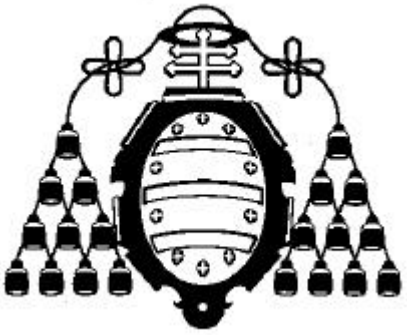
UNIVERSIDAD DE OVIEDO

Departamento de Informática



TESIS DOCTORAL

LIIBUS: Arquitectura de Sistemas Interoperables entre Middlewares Heterogéneos usando Máquinas Abstractas Reflectivas Orientadas a Objetos.

Presentada por:

Francisco Domínguez Mateos

para la obtención del título de Doctor en Informática.

Dirigida por el

Profesor Doctor. D. Juan Manuel Cueva lovelle.

Mayo de 2005

Agradecimientos

A mi madre, a la memoria de mi padre y a mi hermano

Quiero agradecer desde estas líneas todo el apoyo moral y material recibido por parte de todos mis eternos amigos del Departamento de Informática de Oviedo, especialmente a Paco, Belén, Hernán, Elena y María Jesús, que desde que los conocí han estado ahí con un consejo o alguna palabra de aliento.

Gracias también, a mis compañeros y amigos del área de Lenguajes y Sistemas de la Universidad Rey Juan Carlos, con quien he pasado momentos inolvidables y en especial a Rafa, Maxi, Ángel Sánchez y Belén Moreno, quienes me han apoyado en todo lo que en sus manos ha estado para que lleve a buen puerto este trabajo. Un agradecimiento especial para mi compañera de despacho Ascensión. Sin ella habría sido imposible conseguir un acabado tan digno y cuya amistad me ha enriquecido como persona.

A Madjid sin el cual no podría haberse hecho realidad la mención europea de esta tesis y cuyo apoyo siempre ha sido incondicional.

A mis amigos personales Fernando, Rober, Fernando Adrián, Javi López, Javi Hernán, María, gracias a mi contacto con ellos he bajado a la realidad y salido del mundo de las ideas donde tiendo a subir. Una especial mención a Federico, por su noble visión de la vida y la amistad.

Es imprescindible mencionar a mi Director de Tesis, Juan Manuel Cueva, cuyo estímulo constante, apoyo incondicional y su comprensión, al abandonar parte de su tiempo de otros proyectos, para finalizar este, han sido decisivos.

Finalmente, esta Tesis no se habría terminado sin la ayuda de mi querida Irene. Su constante presencia y aliento durante estos últimos meses, donde ha tenido que sufrir mi inevitable estrés, han sido determinantes para afrontar con ganas e ilusión la última parte de su realización.

Resumen

El creciente uso de dispositivos basados en computadoras tanto en el trabajo como en casa, debido a sus muchas aplicaciones, hace que sea muy importante construir sistemas de comunicación entre ellos de forma homogénea. Uno de los principales retos para conseguir un sistema integrado es el problema de la heterogeneidad.

Un sistema heterogéneo es el que está construido por muchas partes diferentes como son: sus interfaces, el software, hardware, sistemas operativos, lenguajes de programación y protocolos de red.

En todos los niveles encontramos muchas soluciones a la heterogeneidad, desde el nivel de heterogeneidad físico, a nivel de enlace, de red, de transporte, etc. En ellos surgieron multitud de problemas que hoy en día están solucionados. Un claro ejemplo de éxito a nivel de transporte y de red es Internet, que nos permite conectar dispositivos completamente heterogéneos, desde teléfonos móviles y computadores, hasta electrodomésticos como la lavadora o dispositivos domóticos. La clave de su éxito es la estandarización global de sus protocolos. Sin embargo a niveles físicos y de enlace, aún no se ha conseguido obtener una solución general, si bien hay multitud de soluciones parciales. Como, por ejemplo, adaptadores, puentes, buses, ruters, conmutadores y túneles.

Uno de los niveles en los que hoy en día no ha sido solucionado el problema de la heterogeneidad es el de middleware o comunicación entre procesos distribuidos. Inicialmente los sistemas middleware se desarrollaron para resolver el problema de heterogeneidad entre comunicación de aplicaciones, a nivel de lenguajes y plataformas y, en este sentido, tuvieron éxito. No obstante la proliferación de este tipo de sistemas ha transferido este problema al de la heterogeneidad entre los propios middlewares. De tal forma que, dos middleware diferentes como CORBA o Java RMI no pueden interoperar.

El problema de la comunicación entre aplicaciones es crucial ya que la tecnología de la información está evolucionando hacia una interacción de aplicaciones a través de la red Internet. Inicialmente, surgieron las aplicaciones en Internet para realizar comunicación entre humanos o entre humanos y aplicaciones. Buena muestra de ello son: Email, Ftp, Web, Telnet, IRC. Pero, hoy en día, la necesidad de interacción entre aplicaciones a través de Internet se está demandando más y más. Por ejemplo, una aplicación de nóminas podría necesitar interoperar con una cuenta bancaria para realizar las transferencias de los sueldos de los empleados y esta interacción puede realizarse sobre algún tipo de plataforma middleware. Pero, ¿qué ocurre si la aplicación de nóminas utiliza Java RMI y el sistema bancario utiliza Servicios Web?

Nuestro objetivo en esta tesis es proponer un modelo arquitectónico que nos permita construir sistemas que solucionen el problema de la interoperabilidad heterogénea entre aplicaciones, tratando de conseguir la mayoría de características descritas más adelante.

Intención de esta tesis

En esta tesis pretendemos definir un modelo arquitectónico para construir buses para plataformas de interoperabilidad middleware heterogéneas, de forma sencilla y estructurada. Para ello, basaremos nuestro modelo en máquinas abstractas reflectivas orientadas a objetos. Pero, para alcanzar estos objetivos, antes hemos tenido que realizar las siguientes tareas:

- Identificar y enmarcar el problema de la interoperabilidad heterogénea entre los sistemas de comunicación de procesos distribuidos.

- Descubrir en qué medida pueden las máquinas abstractas reflectivas orientadas a objetos ayudarnos a conseguir solucionar el problema identificado.

- Definir las características que debe tener un sistema que permita la interoperabilidad entre plataformas heterogéneas.

- Enumerar las técnicas se están aplicando actualmente para resolver el problema de la heterogeneidad. Con este fin, hemos analizado un amplio abanico de documentación acerca de las técnicas que permiten la interoperabilidad heterogénea entre dos o más plataformas específicas.

El modelo arquitectónico debe permitir que las arquitecturas resultantes de su instanciación permitan el desarrollo de sistemas con las siguientes características:

- Heterogeneidad, permitiendo la interoperabilidad entre lenguajes diferentes, sistemas operativos heterogéneos y plataformas middleware distintas.

- Transparencia, de tal forma que elementos de una determinada plataforma vean los elementos de las plataformas externas como si perteneciesen a su misma plataforma.

- Tecnología no intrusiva, no es necesario realizar ningún cambio en las aplicaciones existentes o en las nuevas pensadas para una plataforma específica.

- Extensibilidad, deberá ser sencillo extender el sistema con mecanismos de comunicación futuros. Incluso deberá ser posible construir una extensión de una nueva plataforma desde otra plataforma diferente.

- Adaptación automática, los envoltorios necesarios para la interoperabilidad se crean solamente si son necesarios y de forma automática, sin necesidad de intervención humana o configuración inicial. Aunque, si es necesaria, sea posible realizarla.

- Uniformidad, los objetos externos no tienen ninguna diferencia con los objetos internos.

- Integrabilidad, podemos interactuar con otras aplicaciones o extender el sistema con servicios adicionales para monitorizar, configurar, planificar, etc. Todo esto de forma sencilla, sin que el programador tenga que aprender una plataforma diferente a la que él está acostumbrado.

Aporte de este trabajo

Para conseguir alcanzar los retos expuestos anteriormente, propondremos nuestro modelo arquitectónico que nos proveerá las siguientes novedosas contribuciones:

- Una taxonomía de tecnologías de comunicación utilizadas para resolver el problema de la interoperabilidad heterogénea.

- Definición de una máquina abstracta orientada a objetos básica y genérica, de alto nivel de abstracción.

- Una serie de técnicas para añadir reflectividad estructural y de comportamiento a máquinas abstractas.

- La contribución principal que es LIIBUS, una Arquitectura de Sistemas Interoperables entre Middlewares Heterogéneos usando Máquinas Abstractas Reflectivas Orientadas a Objetos.

- Tres instancias bidireccionales de este modelo para las siguientes plataformas: CORBA, DCOM y Servicios Web.

- Un prototipo basado en la máquina abstracta de Oviedo3, denominada Carbayonia, que permitirá la interoperabilidad de Servicios Web a objetos DCOM.

- La idoneidad de las máquinas abstractas como base para construir motores de transformaciones e interoperabilidad.

- Una proposición para un nuevo paradigma de desarrollo del software utilizando máquinas abstractas como base para la construcción de sistemas software.

Palabras Clave

Interoperabilidad, sistemas heterogéneos, middleware, sistemas distrubuidos, máquinas abstractas, reflectividad, instrospección, arquitecturas, patrones de diseño, objetos, computación masiva, integración de aplicaciones, arquitecturas orientadas a servicios.

Abstract

The ever increasing use of computer based device in work and at home, and the many different applications make very important to build seamless communications. One of the main challenges for achieving an integrated system is the problem of heterogeneity.

A heterogeneous systems is one made of many unrelated parts, including interfaces, software, hardware, operating systems, programming languages, and network protocols.

At all levels we find lot of heterogeneous solutions, since physical level heterogeneity, link level, network, transport, etc. a number of problems aroused and were solved. For instance, Internet is sample of success at network and transport heterogeneity solution, the key of this is the standardisation of it. At levels where as not be able to get a solution as physical or link level, a number of partial solutions are raised as: adapter, bridges, buses, routers, switches and tunnelling.

Now days the middleware or process communication protocol is one of the communication level where it has not been solved the heterogeneous problem. Initially, middleware system where developed to solve the heterogeneity at application communications, and they did. But the boom of middleware system has transferred the problem of heterogeneity to this layout. Now two different middleware platforms as CORBA and Java RMI can not interoperate.

The application communication problem is crucial since the IT technology is evolving to make applications interact between them through Internet network. Firstly arouse human-human or human-application Internet communication applications such as: Email, Ftp, Web, and Telnet. But now days, application to application Internet communications are been demanding more and more. Since B2B communication to C2B, for instance a payslip application could need to interact with a system bank account to carry out salary transactions to company employees, this interaction must be done through some kind of middleware platform. But what happen if the payslip application uses one for instance Java RMI and the system bank account uses one other different as Web Services?

Our proposal is to define an architectural model to build systems that allow heterogeneous interoperability between applications with most of the features indicated and described bellow.

Thesis aims

It is the aim of this thesis to define an architectonical model to build heterogeneous interoperable middleware platforms buses, in an easy and structured way. To allow that, this model will be based upon object oriented reflective abstract machines. In order to achieve these objectives we first have to address some issues.

- Discover the heterogeneous process communication interoperability problem.

- How object oriented abstract machines and reflection can support interoperability?

- What are the requirements to build heterogeneous interoperable platforms?

- What existing techniques are appropriate in developing solutions for theses requirements? The literature survey shall examine the building blocks of that enable heterogeneous interoperability between two or more specific platforms.

The architectural model designer should allow the architecture instance enables the development of systems with the next requirements:

- Heterogeneity, allows heterogeneous language interoperability, operative systems different, and heterogeneous middleware platforms.

- Transparencies, elements from a specific platform, see elements from other external platforms as if they belong to the same platform.

- Not intrusive technology, there is not necessary to do changes nor to existing applications or the new ones designed for a specific platform.

- Extensibility, it could be easy extend the system with future communication mechanisms. It would be possible to build an extension for a new platform in other different platform.

- Adaptation Automated, proxies or wrappers are created only if they are needed and in an automatically way, without any human being intervention or configuration.

- Uniformity, external objects have not any difference with internal ones.

- Integrabillity, we can interact with other applications or extend the model with additional services to monitoring, configuring, scheduling, etc. In an easy way, without the programmer has to learn a new different platform of that they are used to work with.

Novel aspects of this work

In order to address the issues discussed earlier we proposed an Architecture that has the following novel contributions:

- A taxonomy of heterogeneous technologies used to solve heterogeneity into interoperability problem.

- Definition of a general object oriented abstract machine.

- Techniques to add structural o behavioural reflection to an abstract machine.

- The main contribution, that is, System Interoperable Architecture between Heterogeneous Middleware using Object Oriented Reflective Abstract Machines.

- Tree instantiations of this model in both directions for each one of these platforms: CORBA, DCOM, Web Services.

- A prototype based upon the Oviedo3 abstract machine named Carbayonia that allows interoperability from Web Services to DCOM.

- The suitability of abstracts machines as a basement for interoperable invocation and transformations engines.

- A proposal for a new software developing paradigm using abstract machines as basement for software system.

Keywords

Interoperability, heterogeneous systems, middleware, distributed systems, abstractats machines, reflection, introspection, architectures, design patterns, objects, masive computation, applications integration, service oriented applications.

Índice Resumido

Índice

Ilustraciones

Capítulo 1:
Introducción

De todos es conocida la revolución informática que ha tenido lugar en lo últimos años. Hoy en día todo el mundo tiene un ordenador en casa, trabaja con él o manipula uno, directa o indirectamente. Pensemos que cualquier electrodoméstico de nuestros hogares lleva incorporado ya, no uno, sino varios procesadores.

En estos momentos estamos asistiendo a una segunda revolución, nos encontramos en sus albores, me refiero a las telecomunicaciones informáticas. Hace unos años comenzó el boom de Internet, todo nos apresuramos a conectarnos con un modem lento y tosco, enchufado al puerto serie de nuestro ordenador. Actualmente nos encontramos en pleno auge de las comunicaciones de banda ancha con las líneas telefónicas de ADSL y las de comunicación por cable, satélite, radio, etc. Pero esta revolución no ha hecho mas que empezar. En un futuro no muy lejano dispondremos de líneas de comunicación de decenas de Gigabits por segundo, miles de veces más rápida que las líneas de banda ancha que utilizamos ahora. Dichas líneas nos proporcionarán todo un mundo de tele-media-comunicación, donde dispondremos de videoconferencia de muy alta calidad de imagen, televisión a la carta, teletrabajo colaborativo o ubicuo, etc. Pero antes de llegar a esto, es necesario solucionar aun muchas barreras tecnológicas, tanto a nivel hardware como software. Este trabajo pretende aportar un pequeño granito de arena, investigando cómo eliminar, en parte, alguna de estas dificultades, en concreto, la de los protocolos de comunicación entre los programas de ordenador.

El problema de la comunicación entre aplicaciones es crucial ya que la tecnología de la información está evolucionando hacia una interacción de aplicaciones a través de la red Internet. Inicialmente surgieron las aplicaciones en Internet para realizar comunicación entre humanos o entre humanos y aplicaciones. Buena muestra de ello

son: Email, Ftp, Web, Telnet, IRC. Pero hoy en día, la necesidad de interacción entre aplicaciones a través de Internet se está demandando más y más. Por ejemplo, una aplicación de nóminas podría necesitar interoperar con una cuenta bancaria para realizar las transferencias de los sueldos de los empleados.

Para solucionar el problema de las comunicaciones entre aplicaciones, se desarrollaron los sistemas middleware. Estos permiten resolver el problema de heterogeneidad entre comunicación de aplicaciones, a nivel de lenguajes, sistema operativo, sistemas hardware y sistemas de comunicación. En este sentido, tuvieron éxito. Actualmente han surgido gran variedad de plataformas y sistemas middleware como son DCOM, CORBA, Java RMI y Servicios Web.

Volviendo con el ejemplo anterior, la aplicación de nómicas y las aplicaciones bancarias necesitan interoperar y esta interacción puede realizarse sobre algún tipo de plataforma middleware. Pero ¿qué ocurre si la aplicación de nóminas utiliza Java RMI y el sistema bancario utiliza Servicios Web? Este es un ejemplo ilustrativo del problema de interoperabilidad entre plataformas middleware heterogéneas y donde se enmarca el trabajo realizado en esta tesis.

