos. Se muestran ejemplos de sus ventajas, como la flexibilidad y cómo se pueden aplicar individualmente propiedades del sistema a otros sistemas. Los elementos más importantes son una máquina abstracta orientada a objetos con arquitectura reflectiva que proporciona portabilidad y el soporte del modelo de objetos para el sistema y un sistema operativo que extiende la máquina dotando a los objetos de manera transparente con las propiedades de seguridad, persistencia, concurrencia y distribución.

La viabilidad de la arquitectura se comprueba mediante la implementación de un prototipo de la máquina abstracta denominada Carbayonia. Además, se estudia en más profundidad la propiedad de la persistencia del sistema operativo, desarrollando un diseño concreto del sistema de persistencia como extensión de la máquina abstracta, e implementando un prototipo del mismo.

En el sistema integral resultante de esta arquitectura se pueden aprovechar en todas las partes del sistema las ventajas de la orientación a objetos, logrando un entorno de computación uniforme moderno, más flexible, coherente, intuitivo y fácil de usar.

Palabras clave

Orientación a objetos, tecnologías orientadas a objetos, modelo de objetos, máquinas abstractas, máquinas virtuales, sistemas operativos, sistemas operativos orientados a objetos, reflectividad, sistemas flexibles, sistemas extensibles, persistencia, sistemas integrales orientados a objetos, Oviedo3.

ABSTRACT

The adoption of the object-oriented paradigm is not done in an integral way in all the components of a computing system. There are languages, databases, user interfaces and other elements using object-orientation which have to change to other paradigms to interact with other elements of the system like the operating system. Even the object models are different. This produces a serious impedance-mismatch and interoperability problem, for the paradigm changes and/or object translations made depending on which element to work with.

The introduction of adaptation layers in conventional, non object-oriented systems, partially and with ad-hoc solutions tries to alleviate these problems. Without solving the overall problem, these layers in fact introduce additional complexity and problems into the system.

A more interesting approach that is researched in this thesis is to build a system offering direct and common support for the object-oriented paradigm to create an integral object-oriented system. In this system all the components, user interfaces, applications, languages, databases, even the operating system itself, share the same object-oriented paradigm.

To verify the advantages of this approach, a reference software architecture to build an integral system called Oviedo3 is described. This system will be used as a research and educational platform in object-oriented technologies. Examples of these advantages are shown, such as flexibility and how properties of the system can be individually applied to other systems. The most important elements are an object-oriented abstract machine with reflective architecture, which gives portability and support for the basic object model of the system and an operating system. The operating system extends the machine, providing objects transparently with security, persistence, distribution and concurrency properties.

The viability of the architecture is verified by the implementation of a prototype of the abstract machine, which is named Carbayonia. Besides, the persistence property of the operating system is examined with more detail. A concrete design of the persistence system as an extension of the abstract machine is developed, and a prototype is implemented.

The integral system resulting from this architecture takes advantage of the benefits of object-orientation in all the system components, achieving a achieving a modern and uniform computing environment, more flexible, coherent, intuitive and easy to use than conventional systems

Keywords

Object-orientation, object-oriented technology, object model, abstract machines, virtual machines, operating systems, object-oriented operating systems, reflectivity, flexible systems, extensible systems, persistence, object-oriented integral systems, Oviedo3.

ÍNDICE RESUMIDO

ÍNDICE

12 LA MÁQUINA ABSTRACTA CARBAYONIA 91

Capítulo 1
INTRODUCCIÓN

Este trabajo describe un sistema integral orientado a objetos que se está desarrollando en el Departamento de Informática de la Universidad de Oviedo. En concreto define las características que debe tener este sistema, así como la estructura de una arquitectura software que lo soporta. Se justifican las decisiones de diseño del sistema, y se profundiza en dos elementos importantes del sistema: la máquina abstracta reflectiva orientada a objetos que proporciona el modelo básico de objetos del sistema y la introducción de la persistencia en el sistema por medio de su sistema operativo orientado a objetos.

Este trabajo junto con el trabajo de otros componentes del grupo de investigación conformará el núcleo de este sistema, denominado Oviedo3. El objetivo es lograr una plataforma de investigación y experimentación en tecnologías orientadas a objetos sobre la que otros investigadores desarrollarán trabajos en áreas como compiladores de lenguajes orientados a objetos, bases de datos, interfaces de usuario, componentes software, etc.

1.1 Organización de este documento

Sistema integral orientado a objetos

La primera parte de se dedica a discutir aspectos relacionados con el sistema integral orientado a objetos y su estructura general. En el capítulo 2 se describen los problemas de la utilización del paradigma de la orientación a objetos en sistemas convencionales, que justifican la construcción de un sistema integral orientado a objetos. En el capítulo siguiente se definen los objetivos que tiene que alcanzar el sistema integral: portabilidad, heterogeneidad, multilenguaje, flexibilidad, transparencia. Tras realizar en el capítulo 4 una revisión de algunos sistemas relevantes a estos objetivos y localizar características interesantes de los mismos, en el capítulo 5 se especifica una arquitectura software para construir el sistema integral. Cada una de las propiedades introducidas en la arquitectura contribuye a lograr alguno de los objetivos del sistema integral: modelo único de objetos con identificador único de objetos, reflectividad, extensión en el espacio del usuario controlada. Cada propiedad es proporcionada por alguno de los elementos constituyentes del sistema: una máquina abstracta reflectiva orientada a objetos para el modelo de objetos que es extendida de manera transparente por un sistema operativo (formado a su vez por objetos) para la seguridad, persistencia, concurrencia y distribución. La combinación de ambos forma un espacio único donde residen todos los objetos del sistema. En el capítulo 6 se presenta el proyecto de investigación Oviedo3, cuyo objetivo es precisamente construir un sistema integral con esa arquitectura sobre el que desarrollar investigación y docencia en diferentes áreas de las tecnologías de objetos.

Gran parte del resto del trabajo se dedica a realizar el diseño más detallado de los elementos y características del sistema apuntados en la estructura general.

Modelo de objetos

Los dos capítulos siguientes se dedican al apartado del modelo de objetos del sistema. Se discute las ventajas de utilizar un modelo único en lugar de dar soporte a varios modelos de objetos y se definen las propiedades que debe incorporar este modelo único: básicamente las utilizadas en los modelos de las metodologías más populares de análisis y diseño orientado a objetos.

Máquina abstracta

La siguiente parte se destina a la máquina abstracta. En el capítulo 9 se resume el modelo de objetos de la máquina y los principios de diseño que se utilizarán. En el siguiente capítulo se hace una revisión panorámica de diferentes máquinas abstractas, cuyas buenas propiedades se usarán en el capítulo 11. En este capítulo se hace una descripción de una arquitectura de referencia para máquinas abstractas que den soporte al sistema integral, así como el potencial buen rendimiento que pueden tener.

La máquina abstracta Carbayonia

Los capítulos 12 y 13 describen una máquina abstracta concreta desarrollada siguiendo la arquitectura de referencia, compuesta de áreas para clases, referencias a instancias e instancias. El capítulo 12 define la estructura de la máquina y de su interfaz, mediante el lenguaje Carbayón de programación de la máquina. En el capítulo 13 se explica la implementación de un prototipo de la máquina y las técnicas utilizadas para ello.

Reflectividad

El capítulo 14 se dedica a estudiar los mecanismos que debe incorporar la máquina para ser extendida por el sistema operativo, especialmente la reflectividad. Se describen los conceptos fundamentales de la reflectividad, así como la selección justificada del tipo de reflectividad que se incorporará en la máquina y su posible implementación.

El sistema operativo SO4

El sistema operativo se compone de objetos normales de usuario que extiende la máquina. Las características del sistema operativo SO4 se describen brevemente en el capítulo , puesto que está todavía en desarrollo por otros investigadores. Se resume el diseño preliminar de la seguridad, la concurrencia y la distribución en el sistema.

Persistencia

La siguiente parte se dedica a la integración de la característica de la persistencia del sistema operativo en el sistema integral. En el capítulo 16 se trata con más detalle el concepto de persistencia y conceptos relacionados y atendiendo a ello se realiza un diseño más detallado del tipo de persistencia e implantación en el sistema. En el siguiente capítulo se mencionan algunos aspectos relacionados con la persistencia en el sistema y el diseño utilizado. La implementación de un prototipo del sistema de persistencia sobre la máquina abstracta se describe en el capítulo 18.

Aplicaciones

A continuación se dedican tres capítulos a mostrar ejemplos de propiedades y aplicación del sistema integral. En el capítulo 19 se muestra como las características del sistema le permiten conseguir los diferentes tipos de la taxonomía de sistemas flexibles. En el capítulo 20 se describen brevemente diferentes entornos de aplicación del sistema, desde sistemas empotrados a grandes sistemas distribuidos, posibles gracias a su flexibilidad. En el capítulo 21 se reseñan otros proyectos relacionados con el sistema integral y cómo algunos de los

resultados integrados en el sistema pueden aplicarse para mejorar otros sistemas, como por ejemplo la plataforma Java.

Conclusiones

En el capítulo 22 se concluye con una recapitulación sobre el principal resultado del trabajo: el sistema integral orientado a objetos y la arquitectura software que permite su existencia. A continuación se detallan una serie de conclusiones más específicas alcanzadas durante el desarrollo de los diferentes elementos que componen el sistema integral.

1.2 Conocimientos previos

En la realización de este trabajo se han supuesto una serie de conocimientos previos, necesarios para la comprensión del mismo. En primer lugar, se necesitan conocimientos del paradigma de la orientación a objetos y de las tecnologías de objetos en general: metodologías de análisis y diseño [Boo94, RBP+91], así como de lenguajes de programación orientados a objetos, especialmente C++ [Str91]. Los otros aspectos fundamentales del trabajo se refieren a elementos de sistema y máquinas abstractas, por lo que es aconsejable tener conocimientos generales de sistemas operativos [Dei90] y de procesadores de lenguajes [Cue95].

Capítulo 2

NECESIDAD DE UN SISTEMA INTEGRAL ORIENTADO A OBJETOS

Actualmente las tecnologías orientadas a objetos son una realidad patente dentro de la industria del software. Aunque no es la "bala de plata" o bálsamo de Fierabrás que solucione los problemas del software, la orientación a objetos (OO) es aceptada como el paradigma de software más adecuado hasta ahora. Existe una extensa bibliografía acerca del tema y obras de referencia "clásicas" como las de Booch [Boo91, Boo94] y Rumbaugh [RBP+91]. Especialmente la primera de estas obras marca un consenso generalizado sobre las características fundamentales del modelo de objetos[1].

Existe pues un paradigma estandarizado de hecho que es utilizado en el desarrollo de aplicaciones por las ventajas que tiene. Entre estas ventajas está el uso de un único paradigma en las diferentes fases del ciclo de vida del software [Joy97]: análisis de requerimientos OO, análisis OO, diseño OO e implementación con lenguajes de programación OO. Los sistemas resultantes son más fáciles de entender y de desarrollar pues se reducen los saltos semánticos al aplicar los mismos conceptos en todas las fases.

2.1 El problema de la desadaptación de impedancias o salto semántico

A pesar del éxito de las tecnologías orientadas a objetos (TOO) en el ámbito del desarrollo de aplicaciones, la adopción del paradigma OO no se ha hecho de una manera integral en todos los elementos que componen un sistema de computación. Existe un grave problema de desadaptación de impedancias, o salto semántico entre los diferentes elementos del sistema. Este problema se produce en el momento en que hay que realizar un cambio o adaptación de paradigma cuando un elemento (por ejemplo una aplicación OO) debe interactuar con otro (por ejemplo el sistema operativo).

2.1.1 Abstracciones no adecuadas de los sistemas operativos

Por una parte, el hardware convencional sobre el que deben funcionar las TOO sigue aún basado en versiones evolucionadas de la arquitectura Von Neumann. Los sistemas operativos ofrecen abstracciones basadas en este tipo de hardware, más adecuadas para el paradigma procedimental estructurado. Así los contextos de ejecución y espacios de direcciones de los procesadores convencionales tienen sus abstracciones correspondientes en el sistema operativo: procesos y memoria virtual, hilos de ejecución y mecanismos de comunicación entre procesos (IPC, *Inter-Process Communication*).

[1] Incluso se está intentando fusionar estas dos metodologías con más uso en un único método unificado [BR95]. Posteriormente se ha unido al proyecto Jacobson, autor de otra de las metodologías más usadas [JCJ+92] y se le ha denominado lenguaje de modelado unificado [BRJ96] (UML, *Unified Modeling Language*).

Las aplicaciones OO se estructuran básicamente como un conjunto de objetos que colaboran entre sí mediante la invocación de métodos. La distancia semántica entre esta estructura y las abstracciones que ofrecen los sistemas operativos es muy grande.

Por ejemplo, los objetos de los lenguajes en que se desarrollan las aplicaciones suelen ser de grano fino (tamaño pequeño). Sin embargo, el elemento más pequeño que manejan los sistemas operativos es el de un proceso asociado a un espacio de direcciones, de un tamaño mucho mayor. Esto obliga a que sea el compilador del lenguaje el que estructure los objetos internamente dentro de un espacio de direcciones. Pero por ejemplo, al desarrollar aplicaciones distribuidas esto ya no funciona puesto que el compilador desconoce la existencia de otros objetos fuera de un espacio de direcciones. Por otro lado, los modelos de concurrencia de objetos suelen disponer de actividades ligeras u objetos activos que deben hacerse corresponder de manera forzada con la noción más gruesa de proceso.

2.1.1.1 Comunicación de alto nivel entre objetos situados en diferentes espacios de direcciones

Un ejemplo de este problema es el mencionado de la comunicación de alto nivel entre objetos de diferentes espacios de direcciones. Se plantea cuando un objeto cliente tiene que invocar un método que ofrece un objeto servidor y los dos objetos no están dentro del mismo espacio de direcciones (métodos remotos), es decir, no están bajo la esfera de control del mismo proceso y compilador. Por ejemplo se da este caso cuando los dos objetos están en máquinas diferentes. Cuando los objetos están dentro del mismo proceso no hay ningún problema, puesto que la invocación de métodos la resuelve el propio compilador directamente. Pero en este caso el compilador no tiene ningún mecanismo que permita realizar la invocación de métodos en espacios de direcciones diferentes, y se debe recurrir a los mecanismos que ofrezca el sistema operativo. Sin embargo, el mecanismo de comunicación de los sistemas operativos no se adapta al paradigma de la OO, ya que están orientados a comunicar procesos. Un ejemplo de estos mecanismos son las tuberías o *pipes* del SO Unix. El programador se ve obligado a abandonar el paradigma OO y encajar de manera antinatural el mecanismo de comunicación OO (invocación de métodos) sobre un mecanismo totalmente distinto pensado para otra finalidad.

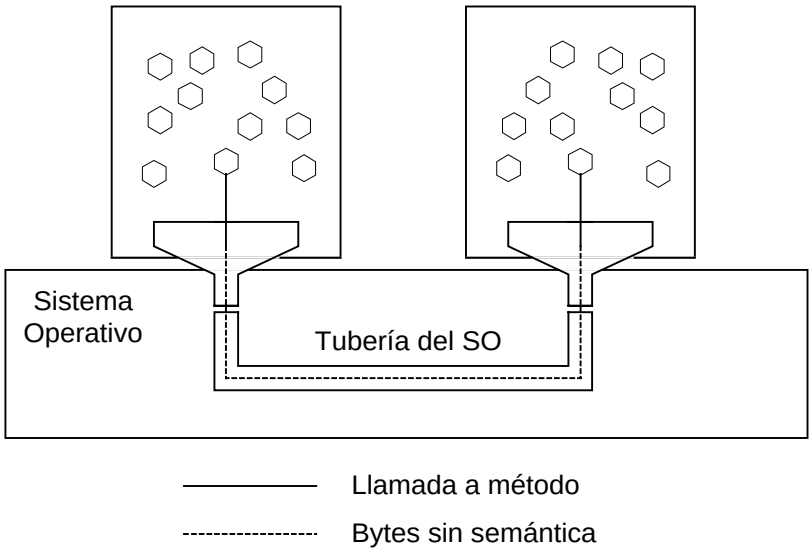


Figura 2.1 Comunicación entre objetos mediante mecanismos de bajo nivel del sistema operativo

La implementación de la OO sobre estos sistemas que no tienen soporte explícito para objetos es muy complicada, como demuestra la implementación del lenguaje Eiffel concurrente [KB90]. Para solucionar los problemas anteriores se acaba recurriendo a la interposición de capas adicionales de software que adapten la gran diferencia existente entre el paradigma OO y los sistemas actuales.

Un ejemplo de software intermedio que en este caso pretende solucionar (entre otros) el problema de la comunicación de alto nivel entre objetos situados en diferentes espacios de direcciones es COM (*Component Object Model,* Modelo de objetos de componentes) [Rog96] y *CORBA (Common Object Request Broker Architecture,* Arquitectura común de intermediarios entre objetos) [OMG95, OMG97]. En CORBA, básicamente se utiliza un gestor de objetos u ORB (*Object Request Broker*, intermediario de peticiones entre objetos) que intermedia en la invocación de método entre el objeto cliente y el servidor. Existe un código de adaptación en el proceso cliente para acceder al ORB local cuando se llama a un método remoto. El ORB se ocupa de hacer llegar la llamada al objeto servidor a través del ORB de la máquina remota, donde se encuentra también un código de adaptación similar.

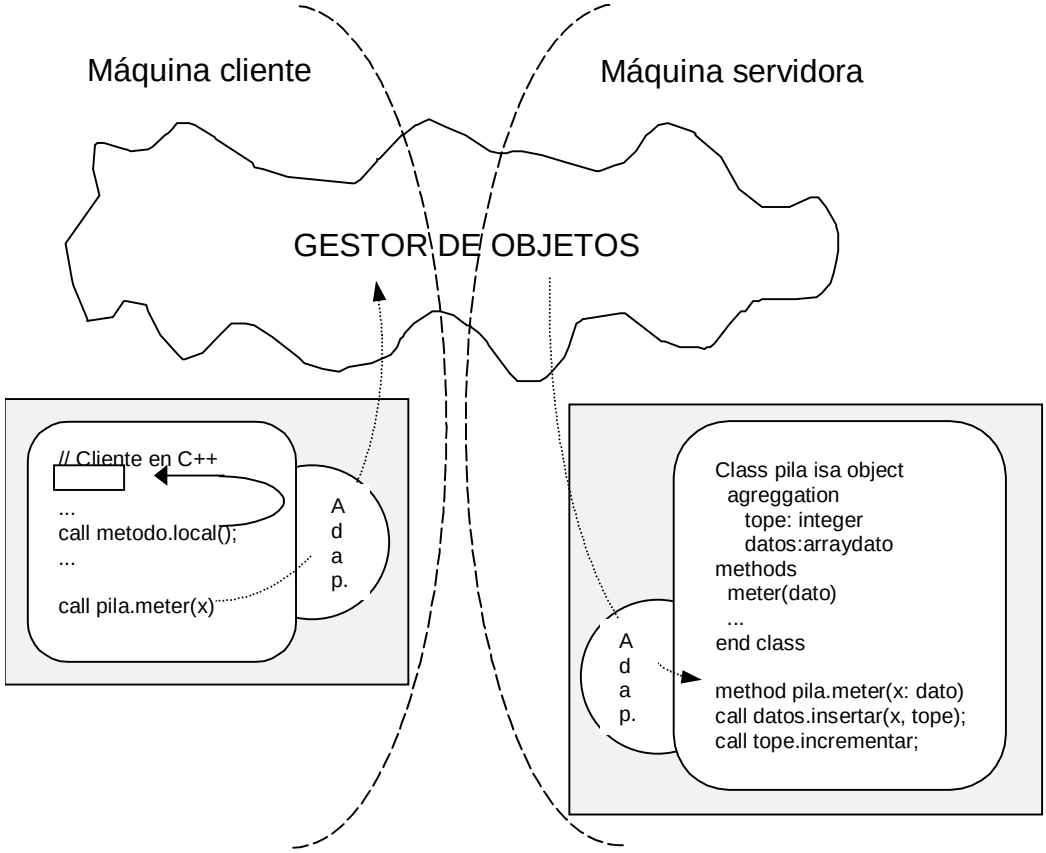


Figura 2.2 Comunicación de alto nivel mediante un gestor de objetos

2.1.2 Desadaptación de interfaces

Otro problema se produce cuando las aplicaciones OO necesitan utilizar los servicios del sistema operativo. La interfaz de utilización de estos servicios está enfocado al paradigma procedimental, normalmente en forma de una llamada al sistema (llamada a procedimiento). Además, como ya se mencionó anteriormente, muchos de estos servicios (por ejemplo la comunicación entre procesos) no sigue el paradigma OO.

El resultado es que el programador/usuario del sistema se ve obligado a utilizar dos paradigmas diferentes en el desarrollo de las aplicaciones. Uno para la parte fundamental de la

aplicación, orientado a objetos, y otro totalmente diferente para la interacción con el sistema operativo.

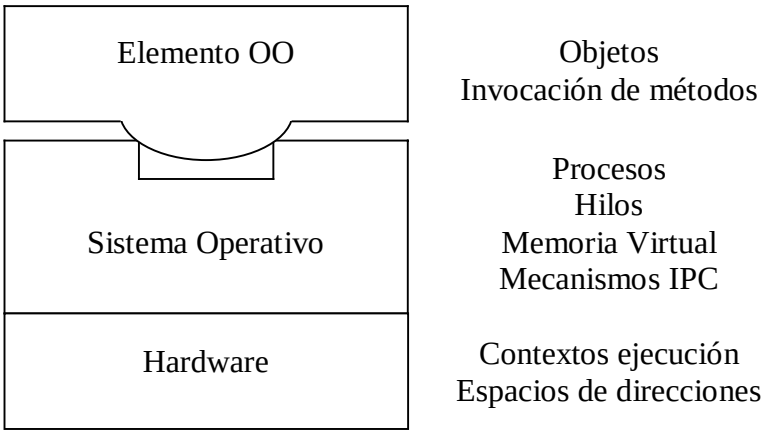


Figura 2.3 Desadaptación entre las aplicaciones OO y la interfaz del sistema operativo

Por ejemplo, en el caso anterior, para la utilización de una tubería en Unix no se puede utilizar la OO. Es necesario utilizar llamadas de procedimiento a la interfaz procedimental del SO, las llamadas al sistema. O bien para utilizar los ficheros que proporciona el sistema operativo, etc.

Esta dualidad de paradigmas produce una disminución de la productividad de los programadores. Cuantos más sean los conceptos diferentes que deba conocer el programador, peor será la comprensión y el dominio de los mismos. La necesidad de aplicar otro paradigma diferente a la orientación a objetos hace que no se saque todo el partido posible a la misma.

También se pueden utilizar capas de adaptación, por ejemplo encapsulando la interfaz del sistema operativo mediante una librería de clases. Así las llamadas al sistema operativo se hacen indirectamente a través de objetos de la librería de clase, de manera OO. La librería, a su vez, se encarga de llamar al sistema operativo.

Un ejemplo de estas capas de adaptación es la librería MFC (*Microsoft Foundation Classes* [Mic97a, Mic97b]), que entre otras cosas, encapsula ciertos servicios del sistema Windows en un marco de aplicación.

2.2 El problema de la interoperabilidad entre modelos de objetos

Incluso aunque diferentes elementos del sistema usen el paradigma de la orientación a objetos, puede haber problemas de desadaptación entre ellos. Es común la existencia de diferentes lenguajes de programación OO, bases de datos OO, interfaces gráficas OO, etc. Sin embargo, el modelo de objetos que utiliza cada uno de ellos suele ser diferente. Aunque sean OO, las propiedades de los objetos de cada sistema pueden diferir, por ejemplo un modelo puede tener constructores y otros no, etc.

Por ejemplo, una aplicación desarrollada usando el lenguaje C++, con el modelo de objetos C++ no tiene ningún problema de comunicación con sus propios objetos. Cuando se pretende usar objetos de otro lenguaje de programación, o interactuar con objetos que no están en el mismo proceso, o bien con una base de datos orientada a objetos aparece un problema de interoperabilidad. Este es debido a que el modelo de objetos del C++ no tiene por qué ser totalmente compatible con el de los otros elementos.

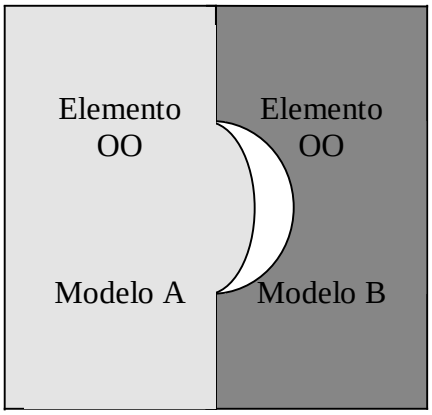


Figura 2.4 Desadaptación entre modelos de objetos diferentes

De nuevo se recurre a la introducción de capas de software de adaptación para solucionar el problema. CORBA también es un ejemplo de este tipo de software. CORBA define un modelo de objetos propio, con unos tipos de datos básicos, etc. Este modelo sólo especifica el interfaz de un objeto, nunca la implementación. Para cada lenguaje se define una correspondencia (*mapping*) entre el modelo de objetos de cada lenguaje y su interfaz dentro del modelo CORBA. Una vez dada esta correspondencia ya se tiene información suficiente para generar el software de adaptación requerido, como se mostró anteriormente. Este se incorpora a la implementación de los objetos cliente y servidor. Los ORB en las máquinas cliente y servidora junto con este software incluido en los programas cliente y servidor permiten la invocación de métodos remotos. Como el objeto cliente invoca la interfaz CORBA del objeto remoto, que es independiente del modelo de objetos remoto, las diferencias entre los modelos de objetos quedan ocultas, permitiendo la interoperabilidad.

2.3 Problemas de las capas de adaptación sobre sistemas tradicionales

Para salvar los problemas que presenta el paradigma de la orientación a objetos, se utilizan soluciones basadas en la adición de una serie de capas de software de adaptación a un sistema operativo tradicional. Esto provoca una serie de inconvenientes:

- **Disminución del rendimiento global del sistema**, a causa de la sobrecarga debida a la necesidad de atravesar todas estas capas software para adaptar los paradigmas. Además, para la propia implementación de estas capas que ofrecen soporte de objetos se necesitan también los servicios del sistema operativo, con lo que nos encontramos con un nuevo salto entre paradigmas
- **Falta de uniformidad y transparencia**. En la práctica, estas soluciones no son todo lo transparente que deberían de ser de cara al usuario. Por ejemplo, la escritura de objetos que vayan a funcionar como servidores suele realizarse de manera diferente del resto de los objetos, como en el caso de CORBA. Esto no se corresponde con la

filosofía de uso y de programación OO, que es más general que este paradigma cliente/servidor que obliga a clasificar los objetos en clientes o servidores a priori.

- **Pérdida de portabilidad y flexibilidad**. La falta de uniformidad produce una pérdida de portabilidad y flexibilidad. El programador se ve forzado a utilizar ciertas convenciones, formas especiales de programar, etc. que impone el uso de la propia capa que reduce la portabilidad de los objetos. Por ejemplo, en el caso de utilizar la librería MFC para utilizar las funciones de Windows se impone una determinada manera de construir los programas, etc. que hace que el código quede ligado indisolublemente a esta librería. Otro ejemplo es el caso de un programa que tenga que utilizar objetos CORBA, su programación es totalmente distinta que si se va a utilizar objetos COM. Además, estos programas sólo podrán funcionar en máquinas a las que también se haya portado los sistemas de adaptación que se utilicen.
- **Aumento de la complejidad del sistema**. La adición de estas capas introduce más complejidad en los sistemas. Por un lado aumentan los problemas de integración entre las diferentes capas, errores en la misma, posibilidades de fallos, etc. Por otro lado, los sistemas se hacen más difíciles de comprender al intervenir tantos elementos y tan dispares.
- **Soluciones parciales**. En general, estas capas sólo ofrecen soluciones parciales a alguno de los problemas presentados. Por ejemplo, para la invocación de métodos remotos pueden existir soluciones, sin embargo no solucionan la utilización OO de recursos del sistema operativo, etc. Esto obliga a combinar diferentes capas para solucionar diferentes aspectos. CORBA es un ejemplo que soluciona la interoperabilidad entre objetos, pero colocándolos en espacios de direcciones diferentes. En el desarrollo intra-programa no puede usarse CORBA, con lo que se pierde la interoperabilidad entre objetos de diferentes lenguajes.
- **Pérdida de productividad**. Todo lo anterior lleva a una pérdida de productividad. La programación se hace a veces engorrosa e incluso compleja. Se distrae al programador con detalles técnicos de la capa (o capas) de adaptación y se le impide concentrarse en la solución de los problemas.

En general, el problema es debido a que los sistemas operativos no soportan el concepto de objeto. Por tanto este concepto es proporcionado internamente por los compiladores de cada lenguaje, estructurándolo sobre las abstracciones que proporciona el sistema operativo. El desconocimiento de la existencia de objetos por parte del SO provoca una serie de problemas que no tienen una solución satisfactoria con la adición de “parches” o capas de software adicional a los sistemas existentes.

2.4 Sistema integral orientado a objetos

Una manera de resolver este problema es crear un sistema homogéneo en el que se utilice en todos sus elementos el mismo paradigma de la orientación a objetos y se dé soporte nativo a los mismos: un sistema integral orientado a objetos. Oviedo3 [CIA97] es un proyecto de investigación que pretende construir un sistema experimental basado en este principio.

En un sistema integral como éste se crea un entorno de computación en el que todos los elementos: lenguajes, aplicaciones, compiladores, interfaces gráficas, bases de datos, etc. hasta los más cercanos a la máquina comparten el mismo paradigma de la orientación a objetos. El sistema comprende el concepto de objeto y es capaz de gestionar directamente objetos.

Al usar el mismo paradigma no existe desadaptación. Desde una aplicación OO tanto el acceso a los servicios del sistema como la interacción con otros objetos se realizan utilizando los mismos términos OO.

Al dar soporte directo a objetos, es decir, al gestionar directamente tanto la representación de los objetos como su utilización se soluciona el problema de la interoperabilidad. Los objetos no son conceptos que sólo existen dentro de los procesos del SO y que sólo son conocidos por el compilador que creó el proceso. Al ser gestionados por el propio sistema pueden "verse" unos a otros, independientemente del lenguaje y del compilador que se usó para crearlos.

No son necesarias capas de adaptación con lo que se reduce la complejidad conceptual y técnica del sistema. Los usuarios/programadores se ven liberados de preocuparse por detalles técnicos y contraproductivos cambios de paradigma y pueden concentrarse en el aspecto más importante que es la resolución de los problemas usando la OO.

2.5 Resumen

Los sistemas actuales no son adecuados para explotar al máximo el paradigma de la orientación a objetos. Los problemas de interoperabilidad y de desadaptación de impedancias provocan una proliferación de capas de software intermedio adicional que intentan aliviar estos problemas. Esto produce una pérdida de eficiencia del sistema y una disminución de la productividad de los usuarios/programadores al aumentar el número y la complejidad de los conceptos que deben manejar. Un sistema integral orientado a objetos en el que todos los elementos comparten el mismo paradigma de la orientación a objetos es una manera prometedora de solucionar el problema.

Capítulo 3

REQUISITOS DEL SISTEMA INTEGRAL ORIENTADO A OBJETOS

La idea de crear un sistema que dé soporte directo y utilice exclusivamente el paradigma de la orientación a objetos nos conduce a la siguiente definición de Sistema integral orientado a objetos:

Un sistema integral orientado a objetos ofrece al usuario un entorno de computación que crea un mundo de objetos: *un único espacio de objetos, virtualmente infinito, en el que un conjunto de objetos homogéneos coopera intercambiando mensajes independientemente de su localización, y donde los objetos residen indefinidamente hasta que ya no son necesitados*.

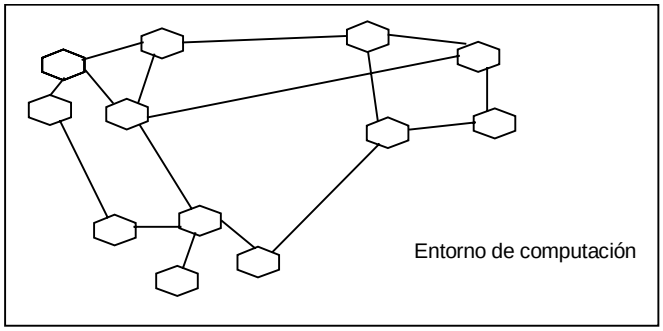


Figura 3.1 Entorno de computación compuesto por un conjunto de objetos homogéneos.

Este sistema permitirá aprovechar al máximo las conocidas ventajas de la orientación a objetos: [Boo94] reutilización de código, mejora de la portabilidad, mantenimiento más sencillo, extensión incremental, etc. en todos los elementos del sistema y no solo en cada aplicación OO como en los sistemas convencionales.

Los requisitos que deben estar presentes en un entorno como éste son los siguientes:

- **Uniformidad conceptual en torno a la orientación a objetos**
- **Transparencia: persistencia y distribución**
- **Heterogeneidad y portabilidad**
- **Seguridad**
- **Concurrencia**
- **Multilenguaje / Interoperabilidad**
- **Flexibilidad**

3.1 Uniformidad conceptual en torno a la orientación a objetos

El único elemento conceptual que debe utilizar el sistema es un mismo paradigma de orientación a objetos. El sistema proporcionará una única perspectiva a los usuarios/programadores: la de objetos, que permite comprender más fácilmente sistemas cada vez más complicados y con más funcionalidad [YMF91].

Modo de trabajo exclusivamente orientado a objetos

La única abstracción es, por tanto, el objeto (de cualquier granularidad), que encapsula toda su semántica. Lo único que puede hacer un objeto es crear nuevas clases que hereden de otras, crear objetos de una clase y enviar mensajes[1] a otros objetos. Toda la semántica de un objeto se encuentra encapsulada dentro del mismo.

Homogeneidad de objetos

Todos los objetos tienen la misma categoría. No existen objetos especiales. Los propios objetos que den soporte al sistema no deben ser diferentes del resto de los objetos.

Esta simplicidad conceptual hace que todo el sistema en su conjunto sea fácil de entender, y se elimine la desadaptación de impedancias al trabajar con un único paradigma. La tradicional distinción entre los diferentes elementos de un sistema: hardware, sistema operativo y aplicaciones de usuario se difumina.

3.2 Transparencia

El sistema debe hacer transparente la utilización de los recursos del entorno al usuario y en general todas las características del sistema, en especial:

Distribución

El sistema debe ser inherentemente distribuido, ocultando al usuario los detalles de la existencia de numerosas máquinas dentro de una red, pero permitiendo la utilización de las mismas de manera transparente en el entorno de computación. Con términos de objetos, los objetos podrán residir en cualquier máquina del sistema y ser utilizados de manera transparente independientemente de cual sea esta.

Persistencia

El usuario no debe de preocuparse de almacenar los objetos en memoria secundaria explícitamente. El sistema se debe de ocupar de que el usuario perciba un único espacio de objetos y transparentemente almacenarlos y recuperarlos de la memoria secundaria.

3.3 Heterogeneidad y portabilidad

El sistema no debe obligar a la utilización de un determinado modelo de máquina para su funcionamiento. Debe tenerse en cuenta la existencia de numerosos tipos de máquinas dentro de la misma red de trabajo de una organización, que posiblemente sean incompatibles entre sí.

Por la misma razón, para llegar al mayor número de máquinas posible, interesa que el esfuerzo para portar el propio sistema de una máquina a otra sea el menor posible.

[1] Se utilizarán indistintamente las expresiones envío de mensaje, llamada o invocación de método y llamada o invocación de operación.

3.4 Seguridad

El entorno de computación debe ser seguro y protegerse frente a ataques maliciosos o errores lógicos. El sistema dispondrá de un mecanismo de protección que permita controlar el acceso no autorizado a los objetos. Por supuesto, este mecanismo debe integrarse de manera uniforme dentro del paradigma OO.

3.5 Concurrencia

Un sistema moderno debe presentar un modelo de concurrencia que permita la utilización de los objetos aprovechando el paralelismo. Dentro de una misma máquina aprovechando la concurrencia aparente y en sistemas distribuidos o multiprocesador la concurrencia real.

3.6 Multilenguaje / Interoperabilidad

El sistema no debe restringir su utilización a sólo un lenguaje de programación. De esta manera la mayoría de los programadores no necesitarán aprender otro lenguaje. Además, algunos problemas se resuelven mejor en un determinado lenguaje que en otros.

Sin embargo, la interoperabilidad entre los diferentes lenguajes debe quedar asegurada, para evitar los problemas de desadaptación anteriores.

3.7 Flexibilidad

Para un sistema experimental y de investigación como éste, la flexibilidad es muy importante. El sistema debe ser fácil de adaptar a entornos diferentes, como por ejemplo sistemas empotrados, sistemas sin disco duro, sistemas multiprocesador. También a los requisitos de las aplicaciones: algunas no necesitarán persistencia, otras una forma especial de la misma, etc. Muy a menudo se debe experimentar reemplazando o añadiendo nuevos servicios para comprobar el comportamiento del sistema, etc. En resumen, el sistema debe permitir eliminar, añadir o modificar funcionalidad de manera sencilla.

Capítulo 4

PANORÁMICA DE SISTEMAS OPERATIVOS RELACIONADOS CON LOS OBJETIVOS DEL SISTEMA INTEGRAL ORIENTADO A OBJETOS

En este capítulo se revisan diferentes sistemas operativos que comparten algunos de los objetivos necesarios para lograr un sistema integral orientado a objetos. Se trata de detectar características comunes y estrategias que sean de utilidad para el diseño de una arquitectura que soporte el sistema integral.

Los sistemas revisados intentan ser representativos de las diferentes tendencias de diseño de sistemas operativos actuales y son una selección de los sistemas examinados. Los aspectos relevantes de estos y otros sistemas para apartados específicos del sistema integral se examinarán posteriormente al tratar estos apartados.

4.1 COOLv2

COOLv2 [LJP93] es una capa de soporte para objetos distribuidos construida sobre el micronúcleo CHORUS [RAA+92]. El proyecto se proponía reducir la desadaptación de impedancias entre las abstracciones de los lenguajes y las abstracciones proporcionadas por el sistema. Para ello extiende el micronúcleo CHORUS con abstracciones más adecuadas para sistemas orientados a objeto, con la idea de reducir la ineficiencia de capas software añadidas.

Abstracciones base

Las abstracciones incluidas en el sistema base son los agrupamientos (***cluster***) y los **espacios de contexto** que abstraen los micronúcleos distribuidos y el almacenamiento persistente.

Un *cluster* es un conjunto de regiones de memoria respaldadas en disco que serán usadas para colocar sobre ellas los objetos. Los *cluster* se mapean sobre espacios de direcciones virtuales distribuidos que forman un espacio de contexto

Soporte genérico en tiempo de ejecución

Sobre las abstracciones base se coloca una capa de software denominada **GRT** (*Generic Run Time,* **soporte genérico en tiempo de ejecución**). El GRT implementa la noción de objetos usando un modelo de objetos básico organizados en clases. También proporciona invocación de métodos y actividades (hilos).

Soportes específicos para lenguajes

Este GRT se complementa con **soportes en tiempo de ejecución específicos** para diferentes lenguajes (C++, Eiffel, etc.). La combinación del GRT con el soporte específico de un lenguaje proporciona el soporte para el modelo de objetos de un lenguaje determinado.

Invocación de objetos

Existen dos maneras de invocar los objetos. Dentro de un *cluster* se accede a los objetos locales utilizando las referencias propias de cada lenguaje, que serán direcciones de memoria virtual (punteros). Para invocar a objetos que no están en el *cluster*, se utiliza un **objeto de interfaz** (*proxy* o representante) que representa al objeto remoto y al que se accede usando un identificador global persistente. El código para estos representantes es generado por compiladores especiales modificados para ello.

Concurrencia

Los objetos se tratan como objetos pasivos. Los hilos de ejecución (**actividades**) viajan de objeto en objeto mediante las invocaciones de métodos.

Persistencia

Los *cluster* se hacen persistentes cuando no hay ningún hilo activo sobre ellos. Se realiza una recolección de basura de los *cluster* no usados en disco.

Colaboración entre los soportes en tiempo de ejecución

Existe un mecanismo de colaboración ad-hoc entre el GRT y los soportes de los lenguajes (retrollamadas o *up-calls*). Este mecanismo es necesario porque el GRT necesita saber en algunos casos información que sólo conoce el soporte del lenguaje. Por ejemplo, al hacer un *cluster* persistente pregunta al soporte específico las referencias que contienen los objetos.

4.1.1 Crítica

A pesar de dar soporte a objetos, no contempla algunos objetivos del sistema integral como la portabilidad y la heterogeneidad. A continuación se mencionan otros problemas:

4.1.1.1 Falta de uniformidad en la Orientación a Objetos

El uso de objetos no tiene una uniformidad total. Por ejemplo se usan dos maneras de referenciar a los objetos, en función de su situación. Dentro del mismo módulo se usan las referencias de cada lenguaje, y para acceder a un objeto remoto se utiliza un mecanismo de acceso diferente basado en un identificador global. Otro ejemplo es la utilización de varios modelos de objetos, aunque esto hace posible soportar cualquier lenguaje, introduce un factor de complejidad adicional en el sistema: la necesidad de comprender todos los modelos de objetos usados para comprender un sistema en su conjunto.

4.1.1.2 Orientación a objetos sólo en el espacio del usuario: pérdida de flexibilidad en el sistema

El soporte para objetos sólo existe en las aplicaciones de usuario. El resto del sistema es convencional y por tanto hay que acceder mediante las interfaces no orientadas a objetos del mismo. Las ventajas de la orientación a objetos para la extensibilidad, etc. no se pueden aplicar al sistema base. Se pierde flexibilidad. Esto se ve incluso en el soporte de los modelos de objetos, el soporte básico se comunica con el soporte específico no mediante un mecanismo genérico OO, si no mediante una interfaz específica tradicional de retrollamadas.

4.1.1.3 Problemas de semántica y de interoperabilidad del soporte multimodelo

El soporte de varios modelos produce un problema en la semántica de los objetos. El estado de un objeto no es fácilmente conocido puesto que su representación está repartida entre el soporte básico y el soporte específico del modelo de cada lenguaje. El sistema no tiene conocimiento exacto de los objetos que existen en el mismo. Además existen problemas de interoperabilidad entre objetos de diferentes modelos.

4.2 SPACE

SPACE [PBK91] es un sistema operativo de los denominados exonúcleo o de núcleo mínimo. La filosofía de estos sistemas es la de proporcionar una funcionalidad mínima dentro del núcleo. Las abstracciones tradicionales de los sistemas operativos se construyen fuera del núcleo sobre las primitivas del exonúcleo.

Primitivas mínimas de SPACE

Los **espacios** representan direcciones de memoria del procesador. Su función es mapear estas direcciones sobre direcciones de E/S y portales. Un **portal** es una generalización de los mecanismos de fallo de página, y especifica un punto de entrada a un dominio (cambio de contexto). Los **dominios** permiten calificar los espacios con un vector de bits de protección más general que los bits de lectura, escritura y modificación de los sistemas de paginación.

Abstracciones a medida en el espacio del usuario

Usando estas primitivas pueden construirse las abstracciones de proceso y memoria virtual de sistemas como UNIX.

Por otro lado, estas abstracciones pueden extenderse, modificarse o reemplazarse, al pertenecer al espacio de usuario y estar fuera del núcleo. De esta manera se puede construir un sistema a medida de los requisitos de una determinada aplicación, entorno o usuario.

En concreto este mecanismo podría ser utilizado para dar soporte a uno o varios modelos de objetos para ser utilizados por las aplicaciones.

4.2.1 Crítica

No existe el concepto de objeto con toda la semántica, organizado en un modelo, etc. En realidad este tipo de sistemas podrían utilizarse para construir sobre ellos un sistema integral, puesto que las abstracciones mínimas que proporcionan no bastan por sí solas para cumplir las necesidades de un sistema integral.

4.2.2 Características interesantes

Como elemento más interesante se destaca la utilización de abstracciones dentro del espacio del usuario.

4.2.2.1 Abstracciones en el espacio del usuario

Todas las abstracciones necesarias se proporcionan en el espacio del usuario. Esto permite cambiar dinámicamente las mismas, dotando de más flexibilidad al sistema. En el sistema integral se debe procurar que todos los elementos que lo compongan pertenezcan conceptualmente al espacio del usuario, para conseguir el máximo de flexibilidad.

4.3 Tigger

Tigger [Cah96a] es un marco de aplicación (*framework*) para la construcción de una familia de sistemas operativos para soporte de objetos que puedan ajustarse a las aplicaciones en el campo de ingeniería concurrente y juegos multiusuario de nueva generación.

Una instanciación del marco genera un sistema operativo concreto (un miembro de la familia), ajustado a los requerimientos de la plataforma destino y las aplicaciones específicas a las que dará soporte. El objetivo fundamental es permitir el soporte de diferentes modelos de objetos para programación distribuida y persistente sin duplicaciones innecesarias. Cada miembro, además, podrá soportar el mismo modelo de diferente manera dependiendo de la plataforma destino.

Estructura general del marco de aplicación

Cada una de las siguientes categorías de clases se encarga de dar soporte a un subconjunto de las abstracciones fundamentales de Tigger.

- **Owl – Objetos persistentes y distribuidos**. Es la encargada de dar soporte a los diferentes modelos de objetos. Usa una estrategia parecida a la de COOLv2, basada en el acoplamiento entre un **soporte genérico en tiempo de ejecución** (GRT, *Generic Run Time*) que es complementado con un **soporte específico** de cada lenguaje (LSRT, *Language Specific Run Time*). Sin embargo, en lugar de proporcionar un único GRT, permite utilizar varios diferentes, en función de las necesidades de cada aplicación.
- **Roo – Hilos**. Soporta hilos y mecanismos de sincronización relacionados. Los soportes de los lenguajes (LSRT) pueden usarla directamente.
- **Kanga – Comunicaciones**. Soporta una abstracción para las comunicaciones.
- **Eeyore – Almacenamiento**. Proporciona contenedores y objetos de almacenamiento. Solo es usada directamente por Owl.
- **Robin – Protección**.

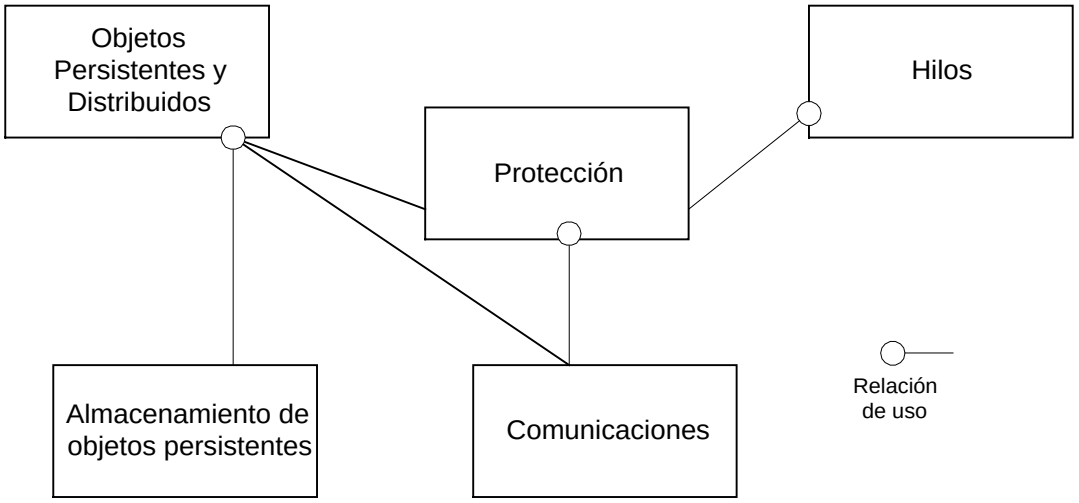


Figura 4.1 Estructura general del marco de aplicación del sistema Tigger.

4.3.1 Crítica

La estructura de soporte de varios modelos es similar a la de COOLv2. Los inconvenientes son muy parecidos a los de ese sistema: falta de uniformidad en la OO, problemas de interoperabilidad y la ocultación al sistema de parte de la semántica de los objetos en los soportes específicos. Esto lo hace adecuado para soportar aplicaciones muy específicas de manera especial, pero no para un sistema integral.

4.3.2 Características interesantes

La utilización de una jerarquía de clases como medio de descripción del sistema es lo más destacable.

4.3.2.1 Jerarquía de clases para describir el sistema

Es interesante la idea de construir el sistema mediante la OO con una jerarquía de clases. Esto permite aprovechar las ventajas de la OO en la construcción del sistema. En el caso del sistema integral, conviene implementar los elementos del mismo de esta manera.

4.4 Sombrero

Sombrero [SMF96] es un sistema operativo que pretende aprovechar el nuevo hardware que proporciona espacios de direcciones virtuales de 64 bits. Este tipo de hardware permite la utilización de un vasto espacio de direcciones único.

Ventajas de un sistema de espacio de direcciones único

En este tipo de sistemas de **espacio de direcciones único** (SAS, *Single Address Space*) tiene una serie de ventajas en diferentes aspectos:

- **Hace innecesarios los procesos**. Los procesos son una abstracción que se usa al tener espacios de direcciones pequeños que se toman como dominios de protección. Esto provoca una sobrecarga al tener que cruzar estos espacios mediante cambios de contexto, llamadas *IPC*, puertos, etc. Al tener un espacio único, se puede utilizar una llamada a procedimiento normal del hardware para cruzar los dominios de protección. Con una comprobación por hardware especializado de la protección no existe sobrecarga.
- **Persistencia**. Se pueden eliminar los ficheros. Al tener tal cantidad de direcciones, puede utilizarse una dirección de memoria virtual para identificar un elemento, y esta nunca necesitará reutilizarse. Se pueden mapear trozos de la memoria en almacenamiento secundario haciéndolos persistentes de manera transparente. En los sistemas normales, no hay tantas direcciones disponibles y estas al final deben reutilizarse, obligando a almacenar la información explícitamente en disco. Por otro lado, los punteros (direcciones de memoria virtual) siguen siendo válidos en disco, pues nunca cambian.
- **Distribución**. Se puede extender el espacio de direcciones único a todos los ordenadores de la red. El espacio de direcciones global se particiona distribuyéndolo entre los diferentes ordenadores. De esta manera el funcionamiento del sistema se extiende transparentemente a la red.

Abstracciones del sistema Sombrero

Las abstracciones fundamentales que proporciona Sombrero siguen la línea anterior. El espacio de direcciones puede particionarse dinámicamente entre los ordenadores de la red. Los **objetos de memoria** representan un trozo de direcciones de memoria. Estos se pueden mapear en **dominios de protección**, que proporcionan una suerte de memoria compartida protegida. La protección se realiza mediante **listas de control de acceso**, que se implementan de manera distribuida. Un hardware especializado controla la protección en cada acceso a memoria o instrucción, incluyendo las que cruzan los dominios de protección.

4.4.1 Crítica

El soporte de un modelo de objetos completo no entra dentro de las características de diseño de este sistema, que en este aspecto tiene una estructura convencional de procesos y datos. Además se necesita un hardware especial para su funcionamiento, lo cual perjudica la portabilidad y la heterogeneidad.

4.4.2 Características interesantes

De interés especial es la obtención de un espacio único de direcciones.

4.4.2.1 Espacio único

La abstracción de un espacio único de direcciones proporciona toda la funcionalidad del sistema en el espacio de usuario. El no hacer distinciones entre los elementos de usuario y del sistema favorece la flexibilidad dinámica en el sistema. No hay que hacer decisiones arbitrarias de en qué espacio colocar una determinada funcionalidad. Para el sistema integral, esto se traduce en conseguir un espacio de objetos único, en el que residan todos los objetos, independientemente de cuál sea su funcionalidad: simples objetos de usuario u objetos del sistema.

4.4.2.2 Persistencia y distribución transparente por identificadores globales uniformes

La dirección de memoria virtual en el espacio único es un identificador único dentro del sistema. Esto permite extender de manera transparente el espacio de direcciones a memoria secundaria con la persistencia y a todo el sistema en red con la distribución. En el caso del sistema integral el espacio estará formado por objetos. El uso de un identificador único de objetos, global en el sistema permitirá realizar mecanismos análogos de persistencia y distribución transparente.

4.5 Inferno

Inferno [DPP+96] es un producto de Lucent Technologies que proporciona un entorno uniforme de ejecución para aplicaciones orientadas a red con sistemas heterogéneos. Aunque presentado después de iniciarse esta investigación, se menciona aquí puesto que su estructura confirmó la línea de las decisiones de diseño que ya estaban efectuadas previamente.

Elementos fundamentales del sistema

El sistema se estructura en torno a tres elementos fundamentales:

- El uso del concepto de **sistema de ficheros jerárquico** para representar cualquier recurso del sistema, ya sea local o remoto.
- Un concepto de **proceso convencional** que utiliza un modelo de concurrencia basado en la **comunicación de procesos secuenciales** [Hoa78] (CSP, *Communicating Sequential Processes*) mediante canales de comunicación.
- Cada usuario o proceso construye una vista privada del sistema construyendo un **espacio de nombres** de ficheros que conecta los recursos que utiliza. Estos espacios pueden importarse desde (o exportarse a) cualquier máquina de la red.

El sistema utiliza básicamente los conceptos de Unix de proceso y fichero, extendiendo este último de manera uniforme para representar todos los recursos del sistema, independientemente de su localización. En este sentido puede verse el sistema como la evolución de los conceptos introducidos por Unix y desarrollados en el sistema Plan 9 [PPT+92] de la propia Bell Labs.

Arquitectura del sistema

La mayor novedad respecto al Plan9 es la estructuración del sistema: Un **núcleo** del SO de red que da soporte a una **máquina virtual** sobre la que funcionan las aplicaciones. Esto le proporciona la independencia de la plataforma. Existe también un **lenguaje de programación modular** desarrollado especialmente para el entorno, llamado Limbo, así como un conjunto de **protocolos de red** para la denominación y el acceso a los recursos, y la seguridad.

- **El núcleo**. Proporciona los servicios básicos que son utilizados por la máquina virtual: creación de procesos, espacio de nombres, acceso a la red, entrada/salida, etc. Además

incluye el protocolo de comunicación **Styx**, que permite la comunicación entre máquinas Inferno sobre cualquier transporte (básicamente para realizar operaciones con ficheros).

- **La máquina virtual Dis**. Soporta el modelo descrito anteriormente que utiliza el sistema. Es una máquina virtual sencilla, con instrucciones de memoria a memoria de tres operandos y algunas instrucciones más para gestión de tipos de datos de más alto nivel como arrays, cadenas de caracteres, procesos y canales de comunicación. Incluye también la gestión automática de memoria mediante un recolector de basura incorporado.

4.5.1 Crítica

Es un sistema con objetivos similares a los del sistema integral, aunque en lugar de usar el paradigma OO como abstracción básica del sistema es una evolución de las abstracciones convencionales de proceso y fichero.

4.5.2 Características interesantes

El concepto de utilizar una máquina abstracta como soporte de todo un sistema operativo es lo más importante de este sistema.

4.5.2.1 Uso de máquina abstracta para heterogeneidad y portabilidad

La idea de usar una máquina abstracta en el sistema integral para lograr la portabilidad y la heterogeneidad ya estaba adoptada antes de presentarse este sistema. Sin embargo, la aparición de este sistema confirmó la validez de la utilización de máquinas abstractas no sólo para implementar lenguajes, si no para dar soporte a entornos completos de computación, como en el caso del sistema integral.

4.6 Clouds

Clouds es un sistema operativo pensado para dar soporte a objetos distribuidos [DAM+90, DLA+91] desarrollado en el Instituto de Tecnología de Georgia. Está implementado sobre el micronúcleo Ra, que funciona sobre máquinas Sun.

Abstracciones de Clouds

Clouds utiliza la abstracción de **hilo** para la computación y la de **objeto** para representar el espacio de almacenamiento.

Un objeto Clouds es un objeto pasivo de grano grueso, que equivale a un espacio de direcciones virtual de un proceso convencional. En este espacio se almacenan los datos y el código del objeto. La estructura interna de un objeto no es conocida por el sistema. Puede ser internamente una colección de objetos programados en C++, pero estos objetos internos no pueden ser utilizados desde fuera del objeto Clouds. A pesar de que el usuario puede crear estos objetos Clouds, no existe el concepto de clases ni de herencia en este sistema.

Los objetos son globales y como tales tienen un nombre único siempre válido dentro del sistema distribuido. La utilización de este identificador único permite la invocación transparente de un objeto independientemente de su localización.

Los hilos no están asociados a ningún objeto determinado, y van ejecutándose en los espacios de direcciones de los objetos, viajando de uno a otro a medida que se van invocando métodos de los distintos objetos. Los hilos pueden tener unas etiquetas especiales que permiten mantener diferentes tipos de atomicidad y consistencia en la invocación a operaciones de los objetos.

4.6.1 Crítica

El sistema no llega a cumplir muchos de los objetivos del sistema integral, como la portabilidad y la heterogeneidad, aunque es uno de los primeros sistemas en dar soporte a objetos directamente por el sistema. Sin embargo el modelo de objetos que soporta también se queda corto para las necesidades del sistema integral:

4.6.1.1 Modelo de objetos alejado del de las aplicaciones

El modelo de objetos que soporta el sistema está muy alejado de los modelos utilizados en las aplicaciones. No soporta el concepto de clases, ni el de herencia. Los objetos son de un grano muy grueso, muy alejado del grano fino de los objetos de las aplicaciones. La estructura interna de objetos dentro de un objeto Clouds es totalmente desconocida para el sistema. El objeto Clouds es una pequeña evolución de un espacio de direcciones (objeto) sobre los que pueden funcionar los proceso (hilos que pasan por el objeto).

4.6.2 Características interesantes

Lo más interesante del sistema es precisamente la idea de dar soporte directo a objetos en un sistema, aunque el tipo de soporte que da Clouds no es suficientemente detallado.

4.6.2.1 Identificador global de objetos para transparencia de localización

Otro aspecto interesante es el uso de un identificador global de objetos, que permite la invocación transparente de un objeto independiente de su localización.

4.7 Choices

Choices [CIM+93] es una familia de sistemas operativos para sistemas multiprocesador de memoria distribuida y compartida.

Arquitectura del sistema

El diseño de Choices se captura mediante un **marco de aplicación** (*framework*) que describe los componentes del sistema en abstracto y la manera en que interactúan. El marco de aplicación es una jerarquía de clases C++.

El **núcleo** en tiempo de ejecución es un núcleo monolítico se implementa como un conjunto de objetos C++ que resultan de una instanciación concreta del marco. Estos objetos implementan los diferentes aspectos del sistema operativo, como la interfaz con las aplicaciones (espacio del usuario) y con el hardware, y los recursos, políticas y mecanismos del sistema. En cualquier caso, son los objetos propios de un programa en C++.

Abstracciones de Choices

A pesar de estar implementado de manera interna con un lenguaje orientado a objetos, las abstracciones que proporciona Choices son las de un sistema operativo más convencional: **procesos** (hilos), **dominios** (espacios de direcciones virtuales) **y objetos de memoria**. Existe una clara división entre el espacio del sistema y el de usuario. Siempre existe un dominio del sistema en el que hay procesos del sistema que funcionan en modo supervisor. En modo usuario puede haber varios dominios de usuario con procesos de aplicación.

Sin embargo, la interfaz del sistema operativo (interfaz a los servicios del núcleo) se proporciona a las aplicaciones mediante un conjunto de objetos colocados en el núcleo. Las aplicaciones acceden a los servicios del sistema invocando métodos de estos objetos. La manera de cruzar la frontera entre el usuario y el sistema es mediante un **objeto representante** (*proxy*) del objeto del sistema dentro del espacio del usuario. Combinando

estos representantes con listas de control de accesos y servidores de nombres se establecen los mecanismos de seguridad del sistema.

Persistencia y distribución

El acceso a objetos remotos es posible mediante una extensión del mecanismo de representantes que utiliza internamente las primitivas de paso de mensajes del sistema para acceder a la máquina remota.

El usuario puede usar objetos persistentes aprovechando los marcos del sistema. Una especialización del sub-marco de sistema de ficheros lo permite. Incluso se pueden usar referencias entre objetos persistentes instanciando una clase que proporciona esas referencias persistentes.

El problema, como se ve, es que al ser los marcos propios del lenguaje C++, toda la programación en el sistema, y los conceptos que maneja el sistema son los de C++. La utilización de otros lenguajes está más restringida.

Jerarquía de marcos

La importancia de Choices está en haber sido pionero en la utilización de la orientación a objetos mediante marcos de aplicación a la construcción de una familia de sistemas operativos.

Figura 4.2 Marco de aplicación superior del sistema Choices.

La estructura es la de una jerarquía de marcos. El marco superior es un conjunto de clases abstractas que describe los componentes básicos del sistema y restricciones que deben cumplir los sub-marcos.

Los sub-marcos representan subsistemas del sistema operativo que introducen especializaciones (aún abstractas) de las clases abstractas del marco anterior y nuevas restricciones. El conjunto de las restricciones define implícitamente las características comunes de la arquitectura de la familia de sistemas Choices.

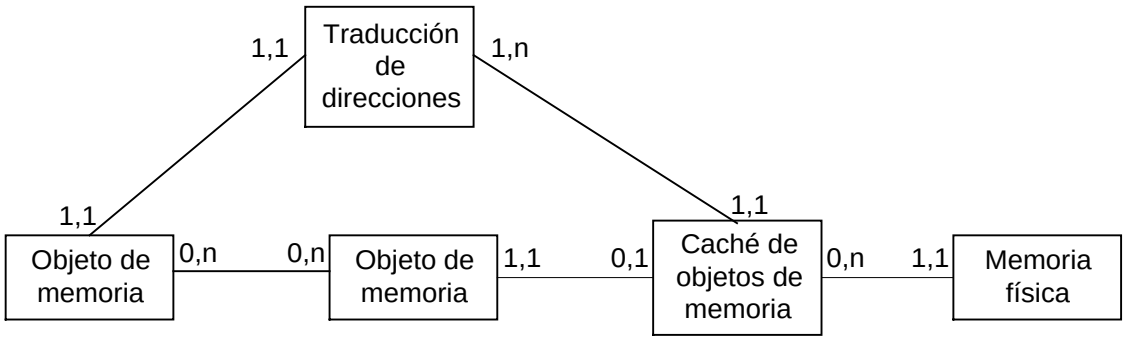


Figura 4.3 Sub-marco de aplicación para la gestión de memoria de Choices.

Un sistema operativo concreto, es decir una instanciación concreta del marco de aplicación Choices, se construye sustituyendo las clases abstractas del marco por clases concretas que definen una implementación determinada. Por ejemplo el uso de un sistema de ficheros Unix, en lugar de uno MS-DOS; la utilización de una determinada política de planificación, etc. Esto permite la generación de sistemas operativos a medida para una determinada aplicación, aunque manteniendo todos unas características comunes.

4.7.1 Crítica

A pesar de utilizar la OO, se utiliza de una manera estática y restringida, y para dar soporte a abstracciones de sistemas operativos convencionales.

4.7.1.1 Separación usuario / sistema: sólo flexibilidad estática

Se separan claramente los elementos del usuario y los elementos fijos del sistema. Esta separación impide aplicar la flexibilidad dinámica a los elementos del sistema. Sólo se consigue la flexibilidad estática al estar desarrollada la implementación como un marco de aplicación OO.

4.7.1.2 Uso restringido al C++

La utilización del sistema y del soporte de objetos está restringido únicamente al lenguaje de construcción del mismo, que es C++. El modelo de objetos del C++ no es muy adecuado para el sistema integral, puesto que en tiempo de ejecución un objeto C++ es simplemente una zona de memoria. Se pierde casi toda la semántica del modelo de objetos.

4.7.1.3 Falta de uniformidad en la Orientación a Objetos

A pesar de estar implementado mediante objetos, las abstracciones que utiliza el sistema son las abstracciones tradicionales de proceso, espacio de direcciones, fichero, etc.

4.7.2 Características interesantes

Como precursor del sistema Tigger revisado anteriormente, utiliza una jerarquía de clases como base de la implementación del sistema operativo, y además proporciona una interfaz de usuario orientada a objetos.

4.7.2.1 Jerarquía de clases de implementación y con interfaz OO para el usuario

Además de utilizar una jerarquía de clases en la implementación, que da flexibilidad estática al sistema, también utiliza una interfaz OO para acceder a los servicios del sistema. El usuario accede a objetos que tienen funcionalidad del sistema. Estos se organizan también mediante una jerarquía de clases que el usuario puede utilizar. En este aspecto si hay una uniformidad de uso OO. En el sistema integral es interesante que la funcionalidad del sistema operativo se proporcione mediante objetos que el usuario pueda usar como cualquier otro objeto, por tanto también estarán organizados en una jerarquía normal de clases.

4.8 SPIN

El sistema operativo SPIN [BSP+95], aunque no directamente relacionado con la orientación a objetos, tiene como objetivo desarrollar un sistema en el que el núcleo del sistema operativo pueda extenderse mediante código de usuario con seguridad. Es decir, permitir que el usuario pueda añadir funcionalidad al sistema operativo. Al existir una clara separación entre el espacio del usuario y el del sistema, esto además permite un aumento del rendimiento al no tener que realizarse cambios de contexto entre los espacios.

Sin embargo la adición de código de usuario al sistema dentro del espacio del núcleo presenta un problema de seguridad. Este código de usuario al estar dentro del núcleo podría, bien por errores de programación o maliciosamente, corromper estructuras y/o código del sistema operativo.

Extensibilidad

La solución del sistema SPIN es proporcionar una infraestructura para ejecutar el código del usuario dentro del núcleo (extensiones denominadas ***spindles***). Se restringe estos *spindles*

a que estén programados en un lenguaje especial seguro, un subconjunto del Modula-3. La utilización de este lenguaje seguro, la existencia en el sistema de un compilador verificado del mismo y un metalenguaje específico para el control en tiempo de enlace permite asegurar que los *spindles* no comprometan la seguridad del núcleo.

La característica más importante de este lenguaje es que dispone de punteros seguros, imposibles de modificar o alterar por el usuario. Esto hace que un programa escrito con esta versión segura del Modula-3 nunca pueda acceder fuera de su propio ámbito, en este caso al resto del núcleo.

4.8.1 Crítica

SPIN no es un sistema diseñado con el objetivo de la orientación a objetos, por tanto no da soporte directo a ningún elemento relacionado con la OO, sus abstracciones son convencionales. Existe una separación clara entre el espacio del usuario y el del sistema, aunque la característica principal de SPIN es permitir la inclusión de código de usuario en el núcleo del sistema para extenderlo.

4.8.1.1 Falta de uniformidad para la extensión

El problema de estas extensiones es que la manera de hacerlas es específica, no uniforme con la forma normal de utilización del sistema por el usuario. Las extensiones se programan con un lenguaje especial y se utiliza un metalenguaje también especial para enlazarlas con el núcleo. Esto es totalmente diferente de cómo se programan las aplicaciones normales de usuario.

4.8.2 Características interesantes

Como uno de los sistemas pioneros que permite extender dinámicamente el sistema, ésta es su característica más importante, junto con la necesidad de controlar esta extensión.

4.8.2.1 Extensibilidad dinámica por código de usuario

Es importante la posibilidad de que el usuario pueda extender dinámicamente la funcionalidad del sistema programando el mismo las extensiones. Esto da mucha flexibilidad al sistema. Para mantener la uniformidad, en el sistema integral esta extensión serán objetos del usuario, y se deberá realizar de manera uniforme y no diferente a la manera de programar normal en el sistema.

4.8.2.2 Seguridad en la extensibilidad

La extensibilidad es un mecanismo muy potente, pero que puede poner en riesgo el buen funcionamiento del sistema. Es importante controlar que el código de usuario no pueda salir de su ámbito y acceder o modificar servicios que no tenga por qué utilizar. En el caso del sistema integral se deberá controlar a los objetos que extiendan el sistema, siempre de manera uniforme con el resto del sistema y no mediante medios específicos.

4.9 Apertos

Apertos [Yok92], una evolución de su predecesor Muse [YTY+89] suele considerarse como el pionero en la aplicación de la reflectividad en los sistemas operativos orientados a objetos.

Es un sistema operativo para dar soporte a objetos y estructurado uniformemente en términos de objetos. La motivación inicial de aplicación es para un entorno de computación móvil, compuestos por ordenadores que pueden estar conectados a la red y también

desconectarse para trabajar independientemente o trabajar remotamente mediante enlaces sin hilos, etc. La gran novedad de este sistema consiste en utilizar una separación de los objetos en dos niveles: meta-objetos y objetos base.

Estructuración mediante objetos y meta-objetos

Los **meta-objetos** proporcionan el entorno de ejecución y dan soporte (definen) a los **objetos base**. Cada objeto base está soportado por un **meta-espacio**, que está formado por un grupo de meta-objetos. Cada meta-objeto proporciona una determinada funcionalidad a los objetos del meta-espacio en que se encuentra. Por ejemplo, un determinado objeto puede estar en un meta-espacio con meta-objetos que le proporcionen una determinada política de planificación, un mecanismo de sincronización determinado, el uso de un protocolo de comunicación, etc.

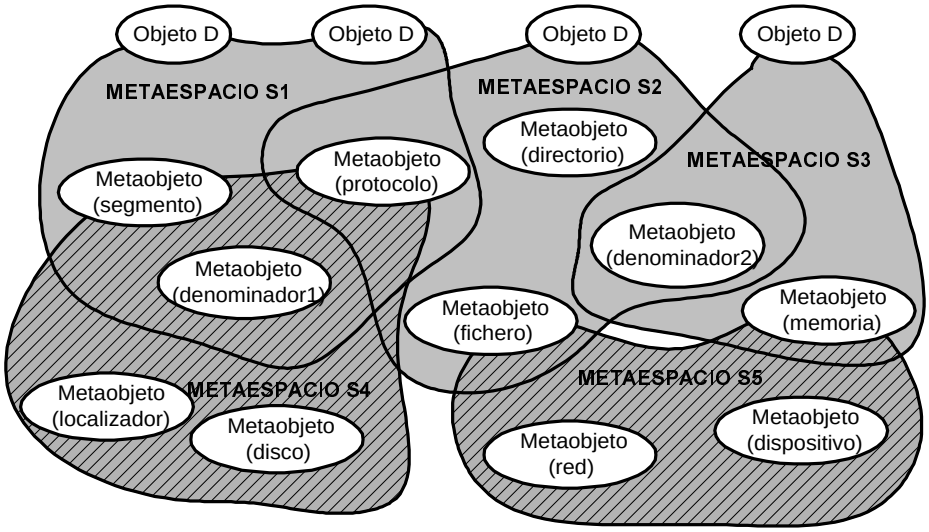


Figura 4.4 Separación entre los espacios de objetos y de meta-objetos en el sistema Apertos.

Flexibilidad

La flexibilidad del sistema se consigue mediante la posibilidad de que un objeto cambie de meta-espacio, o la adición de nuevos meta-objetos que proporcionen funcionalidad adicional. Por ejemplo, un meta-objeto podría migrar de un meta-espacio dado a otro meta-espacio que use un protocolo de comunicaciones para una red sin hilos cuando el ordenador en el que se encuentre se desconecte físicamente de la red.

Reflectividad

La reflectividad se produce al existir una interfaz bidireccional entre los objetos base y los meta-objetos que los soportan. Los objetos pueden dialogar con sus meta-objetos, usando un punto de entrada al meta-espacio denominado **reflector**. A su vez, los meta-objetos influyen en el funcionamiento de los objetos base. Por otro lado ambos, objetos y meta-objetos, comparten el mismo marco conceptual al estar descritos en los mismos términos de objetos. Los meta-objetos también pueden considerarse como objetos, por lo que tendrían su propio meta-meta-espacio y así sucesivamente. Esta regresión termina en un meta-objeto primitivo que no tiene meta-meta-espacio (se describe a sí mismo).

Los objetos y meta-objetos se organizan en una jerarquía de clases, al igual que los reflectores. Apertos se estructura usando una serie de jerarquías predefinidas de reflectores y meta-objetos.

Jerarquía de reflectores

La jerarquía de reflectores define la estructura común de programación de meta-objetos del sistema. Un reflector, como punto de entrada a un meta-espacio, ofrece a los objetos base una serie de operaciones que pueden usar de sus meta-objetos. La jerarquía base define las operaciones comunes del sistema. El reflector mCommon contiene una operación común a todos los reflectores, canSpeak que permite comprobar la compatibilidad entre meta-espacios a la hora de la migración de los objetos. Otros reflectores son mRealtime, que ofrece a los objetos una planificación en tiempo real, mDriveObject, que proporciona meta-operaciones para programación de controladores de dispositivos, etc.

MetaCore es un meta-objeto terminal en el que finaliza la regresión de meta-meta-objetos. Puede considerarse como el equivalente de un micronúcleo de otros sistemas. Proporciona a todos los demás objetos del sistema con los elementos básicos del sistema, es decir, las primitivas comunes de separación entre objetos y meta-objetos y de migración de objetos. Proporciona el concepto de contexto de ejecución para virtualizar la CPU y las primitivas básicas de la computación reflectiva: M hace una petición de meta-computación (llamada al metaespacio). R reanuda la ejecución de un objeto (retorno de la llamada al metaespacio).

4.9.1 Crítica

Este sistema sí que utiliza como única abstracción los objetos y da soporte directo a los mismos. Existen, sin embargo, algunos aspectos que no están totalmente de acuerdo con los objetivos del sistema integral. Por ejemplo no hay previsión directa para la portabilidad.

4.9.1.1 Complejidad de estructura

La separación de la estructura del sistema en múltiples meta-espacios recursivos, aunque da mucha flexibilidad al sistema, también complica la comprensión del mismo de manera sencilla por el usuario.

4.9.1.2 Falta de uniformidad por la separación espacio/meta-espacio de objetos

La separación completa de ambos espacios introduce una cierta falta de uniformidad en el sistema. La jerarquía de meta-objetos está totalmente separada de la de los objetos de usuario. La programación de meta-objetos se realiza por tanto de una manera diferente a la de los objetos normales.

4.9.1.3 No existe mecanismo de seguridad uniforme en el sistema

Muchos elementos del sistema no se definen, y se deja que se implementen mediante meta-objetos. Sin embargo, en casos como la seguridad, es conveniente para la uniformidad y la fácil comprensión del sistema que exista un mecanismo básico y uniforme de protección, que forme parte integrante del núcleo del sistema.

4.9.2 Características interesantes

La característica principal es la utilización de la reflectividad como elemento básico del sistema operativo.

4.9.2.1 Reflectividad para la flexibilidad

La adopción de la reflectividad en el sistema es muy importante para lograr la flexibilidad en el sistema de manera uniforme. Esta propiedad permite describir mediante objetos los propios elementos de la máquina. De esta manera se unifican dentro del paradigma de la OO los elementos del usuario y los elementos del sistema que los soportan. Esto permite la

extensibilidad del sistema de una manera uniforme, al permitir que los objetos de usuario puedan acceder a los objetos del sistema usando el mismo paradigma de la OO.

4.10 Resumen de características de los sistemas operativos revisados

Nombre / Año	COOLv2 (1993)	SPACE (1991)	Tigger (1995)	Sombrero (1996)	Inferno (1996)
Área aplicación	Aplicaciones distribuidas		Aplicaciones avanzadas		Aplicaciones de red
Característica principal	Soporte varios modelos objetos	Núcleo mínimo	Marco aplicación familia SO	Espacio único de direcciones	Uso máquina virtual
Heterogeneidad	No	No	No	No	Sí
Soporte objetos	Sí	Básico	Sí	Básico	No
Granularidad	Fina	-	Fina	-	-
Semántica modelo objetos	Completa	Zona memoria	Completa	Zona memoria	-
Multilenguaje	Sí		Sí		
Estructuración	Micronúcleo / capa soporte básico objetos / capas modelos específicos	Exonúcleo / funcionalidad en espacio usuario	Núcleo capas soporte básico objetos / capas soporte modelos específicos	Núcleo en el mismo espacio que el usuario	Núcleo / máquina virtual / aplicaciones usuario
Separación usuario/sistema		No (todo en espacio usuario)		No	
Interfaz OO SO	No	No		No	No
Uniformidad OO		-			
Homogeneidad objetos	Sí dentro del mismo modelo	-	Sí dentro del mismo modelo	-	-
Identificador objetos	Dual, según invocación local / remota	-	Dual, según invocación local / remota	-	-
Interoperabilidad	No completa entre distintos modelos	-	No completa entre distintos modelos	-	-
Transparencia distribución	No completa, acceso diferente objetos locales /remotos	-	No completa, acceso diferente objetos locales /remotos	-	Sí

Tabla 4.1 Resumen de características de sistemas operativos.

Nombre / Año	Clouds (1989)	Choices (1988)	SPIN (1995)	Apertos (1992)
Área aplicación	Sistema distribuido	Sistemas multiprocesador		Sistemas móviles
Característica principal	Soporte objetos	Marco aplicación familia SO	Extensible por lenguaje seguro	Reflectividad
Heterogeneidad	No	No	No	No
Soporte objetos	Sí	Sí	No	Sí
Granularidad	Gruesa	Fina	-	Fina
Semántica modelo objetos	Basado en objetos	Completa C++	-	Completa
Multilenguaje	Sí	No		
Estructuración	Micronúcleo / objetos servidores	Micronúcleo objetos / aplicaciones usuario	Núcleo extensible / aplicaciones usuario	Meta-objetos / objetos
Separación usuario/sistema	Sí	Sí	Sí	No
Interfaz OO SO	Sí	Sí	No	Sí
Uniformidad OO	(dentro de los objetos Clouds)		-	(posible según meta-objetos)
Homogeneidad objetos	Sí	Sí	-	Posible
Identificador objetos	Único	No	-	Posible
Interoperabilidad	Sí	Sí	-	Posible
Transparencia distribución	Sí	Sí	-	Posible

Tabla 4.2 Resumen de características de sistemas operativos (segunda parte).

Capítulo 5

ARQUITECTURA DEL SISTEMA INTEGRAL ORIENTADO A OBJETOS

A continuación se describe una arquitectura que permite obtener un sistema integral orientado a objetos con los objetivos especificados en el capítulo 3.

Esta arquitectura [ATA+97] integra diferentes tendencias en el campo de los sistemas operativos avanzados, especialmente con características de OO, mostradas en el capítulo 4. En general, estas tendencias se han aplicado para solucionar problemas concretos en un sistema operativo determinado. Sin embargo, la arquitectura de este sistema permite aplicar estas tendencias de manera natural, conjunta y uniforme para obtener un sistema integral con las propiedades deseadas. La característica más importante es la integración fluida de las mismas dentro del marco general de la OO sin romper el objetivo fundamental del sistema.

5.1 Maquina abstracta OO + Sistema Operativo OO = Sistema Integral OO

La estructura consiste en una máquina abstracta orientada a objetos que proporciona soporte para un modelo único de objetos de todo el sistema. Una serie de objetos que forman el sistema operativo extienden la funcionalidad de la máquina proporcionando a los objetos diferentes facilidades de manera transparente.

5.2 Propiedades fundamentales de la arquitectura

Las propiedades más importantes que incorpora la arquitectura anterior en cada uno de sus dos elementos, la máquina abstracta y el sistema operativo, se reseñan a continuación.

5.2.1 Máquina abstracta orientada a objetos

Dotada de una arquitectura reflectiva, proporciona un modelo único de objetos para el sistema.

5.2.1.1 Modelo único de objetos

La máquina proporciona al resto del sistema el soporte para objetos necesario. Los objetos se estructuran usando el modelo de objetos de la máquina, que será el único modelo de objetos que se utilice dentro del sistema.

5.2.1.2 Reflectividad

La máquina dispondrá de una arquitectura reflectiva [Mae87], que permita que los propios objetos constituyentes de la máquina puedan usarse dentro del sistema como cualquier otro objeto dentro del mismo.

5.2.2 Sistema Operativo Orientado a Objetos

Estará formado por un conjunto de objetos que proporcionen funcionalidad que en otros entornos se considera parte del sistema operativo.

5.2.2.1 Transparencia: persistencia, distribución, concurrencia y seguridad

Estos objetos serán objetos normales del sistema, aunque proporcionarán de manera transparente al resto de los objetos las propiedades de **persistencia**, **distribución**, **concurrencia** y **seguridad**.

5.2.3 Orientación a objetos

Se utiliza en el sistema operativo al organizarse sus objetos en una jerarquía de clases, lo que permite la reusabilidad, extensibilidad, etc. del sistema operativo.

La propia estructura interna de la máquina abstracta también se describirá mediante la OO.

5.2.4 Espacio único de objetos sin separación usuario/sistema

La combinación de la máquina abstracta con el sistema operativo produce un único espacio de objetos en el que residen los objetos. No existe una división entre los objetos del sistema y los del usuario. Todos están al mismo nivel, independientemente de que se puedan considerar objetos de aplicaciones normales de usuario u objetos que proporcionen funcionalidad del sistema.

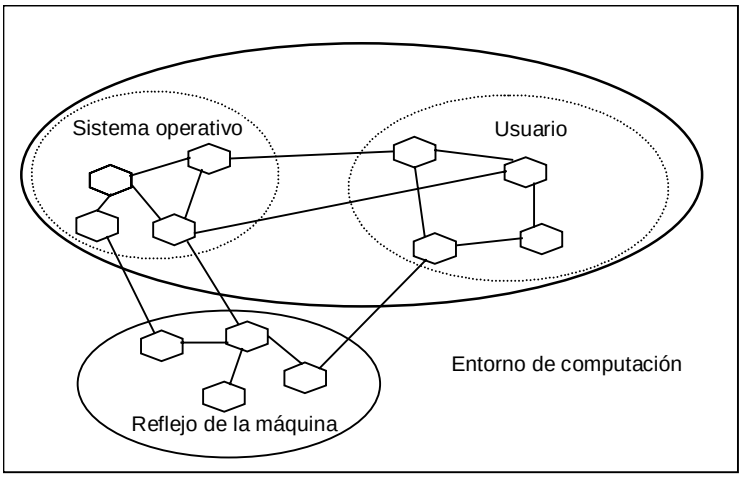


Figura 5.1 Espacio único de objetos homogéneos.

5.2.5 Identificador de objetos único, global e independiente

Los objetos dispondrán de un identificador único dentro del sistema. Este identificador será válido dentro de todo el sistema y será el único medio por el que se pueda acceder a un objeto. Además, el identificador será independiente de la localización del objeto, para hacer transparente la localización de un objeto al resto del sistema.

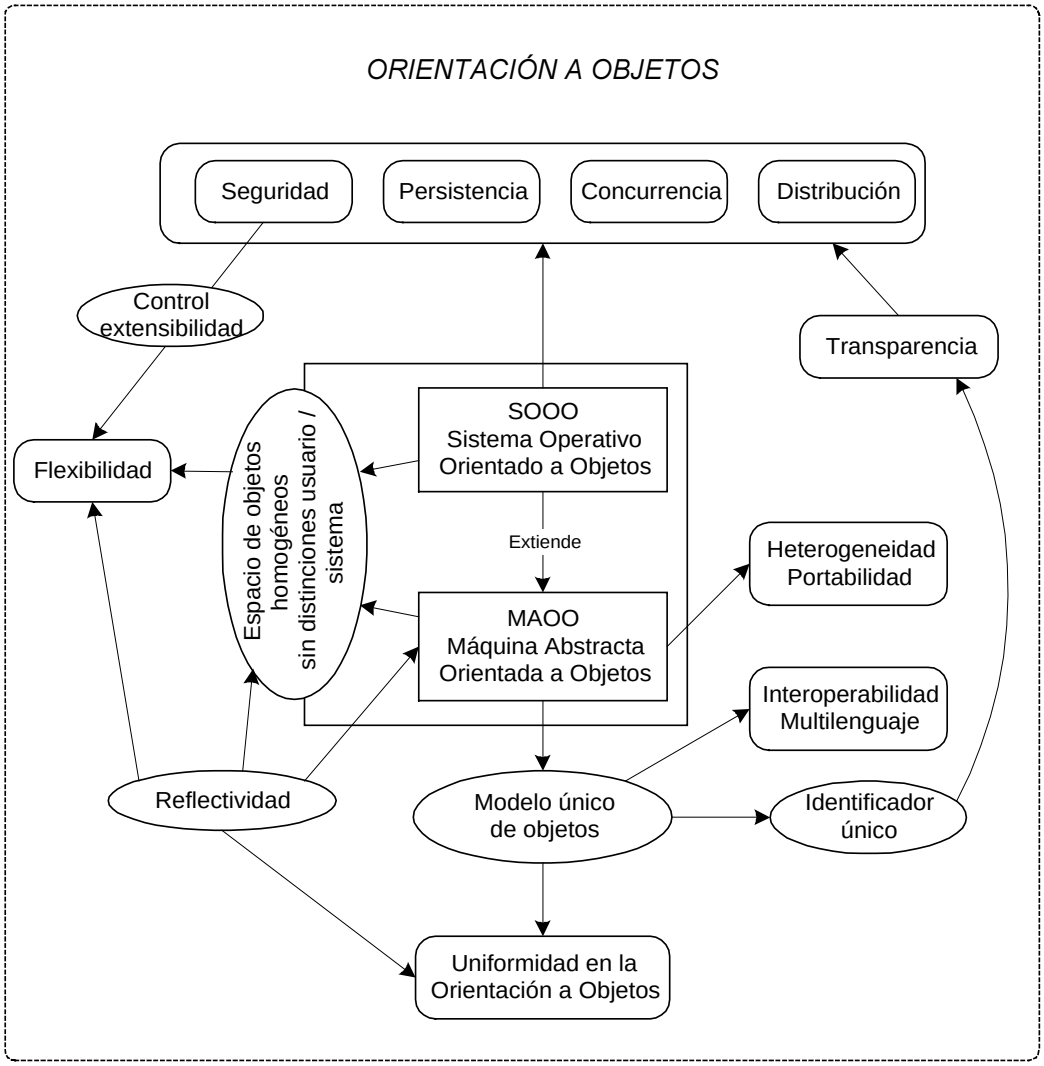


Figura 5.2 Propiedades incorporadas en la arquitectura del sistema integral y relaciones con los objetivos del mismo.

5.3 Contribución de las propiedades a los objetivos del sistema integral

Estas diferentes propiedades contribuyen a lograr los objetivos marcados para el sistema integral.

5.3.1 Uniformidad conceptual en torno a la orientación a objetos

El modelo único de objetos y la reflectividad contribuyen a lograr una uniformidad conceptual en torno al paradigma de la orientación a objetos.

5.3.1.1 Modelo único de objetos

Se consigue con el soporte de objetos y el modelo único de objetos que proporciona la máquina. Al estar el sistema basado en la máquina abstracta, los elementos que proporcione ésta serán los únicos que se puedan utilizar en el sistema. Dado que el único elemento que ofrece es la orientación a objetos, este se tiene que usar uniformemente en el resto del sistema.

5.3.1.2 Reflectividad

La reflectividad permite que esta uniformidad se extienda hasta la propia máquina. Al permitir la descripción de la máquina en los mismos términos que los objetos que soporta la misma, todo el sistema en su conjunto, incluyendo la propia máquina, comparte uniformemente el mismo paradigma.

5.3.2 Interoperabilidad/Multilenguaje

La interoperabilidad entre diferentes objetos escritos en diferentes lenguajes se logra utilizando un modelo único de objetos.

5.3.2.1 Modelo único de objetos

El soporte de un modelo único de objetos permite la utilización de múltiples lenguajes para crear los objetos. Los objetos de cada lenguaje se crearán en el modelo común de la máquina abstracta, en lugar de en un modelo propio bajo control de cada compilador. De esta manera, una vez creado un objeto en el sistema, el lenguaje de creación del mismo es irrelevante, ya que el objeto pasa a formar parte del sistema gestionado por la máquina. Una vez que los objetos están todos en un modelo común, pueden interoperar libremente sin desadaptación.

5.3.3 Heterogeneidad/Portabilidad

La máquina abstracta permite obtener portabilidad en plataformas heterogéneas.

5.3.3.1 Máquina abstracta

Una máquina abstracta presenta un único sistema para el que se desarrollan las aplicaciones.

Heterogeneidad de plataformas

La utilización de una máquina abstracta permite que el sistema use cualquier tipo de plataforma. Todos los programas se compilarán para ser usados en la máquina abstracta, generando instrucciones propias de la misma. Puesto que la máquina abstracta no será dependiente de ninguna plataforma real, basta con desarrollar un emulador de la máquina abstracta en cada plataforma para que el sistema pueda funcionar en la misma. El funcionamiento del sistema queda asegurado aún en un entorno heterogéneo.

Portabilidad del sistema

La portabilidad a estas plataformas es muy sencilla, ya que basta con crear el emulador de la máquina para cada plataforma. El resto del sistema funcionará ya inmediatamente sin modificaciones ya que estará escrito para la máquina abstracta. El esfuerzo para trasladar el sistema a una nueva plataforma se ve muy reducido. Además, aunque no afecte al funcionamiento del sistema, el propio emulador se puede escribir con un lenguaje orientado a objetos, lo que reduce aún más el esfuerzo.

Movilidad de aplicaciones (objetos)

Por otro lado, esta independencia de la plataforma permite que los objetos puedan moverse libremente entre los diferentes ordenadores sin necesidad de ningún tipo de modificación. A todos los efectos, la plataforma de los objetos es la propia máquina abstracta. Aunque los objetos

se muevan de un ordenador a otro, al existir en todos la misma máquina abstracta, los objetos no necesitan cambios.

5.3.4 Flexibilidad

Un espacio único de objetos combinado con la reflectividad y la propia orientación a objetos dotan de flexibilidad al sistema.

5.3.4.1 Espacio único de objetos

Al existir un único espacio de objetos no hay una distinción entre elementos del sistema y del usuario. No existen barreras que cruzar para comunicar un objeto del usuario con uno del sistema. El mismo paradigma se utiliza tanto para el desarrollo de objetos de usuario como para el desarrollo de objetos que den funcionalidad al sistema. No se necesitan herramientas ni procedimientos especiales para añadir nueva funcionalidad al sistema o adaptar la ya existente [SS96]. La flexibilidad del sistema aumenta al usar simplemente los mismos mecanismos que para desarrollar aplicaciones de usuario. Además de la homogeneidad, todos los objetos y no sólo algunos de ellos [YMF91] se benefician de los servicios proporcionados por el sistema.

5.3.4.2 Reflectividad

La reflectividad aumenta la flexibilidad del sistema al situar los propios objetos de la máquina abstracta al mismo nivel que los demás objetos del sistema. Las ventajas del apartado anterior se hacen extensivas a los componentes de la propia máquina. Esto permitirá extender y adaptar la funcionalidad de la máquina de manera muy sencilla.

5.3.4.3 Orientación a objetos

El uso extensivo de la orientación a objetos permite aprovechar todas las ventajas de la misma, entre ellas la flexibilidad. La organización de todos los objetos del sistema en una jerarquía de clases permite aprovechar la extensibilidad, reusabilidad y mantenibilidad [Boo94] no solo en las aplicaciones de usuario, sino también en los elementos del sistema operativo. Aún más, los propios objetos y clases del SO, al estar en el mismo nivel que los objetos de usuario, pueden ser utilizados directamente por las aplicaciones de usuario.

5.3.5 Control de la flexibilidad

La flexibilidad se controla mediante el mecanismo de protección del sistema.

5.3.5.1 Seguridad

La gran flexibilidad del sistema permite que sea muy fácil modificar el mismo. Como ya se mostró, es necesario tener un control que permita conservar la consistencia del sistema, para evitar problemas producidos por aplicaciones maliciosas. La existencia de un mecanismo de protección de objetos genérico y uniforme permite la aplicación del mismo para el caso particular de que se modifiquen objetos que dan funcionalidad al sistema. Se mantiene una uniformidad al estar el propio sistema construido mediante objetos del mismo nivel que los del usuario. El mecanismo general de protección de objetos se utiliza también para los objetos del sistema. No se necesita un mecanismo especial para controlar las extensiones que se hagan al sistema.

5.3.6 Transparencia

La transparencia de uso de ciertas propiedades se proporciona mediante el sistema operativo orientado a objetos y es facilitada por la utilización de un identificador de objetos único.

5.3.6.1 Sistema operativo orientado a objetos

Los objetos del sistema operativo proporcionan al resto de los objetos la funcionalidad de persistencia, distribución, seguridad y concurrencia de manera transparente. Al extender las capacidades básicas de la máquina abstracta, estas capacidades ampliadas se hacen extensivas de manera automática y transparente para el resto de los objetos.

5.3.6.2 Identificador único

El uso de un identificador único permite que la localización de un objeto pueda ser totalmente transparente para el resto de los objetos del sistema. Al accederse a un objeto únicamente mediante su identificador, un objeto simplemente invocará un método de otro objeto, siendo transparente para él objeto que invoca la localización del otro objeto. El sistema se encargará de realizar la invocación y localizar la situación concreta del objeto en cuestión (mediante los objetos del sistema operativo que proporcionan la propiedad de distribución, persistencia, etc.)

5.4 Resumen

Para conseguir un sistema integral orientado a objetos se utiliza una arquitectura compuesta por dos elementos principales: una máquina abstracta orientada a objetos y un sistema operativo que extiende la misma. En cada uno de estos elementos se disponen ciertas características del sistema. Las propiedades de estos elementos y características aprovechadas individualmente en otros sistemas, se integran de manera fluida dentro del marco común de la orientación a objetos, contribuyendo en su conjunto a lograr los objetivos del sistema integral.

En los siguientes capítulos se describirán las decisiones de diseño que motivan la elección de los elementos anteriores, así como sus características más detalladas, escogidos para lograr los objetivos anteriores, entre ellas la elección de un único modelo de objetos en lugar de dar soporte a varios, las características concretas del mismo, la arquitectura e implementación de una máquina abstracta que soporte el modelo, la elección del tipo de reflectividad para lograr la flexibilidad y extensibilidad en la máquina, la descripción del sistema operativo, etc.

Capítulo 6

EL SISTEMA INTEGRAL ORIENTADO A OBJETOS OVIEDO3

Oviedo3[1] [CIA+96] es un proyecto investigación que está siendo desarrollado por un equipo de profesores y alumnos del grupo de investigación en Tecnologías Orientadas a Objetos de la Universidad de Oviedo, que pretende construir un sistema integral orientado a objetos[2] con la arquitectura descrita en el capítulo 5.

Los elementos básicos que proporcionarán el espacio de objetos del sistema integral son la máquina abstracta Carbayonia[3] y el sistema operativo SO4[4] que extiende las capacidades de la misma. La definición de la máquina abstracta Carbayonia y su lenguaje Carbayón[5] se describe en el capítulo 12, y en el capítulo 13 se describe la implementación de un prototipo de la misma. Una descripción preliminar de las características del sistema operativo SO4 está en el capítulo 15. En el capítulo 16 se encuentra una discusión más completa del sistema de persistencia de SO4, junto con la implementación de un prototipo de la misma sobre la máquina Carbayonia (en el capítulo 18).

La combinación de estos dos elementos forma el sistema integral orientado a objetos (SIOO) Oviedo3. Esta plataforma permite dar un soporte directo a las tecnologías orientadas a objetos, y desarrollar más rápidamente todos los demás elementos de un sistema de computación. Todos estos elementos, desde la máquina hasta la interfaz de usuario utilizarán el mismo paradigma de la orientación a objetos.

[1] Oviedo3: Oviedo Orientado a Objetos.
[2] La página *web* del proyecto se encuentra en la dirección URL: http://www.uniovi.es/~oviedo3/
[3] Que sería algo así como “Tierra de carbayos”, puesto que su lenguaje se llama Carbayón.
[4] SO4: Sistema Operativo para Oviedo3.
[5] Carbayo grande. Carbayo es el nombre asturiano del roble. El Carbayón es el árbol representativo de la ciudad de Oviedo, hasta el extremo de que a los ovetenses también se les llama carbayones.

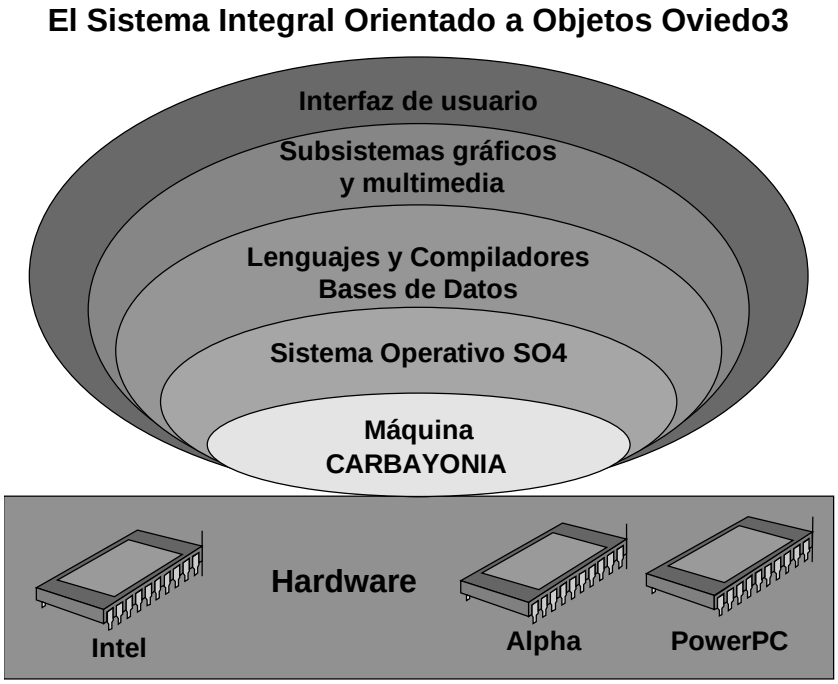


Figura 6.1 Esquema general del sistema integral orientado a objetos Oviedo3

6.1 Investigación en tecnologías orientadas a objetos

Los fines del proyecto Oviedo3 son tanto didácticos como de investigación. En lo que a investigación se refiere, se pretende usar el sistema como plataforma de investigación para las diferentes áreas de las tecnologías de objetos en un sistema de computación: la propia máquina abstracta y sistema operativo orientado a objetos, así como lenguajes y compiladores OO, bases de datos OO [MCG96], sistemas gráficos y multimedia, interfaces de usuario, etc.

Un elemento muy importante dentro de estas áreas será la definición de un modelo de componentes software en el sistema, que se utilizarán para la construcción del resto de los elementos. De especial interés en el caso de sistemas distribuidos en redes como Internet es el campo de los agentes móviles.

6.1.1 Campos de investigación que se están desarrollando sobre la plataforma Oviedo3

Algunos investigadores de la Universidad de Oviedo ya están desarrollando su investigación sobre tecnologías de objetos usando como plataforma el sistema integral orientado a objetos Oviedo3. La lista de los campos de investigación actuales (además de la propia máquina abstracta y el sistema operativo básicos) es la siguiente:

- Procesadores de lenguaje orientados a objetos
- Lenguajes de programación avanzados
- Recolección de basura
- Bases de datos orientadas a objetos
- Modelos de componentes software
- Interfaces de usuario
- Agentes móviles

- Sistemas de información geográfica
- Sistemas de métricas para estimación de proyectos software orientados a objetos

Por otro lado, el propio desarrollo de todos estos elementos es un excelente banco de pruebas para verificar la validez del sistema integral como soporte de las tecnologías orientadas a objetos (y de la robustez para la construcción de sistemas complejos). El desarrollo de los elementos anteriores debe ser mucho más sencillo con la arquitectura propuesta del sistema integral al ofrecerse un soporte directo a la orientación a objetos. Además de comprobarse de manera experimental hasta que punto se alcanza este objetivo, las experiencias acumuladas con la arquitectura se utilizarán para refinarla y mejorar aquellas partes en las que se hayan descubierto problemas.

6.2 Docencia: formación temprana en tecnologías venideras

Las tecnologías orientadas a objetos han formado parte del currícula de los planes de estudios de las ingenierías en Informática de la Universidad de Oviedo, incluso antes de su adopción total por parte de la industria. La investigación en tecnologías orientadas a objetos ha permitido a los docentes tener la preparación suficiente en estas tecnologías como para transmitirlas a la docencia[1] con el tiempo suficiente para no quedarse atrasados.

El grupo de investigación en Tecnologías Orientadas a Objetos se ha decidido a acometer el desarrollo del Sistema Integral Orientado a Objetos Oviedo3, como plataforma para la investigación fundamentalmente y también para la docencia, en lugar de utilizar sistemas comerciales existentes por las siguientes razones:

- La mayor parte de los sistemas comerciales no son orientados a objetos. Aquellos que sí lo son no tienen disponible para fines didácticos o de investigación los códigos fuente.
- Los sistemas comerciales tienen muchas restricciones debido a compatibilidad con el software más antiguo y a sus condicionantes de eficiencia sobre las arquitecturas hardware del mercado.
- En los sistemas comerciales priman como es lógico los intereses comerciales sobre los académicos, científicos y didácticos.
- Los sistemas comerciales no suelen utilizar tecnologías avanzadas hasta que éstas son ampliamente aceptadas.

El sistema integral orientado a objetos que se describe en este trabajo, con todos los elementos y soluciones que rodean, es un ejemplo del tipo de sistemas que está dejando entrever la evolución de las actuales tendencias de la informática [BPF+97]. Tarde o temprano estos conceptos se abrirán camino y serán incorporados en mayor o menor medida en sistemas comerciales. Por eso es muy importante que los profesores y alumnos[2] puedan comenzar a conocer y tener experiencia con este tipo de sistemas.

Disponiendo de este sistema integral, los profesores se formarán en la nueva tecnología antes de su expansión y estarán preparados cuando llegue el momento de traspasarla a la docencia. Los alumnos, por su parte serán conscientes de la tendencia en la evolución de los sistemas y habrán tenido un contacto previo con la misma, con lo que estarán mejor preparados cuando alcance los sistemas comerciales.

---

[1] Incluyendo un libro de texto [CGL+94] que se utiliza para la docencia en la propia Universidad de Oviedo y en otras universidades españolas.

[2] Mediante conferencias y la realización de proyectos fin de carrera

Capítulo 7

MODELO ÚNICO DE OBJETOS DEL SISTEMA INTEGRAL

En este capítulo se justifica la utilización de un modelo único de objetos en el sistema en lugar de dar soporte a varios modelos de objetos. Posteriormente se examinan varias opciones posibles para su adopción como modelo, concluyendo con la descripción del modelo único elegido, basado en el descrito por Booch en [BOO94].

7.1 Modelo único versus varios modelos

Algunos autores [GKS94] proponen estructurar un sistema operativo de una manera multinivel. La organización del sistema se compone de una serie de niveles o infraestructuras, cada una de las cuales ofrece soporte recursivamente a una serie de instancias u objetos de niveles superiores. No se trata de una arquitectura en capas, si no de que cada nivel proporcione una infraestructura especializada que sea óptima para las instancias que soporte. En cierta manera, el sistema Apertos [Yok92] es un ejemplo de este tipo de estructuración, al descomponerse en una regresión de meta-espacios, cada uno de los cuales da soporte a un conjunto de objetos base.