Han surgido una gran cantidad de técnicas que permiten resolver el problema planteado en el ejemplo anterior, desde soluciones toscas como reconstruir el programa de nóminas para que interaccione con Servicios Web a utilizar técnicas de Integración de Aplicaciones de Empresas que consisten en incorporar enormes sistemas software sobre potentes servidores que, tras una serie de sofisticadas operaciones de configuración permiten realizan todo tipo de transformaciones de datos y comunicaciones.

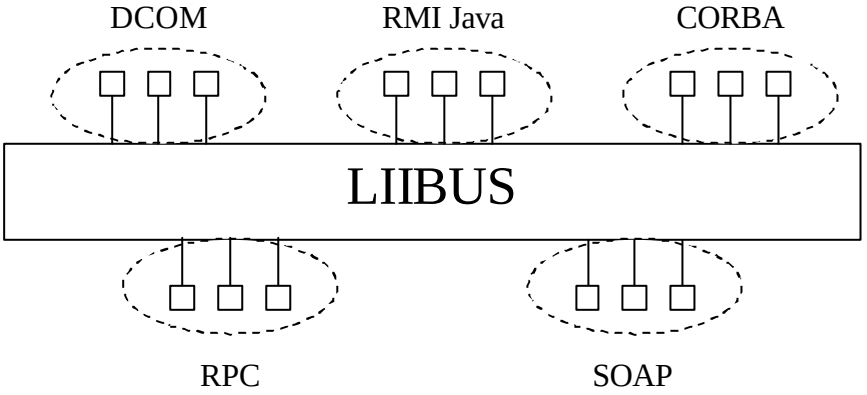
Nuestro planteamiento consiste en instalar una capa software que permita, sin necesidad de configuración ni de modificación de ninguno de los sistemas, que ambas aplicaciones se comuniquen. Esta capa software la vamos a construir aprovechando las ventajas que nos aportan las máquinas abstractas reflectivas. Realmente lo que propone esta tesis es un modelo arquitectónico para construir dicho software, más que una capa concreta basada en una máquina abstracta concreta.

En el siguiente capítulo describiremos con algo más de detalle los objetivos planteados en esta tesis y cómo han sido llevados a cabo.

Capítulo 2:
Objetivos

El objetivo principal e inicial de esta tesis es el desarrollo de:

"un modelo arquitectónico basado en máquinas abstractas, que dé lugar a sistemas software que permiten la interoperabilidad entre diversas y heterogéneas plataformas de middleware".



A lo largo de este trabajo llamaremos a estos sistemas Buses Middleware o middlebuses. Este nombre surge debido a que su comportamiento es similar al de un bus hardware pero aplicado a middleware, es decir, si dentro del bus están conectados varios elementos y se añade uno nuevo, este puede comunicarse con los elementos anteriores y viceversa. Lo mismo ocurre con los middlebuses, si en uno determinado hay conectados varios middleware y se conecta uno mas, este podrá interoperar con todos los anteriormente conectados y a su vez estos también podrán hacerlo con el recién llegado.

Todo esto se consigue de forma transparente y no intrusiva en la capa de middleware, es decir, no hay que hacer ninguna modificación en las aplicaciones existentes. Así, por ejemplo, un objeto CORBA puede utilizar un objeto RMI Java de forma completamente transparente. Este modelo arquitectónico permitirá que los sistemas construidos a partir de él tengan las siguientes características:

- Heterogeneidad, permite la interoperabilidad de lenguajes, sistemas operativos y además middlewares, heterogéneos.

- Transparencia, los elementos de una plataforma origen ven a los elementos de las demás plataformas destino como si fuesen elementos de la misma plataforma origen.

- Tecnología no intrusiva, es decir, no hay que hacer cambios, ni en las aplicaciones ya existentes que utilicen los middleware heredados ni las nuevas aplicaciones se deben diseñar para utilizar este nuevo sistema.

- Extensibilidad, el sistema podrá ampliarse fácilmente con nuevos sistemas de comunicación futuras, incluso aunque no sean middleware. Se puede construir una extensión de una plataforma en otra plataforma diferente a la original.

- Eficiencia, no se realiza ningún tipo de conversión temporal, ni ningún canal intermedio.

- Completamente automatizado, la creación de proxys de los objetos a utilizar se realiza de forma automática, pretendiendo minimizar las tareas de configuración.

- Uniformidad, los objetos externos no tendrán diferencia alguna con los objetos internos.

- Integrabilidad, se podrán integrar elementos de otras plataformas o sistemas a nuestros sistemas de forma sencilla. Nuestros desarrolladores no tendrán que tener conocimientos sobre las plataformas heredadas que integramos.

Desarrollaremos un modelo arquitectónico general, basado en máquinas abstractas reflectivas orientadas a objetos, que modela nuestro sistema y, posteriormente, veremos varias instanciaciones concretas aplicadas a varias plataformas específicas como son CORBA, DCOM y Servicios Web.

El modelo arquitectónico será construido siguiendo el Proceso de Desarrollo Unificado [PUD00] y viene expresado en UML, el Lenguaje de Modelado Unificado [UML99], que se ha convertido en el estándar para el modelado de sistemas orientados a objetos. Se ha seguido un método *iterativo e incremental*, realizando varias iteraciones sobre los Flujos Fundamentales: Requisitos, Análisis, Diseño, Implementación y Pruebas. Por otro lado, solo hemos realizado iteraciones sobre las Fases de Inicio y Elaboración ya que se pretende obtener un diseño arquitectónico que, precisamente, es el hito fundamental de la Fase de Elaboración. Nos hemos centrado en el concepto de *mecanismo* definido en el PUD para definir una colaboración genérica a partir de

Patrones de Diseño [Gamma00] y así obtener un **modelo arquitectónico orientado a patrones y basado en máquinas abstractas reflectivas orientadas a objetos** [Busch96]

La mayoría del código ejemplo vendrá dado en Smalltalk [Gold83] ya que la máquina general abstracta (MA) definida como abstracción sencilla de alto nivel de máquinas abstractas, ha sido ideada en este lenguaje, utilizando para ello la plataforma Squeak [Squeak05].

Para comprobar que las arquitecturas resultantes del modelo arquitectónico funcionan adecuadamente y con el objetivo de realizar trabajos de campo, al poder evaluar una implementación real de una arquitectura concreta del modelo arquitectónico, construiremos un prototipo basándonos en la maquina abstracta orientada a objetos llamada Carbayonia, definida dentro del sistema integral orientado a objetos denominado Oviedo3, que llamaremos OviBus.

En la siguiente sección, describiremos cómo están organizados los capítulos de este libro y qué tareas de las necesarias para obtener los objetivos propuestos anteriormente han sido llevadas a cabo en cada uno de ellos.

Capítulo 3:
Organización

La estructura de este libro está formada por cinco grandes bloques: contexto, antecedentes, planteamiento, solución y viabilidad. Estos bloques abarcan del Capítulo 4 al Capítulo 18. A continuación los expondremos desde el punto de vista de cómo el autor[†] ha ido adquiriendo conciencia de cada una de ellas.

Esta tesis ha sido desarrollada como resultado del trabajo realizado a lo largo de varios años de investigación en el campo de los sistemas distribuidos, los sistemas orientados a objetos, los sistemas de componentes, las máquinas abstractas, los sistemas reflectivos, la teoría de autómatas y los lenguajes formales y la ingeniería del software. Esto ha provocado que el autor, durante este tiempo, haya ido desarrollando una visión general tanto del estado del arte como del estado de la solución.

3.1 Antecedentes

De esta forma, ha ido descubriendo qué limitaciones encuentran actualmente los sistemas middleware, dando como resultado un estudio sobre los sistemas distribuidos, varios sistemas middleware y los sistemas que pretenden resolver la heterogeneidad. Dichos estudios han llevado a la elaboración del "Capítulo 6: Sistemas Distribuidos", "Capítulo 7: Plataformas LII" y "Capítulo 8: Heterogeneidad en sistemas distribuidos" respectivamente. Formando estos capítulos lo que se viene denominando estado del arte o antecedentes previos.

[†] Partes de este capítulo lo expresaremos en tercera persona, debido a que haremos referencia explícita al autor. El lector observará que en el resto del libro utilizamos principalmente la primera persona del plural.

3.2 Contexto

Por otro lado, y entrando en el dominio de la solución, identificó cómo pueden las máquinas abstractas y los sistemas reflectivos ayudar a construir sistemas, aportando sus cualidades subyacentes. Estos resultados han sido expuestos en el "Capítulo 4: Máquinas abstractas" y en el "Capítulo 5: Reflectividad". Estos capítulos, pese a no pertenecer al estado del arte propiamente dicho, si aportan el contexto en el que se enmarca la solución planteada y resultan imprescindibles para comprenderla.

El resto de capítulos del libro forman parte de la solución. Primero, realizaremos su planteamiento, posteriormente, la construimos y, para finalizar, trataremos de demostrar su viabilidad.

3.3 Planteamiento

Una vez analizados los requisitos que deben cumplir los sistemas distribuidos, identificado el problema de la interoperabilidad entre middleware heterogéneos y evaluadas las soluciones existentes actualmente en el mundo de los sistemas distribuidos, se procede a identificar la ausencia de alguna cualidad o característica en ellos y a plantear su aproximación tratando de eliminar dicha ausencia. Esto es lo que se realizar en el "Capítulo 9: Planteamiento del problema". Posteriormente, se detallan todos los requisitos que el planteamiento anterior lleva consigo, que se describen en el "Capítulo 10: Requisitos del sistema" y, finalmente, identifican cualidades y restricciones del diseño, expuestas en el "Capítulo 11: Cuestiones de diseño".

Una vez planteados los objetivos, en nuestro caso, la realización de un modelo arquitectónico, es el momento de llevarlos a cabo a través de los siguientes capítulos.

3.4 Solución

En el "Capítulo 12: Una máquina abstracta orientada a objetos" entran en juego los conocimientos adquiridos en máquinas abstractas, sistemas orientados a objetos y teoría de autómatas y lenguajes formales, dando como resultado la creación de una máquina abstracta general de alto nivel de abstracción orientada a objetos y denominada MA, que nos servirá como pilar fundamental sobre el que basaremos nuestro modelo arquitectónico.

En el "Capítulo 13: Reflectividad en máquinas abstractas" se ponen en práctica los conocimientos adquiridos en el estudio de los sistemas reflectivos, definiendo un modelo para incorporar reflectividad estructural a máquinas abstractas, utilizando como patrón la máquina básica general MA.

Una vez expuestos los pilares necesarios para construir nuestro modelo arquitectónico procedemos, en el "Capítulo 14: Diseño arquitectónico", a realizar una primera aproximación y, en el "Capítulo 15: Colaboración genérica", a construir una versión más detallada del mismo. En estos capítulos fueron necesarios los

conocimientos de ingeniería del software principalmente en procesos de desarrollo, patrones de diseño y arquitecturas orientadas a patrones.

Queda, a partir de este momento, expuesta y finalizada nuestra solución al problema de la interoperabilidad de sistemas middleware heterogéneos para invocación de métodos remota. El resto de capítulos tratará de comprobar la viabilidad del modelo arquitectónico desarrollado.

3.5 Viabilidad

Para comprobar la viabilidad de la solución propuesta, hemos dedicado los siguientes tres capítulos: "Capítulo 16: Varias instanciaciones de la colaboración", "Capítulo 17: Implementación de un prototipo", "Capítulo 18: Evaluación del modelo". En el primero comprobamos cómo se adapta nuestro modelo arquitectónico a diversas plataformas middleware, para ello creamos varias instancias del mismo, en concreto, para CORBA, DCOM y Servicios Web. Se comprueba que cualitativamente, a nivel de diseño arquitectónico los resultados obtenidos son buenos. En el siguiente capítulo, se desarrolla un prototipo concreto basado en una máquina abstracta real, Carbayonia del Sistema Integral Orientado a Objetos Oviedo3, así comprobamos que la implementación de los diseños arquitectónicos resulta viable. En el último capítulo, tratamos de realizar valoraciones tanto cuantitativas como cualitativas de las dos aproximaciones anteriores, llegando a la conclusión de las instancias del modelo cumplen muy aceptablemente los requisitos planteados anteriormente y que los sistemas software construidos a partir de ellas funcionan adecuadamente, si bien los resultados en términos de rendimiento no resultan muy eficientes.

Capítulo 4:
Máquinas abstractas

En este capítulo se define el concepto de máquina abstracta como concepto contrapuesto a máquina virtual y máquina real o física. Así mismo veremos que ventajas nos aportan las máquinas abstractas, de las cuales podrán beneficiarse los sistemas o arquitecturas que, como la nuestra, las utilicen como base para la resolución de un determinado dominio de problema.

Al final del capítulo se comentará qué ventajas aporta el concepto de máquina abstracta a esta tesis y se establecerá una clasificación de características de las mismas que resultan útiles en el desarrollo de la misma.

4.1 Procesadores

Un programa es un conjunto ordenado de instrucciones que se dan al ordenador indicándole las órdenes necesarias para llevar a cabo la ejecución de algún determinado algoritmo [Cueva94]. Los procesadores computacionales son los encargados de ejecutar, animar o interpretar a los programas.

Todo programa lleva asociado, además de las instrucciones que lo describen, un conjunto que datos sobre los que realiza consultas, modificaciones o creaciones de los mismos. El objetivo final de todo programa es el de procesar un conjunto de datos, este procesamiento puede ir dirigido por otros datos, que a su vez pueden también modificarse.

Se puede ver un procesador como un programa en el que parte de sus datos, las instrucciones, son utilizados para procesar otros datos. La ejecución de estos programas puede ser realizada de dos formas: **física**, cuando es una máquina (hardware) la que realiza la propia ejecución del programa, o **lógica,** cuando la ejecución del programa es llevada a cabo por otro programa (software).

4.1.1 Procesadores Físicos

Un procesador hardware es una implementación física de un programa procesador computacional, los *datos instrucciones* que lo constituyen se ven sustituidos por elementos funcionales, secuenciales y de memoria físicos. Estos elementos físicos suelen ser implementados con tecnología electrónica digital en forma de chips.

A la hora de realizar procesadores físicos, existen varias configuraciones arquitectónicas para coordinar estos elementos funcionales, secuenciales y de memoria. Una de las arquitecturas más utilizadas y que tienen una correspondencia directa con el modelo de programa procesador con datos para instrucciones y datos calculables es la denominada arquitectura de Harvard [Aguilar04]. En la Ilustración 4.1 se muestra dicha arquitectura.

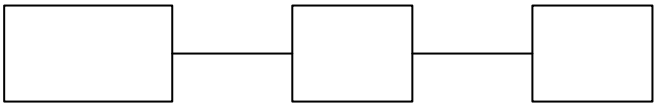


Ilustración 4.1: Arquitectura Harvard

Vemos que este procesador o Unidad Central de Procesos (CPU) dispone de dos entradas de datos diferenciadas por un lado las instrucciones y por otro los datos calculables.

Sin embargo la arquitectura más utilizada es la de Von Neumann [Mora02] donde se unen dentro de la misma memoria física las instrucciones y los datos. Con este modelo se pueden procesar las instrucciones como si fuesen datos, si bien el sistema no es tan eficiente como el anterior.

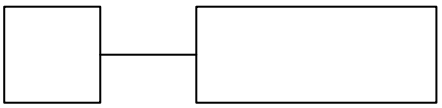


Ilustración 4.2: Arquitectura Von Neumann

Normalmente estas arquitecturas se implementan con dispositivos digitales, donde los elementos funcionales vienen creados por Unidades Aritméticas y Lógicas (ALU), los elementos secuenciales vienen implementados a partir de Unidades de Control (UC) síncronas y los elementos de memoria vienen dados por registros y memorias digitales [Dormi00].

Si bien los procesadores físicos son extremadamente eficientes, su flexibilidad resulta muy pobre debido, como ya hemos mencionado, a que las instrucciones se han convertido en elementos físicos difícilmente cambiables o manipulables. A pesar de que los programas que ejecutan sí pueden cambiar, modificando la secuencia de las instrucciones en memoria, las operaciones que definen las instrucciones son invariables.