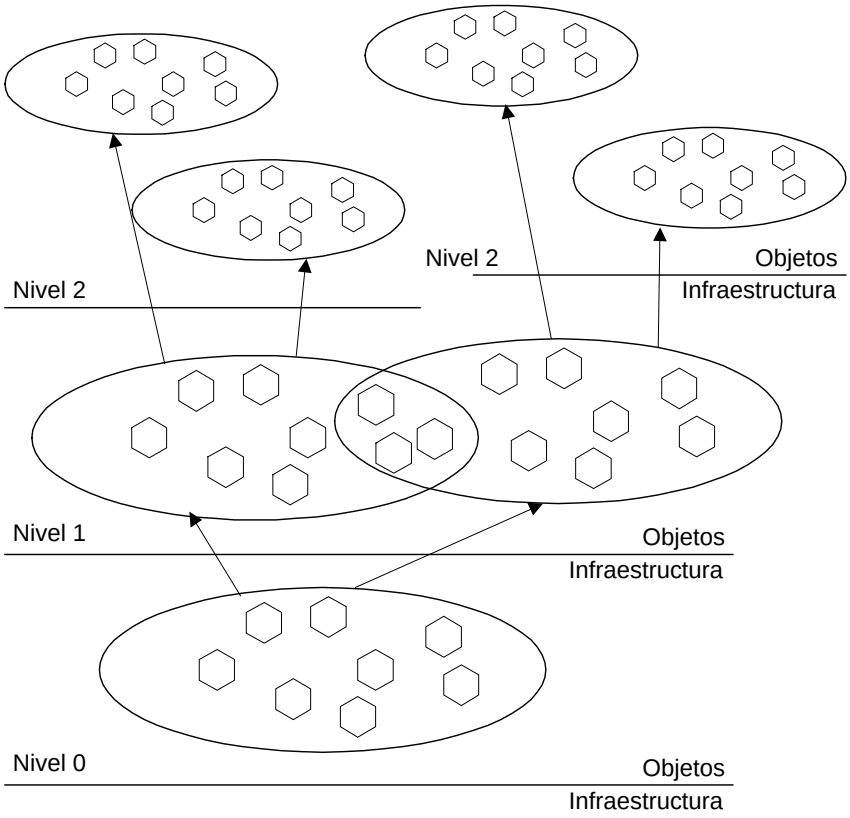


Figura 7.1 Estructuración multinivel de un sistema operativo.

Aplicando esto en términos de modelos de objetos, se trataría de ofrecer diferentes modelos de objetos que den soporte a instancias de cada modelo. A su vez, estas instancias podrían formar su propio modelo de objetos y dar soporte a otras instancias, aunque esto no es común. Normalmente sólo hay dos niveles. En uno se sitúan los soportes de diferentes modelos de objetos y en el superior las instancias.

Ejemplos de sistemas que ofrecen soporte para varios modelos de objetos son los sistemas COOLv2 [LJP93] y Tigger [Cah96a]. Estos sistemas están enfocados a dar soporte a los modelos de objetos de diferentes lenguajes. Para ello disponen de un soporte genérico en tiempo de ejecución para un modelo de objetos que es complementado por un soporte específico de cada lenguaje. La combinación de ambos soportes constituye el soporte específico de cada lenguaje.

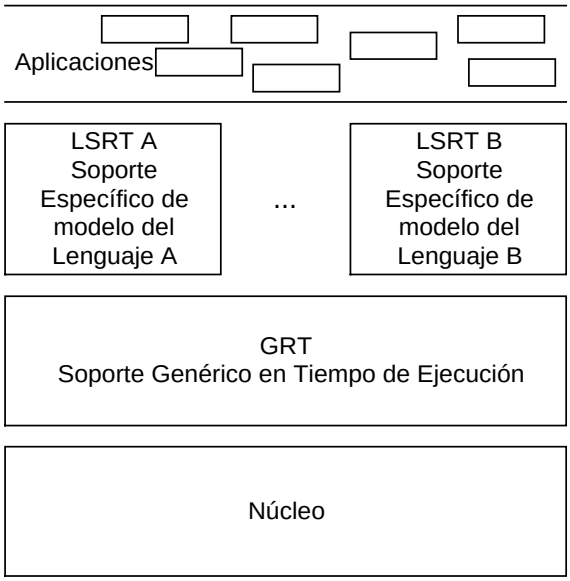


Figura 7.2 Soporte para varios modelos de objetos mediante soportes genéricos y específicos en tiempo de ejecución

7.1.1 Adaptación a las aplicaciones

Las ventajas de estas aproximaciones se fundamentan en la posibilidad de utilizar cualquier lenguaje de programación orientado a objetos, sin necesidad de llegar a ningún compromiso. Todas las características del lenguaje pueden utilizarse en principio sin ninguna restricción. Basta con crear la infraestructura que de soporte al modelo del lenguaje deseado con las herramientas que ofrezca el sistema. Las aplicaciones pueden utilizar un lenguaje específico apropiado a las mismas.

En la misma línea, las aplicaciones pueden utilizar una infraestructura que se ajuste exactamente a sus necesidades, creando un nivel de infraestructura particularizado dentro del sistema que soporte estas necesidades.

7.1.2 Problemas de aislamiento

Sin embargo, esta división entre infraestructuras recursivas e instancias, o la utilización de diferentes modelos de objetos, presenta un cierto problema de aislamiento. Por ejemplo, en el caso de diferentes modelos, es complicado comunicar los objetos de los diferentes modelos entre sí. Estos modelos pueden estar tan alejados semánticamente entre sí que la comunicación sea imposible. Del mismo modo, la reutilización de un objeto creado dentro de

un modelo en una aplicación que utilice un modelo diferente se complica. Nos encontramos de nuevo con problemas de desadaptación de impedancias y de interoperabilidad.

Dividir el soporte de un modelo de objetos en dos partes, una más general y otra más específica para cada modelo puede aliviar algo estos problemas, al igualarse diferentes modelos en la parte de utilización del soporte común. Sin embargo se introducen problemas adicionales al estar parte de la información de los modelos en los soportes específicos, con lo que hay que establecer mecanismos particulares en cada soporte específico de acceso a esta información y con el resto de los soportes, para permitir la interoperabilidad.

7.1.3 Complejidad conceptual y falta de uniformidad

El uso de diferentes modelos o infraestructuras introduce una complejidad conceptual adicional en el sistema. El usuario debe conocer las características particulares de cada uno de los modelos (o infraestructuras) utilizadas en el sistema, o al menos de las que va a utilizar. Dado que las ventajas del uso de varios modelos se obtienen si efectivamente se usa más de uno, el resultado es que esta falta de uniformidad conceptual con la complejidad que acarrea reduce la productividad del usuario que usa el sistema.

7.2 Tipo del modelo único

Muchos de los sistemas que utilizan un modelo único se plantean utilizar el modelo de objetos de un lenguaje de programación determinado, extendiendo de esta manera el lenguaje de programación a todos los elementos del sistema.

7.2.1 Sistemas para C++

Un ejemplo de este tipo de sistemas es Panda [ABB+93]. Panda es un sistema que pretende utilizar el lenguaje C++ sin modificaciones para la realización de programación paralela y distribuida. Se estructura como un pico-núcleo (núcleo mínimo) que proporciona unas abstracciones mínimas que son complementadas en espacio de usuario con unas librerías de clases en tiempo de ejecución que soportan el resto de las abstracciones. Estas abstracciones extienden el lenguaje C++ con hilos (objetos pasivos), un espacio común de objetos compartidos, distribución y persistencia.

Es una muestra de la eliminación de la desadaptación de impedancias entre los lenguajes OO y los sistemas, sin la penalización de las capas adicionales de adaptación, como demuestra la implementación sobre este sistema de soporte para un lenguaje especializados en paralelismo y distribución como COIN [Buh90].

7.2.2 Problemas de los modelos de objetos de los lenguajes de programación

La adopción del modelo de objetos de un lenguaje de programación conlleva una serie de problemas.

7.2.2.1 Uso de múltiples lenguajes

El uso del modelo de un único lenguaje de programación restringe demasiado la utilización del sistema. En el caso de Panda, sólo programadores en C++ podrán utilizarlo.

7.2.2.2 Modelos no pensados para los requisitos de un sistema operativo

Los modelos de la mayoría de los lenguajes de programación no están pensados para los requisitos de un sistema como el que nos ocupa.

Transparencia de localización

Esto se nota especialmente en áreas como la transparencia de localización. Los lenguajes, como por ejemplo C++, utilizan como manera de referenciar a un objeto (identificador) la posición de memoria virtual que utiliza. Aunque internamente para un programa es muy eficiente, esto evidentemente no funciona en un entorno distribuido en el que los objetos no tienen que estar en la misma máquina y ni siquiera en el mismo espacio de direcciones.

7.2.2.3 Heterogeneidad

La existencia de múltiples plataformas causa problemas a los modelos de objetos de lenguajes como C++ que, en general, dejan demasiados detalles al arbitrio de cada implementación concreta. Por ejemplo en el caso de la longitud de los tipos de datos básicos, etc. Esto no permite la movilidad de los objetos con facilidad entre diferentes plataformas, ya que la longitud de los tipos de datos básicos puede cambiar, etc.

7.2.2.4 Poca semántica del modelo de objetos

En muchos casos, la semántica del modelo de objetos del lenguaje se pierde tras el proceso de compilación. En el caso de C++, en tiempo de ejecución, un objeto es simplemente una zona de memoria, sin estructura [Str91]. La invocación de métodos, etc. la realiza el soporte en tiempo de ejecución del lenguaje usando una manera propia de cada compilador para colocar información de las clases, etc. Algunos sistemas, como CHEOPS [Sch97], retienen la información del modelo de objetos de C++, con una representación del mismo en tiempo de ejecución, eliminando en gran medida este problema.

7.3 Modelo único cercano al modelo de las metodologías

Es necesario utilizar un modelo que sea lo suficientemente potente como para poder representar los modelos de los lenguajes de programación más utilizados. Además deberá incorporar los conceptos del paradigma de la OO más aceptados, es decir, retener toda la semántica del modelo general de objetos implícito en las obras de referencia de la tecnología OO. De esta manera el número de usuarios que podrán utilizar y comprender el sistema sin dificultad será el mayor posible. Por otro lado necesitará conceptos que permitan realizar las características del sistema integral como la transparencia de localización.

Estos requisitos son en parte los mismos que tiene que tener en cuenta una metodología de análisis y diseño OO para escoger un modelo de objetos para sus fases. Una de las metodologías que más aceptación ha alcanzado es la metodología de Booch. Sobre ella se ha establecido un consenso generalizado aceptado por la mayoría de la comunidad de la orientación a objetos sobre cuales son las características más estandarizadas que debe de tener un modelo de objetos. Estas se encuentran descritas por Grady Booch en la obra de referencia *"Object-Oriented Analysis and Design with Applications, 2nd edition"* [Boo94]: Abstracción y encapsulamiento, herencia y polimorfismo son las propiedades fundamentales.

Usando un modelo con estas características se está relativamente cerca de los modelos de los lenguajes más utilizados, como C++, Smalltalk y Java. Los compiladores pueden generar los objetos usando el modelo único sin perder muchas características del modelo del lenguaje. Esto permite que los objetos puedan ser usados desde otro lenguaje, logrando los objetivos de multilenguaje e interoperabilidad de objetos.

Por otra parte el uso de un modelo similar al usado por las metodologías con más aceptación garantiza una fácil comprensión del mismo y el alcance al mayor número de usuarios posible.

Al utilizarse para las fases de análisis y diseño, obviamente se retiene toda la semántica del modelo de objetos, puesto que es la que se utiliza en estas fases para modelar la aplicación.

Por tanto, con la utilización de un modelo de objetos similar al de una de las metodologías más utilizadas como la de Booch, complementado con elementos específicos para el sistema integral, como un identificador global, se consiguen los objetivos:

- Accesible para el mayor número de usuarios posible.
- Soporte para múltiples lenguajes, especialmente los más utilizados.
- Toda la semántica del modelo de objetos.
- Adecuado para las características necesarias del sistema integral

7.3.1 Posibles inconvenientes del uso de un modelo único

Se puede argumentar que la utilización de un modelo único implica necesariamente que se pierden características de los modelos de objetos de los diferentes lenguajes. Esto evidentemente es cierto, al utilizar un único modelo se deben hacer corresponder las características de los modelos de los lenguajes con las del modelo único. En ciertos casos alguna característica no tendrá una correspondencia sencilla o incluso no existirá la correspondencia.

7.3.1.1 Pérdida de características de los lenguajes

Por tanto ciertas características de los lenguajes no podrán utilizarse o sólo se podrán utilizar con restricciones. Sin embargo en la práctica esto no es un inconveniente muy grande con los lenguajes más utilizados:

- Características **escasamente usadas**. Por una parte, al estar estos lenguajes cercanos al modelo único, las características que se pierdan no serán fundamentales en el lenguaje, ya que estas sí que estarán recogidas en el modelo único. En general serán características muy particulares y como tales su utilización en las aplicaciones sería pequeña.
- Características **no fundamentales**. Por otro lado, en cualquier caso, sólo las características del modelo de la metodología se usan en las fases de análisis y diseño y nunca características especiales de un lenguaje. Por tanto, no será necesario recurrir a estas características de un lenguaje en la fase de implementación. Basta simplemente con usar las características de la metodología que son precisamente las que se encuentran en el modelo único. Aunque algunos elementos nuevos, como la genericidad y las plantillas se empiezan a incorporar a las metodologías [BRJ96], no existe aún un consenso sobre si son elementos fundamentales del modelo de objetos. Normalmente se pueden solucionar mediante otros elementos del modelo. Por ejemplo, la genericidad puede resolverse mediante herencia múltiple.

7.3.1.2 Dificultad de uso de ciertos lenguajes

Es más difícil la utilización en el sistema de ciertos lenguajes que no utilicen los conceptos más usuales de la orientación a objetos, precisamente los usados en el sistema. Por ejemplo, lenguajes como Self [US87] no utilizan los conceptos de herencia y no tienen una representación directa en este sistema.

Dado que el objetivo del sistema es dar soporte directo a un modelo de objetos ampliamente aceptado y utilizado como el de las metodologías, esta dificultad de uso de

lenguajes poco difundidos con modelos especiales es un inconveniente menor que hay que aceptar.

7.3.1.3 Imposibilidad de experimentación con otros modelos de objetos

Relacionado con el punto anterior, se arguye que el soporte para un modelo de objetos único concreto hace, como es natural, que sea difícil experimentar en el sistema con otros modelos de objetos [PBK91].

Efectivamente, es difícil la experimentación con modelos de objetos que difieran radicalmente del modelo de objetos único del sistema. En cualquier caso, el objetivo fundamental del sistema no es experimentar con nuevos modelos de objetos, si no aprovechar la madurez del paradigma OO que ha llevado a la aceptación por la mayor parte de la comunidad OO de unos conceptos básicos y utilizarlo en todos los aspectos de un sistema. Así más que experimentar con conceptos nuevos en los modelos de objetos, se trata de aprovechar al máximo las ventajas del paradigma ya establecido en todos los elementos de un sistema, en lugar de sólo en algunos como se ha venido realizando hasta ahora.

7.4 Resumen

El uso de varios modelos de objetos no es adecuado para un sistema integral por razones de simplicidad conceptual y facilidad de interoperación entre los objetos. En el caso de un modelo único, una opción es utilizar el modelo de un lenguaje determinado. Sin embargo, esto impide la utilización de diferentes lenguajes, y además estos modelos no están pensados para ser utilizados fuera del contexto de su lenguaje, como parte de un sistema más general y suelen carecer de algunas propiedades semánticas normalmente asociadas al paradigma OO. Teniendo en cuenta el objetivo principal de aprovechar las tecnologías orientadas a objetos, la necesidad de soportar múltiples lenguajes y de ser fácil de comprender por el mayor número de usuarios posibles, la mejor elección para un sistema integral es un modelo similar al usado en las metodologías de análisis y diseño más utilizadas, como Booch, complementado con características adicionales útiles en un sistema integral, como identificadores globales.

Capítulo 8

DEFINICIÓN DEL MODELO DE OBJETOS DEL SISTEMA INTEGRAL

En este capítulo se definen las características del modelo único de objetos escogido como más adecuado para un sistema integral orientado a objetos. Se basa en las características más aceptadas que debe tener un modelo de objetos recogidas en la metodología de Booch [Boo91, Boo94]. Aunque existen muchos modelos de objetos y opiniones diferentes, estas características descritas en *"Object-Oriented Analysis and Design with Applications"* son generalmente aceptadas de manera tácita como las más representativas de la OO, siendo incluso la denominación usada por el autor la de "El Modelo de Objetos".

8.1 Características del modelo de objetos de Booch

Propiedades fundamentales:

- Abstracción
- Encapsulamiento
- Modularidad
- Jerarquía de herencia y de agregación

Propiedades secundarias:

- Tipos
- Concurrencia
- Persistencia

8.1.1 Abstracción y encapsulamiento. Clases

Estas propiedades suelen describirse como un conjunto. La **abstracción** se define en [Boo91, Boo94] como *"Una abstracción denota las características coincidentes de un objeto*[1] *que lo distinguen de todos los demás tipos de objeto y proporciona así fronteras conceptuales nítidamente definidas respecto a la perspectiva del observador".* Podría resumirse indicando que un objeto presenta una determinada interfaz que muestra su comportamiento esencial. Este comportamiento se presenta mediante una serie de métodos u operaciones que se pueden invocar sobre el objeto.

Esta propiedad se suele combinar con la de **encapsulamiento**, que establece el principio de ocultación de información: *"el encapsulamiento es el proceso de almacenar en un mismo compartimento los elementos de una abstracción que constituyen su estructura y su comportamiento, sirve para separar la interfaz contractual de una abstracción de su implementación"*. La implementación de un objeto (la implementación de sus métodos y sus

[1] Se utilizarán los nombres objeto e instancia indistintamente

estructuras internas o estado interno) debe estar totalmente encapsulada en el mismo y separada de su interfaz. La única manera de utilizar un objeto es mediante los métodos de su interfaz. No se puede acceder directamente a la implementación de un objeto.

También puede considerarse que el objeto encapsula en general toda su semántica, incluyendo las propiedades implícitas que puedan existir en el modelo de objetos, como la concurrencia, que se describen más adelante.

Como unidad de abstracción y representación del encapsulamiento, la mayoría de los sistemas OO utilizan el concepto de **clase**. Todos los objetos que tienen las mismas características se agrupan en una misma clase. La clase describe las características que tendrán todos los objetos de la misma: la representación del estado interno y su interfaz. Así, cuando se crea un nuevo objeto, se crea a partir de una clase, de la que toma su estructura.

8.1.2 Modularidad

La **modularidad**[1] permite fragmentar un problema complejo en una serie de conjuntos de objetos o módulos, que interactúan con otros módulos. En cierta manera, representa un nivel superior al nivel de encapsulamiento, aplicado a un conjunto de objetos.

8.1.3 Jerarquía. La relación "es-un" (herencia). La relación "es-parte-de" (agregación)

La **jerarquía** es "una clasificación u ordenación de abstracciones". Permite comprender mejor un problema agrupando de manera jerárquica las abstracciones (objetos) en función de sus propiedades o elementos comunes, descomponiendo en niveles inferiores las propiedades diferentes o componentes más elementales.

Las dos jerarquías más importantes son las representadas por las relaciones "**es-un**" y "**es-parte-de**".

8.1.3.1 Herencia. La relación "es-un"

Es una relación que se establece entre las diferentes clases de un sistema. Esta relación indica que una clase (**subclase**[2]) comparte la estructura de comportamiento (las propiedades) definidas en otra clase (**superclase**[3]). La clase "hereda[4]" las propiedades de la superclase. Se puede formar de esta manera una jerarquía de clases[5]. Un objeto de una clase también "es-un" objeto de la superclase (por ejemplo, un coche es un vehículo). En el caso en que una clase pueda heredar de más de una superclase se habla de **"herencia múltiple"** por contraposición al caso de **"herencia simple"**.

8.1.3.2 Agregación. La relación todo/parte ("es-parte-de")

Las jerarquías "es-parte-de" describen relaciones de agregación entre objetos para formar otros objetos de un nivel superior. Permiten definir un objeto en términos de agregación de sus partes, de objetos componentes más elementales (chasis, motor y carrocería son parte de un coche).

---

[1] Un excelente tratamiento sobre la modularidad y la orientación a objetos en general se encuentra en [Mey88, Mey97].
[2] O clase hija.
[3] O clase padre.
[4] O "deriva de".
[5] Las clases situadas por encima de una clase en la línea de herencia son los ancestros, antepasados o ascendientes. Las que derivan de ella son los descendientes.

8.1.4 Tipos y polimorfismo

"Los tipos son la puesta en vigor de la clase de los objetos, de manera que los objetos de tipos distintos no pueden intercambiarse, o, como mucho, pueden intercambiarse sólo de formas muy restringidas". En realidad un **tipo** denota simplemente una estructura común de comportamiento de un grupo de objetos. Normalmente se identifica el concepto de tipo con el de clase[1], haciendo que la clase sea la única manera de definir el comportamiento común de los objetos. De esta manera la jerarquía de tipos se funde con la jerarquía de clases. Así un objeto de un subtipo (subclase) determinado puede utilizarse en cualquier lugar en el que se espera un objeto de sus tipos ancestros (clases ancestras).

La existencia de tipos permite aplicar las normas de comprobación de tipos a los programas, ayudando a la detección de un número de errores mayor en el desarrollo de un programa. La **comprobación de tipos** permite controlar que las operaciones que se invocan sobre un objeto forman parte de las que realmente tiene. En la comprobación de tipos puede haber varios niveles de comprobación: comprobación estricta en la compilación de un programa (**comprobación estática**), sólo comprobación en tiempo de ejecución (**comprobación dinámica**) o una estrategia híbrida.

8.1.4.1 Enlace estático y dinámico. Polimorfismo

El enlace hace referencia al momento en el que un nombre (o referencia) se asocia con su tipo. En el **enlace estático** esto se realiza en tiempo de compilación. De esta manera la invocación de una operación sobre un objeto siempre activa la misma implementación de la operación.

En el **enlace dinámico**, el tipo del objeto se determina en tiempo de ejecución y el sistema determinará cuál es la implementación de la operación que se invocará en función de cual sea el objeto utilizado. Esto normalmente se denomina **polimorfismo** puesto que podemos aplicar el mismo nombre de operación a un objeto y en función de cual sea este objeto concreto el sistema elegirá la implementación adecuada de la operación.

Este mecanismo de enlace dinámico permite crear programas generales en los que el tipo de objeto sobre el que operen sea determinado en tiempo de ejecución. Por ejemplo un programa gráfico que simplemente indique a un objeto que realice la operación de dibujar. En tiempo de ejecución se enlaza dinámicamente con la implementación de la operación "dibujar" correspondiente al objeto concreto en tiempo de ejecución que se trate: un círculo, un rectángulo, etc.

El polimorfismo se suele considerar como una propiedad fundamental del modelo de objetos. Combinado con la herencia ofrece un compromiso entre la comprobación de tipos en tiempo de ejecución y la flexibilidad del enlace dinámico. De esta manera se puede usar el polimorfismo en un árbol de herencia compartiendo el mismo nombre de operación para las superclase y sus subclases, pero restringiendo la utilización de las operaciones a las que se pueden aplicar sobre el objeto (las de su árbol de herencia), siempre que pertenezca a una de las clases de la jerarquía.

8.1.5 Concurrencia

"La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo". La mayoría de los lenguajes OO están pensados para sistemas tradicionales con conceptos como fichero y proceso. Esto hace tender a que los objetos se consideren entidades

[1] En este trabajo se asociará siempre el concepto de tipo con el de clase, salvo indicación explícita de lo contrario.

"teledirigidas" por el flujo de ejecución secuencial del programa. Sin embargo, entre las propiedades de un objeto está la de la **concurrencia**, es decir, la posibilidad de tener actividad independientemente del resto de los objetos. Esto permite describir sistemas de una manera más cercana a la realidad, en la que los objetos en muchos casos tienen funcionamiento independiente.

8.1.6 Persistencia

"Es la propiedad de un objeto por la que su existencia trasciende el tiempo (es decir el objeto continúa existiendo después de que su creador deja de existir) y/o el espacio (es decir, la posición del objeto varía con respecto al espacio de direcciones en el que fue creado)".

Para que los sistemas OO sean más cercanos a la realidad, es necesaria la **persistencia**, es decir, un objeto no tiene por qué dejar de existir por el simple hecho de la finalización del programa que lo creó. El objeto representa una abstracción que debe permanecer como tal hasta que ya no sea necesaria su existencia dentro del sistema. Hay que hacer notar que no sólo los datos de un objeto deben persistir, la clase a la que pertenece también debe hacerlo para permitir la utilización del objeto con toda su semántica.

8.1.7 Distribución

Aunque no denominada como tal, la propiedad de la **distribución** es referida con el término de **persistencia en el espacio**. Un objeto puede moverse (en el espacio) por los distintos ordenadores que conformen el sistema de computación, manteniendo todas sus propiedades como en el caso de la persistencia. Es decir, los objetos son distribuidos.

Como se ve, ciertos objetivos del sistema integral orientado a objetos, como la concurrencia, la persistencia y la distribución, ya se encuentran reconocidos como parte fundamental del modelo de objetos. Sin embargo, la mayoría de los lenguajes y sistemas no suelen contar estas propiedades dentro de sus modelos de objetos.

8.2 Características adicionales necesarias

Algunas características necesarias para un sistema integral no están recogidas dentro de este modelo de Booch:

8.2.1 Relaciones generales de asociación

En un sistema, además de las relaciones "es-un" y "es-parte-de", existen más relaciones entre objetos. Es conveniente conocer qué objetos están relacionados con otros. Esta información puede ser útil cuando se tienen en cuenta aspectos de distribución de los objetos. Por ejemplo, al mover un objeto a otro ordenador, es interesante mover también los objetos que están relacionados, puesto que es muy probable que se intercambien muchos mensajes entre sí. Con la relación "es-parte-de" se pueden conocer los objetos integrantes de otro objeto y moverlos a la vez. Sin embargo, en el caso de un libro y la estantería donde está situado, la relación que existe entre ellos no puede modelarse como "es-parte-de". Si no existe otro tipo de relación en el modelo, el sistema no tendría conocimiento de esta relación y no podría explotarlo al igual que lo puede hacer con la relación "es-parte-de".

De hecho, relaciones adicionales a las de agregación y herencia son utilizadas por las metodologías. La propia metodología de Booch dispone de relaciones de uso entre objetos.

Como compromiso, se añade un tercer tipo de relación al modelo de objetos: la **relación de asociación** "asociado-con", que represente en general la existencia de otras relaciones diferentes a las de herencia o agregación entre objetos.

8.2.2 Identidad única de objetos

Al igual que en el mundo real, todos los objetos tienen una identidad que los diferencia del resto de los objetos. En un sistema integral homogéneo en el que todos los objetos están al mismo nivel, tiene que existir un mecanismo para que el sistema distinga un objeto de otros, al objeto de trabajar con el mismo. Al usarse sólo OO lo único que el sistema hará con un objeto es invocar uno de sus métodos. La manera de hacer esto es mediante un identificador que tenga cada objeto.

En lenguajes como C++, se utiliza la posición de memoria que ocupa el objeto para acceder al mismo. Sin embargo, en un sistema global como el nuestro, con necesidades de persistencia, distribución, etc. esto no es válido pues un objeto puede cambiar de ordenador. Es necesario utilizar un identificador global que identifique de manera unívoca cada objeto dentro del sistema hasta que el objeto desaparezca. Este identificador único será la única manera de acceder al objeto dentro del sistema. Como se verá, la utilización de un **identificador único** para los objetos facilita la realización de propiedades como la persistencia, distribución, etc. de manera transparente.

8.2.3 Excepciones

Las excepciones son un concepto que está orientado fundamentalmente a la realización de programas robustos mediante el manejo estructurado de los errores. Básicamente, una **excepción** es un evento que se produce cuando se cumplen ciertas condiciones (normalmente condiciones de error). En estos casos se "lanza una excepción". La excepción será "atrapada" por un manejador de excepciones que realizará el tratamiento adecuado a la misma.

Las excepciones son un concepto importante con soporte en lenguajes populares como C++ y Eiffel. Parece adecuado, pues, introducirlo en el modelo de objetos del sistema para promover el desarrollo de programas robustos. Por otro lado, puede utilizarse en la propia definición de la máquina abstracta, como se verá. También permite la integración fluida en el modelo de objetos de ciertas tareas fuera del ámbito de la programación típica, pero que debe realizar un sistema integral, como la gestión de interrupciones de dispositivos, etc. que pueden asimilarse a excepciones.

8.2.4 Seguridad

Aunque no sea de gran importancia en las fases de diseño y análisis de aplicaciones, en un sistema de computación real completo la seguridad es muy importante. Debe existir un mecanismo que permita la protección de los recursos del sistema, en este caso de los objetos.

En el caso del sistema integral, el objetivo de la protección son los objetos, más concretamente la utilización de los métodos de un objeto por parte de otros objetos. Se trata de posibilitar que sólo ciertos objetos puedan invocar métodos de otros objetos y en el caso de cada objeto qué métodos concretos puede invocar. Por ejemplo, un objeto estudiante podría invocar el método leerNota de un objeto acta, pero no el método ponerNota. Un objeto profesor podría invocar ambas operaciones. Es importante que al igual que en el resto de los conceptos del sistema, este mecanismo sea uniforme e igual para todos los objetos.

8.3 Modelo único: integración de características recogidas del modelo de Booch con características adicionales

El modelo de objetos de Booch deja abiertas diferentes posibilidades de realización de las propiedades mencionadas anteriormente. Aunque esto permite acoger muchos lenguajes y sistemas dentro de este marco de referencia, para el sistema integral se necesitan concretar ciertos aspectos en cada apartado.

Guiados por el objetivo de conseguir un modelo de objetos lo más "estandarizado" posible, se escogen las características más utilizadas en los lenguajes de programación populares y en general más aceptadas entre la comunidad OO, junto con las características adicionales descritas anteriormente.

- Abstracción y encapsulamiento
  - **Identidad única de objetos**
  - **Clases**
- Jerarquía
  - **Herencia múltiple (es-un)**
  - **Agregación (es-parte-de)**
  - **Relaciones generales de asociación (asociado-con)**
- Modularidad
- Tipos
  - **Comprobación de tipos, incluso en tiempo de ejecución**
  - **Polimorfismo**
- **Excepciones**
- **Concurrencia**
- **Persistencia**
- **Distribución**
- **Seguridad**

No todas estas características tienen que estar implementadas por el mismo elemento del sistema integral. De hecho, algunas de ellas, como la persistencia, es más interesante introducirlas de manera que (aunque transparente) su uso pueda ser opcional (como se verá). Por tanto, podemos dividir las características en grupos en función del elemento del sistema que se ocupe de implementarlas.

8.3.1 Máquina abstracta

Se encargará de proporcionar las características fundamentales de este modelo de objetos, especialmente las más cercanas a los elementos normalmente encontrados en los lenguajes orientados a objetos.

- **Identidad única de objetos**
- **Clases**

- **Herencia múltiple (es-un)**
- **Agregación (es-parte-de)**
- **Relaciones generales de asociación (asociado-con)**
- **Comprobación de tipos, incluso en tiempo de ejecución**
- **Polimorfismo**
- **Excepciones**

8.3.2 Sistema Operativo

Se encargará de conseguir las propiedades que más comúnmente se asocian con funcionalidad propia de los sistemas operativos:

- **Concurrencia**
- **Persistencia**
- **Distribución**
- **Seguridad**

Hay que reseñar que bajo el epígrafe de "sistema operativo" se agrupa todo lo referente a la manera de conseguir esta funcionalidad. En general, esta se facilitará mediante un conjunto de objetos normales que extiendan la máquina abstracta para proporcionar a todos los objetos del sistema estas propiedades de manera transparente.

Sin embargo, en el diseño del sistema operativo, se prevé que pueda ser necesario por determinadas razones, incluir elementos propios de funcionalidad anterior como parte de la máquina abstracta, o al menos repartir la misma entre los objetos del SO y responsabilidades introducidas en la máquina.

El apartado de la modularidad puede ser dejado a capas superiores del sistema, ya que está relacionado con aspectos como el desarrollo y modelo de componentes, etc.

8.4 Resumen

Para el sistema integral orientado a objetos se utiliza el modelo de objetos de Booch como base para el modelo de objetos único del sistema. Se concretan las propiedades generales del modelo de objetos de Booch en: Clases, Herencia múltiple ("es-un"), Agregación ("es-parte-de"), Comprobación de tipos, incluso en tiempo de ejecución, Polimorfismo, Concurrencia, Persistencia y Distribución. A estas propiedades se le añaden otras necesarias para el sistema: Identidad única de objetos, Relaciones generales de asociación ("asociado-con"), Excepciones y Seguridad. De todas estas propiedades la máquina abstracta será la encargada de implementar las fundamentales. El resto (Concurrencia, Persistencia, Distribución y Seguridad) serán proporcionadas transparentemente al resto de los objetos por el sistema operativo.

Capítulo 9

REQUISITOS DE LA MÁQUINA ABSTRACTA PARA EL SISTEMA INTEGRAL ORIENTADO A OBJETOS

La máquina abstracta que se utilice deberá proporcionar las características del modelo único de objetos descritas en el capítulo 8:

- **Identidad única de objetos**
- **Clases**
- **Herencia múltiple (es-un)**
- **Agregación (es-parte-de)**
- **Relaciones generales de asociación (asociado-con)**
- **Comprobación de tipos, incluso en tiempo de ejecución**
- **Polimorfismo**
- **Excepciones**

9.1 Principios de diseño

En cualquier caso existen una serie de principios de diseño que la máquina abstracta debe mantener, para no desmarcarse de la filosofía común del sistema:

9.1.1 Elevación del nivel de abstracción. Acercamiento de la OO al hardware

Se trata de acercar la gran distancia semántica existente entre el hardware tradicional y las tecnologías orientadas a objetos, haciendo que las aplicaciones trabajen con una máquina que utilice sus mismos términos OO.

Esto es una continuación de la tendencia existente desde los primeros momentos de la informática. La evolución de la programación, primero desde el código máquina al lenguaje ensamblador, y la utilización progresiva de lenguajes cada vez de más alto nivel hasta llegar a los lenguajes orientados a objeto son un ejemplo de esta tendencia.

Simplemente se trata de continuar esta tendencia con la arquitectura básica sobre la que funcionan las aplicaciones. El hardware ha evolucionado soportando conceptos utilizados en los lenguajes y entornos, aunque con una orientación al paradigma estructurado-procedimental y sistemas operativos para el mismo. Ejemplos son el soporte para la programación estructurada, con llamadas a procedimientos que ya proporcionan los procesadores (instrucciones *CALL*). También el soporte para multiprocesamiento (cambio de contextos por hardware, etc.) y la protección de procesos (memoria virtual y segmentación, etc.).

El auge de las tecnologías orientadas a objetos hace que ya sea el momento de proporcionar un soporte similar para estas tecnologías, en forma de máquina abstracta por razones de flexibilidad, portabilidad, etc. Esto no excluye que también deba tenderse a incluir este soporte en procesadores reales.

9.1.2 Uniformidad en la abstracción

La máquina deberá utilizar únicamente como elementos a manejar los del paradigma de la orientación a objetos, siguiendo la línea de uniformidad del sistema en torno a la abstracción de objetos.

9.1.3 Simplicidad

En sintonía con el principio anterior, se tenderá hacia la mayor sencillez posible en el diseño de la máquina. El número de conceptos que el usuario maneje deberá ser el menor posible. Se trata de hacer que el sistema sea fácil de aprender y, por tanto, de utilizar. Cuantos menos elementos no imprescindibles se introduzcan en la máquina abstracta, mayor será la facilidad de uso.

9.1.4 Portabilidad y heterogeneidad

Son dos objetivos del sistema que deben recordarse para el diseño de la máquina abstracta. No deberá introducirse en este diseño ningún elemento que comprometa alguno de estos objetivos.

Capítulo 10
PANORÁMICA DE MÁQUINAS ABSTRACTAS

En este capítulo se revisa una serie de máquinas que incorporan características de la orientación a objetos en su arquitectura o bien tienen características comunes con alguno de los objetivos del sistema integral. De esta manera se detectarán aciertos e inconvenientes de las distintas máquinas de acuerdo con los objetivos de diseño. El objetivo es encontrar una máquina que sirva para el sistema integral. En caso de no encontrar ninguna que se ajuste a los objetivos del sistema, se trata de identificar características interesantes para su incorporación en una máquina de nuevo diseño.

10.1 Máquinas abstractas versus máquinas reales

Las máquinas descritas son una selección tanto de máquinas reales implementadas en hardware como máquinas implementadas en software.

Conviene recordar que el nombre de **máquina abstracta** indica que la máquina tiene unas determinadas especificaciones que definen su funcionamiento. En este sentido estas especificaciones no necesariamente tienen que estar implementadas en hardware.

Cualquier tipo de máquina, esté implementada en hardware o no, se puede considerar como una máquina abstracta. Sin embargo, el nombre de máquina abstracta (o **máquina virtual**) se suele utilizar para hacer referencia a una máquina cuya definición no se realiza con el objetivo único de implementarla en hardware.

Normalmente el nombre se utiliza para aquellas máquinas que, por tanto, su implementación se realiza mediante software (emulador o **simulador** de la máquina). Sin embargo, un ejemplo de que cualquier máquina real puede considerarse una máquina abstracta son las emulaciones de máquinas reales implementadas en otras máquinas reales. Por ejemplo, la emulación de toda la arquitectura de un PC (procesador Pentium) utilizando un PowerMac [Tro97] (procesador PowerPC).

10.2 Máquinas reales

Dentro de las máquinas diseñadas para ser implementadas en hardware se seleccionan como los más representativos para su revisión a los procesadores IBM System/38, Intel iAPX432, Rekursiv y MUSHROOM.

10.2.1 IBM System/38

Bajo el nombre de IBM System/38 se agrupaba una familia de miniordenadores de gestión presentada el año 1978. La principal característica de esta familia es la utilización de procesadores con una interfaz de la máquina de más alto nivel que el común de la época [SH79].

Interfaz de la máquina de alto nivel independiente de la implementación

Por debajo de la interfaz de la máquina de alto nivel (instrucciones convencionales de usuario) se disponen dos niveles de instrucciones de microcódigo:

- **Microinstrucciones de memoria principal**. Que son similares a las instrucciones convencionales y se almacenan en memoria principal.
- **Microinstrucciones del almacenamiento de control** (*control store*). Se guardan y ejecutan en el almacenamiento de control. Entre otras cosas, este almacenamiento contiene un intérprete para las instrucciones de memoria principal. Puede bloquear rutinas frecuentes en el almacenamiento de control y usar la memoria principal para guardar otras que se cargarán según se necesiten.

De esta manera puede modificarse la implementación de bajo nivel de la interfaz de instrucciones de la máquina sin cambiar estas. Esta disposición en capas permite proporcionar fácilmente versiones mejoradas del sistema sin romper las aplicaciones de usuario.

Instrucciones de más alto nivel basadas en objetos

Las instrucciones del sistema tienen un nivel de abstracción alto y en general están basadas en objetos. Los objetos del sistema tienen un formato interno que no es visible para el usuario.

Existen unos **tipos básicos** de objetos del sistema (enteros, etc.) y objetos de usuario (**espacios de usuario**) que para la máquina son un conjunto de bytes que el usuario puede manipular.

Hay una cierta forma de instrucciones polimórficas para los tipos básicos (no para objetos de usuario). Por ejemplo la misma instrucción `ADD` sirve para sumar enteros, números en coma flotante, etc.

Capacidades

Para hacer referencia a los objetos básicos y a bytes dentro de un espacio de usuario la máquina gestiona directamente punteros. Un puntero es una zona de memoria de 16 bytes que contiene el identificador del objeto, permisos de acceso al mismo, etc. Ya que contienen información de protección pueden considerarse **capacidades** [DH66, Lev84]. Como pueden almacenarse en espacios de usuario, que pueden ser manipulados por el mismo, el hardware protege estas capacidades para que el usuario no pueda cambiar la información de protección. Esto se logra mediante bits de marca (*tag bits*) que identifican la memoria que contiene estos punteros. Se proporcionan, además, instrucciones especiales para manipular de manera coherente estos punteros. En caso de que el usuario acceda directamente a los espacios de usuario (puede hacerlo), se desactivan los bits de marca pues la información de protección podría haber sido cambiada.

Almacenamiento de nivel único

El sistema unifica la memoria principal con el almacenamiento secundario mediante el **almacenamiento de nivel único** (*single level store*). Para ello se usa un espacio de direcciones de memoria virtual único de 48 bits que abarca la memoria principal y la secundaria. Se utiliza paginación con marcos de página de 512 bytes y un hardware de traducción de memorias virtuales basado en tablas *hash*.

Mecanismo de seguridad

También se gestiona directamente un mecanismo de seguridad basado en una matriz de control de acceso implementada por hardware. Mediante tablas *hash* se almacena, para cada proceso y cada objeto (identificado por su dirección virtual) los permisos de acceso al mismo.

Ejecución de programas

Los programas con instrucciones de la máquina no se ejecutan directamente, si no que usando una instrucción especial se convierte el programa en un objeto programa (*program object*). Este objeto programa así convertido contiene la lista correspondiente de instrucciones de memoria principal que es ejecutada por el intérprete del almacenamiento de control. Cada instrucción normal se corresponde con una instrucción de microcódigo. Instrucciones muy complejas se transforman en instrucciones especiales SVL que llaman a una rutina de microcódigo que las ejecuta.

Por otro lado, también se gestiona por hardware tareas, conmutación entre tareas, comunicación entre tareas mediante colas de mensajes, etc.

10.2.1.1 Crítica

La arquitectura era muy avanzada para la época, y la familia de ordenadores que la usaba alcanzó gran éxito. Sin embargo, en realidad las aplicaciones que se utilizaban no sacaban partido de las características de orientación a objetos de la máquina, ya que estaban escritas en lenguajes no orientados a objetos.

Falta de uniformidad OO

A pesar de tener ciertas características de orientación a objetos, como el polimorfismo en algunas instrucciones, el uso de identificadores de objetos, etc. falta uniformidad en la aplicación de la misma. Por ejemplo, las operaciones polimórficas sólo pueden aplicarse a los tipos básicos y no a los objetos de usuario. Aparecen muchas instrucciones específicas para gestionar funcionalidades como conmutación entre tareas, comunicación por mensajes, etc. que se salen del marco de la OO y se solucionan con instrucciones ad-hoc.

En cualquier caso, el sistema no se estructura uniformemente sobre el paradigma OO, se da soporte a conceptos de sistemas tradicionales: tareas y comunicación entre tareas, etc.

Poca semántica OO

Los conceptos OO son muy reducidos. Los objetos de usuario (espacios de usuario) son para la máquina simples zonas de memoria que el usuario puede manipular libremente. El nivel de abstracción de los objetos de usuario, es bajo. Por tanto no se pueden aplicar todos los elementos de la OO a los objetos de usuario. No existe el concepto de herencia, etc.

Complejidad. Demasiadas características en hardware. Poca flexibilidad

La máquina se hace cargo de demasiadas funciones: tareas, protección, etc. Como ya se mencionó, esto lleva a la existencia de demasiadas instrucciones, específicas para cada una. Por otro lado, hacerse cargo de tanta funcionalidad implica la elección de determinadas políticas, como el uso de una matriz de control de acceso para la seguridad, y de decisiones de implementación, como el uso de tablas *hash*. Esto conlleva poca flexibilidad al obligar a utilizar esas decisiones que puede que no sean acertadas para las necesidades del usuario. A pesar de que en algunos casos pudieran cambiarse al estar ocultas por la interfaz de alto nivel, esto es difícil, al requerir modificaciones en el hardware.

10.2.1.2 Características interesantes

En cualquier caso, aparecen muchas ideas que están en sintonía con los objetivos del proyecto: La utilización de identificadores para los objetos, la necesidad de un mecanismo de protección ligado a los identificadores de los objetos no accesible directamente por el usuario, la unificación de la memoria principal y la secundaria en un solo espacio de direcciones, etc.

Interfaz de la máquina del más alto nivel posible e independiente de la implementación interna

Sobre todo es muy importante la idea de desligar lo más posible la interfaz de la máquina de la implementación interna. Se trata de que la interfaz de la máquina sea del más alto nivel posible y que nunca aparezcan en la interfaz elementos que obliguen a que la implementación interna tenga que estructurarse de una manera de bajo nivel determinada. De esta manera la implementación puede cambiarse sin necesidad de compromisos debido a la existencia de aplicaciones ya escritas.

10.2.2 Intel iAPX 432

El Intel iAPX 432 [Int81] fue un microprocesador basado en objetos desarrollado en los primeros años 80. Lo interesante de este procesador es que su diseño se realizó conjuntamente con un sistema operativo (iMAX) y el lenguaje ADA para lograr una uniformidad conceptual entre la arquitectura hardware, el sistema operativo y el lenguaje de programación [KCD+81].

El sistema trata de eliminar la distinción entre hardware y software proporcionando una visión uniforme del sistema al usuario. En este caso el paradigma que se usa para dar la visión uniforme es el de los conceptos del lenguaje ADA. ADA puede considerarse un lenguaje basado en objetos con encapsulamiento.

Objetos y direccionamiento

Los **objetos** son segmentos de memoria de hasta 128K. La estructura de un segmento (objeto) se divide en dos partes. Una contiene los datos propios del objeto (hasta 64K) y la otra se utiliza para referenciar otros objetos.

Este direccionamiento de objetos está basado en **capacidades**. En la parte de referencias del segmento se almacenan **descriptores de acceso** (capacidades) que constan de una parte que identifica el objeto (dentro de la tabla global de descriptores) y los permisos de acceso que se tienen para hacer referencia a dicho objeto.

La **tabla global de descriptores de objetos** se utiliza para localizar los objetos en el sistema. Cada entrada en la tabla contiene un descriptor que describe el segmento del objeto: dirección base y longitud, tipo del objeto, e información adicional, por ejemplo para recolección de basura.

Los objetos son simples zonas de memoria sin semántica asociada (objetos genéricos). Únicamente tienen semántica determinados tipos de objeto que utiliza el procesador. Entre los objetos básicos que gestiona la máquina se encuentran objetos para representar el procesador, procesos, almacenamiento y puertos. Para la protección dispone de **dominios**, que son un concepto similar al paquete de ADA.

Operaciones sobre los objetos

La máquina dispone de instrucciones específicas que utilizan y manipulan estos objetos básicos. Por ejemplo existen instrucciones send y receive para comunicar procesos que permiten pasar un objeto como parámetro, etc. También operaciones implícitas de la máquina, como la planificación de procesos usan los objetos básicos.

La sinergia en el diseño del sistema operativo y el procesador es grande. La funcionalidad se reparte entre ambos. Determinadas operaciones se implementan en hardware por ser importante su rapidez, requerir seguridad por hardware, etc. El sistema operativo cooperará con el procesador extendiendo la semántica que proporciona el hardware: gestión de objetos complejos, inicialización de objetos, etc.

10.2.2.1 Crítica

De este procesador se pensó incluso en ser el sucesor de Intel para la arquitectura x86 (el 80286 sería un puente intermedio hacia la misma). Sin embargo no tuvo éxito en el mercado, debido probablemente a que el equilibrio entre la funcionalidad adicional y el rendimiento no era adecuado para la época. El alto nivel de las características del iAPX 432 no tuvo una implementación suficientemente rápida: el procesador era demasiado lento frente a otros procesadores con las aplicaciones de la época.

Monolenguaje

La integración con el lenguaje ADA es demasiado grande. Al estar tan adaptado al lenguaje ADA, la utilización de otros lenguajes se dificulta.

Poca semántica de objetos

La manera de reparto de las características entre el sistema operativo y el procesador hace que la funcionalidad OO que implementa la máquina sea desigual. Por ejemplo, sólo los objetos básicos tienen semántica dentro de la máquina. Para los objetos genéricos, la máquina simplemente los considera como una zona de memoria.

Esto hace que el procesador sea prácticamente inútil sin el sistema operativo. Además prácticamente obliga a utilizar ese sistema operativo en concreto, sin poder elegir la funcionalidad mínima que se necesite.

Por otro lado, el sistema es simplemente basado en objetos, sin utilizar características comunes como la herencia.

No hay uniformidad de objetos. Distinción objetos usuario/sistema. Proliferación de instrucciones

Para los tipos de objetos básicos se definen instrucciones máquina específicas para los mismos, en lugar de utilizarlos como objetos normales mediante sus operaciones. Se distingue entonces la utilización de objetos del sistema (con operaciones específicas), de los objetos de usuario (genéricos, mediante llamadas a sus operaciones). Así por ejemplo un objeto de tipo puerto de comunicaciones tiene definidas las instrucciones send y receive. Para objetos de usuario se utiliza una instrucción genérica de llamada call.

La uniformidad no es total al existir aún la distinción entre objetos de usuario y de sistema. Esto produce una gran proliferación de instrucciones máquina para los objetos del sistema.

Falta de uniformidad OO

A pesar de dar cierto soporte a objetos, en realidad el sistema no está estructurado únicamente sobre el paradigma de la orientación a objetos. Se utilizan los objetos para dar soporte a conceptos de sistemas tradicionales: procesos y puertos de comunicación, planificación de procesos, etc.

10.2.2.2 Características interesantes

A pesar de no conseguirlo totalmente, la idea de eliminar las diferencias entre el hardware, sistema operativo y aplicaciones de usuario es una de las ideas fundamentales del sistema integral de objetos. Este es un sistema pionero en la filosofía de la uniformidad conceptual. La extensión de la semántica de la máquina mediante el sistema operativo es similar a la estructura propuesta para el sistema integral, en la que el sistema operativo compuesto por objetos normales de la máquina proporciona funcionalidad adicional a los mismos.

10.2.3 Rekursiv

Rekursiv [HGP+96, Pou88] es un microprocesador basado en objetos diseñado por la empresa Linn Smart Computing. El nacimiento de este microprocesador fue debido a la necesidad de mejorar el rendimiento del lenguaje Lingo, insuficiente con el hardware de la época. Este lenguaje fue desarrollado por la misma firma para implementar la gestión integral de la fábrica de componentes de alta fidelidad Linn.

Procesador basado en objetos. Identificadores de objetos. Distribución

Para reducir la distancia semántica entre el hardware y la orientación a objetos, el procesador proporciona el concepto de objetos y gestiona directamente la persistencia de los mismos. Cada objeto se identifica por un **identificador de objeto** (OID, *Object IDentifier*) de 40 bits, que la máquina asigna en su creación. Este identificador es la única referencia que se puede utilizar para acceder a un objeto, únicamente la máquina conoce la dirección interna de memoria de un objeto. En la memoria perteneciente a un objeto se pueden almacenar datos e instrucciones. Como parte de los datos se pueden almacenar identificadores de otros objetos.

La distribución se facilita dado que como parte del identificador de un objeto se incluye la identificación del procesador en que fue creado.

Agotamiento de los identificadores de objetos

Para reducir la posibilidad de que se agoten los identificadores disponibles (2^{40}), el procesador utiliza dos bits del identificador como marcas (con lo que en realidad sólo hay 2^{38} identificadores). El primero indica si los 38 bits pertenecen a un objeto o son un simple conjunto de bits sin tipo. El segundo indica que el objeto es un objeto compacto, que no necesita identificador y cuyo valor está contenido en los 38 bits restantes. Esto sirve para objetos de tipo entero, carácter, cadenas de caracteres, etc.

Aún así, se puede agotar el espacio de identificadores. En este caso se recurre a una solución drástica: se recorre el disco renumerando todos los objetos con identificadores consecutivos, para recuperar el espacio intermedio de identificadores no usado. Esto requiere un tiempo grande, con lo que debería hacerse fuera de línea. Es importante que ningún objeto creado escape al alcance del procesador, puesto que si se hace la operación de renumeración, no alcanzaría a estos objetos y existirían inconsistencias en los identificadores.

Persistencia. Fallos de objeto

El procesador gestiona directamente la persistencia de los objetos por hardware [HB87]. Una conexión directa a discos de almacenamiento secundario permite extender transparentemente la memoria de objetos del procesador a todo el espacio disponible en disco. Para almacenar los objetos en disco, al identificador del objeto se le añade una referencia adicional que representa el tipo del objeto. Esta referencia es a su vez un identificador de un objeto. Se deja a capas superiores del software que interpreten esta referencia de manera conveniente.

Cuando se hace referencia a un objeto que no se encuentra en la memoria del procesador, se produce un **fallo de objeto**. El hardware se encarga de manera transparente de buscar el objeto en disco y cargarlo en la memoria de manera apropiada.

Gestión de memoria. Compactación de huecos

El procesador también realiza transparentemente una recolección de basura de objetos no usados en memoria. En realidad lo que realiza es una **compactación de huecos** en memoria. Para ello divide la memoria en dos partes, de las que usa en inicio sólo una. Cuando esta parte se agota (ya no caben más objetos), graba todos los objetos al disco. Así puede detectar qué objetos son nuevos (no existían en el disco), o han sido modificados. Estos son los que se dejarán en memoria, simplemente se copian consecutivamente en la otra parte de la memoria, quedando compactados los huecos. En realidad se trata de mantener en la memoria los objetos más recientemente usados.

En caso de que aún así no haya suficiente espacio en memoria, se expulsan objetos al disco para liberar espacio. En último extremo se usa también la otra parte de memoria.

Juego de instrucciones de alto nivel programable

La distancia entre las aplicaciones y el hardware se puede reducir aún más puesto que el juego de instrucciones del procesador es programable. Cada aplicación puede definir su propio juego de instrucciones de alto nivel, que internamente se definen en términos de instrucciones de microcódigo de más bajo nivel [Har86]. El nombre Rekursiv hace referencia a que en la definición de una instrucción de alto nivel se puede llamar recursivamente a la propia instrucción.

Se pueden utilizar hasta 2^{10} **instrucciones primitivas**, puesto que el código de operación tiene 10 bits. La definición en microcódigo de estas primitivas se almacena en el almacenamiento de control de 16Kpalabras de microcódigo. Una tabla adicional hace corresponder cada instrucción primitiva con la dirección de inicio en el almacenamiento de control de su definición en microcódigo.

Una memoria caché dentro del procesador permite almacenar trozos de código para un acceso más rápido. En esta zona podrían almacenarse, por ejemplo, las rutinas que implementasen un intérprete de un lenguaje, como por ejemplo Smalltalk [HB86], acelerando aún más su rendimiento.

10.2.3.1 Crítica

Falta de semántica OO

El procesador es basado en objetos. No soporta conceptos como la herencia, etc. Por otro lado, no incorpora toda la semántica de objetos dentro del propio procesador. Por ejemplo se deja la interpretación del tipo del objeto a capas superiores de software.

Además, la manera de funcionamiento no es totalmente OO. Aunque se pueden definir las instrucciones que se deseen, y esto facilita la implementación de lenguajes orientados a objetos, la manera de funcionamiento no es OO, no existe el concepto de llamada a método como tal, etc.

Demasiada complejidad provocada por una excesiva flexibilidad en las instrucciones

La excesiva flexibilidad debida a la posibilidad de cambiar el juego de instrucciones definiéndolas a voluntad produce una complejidad muy grande. Por una parte el número de instrucciones es tan elevado que es prácticamente imposible conocerlas todas. Por otro lado, la posibilidad de cambiarlas es contraproducente para el objetivo de la uniformidad en un

sistema integral distribuido. Si cada aplicación puede cambiar las instrucciones, no existe un marco común cuya comprensión permita utilizarlo en todos los ámbitos. En realidad, más que orientadas a objetos, estas instrucciones más bien pueden considerarse como macros, trozos de programa cuya ejecución es más rápida al expandirlos en línea el procesador.

Poca flexibilidad por la implementación hardware

Como ocurre con las implementaciones en hardware, el problema es que las decisiones de diseño que se utilicen son casi imposibles de modificar. Por ejemplo, aunque el concepto de gestionar la persistencia directamente es muy interesante, el problema es que obliga a utilizar unos algoritmos de compactación de huecos, almacenamiento en disco, etc. que no pueden ser cambiados. Esto reduce la flexibilidad, impidiendo la adaptación o afinamiento de los mismos. Por otro lado, obliga a la utilización de estos recursos, incluso en ciertos casos en que no sean necesarios, por ejemplo un sistema empotrado que no necesite compactar huecos, ni persistencia.

No hay previsión para la seguridad

No existe ninguna previsión para la implantación de un mecanismo de seguridad, como las capacidades. Posiblemente sea debido a ser parte de los objetivos de aplicación del procesador, más pensado para dar soporte a una única aplicación concreta en la que para todos los objetos se sabe que no comprometerán la seguridad del sistema.

10.2.3.2 Características interesantes

A pesar de las avanzadas ideas presentes en este procesador, no tuvo éxito comercial. Sin embargo, demostró la viabilidad de la construcción de procesadores que aumentaran el nivel del hardware acercándolo a las aplicaciones OO. Ejemplo de esto son el rendimiento de implementaciones de lenguajes como Smalltalk que se realizaron sobre Rekursiv.

Es muy interesante la idea de proporcionar la persistencia de objetos de manera transparente usando técnicas de memoria virtual: la extensión del espacio de memoria del procesador mediante el espacio en disco. Las técnicas utilizadas para implementar la persistencia pueden ser utilizadas como referencia. De nuevo se ve la importancia de la existencia de un identificador de objetos. También puede ser una referencia válida las estrategias de reducción del uso de identificadores. Aunque el número de instrucciones sea excesivo, es importante la independencia de las mismas de la implementación interna. En general, si se diseña bien el juego de instrucciones, la interfaz de las instrucciones de la máquina no tiene por qué condicionar que las aplicaciones usen unas determinadas estructuras internas impuestas por la máquina.

10.2.4 MUSHROOM

El proyecto MUSHROOM [WW93], aunque no exactamente una máquina orientada a objetos, trata de desarrollar estructuras hardware mejores sobre las cuales desarrollar la orientación a objetos.

Direccionamiento en 2 niveles para soportar objetos sencillos. Cachés y hardware eficiente

La idea fundamental es abandonar el direccionamiento lineal de memoria tipo Von Neumann por un **direccionamiento en 2 niveles**: segmentos (que serán los identificadores de objetos) y desplazamientos (de 8bits) dentro de los segmentos para acceder a la información de los objetos. Es decir, un objeto es un segmento de memoria de 256 bytes de longitud. El segmento no está relacionado con la dirección física de almacenamiento del objeto. Las

instrucciones son del tipo convencional RISC, de carga y almacenamiento. En lugar de usar direcciones de memoria lineal usan este direccionamiento a dos niveles.

Se necesita hardware especializado para soportar este tipo de direccionamiento. Los objetos mayores de 256 bytes se construiran por software mediante tablas del objeto que contengan todos los segmentos que lo compongan. Es importante la existencia de **memorias caché** para los datos y la traducción de direcciones para gestionar eficientemente estas estructuras.

Traducción de direcciones

El mecanismo de **traducción de direcciones** (de identificador de objeto a dirección real de memoria) puede usar **tablas de objetos** de hasta 4 niveles. Existe una memoria caché que almacena las últimas traducciones. Cuando un identificador no está en la caché, una función software proporciona al hardware la dirección de la tabla de objetos correspondiente y la traducción continúa automáticamente.

Transferencia eficiente de objetos de memoria principal a secundaria

Se utiliza un **mecanismo de paginación** mejorado en el acceso a memoria secundaria para gestionar los objetos de bajo nivel (al fin y al cabo son simplemente páginas). Cuando un objeto está inactivo, saldrá de la caché y seguirá en memoria principal pero se le coloca en una lista de objetos inactivos. En caso de que el sistema de paginación necesite espacio en memoria, toma objetos de la lista de inactivos hasta llenar un bloque de disco y los graba. Se sigue, por tanto, una política del menos recientemente usado.

Sistema de recolección de basura a dos niveles

El sistema de **recolección de basura** funciona a dos niveles. En un primer nivel se realiza en la caché de datos y en un segundo nivel en memoria principal. De esta manera no hay interferencia entre ellos y es más eficiente.

Caché visible por software para asignación eficiente de objetos

Utilizando una caché que pueda ser gestionada por software, se pueden crear directamente los objetos en la caché. Esto permite realizar recolección de basura en la propia caché, con un límite máximo de tiempo empleado. Conocer el tiempo máximo que tarda la recolección de basura es importante para su uso en sistemas en tiempo real.

Espacio de direcciones suficientemente grande

Es necesario tener un espacio de direcciones que sea suficientemente grande para dar identificadores a todos los posibles objetos.

Aplicación a arquitecturas convencionales

El proyecto propone aplicar algunas de estas técnicas a procesadores convencionales. Serían modificaciones menores a estos procesadores que permitirían dar un soporte más adecuado a los patrones de comportamiento de las aplicaciones orientadas a objetos. Fundamentalmente se trata de adoptar el direccionamiento a dos niveles y la posibilidad de que la caché sea controlable por software.

10.2.4.1 Crítica

En realidad el enfoque de este proyecto está más destinado a lograr la eficiencia a muy bajo nivel de los patrones de uso de la OO, pero sin abandonar del todo estructuras convencionales de hardware.

Poca semántica del modelo de objetos

Permite que se pueda implementar más eficientemente lenguajes orientados a objetos que en arquitecturas más convencionales, pero en realidad no da soporte directo a los conceptos OO. Un objeto es simplemente un segmento de memoria de 256 bytes sin semántica asociada. Objetos más complejos ya no se manejan directamente. El juego de instrucciones es un juego convencional no orientado a objetos.

10.2.4.2 Características interesantes

Lo más interesante del proyecto es la producción de ideas que podrían mejorar la eficiencia del soporte de objetos en arquitecturas convencionales. Dado que el soporte hardware de los conceptos OO no parece tener éxito, al menos hasta ahora, es bueno mejorar el rendimiento de los procesadores convencionales cuando sobre ellos se utilizan sistemas OO. Es un ejemplo menos radical de acercamiento del hard a las tecnologías OO.

Aún así, de aplicación directa es la necesidad de desligar el direccionamiento físico de la memoria y sustituirlo por un direccionamiento a través de identificadores de objetos. También puede aplicarse la idea de separar la recolección de basura en dos niveles que no interfieran el uno con el otro, caso de utilizarse recolección de basura.

10.3 Inconvenientes de las máquinas reales en general

Pueden distinguirse una serie de inconvenientes comunes genéricos de las máquinas reales.

10.3.1 Inconvenientes propios del hardware

Existen una serie de inconvenientes de estas máquinas que se deben a su naturaleza propia de hardware:

10.3.1.1 Sustitución del hardware existente

Para aprovecharlas es necesario utilizar cada procesador particular. Esto requiere deshacerse del hardware existente y sustituirlo de golpe por estos nuevos procesadores. Este tipo de inversión es demasiado arriesgada y es una causa del fracaso comercial de la mayoría de estos procesadores orientados a objetos.

10.3.1.2 Falta de portabilidad binaria y heterogeneidad

Precisamente al ser plataformas hardware, para la portabilidad de las aplicaciones de unos ordenadores a otros es necesario que tengan la misma arquitectura. Teniendo en cuenta que es improbable la estandarización de un nuevo procesador OO de la misma manera que en el caso de la arquitectura x86 de Intel, el entorno de un sistema integral es heterogéneo. La portabilidad de las aplicaciones sería muy reducida perdiéndose uno de los objetivos del sistema integral.

10.3.1.3 Inflexibilidad de la implementación hardware

Independientemente de las características incluidas en el hardware, el problema es que es imposible (o prácticamente imposible) cambiar éstas. Por tanto, las aplicaciones deben plegarse a las imposiciones resultantes de estas decisiones, sean o no adecuadas para la aplicación.

10.3.2 Difícil elección de características a soportar por el hardware

Es difícil decidir qué características de la OO o adicionales deben ser incluidas en el hardware. En general, el problema suele ser que no se incluyen todas las características

básicas de la orientación a objetos, si no un subconjunto arbitrario de ellas, como en el caso del iAPX 432. En otras ocasiones se incluye demasiadas características adicionales en el propio procesador, como en el Rekursiv.

10.3.3 No se incluye toda la semántica del modelo de objetos. No son arquitecturas totalmente OO

En cualquier caso, el principal problema es que estas arquitecturas no incorporan toda la semántica aceptada del modelo de objetos. Probablemente para facilitar la implementación en hardware, no existe el concepto de herencia, etc. En general, los juegos de instrucciones no son orientados a objetos, si no que pueden utilizar objetos en algunos casos, etc.

10.4 Máquinas abstractas

La característica fundamental de las máquinas abstractas es que su principal destino es la implementación software. Aunque no se excluye que se puedan implementar mediante hardware, la certeza de que existirán mediante software elimina los inconvenientes ligados a la naturaleza hardware de las máquinas reales.

No es necesario sustituir el hardware existente con anterioridad para utilizarlas. Basta desarrollar el simulador software necesario para cada plataforma. De esta manera se pueden utilizar en entornos heterogéneos sin dificultad, y el código binario de la máquina abstracta puede funcionar en cualquier plataforma (en la que se haya implementado el simulador). Además se tiene la flexibilidad del software que permite modificar fácilmente decisiones de implementación, añadir o eliminar características, etc.

Intentando cubrir el abanico de características de las máquinas abstractas, se revisan UNCOL, ANDF, la Máquina-p, la máquina de Smalltalk, la máquina JVM de Java y la máquina Dis del sistema Inferno.

10.4.1 UNCOL

La idea de utilizar un juego de instrucciones de una máquina no real no es nueva. El proyecto UNCOL (*Universal Computer Oriented Language*) [Ste61] proponía la utilización de un **lenguaje intermedio universal**. El objetivo era la reducción del esfuerzo de implementación de compiladores de diferentes lenguajes para distintas arquitecturas. Los compiladores de cualquier lenguaje generarían código para este lenguaje intermedio (el juego de instrucciones de una máquina abstracta). A partir de este lenguaje intermedio se usarían generadores de código para diferentes plataformas hardware. De esta manera el compilador de un lenguaje serviría para todas las plataformas y el generador de código de una plataforma para todos los lenguajes.

La finalidad del UNCOL no era la portabilidad del código, si no la facilidad de desarrollo de compiladores. El proyecto era demasiado ambicioso para la época, y nunca llegó a ser completamente definido ni implementado. Se propusieron algunos UNCOL pero poco detallados. Sin embargo la idea de uso de un lenguaje intermedio para independizar el código de la plataforma fue aprovechada posteriormente.

10.4.2 ANDF

ANDF (*Architecture Neutral Distribution Format*) [Mac93a, Mac93b] es un lenguaje intermedio de bajo nivel, independiente del lenguaje y de la arquitectura. Su objetivo es permitir la distribución universal de software mediante un formato único.

Puede considerarse un heredero del UNCOL. Ayuda a conseguir la portabilidad del software tal y como se desarrolla en la actualidad. Existen problemas conocidos cuando se distribuye código fuente para lograr la portabilidad: existencia de diferentes dialectos e implementaciones del código fuente, dependencias del entorno, necesidad de uso de determinadas librerías específicas, etc. Cuando se distribuye código binario el inconveniente es que sólo funciona en plataformas compatibles; además, aún en este caso se necesita compatibilidad de formatos objeto, hay diferencias en el acceso a los recursos del sistema, etc.

ANDF ayuda a solucionar estos problemas mediante la definición de un lenguaje intermedio que tiene en cuenta todos estos problemas. Por ejemplo, el lenguaje es independiente de la plataforma y del lenguaje fuente. Tiene una serie de tipos de datos básicos cuya definición es concreta e independiente de la plataforma. Las instrucciones son suficientemente genéricas para ser utilizadas por los lenguajes procedimentales más utilizados. Existen mecanismos para indicar el uso de librerías que utiliza el programa, etc.

La manera de utilización es la típica del UNCOL: el programa en lenguaje fuente se pasa por un generador de ANDF. Además se indican los interfaces externos (librerías) que utiliza el programa. Se genera un código en ANDF que contiene toda la información en un formato independiente. Este código ANDF se puede distribuir a cualquier plataforma. En cada plataforma un instalador se encarga de tomar el código ANDF, enlazarlo con los interfaces que utiliza tal y como existen en la plataforma y generar un ejecutable en la plataforma destino.

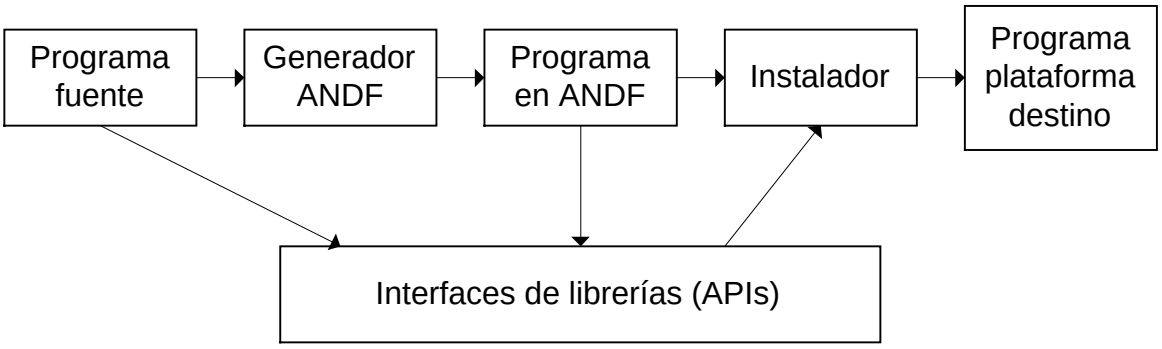


Figura 10.1 Utilización de ANDF

10.4.2.1 Crítica

Es un proyecto de aplicación de las ideas del UNCOL más moderno, aunque enfocado a arquitecturas convencionales.

Demasiado bajo nivel. No pensado para OO

El lenguaje es un lenguaje de demasiado bajo nivel y pensado para los programas software escritos en lenguajes tradicionales y en arquitecturas tradicionales. Por otro lado sólo ataca el problema de la distribución del mismo código a distintas plataformas. Esto no resuelve el problema de la movilidad de código (portabilidad del binario) ya que es necesario instalarlo, convertirlo al código binario de cada plataforma.

Al no estar pensado para tecnologías OO, el problema de la interoperabilidad entre objetos no se resuelve tampoco.

10.4.2.2 Características interesantes

El propio uso de una forma independiente de la plataforma del código (lenguaje intermedio de una máquina abstracta) es fundamental en un sistema integral orientado a objetos, pero sin la necesidad de instalación.

Lenguaje intermedio sin aspectos dependientes de la implementación

Es importante también establecer una definición concreta de todos los elementos del lenguaje: objetos básicos del sistema, etc. Es decir, no deben existir en el lenguaje intermedio definiciones que usen las aplicaciones cuya concreción quede a cargo de cada implementación (como por ejemplo el tamaño de un entero en C). Esto conlleva diferentes comportamientos del mismo código dependiendo de la plataforma, lo que va en contra de la portabilidad y la heterogeneidad.

10.4.3 Máquina-p

El Código-p [NAJ+76] es el código de la máquina abstracta **Máquina-p**. Fue desarrollado como lenguaje intermedio para un compilador de Pascal estándar [NAJ+76], para facilitar la portabilidad de programas Pascal a diferentes plataformas. Se desarrollaron intérpretes de Código-p (máquinas-p) para numerosas plataformas.

Es estrictamente una máquina de pila, ya que el lenguaje Pascal es un lenguaje estructurado y orientado a pila. La estructura interna fundamental de la máquina es una pila en la que se representa la pila de ejecución de un programa Pascal.

Las instrucciones de la máquina (el **Código-p**) están orientadas a esta estructura. Permiten introducir parámetros en la pila, llamar a un procedimiento (que tomará sus parámetros de la pila), retornar de un procedimiento (que devolverá el resultado en la pila), realizar operaciones aritméticas, lógicas, etc. con los valores almacenados en la pila, etc.

10.4.3.1 Áreas de la Máquina-p

La máquina se compone de tres áreas:

- **Depósito de código** (*Code Pool*). Compuesto por un conjunto de segmentos en el que se almacenan los procedimientos (el Código-p de los procedimientos). Los segmentos de código pueden relocalizarse en memoria en caso de necesidad, e incluso llegar a intercambiarse con memoria secundaria (*swapping*).
- **Pila de ejecución**. Permite realizar la ejecución de un programa Pascal; en esencia llamadas a procedimientos y operaciones internas de los procedimientos. Compuesta por un conjunto de registros de activación de Pascal apilados. Cada llamada a un procedimiento crea un registro de activación. En este registro se almacena información de estado de la activación del procedimiento, como la dirección de retorno del procedimiento, las variables locales del mismo, la dirección del registro de activación anterior, etc. Además, se utiliza para realizar todas las operaciones que desarrolle un procedimiento.
- **Segmentos de datos globales**. Donde se puede almacenar información global que pueda ser accedida por todos los procedimientos, como las variables globales, etc.

El Código-p es uno de los lenguajes intermedios que más éxito han tenido, incluso en el campo comercial, habiendo marcado una trayectoria en la construcción de compiladores que incluso continúa en la actualidad. Por ejemplo, en los compiladores de Microsoft se utilizó extensamente una variante del Código-p [Mic91].

Su mayor difusión se alcanzó cuando la UCSD (Universidad de California en San Diego) lo adoptó para su proyecto de Pascal portable. Llegó a disponer de soporte para multitarea. Incluso se desarrolló un sistema operativo completo basado en la Máquina-p. El UCSD p-System [Cam83] era un sistema operativo portable, escrito completamente en Pascal y sobre la Máquina-p. Tenía soporte para Entrada/Salida, sistema de ficheros, soporte para impresoras, controladores de dispositivos, etc. Entre otras plataformas, se comercializó para el IBM-PC original.

La Máquina-p también se implementó en hardware. Western Digital desarrolló el WD9000 Pascal Microengine en 1980, una implementación hardware de la Máquina-p que se utilizó en microordenadores comerciales.

10.4.3.2 Crítica

El principal problema con esta máquina es su enfoque para dar soporte al paradigma procedimental.

No es OO. El bajo nivel de la estructura de pila condiciona el soporte de lenguajes y la implementación

Es evidente que esta máquina no está destinada a dar soporte a la orientación a objetos. Está enfocada únicamente a dar soporte al lenguaje estructurado Pascal. La estructura interna de pila hace que además del Pascal, pueda hasta cierto punto dar soporte a lenguajes que estén basados en pila, como C. Sin embargo, la imposición de la estructura de pila hace que otro tipo de lenguajes no tengan un soporte tan sencillo. La orientación a pila del repertorio de instrucciones hace que éste no sea de un nivel uniforme. La estructura fundamental de alto nivel, la llamada a procedimiento, se mezcla con operaciones de bajo nivel con una pila. Además, la pila impone una estructura interna de bajo nivel en la máquina, que en cierta manera condiciona la implementación.

10.4.3.3 Características interesantes

La Máquina-p es un punto de referencia fundamental en el campo de las máquinas abstractas. Su existencia y su diseño interno siguen aún teniendo gran importancia. Por ejemplo, la plataforma Java debe gran parte de sus ideas a todo lo que representa la Máquina-p.

Aunque no existan características técnicas que se puedan adaptar directamente para los objetivos del sistema, la Máquina-p sirve de referencia para demostrar la viabilidad de ciertas premisas y decisiones de diseño del proyecto.

Máquinas abstractas viables

La difusión y el éxito del Código-p y la Máquina-p demuestran la viabilidad del enfoque basado en máquinas abstractas para la portabilidad del software. El hecho de que hayan tenido éxito en una época en la que el rendimiento del hardware era mucho menor y el precio mucho mayor hace pensar que a medida que aumenta el rendimiento del hardware y cae su precio su viabilidad sea aún mayor.

Es posible desarrollar sistemas operativos sobre una máquina abstracta

También es importante resaltar que una máquina abstracta no sólo puede dar soporte a programas escritos en un lenguaje. El UCSD p-System demuestra que se puede desarrollar totalmente un sistema operativo estrictamente alrededor de una máquina abstracta. Y no sólo destinado a la investigación, si no que también puede ser un sistema comercial en toda regla.

Implementación en hardware posible

El desarrollo de implementaciones en hardware de la Maquina-p confirma la posibilidad de desarrollo de hardware que incorpore conceptos de alto nivel más cercanos a los lenguajes de aplicación. Estos desarrollos hardware pueden estar basados en una máquina abstracta diseñada fundamentalmente para su implementación mediante software de emulación (intérpretes del código de la máquina abstracta).

10.4.4 La máquina virtual de Smalltalk

Smalltalk [GR83] se suele considerar como uno de los lenguajes orientados a objeto más puros. Prácticamente desde su concepción se utilizó como vehículo de implementación de Smalltalk una máquina virtual. En este caso, más que por razones de portabilidad, el uso de una máquina virtual interpretada es para aprovechar las ventajas de los intérpretes en cuanto a disponer de un sistema dinámico de desarrollo y depuración rápido. De hecho, esto permite que el propio entorno de desarrollo de Smalltalk (sistema Smalltalk) esté escrito en Smalltalk y pueda modificarse como cualquier otro programa. En cualquier caso, también se facilita la portabilidad del entorno a otras plataformas.

Una máquina virtual para Smalltalk-80 se describe en [GR83, Kra81]. En ejecución, un sistema Smalltalk se compone de la **máquina virtual** y una **imagen** que es la representación para la máquina de los objetos que componen el sistema. El compilador del sistema convierte el código Smalltalk de los métodos en un conjunto de instrucciones de la máquina virtual (código de la máquina virtual). La representación es simplemente un conjunto de bytes que representan las instrucciones de la máquina virtual con sus operandos, por eso se le suele llamar ***bytecode*** (código de bytes) tanto al conjunto como a cada instrucción individual. El lenguaje Smalltalk sólo maneja el concepto de objeto, lo que también se refleja en la máquina virtual.

En la implementación de la máquina virtual pueden distinguirse tres elementos: el gestor de memoria (storage manager), el intérprete de los bytecodes y una serie de subrutinas primitivas.

10.4.4.1 El gestor de memoria (storage manager)

Se ocupa de gestionar la memoria para representar los objetos dentro de la máquina.

Cada **objeto** se identifica internamente mediante un **identificador** (*object-pointer*, puntero a objeto, en la terminología de esta máquina) que es usado por el resto de los elementos para hacer referencia al objeto. La máquina mantiene una tabla que le permite localizar a un objeto a través de su identificador.

En la representación de un objeto, se dispone un puntero a la clase de la que deriva y los datos propios del objeto (punteros a sus objetos componentes). Las **clases** se representan a su vez mediante un objeto normal.

Por razones de eficiencia, ciertos **objetos primitivos**, como enteros, caracteres, etc. se representan directamente con sus valores codificados en los datos en lugar de mediante un puntero a los mismos.

Cada objeto tiene un contador que indica el número de objetos que le hacen referencia. El gestor actualiza este contador adecuadamente según el uso del objeto y lo utiliza para realizar **recolección de basura** por cuenta de referencias (reference-counting).

10.4.4.2 El intérprete de bytecodes. Máquina de pila

Los **métodos** se representan como objetos normales. Los datos de un método son los *bytecodes* que lo definen.

El juego de instrucciones de la máquina está basado en una máquina de pila bastante convencional. Es similar al Código-p. Cuando se llama a un método, la máquina crea una pila de evaluación para el método y una zona de memoria en la que almacena las variables locales.

Al tratarse de gestionar únicamente objetos, el **juego de instrucciones** es muy compacto, simplemente se pueden realizar estas operaciones:

- **Apilar** un objeto en la pila (se apila el puntero al objeto) (push objeto)
- **Almacenar** la cima de la pila en una variable (store into variable)
- **Eliminar** la cima de la pila (pop).
- **Enviar un mensaje** a un objeto tomando los parámetros de la pila (send mensaje).
- **Devolver** la cima de la pila como resultado de la ejecución de un método (ret)
- **Control de flujo de ejecución**. Salto incondicional o condicional (según el valor de la cima de la pila) a una posición dentro del código de bytes. En principio podría realizarse enviando los mensajes correspondientes al objeto método, pero por razones de eficiencia se representa como instrucciones.

Existe un conjunto de punteros globales de la máquina que permiten el acceso a variables globales, variables de clase, etc.

10.4.4.3 Subrutinas primitivas

Una serie de métodos muy utilizados se codifica mediante subrutinas primitivas de la plataforma. En lugar de llamar de manera normal a estos métodos, el intérprete llama a la versión primitiva del mismo, para ganar eficiencia. Cada método se representa en ejecución mediante un objeto. Para detectar cuándo tiene implementación primitiva un método, se marcan de manera especial estos objetos.

10.4.4.4 Crítica

Los principales inconvenientes de la máquina Smalltalk son debidos a su nacimiento como soporte para un lenguaje determinado, además de una cierta mezcla de niveles de abstracción.

Máquina de pila. Mezcla de nivel alto y bajo de abstracción.

La máquina virtual se estructura internamente como una máquina de pila, y esto se refleja en las instrucciones de la máquina. Sufre un problema análogo a la Máquina-p del Pascal. La representación de los métodos en términos de operaciones con una pila reduce el nivel de abstracción de la máquina. A pesar de tener gran uniformidad conceptual en torno a los objetos, se reduce el nivel de abstracción de la OO al tener que descender de nivel para representar las estructuras de un programa OO en términos de una pila. Se mezcla el concepto de alto nivel de llamada a método con las operaciones de bajo nivel de la pila. Esto condiciona la manera de implementar la máquina virtual y puede complicar el uso de la máquina con otros lenguajes.

Específica de Smalltalk. Modelo de objetos distinto del de las metodologías

La máquina está desarrollada para dar soporte específico a Smalltalk y contempla todas las peculiaridades del lenguaje. Sin embargo esto hace difícil o en algunos casos imposible su utilización para dar soporte a otros lenguajes orientados a objeto.

Por otro lado, el modelo de objetos de Smalltalk, a pesar de considerarse un modelo de objetos de los más puros, no es el que se ha impuesto como modelo de objetos estándar de las metodologías. No dispone de herencia múltiple, por ejemplo. Tampoco están presentes excepciones, ni otras características importantes del modelo de objetos que se busca para el sistema integral, como identificadores globales de objetos.

En algunos casos se sacrifica la uniformidad OO pensando en razones de eficiencia, como cuando objetos primitivos como los enteros, etc. se representan directamente en lugar de representarse como objetos normales, con su identificador, etc.

Soporte para lenguaje, no para un sistema completo. No extensible.

La máquina está pensada para dar soporte en el nivel del lenguaje. Aunque dentro del lenguaje el sistema de desarrollo Smalltalk puede considerarse en cierta manera un sistema operativo, no existen previsiones en la misma para dar soporte a las necesidades de un sistema integral, con un sistema operativo y otros elementos adicionales.

Por ejemplo, el puntero de objetos es un identificador interno para la ejecución, no un verdadero identificador de objetos. No hay previsión para la distribución de objetos, ni para una persistencia transparente verdadera[1]. Tampoco existen mecanismos que permitan que las capacidades de la máquina sean extendidas por objetos del usuario, o para interactuar con un sistema operativo, etc.

10.4.4.5 Características interesantes

La máquina de Smalltalk es la primera máquina abstracta para un lenguaje orientado a objetos con éxito comercial. Ha sido una de las razones del éxito (moderado) de Smalltalk y de la consideración de Smalltalk como sistema de referencia de las ventajas de la orientación a objetos. A pesar de que los requisitos de diseño de la época no la hacen adecuada para las necesidades de un sistema integral OO, existen una serie de ideas interesantes a considerar, como las que aparecen a continuación.

Juego reducido de instrucciones

Una máquina orientada a objetos no necesita un juego de instrucciones grande. Cuanto más se acerque este juego de instrucciones a la simplicidad de conceptos de la OO, más sencilla de comprender será la máquina. Esto también permitirá no inclinar el soporte hacia un tipo especial de lenguajes y que la implementación pueda variar de la forma más radical, si es necesario. En el caso de la máquina de Smalltalk se necesitan algunas instrucciones más por el hecho de ser una máquina de pila.

Uso de primitivas de manera transparente

Una manera de aumentar la eficiencia sin sacrificar la uniformidad ni la independencia del código es la utilización de implementaciones primitivas de ciertos elementos de la máquina. Cuando se necesite usar esos elementos mediante el procedimiento normal, en estos casos especiales la máquina de manera transparente accede a una implementación primitiva más rápida. La máquina de Smalltalk lo hace por ejemplo con los métodos más usados. Esto puede extenderse a cualquier otro elemento, pero siempre de manera uniforme que no imponga una dualidad objetos especiales / objetos normales.

[1] Simplemente se guarda una instantánea de la imagen de memoria de la máquina virtual. Para objetos individuales se proporcionan métodos para grabar y escribir su contenido en disco.

Uso de bytecode

La utilización de una representación más compacta del código que conforman los métodos de los objetos puede facilitar la interpretación del mismo y su movimiento. Esto no es estrictamente necesario, simplemente puede ser más conveniente.

10.4.5 JVM. La máquina virtual de Java

Java [AG96] es un lenguaje de programación orientado a objetos, anunciado como un C++ sin sus inconvenientes. Por ejemplo, Java libera al programador de la gestión de memoria, automatizándola mediante un recolector de basura. El gran éxito de Java no es tanto debido al lenguaje como al conjunto de tecnologías que le rodean, la plataforma Java [KJS96]. El elemento fundamental de este éxito es la asociación existente entre el lenguaje y una máquina virtual JVM (*Java Virtual Machine*) creada especialmente para darle soporte. La portabilidad que proporciona la máquina virtual, junto con la coincidencia en el tiempo del lanzamiento de Java y la explosión de la Internet son las razones fundamentales de su éxito. La Internet es una red heterogénea, y el uso de una máquina virtual es ideal en esas circunstancias: el código para la máquina virtual puede moverse por la red, independientemente de cuál sea la plataforma destino.

Aunque la plataforma Java, incluyendo la máquina virtual fue presentada al público después de haberse iniciado este proyecto, se incluye aquí debido a que quizás sea la máquina virtual más popular de todos los tiempos. También con objetivo de analizar sus características y estudiar si pueden incluirse o incluso adoptar la máquina para el proyecto.

Modelo de Java

La máquina virtual de Java [LY97] está totalmente adaptada para soportar el lenguaje Java: lenguaje OO con clases y herencia simple, herencia múltiple de interfaces, enlace dinámico, concurrencia basada en monitores, excepciones, etc.

10.4.5.1 Estructura de máquina de pila

Tecnológicamente, la máquina de Java es heredera directa de la Máquina-p para el Pascal. Su estructura como máquina de pila convencional es muy similar a la Máquina-p. Los operandos de las instrucciones se toman de la pila y el resultado se deja en la pila. La unidad básica de la máquina es la palabra, normalmente de 32 bits. Cada palabra almacena una referencia (una dirección de memoria).

Existen cuatro partes fundamentales en las que se divide la máquina virtual de la plataforma Java:

- **Área de métodos**. Es el equivalente al depósito de código de la Máquina-p. Almacena información acerca de las **clases** Java: código de los **métodos** (*bytecodes*), constantes y métodos de clase e información descriptiva e interna de las clases. A diferencia de la Máquina-p, las clases se pueden cargar de manera dinámica en ejecución cuando se hace referencia a ellas.
- **Montón** (*Heap*). Almacena los objetos (**instancias**). Un recolector de basura incorporado recupera el espacio de los objetos no usados.

  El acceso a los objetos se realiza siempre de manera indirecta a través de una referencia a objeto (*object reference*) de una palabra de tamaño. Estas referencias son similares a identificadores de objetos.
- **Pila de ejecución**. Existe una por cada **hilo** de ejecución. Un planificador incorporado concede tiempo de ejecución a cada hilo. En cada momento se estará ejecutando un

método determinado (el **método actual**). Almacena registros de activación (como la Máquina-p para Pascal) de Java. Cada registro de activación se compone conceptualmente de tres partes:

- **Variables locales** que usa el método actual
- **Pila de operandos** para el cálculo de las operaciones
- **Entorno de ejecución**: información de la ejecución de la llamada actual, como por ejemplo la dirección de retorno, una referencia a la clase del método actual que se utiliza para poder resolver dinámicamente las llamadas a métodos, acceso a constantes de clase, etc. Esto es necesario pues al ser una máquina de pila, la operación de llamada a un método se describe indicando un desplazamiento dentro de la información de la clase, en lugar de indicar directamente el nombre del método.

- **Registros de ejecución de un hilo**, indican la siguiente instrucción a ejecutar y dónde se localizan los tres componentes del registro de activación para el método actual.
  - **Contador de programa** (PC, *Program Counter*), indica la dirección de la siguiente instrucción (*bytecode*) a ejecutar.
  - **Variables locales** (vars), indica dónde empieza la zona de variables locales para el método actual.
  - **Entorno de ejecución** (frame), indica dónde comienza la información sobre el entorno de ejecución del método actual.
  - **Cima de la pila** (optop), indica la posición de la cima de la pila de evaluación.

  Los tres últimos registros apuntan en la pila de ejecución del hilo a las posiciones correspondientes del registro de activación.

10.4.5.2 Tipos de datos básicos

La máquina virtual de Java se hace cargo directo de la gestión dos tipos de datos básicos diferentes:

- **Tipos primitivos**. Los tipos primitivos sobre los que se hacen corresponder los tipos primitivos del lenguaje Java: números enteros, en coma flotante y de doble precisión (int, float, double). Es importante señalar que los tipos primitivos de Java no son objetos.
- **Tipo referencia**. Son punteros a objetos (*reference*). Existen para gestionar los objetos normales. Debido a cómo es el lenguaje Java pueden ser de tres formas: de instancia, de interfaz o de array.

10.4.5.3 Juego de instrucciones

Esta dicotomía entre tipos básicos y objetos se ve reflejada en el juego de instrucciones. Existen instrucciones específicas de cada apartado para cada tipo de datos: Existe una instrucción de suma diferente para cada tipo de datos básico, las de control de flujo también, hay una instrucción específica para retornar un valor de cada tipo básico, etc. Cada instrucción se prefija con la letra inicial del tipo básico. Incluso para cada una de ellas hay hasta 5 variantes adicionales que especifican directamente, por ejemplo, el valor a poner en la pila (ipush0, ipush1, etc.). En general se añaden muchas instrucciones para dar soporte de manera directa (supuestamente más eficiente) ciertos casos específicos que se consideran importantes. Esto incrementa notablemente el juego de instrucciones hasta unas 231.

Gestión de tipos primitivos (sobre 145 instrucciones)

- **Carga y almacenamiento** (cerca de 90 instrucciones). Para la carga de operandos (variables) en la pila (load) y almacenamiento de valores de la pila en variables (store), carga de constantes en la pila y ampliación de acceso a un rango de variables locales mayor.
- **Operaciones aritméticas** (cerca de 40 instrucciones). Para la realización de operaciones aritméticas con los tipos básicos: suma, resta, división, multiplicación, resto, desplazamientos, incremento en uno.
- **Conversión entre tipos básicos** (sobre 15 instrucciones). De cada tipo básico a los otros.

Control de flujo (sobre 30 instrucciones)

Instrucciones para cambiar el flujo secuencial normal de instrucciones dentro de un método.

- **Salto condicional**. En función del resultado de una comparación con cada tipo básico. Existe un abanico mayor de instrucciones con comparaciones con enteros.
- **Salto condicional estructurado**. Permiten establecer tablas de saltos a direcciones y seleccionar uno de estos saltos en función de un desplazamiento o una clave.
- **Salto incondicional**. Salto a una posición de código dentro del método de manera absoluta. Una variante permite la implementación de excepciones.
- **Sincronización**. Implementan la gestión de la concurrencia basada en monitores.

Gestión de la pila de ejecución (sobre 10 instrucciones)

Instrucciones que permiten gestionar la pila de ejecución: extraer un elemento de la pila (con dos variantes), duplicar la cima de la pila (con 7 variantes) e intercambiar los dos elementos superiores de la pila.

Gestión de objetos (sobre 40 instrucciones)

Aunque tanto los objetos individuales como los arrays de objetos son objetos, la máquina los trata por separado. Existen dos conjuntos de instrucciones separados para manipularlos.

- **Creación y manipulación de objetos** (sobre 30 instrucciones)
  - Creación de nuevas instancias (new).
  - Creación de nuevos arrays de objetos. Con variantes específicas para array de tipos básicos (newarray), array de objetos (es decir, arrays de referencias, anewarray) y arrays multidimensionales.
  - Carga de un elemento de un array en la pila. Con variantes en función del tipo de los elementos del array.
  - Almacenamiento de un elemento de la pila en un array. Con las variantes correspondientes.
  - Comprobación de la longitud de un array.
  - Comprobación del tipo al que pertenece un objeto.
  - Acceso de lectura/escritura a los campos de un objeto y de una clase.

- **Invocación de métodos** (sobre 10 instrucciones)
  - Invocación de un método de un objeto (invokevirtual), que se indica mediante un desplazamiento dentro de la información de la clase.
  - Invocación de un método de clase.
  - Invocación especial de métodos (2 variantes).
  - Retorno de un método (return). Variantes en función del tipo de valor de retorno.

10.4.5.4 Características adicionales

Existen otras características adicionales en la plataforma Java. Por ejemplo, dado que uno de los objetivos es la utilización de código que pueda ser descargado de la Internet, existen una serie de restricciones de seguridad que intentan evitar daños malintencionados. Por una parte, el componente que se encarga de cargar dinámicamente las nuevas clases en el área de métodos tiene un verificador de clases que comprueba que el código de la clase cumple una serie de condiciones para que sea seguro (por ejemplo que no intenta acceder fuera de su ámbito). Por otro lado se puede asociar un gestor de seguridad a una determinada aplicación limitándole el acceso a un conjunto de recursos que no pueden causar daños al sistema. Por ejemplo se impide el acceso al sistema de ficheros general del sistema. Esta política de seguridad, denominada Caja de Arena (*Sandbox*), aunque efectiva, es demasiado restrictiva.

10.4.5.5 Crítica

Muchos de los problemas que presenta esta máquina vienen derivados de su nacimiento para soportar un único lenguaje, como en el caso de la máquina de Smalltalk.

Demasiado adaptado al modelo de Java. Modelo OO no adecuado

Muchas de las decisiones de diseño de la máquina, como por ejemplo el juego de instrucciones, están demasiado adaptadas al lenguaje Java. Esto dificulta la utilización de la máquina en otros contextos:

- **Modelo no suficientemente general**, no es el de las metodologías. Al utilizar el modelo de objetos de Java, el soporte para otros lenguajes de uso común se dificulta. Por ejemplo, no existe el concepto de herencia múltiple[1], lo que hace muy difícil el soporte de un lenguaje tan usado como C++.
- **Modelo de objetos pasivo**. El modelo de concurrencia que se utiliza es el de objetos pasivos. Los hilos van viajando y ejecutan los métodos de los objetos manipulando sus datos. Esta estructura de objetos pasivos se adapta muy bien a las características de una máquina de pila. Sin embargo, para un sistema integral, un modelo activo parece una mejor elección.

Falta de uniformidad OO. Dicotomía entre tipos básicos / objetos

Al igual que en el lenguaje Java, existe una dicotomía entre los tipos básicos de la máquina, que no son objetos, y los objetos. Para los tipos básicos existe un conjunto de instrucciones específico, totalmente diferente de las que se usan para los objetos. No existe una uniformidad en la OO, se manejan dos conceptos totalmente diferentes.

[1] Sí existe herencia múltiple de interfaces, pero no de implementación.

Máquina de pila. Interfaz de alto y bajo nivel a la vez

La interfaz de utilización de la máquina es a la vez de alto nivel y de bajo nivel. Por un lado existen instrucciones de muy alto nivel, como la de creación de un objeto, invocación de un método, etc. Por otro lado éstas conviven con instrucciones de un nivel de abstracción muy alejado del anterior, como por ejemplo operaciones que manipulan la pila. Parte de la culpa de esta mezcla de niveles de abstracción es debido al uso de una máquina de pila, con problemas similares a los de la Máquina-p y los de Smalltalk (incluso más agravados).

Incluso se mezclan elementos de bajo nivel en una instrucción de alto nivel como la de invocación de método: el método a invocar se especifica mediante un desplazamiento dentro de los datos de la clase.

Juego instrucciones excesivo y no uniforme

El número de instrucciones es excesivo y no uniforme. La propia arquitectura en máquina de pila produce un número de instrucciones adicional, aunque en el caso de la máquina de Smalltalk se ve que no tiene por qué ser tan grande. En este caso, la dualidad entre tipos básicos y objetos contribuye a esto. Luego se añade la introducción en muchos casos de una versión de cada instrucción por cada tipo básico. También contribuye la existencia de instrucciones específicas para determinados casos con ánimo de aumentar la eficiencia. Sin embargo esto último se hace de manera un tanto arbitraria: en algunos puntos se introducen instrucciones especiales para optimizar a bajo nivel. En otros casos se deja sólo especificado para que la optimización la realice el implementador.

Pérdida de flexibilidad de implementación

Esta falta de uniformidad puede complicar la implementación y la utilización de la máquina. Ciertas optimizaciones están impuestas por la arquitectura en forma de instrucciones especiales, lo que obliga a traducir los programas de una determinada forma. Esto puede limitar la aplicación de la optimización en otros puntos donde no esté definida.

De una manera similar, la arquitectura de pila y la mezcla de niveles de abstracción en la interfaz produce el mismo efecto: por una parte obliga a usar un determinado tipo de correspondencias forzadas entre un lenguaje y la máquina, y por otro lado restringe la gama de implementaciones posibles de la máquina.

El esfuerzo de implementación de la máquina es relativamente grande, debido al elevado número de instrucciones. Esto reduce la posibilidad de realizar muchas implementaciones diferentes de la misma, para experimentar con diferentes aproximaciones de diseño, incluso que difieran radicalmente entre sí.

Una estrategia que aumenta la sencillez es dejar un juego uniforme de instrucciones más reducido. De esta manera se puede hacer que la optimización sea totalmente realizada de manera interna dentro de cada implementación de la máquina y puede experimentarse con muchas implementaciones diferentes, puesto que realizar cada una es menos costoso. Se favorece la portabilidad, ya que al menos una versión sencilla aunque no optimizada de la máquina es fácil de realizar.

No pensado para SO ni para la extensión de la máquina

La máquina está orientada a dar soporte a un lenguaje. No existen muchas previsiones para dar soporte a entornos completos[1], ni para extender o modificar la funcionalidad de la

[1] Aunque por ejemplo existe el proyecto JavaOS [MHM96] para desarrollar alrededor de Java un sistema completo, se centra únicamente en sistemas empotrados monousuario, sin los requerimientos de un sistema multipropósito, como por ejemplo la protección.

máquina adaptándola a las necesidades de un sistema. Por ejemplo, no se puede actuar sobre el recolector de basura o el planificador. Ambos son parte integrante de la máquina y no se pueden modificar.

10.4.5.6 Características interesantes

Lo más interesante de la máquina virtual de Java es la masiva aceptación que ha tenido en el mundo comercial. Esto prueba de nuevo que la utilización de máquinas abstractas está justificada para un proyecto en el que es importante la heterogeneidad y la portabilidad. De la misma manera se reafirma la posibilidad de la implementación en hardware: Sun está planeando la introducción de procesadores basados en el núcleo picojava [Way96] que implementen la máquina virtual de Java (microjava y ultrajava).

Carga dinámica de clases

Desde el punto de vista técnico, un aspecto interesante que aporta la máquina de Java es la carga dinámica de clases. En lugar de cargar inicialmente todas las clases que se necesitan se van cargando a medida que se hace referencia a las mismas. Esto es importante sobre todo en entornos de ejecución dinámica y distribuida en los que no está predeterminado el curso de ejecución de un programa (las referencias que va a hacer) y los objetos pueden moverse de una máquina a otra (con lo que habrá que cargar la clase para acceder a los mismos)

10.4.6 Dis. La máquina virtual del sistema Inferno

Inferno [DPP+96] es un producto de Lucent Technologies que proporciona un entorno uniforme de ejecución para aplicaciones orientadas a red con sistemas heterogéneos. Se basa en la utilización de la máquina virtual Dis [Luc96].

La máquina está pensada para dar soporte a los conceptos básicos sobre los que se define el sistema: proceso y espacio de nombres jerárquico particular de cada proceso para representar cualquier recurso local o remoto. Existe un lenguaje modular, Limbo, desarrollado especialmente para este sistema.

La arquitectura de la máquina es de instrucciones con 3 operandos y del tipo memoria-a-memoria. Las características más importantes son las siguientes:

Existe una fácil equivalencia entre las instrucciones de la máquina y las existentes en procesadores normales. El número de instrucciones es bastante elevado (119).

Gestiona una serie de tipos básicos: byte, palabra (*word*), etc. y otros de más alto nivel como arrays, cadenas de caracteres, procesos y canales de comunicación entre procesos. Cada tipo de datos tiene sus propias instrucciones particulares para crear, operaciones aritméticas, etc.

Gestiona un modelo de hilos concurrentes basado en el paradigma de la comunicación de procesos secuenciales [Hoa78] (CSP, *Communicating Sequential Processes*). Existen instrucciones que realizan la comunicación mediante canales, al estilo del lenguaje paralelo Occam [INM88]: enviar por un canal, recibir por un canal, recepción alternativa entre canales, etc. (send, receive, alt, etc.). También para crear procesos, etc.

Cada hilo de ejecución tiene una pila de registros de activación, y accede a una memoria del módulo que se ejecuta, indicados por los punteros de registro de activación (fp, *frame pointer*) y de datos del módulo (mp, *module pointer*), respectivamente. Todo el direccionamiento es relativo respecto a estos punteros. En otra zona de la memoria se almacena el segmento de código ejecutable.

Existe una recolector de basura integrado en la máquina que utiliza dos algoritmos diferentes: uno por cuenta de referencias y otro mediante marca y barrido.

10.4.6.1 Crítica

Se trata de un sistema que no utiliza la orientación a objetos, si no un modelo más procedimental. Existe soporte específico para los tipos básicos, pero no para tipos creados por el usuario. El número de instrucciones es elevado. Por último, ciertas características de la máquina no se pueden modificar, como la recolección de basura, al estar integradas en la misma.

10.4.6.2 Características interesantes

A pesar de no ser OO, y haber aparecido después de iniciado el proyecto, se ha revisado esta máquina puesto que es muy relevante para el proyecto en otro sentido. Es una máquina diseñada para dar soporte a un sistema distribuido en un entorno heterogéneo. Aunque en cierta manera está orientada a un lenguaje determinado, el objetivo no es únicamente soportar el lenguaje, si no todo un sistema operativo completo. Es un sistema comercial que demuestra que es posible basar en una máquina abstracta el soporte no sólo para un lenguaje específico, si no para un entorno o sistema operativo completo.

Desde el punto de vista técnico, es interesante observar que el soporte de concurrencia se ha considerado desde el principio en el sistema. Se debe tener en cuenta la posibilidad de incluir en una máquina soporte directo, quizás como Dis en forma de instrucciones primitivas, para la concurrencia. En cualquier caso, este apartado, por su importancia se deja a estudio del subsistema del SO dedicado a la concurrencia.

10.5 Inconvenientes de las máquinas abstractas revisadas en general

En esta sección se resumen los principales inconvenientes de las máquinas abstractas, comunes a la mayoría de las mismas. Dado el enfoque hacia la OO, se hace más hincapié en las máquinas más orientadas a este paradigma.

10.5.1 Máquinas de pila. Imposición de una estructura interna de bajo nivel

La mayoría de las máquinas abstractas son máquinas de pila. En general, la utilización de una estructura interna de bajo nivel como la pila impone una serie de condicionantes para el uso de la máquina. Por una parte eso se refleja en la necesidad de añadir instrucciones para el manejo de la estructura de bajo nivel. Por otro lado, se dificulta la escritura de compiladores de lenguajes, al tener que descender a un nivel de abstracción más bajo. Por último se limita la variedad de implementación en la propia implementación de la máquina abstracta. Las posibilidades de elección (y de innovación) en las estructuras de la implementación se restringen mucho al tener que utilizar obligatoriamente como estructura de bajo nivel una pila.

10.5.2 Falta de uniformidad en la OO. Mezcla de niveles de abstracción

En muchos casos existe una mezcla de paradigmas. Por ejemplo, la máquina de Java distingue entre tipos de datos básicos y objetos. Para gestionar los tipos de datos básicos se utilizan instrucciones específicas para los mismos y para los objetos la construcción de invocación a métodos. El paradigma OO se mezcla con otros conceptos. Además, se mezcla el nivel de abstracción de la OO, más elevado, con un nivel más bajo de operaciones elementales con tipos de datos básicos.

Esta mezcla de niveles de abstracción también se observa en otro sentido en la máquina de Smalltalk, que no sufre el problema anterior. La utilización ya mencionada de instrucciones para una pila hace que junto con construcciones de alto nivel OO (llamada a métodos) existan construcciones de bajo nivel (manejo de la pila).

10.5.3 Falta de soporte para el modelo OO completo

Aunque el soporte para la OO en las máquinas OO es más amplio que en las máquinas hardware, se soporta el modelo de objetos de un lenguaje específico, como Smalltalk o Java. Estos modelos de objetos no incorporan todas las propiedades que se necesitan para el modelo de objetos de un sistema integral. Ausencias notables son la falta de soporte para la herencia, y sobre todo la inexistencia de relaciones genéricas de asociación entre los objetos.

10.5.4 Inflexibilidad en la incorporación de ciertas características

A pesar de ser máquinas implementadas en principio en software, existe una imposibilidad de modificar ciertas características adicionales a la OO de las máquinas. Por ejemplo, el uso del recolector de basura que incorporan algunas máquinas es obligatorio, forma parte indisoluble de las mismas. No puede ser sustituido por otro que implemente un algoritmo diferente o incluso simplemente no utilizarlo. Todo lo más que se puede hacer es elegir entre varias opciones disponibles o desactivarlo. Más interesante sería que este tipo de elementos fueran extensiones de la máquina que pudieran ser utilizadas o no, y fueran fáciles de sustituir por implementaciones mejores, etc., en lugar de ser parte integrante de la máquina.

10.6 Ideas tomadas de las máquinas revisadas

Ninguna de las máquinas revisadas reúne todos los requisitos necesarios para dar soporte a un sistema integral como el que se propone. Existen, sin embargo, diferentes buenas ideas presentes en alguna o comunes a algunas de las máquinas revisadas, o necesarias para evitar problemas detectados en las mismas. A continuación se resumen las ideas más importantes que se tendrán en cuenta en el diseño o la implementación de una nueva máquina abstracta orientada a objetos para dar soporte a un sistema integral.

10.6.1 Interfaz de instrucciones OO de la máquina de alto nivel, no relacionada con estructuras de implementación

La interfaz de instrucciones de la máquina es lo único que usarán las aplicaciones. Es importante que esta interfaz no imponga determinadas estructuras internas de bajo nivel de implementación. De esta manera, la implementación de la máquina puede hacerse de la mejor manera posible e incluso cambiarse de manera radical, favoreciendo la experimentación. Cada implementación puede utilizar las optimizaciones y estructuras internas que estime convenientes, sin ninguna restricción impuesta por la existencia de estructuras de bajo nivel en la interfaz. Las aplicaciones, sin embargo no necesitarán modificaciones. Es decir, la interfaz de la máquina se circunscribe al lenguaje que formen las instrucciones de la máquina, que puede describirse en forma simbólica de lenguaje ensamblador. La unidad de descripción de este lenguaje (descripción de comportamiento) será el elemento básico que describe comportamiento en un sistema orientado a objetos: la clase.

10.6.2 Interfaz OO pura con juego reducido de instrucciones

También es importante que la interfaz de la máquina sea totalmente OO, sin utilizar otros conceptos adicionales. Se trata de mantener la uniformidad OO también en las instrucciones de la máquina. Además, se evita la proliferación de instrucciones específicas para

manipulación de estructuras u objetos especiales. Una interfaz pura OO favorece la independencia de las estructuras internas de implementación y necesita pocas instrucciones. Además, es más sencillo desarrollar una máquina con pocas instrucciones, al menos una implementación rápida sin optimizaciones. Esto permite portar la máquina rápidamente a nuevas plataformas. Posteriormente se pueden realizar implementaciones cada vez más eficientes si se desea.

10.6.3 Objetos homogéneos – Uniformidad OO

La distinción entre diferentes tipos de objetos en el sistema (objetos del hardware, objetos de usuario, etc.) lleva a una falta de uniformidad innecesaria. Además produce en muchos casos la introducción de instrucciones específicas para objetos especiales. Todos los objetos de la máquina tendrán la misma consideración.

10.6.4 Uso de primitivas de manera transparente

Para aumentar la eficiencia sin sacrificar la uniformidad ni la independencia del código se podrán utilizar implementaciones primitivas de ciertos elementos. Ejemplos de esto pueden ser las clases y objetos básicos del sistema. A todos los efectos estos objetos serán iguales al resto, sin embargo, cuando se acceda a ellos (sin ninguna distinción externa) la implementación de la máquina accederá a implementaciones primitivas de los mismos. Es importante recalcar que esto no hace diferentes en cuanto a su uso y comportamiento a estos objetos. Esto permite que cada implementación pueda utilizar implementaciones primitivas de los elementos que desee, sin afectar a las aplicaciones.

10.6.5 Extensión de la máquina mediante el sistema operativo

La máquina dispondrá de las características fundamentales del modelo de objetos. Sin embargo, características adicionales serán realizadas por el sistema operativo (mediante un conjunto de objetos normales), que extenderá transparentemente las propiedades de los objetos de la máquina (por ejemplo la persistencia, o recolectores de basura). Es necesario, pues, que la máquina disponga de una serie de mecanismos que permitan al sistema operativo desarrollar esta tarea.

10.6.6 Direccionamiento de objetos separado del almacenamiento físico

Es necesario utilizar un identificador para cada objeto del sistema. Este identificador no debe estar relacionado de ninguna manera con la posición o la manera de almacenamiento del objeto. Es un aspecto más de la interfaz independiente de las estructuras internas, en la interfaz se usarán exclusivamente identificadores de objetos para direccionar los objetos, nunca se podrán ver estructuras internas de almacenamiento.

10.6.7 Carga dinámica de clases

Es apropiado que la máquina disponga de un mecanismo que permita cargar dinámicamente el código de las clases. En un entorno de ejecución dinámico, heterogéneo y distribuido, en el que los objetos (con sus clases asociadas) pueden viajar por los nodos de la red, no siempre se conocen de antemano las clases a las que se va a hacer referencia. Este mecanismo permite cargar el código de una clase sobre la marcha, justo en el momento en que se haga referencia a la misma.

10.6.8 Uso de representación compacta del código (bytecode)

El código de las clases (unidad de descripción de los programas del lenguaje de la máquina) puede ser representado de manera compacta. En lugar de proporcionar a la máquina

el código en forma de ensamblador, se puede utilizar una representación del mismo mediante *bytecodes* u otra estructura de representación más compacta. Aunque conveniente para reducir el tamaño físico de los programas, esto no es estrictamente necesario.

10.6.9 Protección de objetos ligada al direccionamiento

Aunque esto será desarrollado por el apartado de seguridad del sistema operativo, la manera que parece encaja mejor con este tipo de sistemas es asociar la protección al direccionamiento de los objetos.

10.7 Resumen de características de las máquinas revisadas

Nombre / Año	IBM System/38 (1978)	Intel iAPX 432 (1981)	Rekursiv (1987)	MUSHROOM (1987)
Característica principal	Independencia interfaz / implementación	Uniformidad entre procesador, SO y aplicaciones	Interfaz instrucciones micro-programable	Mejoras en sistemas convencionales para la OO
Soporte objetos	Básico	Sí	Sí	Básico
Semántica modelo objetos	Zona memoria	Basado en objetos – zona memoria	Basado en objetos	Segmentos 256 bytes
Nivel juego instrucciones	Medio	Medio	Alto	-
Tamaño juego instrucciones	Medio	Medio	Grande	-
Extensible	No	Sí	No	
Uniformidad OO				-
Homogeneidad objetos	No, objetos básicos / genéricos usuario	No, objetos básicos / genéricos de usuario	Sí	
Interfaz pura OO	No, sólo para objetos básicos	No, instrucciones especiales para objetos básicos	Según micro-programación	
Identificador objetos	Sí	Sí	Sí	Sí
Protección uniforme	Capacidades	Capacidades	No	-
Otras características			Recolección de basura y persistencia transparente integradas	

Tabla 10.1 Resumen de características de máquinas reales

Nombre / Año	ANDF (1993)	Máquina-p (1976)	Máquina Smalltalk (1983)	JVM (1996)	Dis (1996)
Característica principal	Distribución universal de software	Soporte paradigma procedimental	Uniformidad OO	Carga dinámica de clases	Soporte programación modular
Estructura	-	Máquina de pila	Máquina de pila	Máquina de pila	Máquina de memoria
Soporte objetos	No	No (procedimientos)	Sí	Sí	No (programación modular)
Semántica modelo objetos	-	-	Completa (Smalltalk)	Completa (Java)	-
Nivel juego instrucciones	Bajo	Alto nivel (proc.) y bajo nivel (pila y tipos básicos)	Alto nivel	Alto nivel (objetos) y bajo nivel (pila y tipos básicos)	Medio
Tamaño juego instrucciones			Pequeño	Grande	Grande
Extensible	No	No	No	No	No
Uniformidad OO	-	-			-
Homogeneidad objetos			Sí	No (objetos y tipos básicos)	
Interfaz pura OO			Sí	No	
Identificador objetos			Sí (uso interno)	Sí (uso interno)	
Protección uniforme		No	No	No	Sí
Otras características			Recolección de basura integrada	Recolección de basura y planificación hilos integradas. Carga dinámica de clases	Recolección de basura y planificación hilos integradas.

Tabla 10.2 Resumen de características de máquinas abstractas

Capítulo 11

ARQUITECTURA DE REFERENCIA DE UNA MÁQUINA ABSTRACTA PARA SOPORTE DE SISTEMAS INTEGRALES

En el capítulo 10 se revisaron diferentes máquinas abstractas. Sin embargo ninguna de ellas reúne todos los requisitos necesarios para dar soporte a un sistema integral orientado a objetos (SIOO). En este capítulo se describe una arquitectura de referencia de una máquina abstracta orientada a objetos con las propiedades necesarias para dar soporte a un sistema integral orientado a objetos.

11.1 Propiedades fundamentales de una máquina abstracta para un SIOO

Las características fundamentales de la misma se derivan de los objetivos fundamentales de un SIOO, así como de las propiedades interesantes de las máquinas revisadas, o para evitar problemas detectados en las mismas.

- Modelo único de objetos, que implementará la máquina
- Identificador único de objetos
- Uniformidad en la OO, único nivel de abstracción
- Interfaz de alto nivel, independiente de estructuras de bajo nivel
- Juego de instrucciones reducido
- Flexibilidad (extensión) (se tratará con detalle en el capítulo 19)

11.2 Estructura de referencia

A continuación se describen brevemente los elementos básicos en que se puede dividir conceptualmente la arquitectura. El objetivo de esta estructura es facilitar la comprensión del funcionamiento de la máquina, no necesariamente indica que estos elementos existan como tales entidades internamente en una implementación de la máquina.

Los elementos básicos que debe gestionar la arquitectura son:

- **Clases**, que se pueden ver como agrupadas en un **área de clases**. Las clases contienen toda la información descriptiva acerca de las mismas.
- **Instancias**, agrupadas en un **área de instancias**. Donde se encuentran las instancias (objetos) de las clases definidas en el sistema. Cada objeto será instancia de una clase determinada.
- **Referencias**, en un **área de referencias**. Se utilizan para realizar la invocación de métodos sobre los objetos y acceder a los objetos. Son la única manera de acceder a un

objeto (no se utilizan direcciones físicas). Las referencias contienen el identificador del objeto al que apuntan (al que hacen referencia). Para la comprobación de tipos, cada referencia será de un tipo determinado (de una clase).

- **Referencias del sistema**. Un conjunto de referencias especiales que pueden ser usadas por la máquina.
- **Jerarquía de clases básicas**. Existirán una serie de clases básicas en el sistema que siempre estarán disponibles. Serán las clases básicas del modelo único de objetos del sistema.

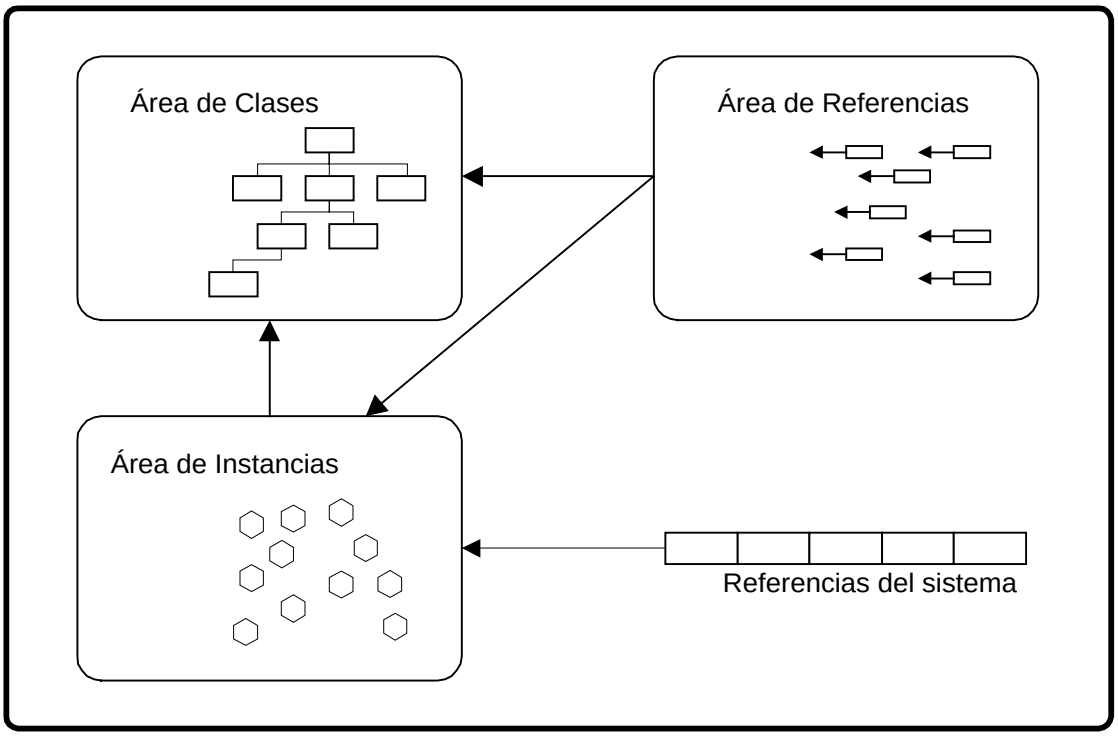


Figura 11.1 Estructura de referencia de una máquina abstracta para el sistema integral.

11.3 Juego de instrucciones

El juego de instrucciones OO de alto nivel deberá ser independiente de estructuras internas de implementación. Para ello se describirá en términos de un lenguaje ensamblador. La unidad de descripción en un SIOO son las clases. Por tanto el lenguaje ensamblador permitirá describir una clase mediante las instrucciones:

11.3.1 Instrucciones declarativas

Esta arquitectura almacena la descripción de las clases, así que puede considerarse que existen instrucciones cuyo resultado es la descripción de la información de una clase:

- **Nombre de la clase**
- **Relaciones de herencia**
- **Relaciones de agregación**
- **Relaciones de asociación**
- **Métodos de la clase**, con los parámetros y referencias locales que utilicen.

11.3.2 Instrucciones de comportamiento

Que permitan definir el comportamiento de la clase. Es decir, el comportamiento de sus métodos. Describirán el código que compone el cuerpo de cada método:

- **Invocación de método** a través de una referencia. Son en esencia la única operación que se puede realizar en un método.
- **Retorno de un método**.
- **Control de flujo**. Instrucciones para controlar el flujo de ejecución de un método: saltos, saltos condicionales, etc.
- **Excepciones**. Instrucciones de control de flujo para gestionar las excepciones
- **Instrucciones para trabajar con los objetos** a través de las referencias:
- **Creación y borrado de objetos** a través de una referencia
- Etc.

11.4 Ventajas del uso de una máquina abstracta

Con la utilización de una máquina abstracta que siga la arquitectura de referencia anterior se consiguen una serie de ventajas:

11.4.1 Portabilidad y heterogeneidad

La utilización de una máquina abstracta dota de portabilidad al sistema. El juego de instrucciones de alto nivel puede ejecutarse en cualquier plataforma donde este disponible la máquina abstracta. Por tanto, los programas escritos para máquina abstracta son portables sin modificación a cualquier plataforma. Basta con desarrollar una versión (emulador o simulador) de la máquina para ejecutar cualquier código de la misma sin modificación: el código es portable y entre plataformas heterogéneas. Como todo el resto del sistema estará escrito para esta máquina abstracta, el sistema integral completo es portable sin necesidad de recompilar ni adaptar nada (únicamente la máquina abstracta).

11.4.2 Facilidad de comprensión

La economía de conceptos, con un juego de instrucciones reducido basado únicamente en la OO facilita la comprensión del sistema. Está al alcance de muchos usuarios comprender no sólo los elementos en el desarrollo, si no también la arquitectura de la máquina subyacente. Esto se facilita aún más al usarse los mismos conceptos de la OO que en la metodología de desarrollo OO.

11.4.3 Facilidad de desarrollo

Al disponer de un nivel de abstracción elevado y un juego de instrucciones reducido, es muy sencillo desarrollar todos los elementos de un sistema integral orientado a objetos, por ejemplo:

11.4.3.1 Compiladores de lenguajes

Es muy sencillo desarrollar compiladores de nuevos lenguajes OO o lenguajes ya existentes. La diferencia semántica que debe salvar un compilador entre los conceptos del lenguaje y los de la máquina es muy reducida, pues el alto nivel de abstracción OO de la máquina ya está muy cercano al de los lenguajes. El esfuerzo para desarrollar compiladores se reduce.

11.4.3.2 Implementación de la máquina

En lo que concierne a la implementación de la propia máquina, también se obtienen una serie de ventajas.

Esfuerzo de desarrollo reducido

El juego de instrucciones reducido hace que el desarrollo de un simulador de la máquina abstracta sea muy sencillo. No hay que programar código para un número muy grande de instrucciones, con lo que el tiempo necesario y la probabilidad de errores disminuyen.

Rapidez de desarrollo

Además, al ser la interfaz de la máquina independiente de estructuras internas, se puede elegir la manera interna de implementarla más conveniente. Para desarrollar rápidamente una implementación se pueden utilizar estructuras internas más sencillas aunque menos eficientes.

Facilidad de experimentación

Todo ello facilita la experimentación. Se puede desarrollar rápidamente una máquina para una nueva plataforma para hacer funcionar el sistema. Posteriormente, debido a la independencia de la interfaz, se puede experimentar con mejoras internas a la máquina: optimizaciones, nuevas estructuras, etc. sin necesidad de modificaciones en las aplicaciones.

11.4.4 Buena plataforma de investigación

Todas las ventajas anteriores la constituyen en la base para una buena plataforma de investigación. La facilidad de comprensión y desarrollo, y la portabilidad y heterogeneidad permitirán que más personas puedan acceder al sistema sobre cualquier equipo y utilizarlo como base para la investigación en diferentes áreas de las tecnologías OO: el ejemplo anterior de lenguajes OO, bases de datos, etc. Esto se aplica también a la propia máquina en sí, cuya estructura permite una fácil experimentación con diferentes implementaciones de la misma.

11.5 Minimización del problema del rendimiento de las máquinas abstractas

Como inconveniente de la utilización de máquinas abstractas se cita comúnmente el escaso rendimiento de las mismas [SS96]. Se manejan cifras que otorgan a los intérpretes una velocidad entre uno y dos órdenes de magnitud más lenta que el código compilado [May87]. Por ejemplo, ciertos intérpretes de Java suelen ejecutar los programas a un 10% de la velocidad de un programa en C equivalente [Way96]. En muchos casos las diferencias muy exageradas son en casos extremos o en comparaciones viciadas de programas OO que por su estructura no se pueden comparar con otros en lenguajes convencionales como C.

En cualquier caso, la propia naturaleza de una máquina abstracta necesita la utilización de un programa para simularla, con lo que el rendimiento tiene que ser menor que si se usase el hardware directamente.

Sin embargo, existen una serie de razones que hacen que este problema del rendimiento no sea tan grave, e incluso llegue a no tener importancia:

11.5.1 Compromiso entre velocidad y conveniencia aceptado por los usuarios

El simple rendimiento no es el único parámetro que debe ser tenido en cuenta en un sistema. Lo verdaderamente importante es la percepción que tengan los usuarios de la utilidad del sistema, que es función del esfuerzo de programación, la funcionalidad de las aplicaciones, y el rendimiento conseguido. El éxito que ha alcanzado la plataforma Java

[KJS96] está basado en la utilización de una máquina abstracta. Esto demuestra que el compromiso entre el rendimiento y la conveniencia de los beneficios derivados del uso de una máquina abstracta ya es aceptado por los usuarios con implementaciones sencillas de una máquina abstracta.

11.5.2 Mejoras en el rendimiento

Existen una serie de áreas con las que se puede mejorar el rendimiento de las máquinas abstractas, reduciendo aún más el inconveniente de su (aparente) pobre rendimiento.

11.5.2.1 Mejoras en el hardware

La tendencia exponencial del aumento de rendimiento del hardware, junto con la disminución de su precio se ha utilizado históricamente en la informática para elevar el nivel de abstracción [BPF+97]. Las máquinas abstractas son una continuación de esta tendencia, como lo fue el paso del ensamblador a los lenguajes de alto nivel. La potencia adicional se destina a elevar el nivel de abstracción (utilizar una máquina abstracta), que hace que los proyectos sean más baratos de desarrollar (hay una relación no lineal entre el nivel de abstracción y el coste de un proyecto). Los beneficios de un mayor nivel de abstracción compensan la pérdida de rendimiento.

Por otro lado, si con los procesadores convencionales actuales e implementaciones sencillas de máquinas abstractas se ha conseguido una gran aceptación, el aumento de potencia del hardware no hará más que minimizar aún más el problema aparente del rendimiento.

11.5.2.2 Optimizaciones en la implementación de las máquinas. Compilación dinámica (justo a tiempo)

La implementación de una máquina abstracta puede optimizarse para que la pérdida de rendimiento sea la menor posible. Una técnica de optimización es la compilación dinámica o justo a tiempo (JIT, *Just In Time*). Se trata de optimizar la interpretación del juego de instrucciones de la máquina. En lugar de interpretarlas una a una, se realiza una compilación a instrucciones nativas (del procesador convencional) del código de los métodos en el momento de acceso inicial de los mismos (justo a tiempo) [ALL+96]. Los siguientes accesos a ese método no son interpretados de manera lenta, si no que acceden directamente al código previamente compilado, sin pérdida de velocidad. Este código nativo compilado puede ir siendo optimizado aún más en cada llamada adicional (generación incremental de código) [HU94].

Ciertas implementaciones de máquinas abstractas que utilizan esta técnica han resultado sólo de 1.7 a 2.4 veces más lentas que un código C++ equivalente optimizado [Höl95].

Otro ejemplo de la posibilidad de optimización en la implementación de máquinas abstractas se comprueba en el producto Virtual PC de la compañía Connectix Corporation. Virtual PC es una aplicación Macintosh que emula un ordenador PC completo por software sobre una plataforma PowerPC-Mac. Se alcanzan relaciones de 3 instrucciones PowerPC por cada instrucción Pentium emulada y de 5 a 9 instrucciones PowerPC por cada 3 instrucciones Pentium [Tro97].

11.5.2.3 Implementación en hardware

En aquellos casos en que no sea aceptable la pequeña pérdida de rendimiento de una máquina optimizada, se puede recurrir a la implementación de la máquina en hardware. Esta

implementación en hardware especializado ofrecería un rendimiento superior al de cualquier implementación software [Way96].

11.6 Resumen

Las propiedades fundamentales que debe tener una máquina abstracta que de soporte a un SIOO son el modelo único de objetos que implementará, con identificador único de objetos, la uniformidad en la OO, una interfaz de alto nivel con un juego de instrucciones reducido y la flexibilidad. Una estructura de referencia para este tipo de máquina abstracta se compone de cuatro elementos fundamentales: áreas para las clases, instancias, referencias para los objetos; referencias del sistema y jerarquía de clases básicas. El juego de instrucciones permitirá describir las clases (herencia, agregación, asociación y métodos) y el comportamiento de los métodos, con instrucciones de control de flujo y excepciones, creación y borrado de objetos e invocación de métodos.

Las ventajas del uso de una máquina abstracta como esta son básicamente la portabilidad y la heterogeneidad, y la facilidad de comprensión y desarrollo, que la hacen muy adecuada como plataforma de investigación en las tecnologías OO. El inconveniente de la pérdida de rendimiento por el uso de una máquina abstracta se ve contrarrestado por la disposición de los usuarios a aceptar esa pérdida de rendimiento a cambio de los beneficios que ofrece una máquina abstracta. Por otro lado, mejoras en el hardware y optimizaciones en la implementación de las máquinas minimizan aún más este problema.

Capítulo 12
LA MÁQUINA ABSTRACTA CARBAYONIA

En este capítulo se describe la máquina abstracta Carbayonia. Carbayonia es una máquina abstracta orientada a objetos que sigue la estructura de referencia marcada en el capítulo 11. Parte de la descripción de la máquina está adaptada de la documentación del primer prototipo de la misma, desarrollado como proyecto fin de carrera de la Escuela Técnica Superior de Ingenieros Informáticos de la Universidad de Oviedo [Izq96].

La máquina es una máquina abstracta orientada a objetos pura que implementa el modelo único de objetos del sistema. Todos los objetos tienen un identificador que se usa para operar con ellos exclusivamente a través de referencias. Las referencias, como los propios objetos, tienen un tipo asociado y tienen que crearse y destruirse. En Carbayonia todo se hace a través de referencias: las instrucciones tienen referencias como operandos; los métodos de las clases tienen referencias como parámetros y el valor de retorno es una referencia.

Todos los elementos aquí descritos pertenecen a una primera versión de la máquina. Es posible que ciertas partes sufran cambios menores a medida que se adquiera más experiencia con el desarrollo de aplicaciones para la máquina.

12.1 Estructura

Para comprender la máquina Carbayonia se utilizan las tres áreas de la máquina de referencia. Un área se podría considerar como una zona o bloque de memoria de un microprocesador tradicional. Pero en Carbayonia no se trabaja nunca con direcciones físicas de memoria, si no que cada área se puede considerar a su vez como un objeto el cual se encarga de la gestión de sus datos, y al que se le envía mensajes para que los cree o los libere.

A continuación se expone una breve descripción de las áreas que componen Carbayonia, comparándola para una mejor comprensión con arquitecturas convencionales como las de Intel x86. Sus funciones se verán mas en detalle en la descripción de las instrucciones.

12.1.1 Área de Clases

En éste área se guarda la descripción de cada clase. Esta información está compuesta por los métodos que tiene, qué variables miembro la componen, de quién deriva, etc. Esta información es fundamental para conseguir propiedades como la comprobación de tipos en tiempo de ejecución (RTTI, *Run Time Type Information*), la invocación de métodos con polimorfismo, etc.

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

12.1.2 Área de Instancias

Aquí es donde realmente se almacenan los objetos (instancias de las clases). Cuando se crea un objeto se deposita en éste área, y cuando éste se destruye se elimina de aquí. Se relaciona con el área de clases puesto que cada objeto es instancia de una clase determinada. Así desde un objeto se puede acceder a la información de la clase a la que pertenece.

Las instancias son identificadas de forma única con un número que asignará la máquina en su creación. La forma única por la que se puede acceder a una instancia es mediante una referencia que posee como identificador el mismo que la instancia. Se mantiene el principio de encapsulamiento. La única forma de acceder a una instancia mediante una referencia es invocando los métodos de la instancia.

12.1.3 Área de Referencias

Para operar sobre un objeto necesitamos antes una referencia al mismo. En éste área es donde se almacenan dichas referencias. El área de referencias se relaciona con el área de clases (ya que cada referencia tiene un tipo o clase asociado) y con el área de instancias (ya que apuntan a un objeto de la misma). Una referencia se dirá que está *libre* si no apunta a ningún objeto. Las referencias son la única manera de trabajar con los objetos[1].

12.1.4 Referencias del Sistema

Son una serie de referencias que están de manera permanente en Carbayonia y que tienen funciones específicas dentro del sistema. En éste momento simplemente se dará una breve descripción, ya que cada una se explicará en el apartado apropiado.

- **this**: Apunta al objeto con el que se invocó el método en ejecución.
- **exc**: Apunta al objeto que se lanza en una excepción.
- **rr** (*return reference*): Referencia donde los métodos dejan el valor de retorno.

12.2 Descripción del lenguaje Carbayón: Juego de instrucciones

El juego de instrucciones de la máquina se describirá en términos del lenguaje ensamblador asociado al mismo. Este lenguaje se denomina Carbayón y será la interfaz de las aplicaciones con la máquina. En cualquier caso, existe la posibilidad de definir una representación compacta de bajo nivel (*bytecode*) de este lenguaje que sea la que realmente se entregue a la máquina. Buscando facilitar la difusión internacional, en el lenguaje se utiliza el idioma inglés. Para facilitar la comprensión de las características del lenguaje se harán comparaciones con el lenguaje orientado a objetos C++ en los lugares adecuados.

En el apéndice B se encuentran algunos ejemplos de programación en lenguaje Carbayón.

Una definición más formal del lenguaje mediante una gramática tipo BCNF aparece en el apéndice H.

La relación de los posibles errores (excepciones) que pueden aparecer en la ejecución de los programas por la máquina está en el apéndice G

12.2.1 Convenio de representación

El código del lenguaje se representa con un tipo de letra especial.

[1] Por tanto, aunque en algunos casos se mencionen los objetos directamente, como por ejemplo “se devuelve un objeto de tipo cadena” se entiende siempre que es una referencia al objeto.

Las palabras en negrita denotan **palabras reservadas**.

Las palabras entre símbolos < y > denotan una <cadena de texto>, como el nombre de una clase.

Las llaves {} denotan repetición del elemento que encierran (una lista separada por comas o por punto y coma).

Los corchetes [] denotan opcionalidad del elemento que encierran.

Pueden introducirse comentarios mediante el carácter "/". Todos los caracteres que se encuentren desde ese carácter hasta el fin de línea se ignorarán.

12.3 Instrucciones declarativas: Descripción de las clases

La unidad básica declarativa de la máquina es la clase, y será la unidad mínima de trabajo que las aplicaciones comuniquen a la máquina. Las clases del modelo único tienen una serie de propiedades que hay que describir: nombre, relaciones de herencia o generalización (*isa*), relaciones de agregación (*aggregation*), relaciones de asociación genéricas (*association*) y métodos de la clase (*methods*). En Carbayonia, esto se describe como:

```
Class <Nombre>
Isa {Clase}
Aggregation {Nombre: Clase;}
Association {Nombre: Clase;}
Methods
  {<Nombre> ([{Clase}])[:Clase] <cuerpo>}
EndClass
```

12.3.1 Class (clase)

El primer elemento permite dar un nombre a la clase. Este nombre deberá ser diferente del de las otras clases y se almacenará, junto con el resto de la información que le siga, en el área de clases.

12.3.2 Isa (herencia)

En ésta parte se enumeran, separadas por comas, todas las clases de las que hereda la clase que se está definiendo (herencia múltiple). Posteriormente se comentará cómo se tratan los conflictos de variables miembro y métodos con el mismo nombre.

12.3.3 Aggregation (agregación)

Aquí se enumeran los objetos que pertenecen a la clase indicando el nombre que se les da a cada uno (en realidad es el nombre de la referencia a través de la que se accederá a los mismos). La clase a la que pertenece cada uno de los objetos se indica poniendo el nombre de la clase separado por dos puntos del nombre del objeto. Se mantiene la semántica de la agregación. Los objetos agregados se crean automáticamente cuando se crea el objeto que los contiene y se destruyen cuando éste se destruye. Además, no se pueden eliminar individualmente, sólo a través de la eliminación del contenedor. Las variables miembro de C++ son parecidas a estos objetos, aunque en C++ no se mantiene la semántica de la agregación.

12.3.4 Association (asociación)

En este lugar se declaran los objetos que participan de una relación de asociación con la clase que se está definiendo. A diferencia de los agregados, dado que son relaciones

genéricas, estos objetos no se crean automáticamente al crear el objeto ni se destruyen al borrar éste. El equivalente en C++ sería declarar un puntero al tipo deseado, cuya gestión recae en el programador (asignarle un objeto, apuntar a otro objeto distinto si la relación se traspasa a otro individuo, destruirlo si es que es nuestra labor hacerlo, etc.)

Por lo tanto, con los miembros agregados puede pensarse que se guarda una instancia del objeto y con los asociados se guarda un puntero al objeto. En Carbayonia todo se hace a través de referencias por lo que para los agregados se crea una instancia a la cual apunta la referencia, y para los asociados la referencia está libre (es responsabilidad del programador hacer que apunte a un objeto creado previamente).

El conjunto de los agregados y relaciones de un objeto es el equivalente al concepto de variables o variables miembro de un objeto de otros lenguajes como C++. También se pueden llamar atributos o propiedades del objeto.

12.3.5 Methods (declaración de los métodos de la clase)

En el siguiente apartado se tratará la definición del cuerpo de los mismos. La declaración consiste en un nombre de método seguido de una serie de parámetros entre paréntesis. Cada parámetro se identifica mediante una clase a la que pertenece. Al cierre de los paréntesis se pone el tipo del valor de retorno si es que existe.

12.4 Características de las clases Carbayonia

A continuación se describe brevemente las características propias de las clases Carbayonia.

12.4.1 Herencia virtual

Un aspecto a destacar es que toda derivación es virtual. Al contrario que en C++, no se copian simplemente en la clase derivada los datos de las superclases. Al retener toda la semántica del modelo de objetos en tiempo de ejecución, simplemente se mantiene la información de la herencia entre clases. Es decir, en la estructura de herencias tipo como la de la figura, la clase D sólo tiene una instancia de la clase A. Se mantiene sincronizada respecto a los cambios que se puedan hacer desde B y desde C. A la hora de crear instancias de una clase, se repite la misma estructura.

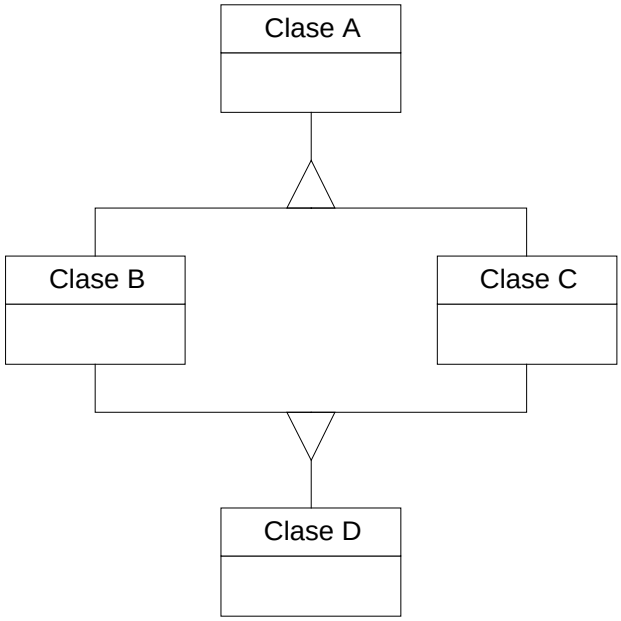


Figura 12.1 Jerarquía de herencia múltiple con ancestro compartido

12.4.2 Herencia múltiple. Calificación de métodos

El problema de la duplicidad de métodos y variables en la herencia múltiple se soluciona dando prioridad a la superclase que aparece antes en la declaración de la clase. Si en la figura anterior se supone que tanto la clase B como la clase C tienen un método llamado M, y desde D (o a través de una referencia a a D) se invoca a dicho método, se ejecutará el método de la clase que primero aparezca en la sección **Isa** de la clase D. En caso de que se desee acceder al otro método se deberá calificar el método, especificando antes del nombre del método la clase a la que pertenece, separada por dos puntos.

```
a.Metodo()             // método de la clase B
a.ClaseB:Metodo()    // método de la clase B
a.ClaseC:Metodo()    // método de la clase C
```

12.4.3 Uso exclusivo de métodos

No existen operadores (véase la clase básica Integer) con notación infija. Los operadores son construcciones de alto nivel que se implementarán como métodos. Esto es lo que hace al fin y al cabo el C++ a la hora de sobrecargarlos.

12.4.4 Uso exclusivo de enlace dinámico (sólo métodos virtuales)

No es necesario especificar si un método es virtual[1] o no, ya que todos los métodos son virtuales (polimórficos). Es decir, se utiliza únicamente el mecanismo de enlace dinámico, siguiendo la línea de una arquitectura OO más pura. El uso de enlace estático restringe en exceso la extensibilidad del código, perdiéndose una de las ventajas de la OO. La posible sobrecarga de ejecución de los métodos virtuales se puede compensar con una implementación interna eficiente.

12.4.5 Ámbito único de los métodos

No se restringe el acceso a los métodos clasificándolos en ámbitos como los private, public o protected del C++. Estos accesos son de utilidad para los lenguajes de alto nivel pero para una máquina no tienen tanto sentido. Si un compilador de alto nivel no desea que se accedan a unos determinados métodos private, lo único que tiene que hacer es no generar código de llamada a dichos métodos. Un ejemplo parecido ocurre cuando se declara una variable const en C++. No es que la máquina subyacente sepa que no se puede modificar; es el compilador el que no permite sentencias que puedan modificarla.

En cualquier caso, en el sistema operativo se diseñará un mecanismo de protección (véase el capítulo 15) que permitirá una restricción de acceso individualizada para cada método y cada objeto. Así se podrán crear ámbitos de protección particularizados para cada caso, en lugar de simplemente en grupos private, public y protected.

12.4.6 Inexistencia de constructores y destructores

No existen métodos especiales caracterizados como constructores ni destructores. Al igual que ocurre con algunos de los puntos anteriores, es el lenguaje de alto nivel el que, si así lo desea, debe generar llamadas a unos métodos que hagan dichas labores a continuación de las instrucciones de Carbayonia de creación y destrucción de objetos.

Sin embargo, no es necesario que se aporten métodos para la gestión de la semántica de los objetos agregados[2], que ya es conocida por la máquina y se realiza automáticamente.

12.4.7 Redefinición de métodos

Para que un método redefina (*overriding*) a otro de una superclase debe coincidir exactamente en número y tipo de parámetros y en el tipo del valor de retorno. No se permite la sobrecarga (*overloading*) de métodos (dos o más métodos con el mismo nombre y diferentes parámetros).

[1] En el sentido de C++ de la palabra virtual: utilizar enlace dinámico con ese nombre de método.
[2] Como por ejemplo su eliminación al eliminarse el objeto que los contiene.

12.5 Ejemplo de declaración de una clase

Como ejemplo de la descripción de clases en Carbayonia, se verá cómo se puede representar parte del siguiente diagrama (Flor, Rosa y Clavel).

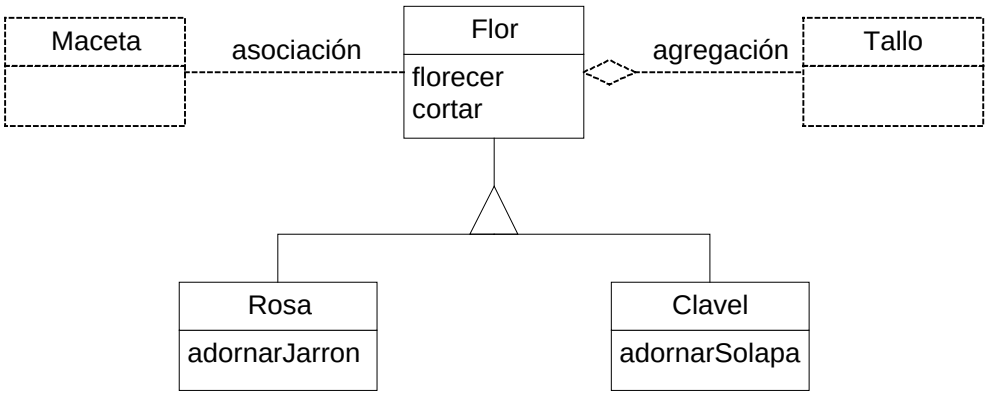


Figura 12.2 Diagrama de clases de ejemplo con asociación y agregación

```
Class Flor Isa Object / Se explica más adelante
Aggregation
  miTallo: Tallo;
Association
  laMaceta: Maceta;
Methods
  florecer(Hora, Amplitud): Olor;
  cortar(Altura);
EndClass
/-----------------------------------
Class Rosa
Isa Flor
Methods
  florecer(Hora, Amplitud): Olor;// Redefinición de florecer
  adornarJarron(Jarron);
EndClass
/-----------------------------------
Class Clavel
Isa Flor
Methods
  florecer(Hora, Amplitud): Olor;// Redefinición de florecer
  adornarSolapa(Solapa);
EndClass
```

12.6 Instrucciones de comportamiento: Definición de métodos

En la declaración de una clase se declaran también los métodos que tiene. Ahora es necesario definir exactamente cuál es el comportamiento de cada método. Para ello se distingue entre la cabecera del método y el código del mismo propiamente dicho. En la cabecera se definen los parámetros del método y las referencias (variables) locales que se utilizarán en el método. En el cuerpo o código se especifican la secuencia de instrucciones que definen el comportamiento del mismo.

12.6.1 Cabecera de método

La forma de describir la cabecera de un método es como sigue:

```
<Nombre> ([{Clase}])[:Clase]
[Refs {<Nombre>:Clase};]
[Instances {<Nombre>:Clase};]
Code
  <Codigo>
EndCode
```

A continuación de la descripción del nombre y los parámetros del método[1] (dentro de la cláusula Methods) se describen las referencias e instancias locales que va a usar el método. Entre Code y EndCode se define el código del método.

12.6.1.1 Refs (referencias)

Aquí se indican todas las referencias locales que usará el método, junto con su tipo, puesto que todas las referencias tienen un tipo asociado. Al entrar en el método se crean automáticamente las referencias locales, que se pueden usar en el cuerpo para trabajar con objetos (crear objetos a partir de las referencias, invocar métodos, etc.). Al finalizar el método, estas referencias locales desaparecen automáticamente.

Las referencias cumplen la función análoga a las variables de tipo puntero de un programa en C++. Inicialmente no apuntan a ningún objeto. Para acceder a un objeto, deben asignarse a un objeto existente o bien crear un nuevo objeto a partir de la referencia.

La idea es que la creación y eliminación de referencias sea realizada automáticamente. En lugar de incluir instrucciones del cuerpo del método que permitan crear y eliminar las referencias, se deja que lo haga la máquina automáticamente. Es una muestra más de la elevación del nivel de abstracción, que libera trabajo al programador (y compiladores), evitando errores como el olvido de eliminar una referencia, etc.

```
Refs
  i: Integer;
  b: ClaseB;
```

Recolección de basura

Hay que recalcar que sólo se liberan las referencias, la instancia a la que apunte una referencia no se elimina automáticamente. Es responsabilidad del programador eliminar estos objetos si se considera adecuado. No existe, por tanto, una gestión automática de memoria (recolector de basura) que se ocupe de realizar esta tarea. Existen situaciones en las que no es conveniente la recolección de basura, por ejemplo en situaciones en las que es necesario asegurar la eliminación de un objeto, generando un registro de auditoría [Mal96]. La posibilidad de habilitar un mecanismo en la máquina que permita realizar recolección de basura se estudiará en versiones posteriores de la misma.

12.6.1.2 Instances (instancias)

Es una cláusula similar a la anterior. La única diferencia es que la máquina gestiona automáticamente la creación y eliminación de la instancia a la que apunta la referencia. Al entrar en el método, además de la referencia, se crea la instancia a la que apunta la referencia. Al finalizar el método, se libera la instancia.

[1] Los parámetros del método son objetos, que, como siempre, se manejan a través de referencias a los mismos. Por tanto realmente los parámetros son referencias.

Las instancias son equivalentes a las variables normales en C++. Además de existir la variable, inicialmente el objeto al que apuntan es creado automáticamente.

Esta cláusula se utiliza en aquellos casos en los que el método necesita objetos locales, pero que sólo se utilizarán dentro del método. En estas situaciones se evitan muchos problemas haciendo que estos objetos se creen y liberen automáticamente sin intervención del programador. Es una situación que eleva el nivel de abstracción, de manera análoga a cómo se hace con las simples referencias.

Inicialización de objetos de clases básicas

El lenguaje permite una construcción especial dentro de la cláusula Instances para las clases básicas Integer, Float y String. Consiste en poder indicar entre paréntesis un valor de inicialización para los objetos de estos tipos.

```
Instances
  i: Integer(10);
  f: Float(3,5);
  s: String('Esto es una cadena de texto');
```

La razón de esta construcción es que el lenguaje está basado en que todas las operaciones se hacen a través de referencias, y eso es lo que se envía y recibe en los métodos. Por tanto no puede hacerse algo como:

```
...
Consola.Write('Hola a todos');
...
```

El texto es una constante de cadena (no un objeto) y no es lo que el método espera. Por tanto, hay que desglosar lo anterior (labor que generalmente será realizada de forma automática por el compilador de alto nivel) en:

```
Método(...)
Instances
  consola: ConStream;
  cadena: String('Hola a todos');
Code
  ...
  consola.Write(Cadena);
  ...
EndCode
```

Ahora lo que recibe el método sí es una referencia a un objeto. La situación es similar a lo que ocurre en la traducción de un programa en C, cuando en la generación de código se sustituyen las cadenas por punteros a la zona estática de memoria donde se han movido cada una de ellas. La única diferencia es que aquí el concepto también se extiende a los enteros.

12.6.2 Code. Código del cuerpo de método

A continuación se describen las instrucciones que se pueden utilizar en el cuerpo de los métodos (entre las palabras Code y EndCode). Todas las instrucciones se terminan con un punto y coma. Estas son las instrucciones de comportamiento que son las que aparecen en las

máquinas convencionales. Siguiendo la filosofía de disponer de un juego de instrucciones reducido, sólo existen en torno a 15 instrucciones.

La relación de estas instrucciones se encuentra en el apéndice F.

12.6.3 Trabajo con objetos a través de referencias

La parte más importante son las instrucciones que permiten trabajar con los objetos. Son operaciones que se pueden realizar con cualquier objeto. Todas estas operaciones se tienen que realizar siempre a través de una referencia. Se necesitan operaciones para ligar instancias a referencias (creación de objetos y asignación de referencias), invocar métodos y eliminar objetos.

12.6.3.1 Creación de objetos

Se puede ligar una instancia a un objeto mediante la operación New, que crea un nuevo objeto del tipo que tenga la referencia, y deja la referencia apuntando al nuevo objeto.

```
New <Referencia>
```

En caso de que no se pueda crear el objeto, se lanza una excepción (por ejemplo por una falta de espacio).

Al crear un objeto se crean a su vez todos los objetos agregados que pertenezcan a dicho objeto. Por tanto, no se debe usar New con estos objetos agregados. Tampoco con las referencias declaradas en Instances, puesto que también se crean las instancias automáticamente. Las referencias de tipo Association se encuentran inicialmente libres (no apuntan a ningún objeto), al igual que las declaradas en Refs. Sobre estas últimas es sobre las que se puede aplicar New. Con las referencias recibidas como parámetros del método, esto depende de la lógica de aplicación que exista en cada caso.

Una restricción muy grande que tienen lenguajes como C++ es que la manera anterior de crear objetos sólo permite crear objetos de un tipo conocido en tiempo de compilación. No se pueden crear (fácilmente) objetos cuyo tipo se determine en tiempo de ejecución. Para permitir esto se puede añadir una segunda versión de la operación New, que tenga un parámetro adicional de tipo cadena (String) o derivado del que se tome el nombre de la clase de la que se creará el objeto[1].

```
New <refString>, <Referencia>
```

Para que no exista ambigüedad en la distinción entre las dos versiones de la instrucción New, cuando esta distinción sea necesaria, se denominará a la segunda versión New2.

12.6.3.2 Asignación de referencias

Si se tiene una referencia que apunta a un objeto se puede hacer que otra referencia apunte al mismo. Esto se hace mediante la instrucción Assign. Esta es la otra posibilidad para ligar una referencia a un objeto. Otras formas son mediante asignaciones implícitas que realiza la máquina con los parámetros de un método y el objeto de retorno.

[1] Con las comprobaciones de seguridad adecuadas. Por ejemplo, la clase que especifica la cadena tiene que ser una subclase del tipo de la referencia.

```
Assign <ReferenciaDestino>, <ReferenciaFuente>
```

Al finalizar la instrucción el objeto se podrá manipular por cualquiera de las dos referencias indistintamente. Se producirá una excepción si la referencia destino no es compatible con el tipo del objeto al que apunta la referencia fuente.

Amoldamiento en tiempo de ejecución

Esta instrucción se beneficia de la comprobación de tipos en tiempo de ejecución por parte de Carbayonia. Si se tiene una referencia ra de tipo Automóvil y otra referencia rb de tipo Bugatti tal que la clase B deriva de A, se permiten las siguientes asignaciones:

```
Assign ra, rb
Assign rb, ra
```

El primer caso es un amoldamiento habitual de la clase hija a la clase base, que siempre puede realizarse (un Bugatti siempre es un Automóvil). Sin embargo, en el segundo caso se presenta la situación opuesta. Si el objeto al que apunta ra es efectivamente un objeto de tipo Bugatti, el amoldamiento se realizará sin problemas. Pero si el objeto no es de tipo Bugatti se producirá una excepción. Si el Automóvil es, por ejemplo, un BMW, no puede asignarse a un Bugatti. Sólo puede hacerse si el Automóvil es efectivamente un Bugatti.

Este segundo caso no se aprovecha en sistemas con comprobación estática de tipos. Si sólo se tuviera en cuenta el tipo de las referencias[1], el segundo amoldamiento se prohibiría, puesto que en tiempo de ejecución la segunda referencia podría apuntar a un objeto de clase Automóvil simplemente. Sin embargo, puede ser que el objeto al que apunte la referencia ra sea un Automóvil, pero un Bugatti. Si este es el caso, la asignación podría hacerse sin problemas (mediante una referencia de tipo Bugatti se apunta a un objeto de tipo Bugatti).

12.6.3.3 Invocación de métodos

Es en esencia la única cosa que se puede hacer con un objeto. El código de un método es básicamente una sucesión de llamadas a métodos.

```
<Referencia>.[<Ambito>:]<Metodo>({<Referencia>})[:Referencia]
```

La llamada al método está formada por la referencia al objeto al que se desea pasar el mensaje seguido del nombre del método. Opcionalmente éste puede ir precedido del **ámbito** de la referencia: uno o más nombres de la jerarquía de clases del tipo de la referencia si son necesarias para deshacer ambigüedades en los casos de herencia múltiple. A continuación van los parámetros referencia, separados por coma y finalmente, si el método tiene valor de retorno, la referencia que recogerá dicho objeto de retorno

```
pantalla.leerPixel (rx, ry) : color
```

En cuanto al emparejamiento de parámetros no es necesario que éstos coincidan exactamente, ya que se puede considerar que internamente se realiza una asignación entre cada tipo de objeto que se envía y el que se espera en el método. Es decir, que se hacen los

[1] Que es la información que se tiene en tiempo de compilación.

amoldamientos necesarios tanto ascendentemente como descendentemente. Por tanto, si un objeto no puede amoldarse a un parámetro se producirá una excepción. Lo mismo ocurre con el valor de retorno a la hora de adaptar los tipos.

Se utiliza la notación anterior para hacerla más parecida a la invocación de métodos en los lenguajes más difundidos. Una manera equivalente de verlo, más similar a la del resto de las instrucciones sería la de una instrucción de llamada, seguida de los parámetros necesarios:

```
Call <refObjeto> <refParam1> <refParam2> ... <refRetorno>
```

12.6.3.4 Encapsulamiento

Como se ha comentado anteriormente, a un objeto sólo se le pueden enviar mensajes. Por tanto, no se puede acceder a sus variables miembro (agregados y asociados). Esto es así debido a que se mantiene el principio de encapsulamiento, que no permite esta práctica. La interacción con un objeto debe hacerse exclusivamente a través de mensajes a éste. Por tanto, si se necesita acceder a las variables miembro de un objeto habrá que crear métodos de acceso en el mismo.

Acceso al objeto actual

Para poder llamar a los métodos del objeto actual (el propio objeto que está siendo ejecutado) se utiliza la referencia del sistema this. Esta referencia apunta al objeto actual, así que se puede utilizar como cualquier otra referencia para llamar a métodos.

```
this.miMetodo(pa, pb)  / Llama a miMetodo del objeto actual
```

12.6.3.5 Eliminación de objetos

La instrucción Delete elimina el objeto al que apunta la referencia

```
Delete <Referencia>
```

Se producirá una excepción en los siguientes casos:

- La referencia está libre.
- La referencia apuntaba a un objeto que ya ha sido eliminado.
- Se intenta liberar un objeto agregado. Estos objetos no se pueden eliminar directamente. Sólo se podrán liberar cuando se libere el objeto del que forman parte, y esto lo hace automáticamente la máquina.

12.6.4 Control de flujo

Estas instrucciones están relacionadas con el funcionamiento del flujo de procesamiento de la máquina.

12.6.4.1 Finalización de un método

La instrucción Exit finaliza la ejecución de un método.

```
Exit
```

Esto libera todas las referencias que se hayan creado mediante Refs y todas las referencias y objetos que se hayan creado mediante Instances.

12.6.4.2 Valor de retorno de un método

Para devolver el objeto resultado de un método se utiliza la referencia del sistema rr (referencia de retorno). La máquina asigna esta referencia al parámetro que se utiliza para recoger el valor de retorno. En el cuerpo del método es necesario, pues, utilizar una asignación a la referencia rr del valor de retorno que se desee.

```
Assign rr, objetoADevolver
Exit
```

Se considera la posibilidad de eliminar la utilización de la referencia del sistema e incorporar la devolución del objeto de retorno en la propia instrucción de finalización de método, por ejemplo Exit objetoADevolver.

12.6.4.3 Salto incondicional

Cambia la ejecución secuencial de instrucciones normal de un método, saltando incondicionalmente a una etiqueta específica que marca una instrucción determinada.

```
Jmp Etiqueta
...
Etiqueta: / Otras instrucciones
```

12.6.4.4 Salto condicional

Dirigen la ejecución del programa a otro punto dentro del mismo método marcado con una etiqueta si se cumple una cierta condición. Estas instrucciones tienen dos parámetros: el objeto sobre el que se comprueba la condición y la etiqueta a la que se salta.

Comprobaciones sobre objetos booleanos

Utilizan un objeto de tipo básico Bool (booleano). Hay dos tipos de salto en función de si el objeto Bool vale verdadero o falso

```
JT <ReferenciaBool>, Etiqueta
JF <ReferenciaBool>, Etiqueta
```

JT salta a la etiqueta si la referencia vale “Verdadero”, continuando la ejecución en la siguiente instrucción si no es así. JF tiene el efecto contrario, salta si el valor Bool vale “Falso”.

Es útil disponer de una variante que permita eliminar directamente el objeto Bool utilizado. Esto es lo que hacen las instrucciones anteriores, a las que se añade una D (de *Delete*, borrar).

```
JTD <ReferenciaBool>, Etiqueta
JFD <ReferenciaBool>, Etiqueta
```

Comprobaciones sobre referencias

En muchos casos lo que se pretende comprobar es simplemente si un objeto existe o no, es decir, si una referencia está libre o no

```
JNull <Referencia>, Etiqueta
JNNull <Referencia>, Etiqueta
```

JNull salta a la etiqueta si la referencia está libre (es *Null*, nula). JNNull salta si la referencia está ocupada (no está libre o no es nula, *Not Null*).

12.6.5 Excepciones

El tratamiento de excepciones, como parte aceptada de los lenguajes orientados a objeto y de las metodologías, está incluido en Carbayonia. Las ventajas que se podrían señalar a la incorporación de las excepciones son:

- **Facilitar la tarea de depuración y mantenimiento de las aplicaciones**. En los procesadores convencionales, las instrucciones se ejecutan sin poder comprobar si se las está utilizando correctamente. Por ejemplo, una instrucción RET saca de la pila la dirección de retorno pero suponiendo que lo que saca es efectivamente una dirección.

  Lo mismo pasa cuando se utilizan punteros: una instrucción MOVE no puede comprobar si va a escribir en una dirección válida, si se está saliendo del rango, si esa zona de memoria ya se ha liberado, etc. Simplemente se deja que el programador se haga responsable de las consecuencias.

  En Carbayonia se debe intentar que las instrucciones puedan detectar el máximo número posible de situaciones en las que se las esté utilizando incorrectamente. Así, por ejemplo, la instrucción Delete produce una excepción si se le pasa una referencia a todo aquello que no sea un objeto válido (una referencia libre, un objeto ya liberado, un objeto que es agregado de otro, etc.). Lo mismo ocurre con el resto de las instrucciones, lo cual es una inestimable ayuda a la hora de la detección de errores.

  Esto es posible gracias a la información que se guarda en las áreas de clases, referencias e instancias. Si se eliminase el uso de excepciones por parte del micro, simplemente en el ejemplo anterior del Delete se perdería una valiosa oportunidad para detectar unos tipos de errores que serían de difícil detección de otra forma.

- **Programación de usuario más robusta**. Al estar las excepciones tan arraigadas en la forma de programar la máquina, se promueve su utilización en los programas de usuario y, por tanto, la generación de un código más robusto por parte de éstos.

Se definen dos instrucciones para el control de las excepciones: Handler y Throw.

```
Handler Etiqueta
...
Throw
```

La misión de la instrucción Handler (manejador) es indicar la dirección donde se debe continuar el flujo de ejecución en caso de que ocurra una excepción (la dirección del manejador). La dirección será aquella que corresponda a la posición de la etiqueta en el código fuente.

Throw (lanzar) lanza una excepción. Cuando se ejecute un Throw, la ejecución del programa continuará en la dirección del último manejador ejecutado. En caso de que no haya

ninguno la ejecución se dará por finalizada. Los distintos manejadores se van apilando de manera que tienen prioridad los últimos que se hayan ejecutado.

La referencia del sistema exc se puede utilizar para pasar un objeto que represente la causa de la excepción. Como en el caso del valor de retorno de un método, se podría utilizar una manera alternativa que consista en que el objeto causa de la excepción sea un parámetro del Throw.

El lanzamiento de una excepción puede ser por un posible error en tiempo de ejecución (salirse del rango de un array) o bien porque el usuario la haya lanzado con la instrucción Throw.

12.6.5.1 Abandono del método actual

La cesión de control a un manejador se realiza aunque para ello haya que abandonar el método actual. Cuando se lanza una excepción en un método que no ha definido un Handler, se retrocede en la secuencia de llamadas que llevó hasta la situación actual intentando encontrar algún método de lo recorridos que hubiese declarado un Handler. Cada método que deba abandonarse por causa de una excepción libera previamente los objetos declarados en la sección Instances del método (se finaliza el método como con Exit).

Los manejadores no solo se extraen cuando se produce una excepción, sino que también se descartan cuando se sale del método en el que fueron declarados sin que se produjese una excepción.

12.6.5.2 Analogía de la pareja Throw / Handler con la invocación a método / Exit

Existen varias similitudes ente la pareja formada por la llamada a un método y el Exit que se ejecuta dentro de éste, y la pareja Throw/Handler.

- La llamada a un método le deja al Exit la dirección donde tiene que retornar, de la misma manera que el Handler se la deja al Throw.
- El Exit retorna la última llamada a un método realizada, lo mismo que el Throw al último Handler ejecutado.
- El Exit descarta todos los Handlers que se encuentren entre él y su llamada al procedimiento. Igualmente el Throw descarta todas las direcciones de retorno de las llamadas a procedimientos que encuentre entre él y su Handler (para ello liberando los objetos de la cláusula Instances).

Los tres puntos anteriores, sugieren que ambos comparten una misma pila en la que se insertan tanto direcciones de retorno como manejadores. Dependiendo de que se ejecute un Exit o un Throw éstos irán retirando elementos de la pila hasta que encuentren su pareja. Sin embargo, esto es un detalle de implementación.

12.6.5.3 Implementación de Try/Catch del C++ con Handler/Throw

A modo de ejemplo, se implementará la construcción Try y Catch del C++ mediante Handler y Throw.

Se trata de sustituir el Try por un Handler. Dicho Handler indicaría la dirección del primer bloque Catch. Este bloque podría comprobar si el objeto lanzado (al cual apunta la referencia del sistema exc) es del tipo del Catch utilizando el método isa, disponible en todos los objetos. Es un método de la clase básica Object que se describirá posteriormente. Indica si el objeto pertenece a la clase que se le pasa como parámetro o no. Si el tipo es el adecuado se procederá

a su tratamiento dentro del Catch. Si no es así se relanzará mediante un nuevo Throw para que sea gestionado por el Handler adecuado a un nivel superior.

C++	Carbayonia

Try
  <Código con Excepciones>
 	
Catch ClaseX
  <Código de Tratamiento>
 	
EndTry
  <Continua la Ejecución>
	Refs
	  b: Bool;
	Instances
	  str: String('ClaseX');
	Code
	  Handler CatchEtq;
	  <Código con Excepciones>
	  Jmp EndTryEtq
	
	CatchEtq:
	  exc.isa(str): b;
	  JTD b, Tratar
	  Throw
	
	Tratar:
	  <Código de Tratamiento>
	
	EndTryEtq:
	  <Continua la Ejecución>

En el caso de varios Catch consecutivos, en vez de relanzar directamente la excepción antes se comprobaría si el tipo del objeto lanzado coincide con algún otro de los Catch.

12.6.5.4 Pérdida de objetos en una excepción

La gran dificultad que surge a la hora de introducir las excepciones es el hecho de que un método puede perder el control si se produce una excepción, y, por tanto ¿qué pasa con los objetos que éste haya creado y no le haya dado tiempo a liberar?.

Si no se diseña cuidadosamente cada vez que se produzca una excepción pueden perderse varios objetos en el área de instancias (pequeño inconveniente a pagar por no tener recolector de basura). Esto también ocurre en C++, ya que aunque se liberan los objetos locales (variables de tipo auto), los que se hayan creado con el operador New se pierden irremisiblemente.

12.7 Jerarquía de clases básicas

Independientemente de las clases que defina el programador (y que se irán registrando en el área de clases), Carbayonia tiene una serie de clases básicas que se pueden considerar como definidas en dicho área de manera permanente.

Estas clases básicas serán las clases fundamentales del modelo único que se utilizarán para crear el resto de las clases. Una aplicación Carbayonia es un conjunto de clases con sus

métodos en los cuales se llaman a otros métodos. Siguiendo este proceso de descomposición, siempre llegamos a las clases básicas y a sus métodos.

En cualquier caso, estas clases básicas no se diferencian en nada de cualquier otra clase que cree el usuario. Desde el punto de vista de utilización son clases normales como otras cualesquiera. Cada implementación de la máquina establecerá los mecanismos necesarios para proporcionar la existencia de estas clases básicas.

Las clases básicas se organizan en una jerarquía, cuya raíz es la clase básica Object.

No se sigue un orden alfabético en la descripción de las clases, sino el más adecuado para la explicación de cada una. Sin embargo, no siempre se ha podido evitar hacer referencia a clases que aun no se han mencionado

12.7.1 Object

La declaración de esta clase es

```
Class Object
Methods
  getClass(): String;
  getID(): Integer;
  isa(String): Bool;
EndClass
```

Esta clase es la base de cualquier otra clase de Carbayonia. Todas las demás derivan de ella aunque no se especifique explícitamente mediante Isa. En esta clase se colocarán todos los métodos que necesiten estar presentes para todos los objetos:

- **getClass(): String**. El método getClass devuelve una cadena (String, clase que se verá posteriormente)que contiene el nombre de la clase a la que pertenece el objeto. Esto permite conocer el tipo verdadero de un objeto desde cualquier referencia que le apunte, independientemente del tipo de la referencia.
- **getID(): Integer**. El método getID devuelve el identificador único del objeto mediante una instancia de un entero. Entre otras cosas, de esta forma se puede averiguar si se tiene dos referencias apuntando al mismo objeto (aunque estas sean de distinto tipo).
- **isa(String): Bool**. El método isa devuelve una instancia de un objeto de tipo Bool que indica si la instancia pertenece a la clase que se le pasa como parámetro o a una derivada de ella. Es decir, indica si el objeto es-un (*is-a*) objeto de la clase que se pasa como parámetro.

Las principales funciones de esta clase son:

- Permitir la construcción de contenedores genéricos. Al ser toda clase derivada de Object (explícita o implícitamente) toda instancia será un Object, por lo que dicho tipo se puede utilizar en las estructuras de datos genéricas.
- Realizar parte de las funciones propias de la comprobación de tipos en tiempo de ejecución (mediante la utilización de sus métodos isa y getClass).

12.7.2 Bool

Esta clase representa valores booleanos.

```
Class Bool
Isa Object
Methods
  setTrue();
  setFalse();
  not();
  and(Bool);
  or(Bool);
  xor(Bool);
EndClass
```

La clase Bool, como es de esperar, solo puede tomar dos valores: true (verdadero) o false (falso). La importancia de ésta clase radica en que las instrucciones de salto utilizan instancias de la misma para decidir la dirección del salto.

- **setTrue() y setFalse()**. Los métodos setTrue y setFalse no llevan argumentos y establecen el estado de la instancia a verdadero y falso respectivamente.
- **not()**. El método not invierte el valor del objeto, de manera que si vale true pasa a valer false y viceversa.

and(Bool), or(Bool) y xor(Bool).. El método and realiza un Y lógico entre el estado del objeto al que se le envía el mensaje y el estado del objeto que se le envía como parámetro (el cual deberá ser de tipo Bool o derivado). De la misma manera los métodos or y xor realizan respectivamente las operaciones O lógico y O exclusivo entre el estado de la instancia que recibe el mensaje y el del parámetro.

12.7.3 Integer

La clase Integer representa un entero con signo.

```
Class Integer
Isa Object
Methods
  add(Integer);
  sub(Integer);
  mul(Integer);
  div(Integer);
  set(Integer);
  setF(Float);
  equal(Integer): Bool;
  greater(Integer): Bool;
  less(Integer): Bool;
EndClass
```

La forma de trabajar con enteros cambia ligeramente a como es lo habitual. En lugar de disponer de instrucciones especiales que operen con enteros, se utiliza un mecanismo totalmente orientado a objetos: métodos de la clase. Por la misma razón no existen operadores infijos con enteros, se llama a un método de un entero para operar con otro entero que se le

pase como parámetro. (El mismo razonamiento se aplica al resto de las clases, como la clase Bool vista anteriormente).

- **add(Integer), sub(Integer), mul(Integer) y div(Integer)**. Los cuatro primeros métodos son las cuatro operaciones básicas. Realizan respectivamente la suma, resta, multiplicación y división del valor del objeto que recibe es mensaje por el valor del objeto que se le envía como parámetro.

  El método de división lanza una excepción si el parámetro vale 0.

- **set(Integer) y setF(Float)**. Los dos siguientes métodos sirven para asignar un valor al entero, bien sea a partir de otro entero o bien mediante la conversión de un número en coma flotante (Float)

- **equal(Integer): Bool, greater(Integer): Bool y less(Integer): Bool**. Los tres últimos métodos se utilizan para comparación de enteros. Reciben un parámetro de tipo entero y devuelven una instancia de tipo Bool que indica el resultado de la comparación: si el objeto es igual, mayor o menor que el entero pasado, respectivamente.

12.7.4 Float

La clase Float representa a un número en coma flotante.

```
Class Float
Isa Object
Methods
  add(Float);
  sub(Float);
  mul(Float);
  div(Float);
  set(Float);
  setI(Integer);
  equal(Float): Bool;
  greater(Float): Bool;
  less(Float): Bool;
EndClass
```

La clase Float es idéntica a la clase Integer excepto en que su estado es de tipo número en coma flotante en vez de un entero.

Los métodos Add, Sub, Mul y Div realizan respectivamente la suma, resta, multiplicación y división del estado del objeto por el estado del parámetro.

Los dos siguientes métodos son para establecer el estado del objeto. Dicho estado puede ser establecido a partir de otro Float o bien de un entero, en cuyo caso se realizará una conversión de tipo.

12.7.5 String

La clase String representa a las cadenas de caracteres

```
Class String
Isa Object
Methods
  set(String);
  setI(Integer);
  length(): Integer;
  get(Integer,Integer);
  insert(String,Integer);
  delete(Integer, Integer);
  getChar(Integer): Integer;
  setChar(Integer,Integer);
  equal(String): Bool;
  greater(String): Bool;
  less(String): Bool;
EndClass
```

- **set(String)**. Establece el valor de la cadena a la pasada como parámetro.
- **setI(Integer)**. Transforma el entero pasado como parámetro a una cadena de caracteres, adquiriendo la cadena este valor.
- **length(): Integer**. Devuelve el número de caracteres que forman la cadena actual.
- **get(Integer, Integer): String**. Devuelve una subcadena de caracteres, la cual comienza en la posición del primer parámetro y tiene tantos caracteres como indique el segundo. Se produce una excepción si la subcadena no está incluida en el String (inicio o longitud inadecuados).
- **insert(String, Integer)**. Inserta una subcadena en la posición indicada por el entero. Se produce una excepción si la cadena actual no llega a dicha posición.
- **delete(Integer, Integer)**. Borra tantos caracteres como indique el segundo parámetro a partir de la posición que indique el primero. Se produce una excepción si el rango no es correcto.
- **getChar(Integer): Integer**. Devuelve un entero cuyo valor es el código ASCII del carácter que esté en la posición indicada. Se produce una excepción si el rango no es correcto.
- **setChar(Integer, Integer)**. Cambia el carácter que ocupe la posición indicada en el primer parámetro con el carácter cuyo código ASCII sea el valor indicado en el entero del segundo parámetro. En caso de que el entero esté fuera del rango 0-255 se producirá una excepción. Así mismo se producirá también una excepción si la posición indicada está fuera de rango.

equal(String): Bool, **greater(String): Bool** y **less(String): Bool**. Devuelve un objeto de tipo Bool que indica si la cadena parámetro es igual, mayor o menor que la actual, respectivamente. La comparación distingue entre mayúsculas y minúsculas.

12.7.6 Array

Esta clase representa un vector unidimensional de objetos.

```
Class Array
Isa Object
Methods
  setSize(Integer);
  getSize(): Integer);
  setRef(Integer,Object);
  getRef(Integer): Object;
EndClass
```

Dado que esta es una arquitectura pura de objetos, los elementos del array son referencias a objetos, no las propias instancias. Tampoco se encarga de liberar los objetos a los que apuntan las referencias al eliminar el array. Podría pensarse en una versión del array en el que los elementos del mismo tuvieran un comportamiento tipo agregado, en lugar de este comportamiento tipo asociación genérica.

- **setSize(Integer)**. El método setSize se utiliza para indicar el tamaño del array. Se puede cambiar el tamaño del array en cualquier momento. Si éste se aumenta de n a m posiciones, las n primeras posiciones del nuevo array serán las mismas de antes. Si se disminuye, las posiciones que sobren simplemente se ignoran.
- **setRef(Integer, Object) y getRef(Integer): Object**. Estos métodos permiten almacenar y extraer elementos en el array. setRef almacena en la posición indicada por el primer parámetro el objeto indicado en el segundo. getRef devuelve el objeto almacenado en la posición indicada en el parámetro. Ambos producen una excepción si el índice se sale fuera del rango de posiciones del array.

12.8 Elementos transitorios de la máquina. Fases de desarrollo

Otras partes del proyecto desarrollan aspectos del mismo que proporcionarán propiedades específicas al sistema, fundamentalmente el sistema operativo. Ejemplos de esta funcionalidad son la interacción con el entorno hardware (entrada/salida) y el desarrollo de un modelo de concurrencia. La funcionalidad que ofrecerá el sistema operativo requiere un estudio pormenorizado por parte de otros investigadores.

Sin embargo, un sistema que no disponga de estos elementos no puede ser utilizado de manera completa. Es importante proporcionar con la primera versión de la máquina una funcionalidad mínima de la parte que luego será desarrollada por el sistema operativo, por dos razones:

- Para el propio desarrollo del sistema operativo se necesita un prototipo de la máquina abstracta que sea utilizable de manera completa. Es posible que se introduzcan modificaciones en la máquina para soportar mejor el SO.
- Paralelamente al desarrollo del sistema operativo otras áreas necesitan la máquina abstracta para avanzar en la investigación: desarrollo de aplicaciones, compiladores, bases de datos, etc. e incluso en la propia máquina abstracta.

A pesar de que el sistema con estos elementos transitorios esté sujeto a cambios según el desarrollo del SO, éstos no tendrán un impacto grande en otras áreas. Las adaptaciones

necesarias por los cambios futuros se compensan por la posibilidad de adelantar en el tiempo la investigación.

12.8.1 Fases en el desarrollo de la máquina abstracta

Según lo expuesto anteriormente, el desarrollo de la máquina se dividirá en las siguientes fases:

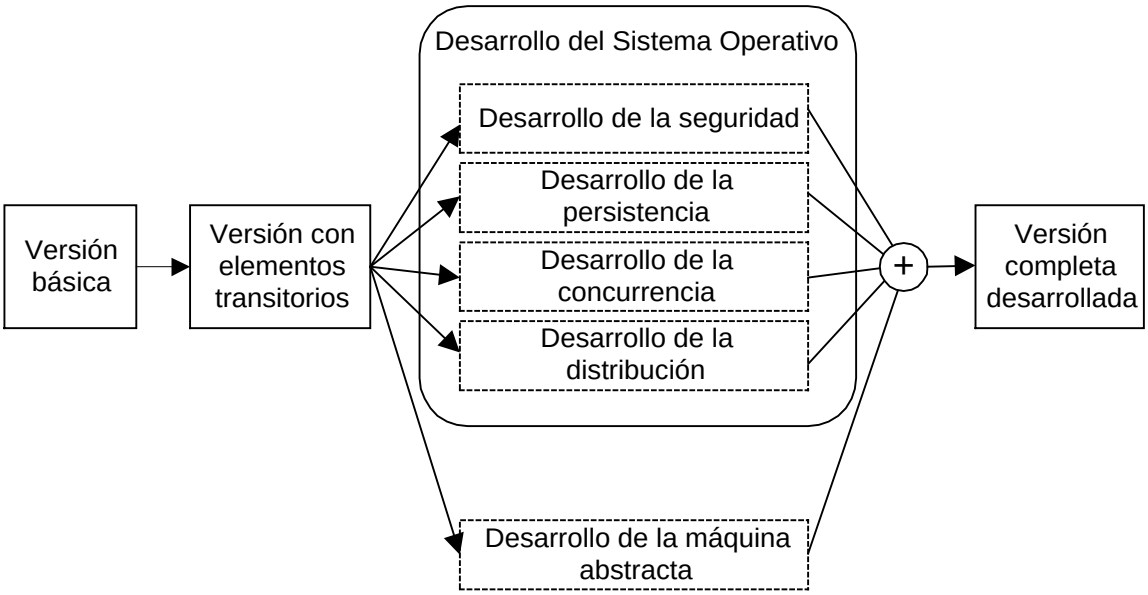


Figura 12.3 Fases en el desarrollo de la máquina abstracta

- **Versión básica**. Que incorpore los elementos fundamentales de la arquitectura de la máquina. Se corresponde con lo descrito de la máquina hasta ahora.
- **Versión con elementos transitorios**. La versión básica se amplía con los elementos necesarios para poder ser utilizada, fundamentalmente los aspectos de entrada/salida y concurrencia.
- **Desarrollo del sistema operativo: seguridad, persistencia, concurrencia y distribución**. Se desarrolla toda la funcionalidad encargada al SO, especialmente estos apartados. Como resultado de este desarrollo se realizarán los cambios necesarios en la máquina abstracta.
- **Desarrollo de la máquina abstracta**. Paralelamente se irá evolucionando la propia máquina abstracta, incorporando características de la misma, como la reflectividad y evolucionando la implementación buscando más optimización.
- **Versión completa desarrollada**. Se integran las diferentes modificaciones de la máquina en una versión final acabada.

En los siguientes apartados se describen los elementos transitorios que se han utilizado para dar un soporte inicial para la concurrencia y la Entrada/Salida

12.8.2 Soporte transitorio para concurrencia

En este apartado se describe el modelo de concurrencia[1] implementado en la máquina de manera transitoria. Una discusión de las alternativas que se manejaron hasta llegar a este diseño se puede consultar en [Izq96].

12.8.2.1 Área de hilos

El objetivo fundamental es permitir la existencia simultánea de varios hilos de ejecución de una manera sencilla que se integre fluidamente en la estructura anterior de la máquina, manteniendo la sincronización entre ellos.

Para ello es necesario que exista un soporte en la máquina para los hilos. Hasta ahora se entendía que conceptualmente existía un único hilo en la máquina. Ahora se trata de dar soporte a varios hilos como el anterior simultáneamente. Puede pensarse que existe una nueva área, el área de hilos en la que se encuentran las estructuras para representar el funcionamiento de estos hilos. Inicialmente la máquina utilizará un planificador integrado para repartir el tiempo de ejecución entre los hilos existentes:

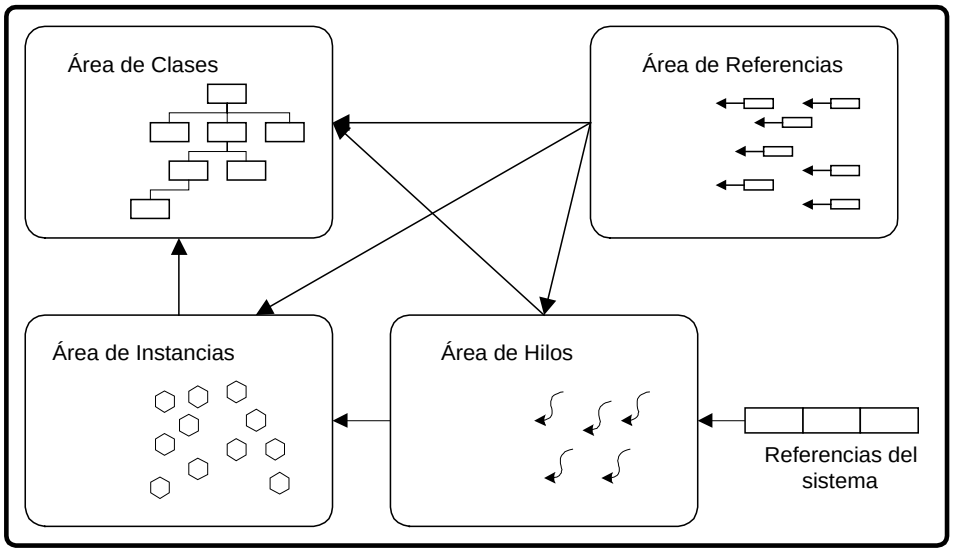


Figura 12.4 Estructura de la máquina Carbayonia con el soporte transitorio para hilos

12.8.2.2 Creación de hilos

El enfoque elegido es que sea la persona encargada del diseño de la clase la que decida para cada servicio (método) si la llamada al método bloquea al cliente o bien se ejecuta concurrentemente con él (creando un nuevo hilo). Los métodos cuya semántica encaja mejor con su ejecución concurrente son aquellos que no devuelven resultado (y con parámetros sólo de entrada). Se pueden considerar que son mensajes que se envían a la clase para que realice algo. Una vez enviado el mensaje, la acción que indique éste sigue su curso sin interacción con lo que haga el objeto cliente. Por tanto, se clasifican los métodos de una clase en dos tipos:

- **Métodos**. Son los que se han venido utilizando hasta ahora. La invocación de un método bloquea al cliente (objeto que lo llamó) hasta que el método finalice. Cuando

[1] Otro aspecto de la máquina que puede sufrir modificaciones relacionadas con la concurrencia es la parte de excepciones, al ser una forma de control de flujo.

se invoca a un método la ejecución se traslada a él hasta que finalice y sigue por el punto donde fue llamado. En la implementación interna se podría usar un único hilo que va avanzando por los diferentes métodos que se llaman.

- **Mensajes**. Son métodos que crean un nuevo hilo que nace en el método invocado. Se les llama así porque se podrían interpretar como el paso de un mensaje del cliente al objeto destino (con sus correspondientes parámetros) para que realice, cuando pueda, una determinada labor. Pero, al igual que cuando se envía un mensaje a una persona, simplemente se comunica el mensaje con lo que se desea y se pasan a hacer otras cosas, no se espera bloqueado a que se realice lo solicitado.

Se añade entonces una nueva cláusula Messages (mensajes) en la declaración de una clase en la que se especifican los métodos que tienen el comportamiento de mensajes. Estos nunca podrán tener valor de retorno.

```
Class <Nombre>
Isa {<Clase>}
Aggregation
  {<Nombre>:<Clase>}
Association
  {<Nombre>:<Clase>}
Methods
  {<Nombre>({<Clase>})[:<Clase>];}
Messages
  {<Nombre>({<Clase>});}
EndClass
```

Esta es una declaración de una propiedad de los métodos que define el creador de la clase. Los objetos clientes invocan los métodos de la misma manera de antes y desconocen si la invocación conlleva espera (bloqueo) o no. Este hecho es totalmente transparente para ellos. Se trata de trasladar la preocupación por el control de la concurrencia a un único punto donde se maneje de manera más sencilla: en la definición de cada clase. El uso de las clases en el resto del sistema no sufre ninguna modificación, ni es necesario preocuparse por el aspecto de la concurrencia.

12.8.2.3 Sincronización de métodos dentro de un objeto

Hay que solucionar posibles problemas de exclusión mutua e independencia en el tiempo bien conocidos en el campo de los sistemas operativos [Dei90]. En este caso, los elementos que intervienen no son procesos, si no los métodos de una clase. Éstos se pueden clasificar en dos tipos:

- **Selectores**: métodos que acceden al estado de un objeto pero no alteran ese estado. En C++ correspondería con las funciones miembro const. Los selectores son métodos que no necesitan exclusión mutua entre ellos de ningún tipo, puesto que no modifican el estado. Por tanto, pueden estar activos simultáneamente todos los selectores que se deseen de un objeto sin problema.
- **Modificadores**: métodos que alteran el estado de un objeto. En este caso, existe la posibilidad de que necesiten trabajar en exclusión mutua con cualquier otro método (no cumplen las condiciones de Bernstein [Dei90] para asegurar la independencia en el tiempo). La manera de asegurar la independencia en el tiempo, en cualquier caso, es obligar a que los métodos modificadores siempre trabajen en exclusión mutua.

Un método en Carbayonia se considera por defecto como modificador, por lo que tendrá exclusión mutua con cualquier otro método de la clase. Solo podrá ejecutarse si no se está ejecutando ningún otro método del objeto (ni siquiera el propio método en otro hilo).

Para identificar a los métodos selectores, se prefijan con la palabra clave Selector[1]. Estos métodos serán concurrentes con cualquier otro selector del objeto. Podrán ejecutarse siempre y cuando no se esté ejecutando un modificador. Por tanto, entrarán cuando no se esté ejecutando ningún método o los que se estén ejecutando sean selectores (incluso otra ejecución del mismo método selector en otro hilo).

```
Class <Nombre>
Isa {<Clase>}
Aggregation
  {<Nombre>:<Clase>}
Association
  {<Nombre>:<Clase>}
Methods
  método1();
  Selector método2();
  Selector método3();
Messages
  {<Nombre>({<Clase>});}
EndClass
```

En el fragmento anterior el metodo1 sólo entrará en ejecución si no se está ejecutando ningún método (metodo1 incluido). El metodo2 entrará siempre que no se esté ejecutando el metodo1 (independiente de si están ejecutándose o no metodo2 y metodo3), en ese caso quedará esperando a que finalice el método1. El metodo3 se comporta de la misma manera.

Con este mecanismo se sigue alcanzando el objetivo de que los objetos clientes no tengan que preocuparse de conocer aspectos de concurrencia de las clases que usan. Únicamente utilizarán los métodos de una manera uniforme. Por otro lado, el creador de la clase tampoco tiene que introducir ningún tipo de código especial para controlar la concurrencia en el cuerpo de los métodos. Simplemente hace unas decisiones de diseño de alto nivel en cuanto a la clasificación de los métodos. Esto basta para lograr una sincronización y ejecución concurrente correcta.

12.8.2.4 Sincronización de grano fino. Semáforos

El problema de la solución anterior es que no se alcanza el grado máximo de concurrencia posible. En algunos casos los métodos modificadores sí pueden trabajar concurrentemente entre sí. Normalmente la reducción de la complejidad y la sencillez de programación compensan sobradamente esto.

[1] También podría hacerse con facilidad que se identificara automáticamente el tipo de un método.

Semaphore

Para aquellos casos en los que se quiera maximizar la concurrencia, introduciendo código de sincronización específico dentro del cuerpo de los métodos, se proporciona una clase Semaphore, que implementa el comportamiento de un semáforo. Esta clase puede utilizarse para la sincronización explícita.

```
Class Semaphore
Isa Object
Methods
  set(Integer);
  wait();
  signal(): Bool;
  test(): Bool;
EndClass
```

- **set(Integer)**. Establece el valor del contador del semáforo.
- **wait()**. Si el contador el mayor que cero lo decrementa y retorna. Si es contador es menor o igual que cero el hilo queda suspendido.
- **signal(): Bool**. Incrementa el contador del semáforo en una unidad. Si hay algún hilo suspendido dentro del semáforo se reanuda uno de ellos. En caso contrario se incrementa el contador.
- **test(): Bool**. Si el contador el mayor que cero lo decrementa y retorna true. Si el contador es menor o igual que cero retorna false. Este método es de utilidad si sólo se desea entrar en la región crítica protegida por el semáforo en caso de que no esté ocupada, ya que en otro caso se realizarían otras tareas. Si se llama a test y el semáforo está ocupado, devuelve false. Es decir, si se hubiese llamado al wait el proceso se hubiese quedado congelado. De esta manera puede dedicarse a otras tareas y volver a mirar mas tarde. Si por el contrario el semáforo está libre realiza un wait y devuelve true. La razón por la que no se puede separar en una función de consulta del estado del semáforo y hacer después el wait es por que otro hilo se podría introducir en medio de las dos llamadas.

12.8.3 Soporte transitorio para Entrada/Salida

Para poder realizar cualquier actividad con la máquina es necesario disponer de un mecanismo que permita que la ejecución de los objetos en la máquina produzca algún efecto en los periféricos de salida. Para ello, se definen una serie de clases que permitan la interacción con el exterior. Se utilizará para ello el concepto de secuencia de entrada/salida (*stream*), con asociaciones a ficheros y a una consola de texto.

12.8.3.1 Stream

Un *stream* (secuencia) no es mas que un depósito en donde se guardan objetos y de donde se recuperan objetos. Es una clase abstracta que permite derivar el resto de las secuencias estándar de Carbayonia.

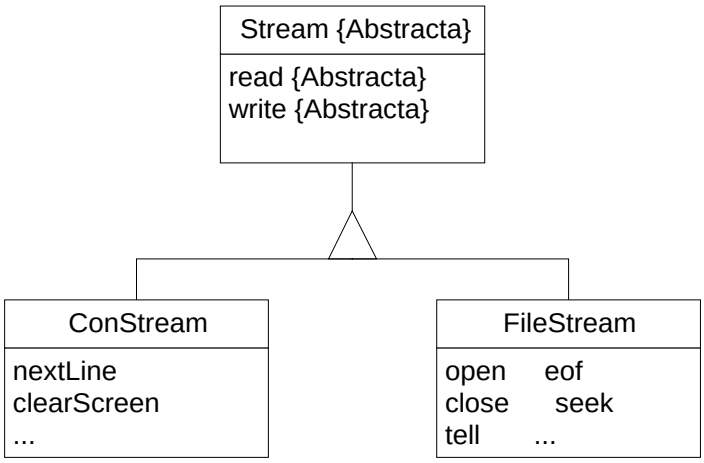


Figura 12.5 Jerarquía de secuencias para el soporte transitorio de Entrada/Salida

```
Class Stream
Isa Object
Methods
  write(Object);
  read(): Object;
EndClass
```

- **write(Object)**. Escribe un objeto en la secuencia.
- **read(): Object**. Lee el siguiente objeto de la secuencia.

Ambos métodos producen una excepción si ocurre algún error en el proceso de lectura o escritura. El método read, además, producirá una excepción si el objeto a recuperar del Stream no es compatible con la referencia utilizada (recuérdese que en el paso y retorno de valores se hace un amoldamiento dinámico que puede producir una excepción si los tipos no son compatibles).

12.8.3.2 ConStream

La clase conStream es una secuencia asociada con una consola, de tal manera que todo lo que se escriba en él va a la pantalla y todo lo que de ella se lea viene del teclado. Esta secuencia permite la comunicación entre las aplicaciones y el usuario.

```
Class ConStream
Isa Stream
Methods
  write(Object);
  read(): Object;
  nextLine();
  clearScreen();
EndClass
```

- **write(Object) y read(): Object**. Escribe el objeto en la pantalla y lo lee del teclado, respectivamente.
- **clearScreen()**. Limpia la pantalla.
- **nextLine()**. Envía a la consola un salto de línea.

12.8.3.3 FileStream

La clase FileStream representa una secuencia situada en un dispositivo de almacenamiento secundario, es decir, un fichero[1].

```
Class FileStream
Isa Stream
Methods
  write(Object);
  read():Object;
  open(String);
  close();
  eof():Bool;
  seek(Integer);
  tell():Integer;
EndClass
```

- **write(Object) y read(): Object**. Escribe y lee el objeto del fichero, respectivamente.
- **open(String)**. Abre la secuencia. Será el primer método a invocar antes de usar un FileStream. El parámetro de dicho método es una cadena que deberá identificar la posición del mismo en el almacenamiento secundario (nombre del fichero).
- **close()**. Cierra la secuencia. Al finalizar el trabajo con la secuencia deberá invocarse al método close para cerrar el fichero.
- **eof(): Bool**. El método Eof devuelve un valor booleano que indica si se está al final del fichero.
- **seek(Integer)**. Seek sirve para posicionarse dentro del fichero, e indica en que posición del mismo (contado en objetos a partir del inicio del fichero) debe realizarse la próxima operación de entrada o salida.
- **tell(): Integer**. Devuelve un entero que indica la posición actual en la cual se realizará la próxima lectura o escritura.

12.9 El fichero de clases. Representación compacta del lenguaje Carbayón

La unidad de información que se comunica a la máquina es la clase. Para describir una clase se utiliza el lenguaje Carbayón presentado anteriormente. Conceptualmente, todo lo que hay que hacer es proporcionarle a la máquina la descripción de la clase e indicarle el método

[1] La propiedad de la persistencia en el sistema operativo hará innecesario el concepto de fichero, los objetos serán persistentes por naturaleza (véase el capítulo 16). Estas secuencias se proporcionan transitoriamente por la persistencia, y para acceder a ficheros de sistemas convencionales.

inicial de la misma que comenzará la ejecución. Por tanto, la máquina debe tener un analizador del lenguaje Carbayón, para poder reconocer la descripción de las clases.

Para simplificar este analizador, en lugar de proporcionarle a la máquina una descripción de las clases en lenguaje Carbayonia, se convertirán previamente estas descripciones a un formato más compacto que resulte más sencillo de reconocer. Es decir, cada clase escrita en Carbayón generará un fichero de clases, que se alimentará a la máquina.

El formato de este fichero es simplemente una representación más compacta de la descripción en Carbayón, y con una estructura sintáctica más sencilla para su fácil reconocimiento. La descripción exacta del formato mediante una gramática BCNF se encuentra en el apéndice I.

Se indica el inicio y el final de las secciones de la declaración usando una representación corta de las secciones. Así por ejemplo, la sección de agregación se representa con la pareja AG/EAG, entre la cual se colocan las referencias agregadas. El resto de los elementos se corresponden con los del lenguaje, eliminando adornos sintácticos. Por ejemplo, la cabecera de un método se codifica colocando letras que identifican el nombre del método y a continuación separados por blancos la lista de parámetros (referencia tipo), etc.

El juego de instrucciones de comportamiento también sigue un proceso similar. Cada instrucción se codifica con un número, y a continuación se coloca la lista de sus parámetros: el objeto que hay que invocar, el nombre del método, y las referencias que son parámetros. Por ejemplo, la llamada obj.metodo(parametro) (call obj metodo parametro) se codificaría como 0 obj metodo parametro. Una sección al final de cada método indica la instrucción a la que hace referencia cada una de las etiquetas.

Además, se incluye en cada instrucción una información adicional que es la línea del código en Carbayón donde aparece la instrucción Carbayón original. Esto puede ser utilizado por un sistema de depuración, para poder ir trazando la ejecución del código en la máquina Carbayonia y a la vez ver las instrucciones correspondientes en el código fuente, con sus comentarios, etc. A continuación se muestra un ejemplo de un programa en Carbayón y el formato del fichero de clases correspondiente:

```
Class Tarchivo

Aggregation
  nombre:String;
  consola:ConStream;

Methods

setNombre(nombreParam:String
)
  Code
    nombre.Set(nombreParam);
    Exit;
  Endcode

getNombre():String
  Refs
   devolucion:String;
  Code
    New devolucion;
    devolucion.Set(nombre);
```

```
LEOOxCLASS TARCHIVO
H OBJECT
 EH
AG
NOMBRE STRING
CONSOLA CONSTREAM
EAG

F M E TARCHIVO::SETNOMBRE
NOMBREPARAM STRING
R VOID
CD 229
0 NOMBRE SET NOMBREPARAM R VOID
10
12 11
EC
LB
ELB

F M E TARCHIVO::GETNOMBRE
R STRING
R
DEVOLUCION STRING
```

Assign rr,devolucion; **Exit**; **Endcode**	ER CD 408 2 DEVOLUCION 18 0 DEVOLUCION SET NOMBRE R VOID 19 3 RR DEVOLUCION 20 12 21 EC LB ELB C:\FUENTE\archivo.asm ENDCLASS

En cualquier caso, este formato del fichero de clases tiene el mismo nivel de descripción semántica que el lenguaje Carbayón. Es simplemente una sintaxis alternativa para el lenguaje, más sencilla de reconocer. Es decir, se mantiene el alto nivel de la interfaz de la máquina.

12.9.1 Otras alternativas de descripción compacta. Protección frente a código malicioso

Una posibilidad interesante a explorar es utilizar otro tipo de descripción compacta del lenguaje Carbayón, como la que utiliza el sistema Juice [KF96] para representar de manera compacta los programas del lenguaje Oberon [RW92]. En lugar de usar una representación compacta del código fuente del programa, se utiliza una representación abstracta del programa, el árbol semántico del mismo. La ventaja sobre el procedimiento anterior es que por su propia definición, todo árbol semántico representa un programa válido. De esta manera es imposible introducir en la máquina código máquina erróneo. Esto es importante en un entorno de red con objetos móviles en los que una máquina tendrá que ejecutar objetos cuya procedencia no puede verificar. Estos objetos no pueden corromper o detener la máquina debido a errores (malintencionados o no) en su código.

En cualquier caso, la preocupación de introducir este código malicioso es mucho mayor en máquinas de más bajo nivel, como la de Java. La máquina de Java incorpora un verificador de *bytecodes* [LY97] para evitar esto. En el caso de Carbayonia, el prototipo que se ha implementado (véase el capítulo 13) hace una función similar a la del Juice. A partir del fichero de clases en formato compacto, reconstruye la estructura semántica del programa. Por tanto, puede detectar estructuras inválidas al cargar la clase.

12.10 Resumen

La máquina Carbayonia es una máquina que sigue la estructura de referencia de una máquina abstracta para dar soporte a un sistema integral de objetos. Implementa el modelo único de objetos del sistema. Es una máquina totalmente orientada a objetos en la que todas las operaciones se realizan sobre objetos y a través únicamente de referencias con identificadores de objetos.

El lenguaje de la máquina se denomina Carbayón. Permite declarar las características de las clases del modelo: relaciones de herencia, agregación y asociación; y métodos. El juego de instrucciones de comportamiento de los métodos permite definir las variables locales y el comportamiento del método.

En la definición de las instrucciones se tiene presente la filosofía de la elevación del nivel de abstracción. Por ejemplo, las referencias locales usadas se liberan automáticamente al

finalizar el método. Un tipo especial de variable local (cuyas instancias sólo se usan dentro del método) es gestionado directamente por la máquina: la instancia a la que apunta la variable se crea automáticamente al entrar en el método y se destruye al salir. Esto facilita la labor del programador. En esta línea se incluye también el tratamiento de excepciones.

El juego de instrucciones es reducido, en torno a quince instrucciones. Se agrupan en tres tipos. Las de gestión de objetos (a través de referencias) permiten crear nuevos objetos, asignar referencias, invocar métodos y eliminar objetos. Las de control de flujo permiten realizar saltos incondicionales (tratamiento de excepciones incluido) o condicionales en el normal flujo de ejecución de instrucciones de un método.

Existen una serie de clases básicas en el sistema. La clase Object es la raíz de la jerarquía de clases. Toda clase del sistema derivará de esta. En ella se incluyen los métodos que compartirán todos los objetos. También existen clases para números enteros y en coma flotante, valores booleanos, cadenas de caracteres y arrays de objetos. Es importante recalcar que estas clases son clases completamente normales y no se diferencian desde el punto de vista de su utilización de las clases de usuario. No existen instrucciones especiales como en otras máquinas para manejar estos tipos básicos. Se utilizan como los objetos normales mediante llamadas a sus métodos.

Para completar el sistema integral, parte de la funcionalidad del mismo se desarrolla en un sistema operativo, que requiere un estudio adicional profundo. Para permitir la experimentación con la máquina mientras se espera a que se desarrolle esta funcionalidad complementaria, se definen una serie de elementos transitorios que permitan trabajar completamente con la máquina en el área de concurrencia y entrada/salida.

El transitorio modelo de concurrencia se basa en la clasificación de los métodos. La llamada a métodos normales no crea un nuevo hilo de ejecución, la llamada a mensajes sí. La sincronización se resuelve indicando qué métodos son selectores (no modifican el estado) y permiten la concurrencia entre sí mismos y cuáles son modificadores, que deben funcionar en exclusión mutua con cualquier otro método. Se añade una clase semáforo por si se desea un control de grano más fino sobre la sincronización.

El mecanismo transitorio de entrada salida utiliza secuencias (*streams*). Se definen clases que son secuencias que permiten la entrada/salida con ficheros y con la consola.

Por último se utiliza una representación compacta de la descripción de las clases en lenguaje Carbayón (el fichero de clases) que es la que se proporciona a la máquina abstracta, en lugar de la notación extensa equivalente

Capítulo 13

IMPLEMENTACIÓN DEL PROTOTIPO DE LA MÁQUINA ABSTRACTA CARBAYONIA

En este capítulo se describe la filosofía de implementación del primer prototipo de la máquina abstracta Carbayonia (simulador por software de la máquina). Este prototipo fue desarrollado como proyecto fin de carrera de la Escuela Superior de Ingenieros Informáticos de la Universidad de Oviedo. Se puede consultar los detalles de la implementación en [Izq96].

El prototipo se ha desarrollado utilizando el lenguaje C++ y funciona en Windows NT y Windows95. También se ha portado al sistema operativo Linux, y debería funcionar también en otras versiones de Unix[1].

Como prototipo, no se hace hincapié en cuestiones de eficiencia. Se trata de desarrollar en un plazo razonable una implementación que permita comprobar en la práctica el funcionamiento de las ideas descritas en el diseño de la máquina. Esto también se refleja en la propia estructura interna del prototipo, en la que prima la sencillez, la claridad, y la facilidad de implementación frente a otras consideraciones. En la propia implementación se hace uso extensivo de las propiedades de la orientación a objetos, especialmente la herencia y el polimorfismo.

Además, este prototipo se usará como base para el desarrollo del resto de los elementos del sistema operativo, lo cual influirá en la propia máquina abstracta, que introducirá las modificaciones necesarias. Por tanto, hasta no finalizar la construcción del resto de los elementos básicos del sistema integral orientado a objetos no tiene mucho sentido preocuparse por cuestiones de eficiencia y sí facilitar la futura modificación del prototipo. En el apéndice D se hace una comparación preliminar de rendimiento con la máquina de Java, para estimar el rango de optimización posible en el prototipo.

13.1 Idea fundamental de la implementación: Reproducir con objetos los elementos de la máquina

La interfaz de la máquina de alto nivel permite utilizar los más variados diseños en la implementación. Como muestra, la idea fundamental en esta implementación sigue la característica tradicional de la orientación a objetos: imitar con objetos del programa todos los elementos de la máquina.

Clases, instancias, referencias, métodos e instrucciones de los métodos serán objetos del programa (TClass, TInstance, TRef, TMethod, TInstruction). Estos objetos se agruparán dentro de contenedores según su tipo: área de clases, área de instancias y área de referencias.

Estos objetos tendrán un comportamiento con métodos que realicen las funciones de los elementos de la máquina. Así por ejemplo, los objetos método tendrán una operación invokeMethod que implemente toda la semántica de la operación de invocación de ese método de la máquina.

[1] En general, funcionará en cualquier plataforma que disponga del producto gcc, el compilador de C++ de GNU.

Las relaciones existentes entre los elementos de la máquina se representan mediante relaciones entre los objetos correspondientes del programa (bien directamente por punteros o por valores de claves). Esto bien en forma de datos propios del C++ (agregación) o haciendo referencia a objetos externos. Así por ejemplo, las clases tienen una lista con los objetos método que representan sus métodos, una lista de sus ancestros (en forma de cadenas de caracteres), agregados y asociados (en forma de lista de referencias). Las referencias se componen de un nombre, un tipo y el identificador de la instancia a la que apuntan, etc.

13.1.1 Hilos de ejecución

Para implementar los hilos de ejecución se utiliza un mecanismo similar al de la Máquina-p [NAJ+76]. Cada objeto hilo (TThread) tiene una pila asociada compuesta por contextos. Un **contexto** representa el contexto de ejecución de un método. Cada vez que en un hilo se llama a un método se apila un contexto. El contexto proporciona los elementos dinámicos de ejecución de un método: las referencias e instancias locales, la instancia sobre la que actúa el método (this), un lugar donde dejar el valor de retorno, etc. Las instrucciones del método actuarán sobre la información del contexto. Al finalizar el método se elimina el contexto.

La máquina tiene incorporado un planificador que va alternando la ejecución de los métodos entre los diferentes hilos, con una política PEPS (Primero en Entrar, Primero en Salir). Hay que recordar que el soporte para la concurrencia es transitorio hasta que se defina el modelo de concurrencia definitivo en el sistema operativo. Mientras tanto, el objetivo de este soporte es simplemente que se pueda comenzar a experimentar con la ejecución concurrente de objetos. Por tanto, no preocupa que el planificador y la política de planificación sean fijas, ni que el mecanismo de planificación sea poco eficiente. Sin embargo, estos aspectos sí que tendrán que ser considerados en el diseño de la concurrencia en el sistema operativo.