4.1.2 Procesadores Software

Un procesador software o lógico es la realización de un procesador computacional mediante un programa que será, a su vez, interpretado por otro

procesador. Este último, normalmente, es un procesador hardware, pero puede ser de nuevo otro procesador software.

Se pueden construir procesadores software para interpretar programas de otros procesadores, estos últimos suelen ser físicos [Cueva98]. En este caso, se utilizan como herramientas de simulación de esos futuros procesadores físicos. Esto se hace así debido a la flexibilidad que aportan los procesadores lógicos, al ser mucho más sencilla su modificación.

La característica de flexibilidad que ofrecen los procesadores lógicos es en la que radica su potencia y utilidad en determinados contextos. Si bien es cierto que, un procesador software siempre va a ser interpretado por otro procesador, con lo que la velocidad de ejecución de los programas va a ser mucho menor que la velocidad de ejecución de programas en procesadores físicos.

4.2 Emuladores, Máquinas abstractas y Máquinas virtuales

Cuando se diseña un procesador computacional sin intención de que sea desarrollado de forma física [Howe99] se le denomina **máquina abstracta**. Procesador desarrollado físicamente es la implementación hardware de una determinada máquina abstracta [Alv98]. La especificación formal de la semántica del juego de instrucciones de un procesador computacional se diseña apoyándose en las modificaciones de los estados de su máquina abstracta.

Se denominan **emuladores** de microprocesadores físicos a los procesadores software que simulan el comportamiento de procesadores físicos sobre otros procesadores físicos diferentes. Un ejemplo puede ser el emulador de Z80 para procesadores de la familia Intel que permite ejecutar software del procesador Z80 sobre ordenadores personales con procesadores de la familia i86 de Intel, ahora denominados Pentium. Normalmente, cuando se habla de emuladores, se refiere a la emulación no solo del procesador si no de toda su plataforma software y hardware. Así, hoy en día, hay multitud de emuladores de unos equipos en otros. Por ejemplo, emuladores de máquinas recreativas en ordenadores personales o incluso emuladores de videoconsolas Game Boy para teléfonos móviles con plataforma Symbian.

Existe un homomorfismo entre intérpretes y procesadores computacionales, de tal forma que un procesador computacional, implementado física o lógicamente es un intérprete del lenguaje que procese. Consideramos un intérprete como un programa que ejecuta las instrucciones de un lenguaje. Las instrucciones se encuentran, normalmente, en un archivo fuente [Cueva98].

Se define **máquina virtual** como un intérprete de una máquina abstracta[2] [Howe99]. Una máquina virtual es, por tanto, un intérprete definido sobre una máquina abstracta. De esta forma, la máquina abstracta es utilizada en la descripción semántica de las instrucciones de dicho intérprete.

[2] Aunque el concepto de máquina abstracta y máquina virtual no son exactamente idénticos, son comúnmente intercambiados.

4.3 Beneficios aportados por las Máquinas abstractas

A continuación mostraremos los beneficios que nos aportan los computadores lógicos representados como máquinas abstractas a pesar del detrimento en rendimiento obtenido en comparación con los procesadores físicos. En posteriores capítulos veremos cuales de estos beneficios serán rentables en los sistemas que se desarrollarán a partir de nuestro modelo arquitectónico basado en ellas.

4.3.1 Formalización

Una de las primeras utilizaciones de máquinas abstractas fue en el campo de la teoría de lenguajes formales y autómatas. En este sentido, una máquina abstracta es un mecanismo lógico ideado para ejecutar un conjunto de instrucciones de un lenguaje formal [Howe99]. De esta forma, una máquina abstracta define una estructura sobre la cual es posible representar la semántica computacional del lenguaje formal asociado. Esta acepción es similar a la vista en la sección 4.2, aunque más rigurosa y formal, y está enfocada más a estudios de computabilidad que a aplicaciones prácticas.

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

Las máquinas abstractas de estados proporcionan el nexo entre los métodos formales de especificación y los modelos de computación [Huggins99]. Éstas amplían la tesis de Turing [Turing36], especificando máquinas más versátiles, capaces de simular cualquier algoritmo en su nivel de abstracción natural (no a bajo nivel) y de forma independiente al lenguaje de codificación utilizado.

Esta línea de investigación, que utiliza el concepto de máquina abstracta para especificar formalmente algoritmos computacionales en su nivel de abstracción natural, está fuera del estudio realizado en esta memoria. Utilizaremos el concepto de máquina abstracta para diseñar un modelo arquitectónico con la mayoría del resto de características positivas mostradas aquí como la portabilidad, multiplataforma, interacción, etc. y no para profundizar en el campo de lenguajes formales o en el de la formalización computacional.

4.3.2 Independencia del lenguaje

En teoría de compiladores e intérpretes se ha utilizado ampliamente el concepto de máquina abstracta. El proceso de compilación toma un lenguaje de alto nivel y genera un código intermedio. Este código intermedio es propio de una máquina abstracta. Con ello se consigue:

- Simplificar el diseño de compiladores [Cueva94] al generar todos los compiladores el código para la misma máquina, y compartir, así, estructuras de traducción.
- Independencia del lenguaje al generar el código, de tal forma que se diseña una máquina abstracta b más general posible,

para que se pueda traducir de esta a cualquier máquina real existente. Para generar el código binario de una máquina real, tan sólo hay que traducir el código de la máquina abstracta a la máquina física elegida, independientemente del lenguaje de alto nivel que haya sido compilado previamente.

- Ahorro en traducción de código de un sistema a otro, es decir si tenemos un código intermedio, para construir un compilador de C++ y Pascal a Intel 80286 y Motorola 68000 habría que realizar cuatro compiladores diferentes. En general para *n* lenguajes a *m* plataformas hay que generar *n***m* compiladores diferentes. Si disponemos de una maquina abstracta intermedia, para *n* lenguajes generaremos *n* compiladores que traduzcan al lenguaje de la máquina abstracta y para *m* procesadores necesitaremos otros m traductores que traduzcan del lenguaje de la máquina abstracta al procesador adecuado, con lo que necesitaremos solamente *n*+*m* compiladores. Esto supone un gran ahorro en recursos temporales y humanos. Se han hecho varios intentos para conseguir esto a nivel universal. En esta línea, el proyecto UNCOL (Universal Computer Oriented Lenguaje) proponía un lenguaje intermedio universal para el diseño de compiladores [Steel60]. El objetivo de este proyecto era especificar una máquina abstracta universal para que los compiladores generasen código intermedio a una plataforma abierta. Si bien en la época de los 60, la tecnología de lenguajes y compiladores no estaba lo suficiente consolidada como para soportar dicho proyecto y fue con ANDF (Architectura Neutral Distribution Format) [Macrakis93], cuyo objetivo era un híbrido entre la simplificación de compiladores y la portabilidad de código (punto siguiente), cuando se consiguió. Un compilador podría generar código para la especificación de la máquina ANDF siendo este código portable a distintas plataformas. Este código ANDF no era interpretado por un procesador software sino que era traducido (o instanciado) a código binario de una plataforma específica. De esta forma se conseguía lo propuesto con UNCOL: la distribución de un código de una plataforma independiente.

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

especificación de una plataforma o máquina abstracta independiente del sistema operativo que permita desarrollar aplicaciones en distintos lenguajes y sistemas operativos. Este proyecto bautizado como "Proyecto Kylix" especifica un modelo de componentes CLX (Component Library Cros-Paltform) que permite desarrollar aplicaciones en C++ Builder y Delphi para los sistemas operativos Win32 y Linux [Kozak00]. Actualmente es la CLR (Common Language Runtime) de .NET la que esta obteniendo mayor difusión tanto a nivel comercial como de investigación [Msdn01].

- Entornos de programación multilenguaje, apoyándose de forma directa en los conceptos de máquinas abstractas y máquinas virtuales, se han desarrollado entornos integrados de desarrollo de aplicaciones multilenguaje. El que actualmente está teniendo más éxito comercial es Visual Studio .NET, en el se pueden integrar tanto un compilador de cualquier lenguaje como herramientas de diseño. Donde la máquina abstracta final utilizada es la definida por el CLR en la plataforma .NET.

4.3.3 Multiplataforma

Antes de ver la utilidad de las máquinas abstractas en los sistemas multiplataformas, definiremos el concepto de plataforma. En nuestro contexto, vamos a considerar plataforma como "la combinación de un procesador y un sistema operativo".

Los beneficios de la multiplataforma se obtienen apoyándose en la noción de emulación, de máquinas abstractas y de su capacidad para especificar un sistema no solo independiente del procesador sino también de la plataforma, es decir, emulando, además, el sistema operativo. Todo esto es posible gracias a la ausencia de una implementación física, que es una de las características de las máquinas abstractas.

De esta forma si un compilador genera, a partir de un programa escrito en un lenguaje de alto nivel, código para una máquina abstracta determinada dicho código podrá ser ejecutado en cualquier plataforma para la que se haya creado un procesador lógico capaz de interpretar las instrucciones de la máquina abstracta para la que se generó. Dicho programa podrá ser ejecutado por cualquier procesador, ya sea hardware o software, que implemente la especificación computacional de un procesador de la máquina abstracta.

Es en esta característica donde actualmente las máquinas virtuales están siendo más utilizadas. A pesar de que hace ya casi 30 años que esta idea de multiplataforma se desarrolló, con la máquina virtual de SmallTalk, el auge comenzó en 1995 con la expansión de Internet y el desarrollo, por parte de Sun Microsystems, del lenguaje Java y su máquina virtual JVM (Java Virtual Machine). Actualmente Microsoft proclama que ha mejorado la máquina y el lenguaje a través de su C# y su máquina virtual basada en CLR, que junto con la definición de servicios distribuidos Web Services, constituyen la plataforma .NET. Sin embargo sigue investigando para obtener, por un lado, un CLR reflectivo [Ortin05] y, por otro, la mejora de C# con un nuevo leguaje denominado C? [Serrano05]

4.3.4 Código móvil

Una de las consecuencias del código multiplataforma visto previamente en 4.3.3 es la posibilidad de obtener código móvil. Es decir, los programas van a poder pasar de una máquina a otra a través de una red de comunicación de computadores y, una vez en su destino, se podrán ejecutar. El aporte de las máquinas abstractas es que el ordenador origen, que suele ser un servidor, y el ordenador destino no tiene por qué ser iguales en nada, ni en procesador ni en sistema operativo ni en hardware. De tal forma que vamos a poder construir sistemas software con código móvil entre plataformas heterogéneas de computadores.

Los Applets son el caso típico de código móvil. Son programas que se encuentra en servidores Web y su código es código Java restringido denominado bytecode [Kramer96]. El cliente, mediante su navegador Web, se conecta al servidor Web donde encuentra los programas utilizando el protocolo HTTP [Marshall03] o HTTPS [Freier96] y descarga el código Java en su navegador. La aplicación es interpretada por la máquina virtual de Java [Sun95] implementada en el navegador de forma independiente a la plataforma y el navegador que son utilizados por el cliente.

Otro típico ejemplo de código móvil son los Agentes Móviles [UBMC], que son programas que van de ordenador en ordenador interactuando localmente y recopilando información según un determinado plan. Estos aportan numerosas ventajas y, entre ellas, la más interesante es la de ahorro en ancho de banda consumido. Si bien presentan un problema de seguridad importante tanto por parte de los sistemas que reciben los agentes como por parte de la información recopilada por los mismos [Perez00].

4.3.5 Interoperabilidad de aplicaciones

Otra ventaja en la utilización de una plataforma abstracta comentada previamente en el apartado 4.3.3 es el hecho de que las aplicaciones pueden interoperar entre sí, con envío y recepción de datos, de forma independiente de la plataforma física sobre la que estén ejecutándose. La representación nativa de la máquina abstracta será interpretada por la máquina virtual de cada plataforma.

Vamos a tener dos tipos de interoperabilidad. Por un lado, la interoperabilidad local, es decir, dos aplicaciones van a poder ejecutarse sobre la misma máquina abstracta y poder interactuar fácilmente dentro del mismo espacio de memoria y compartiendo las mismas representaciones de datos. Esto es posible, gracias a que el código para ambas aplicaciones ha sido desarrollado para la misma máquina abstracta. Por otro lado, esta misma interoperabilidad puede producirse aunque las aplicaciones se encuentren en máquinas localizadas geográficamente en diferentes ubicaciones. La comunicación va a resultar sencilla ya que compartirán el mismo formato de datos y estructuras de información, salvo que, en este caso se necesitará un mecanismo común de comunicación.

Este es el caso de protocolo RMI (Remote Method Invocation) de Java [Gosling96], con lo que podemos enviar los datos en su representación nativa. No estamos restringidos al envío y recepción de tipos simples de datos, sino que es posible también enviar objetos, convirtiéndolos a una secuencia de bytes, *serialization* [Campione99].

Mediante el paquete de clases ofrecidas en java.net, es posible implementar un protocolo propio de un tipo de aplicación sobre TCP/IP o UDP/IP [Raman98]. Si deseamos obtener un mayor nivel de abstracción para el desarrollo de aplicaciones distribuidas, podemos utilizar RMI sin sobrecargar tanto la red de comunicaciones [Sun97]. RMI desarrolla un protocolo de interconexión de aplicaciones Java denominado JRMP (Java Remote Method Protocol) que permite, entre otras cosas, invocar a métodos de objetos ubicados en otras máquinas virtuales.

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

El uso de máquinas abstractas para obtener interoperabilidad entre aplicaciones distribuidas es ventajosa con respecto a otras tecnologías que tratan de solucionar la interoperabilidad entre aplicaciones sin su concurso. Es este el caso de CORBA [OMG95], donde se define un protocolo de interoperabilidad general GIOP (General Inter-ORB Protocol) [OMG04]. En este caso surgen los siguientes inconvenientes:

- Requiere una elevada cantidad de código adicional para implementar el protocolo y la traducción de la información enviada por la red. Este código recibe el nombre de ORB (Object Request Broker).

- Aumenta el volumen de información enviada a través de la red de ordenadores al implementar un protocolo de propósito general que proporciones un mayor nivel de abstracción.

- Se obtiene lentitud e imprecisión en las conversiones de representación de información para diferentes plataformas y lenguajes de programación.

4.3.6 Sistemas interactivos

En los años 80, se introdujo el concepto de programación orientada a objetos, con lo que el nivel de abstracción de los lenguajes se elevó considerablemente [Booch94], dando lugar a sistemas con abstracciones más cercanas a la forma de pensar del ser humano. Smalltalk fue uno de los primeros sistemas orientados a objetos. Su propósito era definir no sólo un lenguaje sino todo un entorno en el cual todo fuesen objetos y todos ellos pudiesen interaccionar entre sí, como ocurre en el mundo real, llegando incluso a considerar a los propios usuarios del sistema como objetos. Para conseguir esto, basaron el desarrollo en una máquina virtual, no por sus características de multiplataforma sino por sus características de intérprete donde se pueden ir modificando elementos del sistema de forma continua, sin necesidad de una sofisticada compilación previa. Todos los objetos residen en la memoria de la máquina abstracta y son accesibles entre sí, de tal forma que la interacción entre aplicaciones es muy sencilla. Además, los usuarios de Smalltalk pueden utilizar una determinada operación de cualquier aplicación en cualquier momento. Algunas de estas operaciones tienen que ver con la construcción de los propios objetos y clases, de tal forma que se pueden ir

construyendo objetos desde cero y añadirle métodos, atributos propiedades como la herencia de forma interactiva. Otras, en cambio, permiten visualizar e inspeccionar la estructura de los objetos, su estado y comportamiento. Esto, posteriormente, desembocó en el concepto de reflectividad, estudiado más detalladamente en el Capítulo 5.

El hecho de que las aplicaciones puedan interactuar fácilmente y que se pueda acceder y modificar los distintos objetos existentes, aumenta la flexibilidad global del sistema pudiéndose representar sus funcionalidades mediante objetos y métodos modificables. Los sistemas desarrollados sobre una máquina abstracta orientada a objetos facilitan al usuario:

- Utilización del sistema con una abstracción más similar a la forma de pensar del ser humano [Booch94].