Para implementar las excepciones, se hace que la pila de un hilo pueda entremezclar elementos para representar los manejadores de excepciones (mediante una clase abstracta TStackElement de la que derivan los contextos TContext y manejadores THandler).

13.1.2 Diagrama de clases general

El resultado es una estructura en tiempo de ejecución en la que los objetos del programa reproducen los elementos conceptuales del modelo de objetos y las relaciones que existen entre ellos según indican los programas escritos en el lenguaje Carbayón. Es decir, que básicamente se hace una representación con objetos en tiempo de ejecución de la estructura de los programas en Carbayonia. Esto se muestra en el siguiente diagrama general de clases del prototipo:

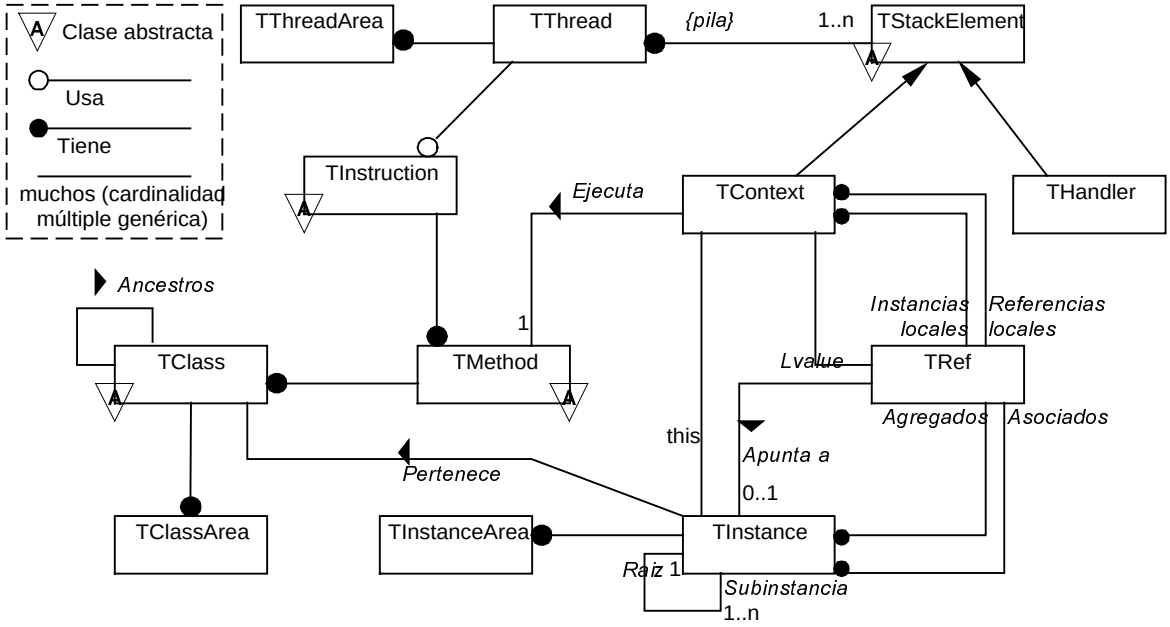


Figura 13.1 Diagrama de clases general del prototipo de la máquina Carbayonia.

13.1.3 Implementación de alto nivel

Otra alternativa podría ser utilizar estructuras de más bajo nivel para implementar la máquina. Por ejemplo simular zonas de memoria de tipo convencional que son gestionadas para almacenar los objetos. O bien utilizar una memoria en la que se almacenan en secuencia el código de las instrucciones (*bytecode*) y que se interpreta mediante un contador de programa que va recorriendo la memoria [Alv94] al estilo de un microprocesador convencional. Sin embargo, la interfaz de alto nivel permite que la implementación sea también de alto nivel, por ejemplo, las instrucciones de un método no se almacenan en un objeto memoria que contiene los *bytecodes* (código máquina) de las instrucciones. En su lugar se representan como una lista de objetos de tipo instrucción que las representan. La máquina reconstruye la estructura semántica de los programas en el lenguaje Carbayón con objetos de la implementación C++, a partir de los *bytecodes* del fichero de clases.

13.2 Implementación primitiva de elementos básicos

En una máquina totalmente orientada a objetos, todas los elementos que la componen se describen en términos de objetos. De esta manera las clases se definen en función de otras clases, estas a su vez en función de otras y así sucesivamente. Sin embargo, esta recursión debe finalizarse en algún momento. Deben existir algunas clases que no se definan en función de otras, su definición debe ser proporcionada directamente por la máquina.

13.2.1 Clases primitivas y clases de usuario

Es decir, la máquina debe proporcionar una implementación primitiva para ciertos elementos constituyentes de la máquina, que permita finalizar esa aparente recursión infinita. Estos elementos serán fundamentalmente las clases, con todo lo que ello conlleva: representación de las instancias y sus métodos. A las clases y métodos que son implementados directamente por la máquina se les denomina **clases y métodos primitivos**. Al resto, que se definen de manera normal a partir de otras clases se llaman **clases y métodos de usuario**.

En cualquier caso, el uso de estas clases primitivas no se debe distinguir del uso las clases de usuario. Esto quiere decir, que por ejemplo, cuando se invoca un método de una clase primitiva se hace de la misma manera que para una clase de usuario. La diferencia será que la máquina, con la clase primitiva invocará un método primitivo de la misma (implementado internamente en la máquina, en este caso en C++) que realice la función de ese método. Si es una clase de usuario, se invocará un método normal de la máquina, ejecutando las instrucciones de la máquina que describen ese método. Este mecanismo que no diferencia en la interfaz la utilización de los elementos primitivos, accediendo transparentemente a la implementación primitiva de los mismos es similar a la técnica utilizada en la máquina de Smalltalk [GR83].

	Primitivas	de Usuario
Clases	No necesitan ser definidas en términos de otras clases. La máquina les da soporte directo.	Se definen en términos de otras clases (agregados y asociaciones), que les dan soporte.
Métodos	Son implementados directamente por la máquina. No necesitan cuerpo.	Son implementados por instrucciones normales de la máquina. Necesitan el cuerpo con estas instrucciones

13.2.2 Selección de elementos primitivos: decisión de implementación

El hecho de que un elemento sea primitivo, no quiere decir que no pueda especificarse de manera completa como un elemento normal. Se podrían especificar por ejemplo los componentes de una clase primitiva (agregados) y sus métodos para facilitar la comprensión de la clase por el usuario. Aunque en realidad al ser implementada directamente por la máquina estos no existan verdaderamente. Lo importante es que el comportamiento implementado coincida con el especificado en la descripción.

Lo anterior puede ser de utilidad puesto que diferentes implementaciones podrían utilizar diferentes elementos primitivos. Por ejemplo, una implementación de la máquina para aplicaciones de cálculo científico podría implementar de manera primitiva una clase Ecuación, con un método resolverEcuación. La eficiencia de esta clase y este método sería mayor. Sin embargo, en otra implementación para otro tipo de aplicaciones la clase Ecuación sería una clase normal de usuario, con sus referencias y métodos escritos de la manera normal.

La selección de qué elementos serán primitivos o no es una decisión de implementación, que no afecta a las aplicaciones en absoluto. Esto sólo afectará al rendimiento de la máquina. Es un ejemplo más de las ventajas del uso uniforme de la OO junto con una interfaz de alto nivel. Por ejemplo, como se vio en el capítulo anterior, pueden proporcionarse objetos con funcionalidad del sistema operativo para el control de la concurrencia, como son los semáforos. Por razones de eficiencia, los semáforos son candidatos a ser proporcionados de manera primitiva.

13.2.3 Clases primitivas del prototipo

En el caso del prototipo se implementarán de manera primitiva todas las **clases básicas** del modelo único: Object, Bool, Integer, Float, String y Array, junto con todos sus métodos. También el resto de las clases de apoyo: Semaphore, Stream, ConStream y FileStream.

De hecho, en la descripción de las mismas ya se había previsto esta implementación primitiva, puesto que no se indicaron referencias componentes ni el cuerpo de los métodos. Esto no quiere decir que en otros prototipos alguna de estas clases no sea primitiva, o bien que otras clases adicionales se implementen como primitivas.

13.2.4 Resolución de la Implementación de elementos primitivos en el prototipo

Falta resolver cómo implementar estos elementos primitivos en el prototipo, manteniendo la uniformidad de uso externo. Para ello se hará un uso extensivo de la herencia y el polimorfismo, que se mostrará para el caso de las clases.

13.2.4.1 Uniformidad de uso. Objeto abstracto que representa una clase en general

Dado que desde el punto de vista externo clases primitivas y de usuario no se diferencian, se utilizará un objeto abstracto que represente cualquier tipo de clase con su comportamiento genérico. Este objeto es el objeto **TClass** descrito anteriormente. El resto de los componentes de la máquina trabajarán con este TClass abstracto, con lo que no diferenciarán si es una clase primitiva o de usuario.

13.2.4.2 Clases derivadas para representar las clases primitivas y de usuario

De este objeto abstracto heredarán otros que representen los dos tipos de clases: primitivas (un TCxxx por cada clase primitiva) y de usuario. Cada una de éstas redefinirá los métodos de la clase abstracta adecuándolos a su comportamiento particular, fundamentalmente dos. Por un lado el que permite la creación de instancias de la clase, que se realizará de manera específica en función de qué tipo de clase sea. Por ejemplo, en el caso de TCInteger se creará un objeto que represente un entero con la estructura interna que desee TCInteger (y conocida sólo por ella). El otro método a redefinir es el que permite preguntar a una clase si tiene un determinado método o no, etc.

Las clases de usuario se representan con **TCUserClass**. La estructura interna que gestione TCUserClass tendrá que representar la información de una clase de usuario normal: la lista de agregados y asociados, nombres y descripción de sus métodos, etc.

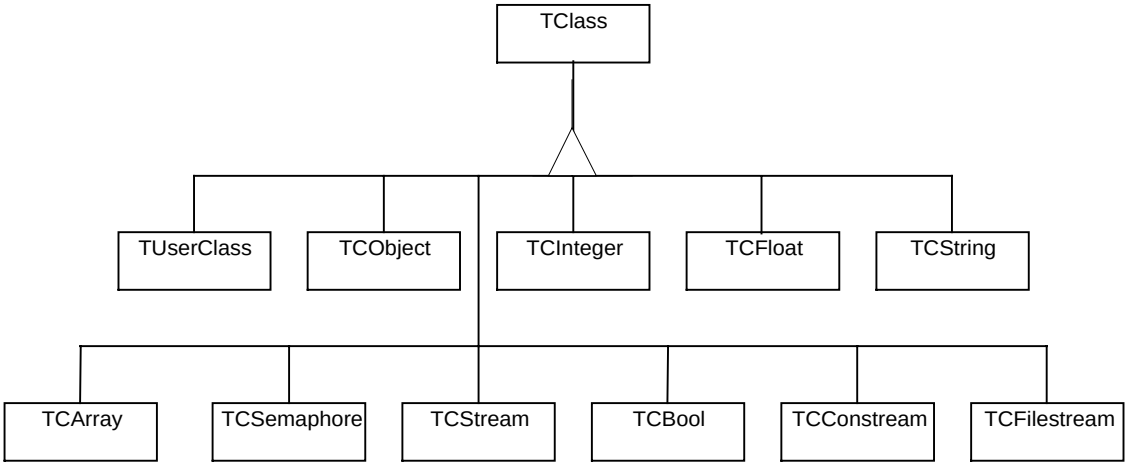


Figura 13.2 Jerarquía de TClass para las clases con implementación primitiva.

Lo importante es que el polimorfismo permite que el resto del sistema (y tampoco el usuario externo) no tenga que preocuparse si trata con una clase primitiva o básica. En función de qué clase sea, se invocará la operación de la TCxxx adecuada. Cada TCxxx encapsula la estructura interna mediante la que se soporta la clase xxx (clases primitivas o una clase de usuario).

13.2.4.3 Instancias y métodos

Dado que las instancias también se representan mediante objetos, se necesita una jerarquía similar para ellas. Existe una clase abstracta que representa una instancia en general (**TInstance**). Cada TCxxx para las clases tiene que tener su TIxxx en las instancias, tanto para las instancias de clases primitivas como para las de usuario. Así TCInteger trabajará con su TIInteger, TCUserClass con TIUserInstance, etc.

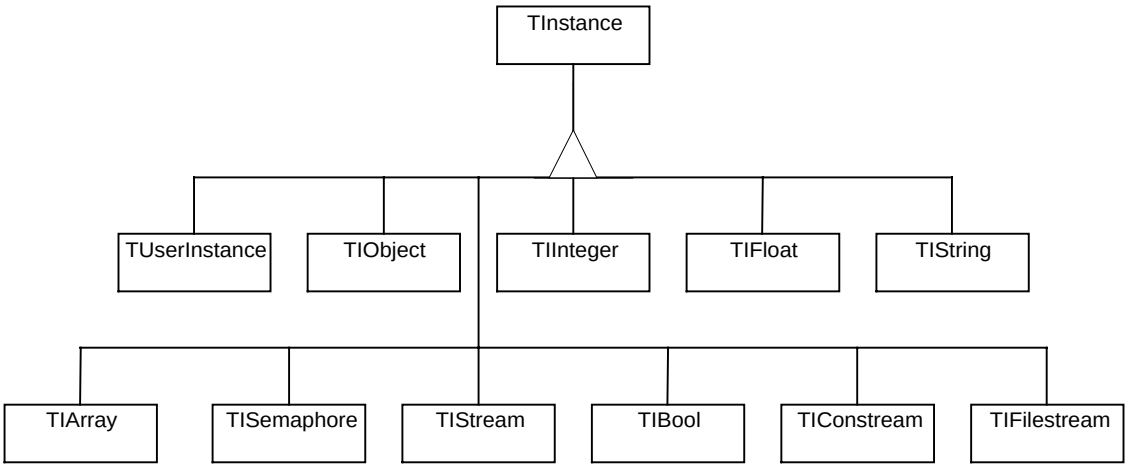


Figura 13.3 Jerarquía de TInstance para las instancias con implementación primitiva.

El mismo procedimiento se aplica a los métodos. De la clase abstracta TMethod se derivarán los métodos de las clases primitivas y (la clase) para un método de usuario. En este caso existirá una clase para cada clase primitiva y de usuario. De cada clase primitiva derivan los métodos de la misma: de TMInteger derivan TMIntegerAdd, TMIntegerSub, etc.

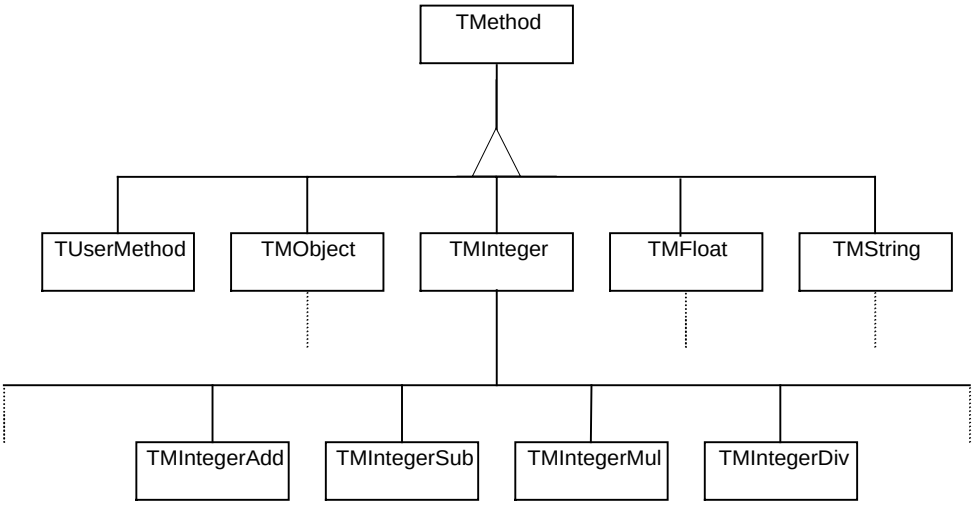


Figura 13.4 Jerarquía de TInstance para las instancias con implementación primitiva.

Aquí se ve más fácilmente cómo se accede a la implementación primitiva de manera transparente. Como parte de la ejecución de un método, existirá una invocación de un método sobre un objeto. El simulador accederá a la instancia y a la clase a través de la referencia. La clase indicará el objeto método que representa al método invocado. Una vez localizado el método se llamará a la operación invokeMethod del mismo. Este proceso es siempre el mismo independientemente de que el método sea primitivo o de usuario. El resto del simulador sólo utiliza objetos abstractos TMethod, con lo que el uso de métodos primitivos es totalmente transparente.

En este momento es donde entra en acción el polimorfismo. Si el método era primitivo, el tipo verdadero del objeto método devuelto será el de un método primitivo. Por ejemplo TMIntegerAdd e invokeMethod llamará a la implementación de este método en TMIntegerAdd, que realizará de manera directa la suma de enteros utilizando la representación interna primitiva de los mismos en la máquina. En caso de ser un método de usuario, se llamaría polimórficamente al invokeMethod de un TMUserMethod, que desencadenaría la ejecución normal de un método mediante la simulación del funcionamiento de las instrucciones de la máquina que lo componen.

13.2.4.4 Instrucciones de comportamiento

Las instrucciones de comportamiento para el cuerpo de los métodos también utilizan el mismo mecanismo, aunque aquí se podría considerar que todas estas instrucciones son primitivas, puesto que son las instrucciones de la máquina. Una clase abstracta **TInstruction** encapsula para el resto del sistema el funcionamiento de las instrucciones, con una operación exec que provoca la ejecución de la instrucción. De ésta derivan todas las instrucciones de comportamiento TInstrNew, TInstrHandler, etc. Cada una redefinirá el método exec para que realice con las estructuras de representación de la ejecución (contextos, etc.) las operaciones necesarias de acuerdo con la definición de la instrucción. Por ejemplo, el exec de la instrucción TInstrJmp actualiza el valor del contador de método[1] (MC, *Method Counter*) para que apunte a la instrucción indicada.

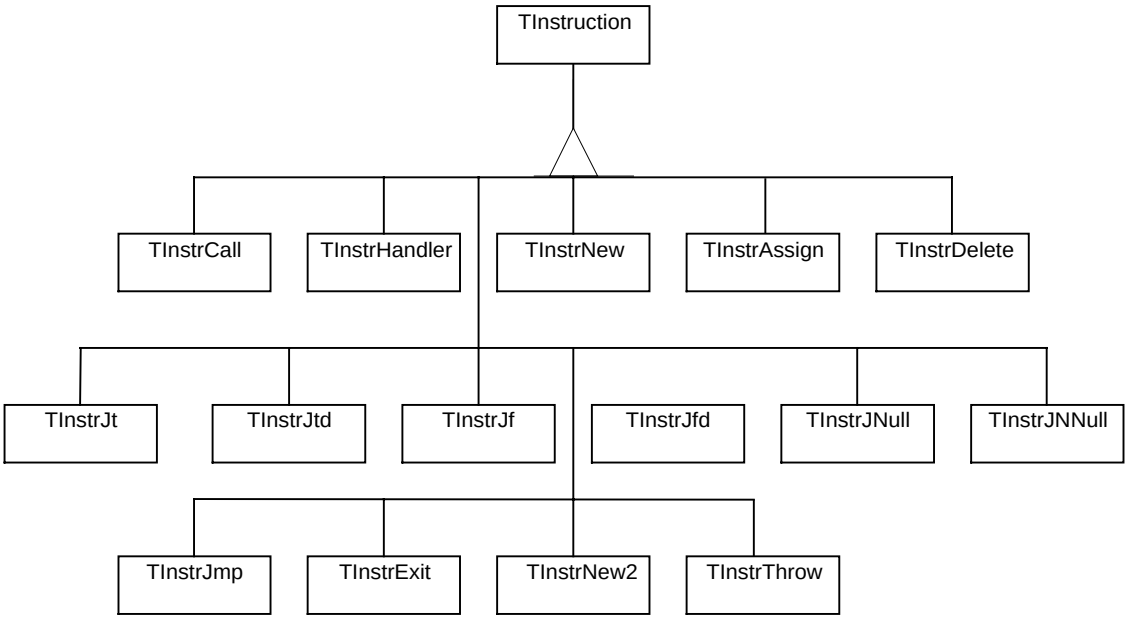


Figura 13.5 Jerarquía de TMethod para los métodos con implementación primitiva.

Este tipo de implementación con este grado de encapsulamiento permite añadir fácilmente clases derivadas sin que el resto del simulador quede afectado. Esto implica que es muy sencillo añadir nuevas instrucciones a la máquina, nuevas clases y métodos primitivos, etc. Basta con añadir una nueva subclase que derive de la clase abstracta que implemente la nueva instrucción, clase, etc. y automáticamente el resto del sistema la usa sin necesidad de otras modificaciones.

13.2.4.5 Ventajas de la uniformidad en el uso externo con implementación primitiva de elementos

Esta implementación primitiva de elementos manteniendo la uniformidad de uso externo de los mismos contrasta con máquinas como la máquina virtual de Java [LY97] en la que existen unos elementos primitivos con operaciones especiales. Los objetos de usuario se construyen en última instancia a partir de estos métodos primitivos. Esto provoca una mezcla de niveles de abstracción, al trabajar de manera distinta según se trate con los datos primitivos o con objetos.

[1] Apunta a la siguiente instrucción a ejecutar dentro del método

Por otro lado se pierde versatilidad de implementación. El implementador no puede elegir cuáles son los elementos primitivos que interesa incorporar en la máquina. Éstos ya vienen predefinidos por la propia definición de la misma. Se pierde una oportunidad de lograr una mayor eficiencia implementando de manera primitiva ciertos elementos en función del área de aplicación (sin problemas de modificación de aplicaciones).

13.3 Entorno integrado de desarrollo

A la máquina se le proporciona una clase en formato del fichero de clases. Para convertir un programa en Carbayón al formato de fichero de clases es necesario desarrollar un traductor (compilador) del mismo. En lugar de desarrollar un simple traductor del lenguaje Carbayón se ha desarrollado un entorno de desarrollo integrado (IDE, *Integrated Development Environment*).

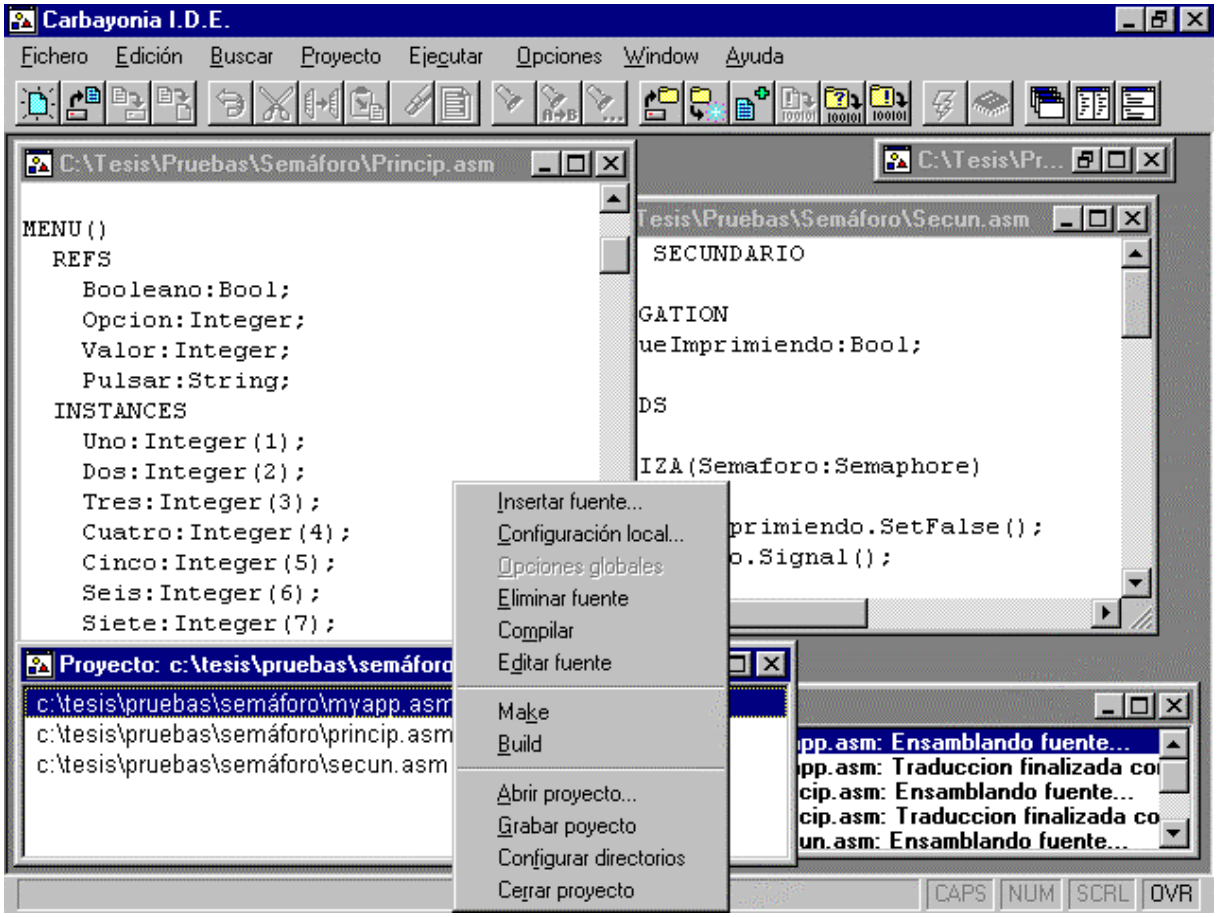


Figura 13.6 Entorno de desarrollo integrado del lenguaje Carbayón para el prototipo de la máquina abstracta Carbayonia.

De esta manera, además de generar el fichero de clases de una clase en Carbayón, se proporcionan las comodidades de un entorno integrado: edición, compilación y montaje centralizados, gestión de proyectos, etc.

Actualmente únicamente se genera el fichero de clases que es alimentado manualmente a la máquina. En un futuro se prevé integrar el entorno con la máquina abstracta para permitir la ejecución de los programas y su depuración directamente desde la ventana de edición del entorno.

El manual de usuario del entorno integrado de desarrollo se encuentra en el apéndice A.

13.4 Resumen

En la implementación del prototipo de Carbayonia se aprovecha el alto nivel de la interfaz de la máquina. Esto permite realizar una implementación interna al estilo tradicional de la OO en la que se utiliza una representación en la que los elementos de la máquina tienen objetos análogos en el programa.

Se hace uso de clases abstractas, herencia y polimorfismo para lograr una encapsulamiento de los elementos de la máquina. Mediante clases abstractas se representa el comportamiento genérico de una clase, instancia, método e instrucción. Clases derivadas implementan de manera concreta clases, instancias, métodos e instrucciones específicos. En el resto del sistema se usa el elemento abstracto que mediante el polimorfismo accederá transparentemente a la implementación concreta de cada instrucción, clase, etc.

Esto tiene dos ventajas. Por un lado permite implementar elementos primitivos de la máquina manteniendo el mismo uso externo que los no primitivos. Así se implementan de manera primitiva las clases, métodos e instancias de las clases básicas del modelo único. Se mantiene la uniformidad de uso. Por ejemplo, cuando un usuario llame a un método, el polimorfismo accederá a la implementación primitiva del método si es de una clase primitiva, si no accederá a la implementación normal del mismo (interpretación de sus instrucciones). Esta uniformidad permite elegir de manera arbitraria qué clases se implementarán como primitivas en función del área de aplicación, para mejorar la eficiencia.

Por otro lado, se pueden añadir fácilmente nuevos elementos, como nuevas instrucciones, clases primitivas, etc. simplemente aportando el código para la clase derivada del nuevo elemento. El polimorfismo hace que sin modificación el resto del sistema utilice los nuevos elementos.

Capítulo 14

MECANISMOS DE EXTENSIÓN DE LA MÁQUINA ABSTRACTA. REFLECTIVIDAD

La máquina abstracta se puede considerar como un micronúcleo para el sistema operativo que proporciona el soporte de objetos básico. El sistema operativo se encargará de complementar este soporte con todo lo necesario para aportar al sistema integral el resto de las características que se consideren necesarias. Las más importantes son la persistencia, la concurrencia, la distribución y la seguridad.

14.1 Mecanismos para proporcionar la funcionalidad del sistema operativo

En general, la funcionalidad asignada al sistema operativo será proporcionada mediante objetos normales de usuario. Sin embargo, en algunos casos será necesario modificar de ciertas maneras la máquina, incluyendo el soporte que sea necesario para la implementación de esta funcionalidad.

En este capítulo se analizan los diferentes mecanismos que pueden utilizarse para lograr este objetivo, destacando entre ellos la reflectividad, que se había identificado (véase el capítulo 14) como un elemento fundamental para lograr la uniformidad en la orientación a objetos:

- Modificación interna de la máquina
- Extensión de la funcionalidad de las clases básicas
- Colaboración con el funcionamiento de la máquina. Reflectividad.

Estos mecanismos podrán usarse por separado o, lo que es más probable, en combinación. Y casi siempre junto con objetos normales de usuario que proporcionen la funcionalidad restante.

Uniformidad en la orientación a objetos

Como en todo el sistema integral, sigue vigente la premisa de mantener la uniformidad dentro del sistema. Cualquier mecanismo de extensión que se utilice no debe romper la uniformidad de trabajo dentro de los mismos conceptos OO.

14.1.1 Modificación de la máquina

Se trata de modificar el funcionamiento o la estructura de la máquina para hacer posible o facilitar la implementación de una característica del sistema operativo. Se trataría de modificaciones que cambiaran la interfaz de alto nivel y que, por tanto, conllevan modificaciones en la implementación interna de la máquina para soportar ese cambio. Las modificaciones en la implementación sin cambio de la interfaz (por ejemplo aumento de eficiencia) no se consideran parte de este apartado. De hecho son totalmente transparentes.

La adición de un elemento adicional a los elementos básicos de la máquina sería uno de estos casos. Por ejemplo, puede decidirse que la máquina incluya soporte directo para hilos y concurrencia. De hecho, en el prototipo de Carbayonia (véase el capítulo 13) se ha incluido de manera transitoria este soporte. Esto requiere modificaciones en el lenguaje. Se añaden calificadores para distinguir los métodos selectores, una cláusula para identificar los mensajes, etc. Internamente se añaden en la implementación estructuras para soportar la concurrencia: hilos, planificación de hilos, etc.

Otro ejemplo de esto sería la adición de una nueva instrucción (o cambiar la semántica de una ya existente) al juego de instrucciones de la máquina. Habría que modificar la implementación de la máquina para incorporar la nueva instrucción. Un ejemplo podría ser instrucciones añadidas para dar soporte a la seguridad dentro de la máquina [Día96].

14.1.2 Extensión de la funcionalidad de las clases básicas

La funcionalidad de las clases básicas puede ser modificada, añadiendo nuevos atributos y métodos a las mismas. Por ejemplo, todos los atributos que se añadan (en el nivel conceptual) a la clase básica Object son heredados por todos los objetos del sistema. Esto permitiría añadir un atributo Nombre de tipo cadena de caracteres. De esta manera, todos los objetos del sistema tendrían la posibilidad de disponer de un nombre simbólico. Dado que las clases se construyen a partir de las clases básicas, modificaciones en éstas afectan al sistema.

Hay que tener en cuenta que cambios en las clases (aunque sean básicas) sólo afectan en la máquina cuando estas están implementadas de manera primitiva. Por ejemplo, dado que la clase Object es una clase primitiva en el prototipo de Carbayonia, cambios en la misma requieren cambios en la implementación. Desde el punto de vista de utilización, simplemente aparecen nuevas operaciones a disposición de los objetos, no cambia la manera de trabajar con los mismos.

El caso de añadir nuevas clases de usuario al sistema que proporcionen funcionalidad del SO, como por ejemplo para un objeto que implemente un servicio de denominación [Alv96b], no se consideran parte de este apartado. Esto no afecta a todos los objetos, simplemente es la manera normal de trabajar en un sistema orientado a objetos. De hecho, esta será la manera más común que utilice el sistema operativo para proporcionar sus funcionalidades.

14.1.3 Colaboración con el funcionamiento de la máquina. Reflectividad

Una posibilidad es permitir que los objetos del sistema operativo (objetos normales de usuario[1]) puedan colaborar con la máquina para extender el funcionamiento de la misma.

Por ejemplo, para proporcionar capacidades similares a la memoria virtual [Dei90] de sistemas operativos convencionales podría hacerse que cuando la máquina se quede sin espacio en el área de instancias para un objeto, se comunique con un objeto de usuario del SO. Este objeto realizaría el intercambio a disco de una serie de objetos liberando espacio en el área de instancias. Esta filosofía puede utilizarse para lograr la persistencia de los objetos [Alv96b].

Sin embargo, el problema es cómo realizar la comunicación entre la máquina y los objetos de usuario sin romper la uniformidad en la OO. La solución es dotar a la máquina de una arquitectura reflectiva OO [Mae87]. La reflectividad hace que la máquina sea expuesta a los objetos de usuario en forma de un conjunto de objetos normales, en la que los objetos de la

[1] Objetos normales o de usuario son aquellos cuya existencia es soportada por la máquina abstracta, y son creados por los usuarios mediante la interfaz de la máquina (el lenguaje Carbayón).

máquina no se diferencian de los objetos normales. Así, los objetos de la máquina pueden usar los objetos normales, y viceversa, usando el mismo marco de la OO común en el sistema.

Cuando un objeto de la máquina no pueda continuar por sí mismo y necesite la colaboración de un objeto de usuario (del SO) para que realice una tarea, simplemente lo llama usando la invocación de método normal. Recíprocamente, los objetos de usuario interactuarán con los objetos de la máquina mediante la invocación normal de métodos para solucionar estas tareas (como la persistencia). Éstos ni siquiera se dan cuenta de que los objetos con que interactúan son de la máquina, para ellos son simplemente otros objetos más.

El sistema se puede ver como un **conjunto de objetos homogéneos** que interactúan entre sí, compartiendo las mismas características (el mismo modelo de objetos). Algunos de ellos serán objetos de la máquina, otros objetos normales de usuario. De estos últimos, algunos implementarán funcionalidad del sistema operativo. Sin embargo, el origen de un objeto es transparente para todos los demás y para sí mismo.

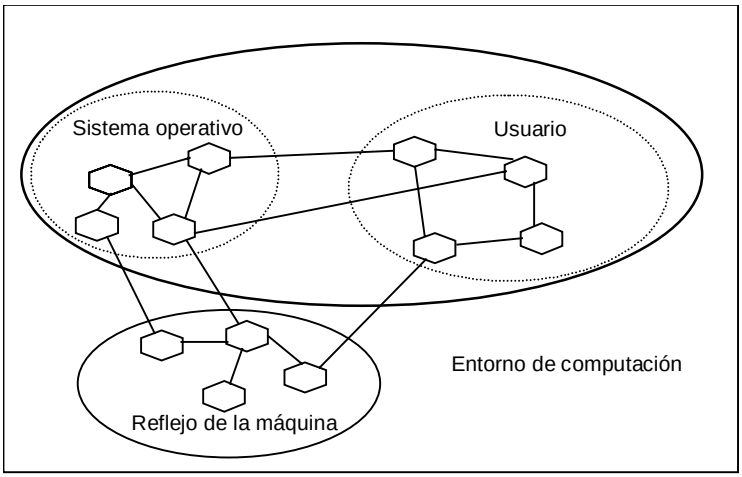


Figura 14.1 Espacio de objetos homogéneo formado por la unificación de los objetos de la máquina con los del sistema operativo y los del usuario.

La reflectividad está reconocida como una técnica importante para lograr flexibilidad (extensibilidad, adaptabilidad, etc.) en los núcleos de los sistemas operativos [Cah96, GW96]. Cuando se utiliza en un sistema integral orientado a objetos contribuye a la uniformidad, como se ve. Debido a la importancia de la flexibilidad para el sistema integral los siguientes apartados se dedican a un estudio más profundo de la reflectividad para hacer una selección de qué tipo de reflectividad y qué alternativas existen para implantarla en la máquina abstracta.

14.2 Reflectividad

El concepto de reflectividad (*reflectivity*) fue introducido en el campo de los lenguajes de programación por Brian Smith [Smi82], desarrollándolo luego en una versión del lenguaje de programación LISP [Smi84]. Posteriormente se aplicó para sistemas orientados a objetos por Patti Maes [Mae87]. El significado concreto de la reflectividad varía según los autores y su ámbito de aplicación. La descripción que viene a continuación es un compendio resumido de los conceptos y terminología de la reflectividad, siempre desde el punto de vista de su aplicación al objetivo de la extensión transparente de la máquina abstracta. La terminología y los conceptos exactos pueden diferir de los usados por otros autores. Dentro de lo posible se intenta mencionar también terminología alternativa.

14.2.1 Sistema base y meta-sistema

Un sistema de computación puede verse conceptualmente como compuesto por dos niveles: el nivel base (o **sistema base**) en el que funcionan los elementos de computación definidos en el sistema y el meta-nivel (o **meta-sistema**[1], o entorno en tiempo de ejecución) que proporciona la infraestructura y los mecanismos que hacen funcionar a los elementos del nivel base de acuerdo con su definición. Podría decirse que el meta-nivel es el agente que da vida al nivel base, permitiendo su existencia.

Un ejemplo puede ser un sistema operativo y los procesos de usuario. Los procesos de usuario forman el nivel base. El sistema operativo es el meta-nivel que da soporte a los procesos de usuario, permitiendo su existencia. Desde el punto de vista de los lenguajes de programación, puede considerarse el nivel base como un programa escrito en un lenguaje de programación y el meta-nivel como el intérprete del lenguaje que da vida a ese programa. Este ejemplo también se adapta a una máquina abstracta orientada a objetos (o a un sistema OO en general). La máquina es el meta-sistema que da soporte a los objetos que forman el sistema base. O visto de otra manera, la máquina es un intérprete de los programas escritos en el lenguaje de la máquina (juego de instrucciones).

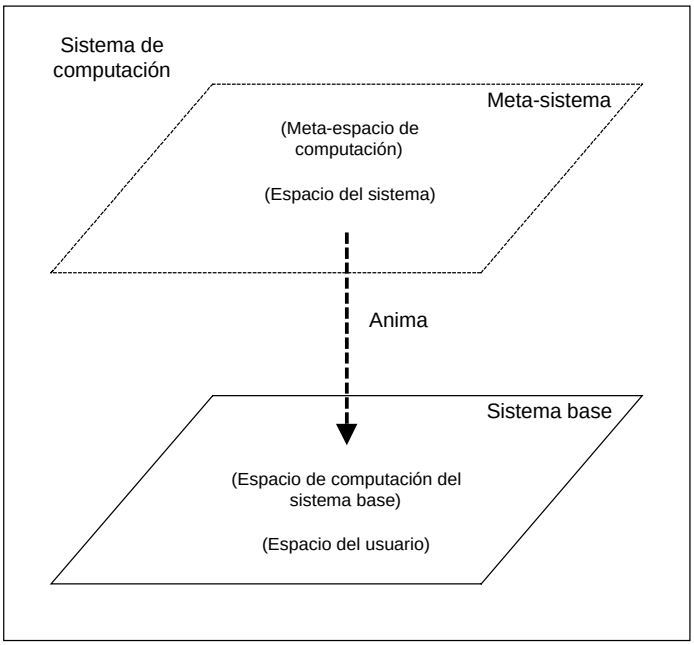


Figura 14.2 Sistema base y su meta-sistema.

Existen dos espacios de computación que suelen estar diferenciados: el **espacio de computación del sistema base**, en el que interactúan los elementos del sistema base y el **meta-espacio de computación** donde trabaja el meta-sistema[2].

Puede considerarse que el meta-sistema proporciona de manera implícita ciertas propiedades a los elementos del sistema base. Por ejemplo, el sistema operativo proporciona implícitamente a los procesos la propiedad de su ejecución concurrente.

Desde otro punto de vista puede considerarse que el sistema base tiene una determinada definición que es interpretada por el meta-sistema. El ejemplo más claro es el de un sistema

[1] Puesto que es un sistema que se utiliza para animar otro sistema

[2] Puesto que el usuario final trabajará con los elementos del sistema base, también se denominan **espacio del usuario** y (meta-)**espacio del sistema**

definido mediante un programa en un lenguaje de programación que es interpretado por un meta-sistema: el intérprete del lenguaje de programación.

14.2.1.1 Torre de meta-sistemas

También puede aplicarse el mismo proceso al meta-sistema. El meta-sistema será un programa que estará interpretado por su meta-meta-sistema. El meta-meta-sistema también tendrá su meta-meta-meta-sistema y así sucesivamente, formando **una torre infinita de meta-sistemas** o intérpretes [Smi82]. A efectos prácticos, en general es más sencillo limitarla a un solo nivel base con su meta-nivel.

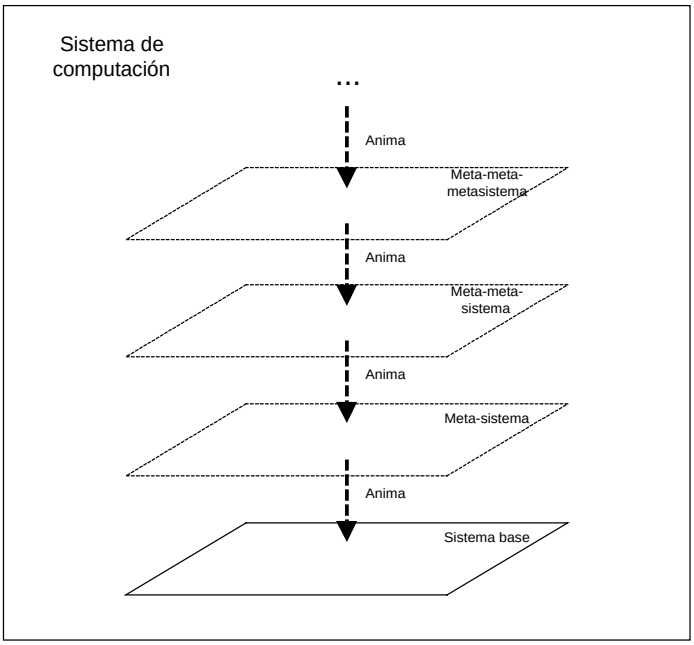


Figura 14.3 Torre de regresión de meta-sistemas.

14.2.2 Concepto de reflectividad

La **reflectividad** puede describirse como *la propiedad que tiene un sistema de razonar acerca de sí mismo y actuar sobre sí mismo cambiando su propio comportamiento*. Es decir, la propiedad por la que el sistema base puede acceder a su meta-sistema, conocer información acerca de sí mismo, e incluso modificar el meta-sistema (cambiando así su propio comportamiento, pues el meta-sistema influye en la propia constitución del sistema).

Ejemplos limitados de reflectividad existen en muchos sistemas. Por ejemplo, el operador `sizeof` del lenguaje C permite a un programa conocer información acerca de sí mismo. Las llamadas al sistema Unix que permiten a un proceso elegir su política de planificación también son un ejemplo limitado de reflectividad. Un sistema se denomina **sistema reflectivo** cuando soporta la reflectividad de manera organizada.

De esta manera, las propiedades implícitas que proporciona el meta-sistema al sistema base se hacen explícitas para el propio sistema base. La reflectividad consistirá en poder conocer y cambiar la propia definición del sistema base por un lado y en poder conocer y cambiar el intérprete de esa definición.

14.2.3 Modelo del sistema (reflejo)

La manipulación y modificación directa del meta-sistema normalmente no se permite puesto que sería muy difícil de controlar y podría provocar inconsistencias en el sistema base

[YW89] (por ejemplo si un proceso pudiera modificar los valores de los registros de su contexto de ejecución). En su lugar se manipula o modifica una representación (modelo del meta-sistema[1] o **meta-modelo)** adecuada que es la que se muestra al sistema base.

Este modelo que se muestra al sistema base es en el que se ve reflejado el sistema base. Es decir, el sistema base, cuando "mira" al meta-sistema que le da vida para conocer información acerca de sí mismo, ve su "**reflejo**[2]" (la representación que el meta-sistema usa para mostrarse al sistema base). De ahí el nombre de reflectividad[3] para esta propiedad.

También puede considerarse el término "reflejo" para el modelo del meta-sistema en el sentido de que el meta-sistema se muestra de una manera indirecta, reflejada a través del modelo.

14.2.3.1 Propiedades del modelo

El modelo del sistema que se utilice en un sistema reflectivo debe reunir una serie de propiedades [YW89]:

- **Nivel de abstracción adecuado**. El modelo debe tener el nivel de abstracción adecuado de acuerdo con el uso de la reflectividad que se pretenda que pueda hacer el nivel base.
- **Cosificación (*reification*)**. El sistema base debe poder manipular el modelo del meta-sistema (**meta-interacción**). La operación que permite esto se denomina **cosificación**[4] (*reification*), pues transforma en entidades tangibles para el sistema base (cosifica) lo que normalmente le es intangible (el meta-sistema).

[1] Aunque no se haga referencia explícita, se entiende que cualquier mención al meta-sistema cuando se cambia o manipula es siempre mediante su modelo o representación.
[2] El reflejo es el resultado de la reflexión (la acción de reflejar). En inglés se denominan con la misma palabra *reflection.*
[3] Otra traducción podría ser reflexividad, entendiendo esta propiedad como la capacidad de reflejar imágenes. También se justifica desde el punto de vista del acceso del sistema a su propia representación, de manera reflexiva.
[4] Otra traducción posible es concretización, puesto que se concreta algo que era vago e impreciso como el meta-sistema.

- **Reflexión (*reflection*)**. Los cambios efectuados en el modelo del meta-sistema deben afectar al subsiguiente comportamiento del sistema. Es decir, los cambios en el meta-sistema se reflejan en el propio sistema. Existe una conexión causal entre el meta-sistema y el sistema.

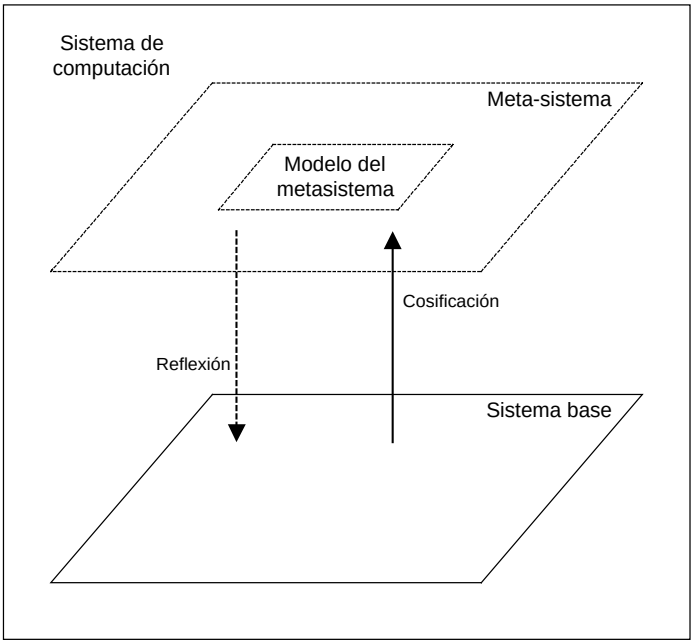


Figura 14.4 Interacción entre el sistema base y el meta-sistema mediante un modelo del meta-sistema.

14.2.4 Uniformidad sistema / meta-sistema: Orientación a objetos

En el caso de un sistema orientado a objetos, el paradigma mediante el que se estructura el sistema es la orientación a objetos. Para introducir la reflectividad en el sistema de manera uniforme, lo más interesante es que el modelo del meta-sistema esté descrito utilizando el mismo paradigma OO que el propio sistema. De este modo, tanto el modelo, como las operaciones de cosificación y reflexión pueden ser utilizados de manera uniforme dentro del mismo paradigma.

En el nivel base se encontrarán los **objetos normales** del sistema y en el meta-nivel se encuentran los objetos que dan soporte a los objetos base. Los objetos del meta-nivel se denominan **meta-objetos**.

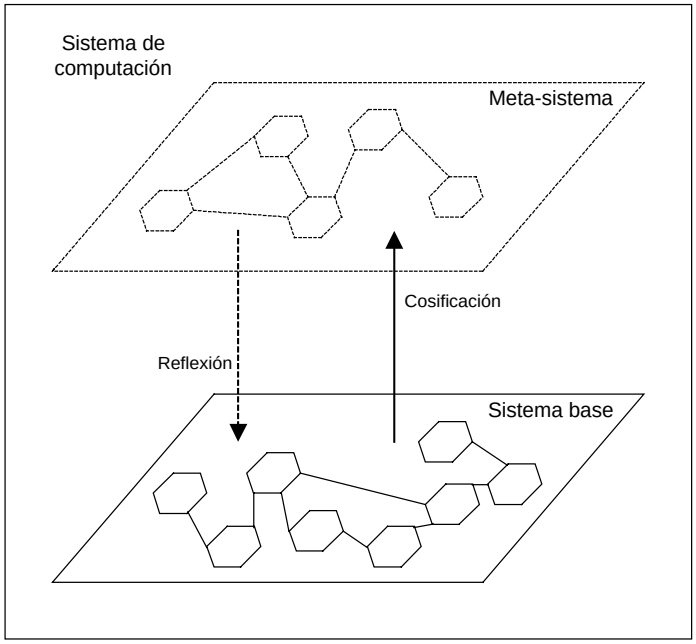


Figura 14.5 Descripción del meta-sistema mediante un modelo de objetos.

14.2.5 Meta-circularidad

Cuando el propio meta-sistema (o modelo del mismo) utiliza el mismo lenguaje de descripción que el sistema base, se tiene un **sistema meta-circular**. Un ejemplo de meta-circularidad es los autocompiladores [Cue95]. Éstos son intérpretes o compiladores de un lenguaje escritos en el propio lenguaje que van a interpretar, utilizados en el área de procesadores de lenguajes

14.2.6 Meta-interfaz y protocolo de meta-objeto (MOP, *Meta-Object Protocol*)

El conjunto de operaciones que ofrece el meta-sistema para controlar su funcionamiento se denomina **meta-interfaz**. En el caso de un meta-modelo orientado a objetos la interfaz se suele denominar **protocolo de meta-objeto** [KRB91] (MOP, *Meta-Object Protocol*).

14.2.7 Meta-interacción explícita e implícita

Pueden distinguirse dos tipos de meta-interacción que hacen que el meta-sistema entre en funcionamiento:

- **Meta-interacción explícita**. Los objetos del nivel base invocan directamente las operaciones de la meta-interfaz del meta-modelo. Por ejemplo, cuando un proceso cambia su política de planificación.
- **Meta-interacción implícita**. El meta-sistema entra en funcionamiento de manera implícita debido a algún evento o acción causado por el funcionamiento normal del sistema base. En una máquina abstracta existe meta-interacción implícita cuando se interpretan las instrucciones del programa. Cada vez que el programa usa una instrucción, la máquina realiza la computación interna necesaria para ejecutarla (la interpreta). Esta es la manera normal de funcionamiento por la que el sistema utiliza el meta-sistema.

14.2.8 Reflexión implícita y explícita

Al igual que en el caso de la meta-interacción, también existen dos tipos de reflexión:

- **Reflexión implícita**. El meta-sistema influye de manera implícita en el comportamiento del sistema como resultado de la meta-interacción. Por ejemplo, como resultado de cambiar la política de planificación, un proceso puede ejecutarse más rápido. Esta suele ser la manera normal de reflejar la meta-interacción.
- **Reflexión explícita**. El meta-sistema refleja de manera explícita el resultado de la meta-interacción al sistema, normalmente devolviendo un valor de retorno. Por ejemplo, devolviendo el tamaño de un tipo de datos de C tras una operación sizeof.

Un tipo especial de reflexión explícita poco común es aquél en el que como resultado de una meta-interacción (explícita o no) el meta-sistema no refleja simplemente un valor de retorno, si no que interactúa en el espacio computacional del propio sistema. Esto permite mayor extensibilidad.

14.2.9 Meta-programación

Se denomina así a la parte de los programas, tanto desde el punto de vista del sistema base (objetos base), como de los propios elementos del meta-sistema, que utiliza las operaciones de cosificación y reflexión. Es decir, las partes del sistema en las que se usa la reflectividad.

14.2.10 Implementación abierta

Los sistemas reflectivos están muy relacionados con la técnica de **implementación abierta** para el diseño de software [KP96]. Esta técnica distingue entre la interfaz base de una abstracción, que ofrece la funcionalidad de la misma y la meta-interfaz, que permite controlar cómo se implementan ciertos aspectos de la abstracción. Por ejemplo, un objeto que ordena listas tendría una interfaz base (la operación de ordenar) y una meta-interfaz que permitiría seleccionar el algoritmo de ordenación. Esta técnica, denominada a veces de "caja transparente", permite lograr sistemas más flexibles que los de implementación cerrada ("caja negra"). Se trata de romper el principio del encapsulamiento de manera controlada para lograr más flexibilidad.

Un sistema reflectivo es un ejemplo de implementación abierta. La abstracción es el meta-sistema, la interfaz base lo forma el funcionamiento normal del sistema base (que producirá una meta-interacción implícita) y la meta-interfaz es la meta-interfaz del meta-sistema. Como se describió anteriormente, la meta-interfaz no accede a todos los detalles de la abstracción (meta-sistema), si no que presenta una visión abstracta de la implementación que oculta (meta-modelo).

14.2.11 Tipos de reflectividad

A continuación se describen dos tipos ortogonales de reflectividad que se pueden encontrar en un sistema.

14.2.11.1 Reflectividad estática y dinámica

Dependiendo de si se modifica o no el estado del meta-sistema, existen dos tipos de reflectividad:

- **Reflectividad estática**. No se cambia el estado del meta-sistema. El sistema consulta su información para poder razonar acerca de sí mismo. El operador sizeof del C permite este tipo de reflectividad.

- **Reflectividad dinámica**. Se cambia el estado del sistema para de esta manera cambiar el propio comportamiento del sistema base. Como por ejemplo en el caso de que un proceso pueda cambiar la política de planificación que se le aplica.

14.2.11.2 Reflectividad estructural y de comportamiento

Aunque caracterizadas fundamentalmente para lenguajes OO basados en clases, pueden distinguirse dos tipos de reflectividad: estructural y de comportamiento [Fer89] en función del elemento del sistema de computación con el que trabajen: la definición del sistema base o el meta-sistema intérprete de esta definición:

- **Reflectividad estructural**. Está relacionada con la cosificación de aspectos estructurales de un programa, como por ejemplo tipos de datos básicos, herencia, etc. Es decir, se trabaja al nivel de la descripción del programa, de la propia definición del sistema base.

  Un ejemplo de reflectividad estructural es la API (Interfaz de los programas de aplicación, *Application Program Interface*) de reflectividad de Java [Sun97] (*Java Reflection API*). Esta interfaz permite que un programa Java gestione la información de la estructura que le compone métodos, clases, etc. Otro ejemplo es el método `getClass()` en la máquina abstracta Carbayonia. Cualquier objeto puede utilizar este método para averiguar el nombre de la clase a la que pertenece.

  Normalmente la reflectividad estructural se utiliza simplemente para que los programas puedan tomar decisiones en tiempo de ejecución consultando su propia definición, es decir, de manera estática. El programa no suele poder modificar estas estructuras. En caso de que un programa modifique sus propias estructuras usando **reflectividad estructural dinámica** (por ejemplo cambiando sus propios métodos) estaríamos ante un programa que se puede escribir a sí mismo. Este tipo de reflectividad estructural en la que se pueden cambiar las propias estructuras de un sistema se suele denominar **reflectividad lingüística**.

- **Reflectividad de comportamiento**. Esta forma de reflectividad se aplica a los aspectos de comportamiento de un sistema. Hace tangibles los elementos que permiten la computación en el sistema base y su comportamiento. Se trabaja al nivel del meta-sistema que interpreta las definiciones del sistema base. De esta manera se puede consultar y cambiar el elemento que interpreta la definición del sistema.

  Éste es el tipo de reflectividad que interesa para extender una máquina abstracta (meta-sistema), como se verá.

14.3 Selección de la arquitectura reflectiva de la máquina: Reflectividad de comportamiento mononivel controlada con reflexión explícita uniforme

El objetivo de proporcionar a la máquina una arquitectura reflectiva es que pueda ser extendida de manera transparente por objetos del sistema operativo, proporcionados en forma de objetos normales de usuario. Se trata de que los objetos puedan adquirir de esta manera propiedades adicionales (persistencia, distribución) sin necesitar modificación en los mismos. La selección del tipo de arquitectura reflectiva debe tener en cuenta este objetivo.

En este apartado se perfilan los elementos fundamentales de la arquitectura reflectiva de la máquina así como técnicas de implementación de la misma. Estos elementos y técnicas se tomarán como base para un futuro desarrollo completo e implementación de la misma.

14.3.1 Reflectividad estructural estática

La reflectividad estructural existe en el modelo de objetos de la máquina (métodos como getClass()). Actualmente la meta-interfaz no permite más que consultar información acerca de la estructura de los objetos. Es una reflectividad estática. No se permite que un objeto modifique su propia estructura, etc.

Aunque la experimentación con la máquina determinará cuál es la mejor meta-interfaz (posiblemente se puedan añadir más métodos, etc.) no es deseable permitir una reflectividad lingüística totalmente libre. El hecho de que un objeto pudiera cambiar, por ejemplo, la semántica de la invocación de métodos rompe la uniformidad conceptual del modelo de objetos. Se introduce un elemento de complejidad adicional que va en contra de la sencillez del sistema. En cualquier caso, no se descarta que se puedan introducir más flexibilidad en este tipo de reflectividad en un futuro.

14.3.2 Reflectividad de comportamiento

Para lograr el objetivo de extensión de la máquina, es la reflectividad de comportamiento el campo que mejor lo permite. Se trata de cosificar los elementos de la máquina que hacen que funcionen los objetos del nivel base. Así ya se permite que los objetos del nivel base consulten y manipulen al meta-sistema[1].

14.3.3 Uniformidad: Meta-modelo de objetos y unificación nivel sistema/meta-sistema

La utilización de un meta-modelo de objetos que se unifica con los objetos del sistema base consigue una uniformidad de uso del sistema en su conjunto.

14.3.3.1 Meta-modelo de objetos

Tiene que existir un mecanismo de interacción entre los objetos del nivel base y del meta-sistema (meta-interfaz). Es decir, la manera en la que se accede al meta-sistema depende del tipo de meta-modelo que se elija. Para lograr la mayor uniformidad y transparencia el modelo del meta-sistema se describirá dentro del propio modelo de objetos que usan los objetos normales. De esta manera la meta-interfaz de acceso a los meta-objetos[2] es del mismo tipo que la interfaz normal de acceso a otros objetos normales[3].

14.3.3.2 Unificación de niveles sistema/meta-sistema

En la mayoría de los sistemas reflectivos se utiliza cosificación explícita. La reflexión suele ser implícita. Es decir, no hay comunicación directa entre el meta-nivel y el nivel base. En los casos en que esta existe, se suele limitar a devolver un valor (por ejemplo una variable de entorno, etc.).

En cualquier caso se suele establecer una clara división entre el espacio del sistema y el espacio del meta-sistema. Esto provoca que la meta-programación se realice de forma diferente a la programación normal [Yok92]. Un ejemplo es el acceso de los objetos base a la meta-interfaz (cosificación), que se realiza de manera diferente del acceso a la interfaz normal de computación.

En lugar de esta separación se propone unificar ambos niveles en un solo nivel. Es decir, eliminar la separación conceptual entre sistema y meta-sistema, entre objetos y meta-objetos.

---

[1] El meta-sistema es la propia máquina abstracta.
[2] Dado que el meta-sistema es la máquina abstracta, se denominarán objetos de la máquina.
[3] U objetos de usuario. Es decir, objetos definidos en el espacio del usuario.

Los objetos normales no sólo comparten el mismo modelo que los meta-objetos[1], también comparten el mismo espacio de computación.

Unificación de las operaciones de invocación a métodos, cosificación y reflexión

Esto permite que el acceso a la meta-interfaz (a los objetos de la máquina) no se distinga del acceso a los objetos normales. Un objeto de la máquina no se distingue de un objeto normal más que en la funcionalidad especial que proporciona. La cosificación o acceso a un objeto de la máquina se realiza mediante una invocación normal de los métodos de este objeto.

También se propone unificar la operación de reflexión explícita con la invocación de métodos. Esto no es común en los sistemas reflectivos y dota de mayor potencia a una arquitectura reflectiva. Se trata de que los meta-objetos puedan a su vez invocar métodos de los objetos normales, gracias a compartir el mismo nivel y modelo.

Esto es de mucha utilidad pues permite extender la máquina haciendo que sus objetos colaboren bidireccionalmente con objetos de usuario de manera transparente, gracias a esta unificación.

Perspectiva uniforme del sistema. Unificación de la programación y la meta-programación

Se establece una perspectiva uniforme de todo el sistema y, en consecuencia, más sencilla, incluyendo en él a la máquina que lo soporta. El sistema es un conjunto de objetos uniforme en el que se unifica la meta-programación con la programación normal.

Meta-circularidad

La unificación de niveles facilita la meta-circularidad. Al igual que las extensiones del sistema se proporcionan mediante objetos de usuario que colaboran con los objetos de la máquina, los propios objetos de la máquina podrían escribirse también como objetos de usuario.

En realidad la unificación de niveles hace que todos los objetos sean iguales. Un objeto se clasificará como meta-objeto por la funcionalidad que proporcione. Es decir, por su aspecto externo, no por el lenguaje interno en que esté escrito (o lo que es lo mismo, el espacio de computación en que resida).

14.3.4 Nivel de detalle del modelo del meta-sistema

Es necesario determinar el nivel de abstracción del meta-sistema que se va a presentar a los objetos del sistema. Es decir, cuáles van a ser los objetos que compongan la máquina. Cuanto mayor sea el nivel de detalle de estos objetos, más finamente se podrá controlar el sistema. Sin embargo, el nivel de complejidad para utilizar el meta-sistema aumenta, al tener que trabajar con un número de objetos mayor.

El nivel de detalle más fino que se podría reflejar al sistema estaría formado exactamente por los objetos correspondientes a la implementación de la máquina.

Teniendo en cuenta un equilibrio entre el mayor control de un grano fino de detalle y la facilidad de uso, parece que lo más adecuado es utilizar como meta-objetos las propias áreas

[1] Por tanto, los meta-objetos u objetos de la máquina tendrán que describirse mediante clases que definan su comportamiento. Las instancias de estas clases serán los objetos de la máquina. En muchos casos para simplificar se hará referencia al objeto de la máquina o a su clase en solitario. Implícitamente siempre se entienden acompañados de su otro elemento asociado.

componentes de la máquina: área de clases, de instancias y de referencias. Las operaciones de la meta-interfaz (MOP) se pueden corresponder con las propias operaciones de los objetos correspondientes en la implementación.

Sigue estando en consideración la posibilidad de utilizar meta-interfaces de grano más fino que permitan un control con más detalle. Por ejemplo, exponer el mecanismo de invocación de métodos, y permitir la intervención él. De esta manera se podrían interponer objetos de control que interviniesen en cada acceso a un objeto al nivel de clase, objeto o referencia [GK97]. Un ejemplo de aplicación sería interponer un objeto de control que en cada acceso a un determinado objeto registre el acceso para tener un histórico de los accesos al objeto.

En cualquier caso la definición concreta de las operaciones de la meta-interfaz será decidida en función de las necesidades que se presenten en la construcción de las propiedades que se encargan al sistema operativo y del grado de detalle en el control que se quiera proporcionar al espacio de usuario.

14.3.5 Reflectividad controlada: mantenimiento de la semántica básica del sistema

Es necesario un control sobre los cambios que permite la reflectividad en el sistema. La flexibilidad que permite la reflectividad puede llegar a ser excesiva. En concreto, no conviene permitir cambios que hagan que la semántica básica del sistema pueda cambiar, un problema derivado del **solapamiento reflectivo** [Ste94] (*reflective overlap*). Este uso no controlado de la reflectividad tiene efectos negativos en la portabilidad de las aplicaciones. Para mantener la uniformidad, la portabilidad, la sencillez y la fácil comprensión del sistema los conceptos básicos del mismo (modelo de objetos) deben mantener siempre el mismo significado. Por ejemplo, la herencia debe tener siempre el mismo significado en todo el sistema y no debe poder cambiar.

Para controlar que no se puedan realizar más que cambios que permitan extender la semántica del sistema, la meta-interfaz se definirá de manera que sólo presente operaciones que no permitan este tipo de cambios no deseados.

14.4 Implementación de la reflectividad en la máquina abstracta

La implementación tiene que conseguir la abstracción de establecer un único nivel en el que existen todos los objetos del sistema dentro del mismo marco conceptual.

14.4.1 Nivel único de objetos. Modelo de meta-objetos igual al modelo de objetos

Hay que conseguir la unificación de los niveles de objetos y meta-objetos. Es necesario, pues, hacer que ambos compartan el mismo marco de objetos. La única operación en el sistema será la invocación de métodos, que realizará su función normal y se unificará con la cosificación y la reflexión explícita.

14.4.2 Autoarranque (bootstrapping)

Teniendo en cuenta esto, en principio no habría necesidad de realizar nada especial para conseguir el nivel único de objetos. Los meta-objetos de la máquina serían simples objetos de usuario y funcionarían normalmente como los demás.

Esto podría ser así partiendo de un sistema inicial que estuviera ya funcionando, pero en la práctica hay que partir de un punto inicial conocido. Es el problema del **autoarranque**

(*bootstrapping*). Para poder definir objetos de usuario que funcionen como objetos de la máquina necesitamos un sistema que permita la existencia de objetos de usuario. Pero para ello necesitamos previamente estos objetos.

14.4.3 Técnica de implementación de clases primitivas

Este es un problema similar al de las clases básicas del modelo de objetos descrito en el capítulo de implementación de Carbayonia (véase el capítulo 13). Para solucionarlo se proporcionaba una implementación primitiva de las clases básicas, que era transparente para todo el sistema, ya que no se diferenciaban en su uso de otras clases.

Autoarranque

Esta misma técnica se puede utilizar para dar una implementación primitiva de los meta-objetos, los objetos de la máquina.

- **Clase primitiva para el meta-objeto**. Se proporciona una clase primitiva para cada meta-objeto. Los métodos de la clase se corresponderán con los que se necesiten en su meta-interfaz (por ejemplo, eliminar un objeto del área de instancias). Las llamadas a métodos de la clase acabarán una implementación primitiva de los mismos.
- **Objetos primitivos de la máquina**. En este caso, además de la clase primitiva, se necesita que exista también una instancia de cada una de estas clases. Se proporcionarán los objetos primitivos (meta-objetos) de estas clases que existen en la máquina (área instancias, etc.). De esta manera se podrá acceder a estos objetos como un objeto normal del sistema. Hay dos maneras de realizar esto, que pueden complementarse:
  - **Instancias predefinidas**. En el área de instancias se dejan predefinidas (existen siempre) las instancias de estos objetos. Podría pensarse que como parte del arranque de la máquina se crean los meta-objetos correspondientes a partir de sus clases.
  - **Referencias globales**. Se pueden utilizar unas referencias globales que apunten (conceptualmente) a estos meta-objetos. Al igual que existe una referencia this que cualquier objeto puede usar para llamarse a sí mismo, se puede tener una referencia para cada meta-objeto (areaInstancias, areaClases, etc.) que permita llamar a sus métodos de manera normal.

Unificación de meta-operaciones con la invocación de métodos

De esta manera se soluciona el problema del autoarranque. También se soluciona el problema de unificar la invocación de métodos normales con la cosificación y la reflexión:

- **Unificación de modelos**. Las clases primitivas de los meta-objetos están en la misma jerarquía de clases del sistema y, en consecuencia, pueden ser usadas como una clase más.
- **Cosificación**: Un objeto (primitivo o no) invoca a un meta-objeto. La llamada va a la clase primitiva del meta-objeto, donde se accede al método primitivo.
  - **Devolución de un valor de reflexión**. Como parte de una llamada de cosificación, se devuelve en el valor de retorno normal del método.
- **Reflexión explícita**. Un meta-objeto invoca otro objeto (primitivo o no). Este meta-objeto (primitivo) conoce la implementación interna de la máquina. Por tanto,

simplemente tiene que llamar a las operaciones internas de la máquina que realizan la invocación de objetos pasando los parámetros adecuados.

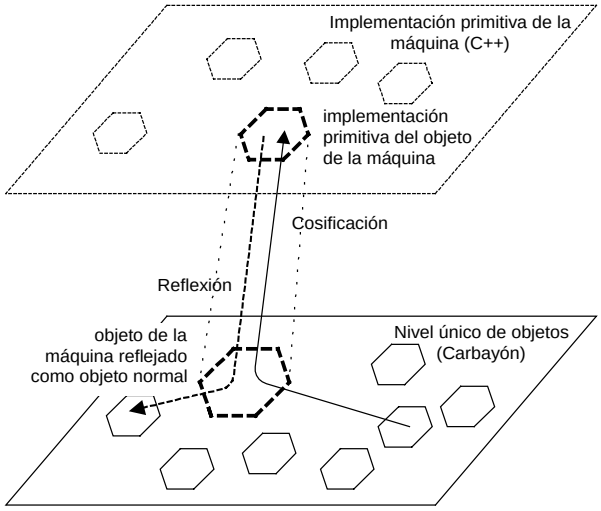


Figura 14.6 Técnica de implementación primitiva para la unificación de meta-operaciones con la invocación normal de métodos.

Es importante remarcar que todas estas distinciones de clases primitivas, llamadas que producen cosificación, etc. en realidad son distinciones en el nivel interno de implementación. Desde el punto de vista externo sólo existe un nivel de objetos donde todos los objetos son del mismo tipo. Para los objetos sólo existen invocaciones de métodos en todos los casos y a objetos de su misma categoría. No se necesita distinguir desde el punto de vista externo entre clases y objetos normales de usuario frente a clases y objetos de la máquina (o que proporcionen funcionalidad del meta-sistema). En caso de hacerla sería de manera subjetiva a través de su funcionalidad.

14.4.3.1 Selección de meta-objetos primitivos: decisión de implementación

Al igual que cualquier otra clase, la decisión de implementar un meta-objeto (y su clase) como primitivo o no, es una decisión de implementación, con razonamientos parecidos a los descritos para las clases en el capítulo 13. Además de por las cuestiones de eficiencia mencionadas en él, la decisión de implementar un meta-objeto como primitivo puede ser por razones de autoarranque (el problema de la recursión infinita que debe terminar en objetos implementados directamente por la máquina). Es necesario que existan al menos los meta-objetos primitivos iniciales mínimos que permitan ir desarrollando el resto del sistema mediante elementos normales de usuario (incluyendo otros meta-objetos).

14.4.4 Fases de introducción de la reflectividad en el sistema

Existen diferentes fases en la introducción de la reflectividad de un sistema, que se describen teniendo en cuenta la aportación que hacen para la extensibilidad del sistema de manera flexible. Es decir, cómo permiten eliminar, modificar e introducir nueva funcionalidad (o servicios) en el sistema. Aunque centrado en el aspecto de objetos que proporcionen meta-funcionalidad, la discusión es aplicable en general a cualquier tipo de funcionalidad (cualquier tipo de clase, por ejemplo las clases básicas del modelo u otras clases de usuario).

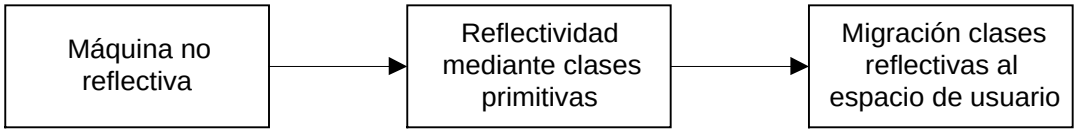


Figura 14.7 Fases de introducción de la reflectividad en el sistema.

14.4.4.1 Máquina no reflectiva

La primera versión de la máquina no incorpora reflectividad de comportamiento (sí existe reflectividad estática estructural mediante ciertos métodos de la clase básica Object). Todo el funcionamiento del sistema está oculto dentro de la implementación y no es accesible. Por ejemplo, si se incorpora un recolector de basura este no se puede modificar ni desactivar. El sistema queda sujeto a la funcionalidad concreta que se haya decidido implementar en la máquina.

Si se quiere cambiar esta funcionalidad por ejemplo para eliminar el recolector de basura (o para añadirlo a una máquina que no lo tenga) hay que parar el sistema, modificar el código fuente de la implementación de la máquina para cambiar la funcionalidad, recompilar, enlazar y volver a arrancar. Esto es un proceso muy lento que coarta la experimentación de nueva funcionalidad en el sistema, sobre todo porque dificulta la corrección de errores. Cada vez que se encuentra un error hay que repetir el proceso anterior para modificarlo. Aún así, la funcionalidad que se introduzca no es accesible por el sistema base.

14.4.4.2 Elementos reflectivos proporcionados mediante clases primitivas

Se añade la reflectividad en el sistema mediante las (meta-) clases y objetos necesarios implementados de manera primitiva. La funcionalidad se implanta en forma de clases del propio modelo del sistema, en lugar de hacerlo de manera interna en la implementación de la máquina.

Cuando se decide utilizar la implementación primitiva (de cualquier clase en general) hay que tener en cuenta que la funcionalidad que proporcionan estos elementos primitivos no puede ser sustituida ni eliminada. Se integra de manera uniforme con el resto del sistema, pero para cambiarla (por ejemplo añadiendo otros objetos primitivos) se necesita cambiar la implementación primitiva. Esto también necesita el tedioso ciclo de parada/modificación/recompilación/arranque. Sin embargo, en este caso el sistema es muy flexible ya que se puede acceder a estos elementos de manera uniforme mediante objetos de usuario.

Para permitir la mayor flexibilidad posible, lo más interesante sería que el número de meta-objetos básicos fuera el menor posible. En concreto el conjunto mínimo de meta-objetos que proporcionan la funcionalidad básica del sistema. Esta funcionalidad básica siempre permanecería en la máquina, luego no sería necesario cambiar nunca estos meta-objetos y, por tanto, entrar en el ciclo anterior.

El resto de la funcionalidad del sistema, incluyendo los objetos con funcionalidad meta-, se proporcionaría mediante objetos normales de usuario, según el punto siguiente. Estos objetos implementarían propiedades o funcionalidad que no se necesiten en todas las circunstancias o entornos. Por ejemplo, el proporcionar la funcionalidad de recolección de basura mediante un objeto de usuario es más flexible que mediante un objeto primitivo. Además de permitir la reflectividad, si este objeto no es necesario (como en un sistema empotrado que cree siempre el mismo conjunto fijo de objetos) simplemente se elimina el objeto de usuario del sistema. No hay que parar el sistema para introducir los cambios en el mismo.

14.4.4.3 Migración de las clases reflectivas primitivas a espacio de usuario

La manera más rápida (y, en algunos casos como el autoarranque, la única) de implantar una clase reflectiva es mediante la implementación primitiva. Sin embargo, una vez que el sistema está funcionando, puede trasladarse la funcionalidad de la implementación primitiva a una clase de usuario. Para garantizar la fiabilidad, puede hacerse este proceso mediante un mecanismo paso a paso que elimina problemas de dependencia mutua [Ste93, MT96]:

- **Implementación inicial**. Se parte de un sistema en el que una cierta funcionalidad se proporciona mediante un meta-objeto implementado de manera primitiva. Algunos objetos del sistema utilizarán este meta-objeto primitivo.
- **Desarrollo y prueba de la versión en espacio de usuario**. Se desarrolla un objeto del usuario con la misma funcionalidad que da el objeto primitivo. Se prueba este nuevo objeto para comprobar que en efecto funciona como el objeto primitivo. En el desarrollo se pueden utilizar todas las facilidades que proporciona el sistema base: orientación a objetos, facilidad de prueba y depuración, etc. Nunca es necesario detener el funcionamiento del sistema. Durante este proceso se evita la dependencia mutua (o **solapamiento reflectivo**) puesto que los objetos ya existentes utilizan la versión primitiva del servicio.

- **Migración a la nueva versión**. Una vez probada la versión del servicio como objeto normal de usuario, se hace que los objetos anteriores pasen a usar este servicio en lugar del primitivo. Para eliminar la versión primitiva del servicio en la máquina no hay más remedio que parar el sistema para eliminarlo de la máquina. Sin embargo, esto sólo hay que hacerlo una vez, puesto que la nueva versión como objeto de usuario ya estará depurada.

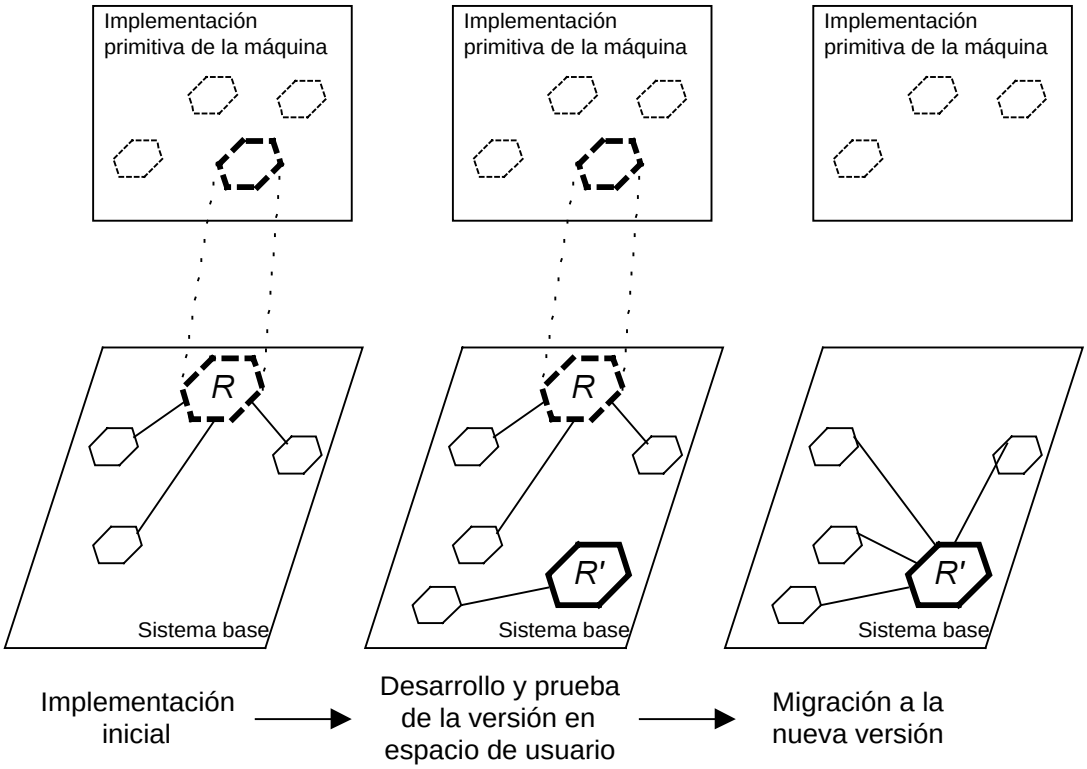


Figura 14.8 Fases para la migración de clases reflectivas primitivas al espacio del usuario.

Todos estos pasos conducen cada vez más a la meta-circularidad. Más y más elementos del sistema pueden estar proporcionados mediante implementaciones de usuario. En último extremo se conseguiría la meta-circularidad total: un sistema descrito totalmente en términos de sí mismo (con implementación de usuario en su totalidad).

14.5 Extensión en el espacio del usuario

La reflectividad permite introducir mayor flexibilidad en el sistema, permitiendo la extensión del sistema mediante elementos del espacio del usuario.

14.5.1 Núcleo monolítico sin reflectividad. No hay extensibilidad dinámica

En una máquina sin reflectividad la funcionalidad se tiene que incluir toda en la implementación primitiva. El software del sistema forma un núcleo monolítico que no se puede cambiar o extender salvo cambiando la implementación primitiva.

No hay ninguna flexibilidad, si se desea extender el sistema añadiendo nueva funcionalidad debe ser de manera estática. No hay más remedio que parar el sistema y trabajar en la implementación primitiva. Se deben programar los cambios en el lenguaje de implementación primitivo (normalmente menos productivo que el entorno que presenta el sistema base). Se recompila el núcleo, se enlaza y se vuelve a arrancar el sistema. En

funcionamiento se debe comprobar si existe algún error. Cada vez que se encuentra un nuevo error debe repetirse este lento proceso.

Tampoco hay flexibilidad con la funcionalidad no deseada. Aunque no se necesite en un entorno de aplicación determinado no hay más remedio que aceptar toda la funcionalidad incluida en el núcleo del sistema. A no ser que se recurra a la modificación de la implementación primitiva se debe cargar con ella (con la sobrecarga innecesaria que acarree, por ejemplo un soporte para persistencia)[1].

Al estar toda la funcionalidad del software de sistema en el núcleo primitivo monolítico, cualquier cambio que se necesite tiene que incluirse en el núcleo, lo que obliga a costosas modificaciones en la implementación primitiva.

14.5.2 Máquina reflectiva. Extensión dinámica

La reflectividad permite lograr una mayor flexibilidad en cuanto a la funcionalidad que incluye un sistema. Su uso permite escribir la mayor parte de los componentes del software de sistema en el espacio de usuario, utilizando de manera uniforme la interfaz de programación de usuario. Los objetos normales de usuario pueden acceder a la funcionalidad del meta-sistema. Se rompe la implementación primitiva monolítica, exponiendo los módulos que la componen de manera uniforme e integrada con el espacio de usuario. En este caso los componentes del sistema serán los objetos del sistema operativo, y gracias a la propia reflectividad los objetos de la propia máquina.

14.5.2.1 Extensión dinámica en el espacio de usuario

La reflectividad mononivel unificada permite que las extensiones se sitúen en el espacio de usuario. Para extender la máquina pueden utilizarse objetos normales de usuario que colaboren con los objetos de la máquina y viceversa. La programación de extensiones de la máquina no se diferencia de la programación normal de objetos de usuario (invocaciones a métodos de objetos) y puede utilizar, por consiguiente, todas las ventajas de la misma (como la orientación a objetos). Es dinámica puesto que al usarse objetos normales de usuario no se necesita parar el sistema.

La extensión es dinámica, al ser objetos de usuario no se necesita parar el sistema. Se pueden añadir, eliminar y modificar estos objetos sobre la marcha, comprobar su buen funcionamiento, etc. Por ejemplo, un usuario podría escribir su propio sistema de persistencia, o mejorar el existente. O eliminarlo, si no necesita esta funcionalidad.

14.5.2.2 Núcleo primitivo

Aunque desde el punto de vista de utilización todos los elementos del sistema pertenecen a un único espacio, puede considerarse que el núcleo del sistema lo forman los objetos implementados de manera primitiva. Los cambios en estos objetos deben realizarse de manera estática. Sin embargo, la reflectividad permite extender este núcleo de objetos implementados de manera primitiva en el espacio de usuario, como se acaba de ver.

Para aprovechar al máximo las ventajas de proporcionar la funcionalidad mediante objetos de usuario conviene que la funcionalidad del sistema (objetos de la máquina) implementada mediante objetos primitivos sea la menor posible. De esta manera, la mayor parte de la funcionalidad de la máquina estará en el espacio de usuario, con lo que se podrán ampliar y adaptar estos objetos, eliminar aquellos cuya funcionalidad no se necesite, añadir nuevos

[1] Este es el problema de muchos sistemas operativos y aplicaciones actuales. Al introducirse en su núcleo monolítico cada vez más y más características que no se pueden eliminar, el usuario acaba pagando un precio (económico y de rendimiento) adicional por una funcionalidad que no necesita.

objetos que den más funcionalidad, etc. Únicamente para los objetos que proporcionen la funcionalidad mínima común de la máquina (o por razones de eficiencia) que siempre se necesitarán y no cambiarán tendría sentido mantener la implementación primitiva. De esta manera puede actualizarse también el software de sistema de manera dinámica, sin necesidad de suspender el funcionamiento del mismo.

14.5.2.3 Colaboración explícita de los objetos de la máquina con los de usuario

No sólo se produce la cosificación de los objetos de la máquina, los objetos de la máquina pueden acceder a su vez de la misma manera uniforme a los objetos de usuario. Esto da un mayor nivel de potencia al sistema reflectivo y es fundamental para que el sistema operativo extienda la máquina para dar propiedades adicionales a los objetos del sistema.

Por ejemplo, para implementar la propiedad de la distribución transparente (o la persistencia) mediante objetos de usuario, es fundamental que los objetos de la máquina puedan colaborar explícitamente con estos objetos de usuario [Alv96a]. Cuando en la máquina se reciba una invocación de un método a un objeto remoto que no se encuentre en la misma, el objeto responsable de la máquina llamará al objeto de usuario que implemente el sistema de distribución. Este objeto de usuario se comunicará con la máquina remota para llevar la invocación de manera transparente al objeto remoto.

14.5.3 Ventajas de la extensión de la funcionalidad en el espacio de usuario

La funcionalidad del sistema proporcionada en el espacio del usuario tiene una serie de ventajas. Algunas de estas son:

14.5.3.1 Mayor productividad en el desarrollo del sistema

En lo que se refiere al desarrollo del propio sistema se alcanza una mayor productividad.

Lenguaje de usuario más productivo.

Al poder escribir la funcionalidad del sistema en el espacio del usuario, se puede utilizar la interfaz de programación normal del espacio de usuario. Normalmente esta interfaz suele ser más productiva que los entornos de los lenguajes que se usan para la implementación primitiva.

Ciclo de desarrollo más corto.

El resultado de los cambios se ve inmediatamente. No es necesario realizar todo el ciclo de parada del sistema, modificación, recompilación, enlazado y rearranque para observar el efecto de las modificaciones. La corrección de errores es mucho más rápida, al no necesitar el proceso anterior cada vez que se corrige uno.

14.5.3.2 Detención del sistema no necesaria

No es necesario parar el sistema para hacer cambios en la funcionalidad del mismo. Los elementos de usuario se pueden eliminar, añadir o modificar en cualquier momento, sin interrumpir el funcionamiento del sistema en su conjunto. Esto es muy importante en muchos casos.

Soporte para dispositivos nuevos

El más evidente es el de un sistema cuya detención acarrea muchos inconvenientes, como un servidor de aplicaciones de empresa. Con la extensión en el espacio de usuario se podría

por ejemplo añadir el controlador de un nuevo dispositivo sin necesidad de detener el sistema[1].

Actualización remota del sistema

En general la extensión dinámica permite actualizar el software de manera sencilla. Un caso a tener en cuenta es el de la actualización remota del software de un sistema. Por ejemplo la actualización de los programas de control de una red de cajeros remotos. Esto tiene cada vez más importancia cuando se trabaja en redes como Internet, en las que la funcionalidad que necesita un sistema tiene que cambiar muy rápidamente. En este tipo de entornos es común utilizar software traído de la propia red para ampliar las capacidades del sistema cliente. Un ejemplo restringido de este tipo de actualización remota son los *applets* Java y los *plug-in* que dotan de capacidades adicionales a los navegadores de Internet.

14.5.3.3 Adaptación del sistema a su uso

La modificación dinámica permite la fácil adaptación de un sistema al uso que se le va a dar.

Eliminación funcionalidad no necesaria

La funcionalidad que no se va a utilizar se puede eliminar sin más, al ser elementos de espacio de usuario. No hay que destinar recursos del sistema para albergar funcionalidad que no se va a utilizar. Por ejemplo, si un sistema empotrado no necesita soporte para persistencia, simplemente puede eliminarlo del sistema sin más.

Adaptación de la funcionalidad a la aplicación

Si una aplicación necesita una determinada funcionalidad pero de manera especial, puede escribirse su propia versión de esa funcionalidad. Por ejemplo, un sistema de persistencia especial para un sistema empotrado que utilice como almacenamiento secundario memoria RAM no volátil en lugar de disco magnético.

Incorporación de funcionalidad adicional

Las aplicaciones pueden utilizar el sistema aumentándolo con la funcionalidad adicional que necesiten y no se encuentre ya en el sistema. Por ejemplo, para un entorno de aplicación en el que se necesite una política de seguridad para trabajo en grupo podrían desarrollarse los objetos que implementasen esta política, haciendo uso interno del mecanismo de protección básico del sistema.

14.6 Resumen

Para que el sistema operativo pueda extender la máquina abstracta y proporcionar transparentemente sus características adicionales a los objetos del sistema se identifican tres mecanismos: modificación interna de la máquina, extensión de la funcionalidad de las clases básicas y colaboración con el funcionamiento de la máquina.

Para la colaboración con el funcionamiento de la máquina se dota a la máquina de una arquitectura reflectiva. En este tipo de arquitecturas los objetos del sistema base pueden acceder y manipular su meta-sistema (objetos de la máquina) que les dan soporte. Para ello se utilizan dos operaciones: cosificación (acceso al meta-sistema desde el nivel base) y reflexión (efecto de la cosificación en el meta-sistema).

[1] Este caso concreto comienza a ser cada vez más importante debido a la progresiva aparición de *buses* con dispositivos que se pueden conectar a la unidad central sobre la marcha.

Para crear una perspectiva uniforme del sistema se propone describir los objetos de la máquina con el mismo modelo y dentro del mismo espacio de los objetos de usuario, unificando las operaciones de cosificación y reflexión con la invocación de métodos. La programación de los objetos normales y los de la máquina (meta-programación) se unifica. En lugar de un modelo con mucho detalle, que podría complicar la comprensión del sistema se utilizarán como meta-objetos las áreas constituyentes de la máquina, no permitiendo que su interfaz deje cambiar las propiedades fundamentales del sistema (para mantener la consistencia).

Esto aporta una gran flexibilidad, puesto que permite la extensión dinámica del sistema por medio de objetos en el espacio de usuario. La reflectividad hace que los propios objetos de la máquina se definan en el espacio de usuario, permitiendo aplicar todas las ventajas del espacio de usuario a los propios objetos de la máquina: eliminación, adición y modificación dinámica de los mismos sin necesidad de detener el sistema y recompilar la implementación primitiva del sistema.

Para la implementación de la reflectividad en el sistema se propone utilizar la técnica de implementación primitiva de clases usada para la implementación de las clases básicas. Cada objeto de la máquina (cada objeto que se quiera reflejar) tendrá una clase que describe sus operaciones, implementada de manera primitiva (en el espacio del sistema). Así se unifican los modelos de objetos y meta-objetos. Además, se predefinirán las instancias adecuadas de estos objetos para hacerlos aparecer ante los demás objetos y lograr la reflectividad y la uniformidad. Paulatinamente se pueden ir migrando estas implementaciones primitivas de meta-objetos al espacio de usuario en dos pasos, sin provocar problemas de integridad.

Capítulo 15

EL SISTEMA OPERATIVO SO4 PARA EL SISTEMA INTEGRAL OVIEDO3

En este capítulo se describe el diseño preliminar del sistema operativo para el sistema integral Oviedo3, denominado SO4[1] [ATA+96, ATA+97]. Al sistema operativo se le encargan todos aquellos cometidos que estén relacionados con funcionalidad típica de sistema. Los elementos más importantes de esta funcionalidad son los que permiten dotar a los objetos de manera transparente con las propiedades de persistencia, concurrencia, distribución y seguridad.

La persistencia permite que los objetos permanezcan en el sistema hasta que ya no sean necesitados. La distribución hace que se pueda invocar un método de un objeto independientemente de la localización del mismo en el sistema distribuido. La concurrencia proporciona al modelo de objetos la capacidad de ejecutar tareas en paralelo. Otro elemento muy importante que debe estar presente en un sistema distribuido con múltiples usuarios es la seguridad o protección del acceso a los objetos. Sólo los objetos autorizados podrán enviar mensajes a otros objetos.

15.1 Transparencia e integración en el modelo

El objetivo fundamental a la hora de implantar estas propiedades es la transparencia. Los objetos adquirirán estas propiedades sin necesidad de hacer nada especial. No tienen por qué ser conscientes ni intervenir en los mecanismos que se usan para lograrlas (excepto en el caso en que sea imprescindible, como los objetos del sistema operativo). Por otro lado es muy importante que la introducción de estas propiedades se integre de manera fluida con el modelo de objetos del sistema. Es decir, que no se necesiten cambios en la semántica del modelo, o que sean menores y no rompan la esencia del modelo.

[1] Sistema Operativo para Oviedo3

Para alcanzar estos objetivos, el sistema operativo utilizará los mecanismos de extensión de la máquina descritos en el capítulo 14, aunque la manera normal será proporcionar estas características mediante objetos normales de usuario[1]. La arquitectura del sistema, basado en la máquina abstracta, ciertas propiedades del modelo de objetos, la reflectividad y la extensión en el espacio del usuario facilitan la implementación del sistema operativo, para alcanzar el objetivo de un mundo de objetos: *un único espacio de objetos, virtualmente infinito, en el que un conjunto de objetos homogéneos coopera intercambiando mensajes independientemente de su localización, y donde los objetos residen indefinidamente hasta que ya no son necesitados*.

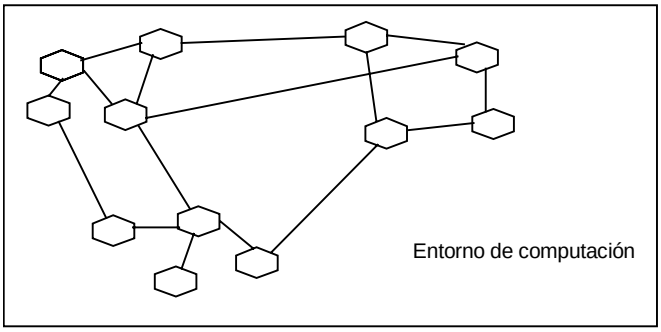


Figura 15.1 Entorno de computación compuesto por un conjunto de objetos homogéneos.

El diseño con profundidad de las propiedades que debe implementar el sistema operativo, así como la implementación de las mismas está siendo investigado por otros miembros del proyecto Oviedo3. A continuación se resumen las características preliminares de estos diseños, adaptadas de [ATA+96, ATA+97]. En el caso de la propiedad de la persistencia, se tiene un tratamiento más detallado en capítulos posteriores. En cualquier caso, dado que estas propiedades se encuentran aún en desarrollo, pueden existir algunas variaciones respecto a la forma definitiva que tomen.

15.2 Seguridad: mecanismo de protección

En un sistema orientado a objetos, la entidad a proteger es el objeto, más concretamente el acceso a las operaciones del objeto. En un sistema de objetos homogéneos como el que nos ocupa, el objetivo del sistema de seguridad [Día96] es proporcionar un mecanismo de protección uniforme para controlar el acceso a los métodos de cualquier objeto del sistema. Es decir, qué objetos tienen acceso a cada operación de un objeto determinado.

15.2.1 Elementos fundamentales de la protección de objetos

Las dos ideas fundamentales que se proponen para la protección de objetos son:

- Uso de capacidades como referencias a los objetos
- Mecanismo de protección en el nivel más interno del sistema

15.2.1.1 Uso de capacidades como referencias a los objetos

Las capacidades [DH66, Lev84] pueden integrarse en los mensajes de invocación a métodos como parte de las referencias a objetos. Una **capacidad** es similar a una entrada a un

[1] Que por su funcionalidad se denominarán objetos del sistema operativo.

espectáculo: la posesión de la misma basta para lograr el acceso. En un sistema de objetos, una capacidad contendrá un elemento que identifique el objeto para el que es válida la capacidad y los permisos de acceso que esta concede (si se puede acceder o no a cada uno de los métodos). Si se quiere dar un cierto acceso a un objeto, simplemente se da una capacidad con los permisos adecuados. Esto se integra de forma sencilla en el modelo de objetos, haciendo que el mecanismo de protección no altere el modelo de objetos del sistema. La capacidad puede enviarse dentro del propio mensaje de petición de la operación, por tanto, no supone apenas adición de carga.

Se han descartado otros mecanismos de protección bien conocidos, como las listas de control de acceso o mecanismos mixtos debido a que estos comportan una alteración no deseada en el modelo de objetos. Necesitan introducir información de protección dentro del objeto y delegar en él la comprobación de protección. Con las capacidades la información de protección de un objeto se almacena en las referencias externas de acceso al mismo. No se necesita cambiar nada en los objetos para que tengan protección.

15.2.1.2 Mecanismo de protección en el nivel más interno del sistema

La seguridad se introduce de manera uniforme en el corazón del sistema, es parte integral del mismo. Para ello basta con hacer que el mensaje únicamente llegue a su destino si el mecanismo de protección comprueba que la capacidad es correcta (se tienen los permisos correspondientes). De esta manera, conceptualmente, el mecanismo de protección mediante capacidades se realizaría como parte integral del mecanismo de envío de mensajes entre objetos (invocación de operaciones)

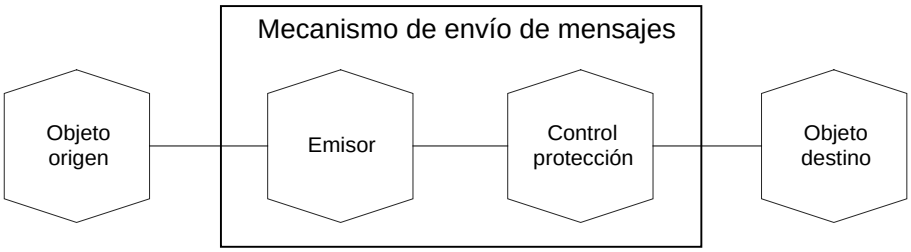


Figura 15.2 Integración de la protección en el mecanismo de envío de mensajes.

La protección se implanta en la invocación de métodos, la única operación que pueden hacer todos los objetos del sistema. No hay cambios en la semántica del sistema puesto que simplemente ahora se comprueba, además, la información de protección.

La introducción de una política de seguridad puede ser realizada por objetos en el espacio de usuario, utilizando internamente la protección por capacidades. Además, esto permite la convivencia de varias políticas de seguridad en un mismo sistema, adaptadas a las necesidades de diferentes tipos de aplicaciones.

15.2.2 Implementación

Para implantar las capacidades será necesario realizar algunas modificaciones en las estructuras de la máquina abstracta y en su funcionamiento. El incorporar la protección en la máquina abstracta obliga a estas modificaciones. Sin embargo, la importancia de establecer un mecanismo de protección confiable para todos los elementos del sistema hace que esté justificado introducirlo como parte fundamental y constituyente de la máquina abstracta. De esta manera todos los objetos dispondrán de este mecanismo y podrán tener siempre su seguridad garantizada.

15.2.2.1 Adición de atributos en las referencias

Para implantar la protección se propone ampliar la arquitectura de la máquina para que las referencias pasen a ser capacidades. Es decir, ampliar el contenido de una referencia con la información de protección. Esta información indicará para cada método si se puede acceder o no al mismo a través de la capacidad.

Figura 15.3 Adición de información de protección en las referencias.

15.2.2.2 Adición y modificación de operaciones con referencias

Las operaciones actuales con referencias: crear y eliminar los objetos a los que apuntan, se mantienen en la máquina y su funcionamiento es similar, simplemente hay que tener en cuenta la existencia de la información de protección.

Sin embargo, deben añadirse operaciones adicionales que tengan en cuenta la nueva categoría de las referencias como capacidades. En principio pueden proponerse las siguientes, algunas cambian su semántica, otras son nuevas: Crear, Eliminar, Copiar, Crear Capacidad Restringida. Esta última es la que permite crear capacidades personalizadas para restringir el acceso a voluntad a un determinado objeto.

15.2.2.3 Modificación del mecanismo de envío de mensajes

El mecanismo de envío de mensajes de la máquina se basa en tomar una referencia, localizar el objeto e invocar la operación deseada. Ahora, tras localizar el objeto mediante el identificador del mismo, se debe examinar la información de protección y si se tienen permisos para ello, invocar la operación deseada. En caso de que no se tengan los permisos necesarios, se lanza una excepción.

15.2.3 Ventajas de este diseño

El uso de una máquina abstracta permite combinar las ventajas tanto de las capacidades segregadas como las dispersas [AC88]:

15.2.3.1 Protección automática de las capacidades

Las capacidades están segregadas, ya que no pueden ser manipuladas por el usuario más que con las operaciones que están definidas al efecto. La máquina garantiza que no pueden modificarse arbitrariamente. Al asegurar la imposibilidad de falsificación o modificación no deseada de las capacidades por "hardware" se evita el coste adicional que suponen las técnicas de capacidades dispersas.

15.2.3.2 Sencillez y facilidad de uso

Por otro lado, se pueden utilizar directamente dentro de estructuras del usuario ya que constituyen precisamente la referencia que utilizan los usuarios en sus estructuras para acceder a los objetos de manera normal. Al permitir la utilización como referencias normales en estructuras de usuario, se facilita el uso del sistema al utilizar un único modo de trabajo con los objetos, sin la dualidad de uso que supone el tener que acceder de una manera especial al área segregada donde están las capacidades.

15.3 Persistencia

La idea básica para integrar la persistencia en el sistema [Alv96b] es proporcionar la abstracción de un espacio de objetos virtualmente infinito en el que residen los objetos hasta que no se necesitan. Podría considerarse como una memoria virtual persistente para los objetos.

Esto puede implementarse extendiendo de manera transparente el área de instancias de la máquina. Se forma un área de instancias virtual en la que se utiliza el almacenamiento secundario para hacer persistir los objetos de manera transparente guardándolos en él cuando no estén (o no quepan) en el área de instancias de la máquina.

En el capítulo 16 se hace un desarrollo más elaborado del diseño del sistema de persistencia.

15.4 Distribución

En un entorno distribuido compuesto por varias máquinas Carbayonia[1] conectadas mediante una red de comunicación, el sistema de distribución [Alv96a] permitirá básicamente la comunicación entre los objetos independientemente de la máquina en la que se encuentren.

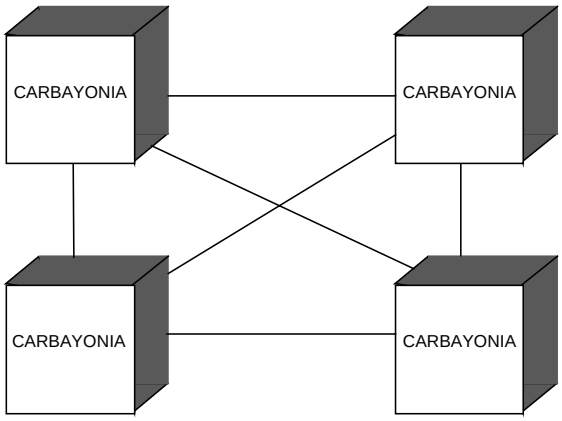


Figura 15.4 Arquitectura distribuida del sistema.

15.4.1 Objetivos del sistema de distribución

Los objetivos básicos del sistema de distribución se pueden reducir a:

- **Transparencia de localización y de acceso [CDK94].** Proporcionar un servicio de localización que entregue los mensajes a los objetos, independientemente de su localización
- **Movilidad de objetos y equilibrio de carga**.

15.4.2 Propiedades del sistema que favorecen la distribución

Las propiedades del sistema que más directamente suponen ventajas para la implementación del sistema de distribución son:

[1] Todas las máquinas abstractas Carbayonia tienen la misma funcionalidad. Sin embargo, el hardware subyacente puede ser heterogéneo.

15.4.2.1 Identificador único de objetos

La implementación de la transparencia de localización y acceso se facilita mucho ya el identificador de cada objeto es único. Simplemente haciendo que sea único también en el conjunto del sistema[1] se consigue referenciar y acceder a los objetos siempre de la misma manera, a través de su identificador, de manera independiente de su localización. Teniendo en cuenta que los objetos se podrán mover en el sistema, no se puede usar una potencial información interna en el identificador que permitiese conocer la máquina de creación del objeto. Es decir, desde el punto de vista de su uso, los identificadores no llevan ningún tipo de información.

15.4.2.2 Objetos autocontenidos

Los objetos del modelo encapsulan totalmente toda su semántica. Para mover un objeto entre máquinas simplemente basta con mover su estado encapsulado. Este contendrá todo lo necesario para que el objeto continúe en la nueva máquina exactamente igual que en la anterior. Por tanto, como parte del estado de un objeto se considera toda la semántica, incluyendo la de la computación (hilos, etc.). Conceptualmente, este hecho simplifica notablemente la implementación.

15.4.3 Ventajas de este diseño

Combinando los aspectos anteriores se consiguen varias ventajas:

- **Espacio único de objetos distribuidos**. Que puede extenderse a un espacio persistente distribuido cuando se utilice juntamente con la propiedad de persistencia.
- **Mejora del rendimiento**. Mediante el equilibrado de la carga en el total del sistema usando la movilidad de objetos.
- **Mejora de la fiabilidad**. Usando un mecanismo de replicación (como en Chorus [BFG+85] o Clouds [DLA+91]), posiblemente en combinación con el mecanismo de persistencia se mejora la fiabilidad del sistema.

15.4.4 Implementación

Para la implementación del sistema se proponen los siguientes puntos.

15.4.4.1 Localización de objetos

Todos los objetos del sistema tienen un identificador único, global para todo el sistema y que se les asigna en el momento de su creación. Este identificador será utilizado durante toda la vida del objeto. Cada máquina Carbayonia lo utiliza para encontrar un objeto dentro del área de instancias (y en su caso le ayudará el sistema de persistencia buscándolo en el almacenamiento secundario)

En el caso de no encontrar el objeto al que hay que acceder en el espacio local de la propia máquina (para invocar un método), la máquina (o más bien el objeto correspondiente del sistema de persistencia) pedirá ayuda al sistema de distribución (de manera reflectiva).

15.4.4.2 Servidores de localización

Para implementar la distribución se utilizará un servidor de localización de objetos de concepción similar al del sistema Clouds [DLA+91]. El esquema de localización de objetos es

[1] Usando un generador de identificadores que garantice esta unicidad dentro del sistema distribuido. Por ejemplo usando en la generación la dirección física de red de cada máquina se puede garantizar que los identificadores serán únicos.

el de múltiples **servidores de nombres** (identificadores, en este caso), el cual va a permitir la movilidad de los objetos. Un servidor de nombres no va a ser otra cosa que un objeto especializado en mantener correspondencias entre identificadores de objetos y su localización dentro del sistema.

Una vez asignado un identificador a un objeto, lo primero que hay que hacer es registrarlo en un servidor de nombres, de manera que quede constancia de su existencia y su localización inicial. Siempre que un objeto se mueva de una máquina a otra, el servidor de nombres habrá de ser advertido del cambio. Finalmente, si un objeto deja de existir en un momento dado, habrá que eliminar de los servidores de nombres toda información relativa a su localización.

Pueden utilizarse técnicas de replicación de las localizaciones de los objetos, repitiendo la información de localización en diferentes servidores. De esta manera se aumenta la fiabilidad y la escalabilidad.

También pueden utilizarse políticas de denominación de más alto nivel, mediante servidores de denominación[1] que asocien un nombre simbólico para un objeto con el identificador interno del mismo.

15.4.4.3 Comunicación entre objetos

Cuando se recibe una petición para enviar un mensaje a un objeto remoto, el servidor de localización se encarga de buscarlo en el sistema. Para acelerar el proceso, puede utilizar una memoria caché auxiliar en la que se almacene la localización de los objetos remotos más utilizados. Una vez encontrada la localización enviará un mensaje a la máquina destino solicitando que se invoque el método sobre el objeto. Cuando finalice la operación la máquina destino devolverá el resultado de la operación que le será entregado al objeto que inició el proceso.

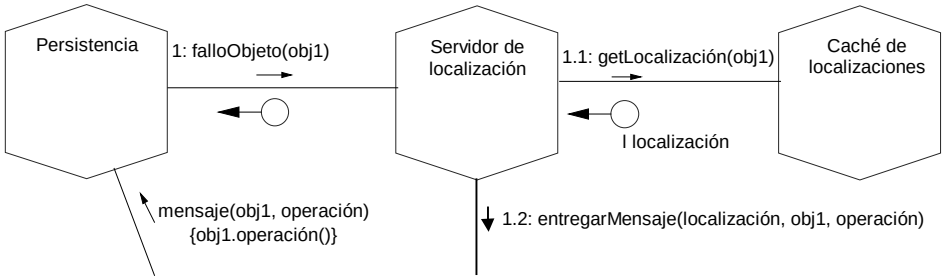


Figura 15.5 Entrega de un mensaje a un objeto remoto.

15.4.4.4 Movilidad

Los objetos del sistema tienen la posibilidad de ver variada su localización. Además de poder moverse a otra máquina de manera arbitraria, el sistema puede decidir mover los objetos por razones de equilibrio de carga. Un objeto del sistema operativo implementará la política de **migración de objetos** que se basará en dos aspectos:

- Carga de cada máquina
- Interacción con objetos remotos

Esto quiere decir que en el caso de que se detecte una alta carga en una máquina procesador o bien un alto grado de interacción entre objetos locales y remotos, se consultará

[1] A veces también se les denomina servicios de directorio.

con un objeto del sistema operativo. Éste estudiará qué objetos son susceptibles de ser movidos y con qué destino.

Una vez elegidos los objetos a mover y el destino de cada uno de ellos, lo primero que habrá que hacer será suspenderlos (ponerlos en un estado de inactividad). A continuación, se informará al servidor de localización de que el objeto va a ser movido y se enviará toda la información necesaria acerca del objeto (su estado encapsulado) a la máquina Carbayonia destino, donde se devolverán a su estado anterior.

15.4.4.5 Interoperabilidad con otros sistemas

Para permitir la interoperabilidad con otros sistemas operativos tradicionales habrá de implementarse una interfaz que permita dialogar a los objetos del sistema con los objetos definidos según otros modelos de objetos. En este momento, el modelo que ofrece la arquitectura más susceptible de ser adoptada desde Oviedo3 es el modelo CORBA [OMG95, OMG97] (aunque no se puede descartar en ningún momento interoperabilidad con otros, como COM [Rog96]). El desarrollo de una interfaz para CORBA no debe tener dificultad puesto que en el sistema integral el concepto nativo ya es el de objeto, y en consecuencia muy cercano a los conceptos de objetos de CORBA.

15.5 Concurrencia

El sistema de concurrencia [Taj96, TAA+97] deberá proporcionar un modelo de concurrencia que permita a los objetos la realización de tareas concurrentes.

15.5.1 Objetivos del sistema de concurrencia

Se trata de optimizar el grado de paralelismo de manera segura con la mayor simplicidad posible.

15.5.1.1 Optimización del grado de paralelismo de manera segura.

Se debe permitir la concurrencia entre objetos, entre métodos del mismo objeto y entre varias ocurrencias del mismo método de un objeto. Todo ello de la manera más segura posible, garantizando la corrección de las operaciones efectuadas y, por tanto, la consistencia del objeto.

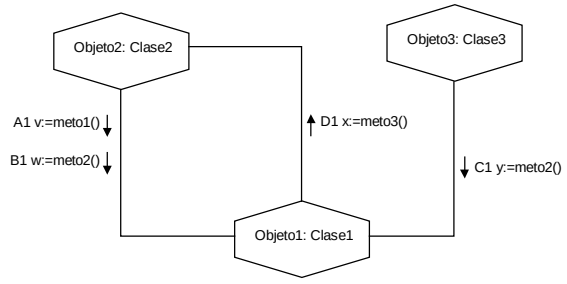


Figura 15.6 Llamadas concurrentes a un objeto.

15.5.1.2 Simplicidad.

Hacer que el modelo de concurrencia sea conceptualmente simple de entender para facilitar su uso, aunque ello implique una reducción del grado de paralelismo.

15.5.2 Elementos principales del modelo de concurrencia

El modelo se basa fundamentalmente en objetos activos multihilo y un sistema de invocación síncrona de métodos caracterizados como exclusivos o concurrentes.

15.5.2.1 Objetos activos multihilo

Los objetos del sistema deberán ser activos, para hacer que su estado encapsule también la computación. Conceptualmente dispondrán de un multiprocesador virtual para múltiples flujos de ejecución [CC91, Pap89], acompañado de los mecanismos de control de concurrencia necesarios para la preservación de integridad y la ejecución correcta de los hilos.

Los objetos activos unifican totalmente la propia computación dentro del modelo de objetos. Otros sistemas como Clouds [DLA+91] o Guide [DRK+89] usan un modelo de objetos pasivo. Esto obliga a proporcionar otra abstracción adicional como procesos o procesos ligeros (hilos).

15.5.2.2 Invocación síncrona

Por razones de simplicidad, se bloquea el hilo de un objeto cuando invoca un método de otro objeto [Pap89].

15.5.2.3 Métodos exclusivos y concurrentes

Cuando se define un método se clasifica como exclusivo o concurrente. Un método exclusivo modifica el estado del objeto y no puede coexistir con otro hilo del objeto. Los objetos se comportan como monitores cuando tratan con este tipo de métodos. Los métodos concurrentes no modifican el estado, así pues, sus hilos son compatibles con otros hilos concurrentes. El comportamiento de un objeto ante la llegada de una invocación de método debe tener esto en cuenta. Conceptualmente se puede pensar que los objetos de usuario usan un mecanismo similar al propuesto para una extensión de Eiffel [Car89]. Existirá un único punto de entrada al objeto donde se decide la creación de un nuevo hilo dependiendo del estado del objeto.

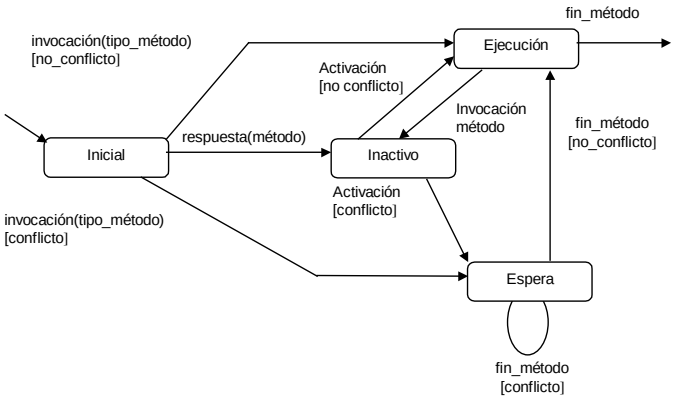


Figura 15.7 Comportamiento de un objeto ante la llegada de un mensaje.

Este tipo de aproximación a la concurrencia se ha vuelto muy popular, e incluso lenguajes como Java [AG96] tienen un modelo de concurrencia similar.

El modelo intenta lograr un equilibrio entre la maximización del paralelismo y la simplicidad conceptual. La invocación síncrona con métodos exclusivos y concurrentes es más sencilla de entender para el programador y el usuario del objeto. En cualquier caso, se mantiene un alto grado de paralelismo.

15.5.3 Implementación

Parte de la funcionalidad del modelo de concurrencia se implementará en la máquina y parte como objetos normales de usuario. La capacidad de ejecución de métodos y la caracterización de métodos como concurrentes o exclusivos se dejará a cargo de la máquina.

La máquina colaborará con objetos del sistema operativo, que implementarán las políticas de planificación.

15.6 Resumen

El sistema operativo SO4 se encarga de todos los cometidos relacionados con funcionalidad típica de sistema. Los más importantes son la aportación a los objetos de manera transparente de las propiedades de seguridad, persistencia, distribución y concurrencia.

Estas propiedades están en fase de desarrollo por otros investigadores. Pueden describirse los elementos fundamentales en el estado preliminar de las mismas. La seguridad se basa en un mecanismo de control de acceso mediante capacidades. La persistencia extiende transparentemente el área de instancias de la máquina con el almacenamiento secundario para formar un área de instancias virtual. La distribución permite la transparencia de acceso y localización y la movilidad de objetos y el equilibrio de la carga. El modelo de concurrencia basado en objetos activos multihilo con invocación síncrona logra un equilibrio entre la maximización del paralelismo y la sencillez conceptual. Se busca que estas propiedades se integren con fluidez en el modelo del sistema, sin modificaciones o al menos que estas sean menores.

La implementación de estas propiedades por el sistema operativo se hace fundamentalmente mediante objetos de usuario. Ciertas características del modelo, la máquina abstracta y la reflectividad, como el identificador único de objetos, facilitan el diseño e implementación de estas propiedades.

Capítulo 16
SOPORTE PARA PERSISTENCIA

Uno de los elementos más importantes que debe proporcionar el sistema operativo es la propiedad de la persistencia de manera transparente al resto de los objetos del sistema. La persistencia es una propiedad que ya se viene considerando como parte integrante del modelo de objetos [Booc94].

En este capítulo se hace un estudio de esta propiedad y las ventajas en general que aporta en un sistema. Posteriormente se describen los diferentes elementos que influyen en la manera de implantar la persistencia para un sistema de objetos. Se proponen los puntos fundamentales de diseño del sistema de persistencia para el sistema integral y con las ventajas que aporta al sistema este diseño, así como los elementos del sistema que facilitan la implantación de la persistencia. Por último se hace un diseño preliminar de la implementación de la persistencia en el sistema.

16.1 Persistencia ortogonal

En un sistema de computación es necesario que cierta información se mantenga durante un plazo de tiempo superior al que permanezca conectado el sistema. En sistemas convencionales se suelen utilizar los ficheros para lograr esta persistencia de la información. El concepto de **persistencia ortogonal** fue introducido por Atkinson [ABC+83]:

Persistencia: Propiedad por la que la información del sistema puede persistir (sobrevivir) durante el tiempo que sea requerida.

Ortogonal: Toda la información puede ser persistente y debe ser manipulada de la misma manera, independientemente del tiempo que deba persistir.

Persistencia completa: Cuando la persistencia ortogonal es aplicada a *toda* la información del sistema.

Los sistemas que soportan persistencia ortogonal tienen una serie de ventajas sobre los sistemas tradicionales, fundamentalmente al proporcionar una abstracción uniforme del almacenamiento.

16.1.1 Abstracción uniforme del almacenamiento

En un sistema con persistencia ortogonal completa, toda la información, independientemente de su duración en el sistema, se trata de la misma manera.

En los sistemas convencionales existe una dualidad en el manejo de la información en tiempo de ejecución de los procesos y el mantenimiento de la misma a largo plazo. Para el primer caso se utiliza la memoria principal (memoria virtual) y en el segundo la memoria secundaria (ficheros).

En estos sistemas es común que existan dos conjuntos de operaciones para tratar la información dependiendo del tiempo que deba persistir la misma. La información que debe persistir a largo plazo se accede de diferente manera que la de corto plazo. Normalmente la de

largo plazo se mantiene en un sistema de ficheros o base de datos (almacenamiento no volátil - secundario) y la de corto plazo es tratada por un programa en un lenguaje (almacenamiento volátil - principal).

Esto obliga a que el programador se ocupe de realizar la transferencia de información entre el almacenamiento a corto plazo (memoria del proceso) al almacenamiento a largo plazo (fichero) y viceversa. Además, la representación de la información es diferente en ambos entornos, con lo que hay que ocuparse de las consecuentes traducciones de formato, con los conocidos problemas de desadaptación de impedancias [ABB+93], en este caso "aplanamiento" de estructuras de datos, formatos de ficheros, etc.

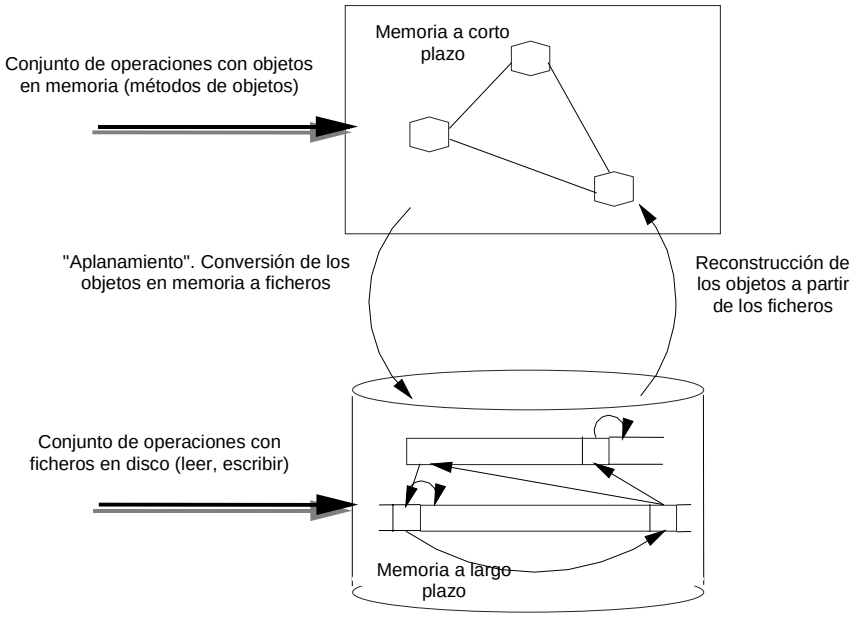


Figura 16.1 Dualidad de abstracciones y problemas de “aplanamiento” y conversión de formatos en los sistemas tradicionales.

Un sistema persistente eliminaría esta dualidad, al tratar por igual todos los objetos y permitir que se acceda a los mismos siempre del mismo modo. El sistema de persistencia debe gestionar de manera transparente la persistencia de los mismos, es decir, el paso entre el almacenamiento a corto plazo y el de largo plazo. El programador sólo utiliza un conjunto de instrucciones para manipular la información.

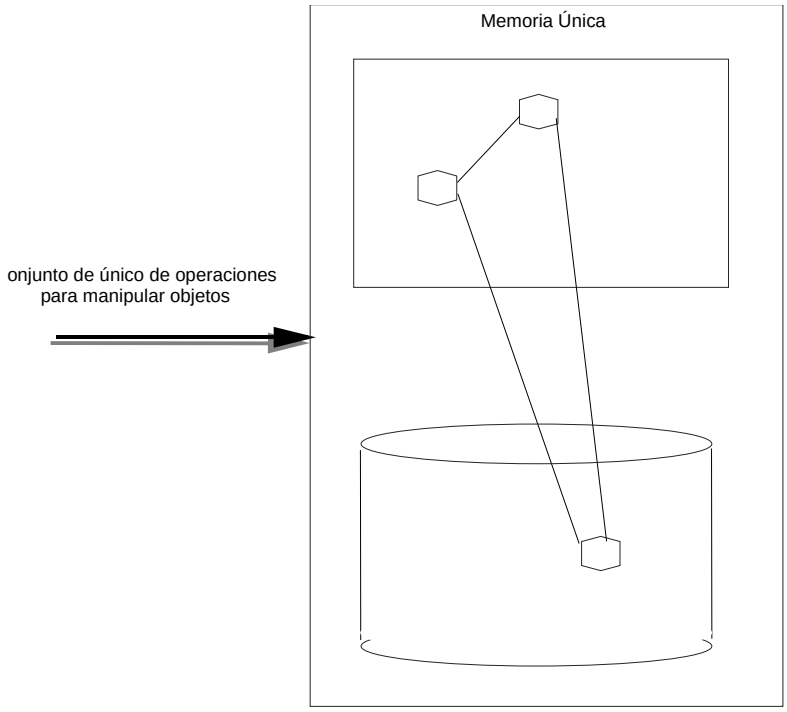


Figura 16.2 Abstracción única sobre la memoria en un sistema con persistencia.

Desde este punto de vista, la persistencia ortogonal proporciona una abstracción uniforme sobre ambos tipos de almacenamiento: principal y secundario.

16.1.2 Estabilidad y elasticidad

Es necesario proteger la información existente en un sistema frente a posibles accidentes, como por ejemplo pérdidas de información debidas a cortes en el suministro eléctrico. En este aspecto, dos propiedades deseables en un sistema con persistencia son la estabilidad y la elasticidad o recuperación (*resilience*[1]) [DRH+92].

Estabilidad: Es la capacidad de un sistema para registrar su estado (*checkpointing*, puntos de verificación) de manera consistente en un medio seguro, de tal manera que el funcionamiento podría reanudarse a partir de ese punto en el futuro.

Elasticidad: Un sistema tiene esta propiedad si puede reanudarse el funcionamiento con seguridad después de una caída del sistema inesperada, como por ejemplo un fallo de alimentación.

Estas propiedades deberían estar presentes en un sistema con persistencia, por ejemplo, un editor debería ser elástico, en caso de un fallo inesperado, al reanudarse el sistema, se debería mantener todo o la mayor parte del documento que se estuviera editando.

[1] Poder de recuperación.

16.2 Sistemas de persistencia de objetos

La mayoría de los sistemas persistentes se han construido por encima de sistemas operativos ya existentes. Estos sistemas operativos convencionales no tuvieron entre sus objetivos de diseño el soporte para la persistencia. De hecho la mayoría de los sistemas operativos únicamente tienen los ficheros como abstracción de la memoria a largo plazo [DRH+92].

El resultado es que normalmente se implementa el sistema de persistencia como una capa completa adicional al sistema operativo, con poca integración con el mismo y la pérdida de rendimiento consiguiente. Por otro lado, muchos sistemas se enfocan a dar soporte de persistencia a un determinado lenguaje [ACC81] específicamente. De esta manera se van añadiendo capas adicionales, una por cada lenguaje, con los problemas de la proliferación de capas de software (véase el capítulo 2).

Otro inconveniente de estos sistemas de persistencia es que utilizan un concepto de objeto muy reducido: normalmente lo tratan simplemente como una zona contigua de memoria [VRH93]. En general se trata a los objetos simplemente como almacenamiento de información, sin tener en cuenta posible semántica adicional que podría contener (encapsulamiento de la computación, relaciones con otros objetos, etc.). En otros casos el sistema está muy ligado al hardware y al uso de conceptos de SO más tradicionales (paginación, procesos, etc.).

Por todo ello parece claro que el camino a seguir es implementar la persistencia como propiedad fundamental de un sistema integral orientado a objetos. Por supuesto, se deben soportar los objetos con toda la semántica del modelo de objetos del sistema: no sólo los datos estrictamente, si no con una semántica más amplia, que incluya como parte del estado de un objeto la computación, las relaciones con otros objetos, etc. De esta manera el soporte de persistencia estará disponible para todos los lenguajes, sin necesidad de un soporte adicional redundante para los mismos.

16.2.1 Indicación de objetos persistentes

El sistema necesita distinguir cuáles son los objetos persistentes (si es que no lo son todos), por ello es fundamental definir cómo se sabe si un objeto es persistente o no, hay varios métodos que se pueden combinar [BM92]:

16.2.1.1 Volcado de memoria

Al finalizar el programa, se vuelca completamente el mapa de memoria del mismo. Cuando se necesita alguno de esos datos, se carga de nuevo la información, como el sistema Smalltalk [GR83].

16.2.1.2 Marcas explícitas

El sistema de persistencia identifica los objetos persistentes mediante unas marcas explícitas que se colocan en los objetos que deben persistir.

16.2.1.3 Accesibilidad.

Son persistentes aquellos objetos que pueden ser accedidos desde una raíz de persistencia dada, bien directamente o indirectamente. Esto necesita que el sistema pueda detectar las referencias a otros objetos contenidas dentro de los objetos.

16.2.1.4 Persistencia completa

Todos los objetos son persistentes siempre. No es necesario indicar de manera especial cuáles son los objetos persistentes puesto que absolutamente todos los objetos del sistema son persistentes. Esta parece la mejor solución, para evitar posibles dualidades de tratamiento en el sistema entre objetos persistentes y aquellos que no lo son, manteniendo así la uniformidad.

Muchos sistemas utilizan mezclas de alguna de las primeras técnicas, ya que no necesariamente son excluyentes entre sí.

16.2.2 Eliminación de objetos persistentes

La persistencia hace que los objetos permanezcan en el sistema de manera indefinida en principio. Sin embargo, puede llegar el momento en que no se necesite más un objeto, para lo cual el sistema debe proceder a su eliminación. La eliminación de los objetos, cuando ya no sean necesarios, puede realizarse de una de dos maneras:

16.2.2.1 Borrado explícito

Se indica de manera explícita cuando ya no se necesita el objeto.

16.2.2.2 Recolección de basura

Cuando un objeto ya no es referenciado por ningún otro, se elimina transparentemente. La gestión de esta modalidad es más complicada aunque la programación es más sencilla al no tener que ocuparse del borrado explícito de los objetos. Suele ir asociada a mecanismos de identificación de objetos persistentes por accesibilidad.

Ambos sistemas pueden ser utilizados simultáneamente en un sistema persistente, combinando la eliminación explícita de los objetos con una eliminación implícita cuando ya no puede accederse a los mismos.

16.2.3 Identificadores de objetos en almacenamiento persistente

El sistema de persistencia debe gestionar de manera transparente el movimiento de los objetos entre el almacenamiento primario cuando sea necesario su uso y el secundario (almacenamiento persistente). Cuando un objeto está en almacenamiento persistente, debe tener un identificador único que permita al sistema localizarlo dentro de ese almacenamiento:

16.2.3.1 Identificador hardware

Puede ser tratado por el hardware con más eficiencia. Por ejemplo una dirección de memoria virtual [VSK+90]. Un posible problema es que el rango de direcciones virtuales no sea suficiente para soportar todos los objetos.

16.2.3.2 Identificador software

Un identificador generado por software, sin relación necesaria con las circunstancias físicas del objeto. Por ejemplo un nombre único [BC85].

Debe tenerse en cuenta que este identificador en almacenamiento persistente no necesariamente coincidirá con el usado en el almacenamiento primario.

16.2.4 Relaciones entre objetos. Identificadores de objetos

Se deben mantener las relaciones entre los objetos exactamente igual que si estuvieran en memoria principal. Es decir, las relaciones (punteros) existentes entre objetos deben mantener siempre su significado, independientemente de dónde resida en cada momento el objeto. Esto se logra utilizando los identificadores de objetos.

Con referencia a este aspecto es importante si el identificador de los objetos en almacenamiento persistente es uniforme o no. Es decir, si el identificador en almacenamiento persistente coincide con el identificador que se usa para acceder a un objeto en memoria:

16.2.4.1 Identificador no uniforme

Cuando se pasa el objeto de memoria persistente a memoria principal hay que convertir el identificador en almacenamiento persistente del objeto al identificador usado en memoria principal (transformación de punteros, *pointer swizzling*). Este proceso hay que realizarlo con el resto de los identificadores de otros objetos (relaciones) que se encuentren en el objeto.

16.2.4.2 Identificador uniforme

El objeto tiene siempre el mismo identificador, independientemente de su localización. En este caso el identificador se conoce como una referencia al objeto, que siempre es válida. No hay que hacer transformaciones para mantener las relaciones entre los objetos.

16.2.5 Memoria virtual distribuida

La utilización de un identificador uniforme parece más adecuada, por su sencillez conceptual. De esta manera el sistema de persistencia proporciona una abstracción de un único espacio de memoria virtual persistente en el que residen los objetos, a los cuales se puede acceder únicamente mediante su referencia.

El sistema de manera transparente trae los objetos a la memoria física del ordenador si no está ya en ella. Sin embargo, la percepción del usuario es un gran espacio de memoria virtual donde están los objetos, que siempre se manejan de la misma manera.

Si el identificador es único para un sistema distribuido, el almacenamiento persistente en conjunto podría formar una memoria virtual distribuida [RSE+92]. Cualquier objeto puede ser accedido desde cualquier punto a través de su identificador (referencia).

16.2.6 Tamaño del almacenamiento

Idealmente el tamaño del almacenamiento persistente (memoria virtual) debería ser infinito. En la práctica esto no es así, dependiendo fundamentalmente de la implementación. Por ejemplo, si se utilizan direcciones virtuales de 32 bits como identificadores de objetos el espacio total disponible es más reducido.

16.3 Persistencia para el sistema integral orientado a objetos

En un sistema integral orientado a objetos (SIOO) todos los elementos que se manejan son objetos. Por tanto, un sistema con persistencia completa haría que todos los objetos existentes en el sistema fueran persistentes

Se elabora a continuación una propuesta preliminar para incorporar la persistencia en el sistema integral orientado a objetos.

16.3.1 Elementos básicos de diseño del sistema de persistencia

En esencia se trata de proporcionar la abstracción de un espacio de objetos potencialmente infinito en el que coexisten simultáneamente e indefinidamente todos los objetos del sistema, hasta que ya no son necesitados. Podría considerarse como una memoria virtual persistente para objetos. El usuario simplemente trabaja con los objetos en ese espacio virtual.

El sistema de persistencia deberá proporcionar con los recursos existentes en el entorno (a partir de la máquina abstracta orientada a objetos Carbayonia) la abstracción anterior.

Los puntos básicos propuestos para el sistema de persistencia del sistema integral se refieren a continuación.

16.3.1.1 Persistencia completa

Absolutamente todos los objetos creados en el sistema son persistentes por definición. Esto incluye también a los objetos del sistema operativo, que deben tener la misma categoría que los objetos de usuario. No es necesario almacenar y recuperar explícitamente los objetos, el sistema lo realiza de manera transparente. Sólo existe un único conjunto de operaciones para manipular los objetos.

16.3.1.2 Estabilidad y elasticidad

El espacio virtual persistente de objetos debe proporcionar las propiedades de estabilidad y elasticidad.

16.3.1.3 Encapsulamiento de la computación

Se propone un modelo de objetos que además encapsule como parte de su estado la computación, que tendrá que ser perfilado por la parte de concurrencia del sistema operativo.

16.3.1.4 Identificador uniforme único

Se utilizará el identificador único de objetos de la máquina para referenciar los objetos, independientemente de su localización física. Este identificador es usado tanto por el usuario como por el sistema internamente para referenciar el objeto en memoria principal y en almacenamiento persistente. Esto tiene mucha similitud con el modelo de SO de espacio de direcciones virtual único [VRH93, SMF96], jugando el papel de las direcciones virtuales los identificadores de los objetos. Este espacio único se extiende a la memoria secundaria, formando un sistema de almacenamiento con sólo un nivel (*single level store* [KEL+62]).

16.3.2 Ventajas

La utilización de un esquema como el que se ha propuesto conlleva unas ventajas, unas particulares y otras propias de los sistemas de persistencia ortogonal completa en general:

16.3.2.1 Permanencia automática

Una vez que se crea un objeto en el sistema, este permanece el tiempo necesario hasta que no sea necesitado más. No es necesario preocuparse de almacenar el objeto en almacenamiento secundario si se desea que no se pierda su información. Esto facilita la programación.

16.3.2.2 Abstracción única de memoria. Uniformidad

La única abstracción necesaria para la memoria es el objeto. El sistema de manera transparente conserva la información del objeto siempre. Al eliminar la dualidad de la abstracción entre memoria de largo plazo y de corto plazo, y sustituirlas por una única abstracción de espacio virtual persistente de objetos se facilita la labor de los programadores. Abstracciones como fichero ya no son necesarias (aunque podrían implementarse como objetos normales, si se desea). Simplemente se trabaja con un único paradigma, que es el de la orientación a objetos.

16.3.2.3 Permanencia de la computación

Al encapsular el objeto también el procesamiento, no sólo se conservan los datos, si no también la computación [KV92, TYT92]. Se mantiene totalmente el estado: datos y proceso. Esta característica se suele considerar ya parte integrante de un modelo de objetos [Booc94].

16.3.2.4 Entorno de computación continuo

Al ser la persistencia ortogonal completa, con estabilidad y elasticidad, se obtiene un entorno continuo, al persistir los objetos del usuario (incluyendo la computación). No son necesarias acciones de registro de entrada (*log-in*), y de salida (*log-out*), como cargar un fichero al iniciar un programa y salvarlo para mantener los cambios. El estado del entorno se mantiene. Se podría dejar el entorno en un estado (por ejemplo editando un fichero), desconectar, y al volver a conectar el estado sería exactamente el mismo (al persistir todo). Esto contrasta con los sistemas actuales en los que cada vez que se conecta hay que reconstruir todo el entorno (arrancar procesos, cargar ficheros, etc.)

16.3.2.5 Sistema continuo

Al ser también objetos normales los objetos del propio SO, son a su vez persistentes. Se logra un sistema continuo. Cuando se apaga el sistema, se mantiene totalmente su estado (una fotografía del mismo): objetos en uso, estado de la computación, etc. Al arrancar de nuevo el sistema se continúa exactamente en el mismo punto en que se apagó. El sistema siempre está "vivo", no hay que hacer que "renazca" cada vez que se arranca ("rebotar" el sistema). Por tanto, se mantiene totalmente el estado del sistema entre interrupciones del mismo, tanto debidas al usuario (desconexión), como inesperadas (fallo de alimentación). El usuario percibe el sistema de una manera más intuitiva y cercana al comportamiento de los objetos del mundo real.

16.3.2.6 Interfaces más intuitivos

Un sistema y entorno continuo permiten interfaces de usuario más intuitivas, al eliminar la pobre abstracción que proporcionan los ficheros y las poco intuitivas acciones necesarias para mantener el estado al desconectar [TYT92]. Por ejemplo, interfaces del tipo "escritorio" se comportarían de una manera más intuitiva y cercana a su funcionamiento en el mundo real. Cuando se abandona el escritorio (desconexión), su estado permanece exactamente igual cuando se regresa a él (conexión), debido a la persistencia. No requiere la existencia de conceptos poco intuitivos como fichero, configuraciones de arranque[1], etc.

16.3.2.7 Memoria virtual persistente distribuida

Se crea fácilmente una verdadera memoria virtual distribuida de objetos al utilizar un identificador único de objetos. Simplemente basta con utilizar un mecanismo que haga que este identificador sea diferente para todos los objetos, independientemente de la máquina en que se crearon. Se utilizará este identificador único en conjunción con el mecanismo de distribución para localizar en todo el sistema distribuido un objeto dado.

16.3.2.8 Eficiencia interna

En principio debe esperarse una buena eficiencia interna al utilizar un identificador uniforme y no necesitar de la sobrecarga del mecanismo de transformación de punteros (*pointer swizzling*), en el paso entre memoria principal y almacenamiento persistente. Por otro lado al usarse como identificador el propio identificador hardware de la máquina, la eficiencia debe ser mayor al no introducir el paso intermedio debido a un identificador software.

[1] Como el AUTOEXEC.BAT del sistema operativo MS-DOS o el .login de UNIX.

16.4 Implementación de la persistencia en el sistema integral

Se dispone de la máquina orientada a objetos Carbayonia, que está basada en la existencia de objetos (con un identificador único siempre válido) y en el uso de referencias a objetos. Estas referencias almacenan los identificadores únicos de objetos.

Las referencias pueden considerarse como punteros a otros objetos, aunque se diferencian de los punteros convencionales por el hecho de que siempre son válidos, al utilizarse el identificador único del objeto. Al no utilizar como referencias valores físicos que pueden cambiar (como posiciones de memoria), no hay que preocuparse de posibles movimientos de objetos dentro del espacio de trabajo, etc.

El elemento fundamental de la máquina es el área de instancias, que almacena los objetos. Esta área es el equivalente a la memoria principal de los sistemas convencionales.

16.4.1 Área de instancias virtual persistente

En esencia, se propone implementar la persistencia como una extensión del área de instancias. Es decir, lograr la ilusión de un **área de instancias virtual**[1] (**memoria virtual**), utilizando para ello un almacenamiento secundario para hacer persistir los objetos, guardándolos cuando no estén (o no quepan) en el área de instancias. Todo ello de manera totalmente transparente para el resto del sistema.

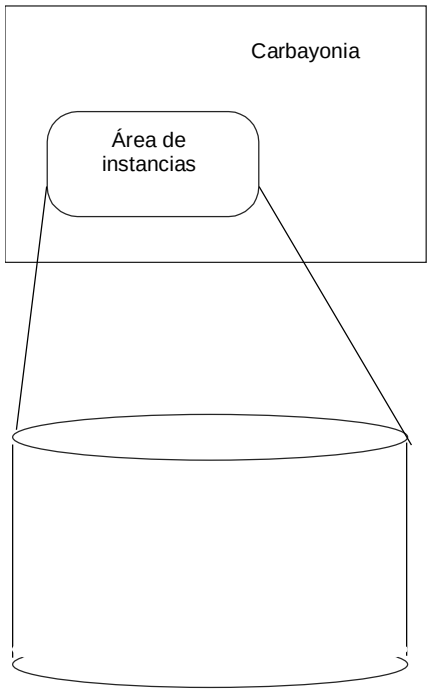


Figura 16.3 Área de instancias virtual.

Se unifican los principios de memoria virtual tradicionales de los sistemas operativos [Dei90] con los de persistencia. El sistema proporciona un espacio de memoria único en el que residen todos los objetos (persisten), virtualmente infinito.

[1] Hay que recalcar que, dentro de la semántica de un objeto, un elemento fundamental es la clase a la que pertenece. Por tanto, aunque se haga referencia únicamente a los objetos, se entiende implícitamente que los mismos mecanismos se aplican para hacer persistir las clases de los objetos. Es decir, la persistencia se aplica de manera análoga al área de clases.

Una posibilidad es que sea el sistema operativo el que intervenga de manera reflectiva en el funcionamiento de la máquina para lograr lo anterior.

16.4.2 Identificador uniforme de objetos igual al identificador de la máquina

Por otro lado, como identificador del objeto en almacenamiento persistente se utilizará de manera uniforme el propio identificador del objeto dentro de la máquina. Es decir, se usa en todo momento un único identificador del objeto, que siempre es válido en cualquier situación en la que se encuentre el objeto: tanto en almacenamiento persistente como en el área de instancias.

16.4.3 Mecanismo de envío de mensajes y activación del sistema operativo por reflexión explícita

Cuando se envía un mensaje a un objeto (en general cuando se necesite acceder a un objeto por alguna razón), se proporciona a la máquina una referencia al mismo (identificador del objeto). La máquina usa este identificador para localizar el objeto en el área de instancias. Si el objeto no está en esta área debe generarse una excepción o mensaje que permita intervenir al sistema operativo. Mediante la reflectividad se hace una reflexión explícita (véase el capítulo 14) que llamará a un objeto del sistema operativo.

16.4.4 Objeto "paginador"

Se activará un objeto colocado al efecto por el sistema operativo. La función de este objeto es precisamente ocuparse del trasiego de los objetos entre el área de instancias el almacenamiento persistente. Este objeto y el mecanismo en general son similares al concepto del paginador externo para memoria virtual de sistemas operativos como Mach [ABB+86], por tanto, se le denominará objeto "**paginador**".

16.4.5 Carga de objetos

Para localizar el objeto en almacenamiento persistente utilizará el identificador del objeto proporcionado por la referencia usada en el envío del mensaje.

Una vez localizado el objeto, se debe colocar en el área de referencias, reconstruyendo en efecto el estado completo del mismo. Para ello se invocarán métodos adecuados en la meta-interfaz del objeto que representa reflectivamente el área de instancias (cosificación).

Al finalizar el trabajo del objeto "paginador", debe regresar el control a la máquina para que continúe con su funcionamiento normal, al ya estar el objeto en el área de referencias.

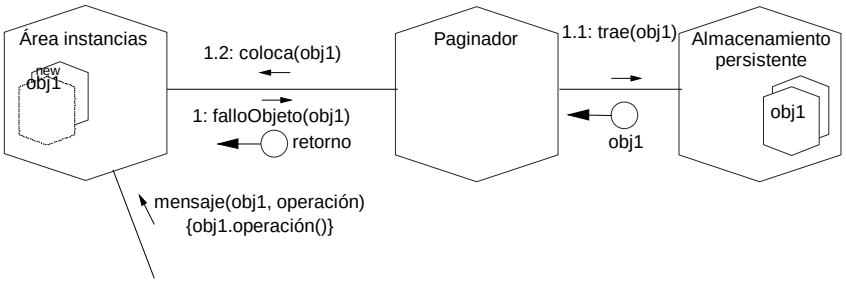


Figura 16.4 Carga de un objeto en el área de instancias.

16.4.6 Reemplazamiento

En caso de no existir sitio en el área de referencias para situar un objeto, bien en la creación de un nuevo objeto, bien por necesitarse la carga del mismo por el motivo anterior,

debe liberarse espacio en la misma. Por un mecanismo similar al anterior se activará el objeto paginador y seleccionará uno o varios objetos que llevará al almacenamiento persistente, registrando su estado y liberando así el espacio necesario. Para seleccionar los objetos a reemplazar puede utilizarse otro objeto que proporcione la estrategia de reemplazo.

Hay que destacar que si el objeto encapsula la computación, también se conserva el estado de la misma.

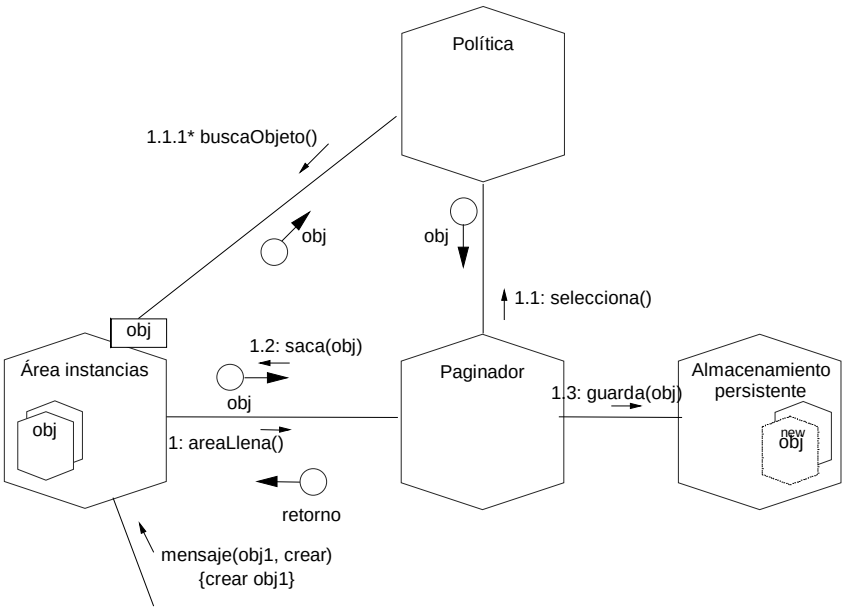


Figura 16.5 Reemplazamiento de un objeto en el área de instancias.

16.4.7 Manipulación interna

Es necesario habilitar un mecanismo que permita a este objeto del sistema manipular el área de instancias, para lo cual se aprovecha la arquitectura reflectiva de la máquina. Algo similar debe hacerse para poder registrar el estado de los objetos, al tener que acceder a su representación interna, bien por el método anterior, bien definiendo unas operaciones primitivas disponibles en todo objeto que lo permitan.

16.4.8 Estabilidad y elasticidad

El objeto “paginador” implementará las propiedades de estabilidad y elasticidad. Su intervención puede ser requerida en cualquier momento, para lograr estas propiedades. Es decir, se puede registrar el estado de un objeto en el almacenamiento persistente sin necesidad de eliminarlo del área de instancias, simplemente para estabilizar el mismo.

16.5 Resumen

La persistencia es un propiedad que aporta el sistema operativo a los objetos del sistema integral, permitiendo lograr una abstracción única de memoria en el sistema, eliminando la desadaptación de impedancias con el almacenamiento secundario.

Se propone para el sistema integral una persistencia completa, que se aplica a todos los objetos del sistema de manera transparente, con estabilidad y elasticidad que permitan recuperarse de caídas del sistema y el uso uniforme del identificador de objetos que evita transformaciones de punteros entre el almacenamiento secundario y el persistente. Esto permite eliminar abstracciones no necesarias para los programadores, como ficheros, ya que

sólo se usarán objetos de manera uniforme, que persistirán de manera automática. Al aplicarse a todos los objetos del sistema se logra un entorno continuo, que permanece igual entre desconexiones y permite interfaces de usuario más intuitivos.

La idea fundamental para la implementación sobre la máquina abstracta es la creación de un área de instancias virtual que extienda el área de instancias al almacenamiento persistente. Para ello los objetos de la máquina colaborarán de manera reflectiva con objetos del sistema operativo por medio de un “paginador” de objetos. Este objeto “paginador” se encarga de colaborar con la máquina para extender el área de instancia. Trae los objetos del almacenamiento persistente al área de instancias y viceversa, de manera transparente, en los casos de fallo de objeto (el objeto no está en el área de instancias cuando lo necesita la máquina) y reemplazo (no hay sitio en el área de instancias y se debe liberar espacio moviendo objetos al almacenamiento persistente).

Capítulo 17

ASPECTOS ADICIONALES RELACIONADOS CON LA PERSISTENCIA

En este capítulo se discuten aspectos adicionales relacionados con la persistencia, tanto aspectos relacionados con el diseño e implementación de la misma, como ventajas que puede aportar el sistema de persistencia del sistema integral sobre otros sistemas.

17.1 Eliminación de objetos explícita y recolección de basura

Hay que decidir el mecanismo de eliminación de objetos: por **borrado explícito** o bien por **recolección de basura**. La recolección de basura siempre implica una sobrecarga adicional en el sistema, aunque para el caso de la memoria principal es aceptable.

En el caso de la persistencia, se complica la implementación de la recolección de basura al poder residir los objetos en el almacenamiento persistente y en el área de instancias normal de la máquina. El tener que realizar la recolección de basura en memoria secundaria es complejo y más costoso.

Por otro lado, si se trabaja en un sistema distribuido, los objetos pueden residir en cualquier nodo. La recolección de basura tendría que ser distribuida, lo que lo hace aún más complicado y costoso.

El borrado explícito es un mecanismo que siempre debe estar presente, puesto que determinadas aplicaciones necesitan tener la seguridad de poder eliminar un objeto en cualquier momento, independientemente de que esté siendo usado por otros objetos [Mal96]. Por ejemplo, en el caso de un sistema bancario, si por alguna razón se decide cerrar una cuenta, debe poderse eliminar explícitamente la misma, puesto que ya no se va a poder usar más, aunque otros objetos sigan manteniendo referencias a la misma.

Pueden utilizarse ambas estrategias, permitiendo por ejemplo el borrado explícito por parte del creador del objeto o bien protegiendo esta operación con el sistema de protección. Inicialmente se puede utilizar el borrado explícito y considerar la posibilidad de añadir posteriormente la recolección de basura.

17.2 Recolección de basura no necesaria

La recolección de basura, sobre todo en un sistema con persistencia y distribución transparente como el sistema integral, es una tarea compleja y costosa, aunque se están realizando avances [FS94]. Sin embargo, la arquitectura de la persistencia en el sistema integral junto con la aparición de medios de almacenamiento masivo más baratos y de más capacidad pueden hacer innecesaria la recolección de basura.

17.2.1 Recolección de basura en memoria principal (área de instancias) no necesaria

En el sistema integral la persistencia será completa. Todos los objetos que existen en el sistema tendrán la propiedad de la persistencia por definición. Existe un único espacio de objetos en el que residen permanentemente los objetos hasta que son borrados. El sistema de persistencia se ocupa gestionar el área de instancias de la máquina, trayendo y llevando los objetos entre esa área y el almacenamiento persistente a medida que son necesitados.

Por tanto, no es necesaria la recolección de basura en el área de instancias. Un objeto puede quedar sin referencias y estar en el área de instancias. Sin embargo, la gestión de la misma al estilo de la memoria virtual hará que tarde o temprano se necesite espacio en el área para trabajar con otros objetos y para liberar espacio este objeto "huérfano" acabará pasando al almacenamiento persistente.

El área de instancias puede considerarse como una memoria caché, o una ventana sobre el espacio global del área de instancias virtual.

17.2.2 Sustitución de la recolección de basura por almacenamiento terciario

Como se acaba de ver el sistema de persistencia hace innecesaria la recolección de basura en el área de instancias. En su lugar habría que realizarla en el área de instancias virtual, el almacenamiento persistente.

Al igual que puede verse el área de instancias como una caché del almacenamiento persistente en memoria secundaria, puede añadirse un **tercer nivel de almacenamiento** y aplicar el mismo mecanismo. Es decir, los objetos residen en el almacenamiento secundario. Cuando un objeto lleva demasiado tiempo sin usarse en el almacenamiento secundario[1], se pasa a almacenamiento terciario.

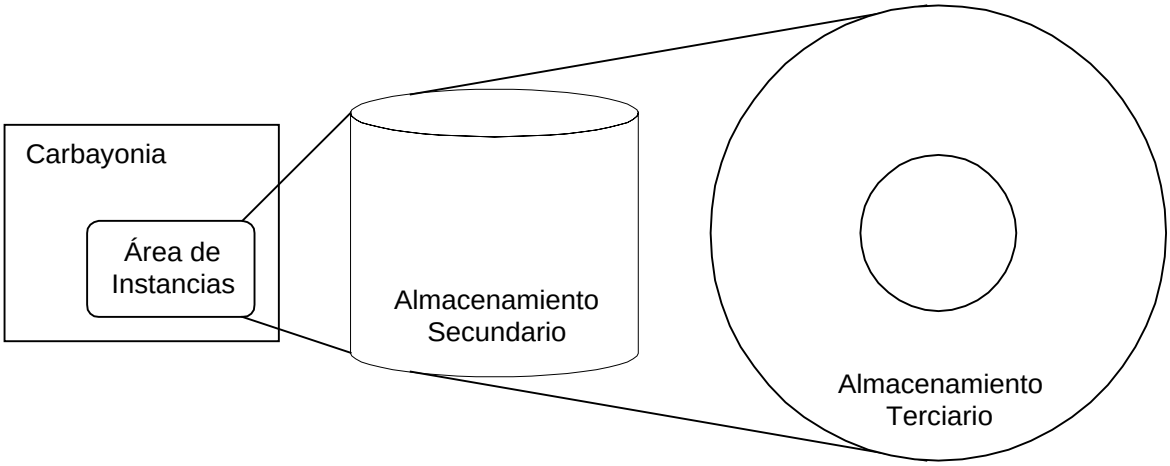


Figura 17.1 Adición de un nivel más de almacenamiento terciario al sistema de persistencia.

En caso de que el objeto fuera referenciado de nuevo, se traería de nuevo del almacenamiento terciario. En el caso de un objeto que ya no tuviera referencias, no se realiza la recolección de basura y se quedaría para siempre en el almacenamiento terciario pues nunca sería referenciado de nuevo.

[1] Posiblemente por ser un objeto "huérfano" al que ya no se le hace referencia.

Es decir, en lugar de buscar los objetos que ya no tienen referencias (hacer recolección de basura) se deja que el objeto vaya a parar al almacenamiento terciario[1].

17.2.3 Costo de la eliminación de la recolección de basura

Por tanto, en lugar de recolectar un objeto, este acabará en el almacenamiento terciario, ocupando permanentemente espacio allí[2]. Este será el costo a pagar por la eliminación de la recolección de basura.

17.2.3.1 Razones que hacen que el costo no sea grande

Sin embargo, este costo es menor de lo que parece por algunas razones que se enumeran a continuación.

Borrado explícito de objetos

La mayoría de los objetos pueden ser borrados de manera explícita, pero automática sin el control del programador, bien por los compiladores, bien por determinados elementos del sistema. Por ejemplo, el propio modelo de objetos de la máquina elimina los objetos locales de un método (Instances). Otro tipo de uso de objetos locales puede ser detectado por un compilador, eliminándose automáticamente cuando no se necesiten.

Bajo coste del almacenamiento terciario

Este tipo de almacenamiento tiene una capacidad muy grande y un coste pequeño. Esto hace que el espacio que se pueda desperdiciar sea tan barato que compense con creces no realizar recolección de basura.

De hecho, en el sistema Plan9 [PPT+92] se utiliza un esquema parecido, con una jerarquía de memoria en tres niveles para el espacio de ficheros. La memoria principal es una caché sobre el espacio en disco, que a su vez es una caché para la memoria terciaria, compuesta por una batería de unidades WORM (*Write Once Read Many Times*, Una Escritura Múltiples Lecturas). Cada día se guardan en el WORM todos los cambios producidos en el disco. Es decir, hay suficiente memoria terciaria para guardar todos los cambios producidos históricamente en los ficheros del sistema.

Recolección de basura reducida

En cualquier caso se facilita la recolección de basura identificando a priori los objetos candidatos a ser recolectados. Estos serán los que se hayan decantado hasta el almacenamiento terciario, con lo que no habrá que recorrer todo el espacio de objetos buscando posibles objetos a recolectar, se parte de los que son candidatos a ello.

Parece, pues, que este tipo de arquitectura elimina la necesidad de costosos mecanismos de recolección de basura, que en el caso de sistemas persistentes y distribuidos aún no están totalmente solucionados. En cualquier caso, es necesario investigar en más profundidad este aspecto, para confirmar experimentalmente esto.

---

[1] Al igual que en el área de instancias un objeto que no se use acabará pasando a memoria secundaria, de esta, si sigue sin usarse (por ser "huérfano"), acabará pasando al almacenamiento terciario.

[2] Haciendo una analogía, si la recolección de basura lo que hace es incinerar los desechos, haciéndolos desaparecer, este mecanismo haría que fueran a un vertedero en almacenamiento terciario.

17.3 Funcionamiento conjunto del mecanismo de invocación de métodos, la persistencia, la seguridad y la distribución

La parte de persistencia tiene relación con la parte de seguridad y protección y el mecanismo de envío de mensajes o invocación de métodos. Puede verse como una conjunción de estas partes el funcionamiento del sistema: la invocación de los mensajes indica el objeto, la parte de persistencia proporciona el objeto y la parte de seguridad comprueba la validez del acceso. Idealmente el resto de los componentes no deben ser conscientes de la existencia de la parte de persistencia. Todos los objetos del sistema ven una memoria virtual o espacio de objetos infinito. Esto también se extiende a la distribución, que permitiría el envío de mensajes entre objetos de diferentes máquinas.

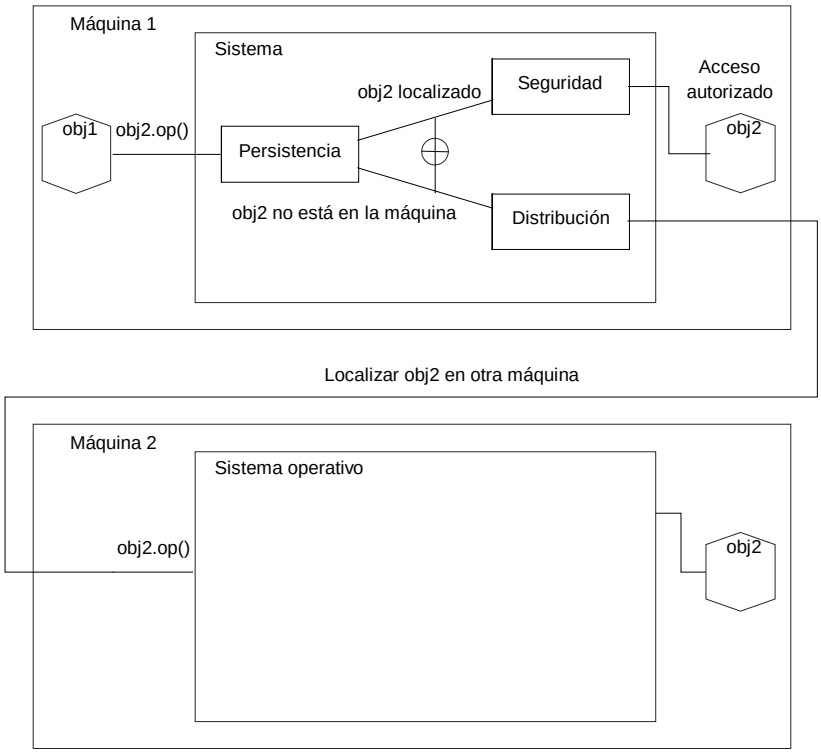


Figura 17.2 Funcionamiento conjunto de las partes del sistema: persistencia, seguridad y distribución.

17.4 Elementos hardware que facilitan la persistencia

Existen una serie de elementos hardware cuyo uso facilita la implementación y mejor aprovechamiento del sistema de persistencia:

17.4.1 Memoria RAM no volátil

La utilización de memoria RAM no volátil como medio de almacenamiento principal en un sistema hace que el uso de la persistencia sea más efectivo desde el punto de vista del usuario y se facilite su implementación.

Uno de los mayores problemas que se presenta al implementar la persistencia es la elasticidad. Es decir, mantener la consistencia cuando ocurre un fallo, el más común es el corte del suministro eléctrico. El problema para lograr la ilusión de elasticidad perfecta es que la sincronía entre los objetos en memoria y en el disco sea la mayor posible. Todos aquellos cambios en los objetos que no se hayan reflejado en el disco se pierden al cortarse la

alimentación. Esto obliga a utilizar costosos algoritmos de consistencia, puntos de verificación, estabilización en disco continua, etc. para que la pérdida sea la menor posible.

Usando memoria RAM no volátil todo lo anterior se simplifica muchísimo. No hay que preocuparse de cortes en el suministro. El sistema tras un corte vuelve exactamente al estado que tenía anteriormente, al mantener exactamente el mismo contenido de memoria. Todo el problema del sincronismo entre los objetos en memoria y su imagen en el disco desaparece, así como la sobrecarga de los algoritmos de consistencia. Simplemente hay que realizar la funcionalidad de área de instancias virtual.

También facilita mucho el uso intuitivo de los ordenadores. Al volver a encender el ordenador, uno se lo encuentra exactamente en el mismo estado que tenía cuando se apagó.

Aunque no de manera generalizada al ser más caro, este tipo de elemento hardware tiene aplicación en sistemas que no puedan permitir pérdida de información, tengan mucho peligro de cortes de suministro o se apaguen y enciendan de continuo y no tengan la potencia de procesador necesaria para la sobrecarga de estos algoritmos. Por ejemplo, sistemas empotrados de control, asistentes portátiles personales, etc.

17.4.2 Soporte para instantáneas de la memoria

Otro tipo de soporte hardware en este sentido, aunque menos ambicioso, serían elementos que permitieran realizar una **instantánea** del estado de la memoria del ordenador[1] y que luego se pudiera recuperar[2]. En lugar de ser directamente la memoria no volátil, se simula almacenando el estado de la memoria en disco duro, por ejemplo. Al arrancar de nuevo, automáticamente se restaura el estado de la memoria a partir de la instantánea, logrando un efecto similar al anterior.

En caso de que no se pueda realizar la instantánea de manera automática frente a una caída de tensión no se tiene toda la funcionalidad anterior. Sin embargo se puede seguir aprovechando para lograr un funcionamiento más intuitivo de los ordenadores. El usuario tiene que activar explícitamente una opción del sistema para apagar el equipo y realizarse una instantánea, o bien utilizar un botón especial del ordenador que tenga este efecto.

Esto facilita la implementación de la persistencia. A la hora de cerrar el sistema habría que realizar una serie de tareas con cierta complejidad para almacenar la información necesaria que permitiera que al arrancar de nuevo el sistema se reconstruyera el estado anterior. Ahora simplemente hay que sacar una instantánea del momento de cierre del sistema. Al arrancar de nuevo se recupera la instantánea para volver al estado anterior.

Los ordenadores portátiles son los candidatos más directos a incluir este soporte. De hecho algunos de ellos ya incluyen funcionalidad similar. Actualmente hay proyectos para incluir de manera estandarizada este tipo de característica en los ordenadores compatibles PC, como la iniciativa On Now. Algunas placas base, incluso para el mercado doméstico, ya incluyen soporte hardware para este tipo de funcionalidad [AOP97]

[1] En realidad aquí hay que incluir también el estado de otros elementos del ordenador necesarios para poder reanudar exactamente el sistema en el estado anterior, por ejemplo el contador de programa del procesador y otros registros.

[2] Aunque menos transparente, podría realizarse algo similar totalmente por software.

17.5 Variantes del esquema del objeto “paginador”. Nivel de detalle de la meta-interfaz de los objetos de la máquina

No necesariamente tiene por qué existir un único objeto “paginador” para realizar la persistencia. Se puede pensar incluso en que cada objeto tenga su propio objeto paginador, o bien por grupos de objetos.

Se podría particularizar así la funcionalidad de cada paginador, adaptándola a las necesidades del objeto que soporta. Por ejemplo algunos objetos no necesitarán elasticidad, pueden elegirse distintos tipos de almacenamiento secundario, utilizar diferentes estrategias de carga y reemplazo de objetos, etc.

Esto permitiría que los usuarios (otros objetos) pudieran diseñar sus propios objetos paginadores. Por ejemplo, un sistema de gestión de bases de datos podría usar un paginador especial para soportar sus objetos internos, como índices, etc., de tal manera que este paginador proporcione un acceso muy rápido a los mismos. Del mismo modo, pueden usarse paginadores que garanticen un nivel mayor de elasticidad para aquellos objetos que se considere tienen una importancia especial.

Todo esto necesita que los objetos de la máquina involucrados dispongan de un meta-interfaz adecuado para realizar este tipo de extensiones con tanto nivel de detalle.

En cualquier caso se podrían reutilizar los objetos usados en la implementación del paginador básico del sistema para crear otros paginadores específicos.

17.6 Algoritmos de implementación interna

Deben estudiarse las diferentes alternativas para la implementación interna del objeto paginador. Básicamente se trata de esquemas que permitan acceder a una tabla que relacione el identificador de cada objeto con su posición física en el almacenamiento secundario.

En una primera fase podemos eliminar la necesidad de tener estabilidad y elasticidad y que funcione simplemente la persistencia. De esta manera se puede disponer más rápidamente de un prototipo con el que experimentar, que es el objetivo principal del sistema. En cualquier caso hay que hacer un diseño en el que se puedan incorporar fácilmente luego.

Otro aspecto que puede involucrar a la implementación de la persistencia es el control de transacciones, aunque esto depende del tipo de transacciones definidas en la parte de concurrencia del sistema operativo.

17.7 Problemas de eficiencia por la granularidad

Aquí se presentan una serie de problemas por el tamaño excesivamente pequeño de algunos objetos (granularidad fina), si todos los objetos se tratan de la misma manera, aunque sean muy pequeños, como por ejemplo enteros.

- **Eficiencia en persistencia**. Puede llegarse a hacer transferencias a disco de unos pocos bytes para esos objetos, ya que el sistema trata todos los objetos por igual. Esto supondría una carga excesiva.
- **Eficiencia en el paso de mensajes**. Esto también se ve reflejado en otros aspectos como la invocación de operaciones (si se hace siempre igual independientemente del objeto). Puede ser muy costoso realizar todo el protocolo de envío de mensajes, con su identificador, comprobación de seguridad, etc. para simplemente sumar dos enteros.

17.7.1 Modificación del modelo de objetos para solucionar problemas de granularidad

Este es un tema que debe estudiarse con mucho detalle, por los problemas anteriores. Puede necesitar modificaciones en el modelo de objetos para solucionar en parte el problema de la granularidad:

- **Objetos anónimos**. Podría tenerse, al estilo de la base de datos O2 [15], objetos normales, con su identificador y objetos anónimos, que no necesitasen esa seguridad, etc. Estos objetos estarían incrustados en otros objetos para su uso exclusivo, con lo cual estos tendrían un mayor tamaño, evitando transferencias de poca información.
- **Grupos de objetos**. Otra idea en ese sentido es pensar en abstracciones por encima de los propios objetos, por ejemplo un grupo de objetos que tengan una cierta relación. De esta manera el SO dispone de más información para realizar optimizaciones en muchas áreas. Podrían formarse grupos como todos los objetos de un usuario, con jerarquías, objetos que colaboren entre sí (paralelismo, puede ayudar en planificación, etc.), simplemente objetos relacionados (se necesitan conjuntamente, deberían almacenarse juntos, etc.)

17.7.2 Uso de relaciones existentes en el modelo de objetos para solucionar problemas de granularidad

Puede tenerse en cuenta para los aspectos anteriores las relaciones existentes en los objetos del modelo de la máquina. Las relaciones de tipo agregación indican que un objeto se compone de otros (similar a la incrustación). Las relaciones de tipo asociación indican que el objeto está relacionado con otros objetos.

Esta información puede ser usada por un objeto paginador para que pueda tomar la decisión de cargar o salvar en el almacenamiento persistente no sólo el propio objeto, si no también los objetos que están relacionados con el mismo. Por ejemplo dado un objeto maceta que tenga agregados un objeto tierra y otro planta, si es necesario trasladar el objeto maceta al almacenamiento persistente, se trasladarían también la tierra y la planta, al estar agregados a la maceta. En cualquier caso el usuario podría indicar de manera explícita los objetos que estuvieran relacionados, como por ejemplo un motor de bases de datos que indica los objetos que tienen una relación entre sí.

Además de para la parte de persistencia, esta información de las relaciones existentes entre los objetos puede ser usada en otras partes del sistema. La parte de distribución puede usarla a la hora de decidir la migración de los objetos, para mover no sólo un objeto, si no todos los que estén relacionados entre sí. También esta información es importante por la planificación, debe procurarse optimizar el paso de mensajes entre los objetos relacionados, ya que estos se comunicarán muy a menudo entre ellos.

17.7.3 Mantenimiento de la uniformidad. Optimizaciones internas en la implementación

En cualquier caso, lo mejor es modificar el modelo de objetos lo menos posible e intentar optimizar todo de manera transparente, manteniendo la mayor uniformidad posible. Por ejemplo, como se acaba de mencionar, los objetos se pueden agrupar implícitamente a partir de la información accesible al sistema en el modelo de objetos. También pueden utilizarse técnicas que esperen a tener encolado un conjunto de objetos antes de almacenarlos en el disco, por ejemplo con el tamaño de una página de disco para resolver este problema. En el caso de los objetos anónimos, el alto nivel de la interfaz de la máquina permite que la implementación trate internamente estos objetos de una manera especial, por ejemplo

mediante la técnica de implementación primitiva de los mismos. Externamente se mantiene la uniformidad del modelo.

Este tipo de problemas son los que hay que tener en cuenta en las diferentes implementaciones de la máquina e intentar su optimización, aunque en general deben estudiarse en conjunto con el modelo de objetos, la máquina abstracta y el resto de los elementos del sistema operativo.

La mejor manera de resolver estos problemas es experimentar con una primera versión del sistema en la que no se tengan preocupaciones de eficiencia, simplemente para validar la filosofía del mismo. Posteriormente se puede comprobar hasta qué punto aparecen estos problemas y en función de esto proponer las modificaciones oportunas en cada nivel, fundamentalmente estrategias de optimización en la implementación.

17.8 Desarrollo de sistemas de gestión de bases de datos orientados a objetos a partir del soporte de persistencia del sistema integral

El soporte de persistencia del Sistema Integral Orientado a Objetos (SIOO) puede servir como base para el desarrollo de sistemas de gestión de bases de datos orientados a objetos (SGBDOO), de una manera más sencilla e integrada [ADA+96]. Disponer de un sistema con persistencia ofrece una serie de ventajas:

17.8.1 Facilidad de desarrollo

La construcción de un SGBDOO se facilita ya que muchas de las funciones que deberían implementarse ya están disponibles dentro de la parte de persistencia del SO. Además, al tratarse de un SIOO, se obtienen las ventajas de la OO: es posible reutilizar sin sobrecarga el código de persistencia ya existente, o bien extenderlo añadiendo únicamente la funcionalidad adicional necesaria para el SGBDOO. Esta funcionalidad adicional puede proporcionarse mediante un motor de bases de datos adecuado que complemente el sistema de persistencia con mecanismos auxiliares para acelerar consultas, mantenimiento de la integridad de las asociaciones, etc.

17.8.2 Mayor rendimiento

Dado que el propio sistema integral ya es orientado a objetos, no existe la necesidad de desarrollar capas superpuestas a un SO tradicional para salvar el espacio existente entre el paradigma del SO y el de la base de datos. El rendimiento es mayor al no tener que atravesar múltiples capas.

17.8.3 Mayor productividad

La programación de aplicaciones de bases de datos es más productiva ya que no es necesario que el programador cambie constantemente de filosofías: una para trabajar con la base de datos y otra para manejar el SO. Siempre se utiliza el mismo paradigma de orientación a objetos en el SO y la base de datos.

17.8.4 Mayor integración en el sistema

La base de datos no tiene por qué considerarse un entorno separado del resto del sistema, como en los SO tradicionales. Dada la cercanía entre el SIOO y el SGBDOO puede considerarse el SGBD como parte integrante del entorno de computación. Es decir los objetos de la base de datos son simplemente unos objetos más dentro de los objetos del sistema

operativo que proporcionan servicios. Es más, puede pensarse en el SGBDOO como el elemento que cumpla el papel de los sistemas de ficheros en los SO tradicionales. El SGBDOO no sería utilizado como un sistema independiente del SO, si no que el usuario podría utilizarlo como sistema de gestión de los objetos del sistema operativo, haciendo consultas sobre los mismos, etc.

Capítulo 18

IMPLEMENTACIÓN DE UN PROTOTIPO DE SISTEMA DE PERSISTENCIA PARA EL SISTEMA INTEGRAL

En este capítulo se describe las líneas fundamentales de la implementación de un prototipo de sistema de persistencia para el sistema integral [OAI+97], desarrollado a partir del prototipo de la máquina abstracta Carbayonia descrito en el capítulo 13. Este prototipo también se ha desarrollado como proyecto fin de carrera de la Escuela Superior de Ingenieros Informáticos de la Universidad de Oviedo. Se puede consultar los detalles de la implementación en [Ort97].

El prototipo se ha desarrollado utilizando el lenguaje C++ y funciona en Windows NT y Windows95.

El objetivo fundamental del prototipo es desarrollar en un tiempo reducido un producto que permita comprobar en la práctica el comportamiento de un sistema de persistencia, desde el punto de vista de la utilización por el usuario. Así pues, la eficiencia es una preocupación secundaria y se ha procurado introducir el menor número de modificaciones en la implementación existente de la máquina abstracta. Por la misma razón se han realizado algunas simplificaciones sobre los objetivos ideales del sistema de persistencia descrito en el capítulo 16. Una comparativa de rendimiento con la versión del prototipo de la máquina sin persistencia se encuentra en el apéndice E

18.1 Elementos de partida

Como elementos de partida sobre los objetivos ideales del sistema de persistencia se toman los que figuran a continuación.

18.1.1 Sin soporte para persistencia de la computación

La persistencia de la computación depende del modelo de concurrencia. El modelo de concurrencia del prototipo de la máquina es transitorio hasta que sea totalmente diseñado por el sistema operativo. El soporte transitorio se basa en hilos separados de los objetos.

Esto complicaría mucho la implementación de la persistencia, si hubiera que hacer persistir también a los hilos, al estar separados de los objetos. Teniendo en cuenta que es posible que el modelo cambie, se deja este punto para una versión posterior.

En cualquier caso, si se define un modelo de concurrencia en el que la computación (hilos) esté encapsulada en el estado de un objeto, el sistema actual lograría prácticamente sin modificación la persistencia de la computación, ya que hace persistir el estado de una instancia.

18.1.2 Implementación primitiva

Para acelerar el desarrollo de la implementación, se realizará de manera totalmente primitiva (integrada en el código C++ del simulador de la máquina), en lugar de en el espacio

del usuario. Aunque esto limita la extensibilidad, el objetivo es comprobar la funcionalidad del sistema y ésta será la misma aunque se implemente de manera primitiva.

18.1.3 Persistencia ortogonal, no completa

El motivo de que todos los objetos sean siempre persistentes (persistencia completa) es por la sencillez conceptual que toma el entorno. Sin embargo, para lograr la uniformidad completa se requiere que se encapsule la computación y que ésta se haga persistente también. Dado que los aspectos de computación (concurrencia) están todavía en fase de desarrollo en el sistema operativo, y existe simplemente un soporte temporal para hilos, se usará una persistencia ortogonal. Cualquier objeto que se desee podrá ser persistente, simplemente indicándolo al sistema. Esto requiere algunos cambios, como un pequeño cambio en la interfaz del sistema y la necesidad de un servicio de directorio. Existirán, pues, objetos temporales y objetos persistentes, aunque no se diferenciarán en su utilización.

18.1.4 Interfaz con el usuario

La interfaz con el usuario reflejará el hecho anterior, mediante unas pequeñas modificaciones.

18.1.4.1 Declaración de objetos persistentes

Los objetos que sean susceptibles de ser hechos persistentes tendrán que pertenecer a una clase persistente. Esto se realiza anteponiendo la palabra clave Persistent a la declaración de la clase[1]. Para los agregados de una clase también se puede elegir cuáles de ellos serán persistentes. Para ello también se antepone Persistent delante de la declaración de estas referencias agregadas.

18.1.4.2 Servicio de directorio reflectivo

Para hacer persistir los objetos se almacenarán en un fichero de intercambio del sistema operativo anfitrión.

Todo objeto de una clase Persistent es susceptible de hacerse persistente si el usuario lo desea. Es decir, es necesario utilizar un mecanismo que permita hacer que un objeto (a través de una referencia) se convierta en persistente. De la misma manera, será necesario poder conectar una referencia a un objeto persistente.

Para ello se desarrolla un sencillo servicio de directorio. Este servicio permite dar un nombre simbólico a los objetos en el sistema de persistencia. Tendrá operaciones que permitan añadir un objeto al sistema de persistencia (convertirlo en persistente) y darle un nombre simbólico, conectar una referencia a un objeto persistente (a través de su nombre simbólico), y eliminar un objeto del sistema de persistencia y comprobar si existe un objeto en el sistema con un nombre dado.

18.1.4.3 Persistence

Para mantener la uniformidad, esta funcionalidad de directorio se proporcionará mediante una clase primitiva Persistence que tenga esas funciones, y se añadirá una referencia del sistema persistence que apunte a una instancia de esa clase[2]. El usuario accederá al servicio de directorio de persistencia a través de esta referencia, invocando sus métodos.

---

[1] Esto es similar a la interfaz de la base de datos orientada a objetos POET [POE97].
[2] Lo cual es una forma de implantar la reflectividad (véase el capítulo 14)

```
Class Persistence
Isa Object
Methods
  exists(String):Bool;
  add (String,Object);
  getObject(String):Object;
  remove(String);
EndClass
```

- **Exists(String):Bool.** Devuelve el valor booleano referente a la existencia del objeto persistente en el sistema de persistencia, cuyo nombre simbólico en el sistema de persistencia es la cadena pasada.
- **Add(String, Object).** Dándole un nombre asociado y el objeto, almacena el objeto en el sistema de persistencia.

  Se produce una excepción si el objeto con ese nombre ya existe.
- **GetObject(String):Object.** Devuelve el objeto asociado en el sistema de persistencia al nombre simbólico pasado. Se produce una excepción si el objeto con ese nombre no existe en el sistema de persistencia.
- **Remove(String)**. Elimina el objeto del sistema de persistencia, cuyo nombre sea la cadena de caracteres pasada como parámetro. Se produce una excepción si el objeto con ese nombre no existe en el sistema de persistencia.

18.1.4.4 Utilización del sistema

La implicación del usuario en el sistema de persistencia se reduce a indicar qué objetos serán temporales y cuáles podrán ser persistentes. En el momento que desee que un objeto sea persistente, simplemente lo añadirá al sistema de persistencia a través del servicio de directorio, dándole un nombre simbólico. A partir de ese momento, el objeto ya no desaparecerá del sistema hasta que no sea borrado. En cualquier momento posterior se podrá conectar una referencia a ese objeto persistente para poderlo utilizar.

En cualquier caso, existe una total uniformidad de uso de los objetos. No hay diferencia entre la utilización de un objeto persistente y uno temporal.

El sistema de persistencia crea un área de instancias virtual en la que pueden residir todas las instancias que se necesiten. En el caso de los objetos persistentes el sistema se ocupa de almacenarlos en disco y traerlos al área de instancias cuando sea necesario, sin intervención del usuario.

En el futuro, cuando se implante la persistencia completa con permanencia de la computación, ni siquiera será necesario indicar que un objeto se haga persistente, ya que lo serán todos. El servicio de directorio para dar nombre a los objetos persistentes tampoco será necesario. Sí que se necesitará un servicio de denominación de alto nivel para asociar nombres simbólicos a los objetos similar a este servicio de directorio, pero al ser todos los objetos persistentes sólo tendrá la función de denominación, no la función implícita de señalar los objetos persistentes.

18.1.4.5 Ejemplo de programación con los nuevos elementos de persistencia

La integración de la persistencia en un sistema orientado a objetos forma un primer paso para la funcionalidad de una base de datos orientada a objetos. En el apéndice C se encuentra

un ejemplo de programación que utiliza estos nuevos elementos de persistencia para desarrollar una pequeña aplicación de bases de datos.

18.2 Filosofía de implementación

La filosofía de la implementación se basa en modificar lo menos posible el simulador ya existente. Para ello se proporcionan mecanismos para facilitar la persistencia en las clases del simulador, pero separando el espacio de trabajo normal de la máquina (que no se modificará apenas) con el espacio de memoria virtual para los objetos de estas clases persistentes.

18.2.1 Persistencia de los objetos de la implementación

El alto nivel de la interfaz de la máquina permite utilizar una implementación de la persistencia con una aproximación distinta a lo normal, más rápida de implementar, pero funcionalmente equivalente. En lugar de trabajar la persistencia directamente en el espacio de los objetos del usuario, se les da esta propiedad indirectamente trabajando la persistencia en el espacio de los objetos de la implementación.

Dado que el simulador existente representa mediante objetos de la implementación los objetos de los programas Carbayonia, para hacer persistir las clases y las instancias Carbayonia basta con hacer persistir los objetos que las representan en el simulador (objetos C++).

El estado de un elemento persistente de Carbayonia queda totalmente definido por los objetos C++ que lo representan del simulador. Deben persistir, pues, las clases, métodos, referencias e instancias del simulador. Haciendo ese hecho transparente, sin modificar el mecanismo de funcionamiento anterior de la máquina se consigue un desarrollo rápido del prototipo.

18.2.2 Separación área trabajo normal / memoria virtual para objetos persistentes

Debe proporcionarse la funcionalidad de memoria virtual al **área de trabajo**[1] de la máquina con respecto a las instancias que son persistentes. Es decir, el espacio de almacenamiento persistente es potencialmente infinito y su uso es transparente y conjunto con el área en la que existen los objetos del simulador.

Con el objeto de modificar lo menos posible el simulador, la funcionalidad de los objetos de la implementación del simulador no cambia. Es decir, desde el punto de vista de la máquina, todos los objetos son objetos temporales que están en el área de trabajo normal.

Los objetos persistentes estarán en una **memoria virtual persistente** respaldada en el archivo de intercambio. Estos objetos persistentes estarán representados en su forma de objetos del simulador.

Para hacer funcionar de manera transparente los objetos persistentes, se creará un **objeto temporal** en el área de trabajo asociado al **objeto persistente**. Es decir, cuando un objeto persistente tenga que funcionar de manera activa (**objeto activo**), lo hará mediante un objeto normal del simulador que lo represente. A la forma activa en el área de trabajo en que se encuentre un objeto persistente se le denominará **representación normal** y a la forma en la memoria virtual persistente, **representación persistente**.

[1] Dado que persisten todos los objetos del simulador, el concepto de área de instancias se refiere aquí al espacio de trabajo normal de los objetos C++ del simulador o **espacio de la implementación**.

Evidentemente tiene que existir un **protocolo** que mantenga sincronizada la representación normal y la persistente, y que permita el paso del área persistente al área normal. En realidad, la representación persistente consiste en la información necesaria que permita reconstruir el objeto normal en el área normal de funcionamiento, no tiene por qué ser una instantánea exacta del dicho objeto.

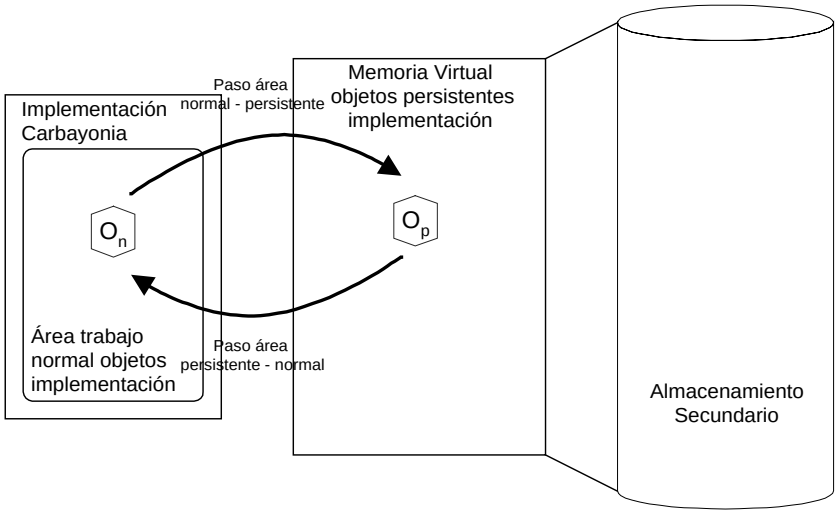


Figura 18.1 Memoria virtual para objetos persistentes de la implementación.

Es decir, el sistema se divide en la máquina abstracta anterior, a la que se le añade un mecanismo de memoria virtual para los objetos persistentes[1]. Sobre este mecanismo se implementa el servicio de directorio de instancias de usuario. La máquina sólo necesita pequeñas modificaciones para pedir a la memoria virtual que le proporcione una instancia de usuario persistente que no está en forma activa y viceversa.

18.2.3 Área virtual ilimitada para clases e instancias

Básicamente, los objetos del simulador representan los elementos de la arquitectura de Carbayonia: Clases e Instancias. El hecho de hacer persistentes en el espacio del simulador los objetos del simulador con un espacio de almacenamiento de la información persistente en principio ilimitado, se reflejará en el espacio de usuario.

El resultado desde el punto de vista del usuario es que dispone de un área de clases de usuario persistentes y un área de instancias persistentes virtualmente infinitas. En ellas podrá usar tantas clases persistentes e instancias persistentes como desee. La implementación se ocupa de hacerlas funcionar transparentemente de modo normal con los recursos existentes.

Hay que tener en cuenta que esta área persistente sólo se limita a los elementos persistentes. Para los objetos temporales existe la limitación de espacio impuesta por el tamaño real del espacio de trabajo del simulador.

18.3 Implementación de la memoria virtual

La implementación de la memoria virtual se realiza enlazándola con el simulador existente, evitando modificaciones en el mismo. Para acelerar el rendimiento se usará una memoria intermedia y un mecanismo de paginación más segmentación.

[1] En realidad en el gráfico anterior tanto la forma normal O_n como la persistente O_p se refieren a los objetos de implementación interna del simulador.

18.3.1 Enlace entre el funcionamiento del simulador y la memoria virtual del sistema de persistencia

Para enlazar de manera transparente el funcionamiento anterior del simulador con la memoria virtual se hace que las clases C++ del simulador que representan los objetos persistentes desciendan de un elemento común que puede ser representado persistentemente (TVMItemTemplate). Las clases cuyos objetos se harán persistentes son las que representan los elementos de un objeto Carbayonia[1]:

- **Clases**. TCObject, TCInteger, TCFloat, etc.
- **Métodos de las clases anteriores**: TMObjectGetClass, TMObjectGetID, ..., TMIntegerAdd, TMIntegerSub, etc.
- **Instancias**: TIObject, TIInteger, TIFloat, etc.
- **Referencias**: TRef.
- **Instrucciones**: TInstrCall, TInstHandler, TInstNew, etc.

Este elemento común encapsula la funcionalidad necesaria para hacerse persistente en la memoria virtual (TVirtualMemory) implementando las funciones necesarias para el protocolo de emparejamiento del objeto en su forma activa y persistente.

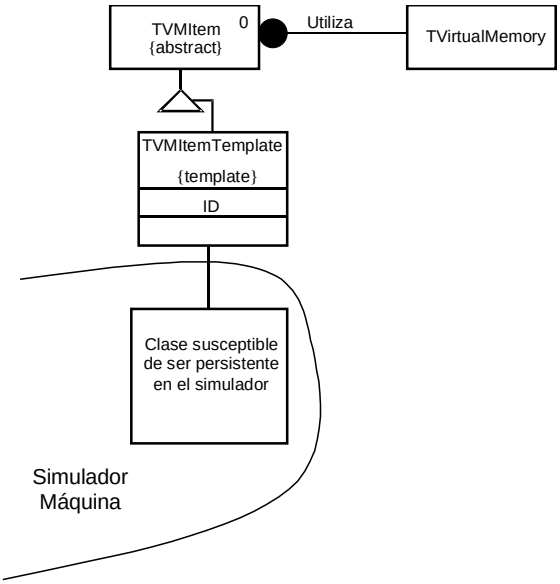


Figura 18.2 Enlace entre el simulador y la memoria virtual a través de la clase abstracta TVMItem.

18.3.1.1 Transparencia de acceso a los objetos persistentes. Punteros inteligentes

El simulador utiliza el operador C++ -> de referenciación de punteros para comunicar sus objetos entre sí, pues utiliza siempre punteros a objetos. Al introducir la persistencia con memoria virtual puede ser que alguno de los objetos a los que se acceda no estén activos y residan en memoria persistente. Para hacer transparente este hecho en el simulador se hace que el operador -> sea inteligente y cuando se acceda a un objeto a través del operador, recupere el objeto de la memoria virtual si no está ya activo.

[1] Puesto que no se contempla persistencia de la computación en este prototipo. Para ello también habría que hacer persistir los elementos del simulador que dan soporte a la computación: los hilos con las pilas asociadas con registros de activación y de excepciones.

En la clase TVMItemTemplate que representa a los objetos persistentes se redefine el operador C++ ->, de tal manera que si el objeto no está ya activo en memoria, se accede a la memoria virtual para reconstruirlo y cargarlo. Como todas las clases del simulador derivan de esta clase, el resultado es que el simulador no es consciente de que cuando accede a un objeto puede que se acceda a la memoria virtual para cargarlo.

De esta manera se logra que la existencia de la persistencia sea transparente para toda la parte de funcionamiento del simulador.

18.3.1.2 Identificador para la memoria virtual

El atributo ID se utilizará para identificar el objeto persistente del simulador en la memoria virtual. Se usará cuando sea necesario localizar el objeto en la memoria virtual para pasarlo a estado activo. Este identificador no tiene ninguna relación con el identificador único de los objetos de usuario. Es un identificador que tienen los objetos persistentes de la implementación para ser localizados en el almacenamiento persistente y sólo se usa en el espacio de la implementación.

18.3.2 Memoria intermedia de la memoria virtual

Para mejorar el rendimiento, en lugar de que la memoria virtual se represente directamente sobre el disco, se utilizará una **memoria intermedia** (caché) que almacene en memoria principal los últimos objetos persistentes utilizados. La funcionalidad de esta memoria intermedia es similar a la del *buffer cache* [Bac86] de entrada/salida del sistema Unix, o la propia memoria física de un ordenador con paginación.

El emparejamiento entre la representación normal y la persistente ahora se realiza sobre la memoria intermedia. Dado que la memoria intermedia es limitada, y en algún momento se agotará, es necesario gestionar el espacio. Para ello se necesita también un mecanismo de **reemplazamiento** (paso de objetos al almacenamiento secundario[1]) y **emplazamiento** (paso del almacenamiento secundario a la memoria intermedia).

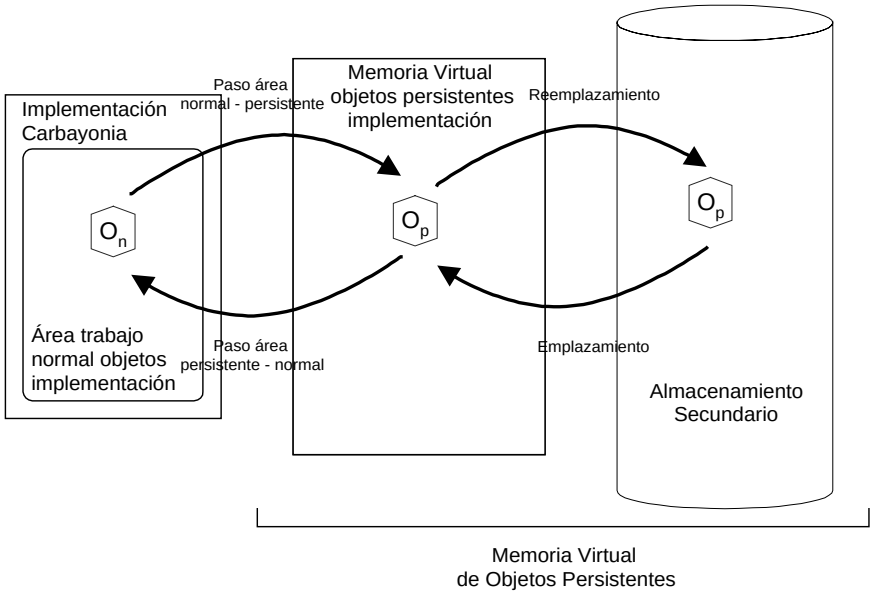


Figura 18.3 Utilización de una memoria intermedia para mejorar el rendimiento de la memoria virtual.

[1] Mediante un fichero de intercambio.

Esta memoria intermedia mejora el rendimiento, ya que las actualizaciones repetidas de los objetos no tienen que hacerse directamente a disco, si no que se van haciendo sobre la memoria intermedia, más rápida. Cualquier objeto persistente que esté activo, estará colocado en esta memoria intermedia y tendrá su pareja correspondiente en el simulador (objeto normal).

La gestión de esta memoria intermedia queda englobada dentro de la clase TVirtualMemory.

18.3.3 Paginación más segmentación

El sistema localiza los objetos persistentes del simulador en almacenamiento secundario a través de su ID persistente. En lugar de transferir estos objetos individualmente se pueden agrupar los objetos en bloques (páginas) para realizar las transferencias. Esto aumenta el rendimiento.

La memoria intermedia[1] se dividirá entonces en bloques de igual tamaño, al igual que el archivo de intercambio. Dentro de cada página se colocarán los objetos persistentes, o más propiamente, la representación que permitirá reconstruir el objeto cuando tenga que estar activo. Al espacio que ocupa la representación de un objeto persistente se le denomina **segmento**.

El segmento tendrá una representación determinada que dé información acerca de la información que contiene. Es decir, cada segmento tendrá una cabecera con su tamaño, el tipo del objeto que lo ocupa, etc. Otro elemento que se puede almacenar ahí cuando el segmento está en la memoria intermedia es la dirección del objeto normal asociado.

Tenemos pues, un sistema que divide la memoria virtual en bloques de igual tamaño, dentro de los cuales se almacenan los segmentos que representan los objetos persistentes. Esto implica que cuando se necesite un objeto en realidad se traerá ese objeto y todos los demás que compartan su misma página. De la misma manera cuando se grabe ese objeto en almacenamiento secundario se grabará con todos los que formen su página.

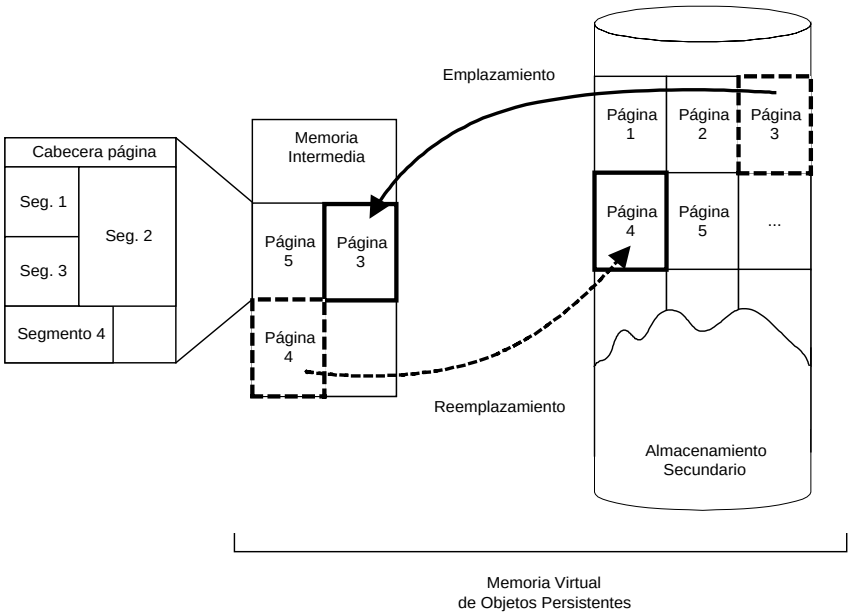


Figura 18.4 Mecanismo de paginación más segmentación en la memoria virtual.

[1] En la implementación actual el tamaño de la memoria intermedia no es fijo. Se van añadiendo y eliminando páginas de la memoria intermedia de manera dinámica.

18.3.3.1 Información de la localización en el identificador persistente

Para acelerar aún más el proceso de localización de un objeto persistente en la memoria virtual, se hace que el identificador en el sistema de persistencia del objeto esté asociado siempre con la misma posición física dentro del sistema de memoria virtual. Es decir, se hace que el identificador lleve la información de la página en la que se encuentra el objeto y el desplazamiento que ocupa su segmento dentro de la página[1].

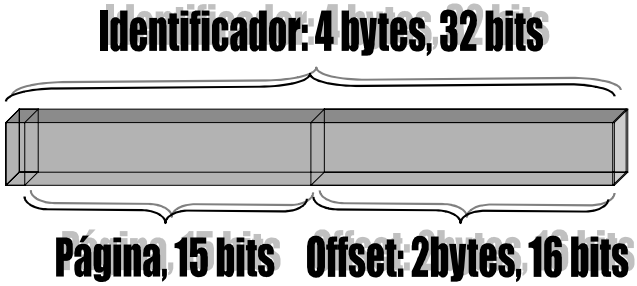


Figura 18.5 Formato del identificador de objetos persistentes.

El primer bit de un identificador se utiliza para etiquetarlo como identificador de un objeto persistente. El resto se pueden utilizar para representar la página y el desplazamiento dentro de la página. El número de bits que se dediquen a cada apartado determina el tamaño de página y el número total de páginas posibles en la memoria virtual. En la figura se representa un ejemplo con 2^{15} páginas y 2^{16} (65.536) bytes por página.

El uso de identificadores de 4 bytes y este sistema de paginación más segmentación tiene una serie de limitaciones:

- El número máximo de bytes que pueden ocuparse para representar objetos persistentes está dado por el número de bits del identificador.
- El tamaño máximo de un segmento (objeto) está limitado al tamaño de página escogido.

18.3.3.2 Tamaño de la página

Para permitir experimentar con cualquier tamaño de página para determinar un tamaño adecuado el atributo BytesBlock de la clase TVirtualMemory permite definir el valor de este tamaño.

18.3.3.3 Acceso a un objeto en el sistema de persistencia

El acceso a un objeto persistente se acelera y sigue el mecanismo tradicional de memoria virtual por **paginación más segmentación** [Dei90]. Cuando se necesita un objeto persistente se proporciona su identificador. Con la parte que indica el número de página se obtiene la página correspondiente. Dentro de esa página, en el desplazamiento que indica la segunda parte de la dirección se encuentra el segmento del objeto. La cabecera del segmento junto con el contenido del mismo tienen la información necesaria para reconstruir la pareja normal del objeto persistente.

Para localizar la página rápidamente en el archivo de intercambio, estas se dispondrán linealmente desde el principio del archivo. De esta manera se puede acceder a una página concreta mediante un acceso directo a su posición dentro del archivo.

[1] Como en un sistema de memoria virtual por paginación más segmentación.

18.4 Otros aspectos de la implementación

Existen otros aspectos de implementación adicionales, como las políticas de emplazamiento y reemplazamiento, estadísticas y la representación de las páginas en la memoria intermedia.

18.4.1 Políticas de emplazamiento y reemplazamiento

Como cualquier memoria intermedia, se pueden utilizar **diferentes políticas de emplazamiento y reemplazamiento** clásicas de la paginación [Dei90]. Para facilitar la experimentación con diferentes políticas, se usa una clase abstracta TPolicy de la que derivan las clases concretas que implementan las políticas: primero en entrar primero en salir (TFIFOPolcy), último en entrar primero en salir (TLIFOPolcy), menos recientemente usado (TLRUPolcy) y aleatoria (TRandomPolcy).

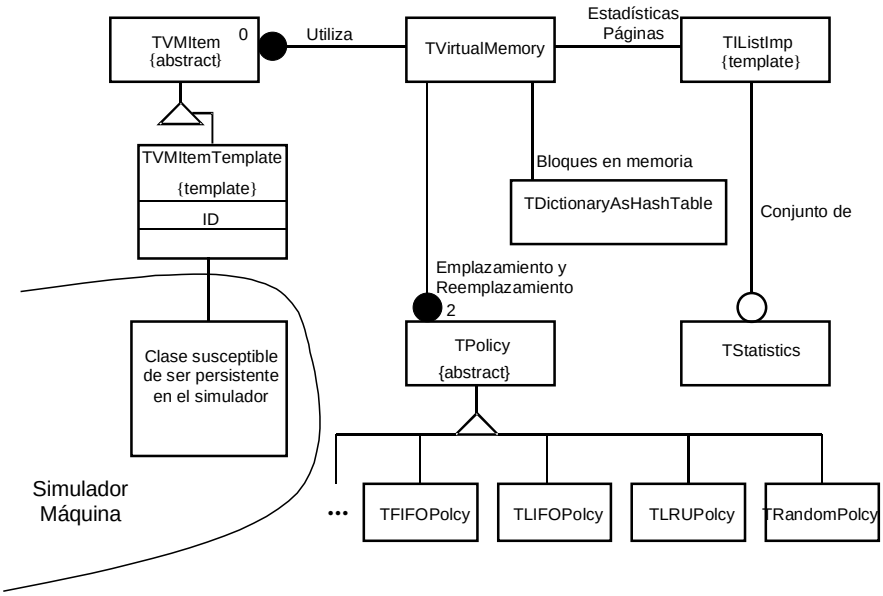


Figura 18.6 Diagrama de clases del sistema de persistencia con políticas de emplazamiento, reemplazamiento y estadísticas.

18.4.2 Estadísticas

Para acelerar en lo posible los accesos a las páginas y facilitar la implementación de las distintas políticas se lleva una estadística completa acerca del comportamiento de todas las páginas de la memoria virtual. Para esto la memoria virtual está asociada a una lista de objetos instanciados de TStatistics. Cada uno de ellos guarda estadísticas de utilización de todas las páginas en disco y memoria.

18.4.3 Representación de las páginas en la memoria intermedia

Para utilizar de forma eficiente la memoria principal y realizar el menor número de transferencias entre las distintas capas de la jerarquía de memoria se ha creado una tabla *hash* de páginas en memoria implementada por la clase TDictionaryAsHashTable.

18.5 Protocolo de emparejamiento objeto en forma normal / persistente

Para permitir la sincronización entre objeto normal y su forma persistente y la creación inicial de la forma persistente, cada una de estas clases tiene que implementar las funciones

que definen el **protocolo de emparejamiento**[1]. Serán usadas por el sistema de persistencia (TVirtualMemory).

Estas están definidas de manera abstracta en TVMItem, y hacen que una clase proporcione la información que representará sus objetos en el sistema de persistencia (Write), y reconstruir el objeto normal a partir de la representación persistente anterior (Read). Además, implementarán un método que indique el tipo de clase que es (Type) y la longitud que ocupa su representación persistente (Size).

18.5.1 Utilización del sistema por el usuario

TVirtualMemory implementa la funcionalidad de memoria virtual para los objetos persistentes. Este será el enlace del simulador con el sistema de persistencia. El funcionamiento de usuario es siempre con objetos normales del área de instancias. Cada objeto del área de instancias del usuario estará representado por un conjunto de objetos del simulador. Si el objeto del usuario es persistente, también lo serán el conjunto de objetos del simulador correspondiente. El sistema de persistencia se ocupa de realizar de manera transparente las actualizaciones necesarias para que estos objetos no desaparezcan. Para ello usa el protocolo entre el objeto temporal del simulador y su representación persistente y graba en última instancia los cambios en el archivo de intercambio. La parte de simulación de la máquina no diferencia entre objetos C++ normales y persistentes, por tanto, desde el punto de vista del usuario, éste tampoco nota diferencia entre un objeto de usuario temporal y uno persistente.

El objeto del simulador temporal es la instancia de TVMItemTemplate de la clase correspondiente y el objeto persistente es el segmento de memoria virtual que contiene la información necesaria para reconstruir el estado del objeto temporal asociado.

Cuando un objeto está activo, tiene objeto temporal asociado, por lo que su página debe residir en la memoria intermedia.

18.5.2 Adición de un objeto al sistema de persistencia

Cuando el usuario haga un objeto de usuario persistente (funcionalidad del servicio de directorio), este se lo comunicará a la memoria virtual, que creará las representaciones persistentes de los objetos usando sus funciones de protocolo. Además, se ajustará el identificador persistente de manera adecuada para localizarlo en el sistema de persistencia.

18.5.3 Reemplazamiento

Cuando por necesidades de memoria o la política de reemplazamiento de la memoria virtual lo crea conveniente se debe sacar una página de memoria virtual, se grabará la página en disco (con los objetos que contiene). Dado que la página ya no está en la memoria intermedia, se tendrán que eliminar del área temporal (área normal) los objetos temporales asociados a los objetos persistentes de la página. Estos objetos pasan a estar inactivos.

18.5.4 Emplazamiento

De la misma forma, cuando se solicite un objeto persistente al sistema de persistencia, por no estar su página en memoria intermedia se utiliza el procedimiento descrito anteriormente para buscar la página a partir del identificador. Se cargará la página en la memoria intermedia y se crearán los objetos temporales asociados a los objetos de la página a partir de la información de su representación persistente almacenada en el segmento.

[1] Siguiendo la terminología CORBA [OMG95, OMG97]

18.6 Bloqueo de páginas

Este funcionamiento del emplazamiento y el reemplazamiento puede provocar una **condición de carrera**. Esta puede suceder cuando la llamada a un método hace que la memoria virtual destruya al propio objeto que llama, y no existirá objeto al que retornar de la llamada.

El escenario es el siguiente: un objeto llama a un método de otro objeto persistente. Si este no está activo, habrá que cargar su página. Pero al cargarla puede que haya que expulsar alguna página de la memoria intermedia, con lo que habrá que eliminar los objetos temporales asociados a ella. Si la página expulsada es precisamente la página en la que está el objeto que llamó inicialmente al método, resulta que se habrá destruido este objeto. De esta manera cuando acabe la llamada, no existirá ningún objeto al que retornar de la misma.

La solución es bloquear páginas de manera adecuada para que no sean expulsadas de la memoria intermedia. Cuando el objeto temporal asociado a un objeto persistente ejecuta uno de sus métodos se bloquea la página a la que pertenece su objeto persistente porque si ésta es liberada, el objeto temporal asociado también será destruido con el consiguiente error en ejecución.

Los mecanismos de bloqueos por página son similares al funcionamiento de un semáforo. Cada vez que se realiza un bloqueo se incrementa en una unidad el contador de bloqueos y cada vez que se desbloquea se decrementa. Una página podrá ser liberada cuando esté en memoria y, además, su contador de bloqueos sea exactamente igual a cero.