- Autodocumentación del sistema y de las aplicaciones. El acceso a cualquier objeto permite conocer la estructura y comportamiento de este en cualquier momento.

- Programación interactiva y continua. Una vez que el usuario entra en el sistema, accede al mundo interactivo del objeto y puede hacer uso de cualquier objeto existente. Si necesita desarrollar una nueva funcionalidad, va creando nuevos objetos, definiendo su estructura y comportamiento, comprobando su correcta funcionalidad y utilizando cualquier otro objeto existente de una forma totalmente interactiva. A partir de este momento el sistema posee una nueva funcionalidad y un mayor número de objetos.

4.3.7 Diseño de sistemas operativos

Las máquinas abstractas han sido también utilizadas en el diseño de sistemas operativos, principalmente cuando se trataba de construir sistemas multiplataforma y/o distribuidas. De esta forma, podían obtenerse ventajas directas de las características comentadas anteriormente, multiplataforma, código móvil e interoperabilidad distribuida. El único requisito necesario será implementar un intérprete de la máquina virtual sobre todas las plataformas en las que deseemos ejecutar nuestro sistema operativo. Los servicios propios del sistema operativo se van a construir sobre la máquina, no bajo ella, como suele ser habitual. De esta forma se consigue:

- Que una aplicación para este sistema operativo sea portable a cualquier plataforma.

- Que se pueda ampliar la máquina a través de los propios servicios ofrecidos por el sistema operativo. Con ello, se puede ver el sistema operativo como una extensión de la máquina virtual [Alv98].

- Portabilidad del propio sistema operativo, puesto que ha sido desarrollado sobre la máquina abstracta. No es necesario, pues, codificar cada servicio para cada plataforma.

- Que la interoperabilidad de las aplicaciones sea uniforme, al estar utilizando únicamente el modelo de computación de la máquina abstracta. En otros sistemas operativos es necesario especificar la interfaz exacta de acceso a sus servicios.

- Que se puedan construir sistemas con distribución implícita, donde se ve a un simple sistema operativo y un único sistema de objetos, pero soportado por varias máquinas con ubicación geográfica diferente [Alvarez00].

- Que la distribución explícita se simplifique, al no ser necesario establecer traducciones de datos. La interacción es directa, al ejecutarse todas las aplicaciones sobre la misma máquina abstracta, en distintas máquinas virtuales o intérpretes.

Existen distintos sistemas operativos desarrollados sobre una máquina abstracta ya sean comerciales, de investigación o didácticos como pueden ser Merlin, Inferno, Oviedo3, etc. por citar algunas.

4.3.8 Coexistencia de sistemas

La utilización de máquinas abstractas nos va a permitir un acceso uniforme a los recursos de una plataforma física, es decir, de una máquina real. De tal forma que definimos una interfaz de interacción con una plataforma física que puede ser dividida en un conjunto de máquinas virtuales.

Esto nos va a permitir la ejecución de distintos sistemas operativos denominados "huéspedes" inmersos dentro de otro denominado "anfitrión". Con ello podemos tener tantos sistemas operativos huéspedes dentro del anfitrión como deseemos y así poder ejecutar las aplicaciones desarrolladas para los sistemas huéspedes dentro del anfitrión con el cual son incompatibles.

Es más, si el sistema operativo anfitrión y el huésped están desarrollados para el mismo procesador computacional nos permitirá cambiar de un sistema a otro sin tener que reiniciar el sistema. Esto es lo que hacía OS/2 de IBM. Cuando dentro de él se ejecutaban aplicaciones desarrolladas para DOS o para Windows de Microsoft. Esto se podía hacer porque los tres sistemas operativos estaban desarrollados para el mismo procesador computacional, en este caso, la familia i86 de Intel.

El sistema operativo VM/ESA de IBM desarrollado para servidores IBM S/390 permitía ejecutar aplicaciones de otros sistemas operativos diferentes, incluso diseñados para procesadores distintos, a través de las apropiadas máquinas virtuales.

Otro producto que utiliza el concepto de máquina virtual para multiplexar el acceso a una plataforma física es VMware Virtual Platform [Jones99]. Ejecutándose tanto en Windows NT como en Linux permite lanzar aplicaciones codificadas para Windows 3.1, MD_DOS, WindowsNT, Linux, FreeDBS y Solaris 7 para Intel.

El principal inconveniente que tiene este tipo de sistemas es que las aplicaciones de los distintos huéspedes no pueden interactuar entre sí, ni siquiera las aplicaciones de un huésped con el sistema operativo anfitrión. Por lo que desde el punto de vista de la interoperabilidad, que es el tema de esta tesis, este uso de las máquinas abstractas carece de aporte para la misma.

4.4 Panorámica

A continuación, analizaremos algunas de las máquinas abstractas orientadas a objetos que hemos considerado más representativas. Desde el punto de vista comercial, Java o el CLR de .NET. Considerando su carácter histórico y pedagógico, Smalltalk. Por último, desde el punto de vista científico como son Oviedo3 y Nitro. Seguidamente, comentaremos las estructuras esenciales internas que las caracteriza para, en posteriores capítulos, definir lo que será nuestra máquina abstracta orientada a objetos básica, que pretende establecer un modelo simplificado y representativo, que utilizaremos como pilar fundamental en el desarrollo de nuestro modelo arquitectónico.

4.4.1 Smalltalk

Un sistema Smalltalk considera dos grandes componentes: la **imagen virtual** que contiene todos los objetos del sistema y la **máquina virtual** que son las rutinas del lenguaje máquina y de los dispositivos hardware que permite el movimiento de los objetos contenidos en la imagen virtual [Gold83].

Cuando se inicia un sistema Smalltalk, la máquina virtual carga todos los objetos descritos en la imagen virtual y comienza la ejecución del sistema interactivo comentado en la sección 4.3.6, formado por un "mundo" de objetos que interaccionan entre sí y con los usuarios.

En esta sección examinaremos la arquitectura de la máquina virtual de Smalltalk y para ello veremos, por un lado, cómo los métodos escritos por los programadores se convierten en una secuencia de instrucciones de ocho bits denominados **bytecodes,** t por el otro, cómo estos bytecodes son interpretados por la máquina virtual. Para finalizar veremos cómo son almacenados todos los objetos creados en el denominado **objeto memoria**.

4.4.1.1 El Bytecode

Todo código escrito en Smalltalk se traduce en secuencias de ocho bits denominadas bytecodes. Este código está basado en una máquina de pila convencional. Existen 256 bytecodes diferentes organizados en cinco categorías: apilar, desapilar, enviar, devolver y saltos.

Veamos un ejemplo de generación de bytecodes, para lo que utilizaremos el siguiente código Smalltalk que implementa el método `intersects` de una clase `Rectangle`:

```
Rectangle>>intersects: aRectangle
    ^(origin max: aRectangle origin) < (corner min: aRectangle corner)
```

El compilador traduce este código en bytecodes que se almacenan en un objeto de la clase `CompiledMethod` y la clase almacena un puntero a esta instancia en su diccionario de mensajes con el selector `intersects`. La instancia de CompiledMethod, además de contener el bytecode, contiene un marco de literales o **literal frame** que son los objetos a los que el bytecode no puede hacer referencia directamente ya que en tiempo de compilación son desconocidos. Estos suelen ser principalmente objetos globales y selectores de métodos de otras clases. Así los selectores de mensajes del ejemplo anterior, `max:`, `origin`, `min:` y `corner` no pueden ser referenciados directamente por el bytecode. Mostraremos el marco de literales después del bytecode.

Rectangle intersects:

0 Apila el valor de la variable receptora `origin` en la pila.
16 Apila el argumento `aRectangle`.
209 Envía un mensaje con el selector `origin`.
224 Envía un mensaje con el selector `max:`.
1 Apila el valor de la variable receptora `corner` en lapila.
16 Apila el argumento `aRectangle`.
211 Envía un mensaje con el selector `corner`.
226 Envía un mensaje con el selector `min:`.
178 Envía un mensaje binario con el selector `<`.
124 Devuelve el objeto de la cima de la pila como el valor del mensaje `intersects:`.

literal frame

```
#max:
#origin
#min:
#corner
```

Para indicar un selector se indica por la posición del mismo en el marco de literales. Por ejemplo el bytecode 211 indica que se envía un mensaje con el cuatro literal del literal frame como selector.

Veamos a continuación cómo son interpretados estos bytecodes por la máquina virtual.

4.4.1.2 El Intérprete

El intérprete de Smalltalk ejecuta las instrucciones bytecode que se encuentran en instancias de la clase `CompiledMethod`. Para ello necesita definir un contexto de ejecución a través de una instancia de la clase `MethodContext`. Este contexto consta de cinco elementos fundamentales:

1. Un puntero a la instancia `CompiledMethod` que contiene el bytecode del método que se está ejecutando.
2. Un **puntero de instrucciones** que identifica el próximo bytecode a ser ejecutado.
3. El receptor y los argumentos del mensaje que representa la instancia de `CompiledMethod`.
4. Las variables temporales utilizadas en `CompiledMethod`.
5. Una pila de ejecución.

Al ser un intérprete de pila, la mayoría de las instrucciones bytecode utilizan la pila. Así: los *bytecode apilar,* indican dónde se encuentran los objetos a apilar en la pila; los *bytecode desapilar*, indican dónde colocar los objetos que se encuentran en la pila; los *bytecode enviar*, sacan de la pila el receptor y los argumentos del mensaje a enviar y por último, los *bytecodes devolver*, apilan el resultado devuelto por un mensaje en la pila.

Una vez definido el contexto de ejecución, el intérprete realiza continuamente un ciclo de tres pasos:

1. Toma el bytecode apuntado por el puntero de instrucciones del `CompiledMethod` que se está ejecutando.
2. Incrementa el puntero de instrucciones.
3. Realiza la función indicada por el bytecode.

El intérprete de la máquina virtual de Smalltalk se caracteriza porque está implementada en el propio lenguaje Smalltalk. Veamos cómo quedaría la implementación del bucle principal del intérprete en dicho lenguaje.

```
VirtualMachine>>interpret
  [true] whileTrue: [self cycle]
```

Como vemos, la operación interpretar es un bucle sin fin donde se realiza la operación `cycle`.

```
VirtualMachine>>cycle
 self checkProcessSwitch.
 currentBytecode := self fetchByte.
 self dispatchOnThisBytecode
```

La operación cycle tiene tres pasos: uno para cambiar de proceso de ejecución, debido a que esta máquina virtual es multiproceso, una llamada a `fetchByte`, para obtener el siguiente bytecode, y, por último, una llamada a `dispatchOnThisBytecode` que interpreta el bytecode indicado.

```
VirtualMachine>>fetchByte
 | byte |
 byte := memory fetchByte: instructionPointer ofObject: method.
 instructionPointer := instructionPointer + 1.
 ^byte
```

La operación `fetchByte` simplemente recoge el byte code apuntado por `instructionPointer` del método `method` y, posteriormente, incrementa el puntero de instrucción.

4.4.1.3 Objeto memoria

Todos los objetos de la máquina están almacenados en un área denominada Heap (montón). Cada vez que se necesita un objeto se debe reservar espacio en el Heap para poder alojarlo. Como se muestra en la Ilustración 4.3, cada objeto está representado por un vector de palabras de 16 bits, donde el primero indica el tamaño del mismo, el segundo es una palabra que apunta a la clase del objeto y el resto son los campos del objeto.

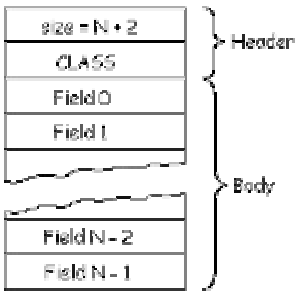


Ilustración 4.3: Objeto en el Heap

Cada objeto es accedido e identificado a través de un puntero al mismo. Este puntero se representa internamente a partir de valores de 16 bits que apuntan a **tabla de objetos,** donde se indica el comienzo dentro del heap del propio objeto. Con esta in dirección se consigue varias cosas. Primero, disminuir el tamaño de los punteros a solamente 16bits. Segundo, que si se cambia de ubicación de un objeto, el puntero a este no cambia, si no los valores de la tabla de objetos. También permite que el tamaño del heap sea mayor de 2^{16}. Por último, añade campos especiales de información al puntero a objeto, como contador de referencia al objeto, segmento y página, en caso de utilizarse memoria paginada, y otros campos especiales que se consideren.

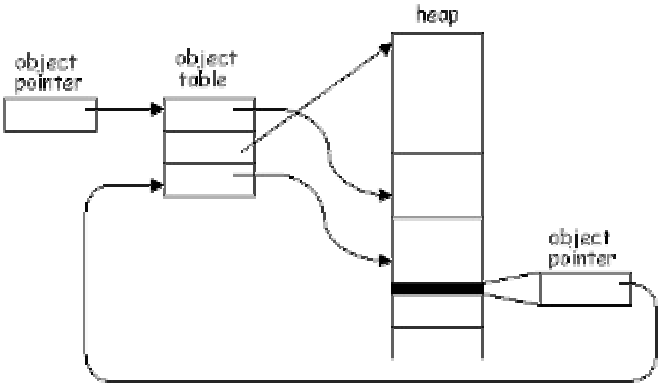


Ilustración 4.4: Tabla de Objetos

El puntero a objetos tiene un tratamiento especial debido a que, cuando su bit menos significativo vale 1, se interpreta no como un puntero sino como un objeto entero con signo primitivo. Esto se hace por motivos de eficiencia, debido a que este tipo de número se utiliza con mucha frecuencia. Por el contrario, cuando vale 0, se interpreta como un puntero normal y nos permitirá tener en el sistema 32k objetos. La tabla de objetos es un vector de 64k palabras de 16 bits, que están agrupadas en celdas de dos. El contenido de cada celda se muestra en la Ilustración 4.5, donde observamos que se

definen un campo contador de referencias y dos campos de direccionamiento paginado (segment, location)

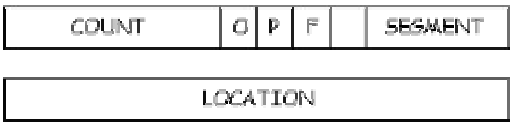


Ilustración 4.5: Celdas de la Tabla de Objetos

Para gestionar los punteros vacíos de la tabla se define una lista enlazada de celdas con el valor F a 1, como se muestra en la Ilustración 4.6

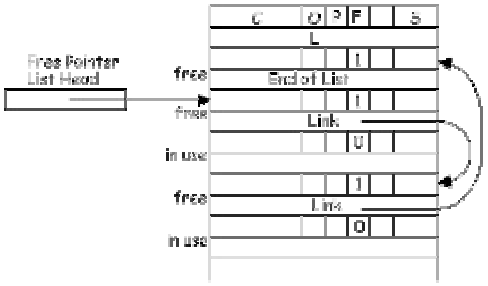


Ilustración 4.6: Lista de huecos en Tabla de Objetos

Todas los punteros a objetos del sistema se encuentran o en las variables definidas en los contextos que se están ejecutando o en las variables globales. Para estas últimas, se define una clase diccionario global especial denominada `Smalltalk` que las contiene todas. Las clases dentro de Smalltalk son a su vez objetos de tal forma que para localizar una clase dentro de Smalltalk hay que buscarla en el diccionario `Smalltalk`.

4.4.2 Java

Actualmente la máquina abstracta más popular y más ampliamente utilizada es la máquina virtual de Java (JVM) desarrollada por Sun Microsystems en 1995, está basada en el lenguaje Java y el código que genera es bytecode. Esta máquina está ampliamente extendida y prácticamente todos los navegadores contienen una, para poder visualizar los ya comentados (sección 4.3.4) Applets.

4.4.2.1 Arquitectura general

El entorno Java está formado por dos programas separados, por un lado tenemos javac que es el compilador de archivos con extensión .java donde se encuentra el código fuente de los programas de Java y que genera archivos de bytecode con extensión .class, por otro java.exe que es el interprete de bytecode generado [Lindh99].

En la Ilustración 4.7, se muestra la arquitectura general del intérprete, que está formado por áreas de datos en ejecución, un cargador de clases y un motor de ejecución.