La clase TVirtualMemory posee dos métodos (Wait *y* Signal) que, identificando un objeto persistente, bloquean su página asociada. El contador de bloqueos es un campo de los objetos TStatistics de cada página puesto que es otro criterio para llevar a cabo los reemplazamientos.

También es necesario, por motivos de eficiencia, bloquear en un momento de ejecución todas las páginas de memoria para que no haya actualizaciones a disco. Esto se lleva a cabo con la llamada a los métodos ya identificados, sin pasar el identificador de ningún objeto.

18.7 Formato del archivo de intercambio y de los segmentos

La memoria virtual se respalda en un **archivo de intercambio**. Este fichero es por simplicidad un fichero del sistema operativo subyacente, donde se grabará la información de los objetos persistentes (segmentos). Para poder utilizar esos objetos persistentes en otros simuladores, es necesario que el archivo de intercambio tenga un formato con una representación uniforme de objetos persistentes que permita que sea interpretado por otra máquina diferente de la que lo creó.

Para que el archivo de intercambio sea compatible debe tener un formato común definido. El actual diseño se basa en la idea de almacenar inicialmente 4 bytes sin signo que identifiquen el tamaño de la página del archivo de intercambio. Una vez establecido éste, se interpreta la información a continuación como bloques de este tamaño fijo, es decir, como la secuencia de páginas que forman la memoria virtual.

Dentro de cada página habrá una serie de segmentos ordenados. La información del segmento se divide en dos partes, como se ha mencionado anteriormente. Primero se almacenará una información de cabecera y a continuación los datos que pueden tener una longitud variable. La especificación completa del formato del segmento se encuentra en el apéndice J.

18.8 Cambios en el sistema

La introducción de la persistencia apenas produce cambios en la utilización del sistema. A continuación se resumen estos.

18.8.1 Máquina Carbayonia y Lenguaje Carbayón

Únicamente se añade la palabra clave Persistent al lenguaje Carbayón para indicar los elementos que son susceptibles de ser persistentes. El resto del lenguaje, juego de instrucciones, etc. no cambia.

En la máquina aparece una clase básica adicional Persistence que implementa el servicio de directorio de persistencia. También aparece una nueva referencia de sistema persistence, que apunta a una instancia del servicio de directorio. Sin embargo, el usuario lo utiliza de manera normal como un objeto más de una clase de usuario.

18.8.2 Fichero de clases

El fichero de clases también sufrirá un pequeño cambio para indicar también los elementos persistentes.

18.8.3 Entorno de desarrollo

El traductor del entorno de desarrollo tiene que reflejar estos pequeños cambios en el lenguaje Carbayón, la nueva sintaxis y la adición al formato del fichero de clases.

Los cambios son tan pequeños que el uso de la máquina y el entorno de desarrollo es casi igual al que había antes. No se describirá la gramática completa del lenguaje ni el uso del entorno de desarrollo para no reiterar en exceso. Pueden consultarse en [Ort97].

18.9 Resumen

Para desarrollar de manera rápida el prototipo de la persistencia sobre la máquina del sistema integral se hace que el usuario declare cuáles son las clases cuyos objetos son susceptibles de ser persistentes. Para que un objeto pase a ser verdaderamente persistente se le da un nombre simbólico y se añade al sistema de persistencia a través de un servicio de directorio proporcionado como una clase reflectiva. Por lo demás, el uso del sistema no se distingue de una máquina sin persistencia.

Para desarrollar un prototipo rápido, la implementación del prototipo de la persistencia aprovecha el alto nivel de la interfaz de la máquina para proporcionar la persistencia a los objetos de usuario mediante la persistencia de los objetos en el espacio de la implementación. Se hacen persistentes los objetos C++ del simulador que representan los elementos de usuario de un programa Carbayonia: Clases, métodos, instancias y referencias, y un mecanismo de memoria virtual para los objetos persistentes. La conexión entre el sistema de persistencia y el simulador es a través de una clase abstracta que representa un elemento persistente de la que derivan las clases del simulador. Esta clase redefine el operador -> de acceso a los objetos para buscar en la memoria virtual de manera transparente al simulador un objeto persistente no cargado. De esta manera se consigue hacer persistentes de manera transparente estos elementos sin afectar al funcionamiento del simulador.

La memoria virtual se respalda con un archivo de intercambio y utiliza un esquema de paginación + segmentación, dividiéndose en páginas de tamaño fijo dentro de las cuales se colocan segmentos de longitud variable que representan el estado persistente de los objetos del simulador. Para acelerar el acceso a las páginas de memoria virtual se coloca una memoria

intermedia y se hace que el identificador en memoria virtual de un objeto persistente indique de manera directa la página y desplazamiento donde se encuentra su segmento. La memoria intermedia puede utilizar una gama de políticas de reemplazamiento y emplazamiento típicas de los sistemas de memoria virtual, así como un bloqueo de páginas para evitar condiciones de carrera.

Para no perturbar el funcionamiento del simulador todos los objetos cuando están activos deben existir como objetos normales del simulador, sean persistentes o no. Cuando el objeto es persistente la página donde reside su estado persistente debe estar cargada en la memoria intermedia. Se establece un protocolo que permite actualizar de manera adecuada la pareja objeto normal / representación persistente asociada cuando el objeto está activo.

La utilización extensiva de la herencia con clases abstractas y polimorfismo y el esquema diseñado permiten que las modificaciones introducidas en la interfaz de la máquina sean mínimas. Así mismo, los cambios en la implementación de la máquina serán adiciones modulares, sin afectar en exceso a los elementos y al funcionamiento preexistente.

Capítulo 19

FLEXIBILIDAD EN EL SISTEMA INTEGRAL ORIENTADO A OBJETOS

En este capítulo se muestra la flexibilidad que proporciona la arquitectura del sistema integral orientado a objetos propuesto. Se utiliza el trabajo de Cahill [Cah96] acerca de la flexibilidad en sistemas operativos, técnicas para obtenerla y su clasificación en tipos como marco de referencia en el que mostrar la versatilidad de las capacidades de flexibilidad de la arquitectura del sistema integral, incluso solucionado el problema del control uniforme de la extensibilidad que sufren otros sistemas.

19.1 Flexibilidad

La **flexibilidad** se puede definir como "*la capacidad de diseñar sistemas que puedan ser ajustados a las necesidades de aplicaciones o dominios de aplicación específicos*" [Cah96]. Es una propiedad cada vez más necesaria para los sistemas operativos[1]. Los sistemas monolíticos tradicionales cada vez son menos capaces para dar acomodo a las cada vez más dispares necesidades de las aplicaciones. Nuevos campos de aplicación, los sistemas distribuidos, nuevos tipos de hardware, utilización de ordenadores en nuevas áreas, etc. imponen unas necesidades que pueden llegar a ser tan diferentes que sólo un sistema flexible puede responder a ellas.

El ejemplo típico de un problema que soluciona un software de sistema flexible es la **falta de una característica** [Dra93]. Es decir, el sistema no proporciona una característica determinada que es fundamental para una aplicación específica. Un intento de solucionar este problema en un sistema monolítico produce el efecto contrario, la abundancia de características. Para intentar dar soporte al mayor número posible de aplicaciones se incluyen muchas características en el sistema. Sin embargo, lo más probable es que la mayoría de las aplicaciones sólo utilicen un número reducido de las mismas, desaprovechándose el resto[2]. En cualquier caso, se usen o no, ya forman parte del sistema y suponen una penalización del rendimiento.

Un **sistema flexible** permitiría ser ajustado para proporcionar sólo las características que se necesiten para cada caso. Incluso podría mejorarse el rendimiento. Los servicios de un sistema llevan implícita normalmente una política determinada que favorece a la mayoría de las aplicaciones (**dilema del compromiso** [KLM+93]). Pero para algunas el servicio "estándar" no es una buena elección. En un sistema flexible este tipo de aplicaciones tendría mecanismos para adaptar el servicio mejor a sus necesidades[3].

[1] En general en el software de sistemas.
[2] También se denomina a este tipo de sistemas software "obeso" (*fatware*), pues al incluir tantas características las aplicaciones "engordan" con partes superfluas que no llegan a tener uso.
[3] Por ejemplo escribiendo una versión del servicio a medida de sus necesidades.

19.2 Arquitectura del sistema integral para la flexibilidad

La arquitectura del sistema permite obtener conceptualmente un entorno en el que un conjunto de objetos estructurado dentro de un modelo de objetos común reside en un espacio único de objetos (de usuario). Los servicios del sistema[1] se proporcionan de manera uniforme mediante ciertos objetos de ese espacio único.

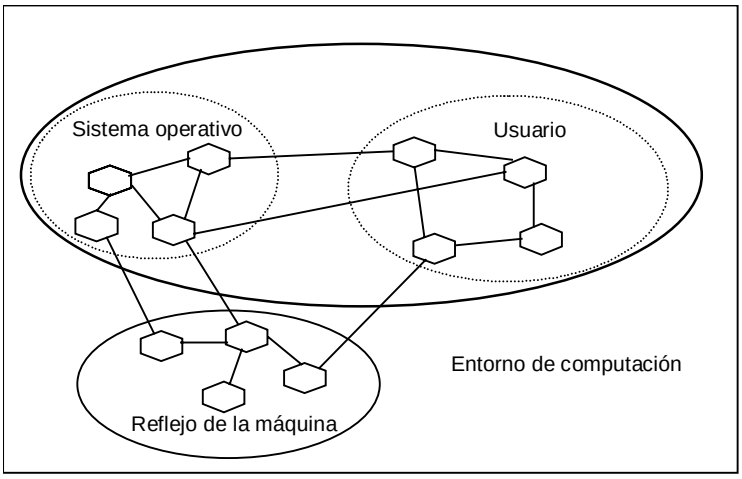


Figura 19.1 Espacio de objetos homogéneo formado por la unificación de los objetos de la máquina con los del sistema operativo y los del usuario.

Como se verá a continuación, la funcionalidad proporcionada en espacio de usuario es un elemento fundamental para lograr la flexibilidad. Otro elemento clave es poder aplicar la orientación a objetos a un sistema. Esta arquitectura combina los dos elementos: conceptualmente toda la funcionalidad está en el espacio de usuario y utiliza la OO. La reflectividad permite que también los objetos de la máquina se incorporen a este espacio único junto con los del sistema operativo y los de usuario. Las ventajas de la OO alcanzan a todo el sistema.

Como se comprueba en las siguientes secciones, el sistema resultante forma un entorno de computación integral OO muy flexible.

19.3 Tecnologías para obtener la flexibilidad

Cahill [Cah96] establece cinco técnicas, no excluyentes entre sí, que se pueden utilizar para lograr la flexibilidad. El sistema integral aúna las propiedades de las diferentes técnicas:

19.3.1 Tecnología de micronúcleos

Se basa en construir un sistema mediante un micronúcleo en el espacio del sistema que proporciona las abstracciones básicas del sistema y de proporcionar el resto de la funcionalidad mediante servidores en el espacio del usuario [RTA+96].

Esto permite trabajar directamente con la interfaz del micronúcleo si los servidores no son adecuados para una aplicación. Estos servidores al estar en el espacio del usuario se pueden reemplazar más fácilmente.

[1] La funcionalidad que se considera de “sistema”, que tradicionalmente reside en los núcleos de los sistemas operativos.

Sin embargo, el propio micronúcleo es un elemento monolítico muy inflexible. Por otro lado, el reparto de los servicios entre el núcleo y el espacio de usuario no se suele poder alterar.

En cierta manera, el sistema integral puede considerarse como compuesto por un micronúcleo, que forman los objetos de la máquina que dan el soporte básico del sistema y que están implementados de forma primitiva. El resto de la funcionalidad del sistema está escrita en el espacio de usuario.

19.3.2 Sistemas operativos específicos para las aplicaciones

Son sistemas en los que el sistema operativo se reduce a un elemento mínimo que intermedia en las peticiones de recursos físicos de las aplicaciones y toda la funcionalidad se proporciona en el espacio de usuario, por ejemplo mediante librerías que se enlazan con las aplicaciones.

Pueden considerarse una extrapolación de la tendencia de los micronúcleos de llevar más funcionalidad al espacio del usuario. En este caso se lleva prácticamente toda. Por eso se suelen denominar sistemas de núcleo mínimo o piconúcleos [ABB+93].

El sistema integral también participa de esta tendencia, pues desde el punto de vista externo toda la funcionalidad, todos los objetos del sistema residen en un espacio de objetos (de usuario) único. En este espacio se colocarán los objetos con funcionalidad del sistema que necesiten las aplicaciones.

19.3.3 Orientación a objetos

Los beneficios de la orientación a objetos [Boo94] pueden aplicarse en la construcción de un sistema operativo. La reutilización del código, mejora de la portabilidad, facilidad de mantenimiento y extensibilidad incremental se pueden aplicar a la construcción de sistemas más flexibles[1] y pueden ser aprovechadas por las aplicaciones mediante el uso de una interfaz del sistema también orientada a objetos.

Russo [Rus91] caracteriza un sistema operativo orientado a objetos como "*un sistema operativo que se estructura internamente como objetos y presenta todos los recursos accesibles a las aplicaciones cliente como un conjunto de objetos*". Dado que esto es precisamente lo que hace el sistema operativo SO4 (y mediante la reflectividad se extiende también a la máquina abstracta), puede calificarse de esta manera.

El estructurar el sistema mediante un conjunto de objetos de la máquina (por la reflectividad) y un conjunto de objetos del sistema operativo, que pueden ser accedidos como objetos normales por parte del usuario permite aplicar todas las ventajas de la orientación a objetos no sólo a las aplicaciones de usuario[2], si no también al propio sistema operativo (y a los propios objetos de la máquina).

Reusabilidad

Un ejemplo de esto es la reutilización del código. Se puede aplicar también a los propios objetos del sistema operativo. Estos, al igual que los objetos de usuario[3], están organizados dentro de la jerarquía de clases del sistema. Por tanto, el código de los objetos del sistema operativo puede ser reutilizado por cualquier otro objeto de usuario, por medio

[1] Esta es precisamente la filosofía fundamental que impregna al sistema integral.
[2] Dar soporte directo a la orientación a objetos para las aplicaciones de usuario es el punto de partida del sistema integral.
[3] De hecho son objetos de usuario.

de la herencia. Por ejemplo, la funcionalidad del sistema de persistencia del sistema operativo puede ser aprovechada para implementar bases de datos orientadas a objetos [ADA+96].

19.3.4 Familias de programas

Son conjuntos de sistemas operativos que comparten un conjunto de características comunes. Se diferencian en que pueden tener funcionalidad adicional o diferentes implementaciones según el ámbito de uso o el hardware destino. Normalmente se estructuran en torno a una serie de servicios definidos mediante una implementación mínima. Cada miembro de la familia escoge qué servicios necesita y en qué forma se implementarán. De esta manera se ajusta el sistema a cada tipo de aplicación y entorno.

En la implementación del sistema integral pueden considerarse los servicios mínimos como los que dan soporte al modelo de objetos y se implementan de forma primitiva. Cada implementación puede elegir la mejor manera de realizarlos en función del ámbito de aplicación y la plataforma destino. De la funcionalidad escrita en objetos de usuario se puede escoger la que se quiere incluir.

19.3.5 Reflectividad e implementación abierta

La implementación abierta permite que se pueda acceder a los mecanismos internos de una abstracción para adaptar su uso a cada caso particular. La reflectividad es un ejemplo de implementación abierta en la que un sistema base puede acceder a su meta-sistema, normalmente dentro del mismo modelo o marco de referencia. De esta forma se puede adaptar el meta-sistema según las necesidades de las aplicaciones del sistema base.

En el sistema integral se incorpora la reflectividad. Se describe el meta-sistema como un conjunto de objetos de la máquina que se organizan dentro del mismo marco que los objetos de usuario. Estos pueden utilizar (y ser utilizados) de manera normal a los objetos de usuario. Así, mediante el uso de una meta-interfaz adecuada, los objetos de usuario pueden indicar al sistema que utilice una determinada política de persistencia, política de planificación, etc.

19.4 Clasificación de los tipos de flexibilidad y ejemplos en el sistema integral

Atendiendo al momento en que se ajusta el software de sistema a las necesidades de las aplicaciones o entornos de aplicación se pueden distinguir dos tipos de flexibilidad: **flexibilidad estática** y **flexibilidad dinámica**, con diferentes variantes.

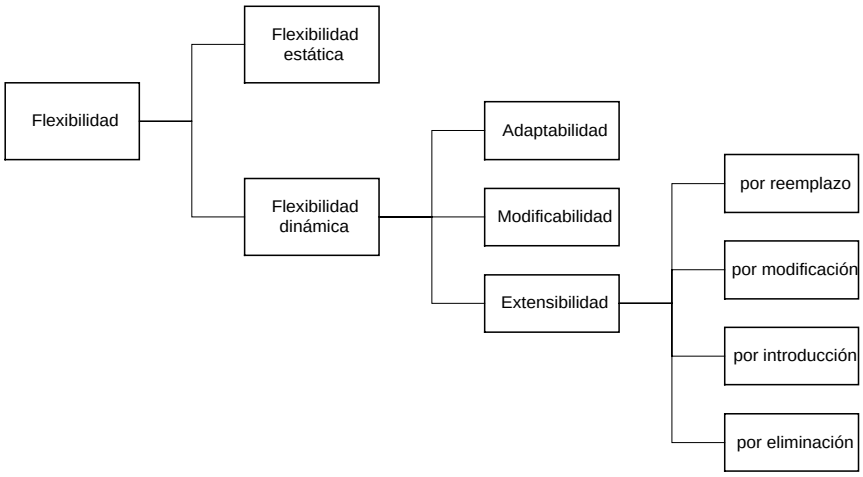


Figura 19.2 Tipos de flexibilidad.

19.4.1 Flexibilidad estática

El ajuste del software del sistema se produce en el momento de su construcción primitiva. Es decir, en el momento de compilación, enlace o carga inicial del mismo, se incluyen en el sistema las características necesarias para el ámbito de aplicación deseado. Como su nombre indica, lo que queda incluido en el sistema ya no puede ser modificado sin repetir el proceso. La flexibilidad se entiende por las facilidades que incorpore el código fuente primitivo del sistema para ser ajustado en el momento de su construcción. Normalmente se suelen utilizar técnicas de programación orientada a objetos, marcos de aplicación, etc. que permiten construir sistemas particularizados mediante la herencia en tiempo de compilación.

La flexibilidad estática es menos potente que la dinámica, pues debe predeterminarse de antemano qué incluir en el sistema primitivo, y acertar en la elección, puesto que no se puede reajustar con facilidad (habría que cambiarlo y recompilar).

En el sistema integral, el ajuste estático del sistema se realiza en el momento de elegir qué objetos del mismo se van a implementar de manera primitiva y cuáles no, en función de la plataforma destino y las aplicaciones. Por ejemplo, para un entorno destinado a la ejecución de aplicaciones de cálculo científico se incluirían como primitivas todas las clases y métodos relacionados con el cálculo, como las de matrices, producto de matrices, etc.

Otro aspecto relacionado con el anterior es la estructura del código fuente de la implementación primitiva, que tiene que permitir hacer estas elecciones con facilidad. El marco de aplicación de la máquina, descrito en el capítulo 13, aprovecha la OO de tal manera que se facilita la adición de nuevas clases primitivas. Mediante la herencia se pueden añadir nuevas clases primitivas, con su nueva funcionalidad, sin afectar al resto de la implementación:

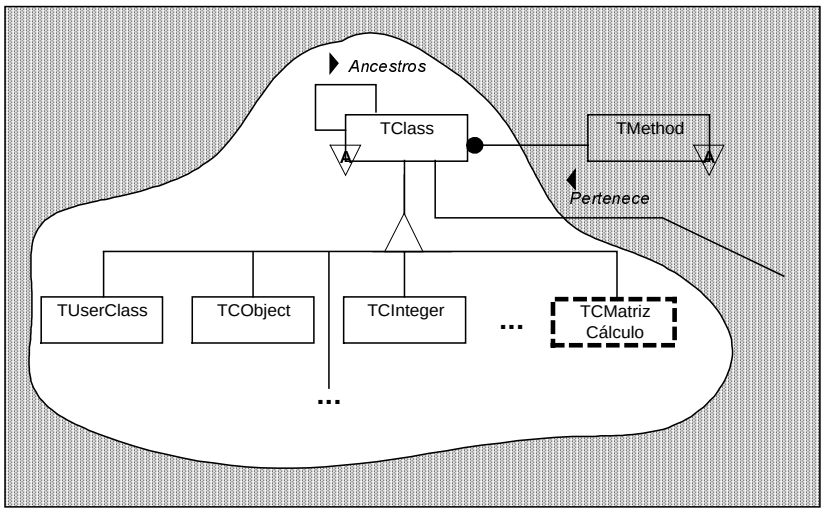


Figura 19.3 Flexibilidad estática por adición de nuevas clases primitivas mediante la herencia.

19.4.2 Flexibilidad dinámica

La flexibilidad dinámica permite ajustar el software de sistema a la medida de las necesidades de las aplicaciones en tiempo de ejecución. Es, por tanto, más potente que la estática puesto que el tiempo y el esfuerzo necesarios para hacer los ajustes son mucho más rápidos, al poder hacerse en tiempo de ejecución. Esto también permite que el sistema de mejor soporte a las necesidades de las aplicaciones no previstas en la implementación primitiva del mismo.

De esta manera la flexibilidad dinámica permite que sea el usuario final el que pueda ajustar el sistema de una manera rápida, sin necesitar acudir al fabricante (al código fuente de la implementación primitiva) para realizar los cambios.

Existen diferentes técnicas que permiten flexibilidad dinámica, en grado creciente de versatilidad:

19.4.2.1 Sistemas adaptables y adaptativos

En un **sistema adaptable** se incluye en la implementación primitiva del mismo un abanico de políticas y servicios predefinidos, válidos para un conjunto amplio de aplicaciones. Además, se proporcionan una serie de interfaces que permitan que las aplicaciones influyan en la selección de estas políticas o servicios (por ejemplo con reflectividad). En lugar de proporcionar una sola política se proporciona un abanico de ellas que pueden escoger las aplicaciones[1]. Un ejemplo son los sistemas que permiten que las aplicaciones den pistas (*hints*) que aconsejan al núcleo la política de memoria virtual que se aplica, como la llamada `madvise` del sistema operativo SunOS o la política de planificación a usar en el caso del Mach [Bla90]. Sin embargo, el conjunto de servicios en los que elegir es fijo y no se puede cambiar.

[1] Normalmente mediante el ajuste de parámetros del sistema, o la elección directa entre una serie de opciones.

En el caso del sistema integral, este tipo de flexibilidad puede incluirse fácilmente. Simplemente basta con proporcionar el conjunto de métodos adecuados en los objetos que proporcionan los servicios del sistema. Los objetos de las aplicaciones utilizarán estos métodos que les permitirán indicar a los objetos del sistema que tipo de política, etc. quieren que se aplique. Por ejemplo, en el caso de la persistencia, el objeto de persistencia puede tener un método que le indique que al almacenar los objetos en disco lo haga de una manera más segura, mediante una doble comprobación de la información grabada.

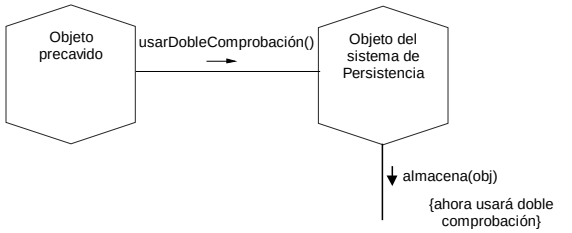


Figura 19.4 Adaptabilidad en el sistema integral mediante métodos que seleccionan la política.

Adaptatividad

Un **sistema adaptativo** es aquél que es capaz de seleccionar por sí mismo las políticas más adecuadas para las aplicaciones en cada momento, por ejemplo basándose en un análisis de su conducta anterior.

El sistema integral ofrece todo el potencial para convertirse en adaptativo. Por una parte, incluir en el sistema elementos que permitan a cada servicio tomar decisiones en función de la situación actual es muy sencillo. Basta con incluir la política como un objeto separado que es consultado por el servicio en el momento de tomar la decisión. En este objeto se incluye el conocimiento necesario para evaluar la situación y escoger la mejor opción. También se puede reemplazar el objeto de la política por otro objeto que implemente otra política diferente (extensibilidad). Por ejemplo en el caso del equilibrio de carga, en función de una serie de parámetros se puede decidir mover los objetos a otras máquinas menos cargadas [Alv96a].

Otro elemento muy importante del sistema integral que ayuda a la adaptabilidad es el alto nivel de abstracción junto con el tipo de modelo de objetos. El sistema tiene toda la información de la semántica de los objetos que existen en el mismo. Gracias a las relaciones de agregación y de asociación conoce de antemano una información fundamental para la adaptabilidad. Por ejemplo, en el caso de la persistencia, en el momento de traer un objeto del almacenamiento secundario puede aprovecharse esta información para traer también los agregados, puesto que se suelen utilizar a la vez que el objeto principal. De la misma manera en la distribución, al mover un objeto se mueve con sus agregados. En este caso puede tenerse en cuenta las relaciones de asociación. Los objetos relacionados entre sí intercambiarán muchos mensajes, con lo que es más conveniente colocarlos cercanos en la misma máquina.

19.4.2.2 Sistemas modificables

Son sistemas que permiten involucrar a las aplicaciones de una manera u otra en la implementación de un servicio del sistema. Un ejemplo de esto suelen ser los micronúcleos como Mach, que permiten que el sistema de memoria virtual del micronúcleo llame a un servidor de usuario, que es el que implementa la política de paginación [ABB+86]. Normalmente este tipo de comunicación se realiza mediante una interfaz ad-hoc, específica de cada servicio.

El sistema integral permite también este tipo de flexibilidad, aunque de una manera más uniforme. Todos los objetos en el sistema comparten el mismo espacio y pueden comunicarse invocando métodos. No existe la distinción entre el espacio del sistema y el del usuario, así que no hay necesidad de utilizar un mecanismo especial para permitir este tipo de colaboración. La analogía puede establecerse cuando el objeto en cuestión es un objeto de la máquina, normalmente implementado de manera primitiva, que colabora con un objeto de usuario, por ejemplo para llevar a cabo la persistencia (véase el capítulo 16). La reflectividad de la máquina permite este tipo de colaboración (y a la inversa) de manera uniforme al colocar ambos objetos dentro del mismo espacio conceptual y modelo de objetos.

19.4.2.3 Sistemas configurables y extensibles

Estos sistemas permiten que (algunos) servicios del sistema se puedan añadir o reemplazar de manera dinámica para soportar nueva o diferente funcionalidad para las aplicaciones. Para ello es importante que esta funcionalidad resida en el espacio de usuario.

Un sistema es **configurable** cuando permite lo anterior para los servicios predefinidos del sistema. Por ejemplo, la mayoría de los micronúcleos permiten en tiempo de ejecución poner en marcha un servicio del sistema en tiempo de ejecución, o bien cambiarlo por otro servidor que dé la misma funcionalidad.

Extensibilidad

Un sistema es **extensible** cuando además de lo anterior, permite añadir nueva funcionalidad en el sistema de manera dinámica. Es el tipo de flexibilidad más potente. En muchos casos se suele englobar con el nombre de extensibilidad a todos los tipos de flexibilidad dinámica, en el sentido de que un sistema extensible normalmente permite también los otros tipos de flexibilidad.

19.5 Extensión en el sistema integral

La extensión en el sistema integral se realiza proporcionando los servicios mediante un conjunto de objetos[1] en el espacio de usuario. Para acceder a los servicios se utilizará una referencia que apunte al objeto y mediante esta referencia invocar sus métodos. La reflectividad del sistema junto con la orientación a objetos permiten una **extensibilidad incremental**, de grano más fino que la que presentan los micronúcleos. Los micronúcleos simplemente permiten el cambio de un servidor completo por otro. En el sistema integral el servicio se descompone en sus objetos integrantes, que pueden tratarse individualmente. La reflectividad permite aplicar uniformemente la OO en las extensiones, simplemente asignando de manera adecuada los objetos a los que apuntan las referencias.

19.5.1 Tipos de extensibilidad

Pueden distinguirse tres tipos de extensibilidad, dependiendo de si se reemplaza, modifica, añade o elimina un servicio del sistema.

19.5.1.1 Extensibilidad mediante reemplazo

Consiste en reemplazar un servicio del sistema (o partes de un servicio) con otro servicio que ofrezca la misma funcionalidad de manera alternativa.

[1] Aunque por sencillez en los ejemplos se engloban bajo un único objeto que proporciona el servicio.

Para reemplazar el servicio, simplemente hay que hacer que la referencia apunte a un nuevo objeto con la misma funcionalidad[1]. Para ello este objeto tiene que ser del mismo tipo que el anterior, es decir, de la misma clase o de clases derivadas. Puede aprovecharse esto para reutilizar código del servicio anterior, heredándolo y cambiando sólo la funcionalidad que interese. El nuevo objeto es compatible con el anterior, se usará la misma interfaz, pero ahora proporcionará la funcionalidad de otra manera. Por ejemplo, se puede cambiar el objeto de persistencia que proporciona el sistema por otro objeto que utilice un mecanismo de persistencia diferente:

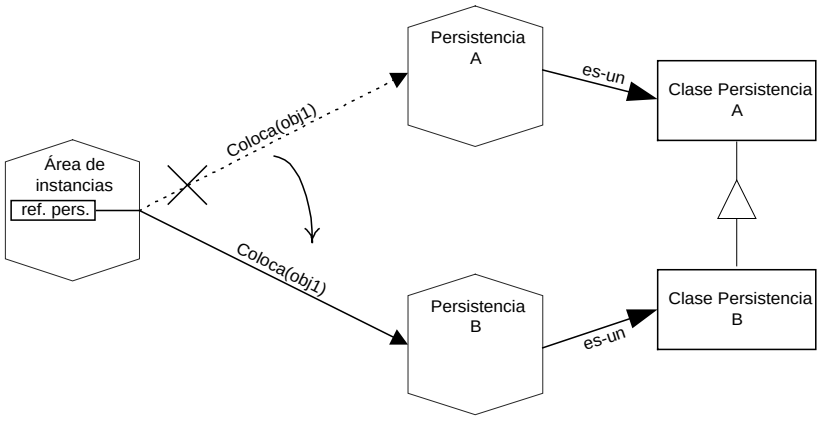


Figura 19.5 Extensibilidad mediante reemplazo por reasignación de referencias.

En este caso, lo más probable es que la interfaz del área de instancias[2] tenga un método que permita cambiar el objeto al que apunta su referencia interna de acceso al objeto de la persistencia (areaInstancias.setPersistencia(nuevoObjetoPersistencia)).

19.5.1.2 Extensibilidad mediante modificación

En la que se extiende un servicio, añadiendo elementos a su representación o a su interfaz.

Es el caso anterior en la que no sólo (conceptualmente) se cambia el servicio, si no que se añaden elementos al mismo. Por ejemplo, en el caso de la persistencia, al heredar se pueden añadir nuevas operaciones a la interfaz. En cualquier caso, para poder usar estas nuevas operaciones, es necesario que los clientes del servicio cambien para utilizarlas.

Sin embargo, como se ve en la figura anterior##, la OO hace que aunque el servicio tenga más funciones, se mantenga al menos la interfaz anterior. El polimorfismo permite que todos los clientes del servicio anterior utilicen sin cambios el nuevo servicio (con la interfaz de la superclase). En el caso de nuevos clientes podrían utilizar ya directamente además de las funciones anteriores, las nuevas funciones de la interfaz.

19.5.1.3 Extensibilidad mediante introducción

En la que se introducen nuevos servicios en un sistema que antes no disponía de ellos.

Por ejemplo, pueden añadirse sobre la marcha nuevos elementos al sistema integral, que proporcionen funcionalidad adicional a la ya existente, integrándose con esta. Un ejemplo sería añadir el soporte para distribución en un sistema que no se hubiera entregado con el mismo. Simplemente bastaría con añadir al espacio de usuario[1] los objetos de ese soporte e

[1] Este mecanismo es similar a la reasignación de manejadores (*handlers*) que utilizan sistemas como Kea [VH96] u Off [BF97] para delegar o redefinir servicios. En el caso del sistema integral, sin embargo, está integrado de manera uniforme con el modelo de objetos y la programación de usuario.

[2] Como es un objeto que proporciona funcionalidad de sistema y probablemente con implementación primitiva, se suele denominar meta-interfaz o protocolo de meta-objeto (MOP, *Meta-Object Protocol*).

integrarlos con el resto del sistema. Para ello existirá como en el caso anterior una interfaz que permita esta integración, haciendo que los objetos del sistema existentes[2] adquieran la referencia al objeto de distribución (máquina.setDistribución(objetoDistribución)). A partir de ese momento todos los objetos del sistema adquieren automáticamente la propiedad de la distribución y pueden invocar métodos de objetos remotos, moverse a otras máquinas, etc.

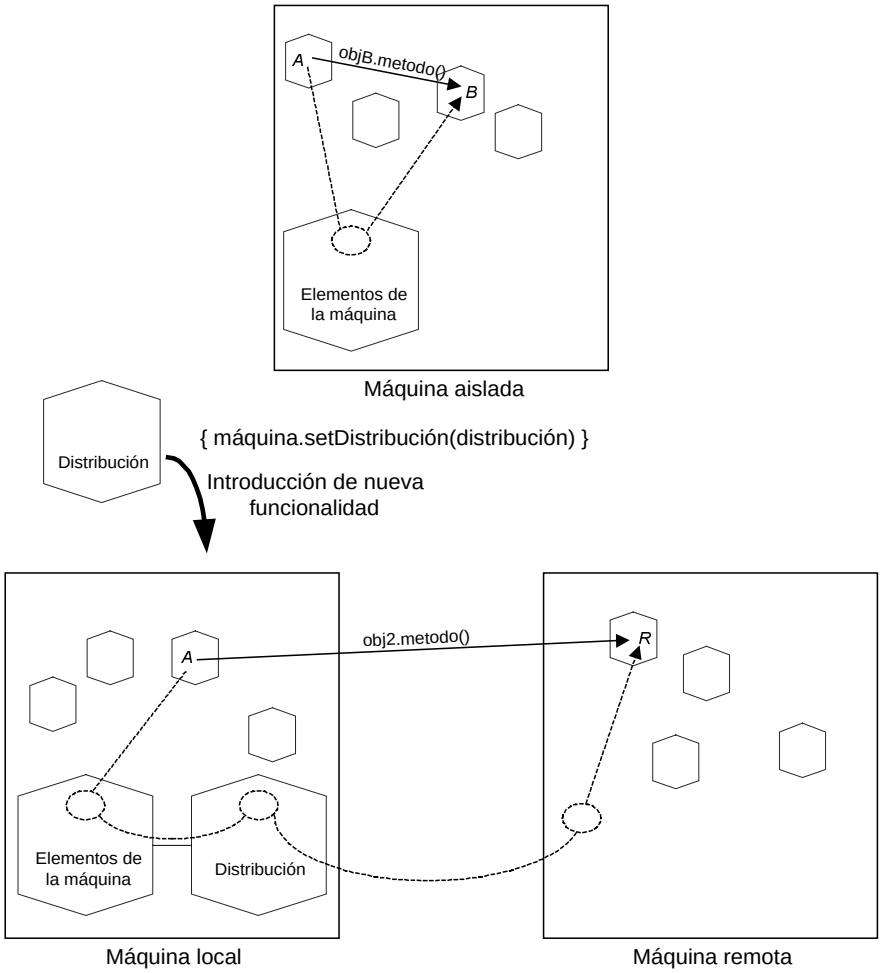


Figura 19.6 Extensibilidad mediante introducción de nueva funcionalidad en el sistema integral.

19.5.1.4 Extensibilidad mediante eliminación

A los tipos anteriores de extensibilidad puede añadirse la posibilidad de eliminar funcionalidad que no sea necesaria en el sistema. Aquella funcionalidad que no esté colocada en el espacio de usuario no podrá ser nunca eliminada dinámicamente del sistema. Este tipo de extensibilidad es importante para no penalizar con funcionalidad adicional a aquellas aplicaciones que no la necesiten.

En el sistema integral, al tener todos los elementos conceptualmente en el espacio del usuario, en principio cualquier objeto del sistema puede ser eliminado si no es necesario (excepto en el caso de que esté implementado de forma primitiva). Esto incluye también a los propios objetos de la máquina. Por ejemplo, los objetos de la máquina que lleven a cabo el mecanismo de protección (véase el capítulo 15) pueden ser eliminados (formarán parte del

[1] Por ejemplo mediante un soporte magnético extraíble.
[2] Los encargados de la invocación de métodos.

mecanismo de invocación de métodos). Esto tiene sentido en un sistema empotrado que funciona de manera aislada, y cuyos objetos están totalmente depurados, por lo que ya no se necesita un mecanismo que proteja los objetos de accesos no autorizados. Esto requiere que los objetos que comprueban la protección no tengan implementación primitiva, para poder eliminarlos dinámicamente[1]. También se necesita que el resto de los objetos de la máquina implicados tengan una interfaz que permita que se adapten a la nueva situación, saltando el paso de comprobación de permisos. Por ejemplo con un método envíoMensajes.eliminarProtección(). Si en un momento posterior se necesitase de nuevo esta funcionalidad se podría añadir dinámicamente, como en el apartado anterior.

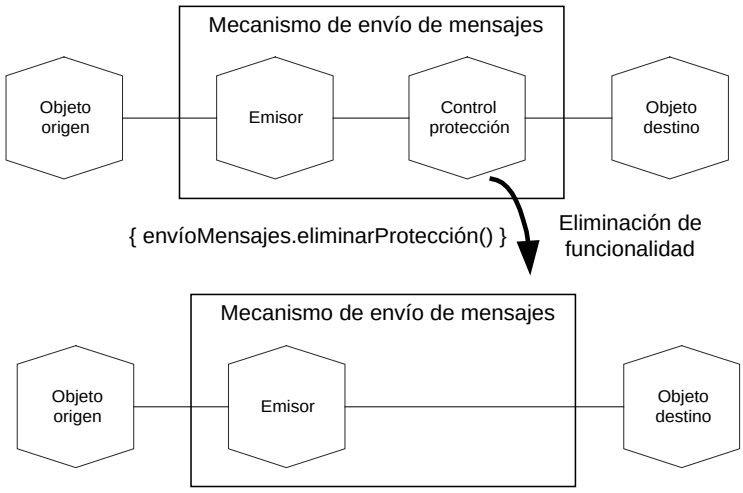


Figura 19.7 Extensibilidad mediante eliminación de funcionalidad no necesaria.

19.5.2 Resumen de la flexibilidad en el sistema integral

Los elementos que utiliza el sistema integral para lograr los diferentes tipos de flexibilidad estática y dinámica (adaptabilidad, configurabilidad y extensibilidad) se resumen a continuación.

19.5.2.1 Espacio único de objetos. Reflectividad.

Existe un espacio común donde están todos los objetos que proporcionan funcionalidad en el sistema, conceptualmente al mismo nivel. Gracias a la reflectividad, la uniformidad de uso de la OO también alcanza a la funcionalidad de la máquina.

19.5.2.2 Implementación primitiva OO. Flexiblidad estática

Sobre los objetos implementados de manera primitiva se puede aplicar flexibilidad estática al estar implementados con un lenguaje OO en un marco de aplicación modificable fácilmente por la herencia.

19.5.2.3 Implementación de usuario. Flexibilidad dinámica

Sobre los objetos no implementados de manera primitiva se puede aplicar la flexibilidad dinámica.

[1] En caso de tener implementación primitiva habría que eliminarlos de manera estática en dicha implementación (flexibilidad estática).

19.5.2.4 Uso de la OO de manera uniforme. Interfaces de control de la flexibilidad

La reflectividad permite usar la OO para todos los objetos del sistema. Se utilizan los mecanismos normales de la OO de manera uniforme para la flexibilidad: asignación de nuevos objetos a las referencias (extensibilidad por reemplazamiento), y la extensibilidad incremental: reutilización de código de servicios por la herencia con polimorfismo y añadiendo nueva funcionalidad (extensibilidad por modificación), etc. Los objetos involucrados tienen que estar acompañados de una serie de interfaces que permitan controlar externamente las asignaciones de objetos, como en la modificabilidad.

19.6 Control de la flexibilidad

La extensibilidad en los sistemas flexibles introduce una serie de problemas[1] que hay que resolver, especialmente en el campo de la seguridad y el mecanismo de protección [GB97]. En los sistemas extensibles actuales se utilizan soluciones no generales.

19.6.1 Control ad-hoc de la extensibilidad

Cuando se añade de manera dinámica código a un sistema ya existente, hay dos aspectos que hay que controlar: la seguridad de funcionamiento del sistema y la protección de los elementos del sistema.

19.6.1.1 Seguridad de funcionamiento

Se trata de que las extensiones que se añaden al sistema no puedan comprometer el funcionamiento del mismo. Si se permite que una extensión acceda sin control a cierta información del sistema podría corromperla. Por ejemplo, en el caso del soporte de un nuevo sistema de ficheros que corrompa el soporte genérico de acceso del sistema.

Por ejemplo, para garantizar la seguridad de funcionamiento se usan diferentes técnicas específicas [SS96], que hacen que las extensiones queden confinadas en su ámbito:

- **Protección software**

  En la que se programan las extensiones en lenguajes especiales fuertemente tipados como Java en el caso de la plataforma Java [KJS96] o Modula3 en el sistema SPIN [BSP+95]. Estos lenguajes garantizan que los programas no acceden información fuera de su ámbito. Otras técnicas son el aislamiento de fallos de software (SFI, *Software Fault Isolation*) que parchea el código binario eliminando los fallos de acceso fuera del ámbito permitido, como en el sistema VINO [SES+96].

- **Protección hardware**

  Las extensiones se colocan en el espacio del usuario. De esta manera no pueden acceder al espacio del sistema (en sistemas con división tradicional usuario/núcleo).

19.6.1.2 Protección del sistema

Se trata de controlar qué elementos del sistema se pueden extender y por quién. No se debe poder extender el sistema sin control. Por ejemplo, permitir que cualquier elemento pueda añadir o eliminar el soporte de persistencia en un sistema, o que una extensión pueda llamar a cualquier función de los servicios del sistema.

---

[1] Algunos autores [DPZ97] arguyen que estos problemas son excesivos y que los resultados de los sistemas extensibles se pueden aplicar en sistemas convencionales sin necesidad de usar un sistema extensible. Proponen, sin embargo, usar estos sistemas flexibles para la investigación por la dinamicidad que introducen en la investigación.

Para proteger el sistema también se emplean soluciones específicas y poco flexibles. Por ejemplo, en el sistema Java, se dividen las extensiones en dos tipos. Las extensiones de confianza (objetos locales) pueden acceder a cualquier elemento del sistema. En las extensiones traídas del exterior por la red no se confía y sólo pueden acceder a una "caja de arena" (*sandbox*) que limita los servicios a los que pueden acceder.

Otros sistemas como SPIN, dividen los servicios en dominios y hacen que las extensiones sólo puedan acceder dentro del dominio en el que se incluyen [SFP+96]. Sin embargo, no hay un control de grano fino dentro del dominio, la extensión puede acceder a todos los servicios del dominio, a no ser que cada servicio implemente su propia política de control de acceso cuando son extendidos.

19.6.2 Control uniforme de la extensibilidad con el mecanismo de control de objetos del sistema integral

Se necesita un mecanismo de control uniforme de grano fino que permita controlar con más detalle qué servicios del sistema se pueden extender y por quién [GB97]. Por otro lado la programación de las extensiones no debe estar limitada por la utilización de unas técnicas especiales para garantizar su seguridad.

El sistema integral dispone de un mecanismo de control uniforme de grano fino basado en capacidades y que se integra en el propio modelo del sistema [Día96]. Este mecanismo se usa para el control de acceso de los métodos de cualquier objeto. Como las extensiones en el sistema integral son simples objetos, el mismo mecanismo de control sirve para controlar las extensiones y las extensiones se programan con cualquier lenguaje que genere objetos del modelo del sistema. Se unifica la programación y el control de las extensiones con el modo de trabajo normal del sistema, no es necesario introducir nuevos mecanismos especiales para controlar la extensibilidad en el sistema.

19.6.2.1 Seguridad de funcionamiento

El propio modelo de objetos del sistema garantiza que un objeto no puede acceder sin autorización fuera de sí mismo. La única manera que tiene un objeto de modificar el resto del sistema es invocando métodos de otros objetos, y esto sólo lo puede hacer si tiene la autorización necesaria. Por consiguiente, un objeto (aunque sea malicioso) nunca podrá modificar nada sin autorización.

Otro problema es que las extensiones estén bien programadas y que realicen adecuadamente su cometido. Este es un problema más bien de ingeniería de software: la construcción de programas correctos. En la mayoría de los casos, las extensiones que involucren a elementos fundamentales del sistema serán escasas, y si se incorporan es porque se tienen suficientes garantías de que funcionarán bien, con lo que este problema es menor [CNK+97]. Cuando por ejemplo se añade una extensión para dar soporte a persistencia es porque se tiene una certeza razonable de que funciona correctamente.

19.6.2.2 Protección del sistema

El mecanismo de protección uniforme permite controlar qué objetos tienen acceso a otros objetos al nivel de métodos individuales. La decisión de qué objetos se pueden extender y por quién es una decisión política del usuario, pero se puede controlar con el grado de detalle necesario con el mecanismo de protección. Por ejemplo, autorizar a un objeto E para extender a otro objeto S es sinónimo de permitir que E tenga acceso a S. Esto requiere que se le pase a

E la referencia del objeto S[1]. La referencia es una capacidad, con lo que no se da simplemente acceso a S, se puede indicar exactamente qué métodos individuales del objeto S se pueden llamar. Por ejemplo, para que un objeto extienda la máquina con un servicio de persistencia es necesario pasarle a este objeto la referencia (capacidad) al objeto de la máquina correspondiente, y se especificará exactamente cuáles son los métodos que puede llamar.

19.7 Resumen

La flexibilidad en el software de sistema es necesaria para dar un mejor soporte a las aplicaciones, más ajustado a sus necesidades. La arquitectura del sistema integral comparte elementos de las diferentes tecnologías descritas por Cahill para lograr la extensibilidad: micronúcleos, sistemas operativos específicos, orientación a objetos, familias de programas y reflectividad. Esto permite lograr los diferentes tipos de flexibilidad descritos por Cahill, de manera uniforme mediante técnicas orientadas a objetos, especialmente la flexibilidad dinámica.

El resultado es un entorno que soporta fácilmente a través de la OO la flexibilidad estática y la dinámica: adaptabilidad, modificabilidad, configurabilidad y extensión por reemplazo, modificación, introducción y eliminación. A esto contribuyen fundamentalmente el espacio conceptualmente único de objetos de usuario con el mismo modelo OO que se extiende a los objetos de la máquina por la reflectividad.

Por otro lado, el mecanismo de control de acceso a los objetos del sistema permite un control uniforme de las extensiones, solucionando de manera general el problema del control de la extensibilidad de otros sistemas extensibles.

[1] Pasarla como parámetro de un método del objeto o bien recibiéndola como resultado de la invocación de un método de otro objeto es la única manera de que un objeto adquiera una nueva referencia.

Capítulo 20

ÁMBITOS DE APLICACIÓN DEL SISTEMA INTEGRAL ORIENTADO A OBJETOS

En este capítulo se describen brevemente algunos ámbitos en los que el sistema integral puede aplicarse de una manera ventajosa, debido a sus características de portabilidad, flexibilidad, homogeneidad, distribución y persistencia.

Puede considerarse al sistema integral como una familia de sistemas que comparten un modelo de objetos y funcionamiento OO común. Las características de la arquitectura del sistema permiten que se adapte a muy diferentes ámbitos de aplicación, aunque compartiendo en todos una filosofía común. Las soluciones para los distintos ámbitos de aplicación no necesitan desarrollar elementos especiales ad-hoc para las mismas, se hacen de manera uniforme dentro del paradigma OO del sistema.

20.1 Sistemas empotrados y de aplicación especial

Los **sistemas empotrados** (*embedded systems*) se caracterizan por ser sistemas de computación destinados a una tarea específica, normalmente con unos recursos limitados, como la memoria disponible. Se llaman empotrados[1] porque están integrados físicamente con el elemento con el que trabajan. El hardware sobre el que funcionan suele ser muy variado, y depende del destino de cada sistema. Ejemplos de sistemas empotrados son sistemas de control de maquinaria industrial, controladores de motores, etc.

[1] También se les denomina sistemas encastrados o inmersos.

20.1.1 Portabilidad. Homogeneidad de desarrollo

La característica de portabilidad de la máquina abstracta hace que el sistema integral funcione de la misma manera en plataformas hardware diferente. Esto permitiría realizar la programación de los sistemas empotrados de una manera homogénea. El desarrollo de sistemas empotrados conlleva normalmente utilizar unas herramientas de desarrollo y modelos de programación diferentes para cada plataforma hardware y sistema desarrollado. Al usar el sistema integral, el modelo de programación y la plataforma destino son siempre los mismos, independientemente del hardware final y del sistema que se trate. Esto ahorra mucho esfuerzo de desarrollo. Además, también se pueden aprovechar las ventajas que proporciona la OO presente de manera uniforme en el sistema.

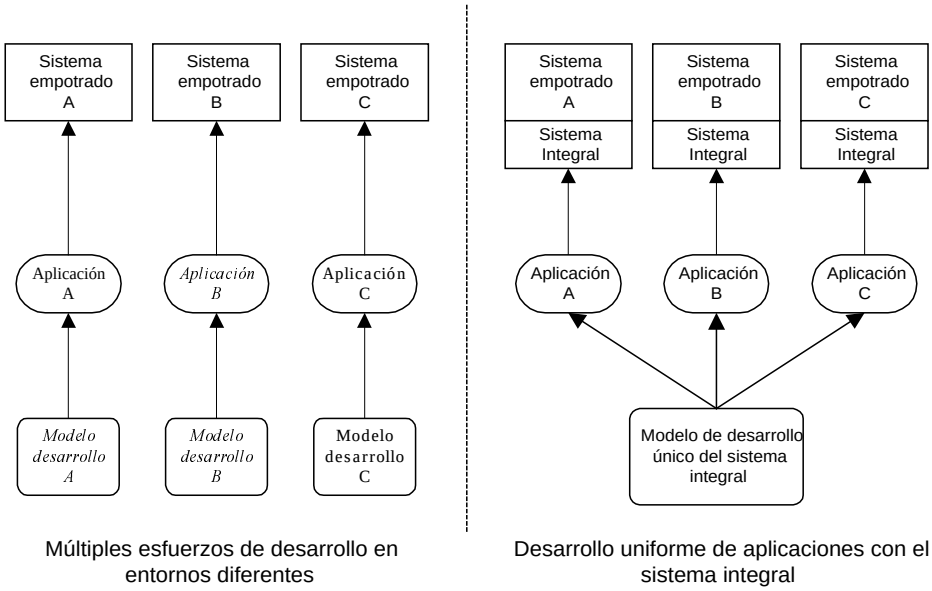


Figura 20.1 Desarrollo uniforme aplicaciones en lugar de múltiples entornos de desrrollo diferentes.

20.1.2 Flexibilidad. Adaptación al entorno final

La flexibilidad del sistema permite adaptarlo a las características físicas del entorno hardware final y la aplicación desarrollada. Por ejemplo, una vez depurada la aplicación y comprobado que no tiene errores, se puede eliminar el mecanismo de protección del sistema para aumentar la eficiencia. O bien, si el sistema va a trabajar de manera aislada y no se necesita soporte de distribución, éste se puede eliminar fácilmente.

20.1.3 Extensibilidad y transparencia. Fácil actualización

La actualización del sistema empotrado es mucho más sencilla. En caso de necesitarse en un futuro algunas características de las eliminadas anteriormente, se pueden incluir de manera dinámica fácilmente, sin detener el sistema. Por ejemplo, se podría añadir el soporte de distribución[1] para conectar en red el sistema empotrado. Simplemente esta actualización permitiría de manera transparente acceder a objetos situados en otros sistemas, sin modificar en nada el resto de la aplicación.

[1] También se puede elegir los elementos del sistema de distribución que se necesiten. Por ejemplo, en este tipo de sistemas no se suele necesitar un servicio de denominación, si no simplemente de localización de objetos a partir de su identificador.

La estructura modular en objetos del sistema permite una actualización incremental del sistema. No hay que cambiar todo el sistema cada vez que se hagan modificaciones. Por ejemplo, una mejora en una parte del sistema se puede añadir sin afectar al resto del sistema. Simplemente se debe cambiar el objeto antiguo por el nuevo objeto mejorado.

20.2 Grandes sistemas distribuidos heterogéneos

Estos sistemas se caracterizan por la existencia de un número muy grande de máquinas conectados en una red común, en las que las máquinas tienen configuraciones hardware diferentes. Un ejemplo típico de este tipo de sistemas es la red de cajeros automáticos de un banco. Existen cientos o miles de cajeros que están conectados al sistema central del banco, y varias marcas y modelos diferentes de cajeros. Puede verse como el caso anterior de sistemas empotrados, conectados en red. Las ventajas mencionadas para ese caso también son aplicables para este. Además, se describen aquí algunas ventajas adicionales:

20.2.1 Portabilidad. Mantenimiento de una versión en lugar de múltiples versiones

Un sistema de este estilo presenta muchos problemas. Un problema muy grande es el mantenimiento del software de aplicación de los cajeros automáticos. Los productos que ofrecen los bancos cambian continuamente, y otro tanto tiene que reflejarse en el software de aplicación de los cajeros. La existencia de muchos modelos diferentes hace que haya que mantener sincronizadas muchas versiones de la misma funcionalidad.

El uso de la máquina abstracta del sistema integral en todos los cajeros elimina el problema de la heterogeneidad de la plataforma. En lugar de hacer múltiples versiones de la aplicación para cada modelo de cajero, se realiza una única versión para el sistema integral, válida para todos ellos. Se reduce el gasto en mantenimiento.

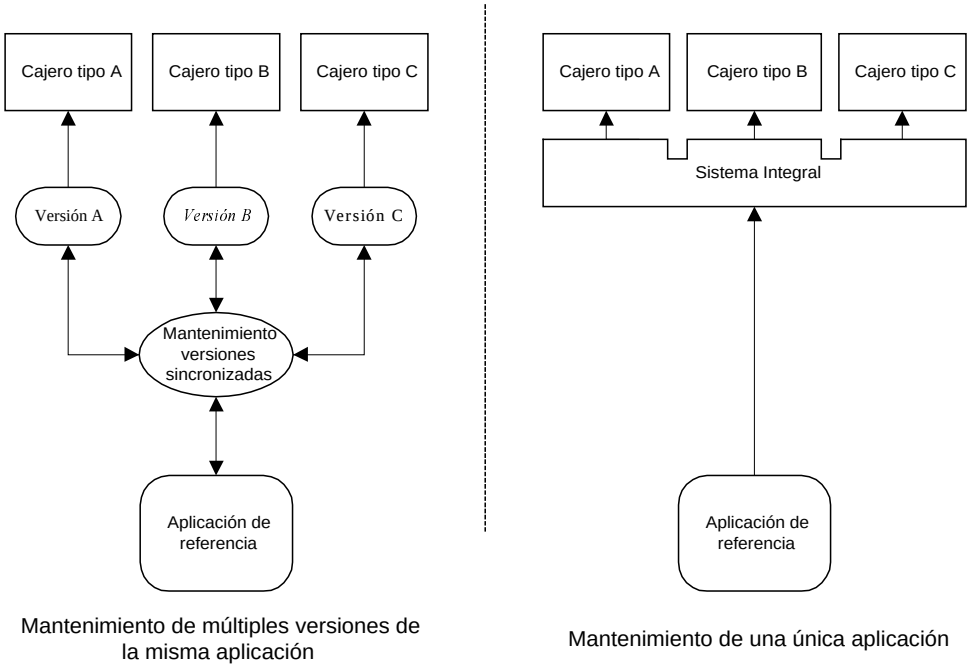


Figura 20.2 Mantenimiento de una única versión de las aplicaciones en el sistema integral en lugar de múltiples versiones.

20.2.2 Entorno único. Homogeneidad de desarrollo con el sistema central

Adoptando el propio sistema integral en el sistema central se consigue mayor productividad al homogeneizarse totalmente el entorno de desarrollo de aplicaciones, que será el mismo para los cajeros y el sistema central. Se puede tener una visión de la red como un sistema único en el que están todos los objetos. En ellos habrá una agrupación física de los mismos de acuerdo a su función: cajero o sistema central. Pero la programación e interoperación de los mismos es uniforme como si estuvieran en el mismo entorno de la misma máquina.

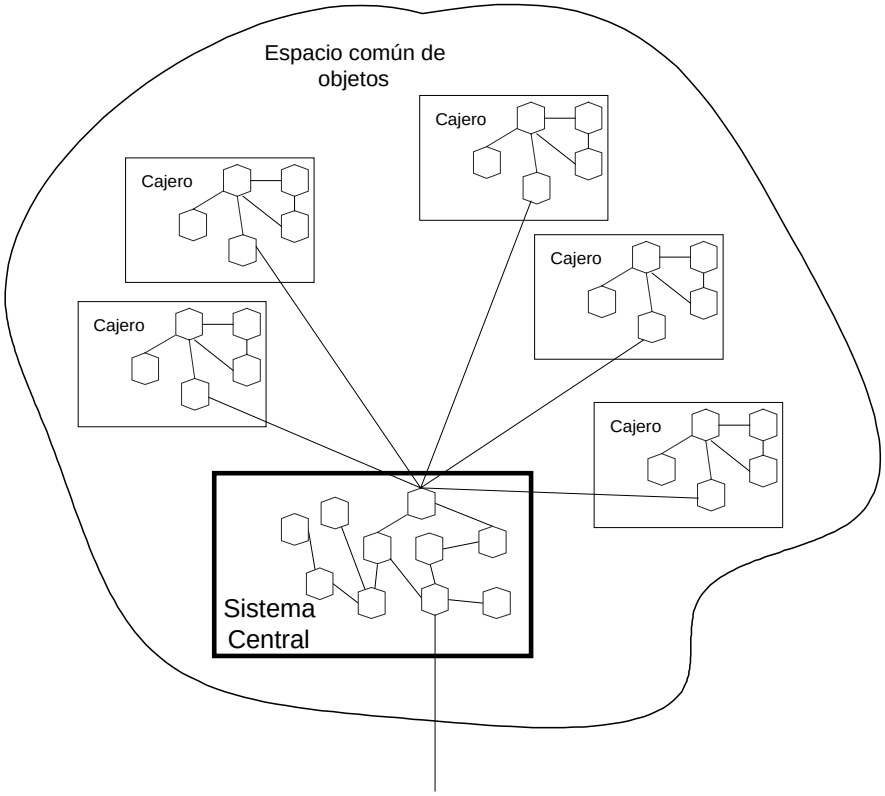


Figura 20.3 Homogeneidad de desarrollo convirtiendo conceptualmente la red en un sistema único de objetos.

20.2.3 Extensibilidad. Actualización incremental, control y monitorización remotos

Otro problema es la actualización de las aplicaciones de los cajeros. Normalmente esto requeriría acceder físicamente a cada cajero para actualizarlo con la versión particular de la aplicación, lo cual es muy costoso.

La inclusión del soporte de distribución en cada cajero permite aplicar la **actualización incremental**[1] de manera remota, así como controlar y monitorizar el sistema desde otro nodo de la red.

20.2.3.1 Monitorización y control remoto

El soporte de distribución permite acceder de manera uniforme a los objetos de cualquier máquina de la red como si estuvieran en el mismo espacio. Cualquier cosa que se puede hacer con la máquina local se puede hacer desde una máquina remota. Por ejemplo la ya

[1] En lugar de sustituir totalmente el software del cajero.

mencionada actualización de cualquier elemento del sistema. O la gestión del mismo desde otro nodo. Simplemente se llaman de manera normal a los métodos de los objetos del sistema empotrado desde la máquina remota.

No se necesitan mecanismos especiales para conocer y controlar el estado de un cajero. Simplemente se llaman a métodos de sus objetos que proporcionan esa funcionalidad. Por ejemplo, se podría desarrollar un objeto Monitor con métodos que devolvieran el estado de cada cajero: dinero restante, número de transacciones, etc.

De la misma manera se podría controlar el funcionamiento del cajero desde un lugar remoto centralizado: ponerlo fuera de servicio, activarlo, etc.

20.2.3.2 Actualización incremental remota

La actualización incremental permite que la actualización remota sea más segura que en el caso de una actualización total. Aunque esta actualización total pudiera hacerse de manera remota, existe el problema de que casi siempre[1] habría que rearrancar el cajero para que las actualizaciones surtieran efecto. Se corre el peligro de que la actualización no haya funcionado bien y esto haga perder el acceso remoto al sistema[2]. En el caso del sistema integral todos los elementos, incluso los del sistema se actualizan en espacio del usuario, con lo que no es necesario rearrancar el mismo. El peligro anterior se reduce.

20.2.4 Seguridad. Confidencialidad y protección del sistema por el mecanismo de protección uniforme

El inconveniente que tiene la flexibilidad de la actualización y el control remoto de los cajeros es la seguridad. Las operaciones descritas sólo deben poder ser efectuadas por elementos autorizados, como por ejemplo la activación y desactivación de un cajero. Además de frente a intrusos maliciosos, esto permite proteger el sistema frente a errores de programación que por equivocación llamaran a esas operaciones.

[1] Por ejemplo cuando se actualizase algún elemento del propio sistema.

[2] Como en el caso de una actualización de digamos, Windows, realizada a partir de un servidor de red que funciona mal y deja el sistema en mal estado y sin soporte de red.

20.2.4.1 Protección del sistema

No hay que desarrollar elementos especiales para garantizar esta seguridad. El sistema integral dispone de un mecanismo de control de acceso uniforme integrado en el modelo de objetos. Para garantizar la seguridad basta con indicar a cada objeto cuáles son los métodos de otro objeto a los que tiene acceso. Así, sólo se concedería permiso para acceder a las operaciones de activación y desactivación del objeto Monitor del cajero al objeto del sistema central responsable de la monitorización. De esta manera el sistema garantiza que el único que podrá activar y desactivar los cajeros será el sistema central y nunca cualquier otro objeto. Lo mismo se puede aplicar a los procedimientos de actualización remota, etc.

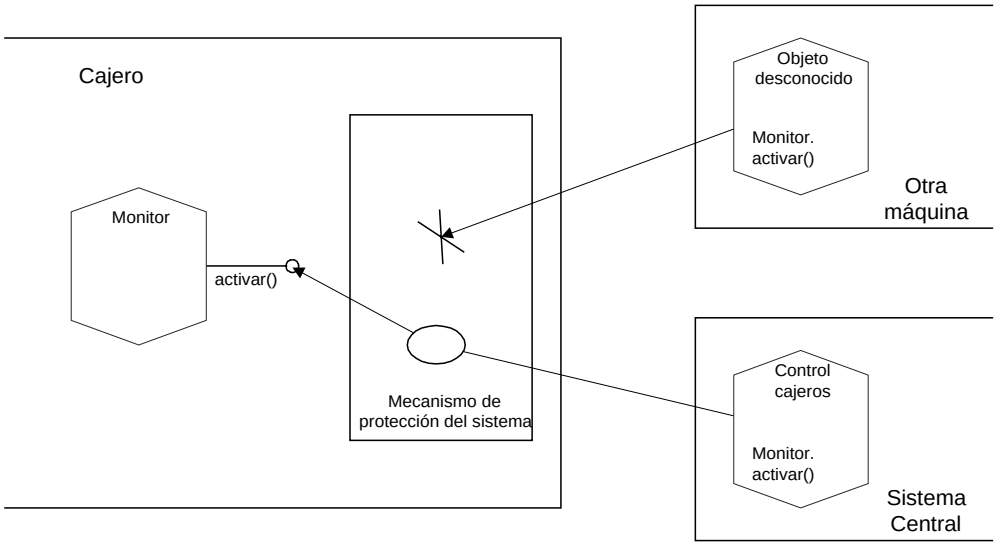


Figura 20.4 Seguridad en el sistema distribuido usando el mecanismo de protección uniforme del sistema integral.

El mecanismo de protección al llegar al nivel de métodos de objetos permite un ajuste fino del grado de autorización que tienen los objetos. Esto permite conceder sólo los permisos mínimos que necesite un objeto para realizar su tarea[1]. Así se minimiza la posibilidad de fallos y accesos no autorizados a otros elementos que nada tienen que ver con su función. Por ejemplo, una actualización del subsistema de monedero de un cajero sólo tendría permisos de acceso a los objetos de ese subsistema. Así nunca como resultado de la actualización del monedero se podrían producir cambios no controlados en otros elementos del sistema[2].

20.2.4.2 Confidencialidad

El mismo principio permite garantizar la confidencialidad en la interoperación del cajero con el sistema central. Sólo los objetos autorizados podrán acceder exactamente al conjunto de operaciones confidenciales que necesiten. Por ejemplo, el objeto registrador de transacciones del cajero sólo podrá trabajar con el objeto servidor de transacciones del sistema central.

[1] Siguiendo el principio del menor privilegio posible.

[2] Problema que padecen muchos sistemas operativos actuales en los que a veces, la instalación de un procesador de textos produce cambios (inadvertidos por el usuario) en elementos con tan poca relación en principio como el acceso a red.

20.3 Sistema de estaciones de trabajo en grupo para red

Se trata de un entorno compuesto por una serie de estaciones de trabajo conectadas en red, en la que a veces existe un servidor de uso común. Un ejemplo de este tipo de sistema es la red de ordenadores de una oficina, o de un grupo de trabajo de una empresa. El sistema integral puede utilizarse en estos ordenadores, con diferentes grados de flexibilidad, partiendo de la utilización más convencional. Se mencionan las posibilidades de aplicación del sistema para mejorar el uso del sistema en ciertas áreas. En todos los casos, el uso del sistema integral permite aprovechar las ventajas de la uniformidad en la OO, persistencia transparente, etc.

20.3.1 Mantenimiento centralizado del software de las estaciones de trabajo

En una utilización convencional, cada estación de trabajo almacena completamente todo su software de aplicación y de sistema[1]. Uno de los problemas que más gasto produce en esta configuración es tener actualizado el software de cada máquina, y controlar que no se instale software no autorizado en los ordenadores. Esto requiere muchas veces un largo y problemático proceso de instalación manual y verificación de cada puesto.

Este es un caso similar al de la actualización de software de los cajeros de un banco. Como se ha visto, el sistema permite una fácil y segura actualización remota incremental del software de aplicación de cada ordenador y de los propios objetos del sistema. El mecanismo de protección del software permite controlar exactamente quién y qué puede hacer en el sistema. Para impedir instalación de software no controlado, basta con permitir al usuario sólo acceso a las aplicaciones y no a los elementos del sistema que permiten la actualización. Estos objetos sólo podrán ser accedidos por objetos (remotamente) del administrador del sistema.

En el caso en que las aplicaciones se carguen de un servidor central común, estas ventajas se aplican al propio software del sistema que está en cada máquina y al impedimento de instalaciones locales no autorizadas.

20.3.2 Entorno de trabajo de usuario igual en cualquier estación de la red: ordenador de red (NC, Network Computer)

En el entorno anterior cada estación de trabajo puede tener un software diferente, al menos de sistema, con lo que cada ordenador queda personalizado para un único usuario, restringiendo su uso para otros usuarios.

Un paso más es lograr un entorno en el que cualquier usuario pueda trabajar de la misma manera (con su propio entorno personalizado) en cualquier ordenador de la red. Para ello se debería reducir el software de sistema local en cada máquina al mínimo común posible y cargar de la red (del servidor) el entorno de trabajo del usuario concreto que se conectase en la máquina. Este tipo de estaciones de trabajo se suelen denominar ordenadores de red.

El entorno anterior se puede conseguir con el sistema integral por una combinación de sus características y una implementación determinada del soporte de persistencia y distribución. Por un lado la portabilidad del sistema hace que el tipo concreto de cualquier ordenador no importe al usuario, todos serán máquinas con el mismo sistema integral. En cada máquina se dejaría residente de manera local los elementos mínimos del sistema integral: los objetos de la máquina que dan el soporte básico que se necesite, la distribución y la persistencia.

[1] En un entorno normal, un sistema operativo para PC como Windows, procesadores de texto, hojas de cálculo, etc.

20.3.2.1 Política de movilidad de objetos en la distribución: bajo demanda a la máquina local

El entorno de un usuario (sus objetos) residirán en un servidor de la red. Al conectarse a un ordenador, el sistema transporta a la máquina local en cuestión los objetos iniciales (raíz) del entorno del usuario, en lugar de transportar totalmente todos sus objetos[1]. A medida que el usuario vaya manipulando a esos objetos, irá haciendo referencia a otros objetos. La distribución transparente del sistema hace que no importe la localización de estos objetos, que residirán en el servidor. La política de funcionamiento del sistema de distribución puede ser simplemente mover[2] estos objetos a la máquina local que origina la solicitud. Es decir, los objetos se van trasladando bajo demanda a la máquina local.

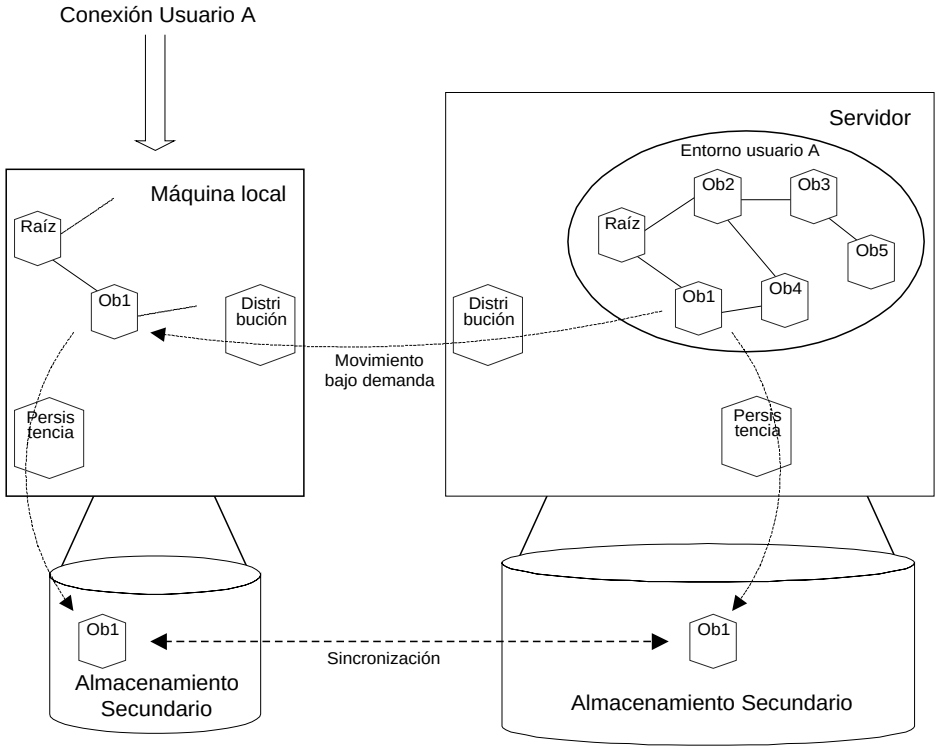


Figura 20.5 Movilidad de objetos bajo demanda para un ordenador de red.

20.3.2.2 Sistema de persistencia

El usuario trabajará de manera normal (persistente) con estos objetos locales. Durante su trabajo, el sistema de persistencia, además de mantener de manera transparente el estado persistente de estos objetos en la máquina local, puede ir sincronizando estos objetos con su copia "maestra" en el servidor, para una mayor seguridad. Al finalizar la sesión, se actualizan todos los objetos a las copias "maestras" que residen en el servidor.

[1] Lo cual sería muy costoso en tiempo y en espacio. Además, es improbable que en una sesión de trabajo accediese a todos sus objetos.
[2] Por seguridad, se haría una copia del objeto.

Otra opción es que el almacenamiento secundario del sistema de persistencia, en lugar de ser en un disco local, sea el propio servidor. Esto evita la existencia de copias de objetos y la sincronización de los mismos. Las estaciones de trabajo no necesitarían ningún tipo de almacenamiento secundario.

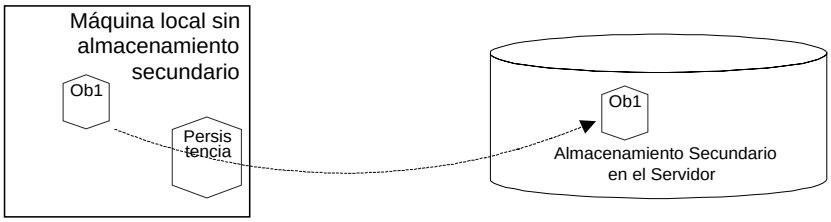


Figura 20.6 Sistema de persistencia para un ordenador de red sin almacenamiento local.

Todas las restricciones de seguridad entre los objetos del sistema integral, distribución, etc. son globales en el sistema (por el uso de un identificador único). Esto hace que la máquina en que funcione un objeto sea indiferente y que se pueda usar cualquier ordenador. Todos son equivalentes.

El único elemento que queda repartido en cada máquina individual es el software mínimo del sistema. Sobre éste también se puede hacer actualización remota del mismo, bien explícitamente desde una administración central, o bien automáticamente en el mismo momento de la conexión de un usuario.

20.3.3 Sistema con computación distribuida entre la estación local y el servidor

En el caso anterior se van moviendo los objetos del entorno del usuario a la máquina local. La distribución transparente en el sistema permite utilizar otras políticas de movilidad de objetos. Por ejemplo, en lugar de mover los objetos a la máquina local se pueden dejar simplemente en el servidor. La invocación de métodos se realizará de la misma manera por la distribución transparente. Ahora el objeto residirá en el servidor y el método se ejecutará allí, devolviéndose el resultado al objeto inicial.

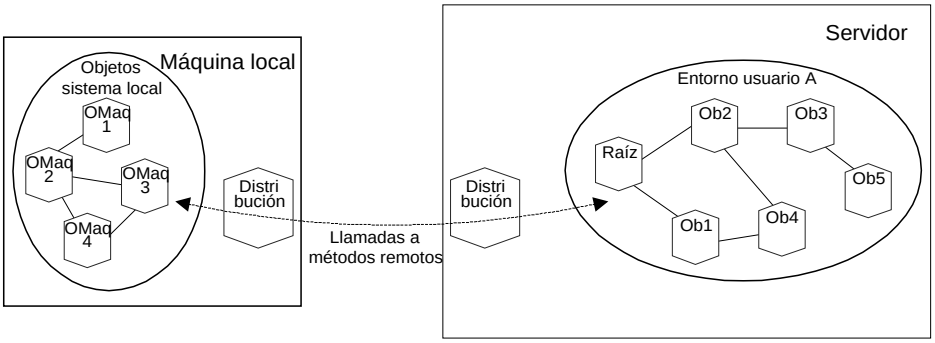


Figura 20.7 Distribución de la computación entre el ordenador de red local y el servidor.

Esto permite utilizar estaciones de trabajo con menos potencia que anteriormente (y un servidor más potente). Normalmente en la estación residirán los objetos básicos del sistema y de la gestión de los periféricos y presentación en pantalla. El resto de los objetos funcionarán en el servidor. En función de la potencia de la estación se podría utilizar una política de movilidad de objetos diferente (equilibrio de carga), moviéndolos a la estación local hasta que esta alcance una carga determinada y a partir de entonces dejándolos en el servidor.

Es decir, pueden diseñarse diferentes políticas de equilibrio de carga para repartir el trabajo dinámicamente entre la estación de trabajo y el servidor. La distribución transparente hace que se pueda mover un objeto a cualquier ordenador sin afectar a otros objetos que lo usen. Por ejemplo, en lugar de un servidor, se pueden usar varios servidores que se repartan la carga de manera equilibrada entre ellos.

20.3.4 Sistema distribuido: ordenador virtual único formado por todos los ordenadores de la red

El siguiente paso en esta evolución es lograr un verdadero sistema distribuido, que trate todos los recursos de la red como recursos globales, sin hacer distinción entre elementos locales y remotos [Gos91].

Este es el objetivo inicial del sistema integral de objetos: lograr un espacio conceptual único de objetos en el que estos colaboran entre sí por medio de la invocación de métodos. Este espacio común estará formado por los espacios de objetos de cada una de las máquinas del sistema. El sistema de distribución se ocupa de que cualquier objeto pueda invocar de manera transparente un método de cualquier otro objeto, independientemente de la máquina en la que resida.

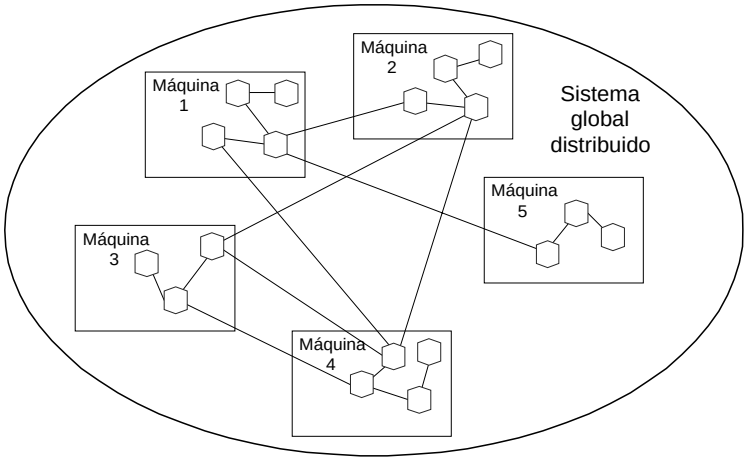


Figura 20.8 Sistema global distribuido.

En este caso, en lugar de implementar una política de equilibrio de carga y movilidad de objetos que los reparta entre una máquina local y el servidor, esta será global. El sistema podrá mover los objetos según considere necesario entre los ordenadores del sistema para equilibrar la carga y aprovechar los recursos de toda la red[1].

El usuario trabaja con sus objetos de manera normal desde cualquier ordenador de la red, pero la localización real de cada objeto y el lugar donde se respalda su estado persistente está totalmente bajo control del sistema. Los objetos del usuario pueden estar en cualquier ordenador de la red.

Todos los ordenadores de la red ofrecen la impresión de un único ordenador (sistema integral orientado a objetos) al que se conectan los usuarios a través de un punto de acceso cualquiera (estación de trabajo) para realizar su trabajo, por medio de la manipulación de objetos.

[1] Siempre de manera que evite la pérdida de trabajo, por ejemplo por fallo de algún nodo.

20.4 Sistema operativo de nueva generación para el Web

El sistema integral reúne muchas de las cualidades que algunos autores proponen como necesarias para los sistemas operativos del futuro. Franklin D. Reynolds describe algunas de las características que necesitará un sistema operativo para la era Web [Rey96]:

- **Extensibilidad del núcleo**. Que permita que las aplicaciones de usuario especifiquen sus propias políticas de gestión de los servicios del sistema, o incorporen unos nuevos más apropiados para cada aplicación concreta.
- **Portabilidad y código móvil**. Que permita que aplicaciones obtenidas de la red se instalen sobre la marcha (*deploy on-demand*) en el momento en que son necesitadas. Para ello es fundamental que el código de las aplicaciones sea independiente del procesador y móvil.
- **Seguridad. Principio del menor privilegio**. La instalación de aplicaciones de la red, cuyo origen y su buen funcionamiento no se puede asegurar implica un problema de seguridad. Es necesario proporcionar un mecanismo que permita a estas aplicaciones hacer su trabajo pero evitando comprometer el sistema. Un mecanismo que permita concederles el grado mínimo de privilegio necesario para hacer su trabajo. Por ejemplo, un editor de un tipo de documento determinado instalado sobre la marcha desde la red, sólo tendría permiso de acceso al documento a editar, que es lo mínimo que necesita para funcionar. Reynolds incluso propone usar capacidades para esto.

El sistema integral posee todas estas propiedades, y han sido comentadas de una u otra forma en la aplicación a los distintos sistemas descritos anteriormente. El sistema está pensado para la extensibilidad en el espacio de usuario, que permite por la reflectividad colaborar a las aplicaciones con los objetos del sistema e instalar nuevos servicios como objetos normales. La máquina abstracta garantiza la portabilidad y los objetos son totalmente móviles al referenciarse de manera uniforme mediante un identificador global en el sistema. El sistema de seguridad está integrado en el principio en el modelo del sistema. Precisamente está basado en el uso de capacidades que le permiten controlar con el nivel de detalle más fino posible el acceso a los objetos, lo que permite utilizar políticas de seguridad que sigan el principio del menor privilegio posible.

Por tanto podría decirse que el sistema integral es incluso el sistema operativo de nueva generación para el Web que proponen otros investigadores como necesidad de evolución futura de los sistemas operativos para este nuevo entorno de aplicación.

20.5 Resumen

Las características del sistema integral orientado a objetos permiten aplicarlo en entornos muy diferentes: desde un sistema empotrado hasta un verdadero sistema distribuido formado por todos los ordenadores de una red.

En cada una de estas aplicaciones se aprovecha siempre las ventajas de la filosofía básica del sistema integral: portabilidad y heterogeneidad, uniformidad en la OO con un modelo de desarrollo único, persistencia y distribución transparente, etc. En cada caso se han comentado de qué forma se pueden aplicar las propiedades del sistema integral para solucionar alguno de los problemas de estos entornos.

Por ejemplo, la heterogeneidad evita tener que desarrollar versiones diferentes de aplicaciones para cada tipo de plataforma hardware, como para cada sistema empotrado o tipo de cajero de un banco. También permite usar cualquier tipo de ordenador como estación de

trabajo. La flexibilidad del sistema permite configurar sistemas particularizados para las necesidades concretas de un sistema empotrado. La extensibilidad permite ampliar fácilmente el sistema si estas aumentan. Esto también permite la actualización remota del software de una máquina, por ejemplo en el caso de la red de cajeros de un banco, siempre controlado de manera fina por el mecanismo de seguridad uniforme del sistema.

Además de los mecanismos básicos de distribución y persistencia, diferentes políticas de aplicación de las mismas permiten desarrollar entornos de trabajo en red cada vez más flexibles, hasta llegar a conformar un ordenador virtual único. El sistema integral ofrece al usuario un espacio de trabajo de objetos formado por la unión transparente de todos los recursos de los ordenadores de la red, gestionados totalmente por el sistema.

Estas características hacen que el sistema incluso disponga ya de las propiedades que algunos investigadores indican como necesarias para los futuros sistemas operativos de nueva generación para el Web.

Capítulo 21

APLICACIÓN DE RESULTADOS A OTROS SISTEMAS Y TRABAJO RELACIONADO

En este capítulo se hace mención de algunos proyectos relacionados con diferentes aspectos del sistema integral. También se comenta brevemente las posibilidades de aplicación de manera individual de los resultados más importantes del trabajo para mejorar otros sistemas existentes.

21.1 Trabajo relacionado

En los siguientes apartados se mencionan otros trabajos relacionados en el campo de los sistemas integrales y la uniformidad en la orientación a objetos y de la flexibilidad y sistemas extensibles.

21.1.1 Sistemas integrales y uniformidad en la OO

Existen otros proyectos con el ánimo de lograr un sistema integral orientado a objetos que comparten algunos elementos comunes con el sistema integral descrito en este trabajo. La mayoría son proyectos recientes desarrollados de manera colaborativa a través de la red Internet, aunque no han alcanzado en general un estado muy avanzado aún. Por ejemplo, Tunes y Merlin son proyectos que también basan su estructura en la OO y una arquitectura reflectiva.

Merlin [MER97] es un sistema desarrollado en la Universidad de Sao Paulo en Brasil. Está basado en una arquitectura reflectiva mediante el lenguaje basado en objetos Self, aunque el modelo de este lenguaje se basa en prototipos y no en la noción más extendida de clases y herencia. Las aproximaciones a la persistencia y la distribución son diferentes y con menos transparencia que en el sistema integral.

Tunes [TUN97] es un proyecto colaborativo para desarrollar un sistema de computación moderno basado en la uniformidad en torno a la OO y con una arquitectura reflectiva. Aunque está en un estado preliminar, parece que la estructura se decanta por el desarrollo de un nuevo lenguaje uniforme para la programación del sistema.

Otro proyecto relacionado es el proyecto LispOS [LIS97]. También mediante desarrollo colaborativo por Internet pretende desarrollar un sistema integral (LispOS, sistema operativo para Lisp), usando como paradigma el modelo del lenguaje Lisp en lugar de la OO. El objetivo es construir sobre sistemas heterogéneos actuales el entorno de las máquinas L desarrolladas con hardware especial en los años 80. El sistema LispOS inicialmente será también un sistema de espacio de direcciones único y reflectivo, con un mecanismo de protección de grano fino integrado en el sistema[1] [Hug97], si bien aún está en una fase muy preliminar de diseño. Para ello también desarrollarán una máquina abstracta, LispVM, que soporte el paradigma de procesos de listas del Lisp.

[1] Aunque la manera de integrarlo está aún sin determinar.

Quizás el primer proyecto con objetivo de la uniformidad en un modelo de objetos fue el soportado a comienzos de los años 80 por el procesador iAPX432 [Int81] revisado en el capítulo 10. Aunque basado en el modelo del lenguaje ADA, proponía un entorno unificado en el que el sistema operativo iMAX [KCD+81] extendía el procesador basado en objetos iAPX432 ofreciendo una visión unificada con objetos ADA para el desarrollo de aplicaciones.

Legion [SW97] es un sistema que utiliza un modelo único de objetos para desarrollar aplicaciones de supercomputación distribuidas entre sistemas heterogéneos geográficamente dispersos, aunque para ello se basa en la adición de capas de software adicional sobre los sistemas operativos ya existentes en los nodos del sistema.

21.1.2 Flexibilidad y sistemas extensibles

Aunque no necesariamente con los mismos objetivos de uniformidad que el sistema integral orientado a objetos, existen otros sistemas que exploran la flexibilidad en el software de sistemas y su extensibilidad al igual que el sistema integral.

Por ejemplo el sistema SPIN [BSP+95] descrito en el capítulo 4 permite la adición al núcleo convencional del sistema de extensiones, codificadas con un lenguaje especial, Modula3, para garantizar la seguridad. Otros sistemas como Exokernel [EKO95] permiten la construcción de los elementos del sistema operativo en el espacio del usuario, aunque no especifica un modelo que permita colaborar entre las extensiones. Apertos [Yok92], analizado igualmente en el capítulo 4 también utiliza la reflectividad y la OO para extender la funcionalidad del sistema, aunque separa el espacio de objetos del de meta-objetos.

21.2 Aplicación de resultados a otros sistemas

El uso de los diferentes aspectos descritos en este trabajo, como una máquina abstracta con arquitectura reflectiva, un modelo único de objetos con protección uniforme, persistencia, etc. dan una gran flexibilidad y potencia al sistema integral. Es la conjunción de estos elementos y la manera uniforme de combinarlos dentro del marco de referencia de la orientación a objetos la que consigue esa flexibilidad y potencia. Sin embargo, también pueden aplicarse de una manera más individualizada a otros sistemas, para mejorar el comportamiento de los mismos en esas áreas, especialmente para mejorar las propiedades de sistemas comerciales.

Uno de los elementos más importantes de la arquitectura del sistema integral para lograr los objetivos de portabilidad y uniformidad es la utilización de una máquina abstracta orientada a objetos. Los sistemas que ya la utilicen estarán ya más cercanos a las ventajas del sistema integral. El sistema comercial más difundido que utiliza una máquina abstracta orientada a objetos es la plataforma Java [KJS96]. Como ejemplo de aplicación de los aspectos constituyentes del sistema integral a un sistema comercial, a continuación se describirá brevemente líneas de aplicación de algunos de estos aspectos para mejorar las propiedades de la plataforma Java.

21.2.1 Aplicación a la plataforma Java

En cualquier caso la aplicación de estos elementos a esta plataforma debe ser sin imponer excesivas modificaciones en la misma. Aprovechar totalmente las ventajas del sistema integral supondría en algunos casos tales modificaciones que resultaría en un sistema distinto. El elemento fundamental de la plataforma es la arquitectura y la definición de las instrucciones de la máquina virtual de Java [LY97], por tanto, los cambios en este elemento básico tendrán que ser mínimos. Esto ya hace que no se puedan aprovechar las ventajas de un alto nivel en la interfaz de la máquina, o de un número reducido de instrucciones.

Las propiedades de la plataforma Java residen fundamentalmente en su máquina virtual, y es en ella donde se concentrarán las mejoras a realizar aplicando elementos del sistema integral, teniendo en cuenta las ventajas descritas por su uso en el sistema integral.

21.2.1.1 Uso de técnicas de implementación de clases primitivas para mejorar el soporte a otros lenguajes y el rendimiento

La difusión de la máquina de Java hace que se plantee la posibilidad de hacer que los compiladores de lenguajes diferentes a Java generen código para esta máquina. Sin embargo, la máquina de Java está demasiado adaptada al lenguaje Java, con lo que la implementación de otros lenguajes sufre de problemas de rendimiento, al no existir una correspondencia tan directa entre sus estructuras y la máquina de Java [Shi96].

Un ejemplo de esto es la dificultad para representar la herencia múltiple de implementación de lenguajes como C++, puesto que la máquina de Java sólo soporta directamente herencia simple. Otro caso es la dicotomía entre tipos básicos (para los cuales hay una especial representación y soporte eficiente) y objetos en general (con menos eficiencia). Lenguajes que sólo manipulen objetos, como Smalltalk pagan una penalización alta de eficiencia al usar la máquina de Java.

Una solución a este último problema puede ser la utilización de las técnicas de implementación primitiva de clases explicadas en el capítulo 13. Tendría que acometerse el estudio de cómo implementar las clases de los objetos básicos de estos lenguajes de manera primitiva sin romper el tratamiento de los mismos como objetos normales dentro de la máquina de Java. De esta manera se conseguiría una mayor eficiencia en el tratamiento de estos objetos pero manteniendo a la vez la uniformidad de utilización de los mismos.

21.2.1.2 Incorporación de la reflectividad en la máquina abstracta para permitir la extensión de la máquina de Java

La máquina virtual de Java tal y cómo se estructura en la actualidad, es una caja negra con implementación monolítica. No se puede influir en nada en su composición ni en su funcionamiento interno, ni extender la funcionalidad de la misma. Hay que aceptar exactamente la funcionalidad que incluye la implementación y la manera de proporcionarla. Por ejemplo, el soporte para planificación de hilos está solidariamente unido a la implementación de la máquina, de forma que sólo puede utilizarse la política de planificación incluida en la implementación. No se puede ni cambiar, ni seleccionar otra política, ni eliminar fácilmente este soporte si el sistema va a ser usado únicamente en forma secuencial.

En el capítulo 14 se mostró como la reflectividad es una técnica que permite precisamente modularizar y abrir la implementación describiéndola en función de un conjunto de objetos que sean posibles de usar y manipular por los objetos de usuario, y así ser extendida en el espacio de usuario. Estas características son muy importantes para que un sistema de computación de soporte a los entornos del futuro (véase el capítulo 20).

Para dotar de estas cualidades a la máquina de Java, es necesario decidir exactamente qué objetos compondrán el conjunto de objetos de la máquina que se van a reflejar a los objetos de usuario. También es necesario decidir la meta-interfaz de la máquina, es decir, el conjunto de métodos que ofrecerán al usuario estos objetos de la máquina de Java (meta-interfaz de los meta-objetos de la máquina). Para implementar la reflectividad, puede usarse también la técnica de implementación de clases primitivas descrita en el capítulo 13, como se explica en el capítulo 14.

Ya existen incluso proyectos que experimentan con la introducción de reflectividad en la máquina de Java1. MetaJava [GK97] incorpora una meta-interfaz en la máquina de Java que permite la reflectividad, aunque la implementación de la misma no es totalmente compatible con la máquina Java actual. MetaJava introduce instrucciones adicionales y necesita reescribir los *bytecodes* de los métodos para las operaciones de llamada a los elementos reflectivos (cosificación).

En cualquier caso, se debería estudiar con detenimiento todos los aspectos mencionados anteriormente para incorporar la reflectividad a la máquina de Java. Una vez iniciado el proceso, de la misma manera que se sigue la estandarización de las APIs de Java, se estandarizaría el modelo de objetos reflejado de la máquina y su meta-interfaz. De esta manera la capacidad de extensión de la máquina alcanzada con la reflectividad podría ser usada con garantía de portabilidad al igual que cualquier otra de las APIs estandarizadas.

Con esto se incorporarían a Java las ventajas del uso de la reflectividad reseñadas en el capítulo 14, lo que permitiría, por ejemplo, cambiar la política de planificación, escribir en espacio de usuario diferentes políticas de planificación, escribir extensiones que colaboraran con la máquina para implementar la persistencia de manera transparente, etc.

21.2.1.3 Mecanismo de control uniforme de grano fino para flexibilizar la seguridad en el sistema

La máquina de Java no tiene ninguna previsión para un mecanismo de control uniforme de grano fino[2]. Esto es la causa de la falta de flexibilidad de la seguridad en la plataforma Java, especialmente en el caso de nuevo código descargado a través de la red (véase el capítulo 19). Al no existir un control de grano fino en el acceso a los métodos de un objeto, las políticas de seguridad son demasiado poco flexibles. Básicamente se traducen en un acceso total a los recursos (objetos) del sistema para las aplicaciones (objetos) locales y en la prohibición de acceso excepto a un conjunto reducido de recursos de las aplicaciones descargadas de la red. No es posible establecer un control más detallado, por ejemplo indicar para una aplicación de listado de ficheros descargada de la red que puede acceder al método leer de un objeto fichero, pero nunca al de escribir. De esta manera se evita que la aplicación produzca daños en el sistema (en caso de ser malintencionada), y a la vez se puede aprovechar su funcionalidad completa.

Al igual que en el sistema integral, se podría incorporar un mecanismo de protección de grano fino que permitiese el control de acceso al nivel de métodos individuales de los objetos. Así las políticas de seguridad podrían ajustarse exactamente a las necesidades de cada objeto, en lugar de agruparlos simplemente en dos categorías: seguros e inseguros.

Este mecanismo debería integrarse de manera uniforme dentro del modelo de objetos de la máquina. Como en el sistema integral, parece que el mecanismo más adecuado por lo fluidamente que encaja en el modelo de objetos son las capacidades [DH66] (véase la parte de seguridad del sistema operativo en el capítulo 15). Puede usarse también el mismo procedimiento para la implementación de la misma sin necesitar muchas modificaciones en las aplicaciones de usuario ya existentes o en la interfaz de la máquina. Es decir, añadir a las referencias a objetos que usa la máquina de Java la información de protección necesaria para el control del acceso.

---

[1] Se está hablando de reflectividad de comportamiento en la máquina virtual. El lenguaje Java ya dispone de un tipo de reflectividad estructural para los elementos del lenguaje mediante *Java Core Reflection* [Sun97].

[2] Esto no es de extrañar puesto que la máquina de Java nació en esencia para dar soporte a un lenguaje. Dentro de una aplicación escrita en un lenguaje la necesidad de un mecanismo de control de acceso uniforme no es tan importante como en un sistema de computación completo.

Un ejemplo de proyecto que también utiliza este tipo de aproximación es SLK (*Safe Language Kernel*, Núcleo de lenguaje seguro). Este proyecto [HCC+97] propone precisamente utilizar las capacidades en Java para permitir un control de grano fino. En lugar de usar directamente las referencias de objetos para implementar las capacidades, proporciona la funcionalidad indirectamente mediante envoltorios de objetos que son usados para acceder a los objetos reales. En estos envoltorios se hacen las comprobaciones de acceso al nivel de cada método. Otros investigadores también proponen aplicar su modelo de capacidades a la plataforma Java [HRM+96, HKM+96].

Dotando a la plataforma Java de un mecanismo de control uniforme como este, sería más adecuada para dar soporte a las necesidades de un sistema de computación completo, como la compartición de recursos, colaboración entre objetos de manera segura y flexible, etc.

21.2.1.4 Propiedades de persistencia y distribución transparentes para aumentar el nivel de abstracción

La inclusión de la propiedad de persistencia como parte de las propiedades de los objetos de la máquina ofrece un nivel de abstracción mayor a los usuarios y programadores del sistema, liberándolos de la necesidad de preocuparse por almacenar y recuperar explícitamente los objetos (véase el capítulo 16). La aplicación de manera transparente de esta propiedad en la plataforma Java presentaría una interfaz más intuitiva a los usuarios y aumentaría la productividad del sistema.

Una manera de implementar la persistencia de manera transparente es utilizar la técnica mostrada en el capítulo 18 para la implementación del prototipo de persistencia del sistema integral. Haciendo persistentes los objetos de la propia implementación de la máquina abstracta se hacen persistentes de manera indirecta los objetos de usuario, y sin cambios en la utilización de los mismos. Sin embargo, la existencia de elementos de bajo nivel en la interfaz de la máquina (máquina de pila) puede hacer más difícil de llevar a cabo esta técnica, y posiblemente el rendimiento no fuera muy bueno. En cualquier caso podrían usarse otros mecanismos, usando la reflectividad (si estuviera incorporada) para escribir los sistemas de persistencia en el espacio del usuario (véase el capítulo 16).

El campo de la incorporación de persistencia a Java es un campo de mucha actividad en el que se han desarrollado muchas propuestas. Algunas tienen la propiedad de la transparencia, como el sistema PJama [AJD+96]. Este sistema proporciona persistencia mediante accesibilidad a partir de una raíz dada (una instancia de la clase PJavaStore). Su arquitectura es similar a la del prototipo de persistencia del sistema integral, aunque trabajando en el espacio de objetos de usuario: un almacenamiento persistente con una memoria intermedia (caché de objetos) gestionada mediante políticas de emplazamiento y reemplazamiento.

En cualquier caso y al igual que para la reflectividad, seguridad, etc. debería llegarse a un consenso y estandarizar el sistema de persistencia de la plataforma para garantizar la portabilidad.

El caso de la distribución es similar, aunque el problema es que para conseguir una distribución totalmente transparente los identificadores globales de objetos del sistema integral facilitan mucho la labor. Este problema debe estudiarse con atención, ya que los identificadores de objetos en la máquina de Java sólo son válidos dentro de la misma. La introducción de identificadores globales podría romper demasiado el esquema actual de la máquina.

21.2.1.5 Entorno de computación completo

La plataforma Java tiene muchas propiedades interesantes. Sin embargo, en algunos aspectos carece de flexibilidad y elementos suficientes para dar soporte a las necesidades de un sistema de computación completo. Estos problemas se pueden mejorar aplicando algunos elementos desarrollados en el sistema integral a la plataforma, en áreas como el soporte para otros lenguajes mediante la técnica de implementación de clases primitivas, inclusión de la reflectividad de la misma manera para facilitar la extensibilidad, capacidades como mecanismo de protección uniforme para una seguridad más flexible y persistencia y distribución transparente. Algunos investigadores también trabajan en la misma línea, aplicando alguno de estos aspectos por separado en la plataforma.

La incorporación de todas estas características permite que la plataforma Java esté mejor preparada para soportar un entorno de computación completo, acercándose más a las ventajas de un sistema integral orientado a objetos total.

21.3 Resumen

Existen algunos proyectos de investigación que comparten alguno o muchos de los objetivos del sistema integral orientado a objetos. En el aspecto de aplicación uniforme de un paradigma con arquitecturas reflectivas los proyectos como Tunes o LispOS son recientes y basados en desarrollos colaborativos a través de Internet. Otros proyectos relacionados con la flexibilidad son los sistemas extensibles como SPIN o Exokernel.

Las ventajas del sistema integral se derivan de la aplicación conjunta y combinada, y de manera uniforme dentro del paradigma de la orientación a objetos, de una serie de elementos. Cada elemento aporta unas características deseables al sistema, que en conjunto le dan su potencia y flexibilidad. Estas ideas desarrolladas para el sistema integral también pueden aplicarse a otros sistemas, especialmente a plataformas comerciales como Java que ya tienen algunos puntos en común con el sistema integral.

De esta manera se pueden transferir algunas propiedades del sistema integral a una plataforma comercial, mejorándola para dar mejor soporte a un sistema de computación completo más cercano al ideal de sistema integral de objetos descrito en este trabajo.

Capítulo 22
CONCLUSIONES

22.1 Sistema Integral Orientado a Objetos

Este trabajo comenzó con la exposición de los problemas que presenta la incorporación de las tecnologías orientadas a objetos a los sistemas actuales. Estos sistemas no permiten explotar totalmente el paradigma de la orientación a objetos, introduciendo capas de software adicionales que intentan aliviar el problema de la desadaptación de impedancias y la interoperabilidad entre objetos. En lugar de continuar añadiendo parches a sistemas no diseñados para la orientación a objetos, se propuso la construcción de un sistema integral orientado a objetos, especialmente diseñado para dar soporte directo y aprovechar en todos los elementos del sistema el mismo paradigma de la orientación a objetos. El proyecto Oviedo3 es un esfuerzo de investigación en la Universidad de Oviedo para construir un sistema integral y desarrollar la investigación en diferentes áreas de las tecnologías de objetos sobre él.

El resto del trabajo se dedicó a verificar que la construcción de este sistema es posible, estableciendo los objetivos del mismo, diseñando una arquitectura que los posibilite, desarrollando en más detalle los elementos de la arquitectura y realizando la implementación de prototipos de las partes básicas de la misma. Además se presentaron ejemplos de las propiedades del sistema y posibles aplicaciones. El resultado es un sistema integral orientado a objetos, un entorno de computación completamente basado en el paradigma de la orientación a objetos, más flexible, coherente, intuitivo y fácil de usar.

En el capítulo 3 se enumeraron los requisitos que debería cumplir un sistema integral orientado a objetos, que se definió como "un entorno de computación que crea un **mundo de objetos**: *un único espacio de objetos, virtualmente infinito, en el que un conjunto de objetos homogéneos coopera intercambiando mensajes independientemente de su localización, y donde los objetos residen indefinidamente hasta que ya no son necesitados"*. Se identificaron los requisitos que debe cumplir un sistema moderno como este: uniformidad conceptual en torno a la orientación a objetos, transparencia, heterogeneidad y portabilidad, seguridad, concurrencia, multilenguaje / interoperabilidad y flexibilidad.

En el capítulo 4 se revisaron diferentes sistemas operativos, para pasar a diseñar en el capítulo 5 una arquitectura para conseguir los objetivos del sistema integral. El capítulo 6 se dedicó a presentar el proyecto Oviedo3, que pretende desarrollar un sistema integral con esta arquitectura como plataforma de de investigación y docencia en Tecnologías Orientadas a Objetos. En los siguientes capítulos se especificaron los diferentes elementos de la arquitectura, mostrando cómo ayudaban a conseguir los objetivos del sistema:

- Un **modelo de objetos único** para todo el sistema con las características de los modelos utilizados en las metodologías de análisis y diseño más extendidas (capítulos 7 y 8)

- Una **máquina abstracta** que implementa el modelo y da portabilidad al sistema, diseñada incorporando las ventajas y solucionado inconvenientes de otras máquinas ya existentes (capítulo 10) cuya arquitectura de referencia se describe en el capítulo 11.
- La incorporación de la **reflectividad** en la máquina abstracta para introducir más flexibilidad en el sistema y permitir las ventajas de la extensión dinámica en el espacio del usuario (capítulo 14).
- Un **sistema operativo** que extiende la máquina abstracta dotando al sistema de manera transparente de las importantes propiedades de seguridad, persistencia, distribución y concurrencia (capítulo 15), que por su importancia están siendo desarrolladas en trabajos específicos.