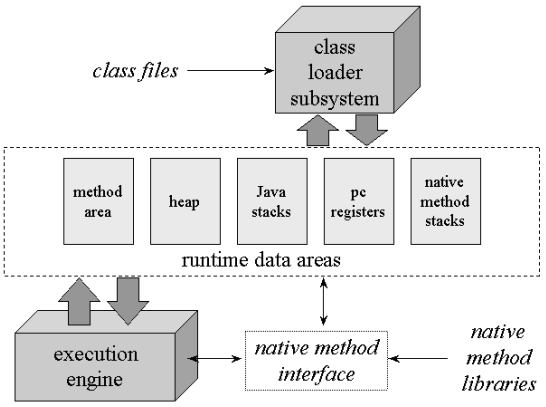


Ilustración 4.7: Máquina Virtual de Java

- El motor de ejecución se encarga de dar vida al bytecode que se encuentra en los métodos compilados de las clases.

- El cargador de clases se encarga de leer los archivos .class y construir dentro de las áreas de ejecución, la información necesaria para que el motor de ejecución de vida al programa construido.

- En el área de ejecución se encuentra toda la información del sistema y lo comentaremos detenidamente en la siguiente sección.

4.4.2.2 Runtime data areas

Está formada por [Ven98]:

- Method Área, que contiene la información acerca de las clases y los métodos que contiene.

- Heap, contienen la información referente a las instancias de las clases.

- Java Stack , es la pila del ejecución, los elementos que contiene se denominan frames que son el contexto de ejecución de los métodos que se están invocando. Cada thread contiene una pila de ejecución donde se van apilando frames, conforme se van realizando llamadas a métodos.

- PC registers, son los contadores de la instrucción bytecode actual a ejecutar en cada thread. Cada thread tienen asociado su propio PC register

- Native Stack, es la pila de ejecución de rutinas no implementadas en Java, rutinas auxiliares e incluso las propias rutinas del intérprete.

4.4.2.3 Instrucciones bytecode

Al igual que Smalltalk el código de las instrucciones de está máquina virtual está organizado en conjuntos de 8 bits. Se disponen de los siguientes grandes grupos de bytecode:

- Gestión de tipos primitivos, que permiten realizar carga y almacenamiento, operaciones aritméticas y conversiones entre tipos básicos.

- Control de flujo, disponemos de saltos condicionales, incondicionales, sincronización y condicionales estructurados.

- Gestión de la pila de ejecución, nos permiten apilar elementos, desapilar e intercambiar dos elementos de la cima de la pila.

- Gestión de objetos, nos permiten crear objetos, arrays de objetos e invocación de métodos, tanto métodos de clases como métodos de instancias.

4.4.3 .NET

Actualmente está emergiendo comercialmente una nueva plataforma de computación, impulsada por Microsoft, que pretende ser abierta, se denomina .NET [Meyer00]. El núcleo de esta plataforma está basado en una máquina abstracta descrita por la especificación **CLI** Infraestructura del Lenguaje Común (Common Language Intraestructure), que en el año 2002 se convirtió en el estándar ECM-335. De esta forma Microsoft permite que se puedan realizar intérpretes o implementaciones de esta máquina de forma abierta. Ya existen varias implementaciones de la CLI como el proyecto Mono para Linux realizado por Ximian [Ximi05] o el proyecto DotGNU Portable.NET [Boll05].

La máquina virtual de .NET está formada principalmente por dos grandes componentes:

- La librería de clases marco **FCL** (Framework Class Library), que proporciona la mayoría de los servcios ofrecidos por esta plataforma.

- El sistema de ejecución **CLR** (Common Language Runtime), implementa todos los requisitos necesarios para permitir la ejecución de las instrucciones de la máquina, denominadas en esta plataforma **IL** (Independnet Language) o lenguaje independiente. Este nombre viene dado por la independencia del lenguaje máquina de cualquier lenguaje de programación o procesador físico.

En la Ilustración 4.8, vemos como el CLR da vida a las aplicaciones formadas por instrucciones del lenguaje LI y llamadas a las API especificadas por CLI y FCL. Vemos también que, el CLR es el encargado de realizar las apropiadas llamadas al sistema operativo subyacente, así como al procesador físico de la plataforma.

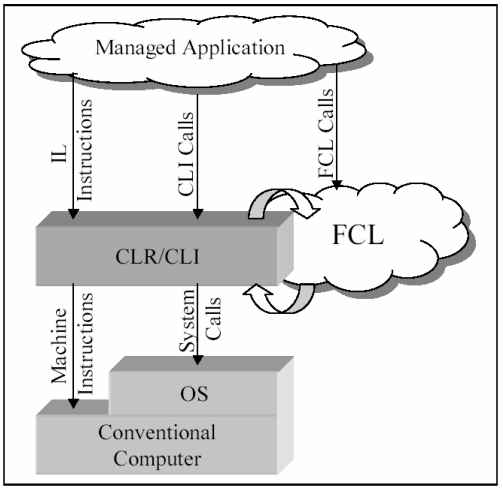


Ilustración 4.8: La máquina virtual de .NET

Realmente el CLR no es un intérprete convencional debido a que está basado en tecnología de compilación JIT. Esta tecnología de compilación comenzó a estudiarse en el proyecto de Sun Microsystems, Self [Smith95]. Consiste en ir compilando a código nativo el código de la máquina conforme se va necesitando, de tal forma que se consigue el máximo rendimiento.

El CLI ha sido diseñado para absorber todas las funcionalidades de un sistema operativo. De esta forma se pretende conseguir que las aplicaciones construidas para el CLI sean independientes del sistema operativo, del hardware y de cualquier lenguaje de programación. Para conseguir esta independencia se han definido especificaciones comunes para el lenguaje **CLS** (Common Language Specificaton), especificación común de tipos **CTS** (Common Type System) y por último un sistema de ejecución común el **VES** (Virtual Execution System) o también llamado motor de ejecución **EE** (Execution Engine).

En la Ilustración 4.9, se muestra la organización del motor de ejecución. Veamos como funciona: todo programa está compuestos por un conjunto de componentes denominados Ensamblados (**Assemblies**) el cargador de clases (**Class Loaer**) es el encargado de leer dichos componentes y construir su representación en memoria a partir de `Vtable` y `class info`, por último conforme se van necesitando, el compilador JIT se encarga de ir compilando el código que se pretende utilizar, generando el código nativo que ejecutará el procesador físico.

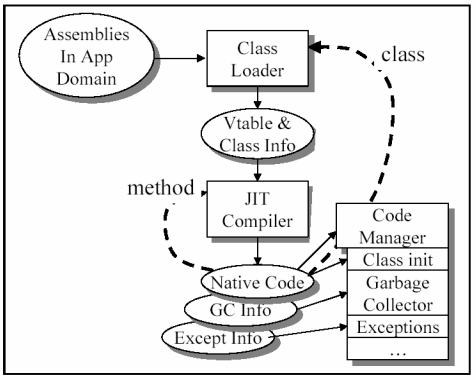


Ilustración 4.9: Organización del Motor de Ejecución

Dentro del motor de ejecución también tenemos recolector de basuras, gestor de excepciones, inicializador de clases, gestor de código, etc.

4.4.4 Oviedo3

A continuación revisaremos el trabajo realizado en el proyecto de investigación experimental denominado Oviedo3 que pretende construir un sistema integral orientado a objetos. Realizado por en la Universidad de Oviedo con la intención de construir un sistema operativo sobre él y evaluar los beneficios comentadas en la sección 4.3.7. En él se define una máquina abstracta que es la que utilizaremos y revisaremos más detenidamente en el Capítulo 17, para utilizarla en nuestro prototipo.

4.4.4.1 Sistema Integral Orientado a Objetos Oviedo3

El sistema integral orientado a objetos [Cueva96] esta formado por la unión de dos partes, por un lado una máquina abstracta orientada a objetos llamada Carbayonia y por otro un Sistema Operativo Orientado a Objetos denominado SO4 cumpliéndose la formula siguiente:

Oviedo3 = Carbayonia + SO4

Uno de los principales objetivos del proyecto es mantener una homogeneidad entre los objetos del sistema operativo, la máquina abstracta y los del propio usuario.

4.4.4.2 El Sistema Operativo SO4

Este Sistema Operativo, SO4 [Alvarez98a], esta totalmente orientado a objetos de tal forma que el usuario percibe el sistema como un entorno en el que puede crear todos los objetos que desee, estos interactúan entre sí para desarrollar las tareas de manera concurrente y controlada mediante los servicios del sistema.

Se pierden los conceptos de direcciones de memoria, ahora cada objeto guarda su propia información internamente, y de procesos, cada objeto esta en un estado activo cuando se invoca a alguno de sus métodos.

Los cuatro servicios básicos que ofrece SO4 son:

1. La seguridad, permite que únicamente los objetos autorizados puedan enviar mensajes a otros objetos.
2. La persistencia consigue que los objetos continúen en el sistema hasta que ya no se necesiten.
3. La distribución o subsistema AGRA, permite la invocación de objetos independiente de su localización
4. La concurrencia proporciona al modelo de objetos la capacidad de ejecutar tareas en paralelo.

Las líneas principales de diseño han sido:

1. Los objetos como la única abstracción. La única entidad que existe es el objeto de cualquier granularidad.

2. Modelo de objetos estándar con las características más comunes a los lenguajes de programación orientados a objetos: encapsulación, herencia y polimorfismo.
3. Un modo de trabajo exclusivamente OO. Un objeto puede solo crear clases heredadas de otras, crear un objeto de una clase y enviar mensajes a otros objetos.
4. Simplicidad. Mantenerse lo más fieles posible a las líneas anteriores, buscando la máxima sencillez.

Algunas características importantes que se desean obtener de los objetos en consonancia con las líneas anteriores son:

1. Objetos homogéneos. Todos los objetos incluidos los del SO son tratados de la misma forma.
2. Transparencia. Los objetos no deben preocuparse de la existencia de los mecanismos del SO para adquirir las características como persistencia, paso de mensajes, distribución, seguridad, etc.
3. Objetos autocontenidos. Toda la información de cada objeto se encapsula en su estado, incluyendo su computación.
4. Semántica completa. Todos los objetos tienen su semántica incrustada dentro del modelo de objetos de tal forma que no se consideran como solamente un espacio de memoria continuo.

Identidad de Objetos. Cada objeto tiene un único identificador, que se usa como una referencia y es la única forma de acceder a él

4.4.4.3 La Máquina Abstracta Carbayonia

Carbayonia [Alvarez97] es una máquina abstracta de alto nivel que proporciona el soporte de objetos básico para construir el sistema completo.

La máquina abstracta soporta el modelo de objetos con las siguientes características: identidad de objetos y abstracción, encapsulación, herencia, las relaciones de genericidad, agregación entre objetos y polimorfismo o paso de mensajes.

La principal característica de la máquina es que cada acción sobre un objeto, siempre que no sea un objeto primitivo, es realizada mediante una referencia al mismo: las instrucciones tienen referencias como operandos, los métodos de las clases tienen referencias como parámetros y el valor de retorno es una referencia.

La arquitectura está compuesta fundamentalmente por cuatro áreas. Un área se podría considerar como una zona o bloque de memoria de un microprocesador tradicional. Pero en esta arquitectura no se trabaja nunca con direcciones físicas de memoria, sino 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 muestra una breve descripción de las áreas que componen la máquina abstracta.

1. Área de clases. Guarda la descripción de cada clase.
2. Área de referencias. Almacena las referencias a los objetos. Cada una tiene un tipo (se relaciona con el área de clases) y apunta a un objeto de ese tipo (se relaciona con el área de instancias)

3. Área de instancias. Cuando se crea un objeto se almacena en esta área y cuando se destruye se elimina de aquí. Se relaciona con el área de clases de manera que cada objeto pueda acceder a la información de la clase a la que pertenece.
4. Área de referencias del sistema. Que tienen funciones específicas y existen de manera permanente en el sistema.

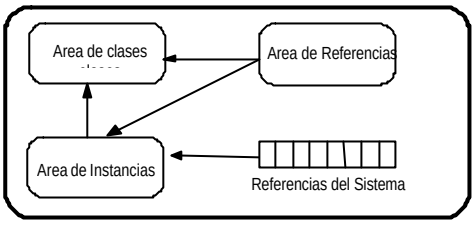


Ilustración 4.10: Arquitectura básica de Carbayonia

La máquina dispone de una arquitectura reflectiva que permite que los propios objetos constituyentes de la máquina puedan usarse dentro del sistema como cualquier otro objeto dentro del mismo.

El lenguaje de la máquina es un lenguaje de bajo nivel orientado a objetos puro. Permite declaración de clases, definición de métodos y gestión de excepciones. Algunas de las principales ventajas derivadas de usar una máquina abstracta para este SO son:

1. Portabilidad, basta con crear el emulador de la máquina para cada plataforma. Actualmente esta disponible para Windows y UNIX.
2. Se elimina el desajuste de impedancias. Hay un modelo común de objetos para cada objeto incluidos los del SO.
3. Independencia del lenguaje de programación.

4.4.5 Nitro

La máquina virtual Nitro [Ortin01] es una Máquina Abstracta Reflectiva no Restrictiva que incorpora tanto reflectividad estructural como de comportamiento a nivel de envío de mensajes y acceso a miembros. Es una máquina orientada a objetos prototipos al estilo de la máquina definida en Self [Ungar91], donde todo son objetos y las clases no existen, deben implementarse como objetos. Es la base de un Sistema Computacional de Programación Flexible desarrollado en la Universidad de Oviedo por el grupo de Tecnologías Orientadas a Objetos al cual pertenezco.

4.4.5.1 Diagrama de Subsistemas

Nitro está formado por tres grandes subsistemas, el intérprete, el procesador del lenguaje y el objeto memoria.

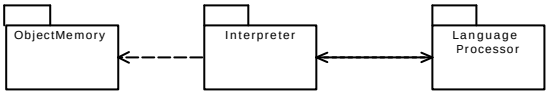


Ilustración 4.11: Subsistemas de Nitro

4.4.5.2 Subsistema LanguageProcessor

Esta máquina no está basada en instrucciones de código, si no que funciona a alto nivel, de tal forma que el subsistema LanguageProcessor va generando los objetos que va necesitando el intérprete conforme se va ejecutando el código de alto nivel en la máquina. Desde este punto de vista este subsistema, a través de su clase principal (Parser), es un generador de instrucciones (`Statement`) que nunca llegan a tener un bytecode determinado. Cada thread de ejecución de la máquina tiene asociado una instancia de Parser que va generando las statement que se están ejecutando en el.

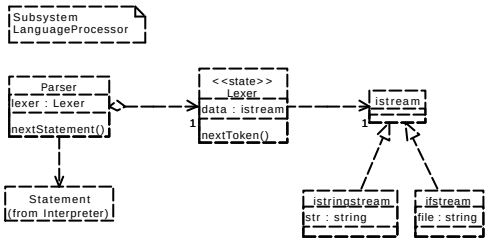


Ilustración 4.12: Nitro, LanguageProcessor

Finalmente comentar que las instrucciones se pueden sacar o bien de una cadena de caracteres o de un archivo. Este último caso, se utilizará cuando se pretenda inicializar el sistema a partir de un código de arranque guardado en un fichero.

4.4.5.3 Subsistema Interpreter

El intérprete de la máquina virtual de Nitro es implementado en este subsistema. Como se comentó anteriormente, las instrucciones no vienen representadas por códigos numéricos, bytecode, como ocurre en el caso de Java y Smalltalk, véase secciones 4.4.1 y 4.4.2, si no que son objeto directamente cuyas clases heredan de la clase abstracta `Statement`.

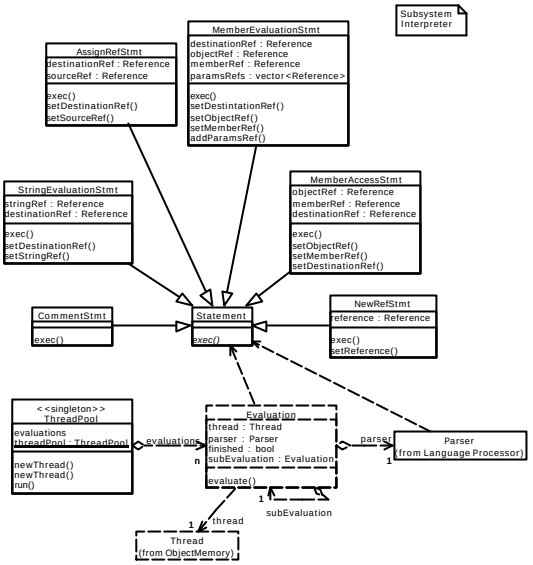


Ilustración 4.13: Nitro, Interpreter

Las instrucciones más importantes son:

- `MemberAccesStmt`, que permite el acceso a un miembro de un determinado objeto.

- `MemberEvaluationStmt`, que interpreta un miembro de un objeto como evaluable, un método, y lo ejecuta. Este método representaría la invocación dinámica.

Cada objeto `Evaluation` representa un hilo del sistema, mediante el atributo `thread`, que trabaja sobre un flujo de entrada a través del atributo `parser`. El sistema tiene en una instancia singleton de `ThreadPool` todos los hilos de ejecución en objetos `Evaluation` y va invocando a cada uno de sus métodos `evaluete` para llevar a cabo la interpretación del lenguaje de la máquina.

4.4.5.4 Subsistema Object Memory

Contiene la representación dinámica de sistema, está formado por la representación de los objetos que vienen dada por la clase `Object` y la representación de referencias que vienen dadas por la clase `Reference`. Por otro lado también define los thread con sus correspondientes contextos de ejecución.

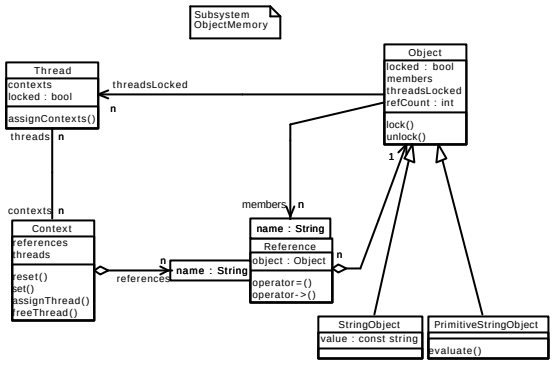


Ilustración 4.14: Nitro, ObjectMemory

Un objeto está formado por un conjunto de elementos miembro (`members`) que son referencias a otros objetos. Al final existen solamente dos tipos de objetos primitivos: el tipo `String` y los objetos primitivos evaluables `PrimitiveStringObject`. Existe una gestión inteligente de referencias a través de `refCount`.

El tipo String, merece una mención especial en esta máquina ya que tiene una serie de características que lo hacen digno de mención:

- Por un lado las cadenas pueden ser anidadas unas en otras, por ello se necesita una gramática de tipo 2 [Aho90] para poder manipularlas así como diferentes tokens para apertura y cierre < y > respectivamente. Así un ejemplo de cadena en esta máquina podría ser: <cadena principal <subcadena numero 1> <subcadena numero2> fin de la cadena principal>.

- Por otro lado y siguiendo con lo indicado en el punto anterior, el objetivo de esta estructuración en el tipo cadena es debido a que pueden representar código que puede posteriormente ser ejecutado, o siendo más precisos, evaluado.

En cuanto las clases `Thread` cada una de ellas representa un hilo de ejecución a partir de una pila de contextos, modelados por la clase `Context`.

4.5 Estructuras esenciales

Una vez estudiadas las máquinas abstractas anteriores, vemos que en esencia todas comparten una serie de características en común: un juego de instrucciones ya sea explícito como es el bytecode en el caso de SmallTalk o Java o implícito como en el caso de Nitro, un motor de ejecución que suele ser un intérprete, menos cuando se aplique la tecnología JIT en el caso de Java o .NET, un espacio de memoria donde guardar tanto las clases como los datos asociados a los objetos instanciados con esas clases, algunos como Nitro tienen un sofisticado sistema de interpretación de las instrucciones, mientras que otro definen un sencillo formato de instrucciones para realizar la interpretación.

De todas las características vistas, queremos resaltar las siguientes tres por la importancia que tiene en nuestro trabajo, estas son: la necesidad de un área de memoria donde guardar las clases y sus instancias, un juego de instrucciones de entre las que existen las instrucciones de invocación a métodos y técnicas de representación de la información ya sean internas o externas a través de mecanismos reflectivos.

En posteriores capítulos veremos la similitud que existen con los mecanismos de invocación dinámica en middleware.

4.6 Resumen

En esta sección hemos hecho un repaso general a los conceptos básicos de máquinas abstractas, posteriormente hemos visto qué beneficios nos aporta el uso de máquinas abstractas, a continuación hemos revisado cinco máquinas abstractas tanto comerciales como de investigación, y finalmente hemos analizados las características comunes que tienen entre sí, desde el punto de vista de los objetivos que queremos conseguir en nuestra tesis. En posteriores capítulos veremos que estas características son idóneas para construir sistemas que interaccionen con plataformas middleware RMI.

Capítulo 5:
Reflectividad

El objetivo de esta tesis es construir un modelo arquitectónico basado en máquinas abstractas reflectivas orientadas a objetos. En este capítulo repasaremos los conceptos relacionados con reflectividad, realizaremos una taxonomía de los diferentes clases de reflectividad que se pueden encontrar en sistemas orientados a objetos, para posteriormente, mostrar como nos puede ayudar este mecanismo en el desarrollo de sistemas interoperables entre middleware heterogéneos.

5.1 Conceptos

5.1.1 Reflectividad

La reflectividad apareció en primer lugar en el campo de la Inteligencia Artificial y posteriormente se propagó a varios campos de la informática, tales como, la programación lógica, la programación funcional o la programación orientada a objetos. En este último campo fue Pattie Maes [Maes87] quien la introdujo.

La reflectividad es la capacidad de un sistema para observar su computación y posiblemente cambiar la forma en que esta es llevada a cabo. La observación y la modificación implican algo subyacente que será observado y modificado. En este caso, como el sistema razona sobre sí mismo, lo subyacente es él mismo, por lo que el sistema debe disponer de una autorepresentación [Cazzola98].

Formalmente, la reflectividad fue introducida en la computación por Brian Smith [Brian82] que la definió como "la capacidad de una entidad de representar, operar o tratar de cualquier modo consigo misma de la misma forma que representa, opera y/o

trata con entidades del domino del problema". Dicho de otra manera, la reflexión es la capacidad de un sistema de manipular, de la misma forma que manipula los elementos que representan el problema del mundo real, elementos que representan el estado del sistema durante su propia ejecución.

Es Bobrow en [Bobrow91] quien considera dos aspectos de la reflectividad, la observación y la modificación. Literalmente dice:

"Reflectividad es la capacidad de un programa de manipular como datos algo que representa el estado del programa durante su propia ejecución. Existen dos aspectos de dicha manipulación: **introspección** e **intervención**. La introspección es la capacidad de un programa de observar y razonar acerca de su propio estado, sin posibilidad de manipularlo. La intervención es la capacidad de un programa de modificar su propio estado de ejecución o alterar su propia interpretación. Ambos aspectos requieren un mecanismo que codifique el estado de ejecución como datos. La **exposición**, del inglés reification, proporciona tal codificación".

Un sistema reflectivo está organizado estructuralmente en dos o más niveles, formando, lo que se denomina una **torre reflectiva**. El primer nivel se denomina **sistema base** y el segundo **metasistema**.

Todos estos conceptos se exponen detalladamente en las siguientes secciones.

5.1.2 Sistema Base y MetaSistema

Conceptualmente, un sistema de computación puede considerarse compuesto por dos niveles: el nivel base o sistema base y el metanivel o metasistema.

El nivel base o sistema base, es el nivel donde se describe el programa y se soluciona el problema externo, describiendo la computación desde el punto de vista de la aplicación. Es decir, la computación que el sistema lleva a cabo como entidades que reflejan el mundo real.

El metanivel o metasistema, es el nivel donde se interpreta el programa descrito en el nivel base, manteniendo información y describiendo cómo se lleva a cabo la computación en el nivel previo. Tienen como tarea hacer funcionar al sistema base y retornar información acerca del propio sistema y su computación [Maes87].

Puede considerarse que el nivel base tiene una determinada definición que es interpretada por el metanivel, que puede verse como un intérprete para el nivel base. El ejemplo más claro es, un sistema formado por un programa en un lenguaje de programación que es interpretado por un metasistema, el intérprete del lenguaje de programación.

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 metanivel 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 metasistema que da soporte a los objetos que forman el sistema base. Visto de otra forma, la máquina es un intérprete de los programas escritos en el lenguaje máquina (juego de instrucciones).

Las entidades que trabajan en el nivel base se denominan entidades base, mientras que las entidades que trabajan en otros niveles, metaniveles, se llaman metaentidades.

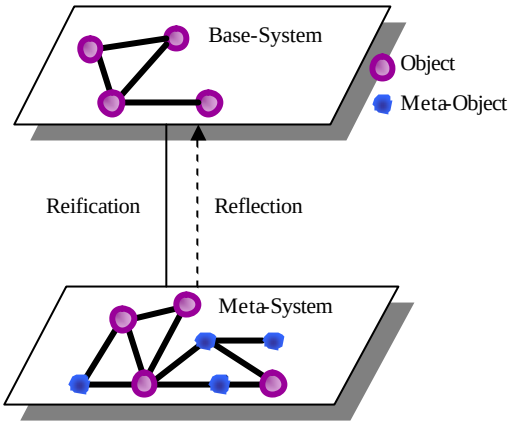


Ilustración 5.1: Sistema Base y Meta Sistema

5.1.3 Torre reflectiva

Si se generaliza un poco, se puede pensar, que hay un sistema bajo el metasistema que le da vida como él hace con el sistema base. Este sería el metametasistema, que a su vez tendría otro metasistema que lo interpreta. Continuando con esta lógica construiríamos lo que se denomina una torre de intérpretes, donde el último metasistema sería interpretado por un procesador físico. Normalmente la torre de intérpretes suele estar formado por dos sistemas, el sistema base y el metasistema. No obstante puede aparecer otro metasistema adicional, en sistemas donde se pretende maximizar la intervención en los sistemas superiores.

5.1.4 Exposición y Reflejo

En los sistemas reflectivos, se identifican dos actividades básicas cuando se lleva a cabo manipulación de los elementos del metasistema: exposición, (del inglés, reification) y reflejo, (del inglés, reflection) [Maes87]

5.1.4.1 Exposición

Los dos aspectos básicos de la reflectividad, introspección e intervención [Bobrow93], requieren un mecanismo que codifique el estado de ejecución de ciertos aspectos del sistema como datos accesibles, creando un modelo del sistema, a la vez que ofrecen mecanismos adecuados para la manipulación de tales aspectos de sí mismo.

La exposición es la capacidad de proporcionar tal representación, transformando y haciendo accesibles a los elementos del nivel base, aquellos aspectos del metasistema que, normalmente, le son intangibles [Sour98].

Como nota adicional, supóngase un sistema orientado a objetos (LPOO, MAOO) que exponga la estructura interna de los objetos, es decir, que codifica, mediante objetos de primera clase, la estructura interna de los objetos. Por ejemplo, la API reflectiva de Java define nuevas clases (field, method, constructor, etc.), para representar los distintos componentes que forman la representación en tiempo de ejecución de un objeto. Los objetos instancias de estas clases representarán los métodos, campos, etc. del objeto en tiempo de ejecución.

5.1.4.2 Reflejo

El reflejo es el resultado de modificar algunos aspectos del sistema por el sistema en sí, gracias a la actuación sobre el modelo ofrecido por el metasistema. De esta forma, cualquier cambio en el modelo en los aspectos expuestos se refleja en el comportamiento del sistema [Cazzola98].

De tal forma que, si es posible modificar el objeto u objetos que describen la estructura interna de los objetos del sistema, se modificaría la representación de los mismos en tiempo de ejecución.

Por ejemplo en Smalltalk es posible añadir un método a una clase en tiempo de ejecución, el reflejo de esa acción tendrá como resultado una clase con un método más en el sistema base. De igual forma se pueden modificar métodos ya existentes con lo que el comportamiento del sistema base cambiará igualmente.

De tal forma que existe una correspondencia entre los aspectos reflejados y su representación, de manera que cualquier modificación en la representación modifica el aspecto representado y viceversa [Sour98].

5.1.5 Arquitectura reflectiva

Un sistema con arquitectura reflexiva, reconoce la reflexión como un concepto fundamental, de tal forma que proporciona herramientas para manipular explícitamente la computación reflexiva. Esto significa que:

- El sistema debe permitir el acceso a la información que representa aspectos de él mismo e incluso permitir manipularla.

- El sistema debe garantizar la conexión causal entre la información manipulada y el aspecto del sistema.

Las arquitecturas reflectivas proporcionan un paradigma nuevo fundamental para razonar sobre los sistemas computacionales. En una arquitectura reflectiva, un sistema computacional se considera formado por dos partes computacionales, una parte de objetos y otra parte reflectiva. La tarea de los objetos computacionales es resolver problemas y devolver información acerca de un domino de problema externo, mientras que la tarea de la parte reflectiva es resolver problemas y devolver información acerca de los propios objetos computacionales.

5.1.6 Metamodelo

Uno de los problemas que se le suele achacar a la reflectividad es la posibilidad de manipulación directa y sin control de la estructura interna del sistema reflejado, lo que derivaría rápidamente en el caos.

Sin embargo, los mecanismos de reflectividad no permiten la manipulación directa del sistema de computación, sino que se ofrece un modelo o representación restringida del sistema (modelo del metasistema o metamodelo) que si está permitido cambiar y manipular [Ferber89a].

Este metamodelo que se muestra al sistema base es en el que se ve reflejado el metasistema. Es decir, cuando el sistema base "mira" al metasistema que le da vida para conocer información acerca de sí mismo, ve su "reflejo" (la representación que el metasistema usa para mostrarse al sistema base). De ahí el nombre de reflectividad para esta propiedad.

5.1.6.1 Requisitos del modelo

El modelo del sistema debe satisfacer varios requisitos para exponer completamente los aspectos del sistema elegidos y permitir la modificación de los mismos, de forma efectiva y segura [Maes87]: un nivel de abstracción adecuado, sobre el que sea fácil razonar y manipular, manteniendo la relación de causalidad del modelo con el sistema.

- Abstracción adecuada del sistema. El nivel de abstracción que el modelo aplica sobre el sistema debe ser adecuado con relación al uso que se pretenda hacer de la reflectividad. Para llevar a cabo la reflectividad en un sistema es necesario definir qué aspectos se quieren reflejar en el modelo, es decir, qué entidades y/o características del metasistema deben exponerse. Estas entidades o aspectos del sistema de computación en sí, serán aquellos sobre los que actúa la parte reflectiva de la arquitectura, para lo que deben representarse o transformarse en algo que pueda ser manipulado.

- Adecuado para el razonamiento y la manipulación. El modelo debe representar al sistema de forma que sea sencillo razonar sobre él y su estado, así como la manipulación del mismo. De esta forma, el sistema base debe poder manipular el modelo del metasistema (metainteracción).

- Relación de causalidad. Conexión causa-efecto entre el modelo o representación del metasistema y el metasistema en sí. El modelo debe estar conectado con el sistema mediante una relación de causalidad, de tal forma que bs cambios efectuados sobre el modelo tengan su reflejo en el comportamiento posterior del sistema.

5.1.7 Metaespacio

El metaespacio de un objeto en el sistema base está formado por el conjunto de objetos del metanivel que dan soporte a este. Así como veremos en la sección 12.3, para

los objetos de nuestra máquina abstracta básica su metaespacio está formado por las clases `MAClass` y `MAInstance`, que son las necesarias para darle vida en el sistema base.

5.1.8 Metacircularidad

Este concepto suele estar asociado a los intérpretes metacirculares, normalmente lo llevan los lenguajes que tienen algún mecanismo de representar el dato código, y suelen ir asociados a algún tipo de acción de evaluación o interpretación de esos datos. Por ejemplo: en LISP tenemos la función `eval` sobre cadena de caracteres, en Smalltalk se aplica el método `value` sobre los objetos de tipo bloque o Nitro con la evaluación de métodos o cadenas.