En el capítulo 12 se desarrolló completamente la especificación de la máquina abstracta Carbayonia y su lenguaje Carbayón como máquina base para el sistema. La implementación de un prototipo de la máquina se describió en el capítulo 13.

Como ejemplo de la extensión de la máquina, se desarrolló con más detalle el diseño del soporte para persistencia del sistema integral y algunos aspectos adicionales en los capítulos 16 y 17. Para comprobar las ventajas de utilización de la misma y validar la aproximación se implementó un prototipo de la misma sobre la máquina abstracta, tal y como se explica en el capítulo 18.

Para mostrar las propiedades del sistema integral, derivadas de la combinación de los elementos del diseño global de la arquitectura, se enumeraron algunos ejemplos de cómo se pueden lograr diferentes tipos de flexibilidad estática y dinámica en el capítulo 19. También se comprueban las ventajas del sistema con algunas aplicaciones del mismo en entornos muy diferentes, desde un sistema empotrado, una red de cajeros automáticos de un banco, hasta un sistema distribuido completo (capítulo 20). Incluso ya reúne las propiedades que algunos investigadores proponen para los sistemas operativos de nueva generación para el Web.

Los diferentes resultados comprobados en este trabajo, que combinados en el sistema integral le confieren muchas propiedades deseables también se pueden aplicar de manera individual a otros sistemas, para mejorarlos. En concreto, en el capítulo 21 se muestra cómo se pueden aplicar a la plataforma Java las ideas desarrolladas para el sistema integral. Entre ellas la implementación primitiva de clases para mejorar el soporte a otros lenguajes, la reflectividad para mejorar la extensibilidad de la máquina de Java, las capacidades como mecanismo de control uniforme para flexibilizar la seguridad, y la implementación como extensiones de las propiedades de persistencia y distribución transparentes.

22.2 Resultados a destacar dentro de los diferentes aspectos del sistema integral

Durante el desarrollo de los elementos del sistema integral, se llegó a algunas conclusiones y resultados que pueden destacarse de manera resumida por apartados:

22.2.1 Modelo de objetos

22.2.1.1 Uso de un modelo único

Para los objetivos del sistema integral de facilitar la comprensión del entorno usando el paradigma de la orientación a objetos es mejor dar soporte a un modelo único de objetos en lugar de a diferentes modelos de objetos (capítulo 7).

22.2.1.2 Adopción de las propiedades del modelo de las metodologías

Para aumentar el número de usuarios del sistema y el soporte a tecnologías orientadas a objetos se incorporan en el modelo las propiedades del modelo de objetos que se utilizan en las metodologías más utilizadas de análisis y diseño orientado a objetos (capítulo 7)

22.2.1.3 Necesidad de propiedades adicionales en el modelo de las metodologías

Para dar soporte a un sistema integral para el futuro, el modelo de objetos debe incluir nuevas propiedades que no se consideran aún o se empiezan a considerar como parte del modelo de objetos: identificador único de objetos, relaciones de asociación entre objetos, persistencia, distribución, concurrencia, seguridad y excepciones (capítulo 8).

22.2.2 Máquina abstracta orientada a objetos

22.2.2.1 Diseño de una arquitectura genérica de referencia para máquinas abstractas orientadas a objetos

Una arquitectura genérica de referencia para diseñar máquinas abstractas orientadas a objetos para un sistema integral se compone de áreas para las clases del sistema, las instancias de esas clases, las referencias que apuntan a los objetos y una serie de referencias especiales de la máquina (capítulo 11)

22.2.2.2 Rendimiento correcto de las máquinas abstractas para soportar un sistema completo

El rendimiento de las máquinas abstractas no es un problema para basar un sistema completo sobre ellas. Aunque intrínsecamente más lento que el código nativo compilado, existen técnicas como la compilación dinámica y otras que hacen que el rendimiento de las máquinas abstractas se aproxime al del código nativo (capítulo 11)

22.2.2.3 Elevación del nivel de abstracción con un alto nivel de la interfaz de la máquina

La interfaz de la máquina abstracta debe presentar un alto nivel de abstracción, independiente de estructuras internas de implementación. Por una parte facilita el uso de la máquina y por otra permite ampliar la gama de posibles implementaciones de la máquina, mejorando la productividad y la experimentación en esta actividad (capítulos 11 y 13).

22.2.2.4 Uso uniforme de la OO en la máquina abstracta

La falta de uniformidad en máquinas que dan soporte a tipos básicos especiales por un lado y a objetos por otro tiene muchos inconvenientes. Para mantener la uniformidad del entorno la máquina abstracta debe usar únicamente el paradigma OO, sin otro tipo de elementos especiales (capítulo 10).

22.2.2.5 Uso de la técnica de implementación primitiva de clases para lograr eficiencia en el uso uniforme

En lugar de bajar el nivel de abstracción de la máquina con un soporte especial más eficiente para tipos de datos básicos, se debe utilizar la técnica de implementación primitiva de clases. Así se proporciona eficiencia en las clases (objetos) básicos del sistema y se, mantiene la uniformidad de uso de la OO (capítulo 13).

22.2.3 Reflectividad

22.2.3.1 Inclusión de la reflectividad en la máquina abstracta para lograr la flexibilidad en el sistema con uniformidad

La inclusión de la propiedad de la reflectividad en la máquina abstracta es fundamental para permitir la extensión de la máquina abstracta por el sistema operativo de manera uniforme dentro del mismo paradigma OO, confiriendo gran flexibilidad al sistema (capítulos 14 y 19)

22.2.3.2 Uniformidad reflectiva mediante la fusión del meta-espacio y el espacio de usuario

Para permitir esa uniformidad total entre los objetos de usuario y los metaobjetos de la máquina, no deben separarse el espacio de meta-objetos (objetos de la máquina) del espacio de los objetos de usuario. Formando un único espacio de objetos, cualquier objeto puede interactuar con cualquier otro, independientemente de que sea un objeto de la máquina (reflejado) o no (capítulo 14)

22.2.3.3 Utilización de la llamada a método como mecanismo único de operaciones reflectivas y reflexión explícita

Con un espacio único de objetos se unifican en una sola operación la operación con objetos de usuario y la operación con los meta-objetos. La llamada a método es el único mecanismo necesario para interactuar con cualquier objeto del espacio único, sea objeto normal o meta-objeto. Esto también permite la reflexión explícita: un meta-objeto llama a un objeto normal. La reflexión explícita es fundamental para permitir la extensión por el sistema operativo. (capítulos 14 y 15).

22.2.3.4 Uso de la técnica de clases primitivas para implementar la reflectividad

La técnica de implementación primitiva de clases puede utilizarse para implementar la reflectividad formando así el espacio único: las clases reflejadas están dentro del espacio normal de clases de usuario (capítulo 14).

22.2.4 Sistema operativo

22.2.4.1 Identificación de la seguridad, persistencia, distribución y concurrencia como propiedades a proporcionar por el sistema operativo

Estas propiedades son deseables en un sistema integral, pero es más flexible proporcionarlas mediante un sistema operativo que extienda la máquina abstracta (capítulos 15 y 19).

22.2.4.2 Uso de extensión de la máquina en el espacio del usuario para lograr flexibilidad y transparencia

La manera de proporcionar las propiedades adicionales a los objetos de la máquina abstracta con flexibilidad es extender la misma con objetos de usuario (objetos del sistema operativo) que colaborando de manera reflectiva con la máquina consiguen estas propiedades de manera transparente (capítulos 5 y 15).

22.2.5 Persistencia

22.2.5.1 Aumento del nivel abstracción mediante la abstracción única de almacenamiento

La utilización de persistencia permite que se aumente el nivel de abstracción a los usuarios, al hacer que estos vean una única abstracción del almacenamiento, en lugar de la dualidad de memoria principal / secundaria (capítulo 16)

22.2.5.2 Uniformidad total con la persistencia completa

En lugar de utilizar algún elemento temporal, existe una mayor uniformidad tratando todos los objetos por igual como siempre persistentes y debe tenderse a ello (capítulo 16).

22.2.5.3 Unificación de técnicas de memoria virtual con la persistencia: Memoria virtual persistente

La aplicación de técnicas de memoria virtual al espacio de objetos persistentes permite crear la abstracción de una memoria persistente virtualmente infinita para almacenar los objetos. (16).

22.2.5.4 Incorporación rápida de la persistencia mediante el espacio de la implementación

El alto nivel de la interfaz de la máquina permite desarrollar rápidamente un prototipo que proporcione persistencia a los objetos de usuario indirectamente haciendo persistentes los objetos que los representan en el espacio de la implementación de la máquina (capítulo 18)

22.2.5.5 Sustitución de la recolección de basura por almacenamiento terciario

La técnica de memoria virtual hace que no se necesite recolección de basura en la memoria principal, puesto que la gestión de memoria virtual lleva los objetos no usados al disco. En lugar de hacer recolección de basura en el espacio persistente en memoria secundaria, puede sustituirse (o mejorarse) de una manera similar mediante un tercer nivel de memoria terciaria barata (capítulo 17)

22.2.6 Flexibilidad

22.2.6.1 Reunión en el sistema integral de las técnicas de flexibilidad

En el sistema integral compendia las propiedades de las diferentes técnicas existentes para lograr la flexibilidad: micronúcleos, familias de programas, sistemas operativos específicos para las aplicaciones, orientación a objetos y reflectividad (capítulo 19)

22.2.6.2 Aplicación de todas las técnicas de flexibilidad estática y dinámica

El sistema integral permite el uso a voluntad de todas las técnicas de flexibilidad estática y dinámica (capítulo 19).

22.2.6.3 Resolución del problema de la seguridad en la extensión dinámica mediante el mecanismo de protección uniforme del sistema

El problema de cómo controlar el uso seguro de la extensión dinámica de los sistemas no necesita soluciones específicas y poco flexibles en el sistema integral, para ello se usa simplemente el mecanismo normal de protección ya existente en el sistema (capítulo 19).

22.2.7 Aplicaciones del sistema

22.2.7.1 Posibilidad de aplicación flexible a un gran rango de ámbitos

La flexibilidad del sistema es tal que se puede utilizar la misma plataforma base en un gran rango de ámbitos, desde sistemas empotrados a sistemas distribuidos completos (capítulo 20).

22.2.7.2 Aplicación a un sistema operativo de nueva generación para el Web

El sistema integral reúne las propiedades que algunos investigadores proponen para los nuevos sistemas operativos para trabajar en los entornos futuros basados en el Web (capítulo 20).

22.2.8 Aplicación de resultados a otros sistemas

22.2.8.1 Mejora de la plataforma Java

La utilización de algunas de las ideas propuestas para el sistema integral mejora las condiciones de la plataforma Java para dar soporte a un entorno de computación completo, acercándola más a las propiedades que debe tener un sistema integral orientado a objetos (capítulo 21)

22.3 Trabajo y líneas de investigación futuras

El carácter de este trabajo de desarrollo del marco de trabajo y la arquitectura básica para la construcción de un sistema integral, deja abiertas muchas líneas de investigación, para completar y mejorar lo ya existente y para desarrollar el resto de los elementos. La dimensión de estas tareas necesita la colaboración de muchos investigadores. Como muestra se reseñan algunas de las líneas más inmediatas relacionadas con los temas más discutidos en este trabajo:

22.3.1 Máquina abstracta

22.3.1.1 Implementación más eficiente

Una primera línea de trabajo evidente es el desarrollo y aplicación de técnicas que mejoren la eficiencia de la implementación de la máquina abstracta. Estos aspectos no han sido tenidos en cuenta en los prototipos actuales y deben mejorarse en el futuro.

22.3.1.2 Formato compacto del fichero de clases

Como se menciona en el capítulo 12, un campo de investigación interesante es el de las técnicas de descripción compacta de las clases que se le proporcionan a la máquina para su ejecución, haciendo que la propia descripción impida la existencia de errores en la misma, garantizando la seguridad.

22.3.1.3 Desarrollo de implementaciones directamente sobre el hardware

Los simuladores de la máquina funcionan actualmente bajo otros sistemas operativos. Para ir evolucionando hacia el desarrollo completo de un sistema autónomo hay que comenzar por realizar implementaciones de la máquina que funcionen directamente sobre el hardware de manera autónoma, sin ayuda de otro sistema operativo. Para evitar tener que desarrollar controladores de dispositivos específicos y en general concentrarse en la propia implementación de la máquina y no en aspectos complementarios, se puede utilizar un paquete de componentes básicos de sistema operativo, como Flux [FBB+96].

22.3.1.4 Aspectos relacionados con la reflectividad

Existen muchos aspectos relacionados con la reflectividad que hay que concretar, como cuál es el mejor meta-modelo de objetos que debe reflejar la máquina, y su meta-interfaz, maneras de implementar la misma, etc.

22.3.2 Sistema operativo

Es necesario diseñar y desarrollar los aspectos fundamentales del sistema operativo. La persistencia[1], seguridad, la distribución y la concurrencia son propiedades muy importantes que deben diseñarse cuidadosamente. Actualmente otros investigadores se están ocupando ya de estos aspectos de manera individual. Cada uno de estos elementos forma una línea de trabajo por sí mismo.

22.3.3 Persistencia

22.3.3.1 Evolución de la implementación hacia el espacio del usuario

El actual prototipo está implementado como parte de la implementación primitiva de la máquina por rapidez de desarrollo. A medida que evolucione la máquina en la incorporación de la reflectividad, se deberán ir migrando estas implementaciones de su forma primitiva hacia el espacio del usuario.

22.3.3.2 Estabilidad, elasticidad y Algoritmos de implementación

Existen muchos algoritmos diferentes que se pueden utilizar en las distintas secciones del sistema de persistencia, que se pueden adaptar, probar y evaluar para los diferentes entornos de aplicación del sistema. Así mismo, deben de introducirse paulatinamente las propiedades de estabilidad y elasticidad en el sistema.

22.3.3.3 Implantación de transacciones

Junto con la parte de concurrencia del sistema operativo, el diseño e implantación de un modelo de transacciones en el sistema es otro de los objetivos que se deben acometer para permitir la aplicación del sistema en más ámbitos. En la implantación de transacciones está muy implicado el sistema de persistencia.

22.3.3.4 Sustitución de la recolección de basura por almacenamiento terciario

Es interesante investigar la posibilidad de añadir un tercer nivel al sistema de persistencia con almacenamiento terciario y comprobar si se puede sustituir totalmente la recolección de basura por este almacenamiento, y en qué casos según la utilización del sistema, o bien una sustitución parcial que ayude a la eficiencia de la recolección de basura, cómo implantarlo, etc.

22.3.3.5 Desarrollo de sistemas de gestión de bases de datos orientadas a objetos e integración en el sistema

Un campo muy amplio es el del desarrollo de sistemas de gestión de bases de datos a partir del soporte de persistencia del sistema. Además del propio desarrollo de la funcionalidad necesaria para un motor de base de datos, procesamiento de consultas, etc. otro aspecto muy interesante es la integración del sistema de bases de datos como parte básica del sistema, en lugar de un entorno separado. El usuario utilizaría el sistema de gestión como parte de su interfaz, lo consultaría para localizar sus objetos, etc. Es decir, cumpliría la funcionalidad de los sistemas de ficheros tradicionales, para los objetos, pero con mucha más

[1] Cuyo diseño e implementación preliminar se han acometido también como parte de este trabajo.

potencia y flexibilidad. Este apartado ya se encuentra en desarrollo por otros investigadores del proyecto.

22.3.4 Aplicaciones

22.3.4.1 Desarrollo de versiones para diferentes entornos

A medida que se vayan completando los elementos básicos del sistema, se puede comenzar a desarrollar versiones del mismo para diferentes ámbitos de aplicación, como los mencionados en el capítulo 20. Para cada ámbito se necesitarán implementaciones alternativas de algunos elementos, como la persistencia y la distribución que habrá que diseñar y desarrollar.

22.3.5 Aplicación a otros sistemas

22.3.5.1 Migración de resultados a otros sistemas como Java

Un campo interesante de trabajo es estudiar cómo se pueden aplicar las ideas desarrolladas para el sistema a otros sistemas comerciales como Java. Por ejemplo la ya mencionada incorporación de un mecanismo de capacidades a la máquina de Java, o la persistencia.

22.3.5.2 Aplicación de la arquitectura del sistema integral a proyectos comerciales

Aprovechando la experiencia y las técnicas desarrolladas para el sistema comercial, pueden aplicarse las mismas a proyectos comerciales que requieran alguna de las propiedades del sistema integral. Aunque no necesariamente se trate de desarrollo de versiones del sistema integral, puede aplicarse la misma filosofía de la arquitectura para desarrollar sistemas específicos adaptados a un proyecto en concreto. Por ejemplo, en el ámbito de desarrollo de aplicaciones para bases de datos, se utilizan lenguajes especiales de desarrollo. Una de las necesidades actuales es la portabilidad y heterogeneidad y el acceso distribuido a estas aplicaciones. La experiencia para solucionar esos problemas en el sistema integral se puede aplicar para desarrollar un sistema de soporte para esos lenguajes especiales que logre las propiedades anteriores.

Apéndice A

MANUAL DE USUARIO DEL ENTORNO INTEGRADO DE DESARROLLO

A.1 Introducción

La aplicación "Carbayonia IDE" es un entorno integrado de desarrollo (IDE, *Integrated Development Environment*) para el desarrollo de programas en Carbayonia en Windows.

Esta aplicación intenta proveer de las comodidades de un entorno integrado (edición, compilación y montaje desde un mismo lugar; gestión de proyectos; manejo de errores; etc.) aplicadas a la creación de programas en Carbayonia.

En un futuro se prevé integrar este entorno con el simulador para poder realizar todas las tareas del ciclo de desarrollo desde un mismo lugar centralizado. De esta manera se podrían ejecutar y depurar los programas desde el mismo lugar en el que se editan.

A.2 Descripción general del entorno

El entorno de desarrollo muestra habitualmente una configuración como la siguiente.

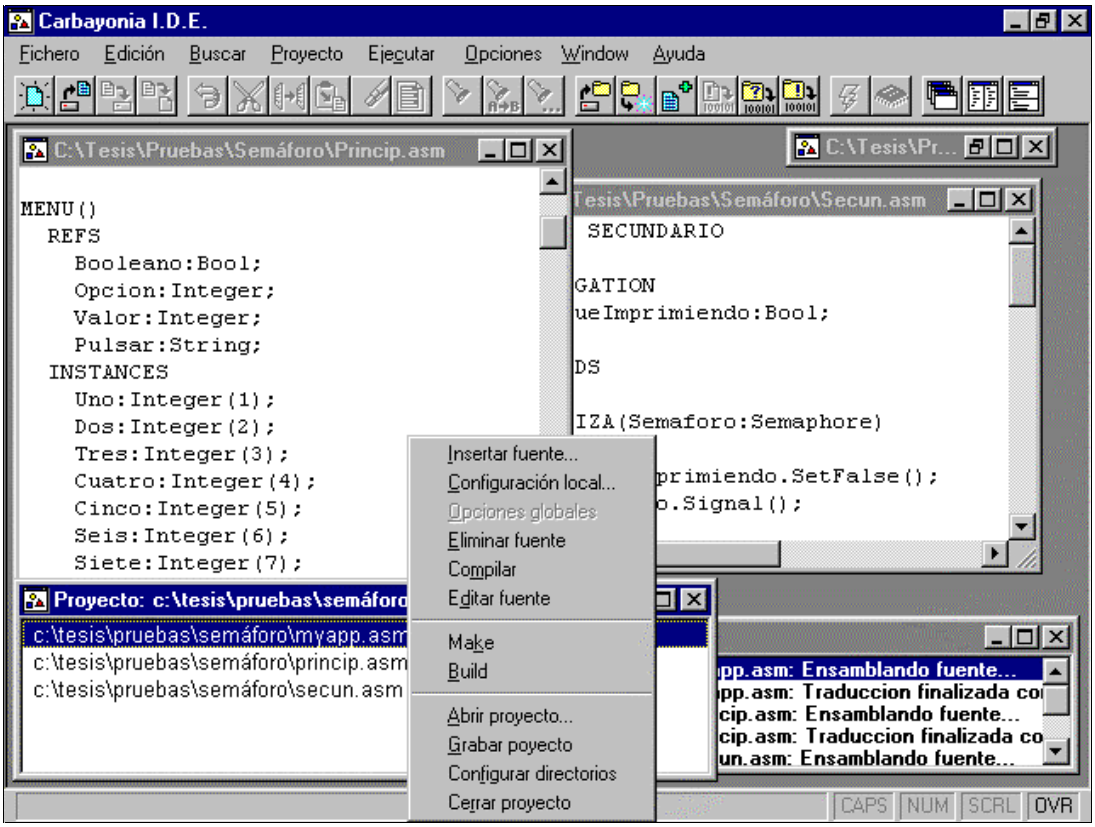


Figura A.1 Entorno de desarrollo.

El entorno consta de las siguientes partes:

- Un **área de título** con el cual se puede mover la ventana en la que está la aplicación. A su izquierda se encuentra el menú de control de la ventana (con el cual se puede cerrar la aplicación, mover, cambiar de tamaño, etc.) y a su derecha los botones de minimizar y maximizar. La aplicación comienza inicialmente maximizada.
- La **barra de menús**, donde se encuentran las principales opciones del programa clasificadas por funciones. La barra de menús se tratará posteriormente en detalle.
- Una **barra de control** que simplifica la elección de las opciones más usuales del programa sin necesidad de moverse por los menús. La barra de control se tratará posteriormente con mayor detalle.
- Un **área de trabajo** sobre el que se sitúan las diferentes ventanas que haya abiertas en la aplicación. Dentro de él podrán estar minimizadas, maximizadas o en tamaño normal.
- Una **barra de mensajes** en donde se da información adicional cuando el usuario pasa por encima de los botones de control o de las opciones del menú.

A.3 La barra de control

La barra de control es una fila de botones situada en la parte superior de la ventana principal que representan las opciones principales del programa. Pulsando el ratón en uno de los botones es una rápida alternativa a elegir una opción del menú.

Los botones de la barra de control son sensibles al contexto en el que esté la aplicación, de tal manera que se deshabilitan cuando la opción a la que representan no se pueda realizar. De esta manera garantizan una interfaz con el usuario consistente en el que no le ofrece una opción que no puede realizar. Por ejemplo, cuando no hay ningún proyecto abierto se deshabilitan los botones de grabar el proyecto, cerrarlo, añadir un elemento, etc.

A continuación se especifica la función de cada botón:

Botón	Opción Menú	Descripción
	**Fichero|Nuevo**	**Crea un nuevo fuente.**
	**Fichero|Abrir...**	**Carga del disco un fichero de texto.**
	**Fichero|Grabar**	**Graba el texto actual en disco.**
	**Fichero|Grabar como...**	**Permite grabar el texto bajo otro nombre.**
	**Edicion|Deshacer**	**Deshace la ultima operación sobre el editor.**
	**Edicion|Cortar**	**Mueve el texto seleccionado al Portapapeles.**
	**Edicion|Copiar**	**Copia el texto seleccionado al Portapapeles.**
	**Edición|Pegar**	**Trae el contenido del Portapapeles al editor.**
	**Edición|Borrar**	**Borra el texto seleccionado.**
	**Edicion|Borrar todo**	**Borra el documento completo.**
	**Busqueda|Buscar...**	**Busca una palabra en el texto.**
	**Busqueda|Reemplazar...**	**Busca y reemplaza una palabra en el texto.**
	**Busqueda|Siguiente**	**Retoma la última búsqueda o reemplazamiento.**
	**Proyecto|Abrir...**	**Abre un proyecto.**
	**Proyecto|Cerrar**	**Cierra el proyecto actual.**
	**Proyecto|Insertar**	**Añade un nuevo fuente al proyecto actual.**

	fuente...	
	Proyecto|Compilar	**Compila el fuente activo.**
	Proyecto|Make	**Compila los fuentes modificados.**
	Proyecto|Construir	**Construye el proyecto entero.**
	Ejecución|Volcar código	**Ejecuta el código.**
	Ejecucion|Ver código	**Abre una ventana de inspección.**
	Ventana|Cascada	**Coloca las ventanas en mosaico vertical.**
	Ventana|Mosaico vertical	**Coloca las ventanas en mosaico vertical.**
	Ventana|Mosaico horizontal	**Coloca las ventanas en mosaico horizontal.**

A.4 La barra de Menús

La barra de menús muestra las opciones generales de la aplicación. Está formada por diferentes menús que agrupan opciones relacionadas entre sí. Al igual que en el caso de la barra de botones, las opciones del menú son sensibles al contexto, de tal forma que todas aquellas que no se pueden seleccionar en ese momento aparecen deshabilitadas.

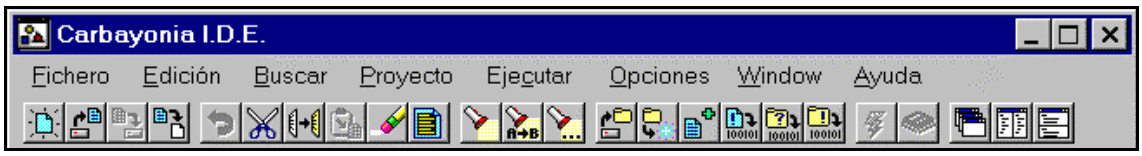


Figura A.2 Barra de menús.

A continuación se describen las opciones de cada uno de los menús.

A.4.1 Fichero

El menú Fichero, como su propio nombre indica, agrupa a todas las opciones relativas al manejo de ficheros.

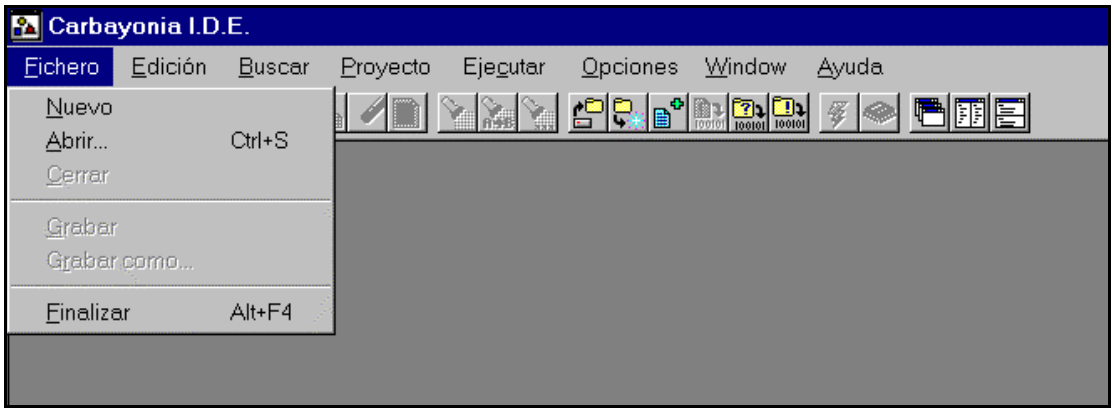


Figura A.3 Menú fichero.

- **Nuevo fuente**. Abre una ventana de edición con un texto sin título.
- **Abrir**. Presenta un cuadro de diálogo donde el usuario puede seleccionar el fuente a cargar para su edición.
- **Cerrar**. Cierra el documento activo.
- **Grabar texto**. Graba el texto activo en disco. En caso de que no tenga título se le presenta el cuadro de diálogo de Grabar como para que elija un nombre.

- **Grabar como**. Permite grabar el texto bajo un nuevo nombre.
- **Finalizar**. Finaliza la aplicación. Previamente comprueba si el proyecto o algún fuente en edición está sin grabar para dar la oportunidad de hacerlo antes de salir. Se puede anular la finalización del programa si se seleccionar cancelar a la pregunta de si desea grabar los datos.

A.4.2 Edición

Este menú recoge todas las opciones relativas a la edición de texto. Por tanto, solamente estarán activas cuando la ventana activa sea una ventana de editor.

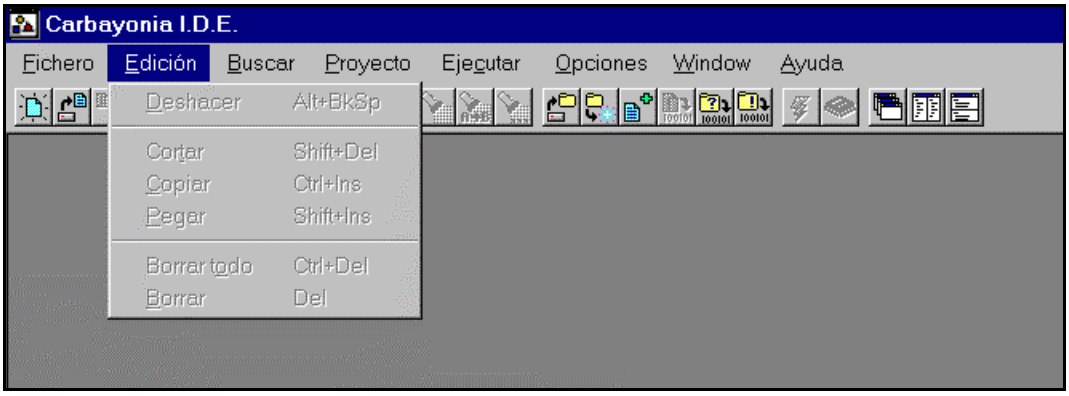


Figura A.4 Menú edición.

- **Deshacer**. Deshace el último cambio que se haya hecho sobre el texto.
- **Cortar**. Mueve el texto seleccionado al Portapapeles, borrándolo de la ventana.
- **Copiar**. Copia el texto seleccionado al Portapapeles, dejándolo también en la ventana.
- **Pegar**. Trae el contenido del Portapapeles a la posición actual del cursor.
- **Borrar todo**. Borra todo el texto de la ventana. Utilícese con precaución. Si se utiliza esta función por error se puede recuperar el texto mediante la opción de Deshacer.
- **Borrar**. Borra el texto seleccionado.

A.4.3 Buscar

Este menú reúne a todas las opciones relativas a la búsqueda y sustitución de textos. Por lo tanto tampoco estarán habilitadas si no se encuentra activa una ventana de edición.

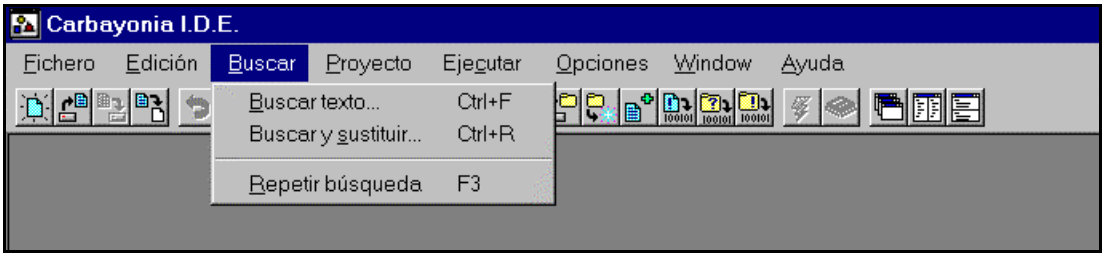


Figura A.5 Menú Buscar.

- **Buscar texto**. Presenta un cuadro de diálogo donde el usuario puede teclear la palabra a buscar así como las condiciones de búsqueda.

Figura A.6 Cuadro de diálogo de buscar.

- **Buscar y reemplazar**. Presenta un cuadro de diálogo donde el usuario puede indicar que texto quiere sustituir y por cual quiere hacerlo, además de las opciones de dicha sustitución.

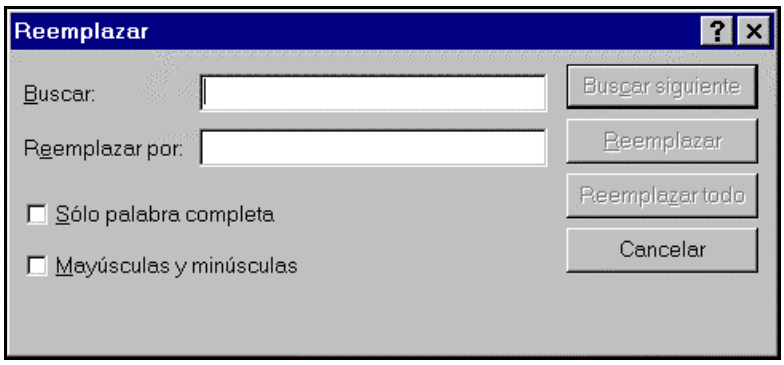


Figura A.7 Cuadro de diálogo de reemplazar.

- **Repetir búsqueda**. Retoma la última operación de búsqueda o sustitución que se haya realizado en el mismo punto donde se dejó.

A.4.4 Proyecto

En este menú se encuentran las opciones relativas a la gestión de proyectos.

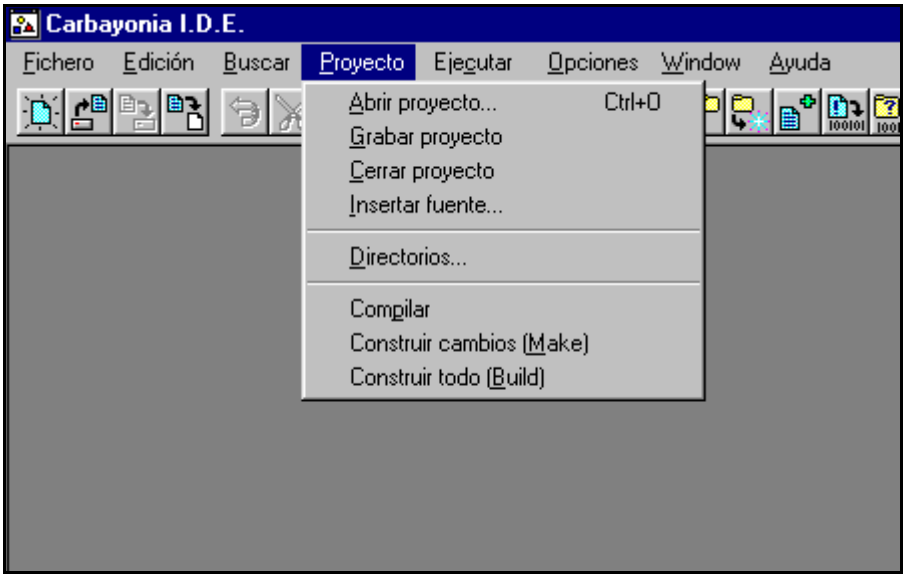


Figura A.8 Menú proyecto.

- **Abrir proyecto**. Presenta un cuadro de dialogo donde teclear un proyecto nuevo o abrir uno ya existente.

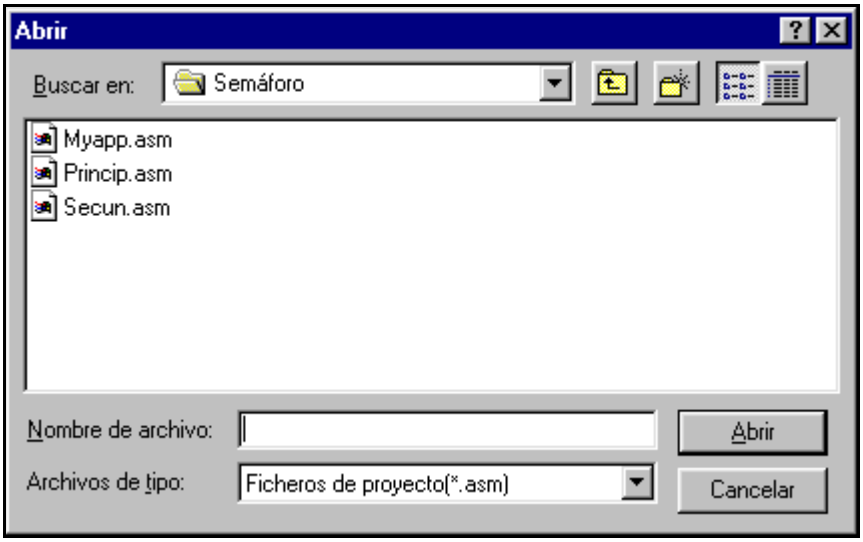


Figura A.9 Cuadro de diálogo de abrir.

- **Grabar proyecto**. Graba el proyecto actual en disco.
- **Cerrar proyecto**. Cierra el proyecto actual, indicando, si fuera necesario, si hay modificaciones sin grabar.
- **Insertar fuente**. Añade un nuevo fuente al proyecto, asignándole las opciones de compilación globales.
- **Directorios**. Aparece un cuadro de dialogo donde el usuario puede especificar donde se guardarán los ficheros que se generen en la traducción y en el enlazado.

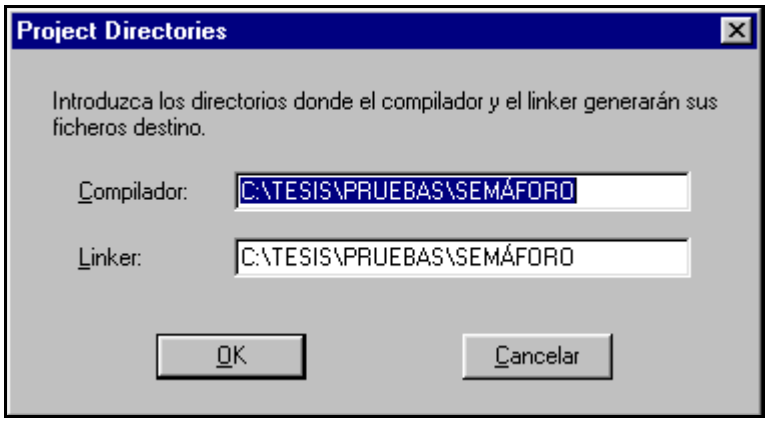


Figura A.10 Cuadro de diálogo de directorios.

- **Compilar**. Compila el fuente activo con sus opciones locales si las tiene.
- **Construir cambios**. Compila todos aquellos fuentes que se hayan modificado y enlaza si fuese necesario.
- **Construir todo**: Compila todos los fuentes y enlaza independientemente de que se hayan modificado o no.

A.4.5 Ejecutar

Este menú recoge un grupo de opciones pendientes de implementar previstas para la futura integración del entorno de desarrollo con el simulador.

Estas opciones actualmente se encuentran en el simulador.

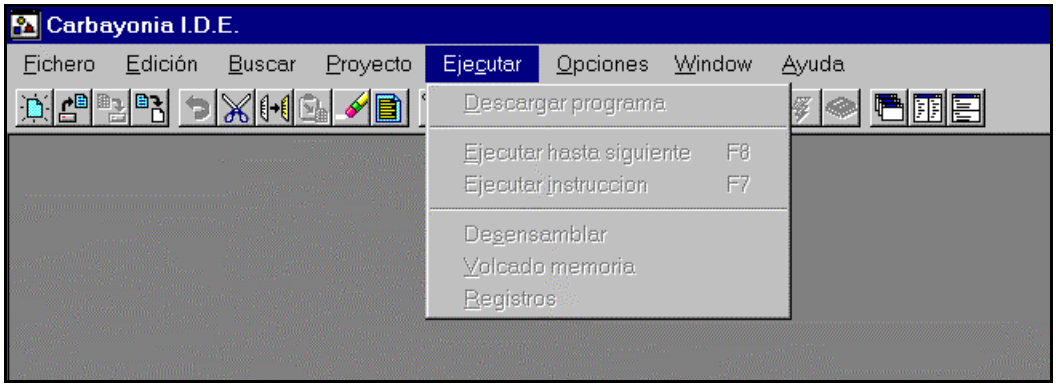


Figura A.11 Menú Ejecutar.

- **Cargar programa**. Lee un fichero ejecutable desde disco.
- **Ejecutar hasta siguiente**. Ejecuta hasta la instrucción siguiente, aunque para ello tenga que ejecutar una rutina completa.
- **Ejecutar instrucción**. Ejecuta una sola instrucción.
- **Desensamblar**. Abre una ventana de depuración donde se puede ver el código y monitorizar su funcionamiento.
- **Volcado memoria**. Abre una ventana de depuración donde se puede ver el área de instancias.
- **Registros**. Se abre una ventana de depuración donde se pueden consultar las referencias del sistema.

A.4.6 Opciones

En este menú se encuentran las opciones de configuración del entorno.

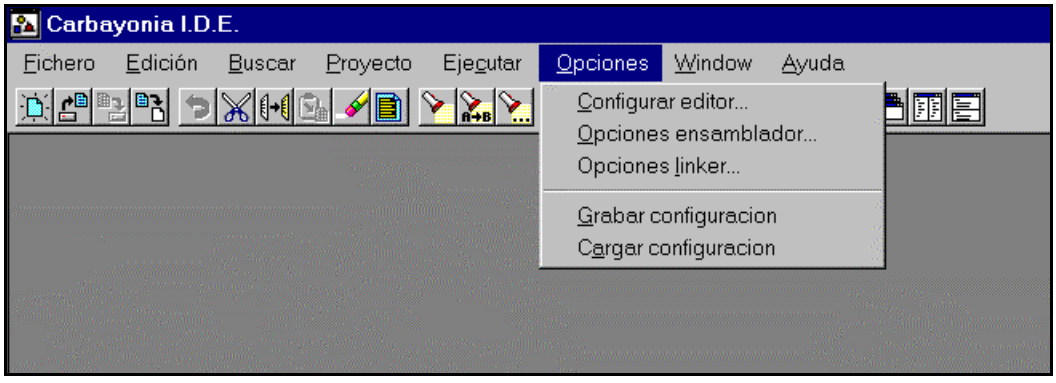


Figura A.12 Menú opciones.

- **Configurar editor**. Presenta un cuadro de diálogo donde el usuario puede elegir el tipo de letra que desea utilizar en los editores.
- **Opciones compilador**. Se presenta un cuadro de diálogo donde se pueden definir las opciones globales del compilador. Estas opciones son las que se aplicarán a todos aquellos fuentes que no pertenezcan al proyecto o que pertenezcan y no tengan configuración local.
- **Opciones enlazador**. Se presenta un cuadro de diálogo donde se pueden definir las opciones del enlazador.

- **Grabar configuración**. Graba la configuración del editor, compilador y enlazador en un fichero para que a partir de ahora sea la configuración inicial al entrar en el entorno. Además, se puede utilizar para copiar configuraciones entre proyectos.
- **Cargar configuración**. Carga la configuración del editor, compilador y enlazador de disco. Esta opción se ejecuta automáticamente al cargar la aplicación.

A.4.7 Window

Este menú agrupa las opciones relativas a la gestión de ventanas.

Debajo de la opción *minimizar* se irán colocando los títulos de todas las ventanas que estén abiertas en el escritorio, permitiendo pasar de una a otras a través de éste menú. Si se abren simultáneamente mas de diez ventanas, aparecen solamente las nueve primeras y debajo una opción titulada "Más ventanas..." la cual presenta una lista desplazable donde poder seleccionar el resto de las ventanas.

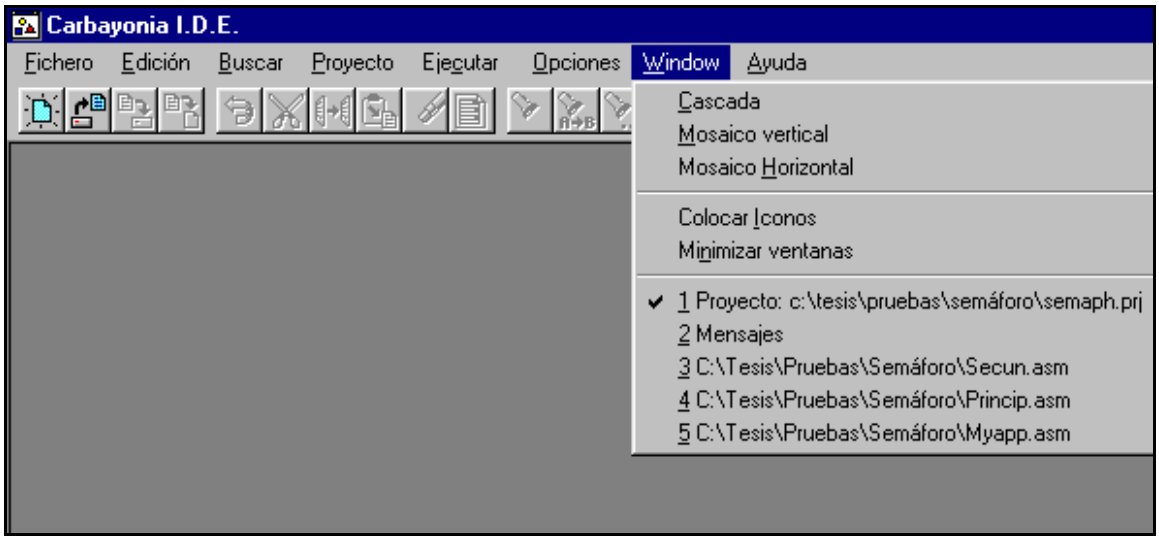


Figura A.13 Menú Window.

- **Cascada**. Sitúa las ventanas no minimizadas en cascada partiendo de la esquina superior izquierda.
- **Mosaico vertical**. Sitúa las ventanas no minimizadas verticalmente de izquierda a derecha.
- **Mosaico horizontal**. Sitúa las ventanas no minimizadas horizontalmente de arriba a abajo.
- **Colocar iconos**. Ordena todos los iconos del escritorio en la parte inferior del mismo de izquierda a derecha.
- **Minimizar**. Minimiza (transforma en iconos) todas las ventanas.

Figura A.14 Ventanas del entorno minimizadas.

A.4.8 Ayuda

En este menú permite obtener información sobre la aplicación.

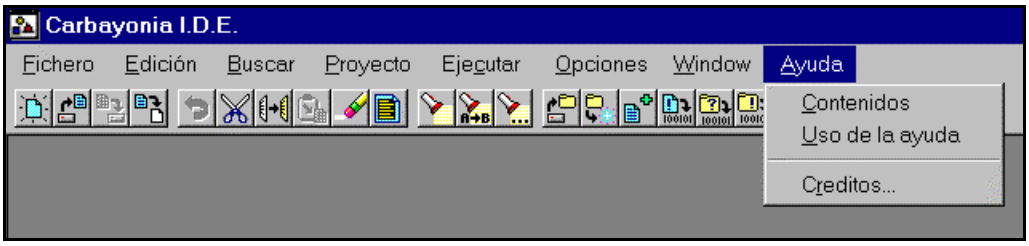


Figura A.15 Menú proyecto.

- **Contenidos**. Presenta la ayuda de la aplicación.
- **Uso de la ayuda**. Presenta información sobre como manejar y moverse en la propia ayuda.

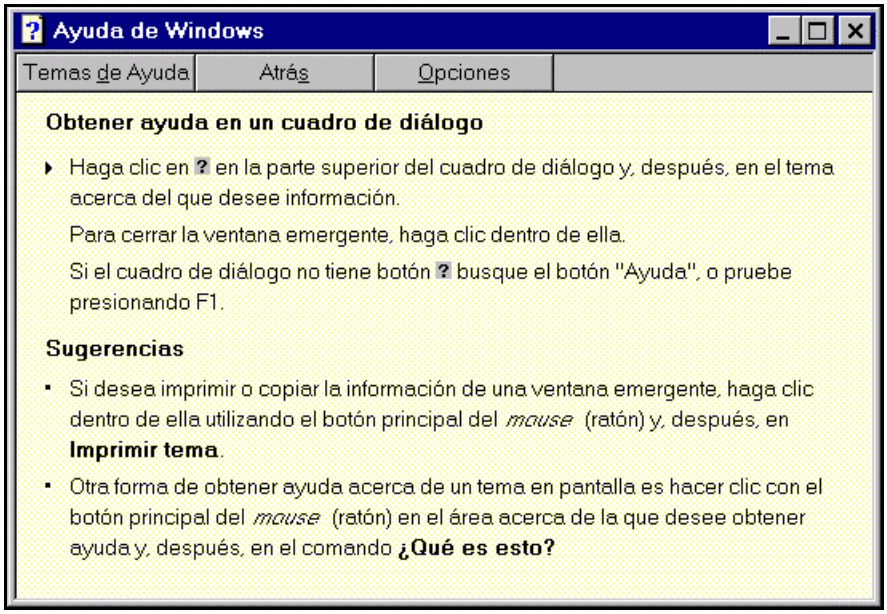


Figura A.16 Uso de la ayuda.

- **Créditos**. Presenta el usual cuadro de dialogo que identifica la aplicación.

A.5 Tipos de Ventanas

Aparte de la barra de menús y de la barra de botones, existe otra manera de seleccionar opciones en la aplicación. Se trata de los menús locales, los cuales se presentan al pulsar el botón derecho sobre el área de una ventana.

Cada ventana tiene su propio menú local que presenta todas las opciones posibles aplicables en esa ventana. Con la práctica se convierten en la principal forma para seleccionar opciones, debido a la comodidad de no tener que mover el ratón hacia la barra de menús (para luego tener que estar desplegando menús) o la barra de botones. Simplemente pulsando el botón derecho del ratón se nos presentan todas las opciones posibles en esa ventana y en ese contexto.

Existen tres tipos de ventanas en el entorno: ventana de edición, ventana de proyecto y ventana de mensajes.

A.5.1 Ventana de edición

Ventana donde se editan los fuentes. Se crean con la opción Nuevo o Abrir y tienen la funcionalidad habitual asociada a cualquier editor de textos.

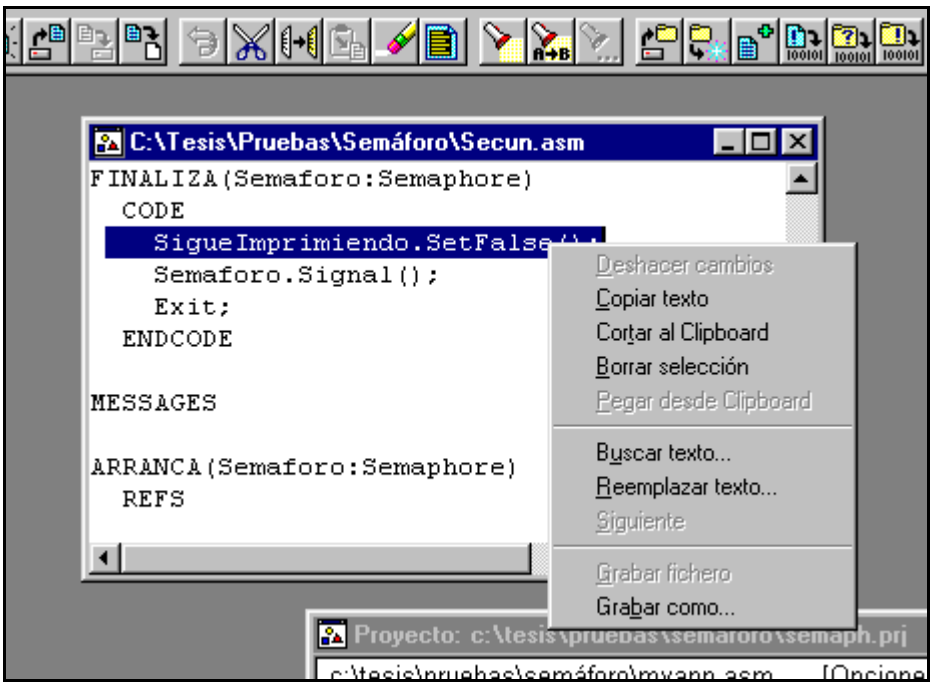


Figura A.17 Ventana de edición.

Su menú local reúne todas las opciones relativas a la edición que en la barra de menús se encuentran repartidas entre los menús Fichero, Edición y Buscar por lo que se repetirán sus descripciones. Véase el apartado dedicado a la barra de menús.

A.5.2 Ventana de proyecto

Ventana donde se muestran los componentes del proyecto actual. Presenta los fuentes que forman el proyecto y que tipo de opciones de compilación tienen.

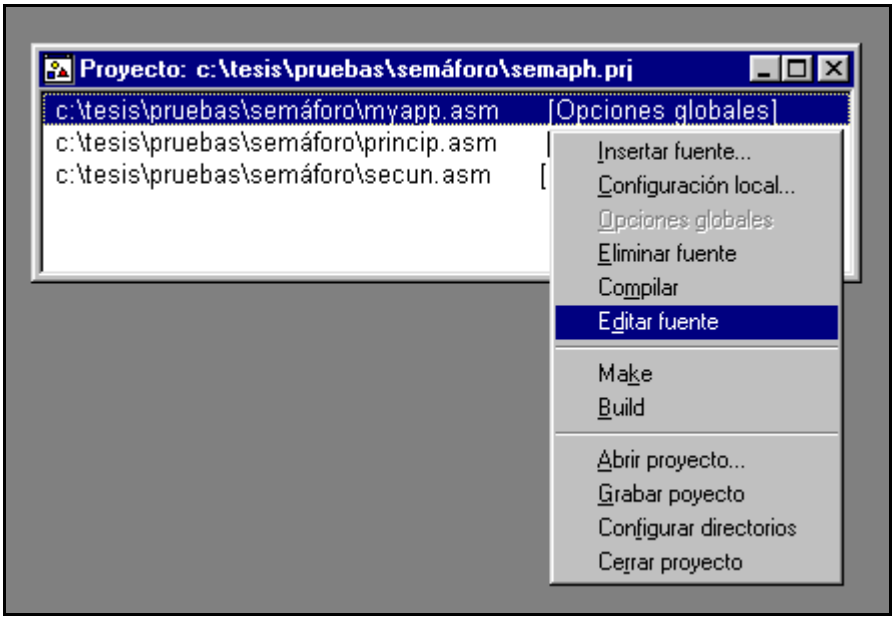


Figura A.18 Menú flotante de proyecto.

El menú local reúne todas las opciones relativas al proyecto. La mayoría de ellas ya se han tratado en el apartado dedicado a la barra de menús.

Las opciones que no se encuentran en la barra de menús son:

- **Configuración local**. Establece una configuración local para el fuente seleccionado dentro de la ventana.
- **Opciones globales**. Establece las opciones globales para el fuente seleccionado, perdiéndose sus opciones locales en caso de que las tuviese.
- **Eliminar fuente**. Elimina el fuente seleccionado del proyecto.
- **Editar fuente**. Esta opción hace que el fuente seleccionado se abra en el escritorio para su edición. Esta opción también se puede conseguir haciendo doble-*click* sobre el fuente deseado.

A.5.3 Ventana de mensajes

A esta ventana es donde van a parar todos los mensajes que se generen en los procesos de compilación y enlazado para su posterior inspección y edición. Por cada mensaje se muestra el fuente destinatario del error y un texto explicativo del error.

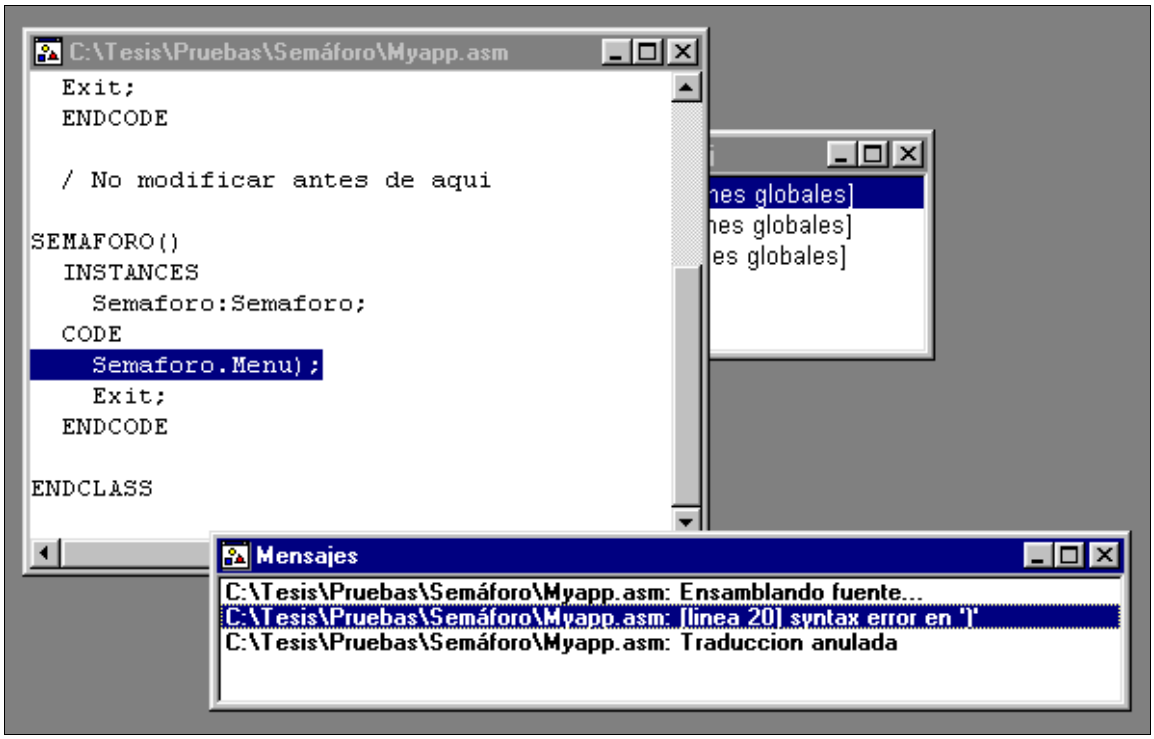


Figura A.19 Error resaltado.

Moviéndose con las teclas del cursor a través de los mensajes, éstos se irán mostrando sobre el fuente correspondiente si éste se encuentra en edición. Si se desea editar un error de un fuente que no se esté editando actualmente, basta hacer doble-*click* sobre el error deseado para que se abra el fuente y se sitúe el editor en la posición del error.

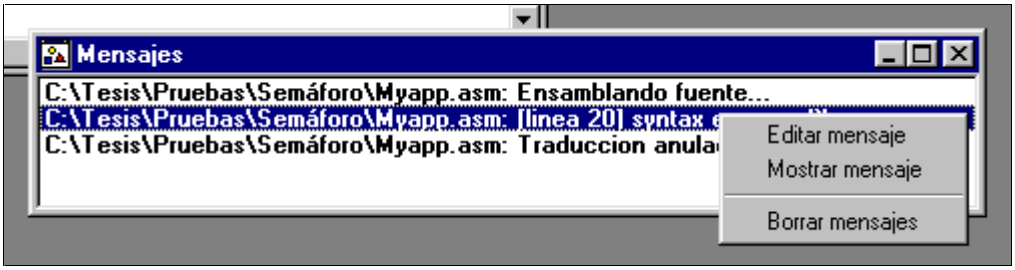


Figura A.20 Menú local de la ventana de mensajes.

- **Editar error**. Abre el fuente correspondiente para que se edite el error o lo activa si ya está cargado.
- **Mostrar error**. Resalta el error en el fuente correspondiente si éste se está editando actualmente.
- **Borrar Mensajes**. Limpia la ventana de mensajes.

A.6 Comprobaciones semánticas del traductor incorporado

El traductor de Carbayón a formato del fichero de clases incorporado en el entorno de desarrollo realizará exclusivamente aquellas comprobaciones que puedan llevarse a cabo con la información disponible en el módulo que se compila.

Por tanto no intentará, por ejemplo, comprobar la signatura de un método que corresponda a otra clase. Dichas comprobaciones cruzadas entre clases se posponen para que sean llevadas por un enlazador o por la propia máquina al cargar las clases.

Las comprobaciones semánticas realizadas por el compilador son:

- **Intento de acceder a una etiqueta no declarada**. Las etiquetas son locales a los métodos, por lo que las referencias a etiquetas sólo se comprobarán con las declaradas en el mismo método en el que son encontradas.

```
Prueba1()
Code
     Jmp etiq   // Error: 'etiq' fuera de ámbito
     Exit
EndCode

Prueba2()
Code
     . . .
etiq:
     Exit
EndCode
```

- **Los nombres de las referencias en las cláusulas Aggregation y Association deben ser distintos entre sí**. No puede repetirse el nombre de una referencia en la misma cláusula en la que está declarada ni tampoco en la opuesta.

```
Class Prueba
Isa claseZ

Aggregation
     ref:claseW;
     ref:claseX;      // Error. Referencia duplicada

Association
     ref:claseY;      // Error. Referencia duplicada
```

- **Los nombres de los argumentos, referencias locales (Refs) e instancias locales de un método deben ser distintas dos a dos**. Por ejemplo se considerará un error semántico que una referencia local se llame igual que uno de los argumentos del método.

```
Prueba1(param1:claseX)
Refs
     param1:claseY;  // Error. Referencia duplicada

Instances
     param1:claseZ;  // Error. Referencia duplicada
```

  Sin embargo se permite que cualquiera de estos grupos de referencias reutilice un nombre presente en las cláusulas Aggregation y Association de la clase a la que pertenecen. En ese caso simplemente la referencia de las cláusulas anteriores queda oculta por la referencia local.

```
Class Prueba
Isa claseZ

Aggregation
     ref:claseW;

Methods // Correcto. El aggregation ref:claseW oculto
Prueba1(ref:claseM)
Code
     .
     .
     .
EndCode

Prueba2()
Refs
     ref:claseY;// Correcto. El aggregation ref:claseW oculto
Code
     .
     .
     .
EndCode

Prueba3()
Instances
```

```
     ref:claseZ;// Correcto. El aggregation ref:claseW oculto
Code
     .
     .
     .
EndCode
```

- **Una etiqueta no puede ser definida dos veces en un mismo método**. Dado que las etiquetas son locales la única forma de repetirlas es en métodos diferentes.
- **No se podrá repetir dos veces un mismo ancestro en una misma cláusula Isa**. Aunque se permite herencia múltiple lo anterior es redundante y por lo tanto se considera un error.

```
Class Prueba
Isa claseZ, claseZ;  // Error: claseZ duplicada redundante
```

- **No puede haber dos funciones (método o mensaje) con el mismo nombre**. El lenguaje no contempla la sobrecarga y por tanto la considera como un error.

```
Class Prueba
Isa claseZ

Methods

Prueba(ref:claseM)
Code
     .
     .
     .
EndCode

Prueba(r1, r2, r3:claseG) // Error. Método duplicado
Code
     .
     .
     .
EndCode
```

A.7 Mensajes de error del entorno integrado de desarrollo

A continuación se da una relación de los errores que pueden aparecen en la ventana de mensajes del entorno integrado.

A.7.1 Mensajes de la traducción

Mensajes de error o advertencias producidas por la traducción del programa Carbayón al formato del fichero de clases.

- **Etiqueta <nombre> no declarada.**

  En una instrucción de salto (JF, JFD, JT, TJD, JNull, JNNull o JMP) o en la instrucción Handler se ha utilizado como operando una etiqueta no declarada.

  Las etiquetas utilizadas por las instrucciones anteriores deben estar dentro del mismo método que la instrucción que la referencia.

- **Referencia <miembro> duplicada (I)**

Se ha utilizado dos veces el mismo nombre de referencia en las cláusulas Aggregate y/o Association.

Un identificador utilizado para nombrar a un miembro agregado o a una asociación no puede volver a utilizarse para denominar a otra referencia de ninguna de las dos cláusulas.

- **Referencia <local> duplicada (II)**

  Se ha utilizado el mismo nombre de referencia como argumento de un método, como referencia local y/o como instancia local.

  Un identificador utilizado para nombrar a un argumento, referencia local o instancia local no puede volver a utilizarse para denominar a otra referencia de ninguno de los grupos anteriores.

- **Etiqueta <nombre> duplicada**

  Se ha declarado dos veces la misma etiqueta en un mismo método.

  Se permite reutilizar la misma etiqueta en distintos métodos ya que el ámbito de ésta es local.

- **Identificador <ancestro> repetido (I)**

  Se ha definido mas de una vez un mismo ancestro en la cláusula Isa.

  Aunque se permite la herencia múltiple, indicar dos o más veces una misma clase en una misma cláusula Isa se considera un error.

- **Identificador <función> repetido (II)**

  Se ha definido mas de una vez un mismo nombre de función.

  En una clase no puede haber dos métodos o mensajes con el mismo nombre. El lenguaje no permite la sobrecarga.

- **[línea <numero>] Error de sintaxis en <token>**

  Se ha detectado un error de sintaxis en la línea señalada.

  El error puede estar provocado por el *token* anterior al mostrado, por el mismo *token* o por el que le sucede, dependiendo del tipo del error.

- **Warning. Ignorado carácter incorrecto en línea <numlinea>**

  Se ha encontrado un carácter desconocido en el fichero de entrada.

  El carácter simplemente es ignorado, por lo que la traducción continua normalmente.

A.7.2 Mensajes de entrada/salida

Mensajes debidos a las operaciones de entrada/salida.

- **Error de apertura**

  No se pudo abrir el fichero fuente.

  El fichero puede que no exista o que esté en uso por otra aplicación.

- **Error de escritura**

  El código objeto no se pudo escribir en el fichero destino.

  Revise sus permisos de escritura en el fichero o asegúrese de que no esté en uso por otra aplicación.

- **Traducción completada**

  El fuente no contiene errores y se ha generado el fichero objeto.

- **Traducción anulada**

  Debido a los errores léxicos, sintácticos o semánticos la traducción del fuente no ha podido llevarse a cabo completamente.

  El fichero objeto no ha sido generado (o se ha borrado lo que de él se hubiese generado).

Apéndice B

EJEMPLOS DE PROGRAMACIÓN EN LENGUAJE CARBAYÓN

B.1 Entrada y salida

En este ejemplo se muestra cómo puede realizarse la entrada/salida. Para ello se construye un programa equivalente al siguiente programa en C:

```
/* Versión en C */
void main (void)
{
     int a[10];

     for (i=0; i<10; i++)
          scanf ("%d", a[i]);

     for (i=0; i<10; i++)
          fprintf ("%d\n", a[i] * 10);
}
```

```
[1]   / Versión en Carbayón
[2]   Class MyApp
[3]   Isa Application
[4]   Methods
[5]
[6]   RUN()
[7]   Refs
[8]         b: Bool;
[9]         num: Integer;
[10]  Instances
[11]        con:ConStream;
[12]        a:Array;
[13]        i:Integer;
[14]        cero:Integer(0);
[15]        diez:Integer(10);
[16]        uno:Integer(1);
[17]  Code
[18]        a.setSize(diez);
[19]        i.set(cero);
[20]  leerEntero:
[21]        i.menor(diez): b;
[22]        Jfd b, finLectura;
[23]        con.read(): num;
[24]        a.setRef (i, num);
[25]        i.add(uno);
```

```
[26]        Jmp leerEntero;
[27]
[28]  finLectura:
[29]        i.set(cero);
[30]
[31]  escribeEntero:
[32]        i.menor(diez): b;
[33]        Jfd b, fin;
[34]        v.getRef(i): num;
[35]        num.mul(diez);
[36]        con.write(num);
[37]        Delete num;
[38]        con.newLine();  /Salto de línea
[39]        i.add(uno);
[40]        Jmp escribeEntero;
[41]
[42]  fin:
[43]        Exit;
[44]  EndCode
[45] EndClass
```

B.2 Comentario sobre las líneas más interesantes

Como puede observarse, un programa en Carbayonia es básicamente una sucesión de llamadas a métodos. A continuación se comentarán algunas líneas del ejemplo a tener en cuenta:

- **21**. El método menor devuelve un objeto de tipo Bool, al cual apuntará la referencia b. Si anteriormente ésta apuntase a otro objeto éste se perdería (si no hubiese ninguna otra referencia apuntando a él).
- **22**. A medida que se van utilizando los objetos de tipo Bool se procede a borrarlos ya que no tienen mayor utilidad. Esto se hace automáticamente al añadir el sufijo D a la instrucción de salto.
- **23-25**. En cada iteración el método read devuelve un objeto de tipo Integer que se recoge mediante la referencia num. A continuación se guarda su referencia en el array, de manera que se pueda volver a utilizar la referencia num para crear otros objetos sin peligro de perderlos.
- **32-40**. El bucle de escritura es muy similar al de lectura. Se va recorriendo el array imprimiendo los números y liberando los enteros. Como se dijo anteriormente un array no se hace responsable de los objetos a los que apunta. Además un método sólo es responsable de los objetos que cree explícitamente con New o bien que le hayan sido entregados por métodos de otras clases, ya que los objetos creados en la cláusula Instances son liberados automáticamente por la instrucción Exit.

B.2.1 Comprobación de tipos en tiempo de ejecución

En este ejemplo se puede observar como trabaja la comprobación de tipos en tiempo de ejecución. Así, en la línea 36 se hace automáticamente un amoldamiento estático (de la clase hija Integer a la clase base Object) y en la línea 34 se hace un amoldamiento dinámico (de la clase base Object a la clase hija Integer). En el primer tipo de conversión no hay ningún peligro, pero en el segundo se hubiese podido producir una excepción si el objeto devuelto por el método getRef no hubiese sido un Integer o derivado.

B.3 Lista circular doblemente enlazada

En el siguiente ejemplo se implementará una clase contenedora consistente en una lista circular doblemente enlazada. Dicha clase guardará referencias a objetos de tipo Object. Dado que todos las clases de Carbayonia descienden de ésta clase, sirve para guardar referencias a cualquier objeto.

B.3.1 Contenedores directos e indirectos

En C++ hay veces en que se produce confusión a la hora de decidir entre un contenedor que guarde los propios objetos o uno que guarde punteros a lo mismos (normalmente llamados contenedores directos e indirectos respectivamente). Esta decisión repercute sobre la forma en que se tiene que utilizar el mismo y sobre los métodos mínimos que deben tener los objetos a almacenar.

Pero realmente la razón más importante para utilizar los contenedores directos es la eficiencia a la hora de trabajar con los tipos básicos (enteros, char, etc.) ya que es caro tener por una parte el contenedor con los punteros y por otro lado los propios datos. Sin embargo, en una arquitectura como la de Carbayonia no hay diferencias entre tipos básicos y objetos, sólo existen objetos que se manejan de manera uniforme mediante referencias. Así que sólo existe un tipo de contenedores que contienen objetos (referencias a los objetos)

Si la extracción de un objeto de un contenedor implica la destrucción del objeto depende de lo que se prefiera en cada caso. De todas formas podrían implementarse dos métodos en el contenedor: uno que elimine sólo la referencia y otro que además los destruya, siendo el usuario el que decide cuál utilizar.

B.3.2 Clase Nodo Doble

Antes de pasar a la implementación de la lista, será necesario definir previamente una clase de apoyo utilizada internamente. Dicha clase es el nodo doble:

```
Class NodoDoble
Isa Object           / no es necesario especificarlo
Association
      left, right: NodoDoble;
      data: Object;
Methods
      setLeft (l:nodoDoble)
      Code
      Assign left, l
            Exit
      EndCode

      setRight (r:nodoDoble);
      Code
      Assign right, r
            Exit
      EndCode

      setData (d:Object);
      Code
      Assign data, d
            Exit
      EndCode

      getLeft (): nodoDoble;
      Code
      Assign rr, left
      Exit
      EndCode

      getRight (): nodoDoble;
      Code
      Assign rr, right
```

```
                Exit
        EndCode

        getData (): Object;
        Code
        Assign rr, data
                Exit
        EndCode

EndClass
```

B.3.3 Clase lista Doble Circular

Una vez definida la clase nodoDoble ya se puede definir la clase listaDobleCircular:

```
Class listaDobleCircular
Isa Object
Aggregation
        num: Integer;  / Numero de nodos que hay en la lista
Association
        current: nodoDoble;
Methods
        init();

        insert (Object);
        extractCurrent();

        next();
        previous();
        getCurrent(): Object;
        numNodos(): Integer;

        destroy();
        checkEmpty();
EndClass
```

El método init deberá ser llamado antes que cualquier otro de los métodos de la lista. Se correspondería con el constructor de la clase en C++.

El método insert sirve para introducir la referencia de un objeto en la lista en la posición actual. extractCurrent realiza la labor opuesta eliminando el nodo actual de la lista. Se recuerda una vez más que lo que se elimina de la lista es la referencia al objeto, no resultando éste afectado.

Los métodos next y previous sirven para moverse por la lista. GetCurrent devuelve una referencia al objeto en la posición actual de la lista.

NumNodos devuelve una instancia de un entero que indica cuantos elementos hay en la lista. Nótese que ese entero deberá ser liberado ya que no se devuelve una referencia a la variable miembro num sino una copia de ella.

El método destroy equivaldría al destructor de la clase. Se encarga de liberar todos los nodos dobles que se hayan creado en la lista (pero no los objetos a los que referencian a través de data).

Finalmente el método checkEmpty en una función interna que lanza una excepción si la lista está vacía. La utilizan otros métodos de la clase para no repetir el código en cada uno de ellos (equivale a una función private del C++).

B.3.4 Métodos de la clase Lista Doble Circular

A continuación se declaran los métodos de la clase. Se ha extraído el cuerpo de los métodos de la definición de la clase por claridad, pero en realidad deberían estar dentro de ella.

```
/----------------------------------------
/ init: inicializa la lista
init()
Instances
      cero:Integer(0);
Code
      num.set(cero);
      Exit;
EndCode

/----------------------------------------
/ insert: introduce una referencia en la lista
insert (ob:Object)
Ref
nodo, tmp:nodoDoble;
Instances
      uno:Integer(1);
Code
      num.add(uno)
      New nodo
      nodo.setData(ob)
      JNull current, listaVacia

/cuando ya hay algun elemento en la lista
      nodo.setLeft (current)
      current.getRight(): tmp
      nodo.setRight(tmp)
      current.setRight(nodo)
      tmp.setLeft(nodo)
      Assign current, nodo
      Exit

listaVacia:   /Cuando no habia ninguno en la lista
      Assign current, nodo
      current.setLeft(current)
      current.setRight(current)
      Exit
EndCode


/----------------------------------------
/ extractCurrent: extrae de la lista el nodo actual
extractCurrent ()
Refs
b:Bool;
      anterior, posterior: nodoDoble;
Instances
      uno:Integer(1);
      cero:Integer(0);
Code
      this.checkEmpty()
      num.sub(uno)
      num.equal(cero): b
      Jtd b, vaciarLista

      / Si queda al menos un nodo
      current.getLeft(): anterior
      current.getRight(): posterior
      anterior.setRight(posterior)
      posterior.setLeft(anterior)
      Delete current / se borra el nodo, no el data
      Assign current, posterior
      Exit

vaciarLista:
      Delete current /se borra el nodo, no el data
                           / current = NULL
      Exit
```

```
EndCode

/----------------------------------------
/ next: el siguiente nodo pasa a ser el actual
next ()
Code
      this.checkEmpty()
      current.getRight(): current
      Exit
EndCode

/----------------------------------------
/ previous: el anterior nodo pasa a ser el actual
previous ()
Code
      this.checkEmpty()
      current.getLeft(): current
      Exit
EndCode

/----------------------------------------
/ getCurrent devuelve una referencia al objecto actual
getCurrent (): Object
Code
      this.checkEmpty()
      current.getData(): rr
      Exit
EndCode

/----------------------------------------
/ numNodos: devuelve un entero con el n° de nodos
numNodos ():Integer
Refs
      tmp:Integer;
Code
      New tmp
      tmp.set(num)
      Assign rr,tmp
      Exit
EndCode

/----------------------------------------
/ destroy: libera los nodos dobles
destroy ()
Code

      handler atrapa
borrarNodo:
      this.extractCurrent()
      Jmp borrarNodo

atrapa:
      Exit
EndCode

/----------------------------------------
/ checkEmpty: genera una excepción si lista vacia
checkEmpty ()
Refs
e:Integer;
Code
      JNull current, vacia
      Exit
vacia:
      New e
      throw e
EndCode
```

Esta lista circular sirve para guardar objetos de cualquier tipo, incluso listas circulares. Esto es, los nodos de la lista pueden ser a su vez listas y así sucesivamente (ya que la lista circular es un Object).

Apéndice C

EJEMPLO DE PROGRAMACIÓN PERSISTENTE: APLICACIÓN DE BASES DE DATOS

Dadas las especificaciones del lenguaje orientado a objetos Carbayón y su ampliación para el sistema de persistencia, se desarrollará una pequeña aplicación de bases de datos orientadas a objetos.

La base de datos orientada a objetos, se basa en la colección de dos tipos de elementos: archivos y autores. Los archivos pueden ser revistas o libros, se utilizará pues el polimorfismo y la herencia.

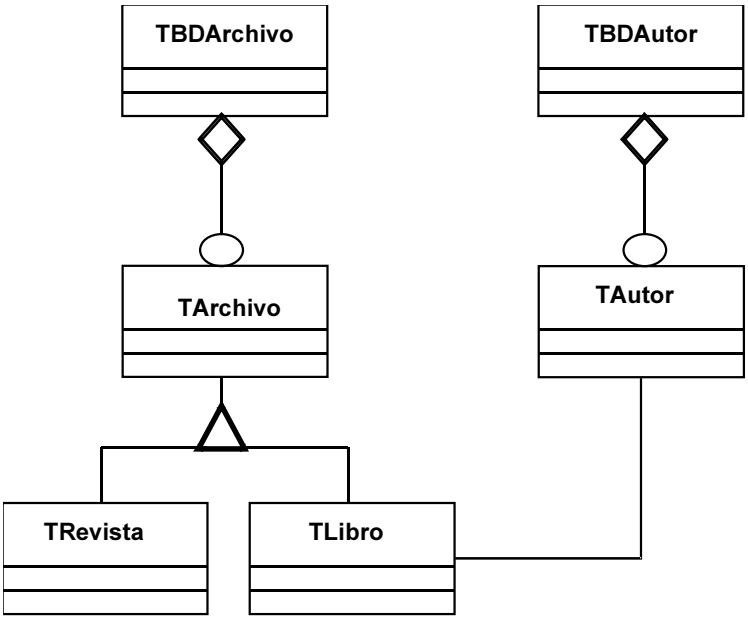


Figura C.1 Diagrama de clases de la base de datos de ejemplo.

Cada libro está asociado al autor, no así una revista. Las consultas cruzadas de libros referente a autores, serán aceleradas mediante esta relación de asociación.

A continuación se muestra la parte principal de las clases implementadas en el código de la máquina abstracta orientada a objetos y persistente.

```
Persistent Class TBDArchivo
Aggregation
  Persistent Archivos:Array;
…
EndClass
```

```
Persistent Class
TsetOfAuthors
Aggregation
  Persistent Autores:Array;
…
EndClass
```

```
Persistent Class TArchivo
Aggregation
  Persistent Nombre:String;
…
EndClass

Persistent Class TLibro
Isa TArchivo;
Aggregation
  Persistent NombreAutor,
          Editorial:String;
Association
  Autor:TAutor;
…
EndClass

Persistent Class TAutor
Aggregation
  Persistent Nombre:String;
  Persistent AnioNacimiento,
         AnioMuerte:Integer;
…
EndClass

Persistent Class TRevista
Isa TArchivo;
Aggregation
  Persistent Numero:Integer;
…
EndClass
```

C.1 Arrays persistentes

La parte fundamental del esquema representado, es la identificación de las bases de datos como arrays persistentes. Los elementos de un array son de un tipo genérico de clase de forma que se puedan describir una serie de métodos polimórficos. Cada vez que se quiera visualizar todos sus elementos, se llama al método Write que todas las clases deberán implementar. De la misma forma, se podrán identificar los objetos de tipo archivo mediante la implementación de los métodos EsLibro o EsRevista que identifiquen el tipo de un objeto. De forma genérica, el tipo de cada objeto se puede determinar mediante la llamada al método getClass de la clase Object (toda clase es hija directa o indirectamente de esta clase).

C.2 Asociaciones

El hecho de establecer la relación entre TLibro y TAutor mediante un enlace de asociación, hace que las consultas cruzadas de libros por autores sean muy rápidas al utilizarse el atributo asociado Autor. Así tenemos algo parecido a las relaciones de las bases de datos relacionales. Estas asociaciones han de ser mantenidas por el usuario y para ello identificamos el atributo agregado persistente NombreAutor.

El hecho de que estas relaciones de asociación deban mantenerse por el usuario, está enlazado con el concepto de asociación. Este concepto establece una relación entre una serie de objetos pero esta relación es externa y por lo tanto se puede romper. Para cada objeto persistente, se deberá guardar la identificación o clave primaria de sus objetos asociados que pueden o no ser persistentes. Estas claves primarias, se pueden almacenar como atributos agregados persistentes.

Una vez recuperado el objeto persistente, se deberá llamar a un método que identifique los elementos asociados y establezca el enlace a partir de las claves primarias agregadas.

La identificación de estas funciones en la clase TLibro se muestra en la siguiente tabla:

Clase:	TLibro
Elemento asociado:	Autor (tipo TAutor)
Clave primaria agregada:	NombreAutor
Método que establece el enlace:	En el ejemplo no se ha implementado puesto que sólo se necesita el autor de un libro para imprimirlo. Esto se obtiene directamente de NombreAutor.

Vemos pues cómo se puede realizar una base de datos orientada a objetos muy simple mediante el lenguaje Carbayón. Todo usuario que deba mantener una base de datos orientada a objetos, deberá pues gestionar una serie de factores comunes a otras aplicaciones similares.

Esta funcionalidad común puede proporcionarse mediante un subsistema de gestión de bases de datos, que utilice un motor de bases de datos adecuados que complemente el sistema de persistencia con mecanismos auxiliares para acelerar consultas, mantenimiento de la integridad de las asociaciones, etc.

C.3 ESPECIFICACIÓN DE LOS MÉTODOS DE LAS CLASES DEL EJEMPLO

C.3.1 TBDArchivo

AutorDeLibro():String

Para todos los archivos del array persistente, visualiza aquellos que sean libros. Permite elegir uno de ellos, y devuelve el nombre del autor asociado (clave primaria del objeto autor).

Write()

Recorre todos los archivos del array persistente y los visualiza ya sean libros o revistas. Para visualizarlos llama a su método genérico Write que la clase TLibro y TRevista implementan de forma particular. Este es el ejemplo mas claro de utilización del polimorfismo.

WriteLibrosDeAutor(Autor:String)

Recorre todos los libros del array persistente de archivos, e imprime todos aquellos que estén asociados con el autor cuyo nombre sea el pasado como parámetro.

WriteLibros()

Visualiza sólo los archivos que sean libros llamando a su método Write.

WriteRevistas()

Visualiza sólo los archivos que sean libros llamando a su método Write.

CrearArchivos()

Método que crea una base de datos de archivos de ejemplo. Crea dos libros de Cervantes, cuatro de Machado y tres de Unamuno. Además crea dos revistas de PC y una de Motocicletas.

AddLibro(AutorString)

Crea un objeto vacío TLibro. Le asocia el autor pasado como parámetro y lo introduce al final del array persistente.

AddRevista()

Crea un objeto vacío TRevista y lo introduce al final del array persistente.

C.3.2 TBDAutor

CrearAutores()

Crea una base de datos ejemplo con tres autores (Machado, Unamuno y Cervantes). Completa sus datos y los introduce en el array de Autores.

Write()

Recorre el array persistente de autores, llamando a sus métodos Write que visualizan las características de éstos.

WriteAutor(AutorParam:String)

Recorre el array para visualizar tan solo el autor cuyo nombre (clave primaria) sea el pasado como parámetro. Para ello, una vez seleccionado, se limita a llamar a su método Write.

SeleccionarAutor():String

Visualiza los autores, y permite seleccionar uno de ellos. Su nombre se devuelve como un String.

AddAutor()

Crea un objeto vacío de tipo TAutor y lo añade al final del array persistente.

C.3.3 TArchivo

SetNombre(NombreParam:String)

Actualiza el atributo Nombre al pasado como parámetro. Si el archivo es un libro, representa su título y si es una revista su nombre.

GetNombre():String

Devuelve el valor de atributo Nombre de la revista o libro.

C.3.4 TLibro

Write()

Método que se implementa de forma particular en libros y revistas para poderlo utilizar de forma genérica en archivos (polimorfismo). Visualiza los datos del libro: Título, Autor y Editorial.

Read()

Pide por consola los tres datos del libro. Actualiza así sus tres atributos.

SetNombreAutor(NombreParam:String)

Actualiza el atributo NombreAutor al pasado como parámetro.

GetNombreAutor():String

Devuelve el nombre del autor asociado.

SetEditorial(NombreParam:String)

Actualiza el atributo Editorial al pasado como parámetro.

EsLibro():Bool

Devuelve un objeto booleano para saber si un archivo es un libro. Este objeto será siempre cierto.

EsRevista():Bool

Devuelve un objeto booleano para saber si un archivo es una revista. Este objeto será siempre falso.

C.3.5 TRevista

Write()

Método que se implementa de forma particular en libros y revistas para poderlo utilizar de forma genérica en archivos (polimorfismo). Visualiza los datos de la revista: Su nombre y su número.

Read()

Pide por consola los dos datos de la revista. Actualiza así sus dos atributos.

SetNumero(NumeroParam:Integer)

Actualiza el valor del atributo Numero al pasado como parámetro.

EsLibro():Bool

Devuelve un objeto booleano para saber si un archivo es un libro. Este objeto será siempre falso.

EsRevista():Bool

Devuelve un objeto booleano para saber si un archivo es una revista. Este objeto será siempre cierto.

C.3.6 TAutor

Write

Método que visualiza los datos de un autor de libros: Su nombre y años de nacimiento y defunción.

Read()

Pide por consola los dos datos del autor. Actualiza así sus tres atributos.

SetNombre(NombreParam:String)

Actualiza el valor del atributo Nombre al pasado como parámetro.

SetAnioNacimiento(Anio:Integer)

Actualiza el valor del atributo AnioNacimiento al pasado como parámetro.

SetAnioMuerte(Anio:Integer)

Actualiza el valor del atributo AnioMuerte al pasado como parámetro.

GetNombre():String

Devuelve el valor del atributo Nombre.

C.3.7 TMyApp

Menu()

Único método de la clase TMyApp que es el programa principal. Crea un objeto de tipo TDBArchivo y otro de tipo TDBAutor y ofrece una serie de operaciones mediante un menú:

Nº	Función	Código
1	Crear una Base de Datos modelo	BDAutores.CrearAutores(); BDArchivos.CrearArchivos();
2	Añadir un Autor	BDAutores.AddAutor();
3	Añadir un Libro	BDAutores.SeleccionarAutor():Cadena; BDArchivos.AddLibro(Cadena);
4	Añadir una Revista	BDArchivos.AddRevista();
5	Listar Libros	BDArchivos.WriteLibros();
6	Listar Revistas	BDArchivos.WriteRevistas();
7	Listar Archivos	BDArchivos.Write();
8	Listar Autores	BDAutores.Write();
9	Consultar Libros por Autor	BDAutores.SeleccionarAutor():Cadena; BDArchivos.WriteLibrosDeAutor(Cadena);
10	Consultar el Autor de un Libro	BDArchivos.AutorDeLibro():Cadena; BDAutores.WriteAutor(Cadena);

C.4 Código

C.4.1 MyApp

```
CLASS MyApp

METHODS

Run()
  CODE
  // Método principal que se limita a ejecutar un menu
  This.Menu();
  Exit;
  ENDCODE


Menu()
  REFS
    Cadena:String;
    Opcion:Integer;
    Booleano:Bool;
    BDAutores:TBDAutor;
    BDArchivos:TBDArchivo;
  INSTANCES
    PersistenciaAutores:String('AUTORES');
    PersistenciaArchivos:String('ARCHIVOS');
```

```
    Cero:Integer(0);
    Uno:Integer(1);
    Dos:Integer(2);
    Tres:Integer(3);
    Cuatro:Integer(4);
    Cinco:Integer(5);
    Seis:Integer(6);
    Siete:Integer(7);
    Ocho:Integer(8);
    Nueve:Integer(9);
    Diez:Integer(10);
    Once:Integer(11);
    Doce:Integer(12);
    Consola:ConStream;
    Msg:String('BASE DE DATOS ORIENTADA A OBJETOS SOBRE CARBAYONIA PERSISTENTE');
    MsgMenu:String('Seleccione una de las siguientes operaciones: ');
    MsgOpcion1:String('   1.- Crear una Base de Datos modelo.');
    MsgOpcion2:String('   2.- Anadir un Autor.');
    MsgOpcion3:String('   3.- Anadir un Libro.');
    MsgOpcion4:String('   4.- Anadir una Revista.');
    MsgOpcion5:String('   5.- Listar Libros.');
    MsgOpcion6:String('   6.- Listar Revistas.');
    MsgOpcion7:String('   7.- Listar Archivos.');
    MsgOpcion8:String('   8.- Listar Autores.');
    MsgOpcion9:String('   9.- Consultar Libros por Autor.');
    MsgOpcion10:String('  10.- Consultar el Autor de un Libro.');
    MsgOpcion11:String('  11.- Salir.');
    MsgPulsar:String('  Pulse una tecla y return para continuar ...');
    MsgSeleccionOpcion:String('  Opcion: ');
  CODE
    Persistence.Exists(PersistenciaAutores):Booleano;
    Jtd Booleano,MenuExistenAutores;
    new BDAutores;
    Persistence.Add(PersistenciaAutores,BDAutores);
    Jmp MenuArchivos;

  MenuExistenAutores:
    Persistence.GetObject(PersistenciaAutores):BDAutores;

  MenuArchivos:
    Persistence.Exists(PersistenciaArchivos):Booleano;
    Jtd Booleano,MenuExistenArchivos;
    new BDArchivos;
    Persistence.Add(PersistenciaArchivos,BDArchivos);
    Jmp MenuComienzo;

  MenuExistenArchivos:
    Persistence.GetObject(PersistenciaArchivos):BDArchivos;

  MenuComienzo:
    Consola.ClearScreen();
    Consola.NextLine();
    Consola.Write(Msg);
    Consola.NextLine();
    Consola.NextLine();
    Consola.NextLine();
    Consola.Write(MsgMenu);
    Consola.NextLine();
    Consola.NextLine();
    Consola.Write(MsgOpcion1);
    Consola.NextLine();
    Consola.Write(MsgOpcion2);
    Consola.NextLine();
    Consola.Write(MsgOpcion3);
    Consola.NextLine();
    Consola.Write(MsgOpcion4);
    Consola.NextLine();
    Consola.Write(MsgOpcion5);
    Consola.NextLine();
    Consola.Write(MsgOpcion6);
    Consola.NextLine();
    Consola.Write(MsgOpcion7);
    Consola.NextLine();
    Consola.Write(MsgOpcion8);
    Consola.NextLine();
    Consola.Write(MsgOpcion9);
    Consola.NextLine();
```

```
    Consola.Write(MsgOpcion10);
    Consola.NextLine();
    Consola.Write(MsgOpcion11);
    Consola.NextLine();
    Consola.NextLine();
    Consola.Write(MsgSeleccionOpcion);
    Consola.Read():Opcion;

    Opcion.Equal(Uno):Booleano;
    Jfd Booleano,MenuOpcion2;
    BDAutores.CrearAutores();
    BDArchivos.CrearArchivos();
    Jmp MenuPedirTecla;

MenuOpcion2:
    Opcion.Equal(Dos):Booleano;
    Jfd Booleano,MenuOpcion3;
    Consola.ClearScreen();
    BDAutores.AddAutor();
    Jmp MenuComienzo;

MenuOpcion3:
    Opcion.Equal(Tres):Booleano;
    Jfd Booleano,MenuOpcion4;
    Consola.ClearScreen();
    BDAutores.SeleccionarAutor():Cadena;
    Consola.NextLine();
    BDArchivos.AddLibro(Cadena);
    delete Cadena;
    Jmp MenuComienzo;

MenuOpcion4:
    Opcion.Equal(Cuatro):Booleano;
    Jfd Booleano,MenuOpcion5;
    Consola.ClearScreen();
    BDArchivos.AddRevista();
    Jmp MenuComienzo;

MenuOpcion5:
    Opcion.Equal(Cinco):Booleano;
    Jfd Booleano,MenuOpcion6;
    Consola.ClearScreen();
    BDArchivos.WriteLibros();
    Jmp MenuPedirTecla;

MenuOpcion6:
    Opcion.Equal(Seis):Booleano;
    Jfd Booleano,MenuOpcion7;
    Consola.ClearScreen();
    BDArchivos.WriteRevistas();
    Jmp MenuPedirTecla;

MenuOpcion7:
    Opcion.Equal(Siete):Booleano;
    Jfd Booleano,MenuOpcion8;
    Consola.ClearScreen();
    BDArchivos.Write();
    Jmp MenuPedirTecla;

MenuOpcion8:
    Opcion.Equal(Ocho):Booleano;
    Jfd Booleano,MenuOpcion9;
    Consola.ClearScreen();
    BDAutores.Write();
    Jmp MenuPedirTecla;

MenuOpcion9:
    Opcion.Equal(Nueve):Booleano;
    Jfd Booleano,MenuOpcion10;
    Consola.ClearScreen();
    BDAutores.SeleccionarAutor():Cadena;
    Consola.NextLine();
    BDArchivos.WriteLibrosDeAutor(Cadena);
    delete Cadena;
    Jmp MenuPedirTecla;

MenuOpcion10:
```

```
    Opcion.Equal(Diez):Booleano;
    Jfd Booleano,MenuOpcion11;
    Consola.ClearScreen();
    BDArchivos.AutorDeLibro():Cadena;
    Consola.NextLine;
    BDAutores.WriteAutor(Cadena);
    delete Cadena;
    Jmp MenuPedirTecla;

  MenuOpcion11:
    Opcion.Equal(Once):Booleano;
    Jtd Booleano,MenuFinal;
    Jmp MenuComienzo;

  MenuPedirTecla:
    Consola.NextLine();
    Consola.Write(MsgPulsar);
    Consola.Read():Cadena;
    delete Cadena;
    delete Opcion;
    Jmp MenuComienzo;

  MenuMostrarEnPantalla:
    Jmp MenuMostrarEnPantalla;

  MenuFinal:
    Exit;
  ENDCODE

ENDCLASS
```

C.4.2 TBDArchivo

```
Persistent CLASS TBDArchivo

AGGREGATION
  Persistent Archivos:Array;
  Consola:ConStream;

METHODS

AutorDeLibro():String
  REFS
    Tamanio:Integer;
    Booleano:Bool;
    Archivo:TArchivo;
    NombreLibro,NombreAutor:String;
    Opcion:Integer;
  INSTANCES
    Contador:Integer(0);
    Uno:Integer(1);
    Msg:String('Elija uno de los libros siguientes:');
    MsgOpcion:String('  Seleccione un libro: ');
    Guion:String('.- ');
    Espacio:String(' ');
    Numero:String;
  CODE
    Consola.Write(Msg);
    Consola.NextLine();
    Consola.NextLine();
    Archivos.GetSize():Tamanio;

  AutorDeLibroBucle:
    Contador.Less(Tamanio):Booleano;
    Jfd Booleano,AutorDeLibroFin;
    Archivos.GetRef(Contador):Archivo;
    Archivo.EsLibro():Booleano;
    Jfd Booleano,AutorDeLibroSeguir;
    Numero.SetI(Contador);
    Consola.Write(Espacio);
    Consola.Write(Espacio);
    Consola.Write(Numero);
    Consola.Write(Guion);
    Archivo.GetNombre():NombreLibro;
```

```
    Consola.Write(NombreLibro);
    delete NombreLibro;
    Consola.NextLine;

  AutorDeLibroSeguir:
    Contador.Add(Uno);
    Jmp AutorDeLibroBucle;

  AutorDeLibroFin:
    Consola.NextLine();
    Consola.Write(MsgOpcion);
    Consola.Read():Opcion;
    Archivos.GetRef(Opcion):Archivo;
    delete Opcion;
    Archivo.GetNombreAutor():NombreAutor;
    Assign rr,NombreAutor;

    delete Tamanio;
    Exit;
  ENDCODE

Write()
  REFS
    Tamanio:Integer;
    Booleano:Bool;
    Archivo:TArchivo;
  INSTANCES
    Contador:Integer(0);
    Uno:Integer(1);
    Msg:String('Lista de los archivos de la Base de Datos:');
  CODE
    Consola.Write(Msg);
    Consola.NextLine();
    Consola.NextLine();
    Archivos.GetSize():Tamanio;

  WriteBucle:
    Contador.Less(Tamanio):Booleano;
    Jfd Booleano,WriteFin;
    Archivos.GetRef(Contador):Archivo;
    Archivo.Write();
    Consola.NextLine;
    Contador.Add(Uno);
    Jmp WriteBucle;

  WriteFin:
    delete Tamanio;
    Exit;
  ENDCODE

WriteLibrosDeAutor(Autor:String)
  REFS
    Tamanio:Integer;
    Booleano:Bool;
    Archivo:TArchivo;
    Cadena:String;
  INSTANCES
    Contador:Integer(0);
    Uno:Integer(1);
    Msg:String('Lista de los Libros del autor seleccionado:');
  CODE
    Consola.Write(Msg);
    Consola.NextLine();
    Consola.NextLine();
    Archivos.GetSize():Tamanio;

  WriteLibrosDeAutorBucle:
    Contador.Less(Tamanio):Booleano;
    Jfd Booleano,WriteLibrosDeAutorFin;
    Archivos.GetRef(Contador):Archivo;
    Archivo.EsLibro():Booleano;
    Jfd Booleano,WriteLibrosDeAutorSiguiente;

    Archivo.GetNombreAutor():Cadena;
    Cadena.Equal(Autor):Booleano;
    delete Cadena;
    Jfd Booleano,WriteLibrosDeAutorSiguiente;
```

```
    Archivo.Write();
    Consola.NextLine;

  WriteLibrosDeAutorSiguiente:
    Contador.Add(Uno);
    Jmp WriteLibrosDeAutorBucle;

  WriteLibrosDeAutorFin:
    delete Tamanio;
    Exit;
  ENDCODE

WriteLibros()
  REFS
    Tamanio:Integer;
    Booleano:Bool;
    Archivo:TArchivo;
  INSTANCES
    Contador:Integer(0);
    Uno:Integer(1);
    Msg:String('Lista de los Libros de la Base de Datos:');
  CODE
    Consola.Write(Msg);
    Consola.NextLine();
    Consola.NextLine();
    Archivos.GetSize():Tamanio;

  WriteLibrosBucle:
    Contador.Less(Tamanio):Booleano;
    Jfd Booleano,WriteLibrosFin;
    Archivos.GetRef(Contador):Archivo;
    Archivo.EsLibro():Booleano;
    Jfd Booleano,WriteLibrosSiguiente;
    Archivo.Write();
    Consola.NextLine;

  WriteLibrosSiguiente:
    Contador.Add(Uno);
    Jmp WriteLibrosBucle;

  WriteLibrosFin:
    delete Tamanio;
    Exit;
  ENDCODE

WriteRevistas()
  REFS
    Tamanio:Integer;
    Booleano:Bool;
    Archivo:TArchivo;
  INSTANCES
    Contador:Integer(0);
    Uno:Integer(1);
    Msg:String('Lista de las Revistas de la Base de Datos:');
  CODE
    Consola.Write(Msg);
    Consola.NextLine();
    Consola.NextLine();
    Archivos.GetSize():Tamanio;

  WriteRevistasBucle:
    Contador.Less(Tamanio):Booleano;
    Jfd Booleano,WriteRevistasFin;
    Archivos.GetRef(Contador):Archivo;
    Archivo.EsRevista():Booleano;
    Jfd Booleano,WriteRevistasSiguiente;
    Archivo.Write();
    Consola.NextLine;

  WriteRevistasSiguiente:
    Contador.Add(Uno);
    Jmp WriteRevistasBucle;

  WriteRevistasFin:
    delete Tamanio;
    Exit;
  ENDCODE
```

```
CrearArchivos()
  REFS
    Unamuno1,Unamuno2,Unamuno3:TLibro;
    Cervantes1,Cervantes2:TLibro;
    Machado1,Machado2,Machado3,Machado4:TLibro;
    RevistaPC1,RevistaPC2,RevistaMoto1:TRevista;
  INSTANCES
    Cero:Integer(0);
    Uno:Integer(1);
    Dos:Integer(2);
    Tres:Integer(3);
    Cuatro:Integer(4);
    Cinco:Integer(5);
    Seis:Integer(6);
    Siete:Integer(7);
    Ocho:Integer(8);
    Nueve:Integer(9);
    Diez:Integer(10);
    Once:Integer(11);
    Doce:Integer(12);
    StrUnamuno:String('Miguel de Unamuno');
    StrCervantes:String('Miguel de Cervantes');
    StrMachado:String('Antonio Machado');
    StrUnamuno1:String('Tres Novelas Ejemplares y un Prologo.');
    StrUnamuno2:String('San Manuel Bueno, martir.');
    StrUnamuno3:String('Abel Sanchez.');
    StrCervantes1:String('Quijote.');
    StrCervantes2:String('Novelas Ejemplares.');
    StrMachado1:String('Abel Martin.');
    StrMachado2:String('Soledades.');
    StrMachado3:String('Campos de Castilla.');
    StrMachado4:String('Nuevas Canciones.');
    StrPCPlus:String('PC Plus');
    StrMotociclismo:String('Motociclismo');
  CODE
    Archivos.SetSize(Doce);

    new Unamuno1;
    Unamuno1.SetNombre(StrUnamuno1);
    Unamuno1.SetNombreAutor(StrUnamuno);
    Archivos.SetRef(Cero,Unamuno1);

    new Unamuno2;
    Unamuno2.SetNombre(StrUnamuno2);
    Unamuno2.SetNombreAutor(StrUnamuno);
    Archivos.SetRef(Uno,Unamuno2);

    new Unamuno3;
    Unamuno3.SetNombre(StrUnamuno3);
    Unamuno3.SetNombreAutor(StrUnamuno);
    Archivos.SetRef(Dos,Unamuno3);

    new Cervantes1;
    Cervantes1.SetNombre(StrCervantes1);
    Cervantes1.SetNombreAutor(StrCervantes);
    Archivos.SetRef(Tres,Cervantes1);

    new Cervantes2;
    Cervantes2.SetNombre(StrCervantes2);
    Cervantes2.SetNombreAutor(StrCervantes);
    Archivos.SetRef(Cuatro,Cervantes2);

    new Machado1;
    Machado1.SetNombre(StrMachado1);
    Machado1.SetNombreAutor(StrMachado);
    Archivos.SetRef(Cinco,Machado1);

    new Machado2;
    Machado2.SetNombre(StrMachado2);
    Machado2.SetNombreAutor(StrMachado);
    Archivos.SetRef(Seis,Machado2);

    new Machado3;
    Machado3.SetNombre(StrMachado3);
    Machado3.SetNombreAutor(StrMachado);
    Archivos.SetRef(Siete,Machado3);
```

```
    new Machado4;
    Machado4.SetNombre(StrMachado4);
    Machado4.SetNombreAutor(StrMachado);
    Archivos.SetRef(Ocho,Machado4);

    new RevistaPC1;
    RevistaPC1.SetNombre(StrPCPlus);
    RevistaPC1.SetNumero(Dos);
    Archivos.SetRef(Nueve,RevistaPC1);

    new RevistaPC2;
    RevistaPC2.SetNombre(StrPCPlus);
    RevistaPC2.SetNumero(Ocho);
    Archivos.SetRef(Diez,RevistaPC2);

    new RevistaMoto1;
    RevistaMoto1.SetNombre(StrMotociclismo);
    RevistaMoto1.SetNumero(Tres);
    Archivos.SetRef(Once,RevistaMoto1);

    Exit;
  ENDCODE

AddLibro(Autor:String)
  REFS
    Libro:TLibro;
    Tamanio:Integer;
  INSTANCES
    Uno:Integer(1);
  CODE
    new Libro;
    Libro.SetNombreAutor(Autor);
    Libro.Read();
    Archivos.GetSize():Tamanio;
    Tamanio.Add(Uno);
    Archivos.SetSize(Tamanio);
    Tamanio.Sub(Uno);
    Archivos.SetRef(Tamanio,Libro);
    delete Tamanio;
    Exit;
  ENDCODE


AddRevista()
  REFS
    Revista:TRevista;
    Tamanio:Integer;
  INSTANCES
    Uno:Integer(1);
  CODE
    new Revista;
    Revista.Read();
    Archivos.GetSize():Tamanio;
    Tamanio.Add(Uno);
    Archivos.SetSize(Tamanio);
    Tamanio.Sub(Uno);
    Archivos.SetRef(Tamanio,Revista);
    delete Tamanio;
    Exit;
  ENDCODE

ENDCLASS
```

C.4.3 TBDAutor

```
Persistent CLASS TBDAutor

AGGREGATION
  Persistent Autores:Array;
  Consola:ConStream;

METHODS
```

```
CrearAutores()
  REFS
    Unamuno,Cervantes,Machado:TAutor;
  INSTANCES
    Cero:Integer(0);
    Uno:Integer(1);
    Dos:Integer(2);
    Tres:Integer(3);
    NombreUnamuno:String('Miguel de Unamuno');
    NacimientoUnamuno:Integer(1864);
    MuerteUnamuno:Integer(1936);
    NombreCervantes:String('Miguel de Cervantes');
    NacimientoCervantes:Integer(1547);
    MuerteCervantes:Integer(1616);
    NombreMachado:String('Antonio Machado');
    NacimientoMachado:Integer(1875);
    MuerteMachado:Integer(1939);
  CODE
    Autores.SetSize(Tres);

    new Unamuno;
    Unamuno.SetNombre(NombreUnamuno);
    Unamuno.SetAnioNacimiento(NacimientoUnamuno);
    Unamuno.SetAnioMuerte(MuerteUnamuno);
    Autores.SetRef(Cero,Unamuno);

    new Cervantes;
    Cervantes.SetNombre(NombreCervantes);
    Cervantes.SetAnioNacimiento(NacimientoCervantes);
    Cervantes.SetAnioMuerte(MuerteCervantes);
    Autores.SetRef(Uno,Cervantes);

    new Machado;
    Machado.SetNombre(NombreMachado);
    Machado.SetAnioNacimiento(NacimientoMachado);
    Machado.SetAnioMuerte(MuerteMachado);
    Autores.SetRef(Dos,Machado);

    Exit;
  ENDCODE

Write()
  REFS
    Tamanio:Integer;
    Booleano:Bool;
    Autor:TAutor;
  INSTANCES
    Contador:Integer(0);
    Uno:Integer(1);
    Msg:String('Lista de los autores de la Base de Datos:');
  CODE
    Consola.Write(Msg);
    Consola.NextLine();
    Consola.NextLine();
    Autores.GetSize():Tamanio;

  WriteBucle:
    Contador.Less(Tamanio):Booleano;
    Jfd Booleano,WriteFin;
    Autores.GetRef(Contador):Autor;
    Autor.Write();
    Consola.NextLine;
    Contador.Add(Uno);
    Jmp WriteBucle;

  WriteFin:
    delete Tamanio;
    Exit;
  ENDCODE

WriteAutor(AutorParam:String)
  REFS
    Tamanio:Integer;
    Booleano:Bool;
    Autor:TAutor;
    Cadena:String;
  INSTANCES
```

```
    Contador:Integer(0);
    Uno:Integer(1);
  CODE
    Autores.GetSize():Tamanio;

  WriteAutorBucle:
    Contador.Less(Tamanio):Booleano;
    Jfd Booleano,WriteAutorFin;
    Autores.GetRef(Contador):Autor;
    Autor.GetNombre():Cadena;
    Cadena.Equal(AutorParam):Booleano;
    delete Cadena;
    Jfd Booleano,WriteAutorSigue;
    Autor.Write();
    Consola.NextLine;

  WriteAutorSigue:
    Contador.Add(Uno);
    Jmp WriteAutorBucle;

  WriteAutorFin:
    delete Tamanio;
    Exit;
  ENDCODE

SeleccionarAutor():String
  REFS
    Tamanio:Integer;
    Booleano:Bool;
    Autor:TAutor;
    NombreLibro,NombreAutor:String;
    Opcion:Integer;
  INSTANCES
    Contador:Integer(0);
    Uno:Integer(1);
    Msg:String('Elija uno de los autores siguientes:');
    MsgOpcion:String('  Seleccione un autor: ');
    Guion:String('.- ');
    Espacio:String(' ');
    Numero:String;
  CODE
    Consola.Write(Msg);
    Consola.NextLine();
    Consola.NextLine();
    Autores.GetSize():Tamanio;

  SeleccionarAutorBucle:
    Contador.Less(Tamanio):Booleano;
    Jfd Booleano,SeleccionarAutorFin;
    Autores.GetRef(Contador):Autor;
    Numero.SetI(Contador);
    Consola.Write(Espacio);
    Consola.Write(Espacio);
    Consola.Write(Numero);
    Consola.Write(Guion);
    Autor.GetNombre():NombreAutor;
    Consola.Write(NombreAutor);
    delete NombreAutor;
    Consola.NextLine;
    Contador.Add(Uno);
    Jmp SeleccionarAutorBucle;

  SeleccionarAutorFin:
    Consola.NextLine();
    Consola.Write(MsgOpcion);
    Consola.Read():Opcion;
    Autores.GetRef(Opcion):Autor;
    delete Opcion;
    Autor.GetNombre():NombreAutor;
    Assign rr,NombreAutor;

    delete Tamanio;
    Exit;
  ENDCODE

AddAutor()
  REFS
```

```
    Autor:TAutor;
    Tamanio:Integer;
  INSTANCES
    Uno:Integer(1);
  CODE
    new Autor;
    Autor.Read();
    Autores.GetSize():Tamanio;
    Tamanio.Add(Uno);
    Autores.SetSize(Tamanio);
    Tamanio.Sub(Uno);
    Autores.SetRef(Tamanio,Autor);
    delete Tamanio;
    Exit;
  ENDCODE

ENDCLASS
```

C.4.4 TArchivo

```
Persistent CLASS TArchivo

AGGREGATION
  Persistent Nombre:String;
  Consola:ConStream;

METHODS

SetNombre(NombreParam:String)
  CODE
    Nombre.Set(NombreParam);
    Exit;
  ENDCODE

GetNombre():String
  REFS
   Devolucion:String;
  CODE
    new Devolucion;
    Devolucion.Set(Nombre);
    Assign rr,Devolucion;
    Exit;
  ENDCODE

ENDCLASS
```

C.4.5 TLibro

```
Persistent CLASS TLibro

ISA TArchivo;

AGGREGATION
  Persistent NombreAutor,Editorial:String;
  Consola:ConStream;

ASSOCIATION
  Autor:TAutor;

METHODS

Write()
  INSTANCES
    MsgNombre:String('  Nombre del Libro: ');
    MsgAutor:String('  Autor: ');
    MsgEditorial:String('  Editorial: ');
  CODE
    Consola.Write(MsgNombre);
    Consola.Write(Nombre);
    Consola.NextLine();
```

```
    Consola.Write(MsgAutor);
    Consola.Write(NombreAutor);
    Consola.NextLine();

    Consola.Write(MsgEditorial);
    Consola.Write(Editorial);
    Consola.NextLine();

    Exit;
  ENDCODE

Read()
  REFS
    Cadena:String;
  INSTANCES
    Msg:String('Introduzca los siguientes datos del libro:');
    MsgNombre:String('  Nombre del Libro: ');
    MsgEditorial:String('  Editorial: ');
  CODE
    Consola.Write(Msg);
    Consola.NextLine();
    Consola.NextLine();

    Consola.Write(MsgNombre);
    Consola.Read():Cadena;
    Nombre.Set(Cadena);
    delete Cadena;

    Consola.Write(MsgEditorial);
    Consola.Read():Cadena;
    Editorial.Set(Cadena);
    delete Cadena;

    Consola.NextLine();
    Exit;
  ENDCODE

SetNombreAutor(NombreParam:String)
  CODE
    NombreAutor.Set(NombreParam);
    Exit;
  ENDCODE

GetNombreAutor():String
  REFS
    Devolucion:String;
  CODE
    new Devolucion;
    Devolucion.Set(NombreAutor);
    Assign rr,Devolucion;
    Exit;
  ENDCODE

SetEditorial(NombreParam:String)
  CODE
    Editorial.Set(NombreParam);
    Exit;
  ENDCODE

EsLibro():Bool
  REFS
    Booleano:Bool;
  CODE
    new Booleano;
    Booleano.SetTrue;
    Assign rr,Booleano;
    Exit;
  ENDCODE

EsRevista():Bool
  REFS
    Booleano:Bool;
  CODE
    new Booleano;
    Booleano.SetFalse;
    Assign rr,Booleano;
    Exit;
```

```
  ENDCODE

ENDCLASS
```

C.4.2 TRevista

```
Persistent CLASS TRevista

ISA TArchivo;

AGGREGATION
  Persistent Numero:Integer;
  Consola:ConStream;

METHODS

Write()
  INSTANCES
    MsgNombre:String('  Nombre de la Revista: ');
    MsgNumero:String('  Numero: ');
  CODE
    Consola.Write(MsgNombre);
    Consola.Write(Nombre);
    Consola.NextLine();

    Consola.Write(MsgNumero);
    Consola.Write(Numero);
    Consola.NextLine();

    Exit;
  ENDCODE

Read()
  REFS
    Cadena:String;
    Entero:Integer;
  INSTANCES
    Msg:String('Introduzca los siguientes datos de la revista:');
    MsgNombre:String('  Nombre de la Revista: ');
    MsgNumero:String('  Numero: ');
  CODE
    Consola.Write(Msg);
    Consola.NextLine();
    Consola.NextLine();

    Consola.Write(MsgNombre);
    Consola.Read():Cadena;
    Nombre.Set(Cadena);
    delete Cadena;

    Consola.Write(MsgNumero);
    Consola.Read():Entero;
    Numero.Set(Entero);
    delete Entero;

    Consola.NextLine();
    Exit;
  ENDCODE

SetNumero(NumeroParam:Integer)
  CODE
    Numero.Set(NumeroParam);
    Exit;
  ENDCODE

EsLibro():Bool
  REFS
    Booleano:Bool;
  CODE
    new Booleano;
    Booleano.SetFalse;
    Assign rr,Booleano;
    Exit;
  ENDCODE
```

```
EsRevista():Bool
  REFS
    Booleano:Bool;
  CODE
    new Booleano;
    Booleano.SetTrue;
    Assign rr,Booleano;
    Exit;
  ENDCODE

ENDCLASS
```

C.4.3 TAutor

```
Persistent CLASS TAutor

AGGREGATION
  Persistent Nombre:String;
  Persistent AnioNacimiento,AnioMuerte:Integer;
  Consola:ConStream;

METHODS

Write()
  INSTANCES
    MsgNombre:String('  Nombre Autor: ');
    MsgNacimiento:String('  Anio de nacimiento: ');
    MsgMuerte:String('  Anio de defuncion: ');
  CODE
    Consola.Write(MsgNombre);
    Consola.Write(Nombre);
    Consola.NextLine();

    Consola.Write(MsgNacimiento);
    Consola.Write(AnioNacimiento);
    Consola.NextLine();

    Consola.Write(MsgMuerte);
    Consola.Write(AnioMuerte);
    Consola.NextLine();

    Exit;
  ENDCODE

Read()
  REFS
    Entero:Integer;
    Cadena:String;
  INSTANCES
    Msg:String('Introduzca los siguientes datos del autor:');
    MsgNombre:String('  Nombre Autor: ');
    MsgNacimiento:String('  Anio de nacimiento: ');
    MsgMuerte:String('  Anio de defuncion: ');
  CODE
    Consola.Write(Msg);
    Consola.NextLine();
    Consola.NextLine();

    Consola.Write(MsgNombre);
    Consola.Read():Cadena;
    Nombre.Set(Cadena);
    delete Cadena;

    Consola.Write(MsgNacimiento);
    Consola.Read():Entero;
    AnioNacimiento.Set(Entero);
    delete Entero;

    Consola.Write(MsgMuerte);
    Consola.Read():Entero;
    AnioMuerte.Set(Entero);
    delete Entero;
```

```
    Consola.NextLine();
    Exit;
  ENDCODE

SetNombre(NombreParam:String)
  CODE
    Nombre.Set(NombreParam);
    Exit;
  ENDCODE

SetAnioNacimiento(Anio:Integer)
  CODE
    AnioNacimiento.Set(Anio);
    Exit;
  ENDCODE

SetAnioMuerte(Anio:Integer)
  CODE
    AnioMuerte.Set(Anio);
    Exit;
  ENDCODE

GetNombre():String
  REFS
    Devolucion:String;
  CODE
    new Devolucion;
    Devolucion.Set(Nombre);
    Assign rr,Devolucion;
    Exit;
  ENDCODE

ENDCLASS
```

Apéndice D

COMPARATIVA DE RENDIMIENTO CON LA MÁQUINA DE JAVA

Se realiza una pequeña comparativa con otra máquina abstracta para tener una idea del rendimiento del prototipo de la máquina abstracta Carbayonia. Por ser la más extendida actualmente, se utiliza la máquina virtual de Java, JVM, en concreto la versión del paquete de desarrollo JDK 1.0.2.