Virtualmente, el intérprete de estos lenguajes consiste en una torre infinita de intérpretes que interpretan al intérprete circula inferior. Por ejemplo: imaginar que la cadena enviada a `eval` en LISP tiene a su vez una llamada a una función `eval` y así sucesivamente todas las veces que se desee. En tal caso se estará construyendo una torre virtual de intérpretes que en realidad es el mismo intérprete.

5.1.9 Teoría de la relatividad de la reflexión

El modo de representar la propia información reflejada, el metamodelo, determina qué tipo de computación puede realizar el sistema con respecto a sí mismo. Dicho con otras palabras la computación que se puede realizar sobre el metasistema depende del las características del metamodelo que lo representa. De esta forma surge el término **reflectividad completa**, cuando se pueden modificar completamente todo el metasistema [Maes87].

5.2 Taxonomías

Esta taxonomía está ideada para sistemas orientados a objetos, donde la reflectividad encaja especialmente bien, debido a la **uniformidad** que se puede mantener, al estar, tanto los elementos del sistema base como los del metasistema, representados como objetos. Ha sido desarrollada principalmente a partir de dos trabajos previos, uno realizado por Walter Cazzola [Cazzola98] y otro realizado por Francisco Ortín [Ortin01].

5.2.1 Qué se refleja

En [Cazzola98], se define varios tipos de sistemas reflectivos en función de qué parte de un sistema orientado a objetos se expone. Considerando las clases, los objetos, los mensajes o el canal de comunicación. Esta clasificación viene de una clasificación previa [Ferber89] donde se consideran solo dos aproximaciones: metaobjetos y exposición de la comunicación.

5.2.1.1 Modelo metaclase

Según la definición de [Graube89], una clase describe tanto la estructura como el comportamiento de sus instancias. La estructura es el conjunto de variables cuyo valor estará asociado a las instancias. El comportamiento es el conjunto de métodos a los que las instancias de la clase serán capaces de responder.

El modelo de metaclase [Cointe87, Briot89] se basa en la equivalencia entre la clase de un objeto y su metaobjeto y, por tanto, implementa la torre reflectiva siguiendo la cadena de instanciaciones.

Las clases serán objetos, instancias de una clase, su metaclase. A su vez, las metaclases forman una jerarquía que deriva de la clase raíz, CLASS.

De esto se deriva que una clase es un metaobjeto que expone una entidad base. De hecho, en los sistemas basados en el paradigma de clase/instancia, la clase de un objeto es considerada normalmente como su metaobjeto, debido a la reflectividad estructural del modelo.

De esta forma, para un objeto O es equivalente, desde el punto de vista del comportamiento, recibir un mensaje M o que su clase reciba el mensaje indicado para gestionar el mensaje, que, en la mayoría de las obras se representa con HANDLEMSG.

El método HANDLEMSG se define en la metaclase y el método de gestión de mensajes por omisión se describe en la metaclase raíz de la jerarquía de metaclases (CLASS), de la que son instancias las clases.

Cuando un objeto recibe un mensaje, debe decidir si utilizar el mecanismo por omisión o pasar el control al metanivel. Esto puede lograrse mediante una entrada definida en el receptor, lo que permite lograr la reflectividad para un objeto, o en su clase, lo que establece un modo reflectivo para todas las instancias.

Este modelo tiene la ventaja de la sencillez con que la clase encaja con el papel de controlador y modificador de la información estructural, porque mantiene esa información. La implementación de este modelo es posible, únicamente en aquellos lenguajes que manejen las clases como objetos, como por ejemplo Smalltalk [Foote89] o CLOS [Bobrow93].

Se han identificado algunos inconvenientes a este modelo: la dificultad de especializar el comportamiento de una instancia concreta, de modificar dinámicamente la metaclase de un objeto por otra o registrar información acerca del objeto en su meta representación. Además, la interacción de las metaclases y la herencia da lugar a la aparición de problemas de compatibilidad entre distintas metaclases.

El problema radica en la especialización del metacomportamiento de una instancia concreta. Todas las instancias de una clase tienen el mismo metaobjeto, por tanto, todas tienen el mismo metacomportamiento.

Para especializar el metacomportamiento para una instancia, es posible usar la herencia construyendo una clase nueva que difiera de la clase original sólo en el metacomportamiento, o bien mantener diccionarios que mantengan información y

métodos de cada instancia. Sin embargo, esto añade complejidad al sistema, quitando parte de la sencillez inicial.

Otro problema radica en la dificultad de *cambiar dinámicamente* el comportamiento de un objeto. Tal como implicaría sustituir su metaclase por otra.

Sin embargo, no todos los lenguajes permiten el cambio dinámico de la clase de un objeto. Es más, el cambio de la clase de un objeto puede derivar en inconsistencias.

Dado que todos los objetos de una clase comparten la misma metaclase, se ven obligados a almacenar la misma metainformación. Esto hace imposible registrar características particulares de cada objeto en su metarepresentación. Esta limitación es crucial dado que uno de los aspectos más importantes de las metaentidades es su capacidad para almacenar información acerca de su objeto base o referente.

Los lenguajes basados en clases organizan las aplicaciones en jerarquías de clases utilizando para ello la herencia, lo que además facilita la reutilización.

Aquellos lenguajes que utilizan el concepto de clase para la refectividad tienen que tener en cuenta, además del enlace clase, subclase, superclase, el enlace de instancia existente entre una clase y su metaclase.

La interacción entre la herencia y la instanciación debe tenerse muy en cuenta a la hora de diseñar librerías de clases. El problema radica en determinar si una clase puede heredar de otra o no, partiendo del hecho de que ambas clases son instancias de metaclases diferentes. Esto da lugar a la aparición del problema de la compatibilidad de metaclases.

5.2.1.2 Modelo metaobjeto

En el modelo de MetaObjetos [Kickzales91], los metaobjetos son instancias de una clase especial, la clase MetaObjeto o de una de sus subclases. Los metaobjetos no se identifican con la clase del objeto sino que se ligan arbitrariamente con los objetos a los que se refieren y únicamente incluyen información de control del comportamiento, dejando la información estructural a la clase. De esta forma, la torre reflectiva se lleva a cabo siguiendo la relación cliente/servidor.

De esta forma cuando un objeto O recibe un mensaje, es equivalente a enviar el mensaje HANDLEMSG a su metaobjeto.

Entidades independientes gestionan tanto la introspección como la intervención de cada entidad base. Cada metaobjeto intercepta los mensajes enviados a su objeto y ejecuta su metacomputación antes de reenviarlo a su objeto base.

En este modelo no existen, en principio, restricciones ni imposiciones sobre la relación entre las entidades base y las metaentidades. Cada metaobjeto puede estar conectado a varios objetos del nivel base y cada objeto puede tener enlazados varios metaobjetos durante su ciclo de vida, cada uno de los cuales se refiere a una parte de su comportamiento. Al conjunto de metaobjetos que componen el entorno de ejecución de un objeto se le denomina metaespacio.

Esta aproximación tiene la ventaja de eliminar, prácticamente, todos los inconvenientes del modelo anterior, permitiendo especializar el metacomportamiento, monitorizar el objeto y modificar el metaobjeto.

Es sencillo especializar el metacomportamiento de un objeto individual desarrollando una nueva clase de metaobjetos que refinen el comportamiento original de alguna forma.

También es sencillo hacer que un metaobjeto monitorice el comportamiento de su objeto, registrando qué sucede y, eventualmente, decidiendo algún cambio.

Partiendo de que los metaobjetos son objetos como cualquier otro, que se crean y destruyen de la misma forma que otros objetos, con un identificador único y conocido y sabiendo que se enlazan con su objeto base a través de alguna asociación con este identificador único, es posible modificar fácilmente el metaobjeto de un objeto asignando a tal asociación el identificador de otro metaobjeto con un comportamiento diferente.

El mayor inconveniente reseñable de este modelo es que un metaobjeto puede monitorizar un mensaje sólo cuando ya ha sido recibido por su objeto referente. Por tanto no se puede actuar sobre él durante la transmisión. Este es el modelo más usado no sólo en lenguajes de programación (3-KRS) sino también en sistemas operativos ApertOS.

5.2.1.3 Modelo metamensaje

Esta última aproximación, consiste en la exposición de la comunicación en sí. Cada comunicación puede ser un objeto y, por tanto, atiende o reacciona al mensaje SEND.

En este modelo, las metaentidades son objetos especiales, denominados mensajes, que exponen las acciones que deberían ejecutar las entidades base [Feber89].

La clase MESSAGE contiene toda la información necesaria para interpretar el mensaje. Cuando se invoca un método, se crea una instancia de la clase MESSAGE y se envía el mensaje SEND a tal instancia. El objeto mensaje está dotado con su propia gestión de acuerdo al tipo de metacomputación requerida. Cuando la metacomputación termina, se destruye el objeto.

La clase de un mensaje define el metacomportamiento realizado por el mensaje de forma que diferentes mensajes pueden tener diferentes clases. Es posible definir diferentes comportamientos para llamadas a métodos para cada o objeto, especificando un modo diferente para cada llamada a método.

La primera ventaja destacable es la facilidad que ofrece para diferencias entre distintos tipos de mensajes: se pueden definir y usar distintos subclases de comunicación en lugar de la clase MESSAGE estándar.

Por otro lado, el uso de diferentes tipos de mensajes permite una extensión incremental del sistema. Es posible incorporar fácilmente nociones como paso concurrente de mensajes.

La principal desventaja de este modelo, es que, los metaobjetos mensaje no están enlazados de ninguna manera con las entidades base que los originan y, por tanto, no pueden acceder a su información estructural.

Además, los metaobjetos mensaje tienen una vida muy corta, existen sólo durante el tiempo que dura la acción que representa.

Debido a todo esto, son inadecuados para monitorizar el comportamiento de los objetos del nivel base o almacenar información sobre la metacomputación (carencia de continuidad), para representar información específica acerca de los objetos o representar el comportamiento de un objeto determinado.

Sin embargo, es una aproximación hábil para combinarse con los modelos anteriores.

5.2.2 Cuanto se refleja

La metaarquitectura de un sistema orientado a objetos está formada por varios elementos tanto conceptuales como estructurales, en función de cual de estos se exponga, resultará alguno de los siguientes tipos reflectivos.

5.2.2.1 Reflectividad Estructural

La reflectividad estructural se puede definir como la capacidad de exponer y manipular tanto el programa que está siendo ejecutado en este momento como sus datos [Cazzola98]. En caso de que se tratase de una arquitectura reflectiva OO, esta exposición se llevaría a cabo utilizando los conceptos de OO, como objeto o clases.

Este modelo tiene en cuenta la extensión de la parte estática de un sistema, es decir, permite exponer y manipular los aspectos estructurales del sistema de computación, como la herencia en un sistema OO, considerándolos como datos, y ofrece mecanismos que permitan la manipulación de dicha representación.

Un ejemplo de reflectividad estructural lo constituye el operador sizeof, que permite averiguar el espacio ocupado por un tipo de dato en tiempo de ejecución, o la API reflectiva de Java que permite obtener, los métodos y campos que componen un objeto en tiempo de ejecución, modificar el contenido de un campo, invocar un método en tiempo de ejecución, etc.

Los sistemas OO parecen ser uno de los campos en donde la reflectividad puede ser más fácilmente implantada, dada la naturaleza inherente de tales sistemas a distribuir en objetos los datos y procedimientos que representan el sistema.

La reflectividad estructural en sistema OO, introducida por P. Cointe [Cointe87], se refiere al uso reflectivo de clases y metaclases para implementar objetos.

En un modelo de clases puro, cada entidad es una instancia de una clase y las clases, a su vez, son instancias de otras clases, llamadas metaclases. Este modelo tiene en cuenta la extensión de la parte estática de los lenguajes OO, es decir, de los aspectos estructurales de los objetos considerados como la implementación de un tipo abstracto de datos.

La reflectividad estructural está muy extendida tanto en los lenguajes funcionales, como Lisp, como en los lenguajes lógicos, como Prolog, que tienen sentencias para manipular la representación del programa. Por ejemplo, Prolog representa los programas como estructuras y es posible acceder al functor, a sus argumentos, etc. Esta reflectividad se fundamenta en el hecho de que ambos son lenguajes interpretados.

En el caso de los lenguajes orientados a objetos, la mayoría son compilados y, por tanto, no hay representación de los mismos en tiempo de ejecución. Una excepción es Smalltalk, que utiliza las clases para llevar a cabo la reflectividad estructural [Foote89].

Esta representación es adecuada, cuando lo que pretendemos es dotar al sistema de mecanismos como la introspección y se aplica fundamentalmente a elementos estructurales del sistema como la herencia.

Como ejemplos conocidos y significativos de reflectividad estructural podemos citar la API reflectiva de Java, el sistema de información de tipos en tiempo de ejecución o RTTI de C++ o SOM.

5.2.2.2 Introspección

Hay sistema que sólo permite la consultar la información del metasistema, sin poder modificarlo o cambiarlo. En estos casos se dice que el sistema dispone de la capacidad reflectiva de introspección. Se puede ver la introspección como una reflectividad estructural restringida solamente a la observación. Es el caso del modelo reflectivo de Java, .NET, CORBA y DCOM por citar algunos. Normalmente esto suele ser debido a que la información del metasistema se consulta a partir de documentos generados previamente en tiempo de compilación a partir de archivos IDL. Se está investigando para aportar reflectividad estructural a estos sistemas. Del tal modo que, OpenCORBA es una versión de CORBA que permite reflectividad estructural o MetaJava también intenta aportar este tipo de reflectividad, para .NET se está investigando actualmente en un proyecto de Microsoft que se está desarrollando en la Universidad de Oviedo.

5.2.2.3 Reflectividad Lingüística

La reflectividad Lingüística nos permite modificar cualquier aspecto del lenguaje de programación. De esta forma que, mediante el lenguaje del sistema base se podrá modificar la estructura del propio lenguaje, por ejemplo: añadir operadores, construcciones sintácticas o nuevas instrucciones.

OpenJava [Chiba98] aporta un sistema reflectivo lingüístico, en tiempo de compilación, para adaptar el lenguaje Java a patrones de diseño como Adaptador (Adapter) o Visitante (Visitor) [Ganma00].

5.2.2.4 Reflectividad del comportamiento

La reflectividad del comportamiento, introducida por P.Maes [Maes87] y B. Smith [Smith82], se refiere al comportamiento del sistema computacional y se puede definir como la capacidad de un lenguaje, en general cualquier sistema, para

proporcionar una exposición completa de su semántica y de los datos que utiliza para ejecutar el programa actual.

La reflectividad del comportamiento manipula el comportamiento del sistema de computación en tiempo de ejecución e implica dotar a un programa P de la capacidad y los mecanismos necesarios para observar y modificar las estructuras de datos utilizadas para su propia ejecución. Es decir, la reflectividad del comportamiento supone la modificación de alguna característica del propio sistema subyacente.

Un posible ejemplo es el ORB reflectivo OpenCorba [Ledoux99] que expone, entre otras características internas del bus, la invocación remota. OpenCorba permite la modificación del mecanismo de paso de mensajes para implantar distintas políticas de envío, lo que, lógicamente afectará a los futuros envíos de mensajes a objetos. Cuando se envían un mensaje a un objeto representante o proxy para su reenvío al objeto remoto al que representa, el mensaje se intercepta a su llegada al objeto representante y se reenvía al servidor remoto. Sin embargo, el control del envío al objeto remoto se realiza en el metanivel, donde se pueden implantar políticas que modelen Java RMI, futuras versiones de CORBA DII o invocaciones locales.

La flexibilidad permitida por esta reflectividad puede llegar a ser excesiva, lo que hace necesario un control sobre los cambios que permite la reflectividad en el sistema. En concreto, las modificaciones permitidas por la reflectividad pueden llevar a inconsistencias si las actualizaciones hechas por el código reflectivo, el que modifica el entorno en tiempo de ejecución, involucran aspectos del sistema que se consideran implícitos. Por ejemplo si el sistema implanta herencia, ésta debe tener siempre el mismo significado.

5.2.2.5 Reflectividad Completa

Cuando la capacidad reflectiva de un sistema es total, es decir que desde el sistema base se tiene acceso a cualquier elemento del metasistema, se tiene un sistema reflectivo completo. Para ello es necesario que el metamodelo ofrezca una representación integra del metasistema. Normalmente esto ocurre cuando el metamodelo ofrece todas las capacidades reflectivas comentadas en las secciones anteriores: estructural, de comportamiento y lingüística.