La máquina de Java es una máquina más madura con fines comerciales, y, por tanto, ha recibido un gran esfuerzo de desarrollo. El prototipo actual de la máquina Carbayonia tiene como objetivo principal una fácil comprensión, evolución y adición de características al mismo. Esto se refleja en un diseño OO muy claro, pero que necesita introducir una serie de niveles de indirección que facilitan la mantenibilidad a costa del rendimiento. En el diseño siempre se ha optado por la claridad frente a la eficiencia, ya que el interés se centra en comprobar el funcionamiento de la filosofía de la propia máquina abstracta.

Por ejemplo, cada instrucción produce un objeto en tiempo de ejecución con su propio método `Exec` que implementa el comportamiento de la instrucción. La ejecución de cada instrucción conlleva llamadas a muchos objetos constituyentes de la máquina: Se pide al hilo actual que devuelva el método actual, sobre el que se pide la instrucción siguiente y a su vez se manda ejecutar ésta. Estos pasos permiten gran claridad en el diseño y fácil modificación, pero podrían cortocircuitarse y realizarse mucho más rápido.

Es de esperar, por tanto, que el rendimiento de la máquina Carbayonia sea varios órdenes de magnitud menor que el de esta máquina comercial. También será relativamente sencillo aumentar grandemente el rendimiento de la misma mediante la eliminación de estas indirecciones. Una vez decidido totalmente el comportamiento de la máquina, en principio no existe nada en el diseño que impida aplicar la mayoría de las técnicas de optimización de máquinas abstractas (por ejemplo compilación dinámica [Höl95] y alcanzar un rendimiento similar al de otras máquinas.

D.1 Programas de prueba

Para estimar el rendimiento de las máquinas se desarrollaron programas que repiten cíclicamente tratamiento de objetos: operaciones de asignación de enteros en arrays, operaciones con enteros, creación de objetos y gestión de cadenas. La comparativa no es exhaustiva, simplemente se trata de obtener una idea aproximada del rendimiento de la máquina Carbayonia.

Se desarrollaron programas que tratan de ser equivalentes para ambas máquinas. En el caso de la máquina de Java, en el propio lenguaje Java, que se compiló con el compilador del JDK. Para reflejar el funcionamiento de aplicaciones orientadas a objetos, que utilizan únicamente objetos, sólo se mide la parte del uso de objetos en la máquina de Java.

Debido a no existir aún soporte para cronómetros en Carbayonia, las mediciones no pueden ser muy exactas, e incluyen el tiempo de carga y establecimiento del simulador de

cada máquina. Por otro lado, para eliminar desplazamientos producidos por la velocidad de ejecución de los bucles, se elimina de las cifras el tiempo consumido sólo para los bucles.

Todos los programas en Carbayonia utilizan una clase MyApp con un método RUN() que llama a un método PRUEBA(). En cada programa este método implementa el cuerpo de cada comparativa:

```
CLASS MyApp

METHODS

RUN()
CODE
      this.prueba;
      exit;
ENDCODE
```

Los programas utilizados en la comparativa son los siguientes:

ARRAY – Creación de un array de enteros, escritura y lectura del mismo

```
PRUEBA()
Refs
      b:bool;
      num:integer;

Instances
      cero:integer(0);
      uno:integer(1);
      size:integer(10000);

      i:integer;
      arr:array;
      iteracion:integer;
      iteraciones:integer(10);
CODE
      arr.setsize(size);
      iteracion.set(cero);
      new num;

bucle0:
      iteracion.less(iteraciones):b;
      jfd b, finalizar;

      i.set(cero);
bucle1:
      i.less(size):b;
      jfd b, finWrite;

      arr.setRef(i, num);

      i.add(uno);
      jmp bucle1;

finWrite:
      i.set(cero);
bucle2:
      i.less(size):b;
      jfd b, final;

      arr.getRef(i):num;

      i.add(uno);
      jmp bucle2;
```

```
class array {

static final int LEN = 10000;

public static void main(String[] args)
{

      Integer num = new Integer(0);
      Integer[] array = new Integer[LEN];

        for (int i=0; i<10; i++) {

                for (int j=0; j<LEN; j++)
                        array[j] = num;

                for (int j=0; j<LEN; j++)
                        num = array[j];
        }
}
```

```
final:
      iteracion.add(uno);
      jmp bucle0;

finalizar:
      exit;

EndCode

ENDCLASS
```

INTEGER – operaciones básicas con enteros

```
PRUEBA()
refs
      b:bool;

instances
      i:integer;
      iteracion:integer(0);
      iteraciones:integer(100000);
      uno:integer(1);
      tres:integer(3);
code

bucle:
      iteracion.less(iteraciones):b;
      jfd b, final;

      i.set(tres);
      i.add(tres);
      i.sub(tres);
      i.mul(tres);
      i.div(tres);

      iteracion.add(uno);
      jmp bucle;
final:
      exit;
endcode

ENDCLASS
```

```
class integer {


public static void main(String[] args)
{

        Integer i,
        numero = new Integer(3);

        for (int iteracion = 0; iteracion <
1000000; iteracion++) {
        i = numero;
        i = new Integer(i.intValue() + 3);
        i = new Integer(i.intValue() - 3);
        i = new Integer(i.intValue() * 3);
        i = new Integer(i.intValue() / 3);
        }

}

}
```

NEW – creación de objetos

```
PRUEBA()
Refs
      b:bool;
      num:integer;

Instances
      cero:integer(0);
      uno:integer(1);

      iteracion:integer;
      iteraciones:integer(5000);
CODE
      iteracion.set(cero);

bucle0:
      iteracion.less(iteraciones):b;
      jfd b, finalizar;

      new num;

      iteracion.add(uno);
      jmp bucle0;

finalizar:
      exit;
```

```
class New {


public static void main(String[] args) {

        for (int i=0; i<5000; i++)
                new Integer(0);


}


}
```

```
EndCode

ENDCLASS
```

STRING – operaciones básicas con cadenas

```
PRUEBA()
refs
      fin:bool;
      i:integer;

instances
      a:string;
      b:string('abcd a todos');

      uno:integer(1);
      cero:integer(0);
      equis:integer(88);
      be:string('b');

      iter:integer;
      iteraciones:integer(5000);

code
      iter.set(cero);

bucle:
      iter.less(iteraciones):fin;
      jfd fin, final;

   a.set(b);     // a = 'abcd a todos'
   a.setchar(uno, equis);
 // Poner una X en el segundo caracter
   a.remove(uno,uno);   // quitar la X
   a.getChar(uno):i;    // Coge la 'c'
a.insert(be, uno);//a = "abcd a todos"

      iter.add(uno);
      jmp bucle;

final:
      exit;

endcode

ENDCLASS
```

```
class string {

public static void main(String[] args)
{
        String a, b = "abcd a todos";

        for (int i=0; i< 5000; i++) {
                a = b;

                StringBuffer sb = new
                           StringBuffer(a);
                sb.setCharAt(1, 'X');
                a = new String(sb);

                a = a.substring(0,1) +
                            a.substring(2);

                char c = a.charAt(1);

                sb = new StringBuffer(a);
                sb.insert(1, "b");
                a = new String(sb);
        }
}
}
```

D.2 Resultados

Los resultados de la comparativa con los programas anteriores se resumen en la tabla siguiente. Se refleja la velocidad relativa de la máquina de Java frente al prototipo de la máquina Carbayonia, además de los órdenes de magnitud de esa velocidad relativa.

	Velocidad relativa de la máquina de Java	Orden de magnitud de la velocidad relativa la máquina de Java
ARRAY	123,5	2
INTEGER	12,7	1
NEW	115,8	2
STRING	27,9	1

Los resultados son los esperados, e incluso mejores. La máquina de Java es claramente más rápida que este prototipo, entre uno y dos órdenes de magnitud más rápida, dependiendo de las operaciones.

La menor diferencia se da en las operaciones con enteros y cadenas de caracteres. En el caso de los enteros, la máquina de Java tiene el inconveniente de no soportar operaciones aritméticas con objetos de tipo entero, lo que obliga a convertirlos al tipo de datos básico `int` que es el que tiene las operaciones. Esto provoca una cierta ralentización.

Para las cadenas de caracteres, la diferencia en parte es debida a la consideración de las cadenas como inmutables en Java, con lo que ciertas operaciones implican una copia (más lenta) de las cadenas[1].

Una situación más cercana a lo esperado es la que reflejan las operaciones con el array y la creación de objetos. La diferencia es de dos órdenes de magnitud (aproximadamente 120 veces más rápido), que refleja más fielmente la pérdida de velocidad que producen las diferentes indirecciones introducidas en el prototipo para facilitar su modificación y extensión.

Teniendo en cuenta esto, es razonable pensar que el rendimiento de la máquina Carbayonia, una vez realizadas optimizaciones en su implementación, sea similar al de otras máquinas abstractas, como la de Java.

[1] Aunque en algún paso se utiliza la clase `StringBuffer` que ofrece una funcionalidad más parecida a las cadenas de Carbayonia.

Apéndice E

COMPARATIVA DE RENDIMIENTO DE LA MÁQUINA CON PERSISTENCIA

Para comparar el rendimiento del prototipo de la máquina abstracta con persistencia, se realizaron una serie de pruebas. Lo esperado es que el rendimiento de la máquina con persistencia sea similar al de la máquina anterior. Esto debería ser así puesto que el diseño de la máquina persistente se ha realizado como un complemento al simulador existente, que prácticamente no sufre modificaciones (véase el capítulo 18).

Para realizar una comparación lo más adecuada posible, se intenta utilizar versiones de ambas máquinas compiladas de manera equivalente, con las mismas opciones de compilación cuando fue posible. En ambas versiones se utilizó una librería de depuración que ralentiza mucho las ejecuciones, por lo que las cifras absolutas obtenidas no son representativas de la velocidad real de las máquinas.

E.1 Programa de prueba

El programa de prueba es un programa iterativo muy sencillo. El cuerpo del programa simplemente establece el tamaño de un array de objetos, lo inicializa, recorre los elementos visualizándolos y luego elimina el array. Puede variarse el número de objetos en el array y el número de iteraciones que se realizan de la operación.

Creación del array.

```
  Consola.Read():Numero;
  Objeto.SetSize(Numero);
```

Asignación de objetos.

```
MainBucle:
  Contador.Less(Numero):Booleano;
  Jfd Booleano,MainFinBucle;
  Objeto.SetRef(Contador,Entero);
  Contador.Add(Uno);
  Jmp MainBucle;
MainFinBucle:
```

Visualización.

```
  Contador.Set(Cero);
MainVisualiza:
  Contador.Less(Numero):Booleano;
  Jfd Booleano,MainFinVisualiza;
  Objeto.GetRef(Contador):Entero;
  Consola.Write(Entero);
  Consola.Write(MsgPunto);
  Contador.Add(Uno);
  Jmp MainVisualiza;
```

Eliminación.

```
  Contador.Set(Cero);
MainBorra:
  Contador.Less(Numero):Booleano;
  Jfd Booleano,MainFinBorra;
  Objeto.GetRef(Contador):Entero;
  delete Entero;
  Contador.Add(Uno);
  Jmp MainBorra;
MainFinBorra:
```

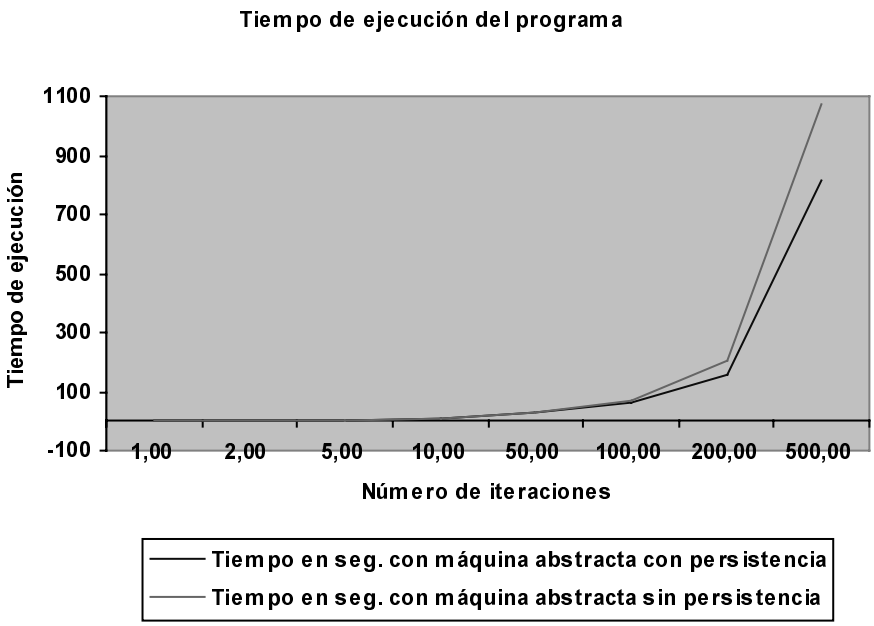
Se mide el tiempo transcurrido desde que se lanza el simulador hasta que este finaliza. Esto hace que se incluya el tiempo de carga del simulador y el de inicialización interna del mismo (**tiempo de establecimiento**), además del propio tiempo de ejecución del programa. Hay que tener en cuenta que en una situación de funcionamiento continuado real como entorno de computación, este tiempo de establecimiento no es importante, ya que el sistema continúa funcionando mucho tiempo sin detenerse.

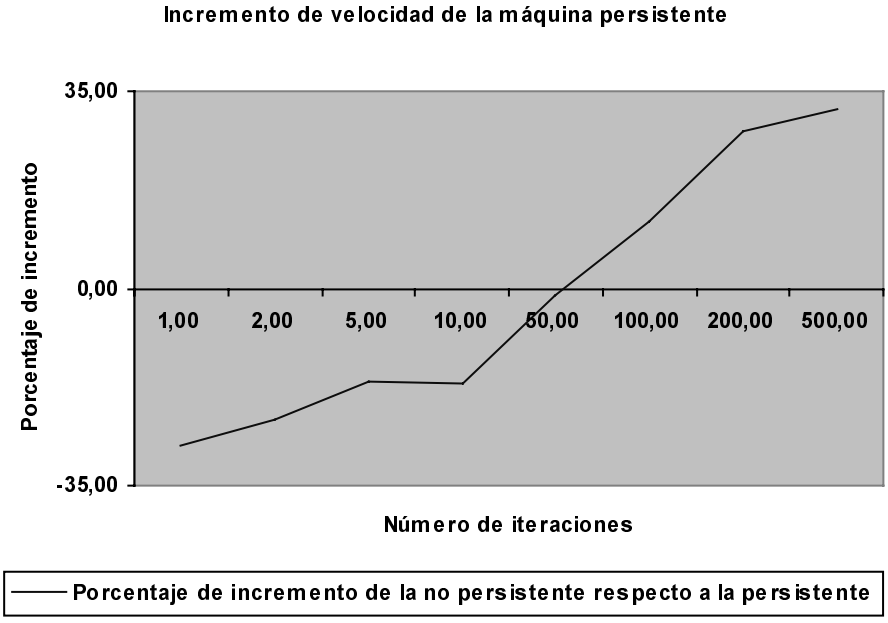
E.2 Comportamiento frente a la máquina anterior no persistente

Para comparara la máquina persistente con la no persistente anterior, se ejecutó el mismo programa en ambas máquinas, únicamente usando objetos temporales[1] en el array.

Número de elementos	Tiempo en seg. con máquina abstracta con persistencia	Tiempo en seg. con máquina abstracta sin persistencia	Porcentaje de incremento de la no-persistente respecto a la persistente
1	2,37	1,71	-27,8 %
2	2,76	2,12	-23,1 %
5	3,94	3,28	-16,7 %
10	6,29	5,23	-16,8 %
50	26,74	27,06	-1,19 %
100	62,13	69,63	12,07 %
200	158	202	27,84 %
500	813	1071	31,73 %

[1] Todas las menciones a objetos temporales y persistentes son a objetos de usuario del programa en Carbayón. No son nunca los objetos internos de la implementación.





En un comienzo la ejecución del programa es más rápida en la máquina sin persistencia que en la que la posee. Esto es debido al efecto del tiempo de establecimiento superior que tiene la máquina con persistencia, pues debe inicializar y cargar la información de los objetos adicionales del sistema de persistencia.

Conforme aumenta el tiempo de computación de objetos temporales, el tiempo de establecimiento del sistema de persistencia influye de forma menor, observándose que llega a ser más rápida la máquina persistente para un número de elementos elevado. Cuando el tiempo de computación hace que la carga del tiempo de establecimiento sea despreciable (para 200 y 500 elementos), la ejecución de la máquina persistente tiende a ser un 32% más rápida que la inicial.

Esta mayor velocidad de la máquina persistente puede ser debida a pequeñas optimizaciones obvias que se realizaron en la versión persistente. También es posible que existiera alguna diferencia en las opciones de compilación.

En cualquier caso, estos resultados confirman la hipótesis de que el sistema de persistencia no reduce de ninguna manera el rendimiento del simulador. La ejecución de los programas que sólo utilicen elementos temporales básicamente es igual que en la máquina anterior.

E.3 Comportamiento usando objetos persistentes frente a temporales

Se trata ahora de examinar cómo afecta en el rendimiento de la máquina la utilización de objetos persistentes. Para ello se ejecutó el mismo programa en la máquina persistente, usando en un caso únicamente objetos temporales en el array, y siendo en el otro caso todos objetos persistentes.

Tiempos de ejecución con objetos temporales

		Elementos				
		1	5	10	100	500
Iteraciones	1	2,34	3,19	4,21	22,58	105
	10	5,15	9,56	15,14	114	554
	100	32,47	71,07	125	1022	5072

Tiempos de ejecución con objetos persistentes

		Elementos				
		1	5	10	100	500
Iteraciones	1	2,58	3,4	5,4	48,31	623
	10	5,26	9,92	16,39	228	558
	100	33,01	73,51	133	98,82	538

Incrementos de tiempo en ejecución de los objetos persistentes respecto a los temporales

		Elementos				
		1	5	10	100	500
Iteraciones	1	-10,25%	-6,58%	-28,26%	-114%	-593%
	10	-2,13%	-3,76%	-8,25%	-100%	0%
	100	-1,66%	-3,43%	-6,4%	98,82%	942%

Cuando el tiempo de computación es pequeño (pocos objetos y pocas iteraciones), la ejecución con objetos temporales es algo más rápida que con objetos persistentes. Esto es debido al tiempo de establecimiento del sistema de persistencia aumenta con el número de objetos persistentes existentes (puesto que se carga siempre en el inicio todo el directorio de objetos persistentes[1] y hay que cargar los propios objetos[2]).

Sin embargo, cuando el tiempo de computación es suficientemente grande para que el tiempo de establecimiento sea despreciable (un número de elementos y de iteraciones grande), la ejecución con objetos persistentes es mucho más rápida que con objetos temporales.

En principio no debería haber mucha diferencia entre estas ejecuciones, excepto el simple establecimiento del sistema de persistencia. De hecho se observa este comportamiento cuando el número de iteraciones produce poco tiempo de ejecución y el número de objetos persistentes no es grande (y tarda poco en cargarse el directorio).

La explicación de la mayor velocidad con objetos persistentes cuando el número de objetos es muy grande se debe a la forma en la que se localiza un objeto internamente en la máquina. En la implementación actual, existen dos tablas diferentes, una para localizar las instancias temporales y otra para las instancias que son persistentes. Esta última es nueva y la implementación es diferente y utiliza una técnica más rápida. Así, la búsqueda en la tabla de objetos temporales es mucho más lenta cuando hay muchos que en la de objetos persistentes.

En caso de utilizar una misma tabla común con el mismo procedimiento de búsqueda, los resultados (como es de esperar) deberían ser iguales con objetos temporales y persistentes (salvando el tiempo de establecimiento), puesto que en la ejecución interna no existen diferencias en el funcionamiento de los mismos.

E.4 Comportamiento con intercambio al almacenamiento secundario

En el caso anterior, aunque se usaron objetos persistentes, toda la información tenía cabida en memoria principal, con lo que no se necesitó realizar intercambio de páginas con el almacenamiento secundario.

Para estimar el efecto del sistema de paginación, se estudió el caso anterior, para un número y tipo de elementos que necesitan alojarse en varias páginas de memoria virtual del sistema de persistencia. En este caso sólo se mide el tiempo de computación, es decir, no se tienen en cuenta los tiempos de establecimiento (y finalización) del sistema de persistencia.

Tiempos de ejecución en un entorno Windows NT 4.0 con condiciones normales

[1] Una mejora posible es cargar la información del directorio bajo demanda, en lugar de cargarla toda siempre.
[2] Sin embargo, la carga (y descarga) de los objetos persistentes es transparente y sí se hace bajo demanda.

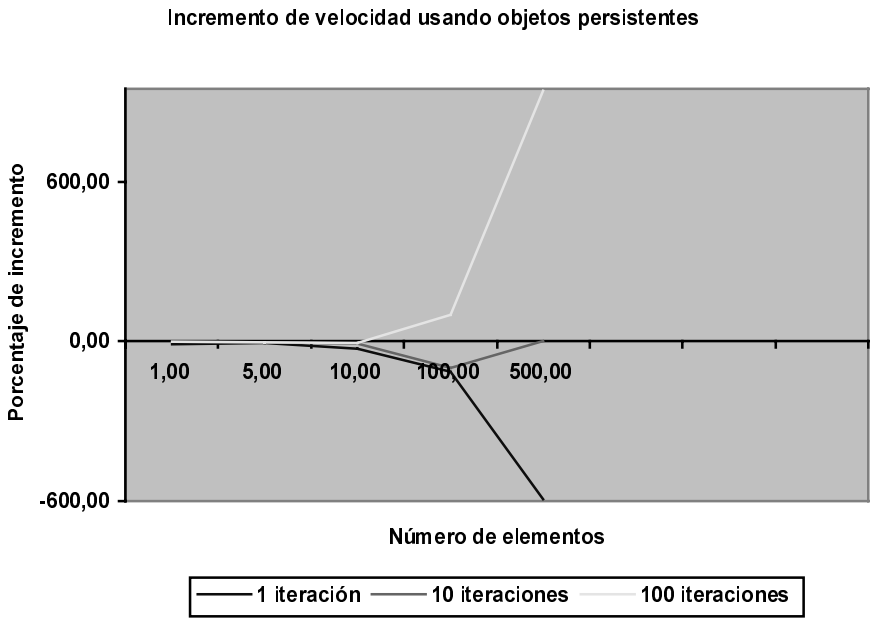
Iteraciones	Elementos	Páginas	Bytes de archivo de intercambio	Medición con objetos temporales	Medición con objetos persistentes
2	42	1	64Kb	22	10
2	84	2	128Kb	70	21
2	210	5	320Kb	395	88

Hay que tener en cuenta que en la implementación actual el tamaño de la memoria intermedia (número de marcos de página) para el sistema de paginación es dinámico. Cuando la memoria libre es mayor, se pueden alojar simultáneamente en memoria un número mayor de páginas del sistema de persistencia.

Tiempos de ejecución en un entorno Windows NT 4.0 con escasez de memoria libre

Los siguientes datos están tomados de la ejecución para 5 páginas en un entorno en el que el sistema estaba colapsado por la escasez de memoria RAM libre. De esta manera se provoca un mayor intercambio de páginas del sistema de persistencia con el disco puesto que no podrán estar alojadas todas simultáneamente en memoria por la escasez de la misma.

Iteraciones	Elementos	Páginas	Bytes de archivo de intercambio	Medición con objetos temporales	Medición con objetos persistentes
2	210	5	320Kb	548	298



En unas condiciones más desahogadas y cercanas al caso anterior en que todos los elementos caben en memoria, se sigue observando el mismo comportamiento. La ejecución con objetos persistentes es más rápida debido a que la localización de instancias persistentes está más optimizada que la de instancias locales. El efecto se acrecienta a medida que el

número de instancias a gestionar es mayor. En este caso el incremento de velocidad con objetos persistentes es un 448%.

En condiciones de poca memoria libre y, por tanto, pocas páginas del sistema de persistencia en memoria, el acceso a los objetos persistentes obliga a estar realizando continuamente intercambio de páginas con el almacenamiento secundario. Para dar una idea del impacto de este intercambio, puede compararse la diferencia de velocidad frente a objetos temporales. En este caso se reduce a un 84% de ganancia.

Conclusión

Puede concluirse, como estimación inicial, que la introducción continuada del intercambio a disco produce una disminución del rendimiento de aproximadamente un 338% en la ejecución con objetos persistentes. Es decir, el intercambio continuado a disco hace funcionar la máquina tres veces más despacio que cuando este no es continuado. Esta sería la sobrecarga que conlleva la utilización de un área de instancias virtual de tamaño superior a la memoria principal. Sin embargo, se necesitan más pruebas con una mayor variedad de situaciones para obtener una cifra más concluyente, con mayor número de páginas y objetos, otros programas, etc.

En cualquier caso, esta sobrecarga que produce la memoria virtual por el intercambio depende de muchos factores: el número de marcos de página disponibles en la memoria intermedia (tamaño de la memoria intermedia), la política de emplazamiento y reemplazamiento, el tamaño de la página, y sobre todo, el propio patrón de acceso a las páginas (objetos persistentes) de las aplicaciones. Este es un campo tradicional de los sistemas de memoria virtual de los sistemas operativos [Dei90] y podrían aplicarse todas las técnicas desarrolladas para los mismos.

Apéndice F

REPERTORIO DE INSTRUCCIONES DE LA MÁQUINA ABSTRACTA CARBAYONIA

A continuación se da una relación del repertorio de instrucciones disponible en Carbayonia ordenado alfabéticamente.

Assign destino, origen

Asignación de referencias

Al finalizar la instrucción la referencia destino apuntará al mismo objeto que la referencia origen.

En el proceso de asignación se produce automáticamente una conversión para amoldar el tipo estático del objeto al tipo de la referencia destino. Por tanto puede ser necesario tanto una conversión descendente como ascendente.

Se produce una excepción en los siguientes casos:

- El objeto al que apunta la referencia origen no es compatible con la referencia destino. Esto ocurre cuando el objeto pertenece a una clase la cual no deriva de la clase de la referencia destino.

Formato	Ejemplos
Assign {ámbito::}destino, {ámbito::}origen	**Assign** rDestino, rOrigen **Assign** refA, B::A::ref

DELETE referencia

Destrucción de una instancia

Elimina la instancia a la que esté apuntando la referencia <referencia>.

Dado que la referencia puede estar apuntando solamente a un bloque del objeto correspondiente a una clase base del mismo, hay que tener en cuenta el tipo estático del objeto para eliminarlo completamente.

Se produce una excepción en los siguientes casos:

- La referencia está libre.
- La referencia apunta a un objeto no existente (ya liberado mediante otra referencia).

Formato	Ejemplos
Delete {ámbito::}ref	**Delete** rDato **Delete** Base1::refAncestro

EXIT

Salida del método actual

Retorna al método anterior en la secuencia de llamadas.

Libera las instancias locales del método y descarta todos los manejadores de excepciones declarados en el mismo.

Formato	Ejemplos
Exit	**Exit**

HANDLER dirección

Activación de un manejador de excepciones

Handler declara la dirección que le acompaña como un manejador de excepciones, de manera que el flujo de control se bifurque a dicho lugar en el caso de que se produzca una excepción.

La dirección declarada en el manejador tiene prioridad sobre todas las demás que se hubiesen declarado anteriormente a ella.

Los manejadores se utilizan cuando se produce una excepción o bien se descartan mediante Exit cuando se sale del método en que fue declarada sin que se produjese una excepción.

Formato	Ejemplos
Handler <etiqueta>	**Handler** atrapa atrapa: <gestion de la excepción>

JF bool, destino

Salto condicional

El MC (*method counter*, contador de método) que indica la siguiente instrucción a ejecutar del hilo, pasa a apuntar a la instrucción que marca la etiqueta si el estado del objeto de tipo Bool del primer parámetro es falso.

Formato	Ejemplos
JF {ámbito::}ref, <etiqueta>	**JF** condición, finBucle

JFD bool, destino

Salto condicional con liberación de instancia

El MC (*method counter*, contador de método) que indica la siguiente instrucción a ejecutar del hilo, pasa a apuntar a la instrucción que marca la etiqueta si el estado del objeto de tipo Bool del primer parámetro es falso.

Seguidamente libera el objeto consultado. Véase Delete.

Formato	Ejemplos
JFD {ámbito::}ref, <etiqueta>	**JFD** condición, finBucle

JMP destino

Salto incondicional

El MC (*method counter*, contador de método) que indica la siguiente instrucción a ejecutar del hilo, pasa a apuntar a la instrucción que marca la etiqueta.

Formato	Ejemplos
Jmp <etiqueta>	**Jmp** finBucle

JNULL referencia, destino

Salto incondicional

El MC (*method counter*, contador de método) que indica la siguiente instrucción a ejecutar del hilo, pasa a apuntar a la instrucción que marca la etiqueta destino, si la referencia situada como primer parámetro está libre (no apunta a ningún objeto).

Formato	Ejemplos
JNull {ámbito::}ref, <etiqueta>	**JNull** refData, finBucle

JNNULL referencia, destino

Salto incondicional

El MC (*method counter*, contador de método) que indica la siguiente instrucción a ejecutar del hilo, pasa a apuntar a la instrucción que marca la etiqueta destino, si la referencia situada como primer parámetro no está libre (apunta a un objeto).

Formato	Ejemplos
JNNull {ámbito::}ref, <etiqueta>	**JNNull** refData, finBucle

JT bool, destino

Salto condicional

El MC (*method counter*, contador de método) que indica la siguiente instrucción a ejecutar del hilo, pasa a apuntar a la instrucción que marca la etiqueta si el estado del objeto de tipo Bool del primer parámetro es verdadero.

Formato	Ejemplos
JT {ámbito::}ref, <etiqueta>	**JT** condición, finBucle

JTD bool, destino

Salto condicional con liberación de instancia

El MC (*method counter*, contador de método) que indica la siguiente instrucción a ejecutar del hilo, pasa a apuntar a la instrucción que marca la etiqueta si el estado del objeto de tipo Bool del primer parámetro es verdadero.

Seguidamente libera el objeto consultado. Véase Delete.

Formato	Ejemplos
JTD {ámbito::}ref, <etiqueta>	**JTD** condición, finBucle

NEW referencia

Creación de una instancia

Crea una nueva instancia en el área de instancias de la misma clase que el tipo de la referencia y asigna dicha instancia a la referencia.

Se produce una excepción si la creación no pudo realizarse.

Formato	Ejemplos
New {ámbito::}ref	**New** rDato **New** Base1::refAncestro

NEW clase, referencia

Creación de una instancia de clase

Crea una nueva instancia en el área de instancias de la clase que indique el primer operando. La referencia que ocupa el segundo operando apuntará a la nueva instancia.

El primer operando deberá ser una referencia de tipo cadena o derivado cuyo valor sea el nombre de la clase a la cual pertenecerá la nueva instancia.

Se produce una excepción si la creación no pudo realizarse.

Formato	Ejemplos
New {ámbito::}ref, {ámbito::}ref	**New** strINTEGER, refInt

NEWCLASS referencia

Creación de una clase

Introduce una nueva clase en el área de instancias. La referencia que lleva como operando deberá apuntar a un objeto de tipo Stream desde donde se leerá la descripción de la misma. Esta instrucción no ha sido incluida de forma definitiva en el repertorio de instrucciones.

Se produce una excepción en los siguientes casos:

- Error de E/S.
- Alguno de los ancestros de la clase no existe en el área de clases.
- La clase de alguno de los objetos agregados de la clase no existe en el área de clases.
- La secuencia no contiene la declaración de una clase.
- Formato de clase incorrecto (corrupto).

Formato	Ejemplos
NewClass {ámbito::}ref	**NewClass** streamRef **NewClass** Base1::streamRef

THROW

Lanzamiento de una excepción

Pasa el flujo de control al último manejador declarado mediante la instrucción del mismo nombre.

Liberará todas las instancias locales de los métodos comprendidos entre el punto donde se produjo la excepción y el método donde continuará el flujo de ejecución. No ocurrirá así con las instancias creadas manualmente con New.

En caso de que no se encuentre ningún manejador el hilo finalizará y su estado quedará con el valor UNHANDLED

Formato	Ejemplos
Throw	**Throw**

Apéndice G

EXCEPCIONES EN TIEMPO DE EJECUCIÓN LANZADAS POR LA MÁQUINA CARBAYONIA

A continuación se incluye una relación de los posibles errores (excepciones) que se pueden producir durante la ejecución del simulador.

1. **TUnknownInstr**
   Se ha intentado cargar un fichero objeto con la definición de una clase que contiene un método con instrucciones desconocidas.
2. **TClassNotFound**
   Se ha intentado cargar un fichero objeto con la definición de una clase que contiene referencias a clases no existentes en el área de clases. Cuando se carga una clase en el área de clases deben estar previamente en ella toda clase ancestro de la de ella y las clases de todo sus miembros agregados.
3. **TInstanceNotFound**
   El hilo ha ejecutado una instrucción la cual hace referencia a una instancia no existente en el área de instancias. Se trata de un error de programación debido a la liberación de una instancia, y esta se sigue usando en otra parte del programa
4. **TMethodNotFound**
   Un hilo ha invocado a un método no existente en la clase a la que pertenece la instancia sobre la que se invocó ni en ninguno de sus ancestros.
5. **TRefNotFound**
   Una instrucción requiere una referencia no existente en el hilo en el que se ejecuta. La referencia no se encuentra entre los argumentos del método, referencias locales, instancias locales ni miembros agregados o asociados de la instancia sobre la que se ejecuta el método ni en ninguno de sus ancestros.
6. **TNumParamError**
   El número de parámetro en la invocación a un método no coincide con la signatura del mismo.
7. **TReturnTypeError**
   El tipo de referencia usado para recoger un valor de retorno no es compatible con el tipo de retorno del método.
8. **TWrongScope**
   El ámbito usado para acceder a una referencia o método no es válido.
9. **TDinamicCast**
   Se ha producido un error al intentar asignar una intancia a una referencia incompatible con su tipo. Esta situación puede darse al convertir los objetos utilizados como parámetros en la invocación de un método a los tipos de los argumentos declarados en el cuerpo del mismo.

10. TNullRef
Se ha intentado realizar una operación no permitida sobre una referencia libre. Esta situación puede ocurrir cuando se intenta utilizar Delete o invocar un método sobre una referencia libre.

11. TAggDeletion
Se ha intentado liberar una instancia agregada. Las instancias agregadas se liberan automáticamente al liberarse la instancia de las que son miembros.

12. TBoolRefReq
En las instrucciones que requieren una referencia apuntando a un objeto de tipo Bool o derivado se ha utilizado un objeto inadecuado.

13. TStringRefReq
En las instrucciones que requieren una referencia apuntando a un objeto de tipo String o derivado se ha utilizado un objeto inadecuado.

14. TInvalidMC
El contador de método (MC) está fuera del rango de instrucciones disponibles en un método. El MC apunta a la siguiente instrucción del método actual en el hilo que se ejecutará.

15. TNullParam
Se ha intentado pasar una referencia nula a un método primitivo que no lo permite. La mayor parte de los métodos de las clases primitivas exigen que se les pasen referencias no libres.

Apéndice H
GRAMÁTICA DEL LENGUAJE CARBAYÓN

En este apéndice se describe la gramática del lenguaje Carbayón. Con el objeto de facilitar el desarrollo de reconocedores del lenguaje, la descripción se hace en términos de las herramientas de desarrollo de compiladores [LMB92] Lex y Yacc[1].

H.1 Componentes léxicos

En este apartado se describe el conjunto de elementos léxicos (*tokens*) que permite el lenguaje. Esta descripción se realiza utilizando el subconjunto de expresiones regulares compatible con la herramienta Lex, un generador de analizadores léxicos.

Expresión	Token
[a-zA-Z_]	Letra
[0-9]	Cifra
"("	Paréntesis de abrir
")"	Paréntesis de cerrar
","	Coma (separador de referencias)
"."	Punto (invocación de métodos)
":"	Dos puntos (declaración de tipo)
"::"	Operador de ámbito
";"	Terminador de línea
[+-]?{cifra}+	Constante entera
[+-] ?{cifra}+("."{cifra}+)?([Ee] [+-]?{cifra}+)?	Constante float
""".*"""	Constante de cadena
{letra}({cifra}|{letra})*	Identificador (excepto palabras reservadas)
"/".*	Comentario hasta fin de línea

El lenguaje además considera con significado especial las siguientes palabras reservadas, las cuales no podrán usarse como identificadores:

ABSTRACT	AGGREGATION	ASSIGN
ASSOCIATION	CLASS	CODE
CONCURRENT	DELETE	ENDCLASS
ENDCODE	EXIT	FLOAT
HANDLER	INSTANCES	INTEGER
ISA	JF	JFD
JMP	JNNULL	JNULL
JT	JTD	MESSAGES

[1] Herramientas que por su gran difusión a través del sistema UNIX se han utilizado para la implementación del prototipo de máquina Carbayonia (capítulo 13) y su entorno de desarrollo (apéndice A).

METHODS	NEW	NEWCLASS
REFS	STRING	THROW
VOID		

H.2 Sintaxis del lenguaje

En este apartado se describe la gramática del lenguaje. Esta descripción se realiza utilizando producciones compatibles con la herramienta Yacc, un generador de analizadores sintácticos.

```
classDef: CLASS ident herencia variables funciones ENDCLASS;

herencia: /* vacio */
     | ISA listaIdent PUNTOYCOMA;
variables:     agregacion asociacion;
agregacion:    /* vacio */
     | AGGREGATION listaRefs PUNTOYCOMA;
asociacion:    /* vacio */
     | ASSOCIATION listaRefs PUNTOYCOMA;

funciones:     metodos mensajes;
metodos:  /* vacio */
     | METHODS funcDefs;
mensajes: /* vacio */
     | MESSAGES funcDefs;

funcDefs: funcDef
     | funcDefs funcDef;
funcDef:  tipoConcurr ident argumentos retorno localRefs
     localInstances codigo;
tipoConcurr:   /* vacio */
     | CONCURRENT
argumentos:    /* vacio */
     | A_PARENT C_PARENT
     | A_PARENT listaRefs C_PARENT;
retorno:  /* vacio */
     | DOSPUNTOS claseRef;

localRefs:     /* vacio */
     | REFS listaRefs PUNTOYCOMA;
localInstances:/* vacio */
     | INSTANCES listaInstances PUNTOYCOMA;

codigo:   ABSTRACT
     | CODE sentencias ENDCODE;
sentencias:    /* vacio */
               | listaSentencias;

listaSentencias:    sentencia
     | listaSentencias sentencia;

sentencia:     labelDef
     | instruccion PUNTOYCOMA;

labelDef: ident DOSPUNTOS;
label:    ident;
```

```
instruccion:   ref PUNTO metodo parametros recoge
     | HANDLER label
     | NEW ref
     | ASSIGN ref COMA ref
     | DELETE ref
     | JT ref COMA label
     | JTD ref COMA label
     | JF ref COMA label
     | JFD ref COMA label
     | JNULL ref COMA label
     | JNNULL ref COMA label
     | JMP label
     | EXIT
     | NEWCLASS ref
     | THROW
     | NEW ref COMA ref

ref: identWithScope;

metodo:   identWithScope;

identWithScope:ident
     | identWithScope CUATROPUNTOS ident;

parametros:    /* vacio */
     | A_PARENT C_PARENT
     | A_PARENT listaIdentWithScope C_PARENT;

listaIdentWithScope: identWithScope
     | listaIdentWithScope COMA identWithScope;

recoge:   /* vacio */
     | DOSPUNTOS ref;

listaInstances:grupoInstances
     | listaInstances PUNTOYCOMA grupoInstances;
grupoInstances:listaIdent DOSPUNTOS claseInstance;
claseInstance: ident | cInteger | cFloat | cString;


cInteger: INTEGER valorInt;
valorInt: /* vacio */
     | A_PARENT intCte C_PARENT;
intCte:   INTCTE;
cFloat:   FLOAT valorFloat;
valorFloat:    /* vacio */
     | A_PARENT floatCte C_PARENT;
floatCte: FLOATCTE;
cString:  STRING valorStr;
valorStr: /* vacio */
     | A_PARENT stringCte C_PARENT;
stringCte:     STRINGCTE;

listaRefs:     grupoRefs
     | listaRefs PUNTOYCOMA grupoRefs;
```

```
grupoRefs:      listaIdent DOSPUNTOS claseRef;
claseRef: ident
     | INTEGER
     | FLOAT
     | STRING;

listaIdent:     ident
     | listaIdent COMA ident;
ident:    IDENT;
```

Apéndice I

FORMATO DEL FICHERO DE CLASES

En este apéndice se describe el formato del fichero de clases para la máquina abstracta Carbayonia, en forma de una gramática compatible con las herramientas de desarrollo de compiladores [LMB92] Lex y Yacc[1].

Este fichero es generado por el entorno de desarrollo y que es el que requiere el simulador como entrada de las clases de usuario.

La descripción se realizará mediante reglas de producción siguiendo el convenio de que los símbolos no terminales se escriben en minúsculas y los terminales en mayúsculas.

```
clase:    firma nombreClase herencia agregación
     asociación funciones epilogo;

firma:    "LEOOxCLASS";

nombreClase:   IDENTIFICADOR;

herencia: "H" listaAncestros "EH";

listaAncestros:ancestro
     | listaAncestros ancestro;

ancestro: IDENTIFICADOR;

agregación:     /* nada */
     | "AG" listaReferencias "EAG";

listaReferencias:   referencia
     | listaReferencias referencia;

referencia:    nombre clase;

nombre:   IDENTIFICADOR;

clase:    IDENTIFICADOR;

asociación:     /* nada */
     | "AS" listaReferencias "EAS";

funciones:      /* nada */
     | listaFunciones;

listaFunciones:función
```

[1] Herramientas que por su gran difusión a través del sistema UNIX se han utilizado para la implementación del prototipo de máquina Carbayonia (capítulo 13) y su entorno de desarrollo (apéndice A).

```
     | listaFunciones función;

función:  "F" tipoFunción tipoExclusion nombreFunción
     signatura cuerpo;

signatura:     parámetros retorno;

tipoFunción:   "M"   /* Si es un método */
     | "S";     /* Si es un mensaje */

tipoExclusión: "C"   /* Si es concurrente */
     | "E";     /* Si es exclusivo */

nombreFunción: clase "::" nombre;

parámetros:    /* nada */
     | listaReferencias;

retorno:  "R" clase
     | "R" "VOID";

cuerpo:   referenciasLocales instanciasLocales  offsetLabels
código;

referenciasLocales:  /* nada */
     | "R" listaReferencias "ER";

instanciasLocales:   /* nada */
     | "I" listaReferencias "EI";

código:   "VOID"
     | offsetLabels "CD" instrucciones "EC"     etiquetas;

offsetLabels:  CONSTANTE_ENTERA;     /* offset etiquetas
                          dentro fichero*/

instrucciones: instrucción
     | instrucciones instrucción;

instrucción:   tipoInstr numLinea;

numLinea: CONSTANTE_ENTERA;

tipoInstr:     call
     | handler
     | new
     | assign
     | delete
     | jt
     | jtd
     | jf
     | jfd
     | jnull
     | jnnull
     | jmp
     | exit
```

```
     | newclass
     | throw;
     | new2;

call:     "0" nombre listaReferencias lvalue;

lvalue:   nombre
     | "VOID";

handler:  "1" nombre;

new: "2" nombre;

assign:   "3" nombre nombre;

delete:   "4" nombre;

jt:  "5" nombre nombre;

jtd: "6" nombre nombre;

jf:  "7" nombre nombre;

jfd: "8" nombre nombre;

jnull:    "9" nombre nombre;

jnnull:   "10" nombre nombre;

jmp: "11" nombre;

exit:     "12";

newclass: "13" nombre;

throw:    "14" nombre;

new2:     "15" nombre nombre;

etiquetas:     "LB" listaEtiquetas "ELB";

listaEtiquetas:etiqueta
     | listaEtiquetas etiqueta;

etiqueta: nombre posición;

posición: CONSTANTE_ENTERA;

epilogo:  ficheroFuente "ENDCLASS";

ficheroFuente: IDENTIFICADOR;
```

Apéndice J

FORMATO DEL ARCHIVO DE INTERCAMBIO Y DE LOS SEGMENTOS DEL SISTEMA DE PERSISTENCIA

En este apéndice se describe el formato común del archivo de intercambio utilizado por el prototipo de la máquina abstracta con soporte de persistencia (véase el capítulo 18).

J.1 Archivo de intercambio

- **Tamaño de página**. 4 bytes sin signo que indican el tamaño de las páginas de memoria virtual del sistema de persistencia.
- **Secuencia de páginas**. A continuación se coloca la secuencia de páginas del tamaño anterior que componen la memoria virtual. Cada página está compuesta por una serie de segmentos que representan los objetos persistentes.

J.2 Segmentos de objetos persistentes

El formato de un segmento dentro de una página se compone de una cabecera de longitud fija que es seguida de unos datos de longitud variable que representan el estado del objeto persistente que se almacena en el segmento:

- **Cabecera (9 bytes)**
  - **Size**. 4 bytes sin signo que indican el tamaño de los datos del segmento, sin incluir el propio tamaño de la cabecera.
  - **Type**. Un byte que indica el tipo de segmento almacenado. Cada segmento ha de ser asociado a un tipo concreto. Se ha impuesto esta condición a los segmentos para no obligar a definir un constructor por defecto. De esta forma los constructores son registrados en el método New de la clase TvirtualMemory del programa C++ del simulador de la máquina. Es necesario crear objetos puesto que las necesidades de memoria pueden hacer que la memoria virtual haya de liberar alguno de vez en cuando. La identificación de cada uno de los elementos mediante un byte es la siguiente:

'E'	Último segmento de la memoria virtual y por lo tanto del archivo de intercambio. Está vacío y por lo tanto tiene longitud cero
'Z'	Elemento libre de la página. Se crea un segmento libre cuando en la página no hay espacio para ningún elemento más.
'D'	Segmento borrado de la memoria virtual. En un solo archivo sería muy costoso borrarlo físicamente, por lo que tan sólo se marca como borrado.
'A'	Segmento añadido. Cuando un segmento tiene longitud variable, usa una lista enlazada de segmentos añadidos. Un ejemplo es el segmento que utilizan las instancias de TIString.

‘s’	Segmento donde se almacena un String.
‘0’	Segmento donde se almacenan instancias de TCObject.
‘1’	Segmento donde se almacenan instancias de TCInteger.
‘2’	Segmento donde se almacenan instancias de TCFloat.
‘3’	Segmento donde se almacenan instancias de TCBool.
‘4’	Segmento donde se almacenan instancias de TCString.
‘5’	Segmento donde se almacenan instancias de TCConstrea.
‘6’	Segmento donde se almacenan instancias de TCFilestream.
‘7’	Segmento donde se almacenan instancias de TCArray.
‘8’	Segmento donde se almacenan instancias de TCSemaphore.
‘9’	Segmento donde se almacenan instancias de TCPersistence.
‘c’	Segmento donde se almacenan instancias de TUserClass.
‘R’	Segmento donde se almacenan instancias de TRef.
‘z’+1	Segmento donde se almacenan instancias de TMArraySetSize.
‘z’+2	Segmento donde se almacenan instancias de TMArrayGetSize.
‘z’+3	Segmento donde se almacenan instancias de TMArraySetRef.
‘z’+4	Segmento donde se almacenan instancias de TMArrayGetRef.
‘z’+5	Segmento donde se almacenan instancias de TMSemaphoreSet.
‘z’+6	Segmento donde se almacenan instancias de TMSemaphoreWait.
‘z’+7	Segmento donde se almacenan instancias de TMSemaphoreSignal.
‘z’+9	Segmento donde se almacenan instancias de TMConstreamClearScreen.
‘z’+10	Segmento donde se almacenan instancias de TMConstreamWrite.
‘z’+11	Segmento donde se almacenan instancias de TMConstreamRead.
‘z’+12	Segmento donde se almacenan instancias de TMConstreamNLine.
‘z’+13	Segmento donde se almacenan instancias de TMFilestreamWrite.
‘z’+14	Segmento donde se almacenan instancias de TMFilestreamRead.
‘z’+15	Segmento donde se almacenan instancias de TMFileFileOpen.
‘z’+16	Segmento donde se almacenan instancias de TMFileFileClose.
‘z’+17	Segmento donde se almacenan instancias de TMFileFileEof.
‘z’+18	Segmento donde se almacenan instancias de TMFileFileSeek.
‘z’+19	Segmento donde se almacenan instancias de TMFileFileTeel.
‘z’+20	Segmento donde se almacenan instancias de TMObjectGetClass.
‘z’+21	Segmento donde se almacenan instancias de TMObjectGetId.
‘z’+22	Segmento donde se almacenan instancias de TMObjectIs.
‘z’+24	Segmento donde se almacenan instancias de TMIntegerAdd.
‘z’+25	Segmento donde se almacenan instancias de TMIntegerSub.
‘z’+26	Segmento donde se almacenan instancias de TMIntegerMul.
‘z’+27	Segmento donde se almacenan instancias de TMIntegerDiv.
‘z’+28	Segmento donde se almacenan instancias de TMIntegerSet.
‘z’+30	Segmento donde se almacenan instancias de TMIntegerEqual.
‘z’+31	Segmento donde se almacenan instancias de TMIntegerGreater.
‘z’+32	Segmento donde se almacenan instancias de TMIntegerLess.
‘z’+33	Segmento donde se almacenan instancias de TMIntegerSetF.
‘z’+35	Segmento donde se almacenan instancias de TMFloatAdd.
‘z’+36	Segmento donde se almacenan instancias de TMFloatSub.

‘z’+37	Segmento donde se almacenan instancias de TMFloatMul
‘z’+38	Segmento donde se almacenan instancias de TMFloatDiv.
‘z’+39	Segmento donde se almacenan instancias de TMFloatSet.
‘z’+41	Segmento donde se almacenan instancias de TMFloatEqual.
‘z’+42	Segmento donde se almacenan instancias de TMFloatGreater.
‘z’+43	Segmento donde se almacenan instancias de TMFloatLess.
‘z’+44	Segmento donde se almacenan instancias de TMFloatSetI.
‘z’+46	Segmento donde se almacenan instancias de TMStringEqual.
‘z’+47	Segmento donde se almacenan instancias de TMStringGreater.
‘z’+48	Segmento donde se almacenan instancias de TMStringLess.
‘z’+49	Segmento donde se almacenan instancias de TMStringSet.
‘z’+50	Segmento donde se almacenan instancias de TMStringSetChar.
‘z’+51	Segmento donde se almacenan instancias de TMStringRemove.
‘z’+52	Segmento donde se almacenan instancias de TMStringGetChar.
‘z’+53	Segmento donde se almacenan instancias de TMStringLength.
‘z’+54	Segmento donde se almacenan instancias de TMStringGet.
‘z’+55	Segmento donde se almacenan instancias de TMStringInsert.
‘z’+56	Segmento donde se almacenan instancias de TMStringSetI.
‘z’+57	Segmento donde se almacenan instancias de TMBoolSetTrue.
‘z’+58	Segmento donde se almacenan instancias de TMBoolSetFalse.
‘z’+59	Segmento donde se almacenan instancias de TMBoolNot.
‘z’+61	Segmento donde se almacenan instancias de TMBoolAnd.
‘z’+62	Segmento donde se almacenan instancias de TMBoolOr.
‘z’+63	Segmento donde se almacenan instancias de TMBoolXor.
‘z’+64	Segmento donde se almacenan instancias de TMPersistenceExists.
‘z’+65	Segmento donde se almacenan instancias de TMPersistenceRename.
‘z’+66	Segmento donde se almacenan instancias de TMPersistenceAdd.
‘z’+67	Segmento donde se almacenan instancias de TMPersistenceRemove.
‘z’+68	Segmento donde se almacenan instancias de TMPersistenceGetObject.
‘m’	Segmento donde se almacenan instancias de TUserMethod.
1	Segmento donde se almacenan instancias de TinstrCall.
2	Segmento donde se almacenan instancias de TinstrHandler.
3	Segmento donde se almacenan instancias de TinstrNew.
4	Segmento donde se almacenan instancias de TinstrAssign.
5	Segmento donde se almacenan instancias de TinstrDelete.
6	Segmento donde se almacenan instancias de TinstrJcc.
7	Segmento donde se almacenan instancias de TinstrJcNull.
8	Segmento donde se almacenan instancias de TinstrJmp.
9	Segmento donde se almacenan instancias de TinstrExit.
10	Segmento donde se almacenan instancias de TinstrNewClass.
11	Segmento donde se almacenan instancias de TinstrThrow.
12	Segmento donde se almacenan instancias de TinstrNew2.
20	Segmento donde se almacenan instancias de TIObject.
21	Segmento donde se almacenan instancias de TIInteger.
22	Segmento donde se almacenan instancias de TIFloat.

23	Segmento donde se almacenan instancias de TIBool.
24	Segmento donde se almacenan instancias de TIString.
25	Segmento donde se almacenan instancias de TIConstream.
26	Segmento donde se almacenan instancias de TIFilestream.
27	Segmento donde se almacenan instancias de TIArray.
28	Segmento donde se almacenan instancias de TISemaphore.
29	Segmento donde se almacenan instancias de TIPersistence.
‘i’	Segmento donde se almacenan instancias de TUserInstance.

GLOSARIO DE TRADUCCIONES

Expresión en inglés	Traducciones preferidas	Otras Traducciones
API (Applications Program Interface)	API (Interfaz de los programas de aplicación)	
Array	Array, matriz	Arreglo
Bootstrapping	Autoarranque	*Bootstrapping*
Cache (memory)	(Memoria) caché	
Capability	Capacidad	
Checkpoint	Punto de verificación	
Customize	Personalizar (el software para una aplicación)	
Deploy on-demand	Instalación sobre la marcha	
Embedded system	Sistema empotrado, encastrado, inmerso	Sistema embebido
Exo, micro, nano kernel	Exo, micro, nano núcleo	
Framework	Marco de aplicación	
Garbage Collection	Recolección de basura, recogida de basura	
Heap	Montón	*Heap*
Hint	Pista (que guía al sistema)	
Impedance mismatch	Desadaptación de impedancias	
Interface	Interfaz (femenino)	
IPC, Inter-Process Communication	Comunicación entre procesos	
Just in Time, JIT	Justo a tiempo	
Lightweight activity	Actividad ligera	
Load balancing	Equilibrio de (la) carga, equilibrado de (la) carga	
Map, mapping	Hacer corresponder, *"mapear"*	
Mark and sweep	Marca y barrido	
Minimal kernel	De núcleo mínimo	
Name service	Servicio de denominación	Servicio de nombrado

Object Request Broker	Gestor de objetos, intermediario entre objetos	
Overloading	Sobrecarga (de métodos)	
Overriding	Redefinición (de métodos)	Anulación
Page frame	Marco de página	
Pointer swizzling	Transformación de punteros	
Proxy	Representante, proxy	
Reference-counting	Cuenta de referencias	
Reflection	Reflexión (acción de reflejar), reflejo (lo reflejado)	
Reflective overlap	Solapamiento reflectivo	
Reflectivity	Reflectividad, reflexividad	
Reification	Cosificación, concretización	
Resilience	Elasticidad, poder de recuperación	
RTTI, Run Time Type Information	Comprobación de tipos en tiempo de ejecución	
SAS, single address space	Espacio de direcciones único	
Single level store	Almacenamiento de nivel único	
Sparse capabilities	Capacidades dispersas	
Stack frame	Registro de activación	
Stack top	Cima de la pila	Tope de la pila
Stream	Secuencia	Flujo
Swapping	Intercambio	
Tag bit	Bit de marca	bit de etiqueta
Tailor	Ajustar, hacer a medida (el software de sistema a una aplicación)	
Template	Plantilla (de objetos)	Planilla, patrón
Thread	Hilo (de ejecución)	Hebra, flujo
Typecast	Amoldamiento de tipos, conversión de tipos	Ahormado de tipos
Up-call	Retrollamada	*Up-call*

BIBLIOGRAFÍA

[ABB+86] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian y M. Young. "Mach: A New Kernel Foundation for Unix Development". En Proceedings of the *Summer Usenix Conference*. USENIX. 1986. Pág. 93-112.

[ABB+93] H. Assenmacher T. Breitbach, P. Buhler, V. Hübsch y R. Schwarz. "The PANDA Sytem Architecture: A Pico-Kernel Approach". En Proceedings of the *4th Workshop on Future Trends of Distributed Computing Systems*. IEEE Computer Society Press. Septiembre de 1993. Pág. 470-476.

[ABC+83] M. P. Atkinson, P. Bailey, K. J. Chisholm, W.P. Cockshott y R. Morrison. "An Approach to Persistent Programming". *The Computer Journal*, vol 26, 4. 1983. Pág. 360-365.

[AC88] M. Anderson y C. Wallace. "Some Comments on the Implementation of Capabilities". *The Australian Computer Journal*. 1988.

[ACC81] M.P. Atkinson, K.J. Chisholm y W.P. Cockshott. "PS-Algol: An Algol with a Persistent Heap". *ACM SIGPLAN Notices*, 17(7). 1981. Pág. 24-31.

[ADA+96] Darío Álvarez Gutiérrez, María Ángeles Díaz Fondón, Fernando Álvarez García, Lourdes Tajes Martínez, Ana Belén Martínez Prieto y Raúl Izquierdo Castanedo. "Persistencia en sistemas operativos orientados a objetos: Ventajas para los sistemas de gestión de bases de datos". Actas de las *Primeras Jornadas de Investigación y Docencia en Bases de Datos (JIDBD'96)*. La Coruña, Junio de 1996.

[AG96] Ken Arnold y James Gosling. *The Java Programming Language*. Addison-Wesley. 1996.
Versión en español: *El Lenguaje de Programación Java*. Addison-Wesley 1997.

[AJD+96] M.P. Atkinson, M.J. Jordan, L. Daynès y S. Spence. "Design Issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system". *Seventh International Workshop on Persistent Object Systems*. 1996.

[ALL+96] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco y Robert Wahbe. "Efficient and Language-Independent Mobile Programs". En Proceedings of the *ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*. Mayo de 1996. Pág. 127-136.

[Alv94] Darío Álvarez Gutiérrez, María Ángeles Díaz Fondón y José María Troya Linero. "La programación orientada a objetos aplicada al desarrollo de un emulador de microprocesador de arquitectura paralela". *Congreso Internacional de Ingeniería de Proyectos. Asturias*. Octubre de 1994.

[Alv96a] Fernando Álvarez García. "Distribución en sistemas operativos orientados a objetos". *II Jornadas sobre Tecnologías Orientadas a Objetos. Oviedo*. Marzo de 1996.

[Alv96b] Darío Álvarez Gutiérrez. "Persistencia en un sistema operativo orientado a objetos". *II Jornadas sobre Tecnologías Orientadas a Objetos*. Oviedo. Marzo de 1996.

[AOP97] AOpen Inc. *AP5T Mainboard User's Guide*. 1997

[ATA+96] 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". Actas de las *II Jornadas de Informática*. Almuñécar, Granada. Julio de 1996.

[ATA+97] Darío Álvarez Gutiérrez, Lourdes Tajes Martínez, Fernando Álvarez García, María Ángeles Díaz Fondón, Raúl Izquierdo Castanedo y Juan Manuel Cueva Lovelle. "An Object-Oriented Abstract Machine as the Substrate for an Object-Oriented Operating System". *Eleventh European Conference in Object Oriented Programming (ECOOP'97)*, Workshop in Object Orientation in Operating Sytems. Jyväskylä, Finlandia. Junio de 1997.

[Bac86] Maurice J. Bach. *The Design of the Unix Operating System*. Prentice-Hall. 1986.

[BC85] A.L. Brown y W.P. Cockshott. "The CPOMS Persistent Object Management Technique". Universidades de Glasgow y St. Andrews, *Persistent Programming Report* 13. 1985.

[BDK92] F. Bancilhon, C. Delobel y P. Kanellakis, ed. *Building an Object-Oriented Database system - The story of O2*. Morgan Kaufmann. 1992.

[BF97] Francisco J. Ballesteros y Luis L. Fernández. "The Network Hardware is the Operating System". En Proceedings of the *6th Hot Topics in Operating Systems (HotOS-VI)*. Cape Codd, Massachussets, EE.UU. Mayo de 1997.

[BFG+85] J. Banino, J. Fabre, M. Guillemont, R. Morisset y M. Rozier. Some Fault-Tolerant "Aspects of the CHORUS Distributed System". *International Conference on Distributed Computing Systems (ICDCS)*. 1985.

[Bla90] David L. Black. "Scheduling Support for Concurrency and Parallelism in the Mach Operating System". *IEEE Computer*. Mayo de 1990.

[BM92] A.L. Brown y R. Morrison. "A Generic Persistent Object Store". *Software Engineering Journal 7*(2). 1992. Pág. 161-168.

[Boo91] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings. 1991.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications, 2nd edition*. Benjamin Cummings. 1994.
Versión en español: *Análisis y diseño orientado a objetos con aplicaciones, 2ª edición*. Addison-Wesley/Díaz de Santos. 1996.

[BPF+97] William J. Bolosky, Richard P. Draves, Robert P. Fitzgerald, Chistopher W. Fraser, Michael B. Jones, Todd B. Knoblock y Rick Rashid. "Operating System Directions for the Next Millenium". En Proceedings of the *6th Hot Topics in Operating Systems (HotOS-VI)*. Cape Codd, Massachussets, EE.UU. Mayo de 1997.

[BR95] Grady Booch y James Rumbaugh. *Unified Method for Object-Oriented Development, Documentation Set, Version 0.8*. Rational Software Corporation. 1995.

[BRJ96] Grady Booch, James Rumbaugh e Ivar Jacobson. *The Unified Modeling Language for Object-Oriented Development, Version 0.9*. Rational Software Corporation. Julio de 1996.

[BSP+95] B.N: Bershad, S. Savage, P. Pardyak, E.G. Sirer, M. Fiunczynski, D. Becker, S. Eggers y C. Chambers. "Extensibility, Safety and Performance in the SPIN Operating System". En Proceedings of the *15th ACM Symposium on Operating Systems Principles (SOSP)*. Copper Mountain, Colorado, EE.UU. Diciembre de 1995.

[Buh90] P. Buhler. "The COIN Model for Concurrent Computation and its Implementation". *Microprocessing and Microprogramming*, V. 30, N. 1-5. North Holland. 1990. Pág.577-584.

[Cah96] Vinny Cahill. *Flexibility in Object-Oriented Operating Systems: A Review*. Technical Report TCD-CS-96-05. Trinity College Dublin. 1996.

[Cah96a] Vinny Cahill. "An Overview of the Tigger Object-Support Operating System Framework". En *SOFSEM'96: Theory and Practice of Informatics. Lecture Notes in Computer Science*, V. 1175. 1996. Pág. 34-55.

[Cam83] F. Campbell. "The Portable UCSD p-System". *Microprocessors and Microsystems*, Núm. 7. Octubre de 1983. Pág. 394-398.

[Car89] D. Caromel. "A General Model for Concurrent and Distributed Object-Oriented Programming". *SIGPLAN Notices*, 24(4). Abril de 1989.

[CC91] R.S. Chin y S.T. Chanson. "Distributed Object-Based Programming Systems". *ACM Computing Surveys*, V. 23, N.1. Marzo de 1991.

[CDK94] G. Coulouris, J. Dollimore y T. Kindberg. *Distributed Systems: Concepts and Design, 2nd edition.* Addison-Wesley. 1994.

[CGL+94] Juan Manuel Cueva Lovelle, María Pilar Almudena García Fuente, Benjamín López Pérez, María Cándida Luengo Díez y Melchor Alonso Requejo. *Introducción a la programación estructurada y orientada a objetos con Pascal*. Cuaderno Didáctico Nº 69, Departamento de Matemáticas de la Universidad de Oviedo. 1994. ISBN: 84-600-8646-1.

[CIA96] Juan Manuel Cueva Lovelle, Raúl Izquierdo Castanedo y Darío Álvarez Gutiérrez. "Oviedo3: Acercando las tecnologías orientadas a objetos al hardware". *I Jornadas de trabajo en Ingeniería de Software*. Sevilla. Noviembre de 1996.

[CIM+93] R. Campbell, N. Islam, P. Madany y D. Raila. "Designing and Implementing Choices: an Object-Oriented System in C++". *Communications of the ACM*. Septiembre de 1993.

[CNK+97] Shigeru Chiba, Takeshi Nishimura, Kenichi Kourai, Atsushi Ohnoki y Takashi Mashuda. "Weak Protection for Reflective Operating Systems". *Eleventh European Conference in Object Oriented Programming (ECOOP'97), Workshop on Reflective Real-Time Object-Oriented Programming and Systems*. Jyväskylä, Finlandia. Junio de 1997.

[Cue95] Juan Manuel Cueva Lovelle. *Conceptos básicos de traductores, compiladores e intérpretes, quinta edición.* Colección Cuadernos Didácticos del Departamento de Matemáticas de la Universidad de Oviedo. 1995.

[DAM+90] P. Dasgupta, R. Chen, S. Menon, M. Person, R. Ananthanarayanan, U. Ramaschandran, M. Ahamad, R. LeBlanc, W. Applebe, J. Barnabeu-Auban, P. Hutto, M. Khalidi y C. Wilkenloh. "The Design and Implementation of the Clouds Distributed Operating System". *Computing Systems* 3(1). 1990.

[Dei90] H.M. Deitel. *Operating Systems, second edition.* Addison-Wesley. 1990. Versión en español: *Sistemas Operativos, segunda edición.* Addison-Wesley. 1993.

[DH66] J.B. Dennis y E.C. van Horn. "Programming Semantics for Multiprogrammed Computations". *Communications of the ACM*, V. 9, N. 3. 1966.

[Día96] María Ángeles Díaz Fondón. "Seguridad en un sistema operativo orientado a objetos". *II Jornadas sobre Tecnologías Orientadas a Objetos*. Oviedo. Marzo de 1996.

[DLA+91] P. Dasgupta, R.J. LeBlanc, M. Ahamad y U. Ramachandran. "The Clouds Distributed Operating System". *IEEE Computer*, 24(11). 1991. Pág. 34-44.

[DPP+96] Sean Dorward, Rob Pike, Dave Presotto, Howard Trickey y Phil Wintebottom. "Inferno: la Commedia Interattiva". En Proceedings of *the Second Symposium on Operating Systems Design and Implementation (OSDI'96) Works in Progress (WIP)*. Saint Malo, Francia. Octubre de 1996.

[DPZ97] Peter Druschel, Vivek S. Pai, Willy Zwaenepoel. "Extensible Kernels are Leading OS Research Ashtray". En Proceedings of the *6th Hot Topics in Operating Systems (HotOS-VI)*. Cape Codd, Massachussets, EE.UU. Mayo de 1997.

[Dra93] Richard P. Draves. "The Case for Run-Time Replaceable Modules". En Proceedings of the *4th Workshop on Workstation Operating Systems*. IEE Computer Society Press. Octubre de 1993. Pág. 160-164.

[DRH+92] A. Dearle, J. Rosenberg, F. Henskens, F. Vaughan y K. Maciunas. "An Examination of Operating System Support for Persistent Object Systems". En Proceedings of the *25th Hawaii International Conference on System Sciences*, Hawaii, EE.UU. Pág. 779-789, 1992.

[DRK+89] D. DeCouchant, M. Riveill, S. Krakowiak, C. Horn, E. Finn, y N. Harris. "Experience with Implementing and Using an Object-Oriented Distributed System". En Proceedings of the *Usenix Workshop on Experiences with Distributed and Multiprocessor Systems*. October 1989. Pág. 301-310.

[EKO95] D. R. Engler, M. F. Kaashoek y J. O'Toole. "Exokernel: An Operating System Architecture for Application-Level Resource Management". En Proceedings of the *15th ACM Symposium on Operating Systems Principles (SOSP)*. Copper Mountain, Colorado, EE.UU. Diciembre de 1995.

[FBB+96] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin y Olin Shivers (MIT). The Flux OSKit: "A Substrate for OS and Language Research". En Proceedings of the *Second Symposium on Operating Systems Design and Implementation (OSDI'96)*. Saint Malo, Francia. Octubre de 1996.

[Fer89] J. Ferber. “Computational Reflection in Class Based Languages”. En Proceedings of the *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’89)*. Nueva Orleans, Luisiana, EE.UU. Octubre de 1989.

[FHL+96] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, Shantanu Goel y Steven Clawson. “Microkernels Meet Recursive Virtual Machines”. En Proceedings of the *Second Symposium on Operating Systems Design and Implementation (OSDI’96)*. Saint Malo, Francia. Octubre de 1996. Pág. 137-152.

[FS94] Paulo Ferreira y Marc Shapiro. “Garbage Collection and DSM Consistency”. En Proceedings of the *First Symposium on Operating Systems Design and Implementation*. Monterrey, California, EE.UU. 1994.

[GB97] Robert Grimm y Brian N. Bershad. “Security for Extensible Systems”. En Proceedings of the *6th Hot Topics in Operating Systems (HotOS-VI)*. Cape Codd, Massachussets, EE.UU. Mayo de 1997.

[GK97] Michael Golm y Jurgen Kleinöder. “MetaJava – A Platform for Adaptable Operating System Mechanisms”. *Eleventh European Conference in Object Oriented Programming (ECOOP’97), Workshop in Object Orientation in Operating Sytems*. Jyväskylä, Finlandia. Junio de 1997.

[GKS94] Sven Graupner, Winfried Kalfa y Frank Schubert. *Multi-level Architecture of Object-Oriented Operating Systems*. Technical Report TR-94-056. ICSI, Berkeley, California, EE.UU. 1994

[Gos91] Andrzej Goscinski. *Distributed Operating Systems: The Logical Design*. Addison-Wesley. 1991.

[GR83] A. Goldberg y D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley. 1983.

[GW95] Brendan Gowing y Vinny Cahill. “Making Meta-Object Protocols Practical for Operating Systems”. *International Workshop in Object Orientation in Operating Systems (IWOOS’95)*. Lund, Suecia. 1995.

[GW96] Brendan Gowing y Vinny Cahill. “Reflection + Micro-Kernels: An Approach to Supporting Dynamically Adaptable System Services”. *CaberNet Radicals '96*. Connemara, Irlanda. 1996.

[Har86] David M. Harland. “A Recursively Microcodable Tagged Architecture”. *ACM SIGARCH Journal*, V. 24, N. 3. Junio de 1986. Pág. 34-40.

[HB86] David M. Harland y Bruno Beloff. “Microcoding an Object-Oriented Instruction Set”. *ACM SIGARCH Journal*. Octubre de 1986.

[HB87] David M. Harland y Bruno Beloff. “Objekt: A Persistent Object Store with an Integrated Garbage Collection”. *SIGPLAN NOTICES*, V. 22, N.4. Abril de 1987.

[HCC+97] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu y Thorsten von Eiken. “SLK: A Capability System Based on Safe Language Technology”. URL: http://www2.cs.cornell.edu/Slk/slk.ps. Septiembre de 1997.

[HGP+86] David M Harland, Hamish I. Gunn, Ian A. Pringle y Bruno Beloff. "The Rekursiv - An Architecture for Artificial Intelligence". En Proceedings of *AI / Europa*. Wiesbaden, Alemania. Septiembre de 1986

[HKM+96] D. Hagimont, S. Krakowiak, J. Mosière y X. Rousset de Pina. "A Selective Protection Scheme for the Java Environment".

[HMR+96] D. Hagimont, J. Mosière, X. Rousset de Pina y F. Saunier. "Hidden Capabilities". *16th International Conference on Distributed Computing Systems*. Mayo de 1996.

[Hoa78] C.A.R. Hoare. "Communicating Sequential Processes". *Communications of the ACM*, vol 21 8. Agosto de 1978. Pág. 666

[Höl95] Urs Hölze. *Adaptative Optimization for Self: Reconciling High Performance with Exploratory Programming*. Tesis Doctoral, Department of Computer Science, Stanford University, EE.UU. Marzo de 1995.

[HU94] U. Hölze y D. Ungar. "Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback". En Proceedings of the *1994 SIGPLAN Conference on Programming Language Design and Implementation*. Orlando, Florida, EE.UU. Junio de 1994.

[Hug97] Dwight Hughes. "A Summary of Project so far". Comunicación personal a las listas de correo lispos@math.gatech.edu y lispvm@math.gatech.edu. 21 de Mayo de 1997.

[INM88] INMOS Ltd. *Occam 2* Reference Manual. Prentice-Hall. 1988.

[Int81] Intel Corporation. *Intel 432 GDP Architecture*. 1981.

[Izq96] Raúl Izquierdo Castanedo. *Máquina Abstracta Orientada a Objetos*. Proyecto Fin de Carrera 9521I. Escuela Técnica Superior de Ingenieros Industriales e Ingenieros Informáticos de Gijón de la Universidad de Oviedo. Septiembre de 1996.

[JCJ+92] I. Jacobson, M. Christerson, P. Jonsson y G. Overgaard. *Object-Oriented Software Engineering*. Addison-Wesley. 1992.

[Joy97] Luis Joyanes Aguilar. *Metodología de ingeniería de software para objetos distribuidos en tiempo real en arquitecturas Cliente/Servidor – Internet/Intranet*. Tesis Doctoral, Departamento de Matemáticas de la Universidad de Oviedo. Noviembre de 1997.

[KB90] M. Karaoman y J.L. Bruno. *Introducing Concurrency to a Sequential Programming Language*. Technical Report. Department of Computer Science, UC Santa Barbara, EE.UU. 1990.

[KCD+81] Kevin C. Kahn, Willian M. Corwin, T. Don Dennis, Herman D'Hooge, David E. Hubka, Linda A. Hutchins, John T. Montague y Fred J. Pollack. "IMAX: A Multiprocessor Operating System for an Object-Based Computer". En Proceedings of the *8th Symposium on Operating System Principles*, V. 15, N. 5. Diciembre de 1981. Pág. 127-136.

[KCM+96] G.N.C. Kirby, R.C.H. Connor, R. Morrison y D. Stemple. *Using Reflection to Support Type-Safe Evolution in Persistent Systems*. Technical Report CS/96/10. University of St. Andrews, Escocia. 1996.

[KEL+62] T. Kilburn, D.B.G. Edwards, M.J. Lanigan y F.H. Summer. "One-level Storage System". *IRE Transactions on Electronic Computers*. Abril de 1962.

También en el capítulo 10 de D.P. Siewiorek, C.G. Bell y A. Newell. *Computer Structures: Principles and Examples*. McGraw-Hill. 1982.

[KF96] T. Kistler y M. Franz. *A Tree-Based Alternative to Java Byte-Codes*. Technical Report No. 96-58. Department of Information and Computer Science, University of California, Irvine, EE.UU. Diciembre de 1996.

[KJS96] Douglas Kramer, Bill Joy y David Spenhoff. *The Java™ Platform White Paper*. JavaSoft. Mayo de 1996.
URL: ftp://ftp.javasoft.com/docs/JavaPlatform.ps

[KLM+93] Gregor Kiczales, John Lamping, Chris Maeda, David Keppel y Dylan McNamee. "The Need for Customizable Operating Systems". En Proceedings of *the 4th Workshop on Workstation Operating Systems*. IEE Computer Society Press. Octubre de 1993. Pág.165-169.

[KP96] G. Kiczales y A. Paepcke. *Open Implementations and Metaobjet Protocols*. Xerox Corporation. 1996
URL: http://www.parc.xerox.com/oi-at-parc/ourpapers/tutorial.ps.

[Kra81] Glenn Krasner. "The Smalltalk-80 Virtual Machine". *Byte*. Agosto de 1981.

[KRB91] G. Kiczales, J. des Rivières y D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press. 1991.

[KV92] J.L. Keedy y K. Vosseberg. "Persistent Protected Modules and Persistent Processes as the Basis for a More Secure Operating System". En Proceedings of the *25th Hawaii International Conference on System Sciences*, Hawaii, EE.UU. 1992. Pág. 747-756.

[Lev84] H.M. Levy. *Capabiliy-Based Computer Systems*. Digital Press. 1984.

[LIS97] *Emergent Technologies Inc. – LispOS.*
URL: http://www.eval-apply.com/LispOS/. Noviembre de 1997.

[LJP93] Rodger Lea, Chistian Jacquemot y Eric Pillevesse. *COOL: System Support for Distributed Object-Oriented Programming*. Technical Report CS/TR-93-68, Chorus systèmes. 1993.

[LMB92] R. Levin, T. Mason y D. Brown. *Lex & Yacc*. O'Reilly and Associates. 1992.

[Luc96] Lucent Technologies. "Inferno: La Commedia Itterativa".
URL: http://inferno.lucent.com/inferno. Noviembre de 1996.

[LW96] Mike Lewis y Andrew Grimshaw. "The Core Legion Object Model". *IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press. Los Alamitos, California, EE.UU. Agosto de 1996.

[LY97] Tim Lindholm y Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley. 1997.

[Mac93a] Stavros Macrakis. *Delivering Applications to Multiple Platforms Using ANDF*. AIXpert. Agosto de 1993.

[Mac93b] Stavros Macrakis. “The Structure of ANDF: Principles and Examples”. URL: ftp://riftp.osf.org/pub/andf/andf_coll_papers/structure.ps. Junio de 1993.

[Mae87] P. Maes. “Concepts and Experiments in Computational Reflection”. En Proceedings of the *1987 OOPSLA*. 1987. Pág. 147-155.

[Mal96] Ashok Malhotra. “Persistent Java Objects: A Proposal”. *First International Workshop on Persistence and Java* (PJ1). Drymen, Escocia. Septiembre de 1996.

[May87] C. May. “MIMIC: A Fast System/370 Simulator”. En Proceedings of the *SIGPLAN’87 Symposium on Interpreters and Interpretive Techniques*, en SIGPLAN Notices, V. 22, N. 7. St. Paul, Minneapolis, EE.UU. Julio de 1987. Pág 1-13.

[MCG96] Ana Belén Martínez Prieto, Juan Manuel Cueva Lovelle y Darío Álvarez Gutiérrez. “Desarrollo de SGBDOO en Oviedo3”. Actas de las *Primeras Jornadas de Investigación y Docencia en Bases de Datos (JIDBD’96)*. La Coruña. Junio de 1996.

[MER97] *The Merlin Project*. URL: http://www.lsi.usp.br/~jecel/merlin.html. Noviembre de 1997.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall. 1997.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, second edition.* Prentice-Hall. 1997.

[MHM96] Peter W. Madany, Graham Hamilton, Tim Mitchell. *JavaOS™: A Standalone Java™ Environment*. JavaSoft. Mayo de 1996.

[Mic91] Microsoft Corporation. *Microsoft C/C++ Version 7.0*. Microsoft. 1991.

[Mic97a] Microsoft Corporation. *Microsoft Visual C++ MFC Library Reference, Part 1*. Microsoft Press. 1997.

[Mic97b] Microsoft Corporation. *Microsoft Visual C++ MFC Library Reference, Part 2.* Microsoft Press. 1997.

[MT96] Jecel Mattos de Assumpção Jr. y Sergio Takeo Kufuyi. *Bootstrapping the Object Oriented Operating System Merlin: Just Add Reflection.* Capítulo 5 de Chis Zimmerman, ed. *Advances in Object-Oriented Metalevel Architectures and Reflection.* CRC Press, 1996.

[NAJ+76] 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 de 1976.

[OAI97] Francisco Ortín Soler, Darío Álvarez Gutiérrez, Raúl Izquierdo Castanedo, Ana Belén Martínez Prieto y Juan Manuel Cueva Lovelle. “El sistema de persistencia en Oviedo3”. *III Jornadas de Tecnologías de Objetos*. Sevilla. Octubre de 1997.

[OMG95] Object Management Group. *Common Object Request Broker Architecture and Specification (CORBA), revision 2*. Object Management Group. Julio de 1995. Disponible en URL: http://www.omg.org.

[OMG97] Object Management Group. *Common Object Request Broker Architecture and Specification (CORBA), revision 2.1*. Object Management Group. Agosto de 1997. Disponible en URL: http://www.omg.org.

[Ort97] Francisco Ortín Soler. *Diseño y Construcción del Sistema de Persistencia en Oviedo3*. Proyecto Fin de Carrera 972001. Escuela Técnica Superior de Ingenieros Industriales e Ingenieros Informáticos de Gijón de la Universidad de Oviedo. Septiembre de 1997.

[Pap89] M. Papathomas. "Concurrency Issues in Object-Oriented Programming Languages". En D. Tsichritzis, ed. *Object-Oriented Development TR*, Centre Universitaire d´Informatique. Universidad de Ginebra, Suiza. 1989.

[PBK91] D. Probert, J.L. Bruno y M. Karaorman. "SPACE: A New Approach to Operating System Abstraction". En Proceedings of the *International Workshop on Object Orientation in Operating Systems (IWOOOS'91)*. Octubre de 1991. Pág. 133-137.

[Piu93] Ian K. Piumarta. *Delayed Code Generation in a Smalltalk-80 Compiler*. Technical Report UMCS-93-7-0. University of Manchester, Inglaterra. 1993.

[POE97] Poet Software. *Poet 4.00 C++ Programmer's Guide*. Agosto de 1997. Disponible en URL: http://www.poet.com.

[Pou88] Dick Pountain. "Rekursiv: An Object-Oriented CPU". *Byte*. Noviembre de 1988.

[PPT+92] Dave Presotto, Rob Pike, Ken Thompson y Howard Trickey. "Plan 9: A Distributed System". En Proceedings of *the USENIX Workshop on Micro-kernels and other Kernel Architectures*. USENIX Association, 4. 1992. Pág. 31-37.

[RAA+92] M. Rozier, V. Abrossimov, F.Armand, I. Boule, M.Gien, M. Guillemont, F. Herman, C. Kaiser, S. Langlois, W. Neuhauser y P. Léonard. "Overview of the Chorus Distributed Operating System". En Proceedings of *the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*. Francia. Abril de 1992. Pág. 39-69

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy y William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall. 1991.

Versión en español: *Modelado y Diseño Orientados a Objetos*. Prentice Hall. 1996.

[Rey96] Franklin D. Reynolds. "Evolving an Operating System for the Web". *ACM Computer*. Septiembre de 1996.

[Rog96] Dale Rogerdson. *Inside COM*. Microsoft Press. 1996.

[RSE+92] S. Russell, A. Skea, K. Elphinstone, G. Heiser, K. Burston, I. Gorton y G. Hellestrand. "Distribution + Persistence = Global Virtual Memory". En Proceedings of the the *International Workshop on Object Orientation in Operating Systems (IWOOOS'92)*. Dourdan, Francia. Septiembre de 1992. Pág. 96-99

[RTA+96] Germán Rodríguez López, Lourdes Tajes Martínez, Fernando Álvarez García, María Ángeles Díaz Fondón, Darío Álvarez Gutiérrez y Juan Manuel Cueva Lovelle. "Project Development in an Advanced OS Subject using the Mach Microkernel". *Bulletin of the Technical Commitee in Operating Systems of the IEEE Computer Society*. Otoño de 1996.

[Rus91] Vincent F. Russo. *An Object-Oriented Operating System*. Tesis Doctoral, Department of Computer Science, University of Illinois at Urbana Champain, Chicago, EE.UU. 1991.

[RW92] Martin Reiser y Niklaus Wirth. *Programming in Oberon: Steps Beyond Pascal and Modula.* ACM Press. 1992

[Sch97] Frank Schubert. "A Reflective Architecture for an Adaptable Object-Oriented Operating System Based on C++". *Eleventh European Conference in Object Oriented Programming (ECOOP'97), Workshop in Object Orientation in Operating Sytems*. Jyväskylä, Finlandia. Junio de 1997.

[SES+96] Margo I. Seltzer, Yasuhiro Endo, Chistopher Small y Keith A. Smith. "Dealing with Disaster. Surviving Misbehaved Kernel Extensions". En Proceedings of the *Second Symposium on Operating Systems Design and Implementation*. Seattle, Washington, EE.UU. Octubre de 1996.

[SFP+96] Emin Gün Sirer, Marc Fiuczynski, Przemyslaw Pardyak y Brian N. Bershad. "Safe Dynamic Linking in an Extensible System". *Workshop on Compiler Support for System Software*. Febrero de 1996.

[SH79] Frank G. Soltis y Roy L. Hoffman. "Design Considerations for the IBM System/38". Digest of Papers, *COMPCON S'79*. 1979. Pág. 132-137.

[Shi96] Olin Shivers. "Supporting Dynamic Languages on the Java Virtual Machine". En Proceedings of the *Dynamic Objects Workshop*. Boston, Massachussets, EE.UU. Mayo de 1996.

[SMF96] Alan C. Skousen, Donald S. Miller y Ronald G. Feigen. *The Sombrero Operating System: An Operating System for a Distributed Single Very Large Address Space – General Introduction.* Technical Report TR-96-005, Arizona State University, EE.UU.1996.

[Smi82] Brian Smith. *Reflection and Semantics in a Procedural Language*. Tesis Doctoral, Massachussets Institute of Technology, EE.UU. 1982.

[Smi84] Brian Smith. "Reflection and Semantics in Lisp". *11th ACM Symposium on Principles of Programming Languages*. Salt Lake City, Utah, EE.UU. Enero de 1984. Pág 23-35.

[SS96] Christopher Small y Margo Seltzer. "A Comparison of OS Extension Methodologies". *USENIX Technical Conference*. Enero de 1996.

[Ste60] T.B. Steel Jr. "UNCOL: Universal Computer Oriented Language Revisited". *Datamation*. Enero/Febrero de 1960. Pág. 18.

[Ste93] Patrick Steyaert. "A Two-Staged Introduction of Reflection Based on Open Implementations". *OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures*. 1993.

[Ste94] Patrick Steyaert. *Open Design for Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks*. Tesis Doctoral, Vrije Universiteit Brussel, Bélgica. 1994.

[Str91] B.Stroustrup. *The C++ Programming Language, Second Edition*. Addison-Wesley. 1991.
Versión en español: *El lenguaje de programación C++, segunda edición*. Addison-Wesley / Díaz de Santos. 1993.

[Sun97] Sun Microsystems. *Java Core Reflection, API and Specification*. Febrero de 1997.

[SW97] Andrew S. Grimshaw y Wm. A. Wulf. "The Legion Vision of a Worldwide Virtual Computer". *Communications of the ACM*, V. 40, N. 1. Enero de 1997.

[TAA+97] Lourdes Tajes Martínez, Fernando Álvarez García, María Ángeles Díaz Fondón, Juan Manuel Cueva Lovelle, Darío Álvarez Gutiérrez y Raúl Izquierdo Castanedo. "Concurrencia en un sistema operativo orientado a objetos basado en una máquina abstracta reflectiva orientada a objetos". *VII Jornadas de Paralelismo*. Cáceres. Septiembre de 1997.

[Taj96] Lourdes Tajes Martínez. "Introducción de la concurrencia en sistemas operativos Orientados a Objetos". *II Jornadas sobre Tecnologías Orientadas a Objetos*. Oviedo. Marzo de 1996.

[Tro97] Eric Trout. "Building the Virtual PC". *Byte*. Noviembre de 1997. Pág. 51-52.

[TUN97] *The Tunes project*.
URL: http://www.eleves.ens.fr:8080/home/rideau/Tunes.html. Noviembre de 1997.

[TYT92] T. Tenma, Y. Yokote y M. Tokoro. "Implementing Persistent Objects in the Apertos Operating System". En Proceedings of *the International Workshop on Object Orientation in Operating Systems (IWOOOS'92)*. Dourdan, Francia. Septiembre de 1992.

[UP87] David Ungar y David Patterson. "What Price Smalltalk?". *ACM Computer*. Enero de 1987. Pág. 67-74.

[US87] D. Ungar y R.B. Smith. "Self, the Power of Simplicity". En Proceedings of the *Conference on Object-Oriented Systems, Languages and Applications*. Orlando, Florida, EE.UU. Diciembre de 1987. Pág 227-242.

[VH96] Alistair C. Veitch y Norman C. Hutchinson. "Kea – a Dynamically Extensible and Configurable Operating System Kernel". En Proceedings of the *Third Conference on Configurable Distributed Systems (IC-CDS'96)*. 1996.

[VRH93] J. Vochteloo, S. Russell y G. Heiser. "Capability-Based Protection in the Mungi OS". En Proceedings of the *IWOOOS'93*. Diciembre de 1993.

[VSK+90] F. Vaughan, T. Schunke, B. Koch, A. Dearle, C. Marlin y C. Barter. "A Persistent Distributed Architecture Supported by the Mach Operating System". En Proceedings of the *1st USENIX Conference on the Mach Operating System*. Burlington, Vermont, EE.UU. 1990. Pág. 123-140.

[Way96] Peter Wayner. “Sun Gambles on Java Chips”. *Byte*. Noviembre de 1996. Pág. 79-88.

[WW93] Mario Wolcko e Ifor Williams. “An Alternative Architecture for Objects: Lessons from the MUSHROOM project”. *OOPSLA’93 Workshop on Memory Management and Garbage Collection*.1993.

[YMF91] Yasuhiko Yokote, Atushi Mitsuzawa, Nobuhisa Fujinami y Mario Tokoro. “Reflective Object Management in the Muse Operating System”. *International Workshop on Object Orientation in Operating Systems (IWOOOS’91)*. Palo Alto, California, EE.UU. Septiembre de 1991.

[Yok92] Yasuhiko Yokote. “The Apertos Reflective Operating System: The Concept and its Implementation”. *International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’92)*. 1992.

[Yok93] Yasuhiko Yokote. “Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach”. En Proceedings of the *International Symposium on Object Technologies for Advanced Software (ISOTAS)*. 1993.

[YTK89] Yasuhiko Yokote, Fumio Teraoka y Mario Tokoro. “A Reflective Architecture for an Object-Oriented Distributed Operating System”. En Proceedings of the *European Conference on Object-Oriented Programming (ECOOP’89)*. Marzo de 1989.

[YTY+89] Yasuhiko Yokote, Fumio Teraoka, Masaki Yamada, Hiroshi Tekuza y Mario Tokoro. “The Design and Implementation of the Muse Object-Oriented Distributed Operating System”. En Proceedings of the *1st Conference on Technology of Object-Oriented Languages and Systems*. Octubre de 1989.

[YW89] Akinori Yonezawa y Takuo Watanabe. “An Introduction to Object-Based Reflective Concurrent Computation”. En Proceedings of the *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*. SIGPLAN Notices, V. 24, N. 4. 1989.