Este tipo de sistemas resulta de muy difícil construcción y la mayoría de los sistemas reales, limitan a priori el conjunto de características accesibles desde el sistema base, adaptándolas al problema concreto que se desea resolver.

5.2.3 Cuándo se refleja

En la fase de desarrollo de un programa hay varias etapas en las que se encuentra disponible cierta información del meta sistema. Normalmente se suele diferenciar entre tiempo de compilación y tiempo de ejecución. Veamos a continuación ambos tipos.

5.2.3.1 Reflectividad en Tiempo de Compilación

La información proporcionada del metasistema al sistema base viene dada en el momento en el que el código fuente está siendo traducido a código máquina.

Normalmente este tipo de reflectividad es introspectivo, si bien pude ser modificarse algún tipo de información que sea generada por el compilador.

Este tipo de compilación es el más antiguo y el más utilizado, ya lo incorporaba el lenguaje C en su función `sizeof()`, que devuelve el tamaño de un determinado tipo. Si bien ha llegado a sofisticarse hasta los niveles que llega OpenJava, que incorpora una técnica de macros que permite añadir reflectividad lingüística a Java para incorporar patrones de diseño, como se indicó en la sección 5.2.2.3.

5.2.3.2 Reflectividad en Tiempo de Ejecución

En este caso la causalidad del mecanismo reflectivo ocurre cuando el sistema está siendo ejecutado por algún procesador. En este caso obtenemos una mayor versatilidad y flexibilidad. Pudiéndose llegar a conseguir una reflectividad completa, de tal forma que el sistema puede adaptarse a un determinado problema reaccionando a un suceso ocurrido en determinadas circunstancias. La mayoría de sistemas reflectivos son de este tipo. Si bien puede haber dos tipos de procesadores que ejecuten el sistema, los físicos o los lógicos, en función de estos podremos clasificar la reflectividad como: interpretada o nativa.

5.2.3.2.1 Ejecución Interpretada

La mayoría de los sistemas reflectivos entran dentro de este tipo, son sistemas en los que le ejecución viene dada por un procesador lógico. El intérprete guarda información en tiempo de ejecución de todos los aspectos del metasistema con lo que no resulta excesivamente difícil exponerlos, el principal inconveniente es que el rendimiento, ya que los sistemas interpretados son más lentos. Todos los sistemas basados en intérpretes incluyen este tipo de reflectividad, como son: Self, Smalltalk, Java o Nitro.

5.2.3.2.2 Ejecución Nativa

En este caso la reflectividad debe conseguirse a partir del código nativo que ejecuta un determinado procesador físico. En este caso resulta más difícil y menos versátil construir los metamodelos. El mecanismo RTTI del ANSI/ISO de C++ ofrece introspección en tiempo de ejecución para un sistema nativo. El proyecto Iguana pretende construir un compilador de C++ con reflectividad computacional en tiempo de ejecución ampliando así a RTTI [Gowing96].

El principal inconveniente de estos sistemas es su complejidad frente a su gran ventaja que es la gran eficiencia propia del código nativo.

5.2.4 Cómo se refleja

Los sistemas pueden ofreces los mecanismos reflectivos de dos formas diferentes, una implícita, es decir se encuentra de forma inherente al propio sistema, sin necesidad de solicitarla, siempre está disponible y la otra es solicitándola explícitamente al modelo reflectivo.

5.2.4.1 Reflectividad implícita

El sistema activa sistemática y automáticamente la computación reflectiva. Esto significa que existen agujeros en el proceso normal de interpretación que la computación reflectiva llenará, o lo que es lo mismo, si el proceso de interpretación consiste en la ejecución de una serie de tareas a_1, a_2,…,a_n, existen uno o mas i para los que la acción a_i no está definida por la máquina subyacente. Es decir algún a_i es, necesariamente, un trozo de computación reflectiva definida por el programa que está siendo ejecutado (o interpretado). Un ejemplo lo tenemos en LIPS cuando la tarea a_i es una operación `eval` sobre alguna variable de código que no es conocida por el intérprete. Otro claro ejemplo es el mecanismo reflectivo lingüístico de C++ con la sobrecarga de operadores.

Un sistema tiene reflectividad implícita cuando siempre está disponible y forma parte del sistema como un servicio más. Por ejemplo, la operación de StringEvaluationStmt en la máquina Nitro, ver sección 4.4.5, utiliza reflectividad implícita ya que la ejecución de esa instrucción depende de lo que interprete la máquina en función de lo que haya en la string. La instrucción MemberEvaluationStmt también, ya que Nitro no tiene un tipo método.

5.2.4.2 Reflectividad explícita

La reflectividad explícita no ocurre automáticamente sino que tiene lugar únicamente cuando la aplicación lo solicita. De esta forma, la computación tiene lugar, normalmente, en el nivel de objetos. Siempre y cuando el programa solicite la computación reflectiva, el sistema, temporalmente, deja este nivel y pasa a realizar alguna tarea en el nivel reflectivo.

Un ejemplo es cuando el programa, de manera explícita solicita el tamaño (parte de la representación interna) de un tipo de datos del lenguaje de programación C mediante el operador sizeof.

Es decir un sistema dispone de reflectividad explicita cuando claramente se le puede pedir información al sistema sobre el metasistema la estructura de un elemento. En la máquina Nitro nuevamente encontramos este tipo de reflectividad, los operadores que utilizan reflectividad estructural explícitamente como el *hash* o el *fistkey*, etc. En estos casos el metasistema realiza la operación internamente y devuelve los resultados. En este caso la semántica de cada una de las operaciones está clara y no depende del contenido de ninguna “string” como ocurría con la reflectividad implícita.

5.2.5 Cómo se accede al modelo

La siguiente clasificación está basada en la forma de expresar la manipulación de los elementos del metamodelo desde el sistema base. Esta puede ser procedural y declarativa.

5.2.5.1 Reflectividad Procedural

En el caso de los sistemas metacirculares, disponemos de un sistema representado en términos de un programa que implementa el sistema, diremos que disponemos de reflectividad procedural.

Esto nos lleva a que el metamodelo expresado a través de elementos de un programa, necesita definir una serie de operaciones sobre las que poder realizar invocaciones y manipularlo. A esto se denomina metainterfaz. Por otro lado, estas operaciones deben ser invocadas siguiendo una serie de criterios, un determinado orden o restringido por un conjunto de restricciones que junto con la interfaz determinan un protocolo a seguir para utilizar el metamodelo o un MOP (Meta Object Protocol).

5.2.5.1.1 Metainterfaces y MOP

El diseño de la implementación abierta propuesto por Kiczales [Kiczales92] razona que, mientras que el modelo tradicional de ocultación de información protege a los clientes de una abstracción de tener que conocer los detalles de cómo está implementada esta, también evita que pueda alterar algunos detalles cuando lo necesite.

Por lo que, se manifiesta como conveniente que los clientes de una abstracción puedan tener algún control sobre decisiones tomadas en la implementación de la misma.

Sin embargo, esta posibilidad de adaptación del entorno de ejecución no debe suponer quebraderos de cabeza para los usuarios. En este punto, no está de más recordar que la razón original de intentar esconder los detalles de implementación era una muy buena razón: simplificar la tarea del programador abstrayendo aquellos detalles con los que se suponía que tendría que trabajar [Wirth84].

Para permitir la adaptación sin incrementar la complejidad, se propone separar la interfaz que presenta un servicio o abstracción en interfaz base o protocolo base, que ofrece a los usuarios la funcionalidad usual de la abstracción o servicio, y metainterfaz o metaprotocolo, que permite a los clientes controlar cómo se implementan ciertos aspectos de la abstracción o servicio [Kiczales92].

De esta forma, en lugar de exponer todos los detalles de la implementación, la metainterfaz debería proporcionar una visión abstracta de la misma que permitiese a los clientes ajustarla en modos bien definidos.

El objetivo final de separar el comportamiento y la implementación es permitir al programador cliente:

- ser capaz de controlar la implementación del servicio cuando lo necesite.
- ser capaz de tratar el comportamiento y la implementación por separado.

Ofreciendo interfaz base y metainterfaz por separado, se ofrece al usuario programador una visión separada de ambos mundos.

5.2.5.1.1.1 Interfaz base

La interfaz de la aplicación se modela como una caja negra tradicional que expone la funcionalidad y no los detalles de implementación.

Por ejemplo, en el caso de la memoria virtual, la interfaz base únicamente lee y escribe bytes, en el caso de un sistema de ficheros abra/cierra ficheros, lee/escribe bytes, etc. Los clientes del servicio escriben los programas sobre la interfaz base de modo tradicional.

5.2.5.1.1.2 Metainterfaz

La metainterfaz articula y expone los detalles de implementación posibilitando a los clientes controlar las decisiones a tomar a la hora de establecer la correspondencia entre un servicio y su implementación final, permitiendo así conseguir una personalización de grano fino.

Por ejemplo, en el caso de una metainterfaz al sistema de memoria virtual, el cliente escribiría código utilizando la interfaz base (leer y escribir bytes de o en memoria) y, en caso de que el sistema de paginación por omisión ofreciese un rendimiento pobre para el comportamiento del cliente, este podría escribir código usando la metainterfaz para indicar un algoritmo alternativo de paginación para mejorar el comportamiento del código base.

5.2.5.1.1.3 Protocolo de MetaObjeto (MOP)

En principio, un MOP, es la metainterfaz de implementación de un modelo de objetos. Dicho de otro modo, especifica la implementación de un modelo de objetos reflectivo.

Si se considera que el metanivel es una implementación del modelo de objetos soportado por el lenguaje, *un MOP especifica qué objetos del metanivel son necesarios para implantar el modelo de objetos. Y la interfaz exportada por cada uno de esos objetos forma parte de la interfaz del MOP* [Gowing96].

En un sistema orientado a objetos donde, tanto las aplicaciones como el propio sistema, se modelan y representan en tiempo de ejecución como objetos que interaccionan entre sí, el metanivel especifica la implementación del modelo de objetos soportado por el sistema en términos de los metaobjetos necesarios para lograr el comportamiento deseado del sistema.

Por ejemplo, sería necesarios metaobjetos para representar las clases, los métodos, los atributos de los objetos, con métodos que permitiesen consultar y modificar la signatura de un método, los atributos que componen el estado interno de un objeto, eliminando o insertando alguno nuevo. También podría ser necesarios objetos que representasen la ejecución de métodos en los objetos permitiendo consultar y modificar el estado de cada una e incluso eliminar alguna.

El metaprotocolo o metainterfaz se refiere, no a la especificación de la interfaz del objeto, sino a la especificación de la implementación de los objetos.

5.2.5.2 Reflectividad Declarativa

Diversos autores han tratado de describir la autorepresentación del sistema sin exponer características de la implementación del propio sistema. Se dice que este tipo de sistemas soportan reflectividad declarativa. Para conseguir esto, la representación del sistema consiste en una serie de sentencias declarativas acerca del sistema. De esta forma la manipulación del metasistema a través del metamodelo consiste en una declaración de una colección de restricciones que el estado y comportamiento del sistema debe cumplir. Un ejemplo de restricción sería: “Las clases que heredan de `stream` deben crear un objeto `log` al finalizar la ejecución del método write”. La relación causal en este tipo de sistemas es más difícil de conseguir ya que vamos a necesitar

algún elemento inteligente que una vez indicadas las restricciones sepa como llevarlas a cabo y así poder mantener la conexión causal.

5.2.6 Quién modifica el sistema

Una vez definido una arquitectura reflectiva es necesario determinar quien va a poder acceder al metamodelo definido obteniendo así nuevas características para el sistema.

5.2.6.1 Reflectividad con Acceso Interno

Es la característica que ofrecen la mayoría de sistemas reflectivos, permitiendo a las aplicaciones modificarse desde sí mismas. Es normal que una aplicación reflectiva durante su ejecución pueda inspeccionarse, modificarse o añadirse nuevos elementos. Si bien los sistemas que permiten el acceso externo también suelen permitir este.

5.2.6.2 Reflectividad con Acceso Externo

En este caso permiten acceder y modificar elementos a otros objetos diferentes a los de la aplicación a la que pertenecen. Normalmente los que incluyen este tipo de acceso incluyen el acceso interno también. Este tipo de reflectividad suele ser peligrosa ya que se corre el peligro de que elementos externos maliciosos corrompan elementos del sistema, incluso elementos críticos que podrían desestabilizarlo y destruirlo. Es por esto que se suelen tomar dos criterios: permitir el acceso y manipulación libre o restringirlo con algún mecanismo de seguridad.

5.2.6.2.1 Acceso Libre

No se restringe el abanico de posibilidades en el acceso al metasistema. En este caso se otorga al programador de un sistema reflectivo la posibilidad de poder modificar cualquier elemento de este. Como se mencionó anteriormente esto puede implicar serios problemas de seguridad o como mínimo desembocar en una semántica del sistema prácticamente incomprensible. Un ejemplo de esto, es un programa escrito en C++ con uso intensivo su mecanismo de Reflectividad Lingüística que es la sobrecarga de operadores, que puede llevar a un código prácticamente ininteligible.

5.2.6.2.2 Acceso Restringido

Para evitar estos problemas de seguridad es necesario, por un lado definir un MOP lo más seguro posible, donde no se permitan accesos a partes críticas del metasistema y por otro lado se necesita establecer un mecanismos de seguridad con restricción a usuarios y a objetos del sistema, por ejemplo una basado en capacidades [Diaz00]. De esta forma se puede obtener un sistema reflectivo seguro y estable.

5.3 Resumen

La reflectividad es la capacidad de un programa de manipular, como si de datos se tratase, la representación del propio estado del programa durante su ejecución. En esta manipulación existen dos aspectos fundamentales: introspección e intervención

Un sistema reflectivo está compuesto por dos o más niveles que constituyen una torre reflectiva. Cada nivel sirve como nivel base para su nivel superior y como metanivel para su nivel inferior.

La reflectividad encaja especialmente bien con los principios promovidos por las tecnologías orientadas a objetos. Para introducir la reflectividad de manera uniforme en un sistema orientado a objetos es interesante que el modelo del metasistema esté descrito con el mismo paradigma orientado a objetos que el propio sistema base. Un objeto del nivel base es un objeto definido por la aplicación y que refiere al ámbito del problema, mientras que un objeto del metanivel es un objeto que da soporte a los objetos del nivel base para lo que representan información acerca de los mismos y determinan su comportamiento.

Existen multitud de formas de caracterizar un sistema orientado a objetos reflectivo dependiendo de qué parte del metasistema se modela: clase, objetos, mensajes; cuanto se refleja: introspección si sólo deseamos consultar, estructural si además deseamos modificar, lingüística si deseamos modificar el lenguaje, computacional si deseamos modificar la computación subyacente y completa si se ofrece una representación íntegra del metasistema; cuando se refleja: si se realiza en tiempo de ejecución o en tiempo de compilación; cómo: si los elementos del modelo hay que crearlos explícitamente o nos vienen dados implícitamente con el sistema base; cómo se accede: de forma procedural o declarativa; y finalmente quién lo modifica: acceso interno, externo libre o externo restringido.

Con esta revisión y clasificación de los conceptos reflectivos se ha pretendido ilustrar la utilidad que nos aportan al permitir construir sistemas más flexibles, adaptables, estructurables e inteligentes.

En la siguiente sección daremos una revisión a los sistemas distribuidos, para de ahí seleccionar las características que se desean obtener de ellos y el tipo de sistemas distribuidos que más se adapta a las características ofrecidas por las máquinas abstractas, para así poder crear un sistema que aprovechando el homomorfismo entre ambas, permita la interoperabilidad transparente bidireccional, para de esta forma, a partir de ahí, obtener una comunicación virtual entre plataformas distribuidas heterogéneas.

Capítulo 6:
Sistemas distribuidos

En este capítulo se pretende definir el contexto en el que se enmarca esta tesis, por un lado definiremos lo que son los sistemas distribuidos, identificaremos las características que poseen estos e identificaremos las problemáticas asociadas a los mismos y en concreto la problemática de la heterogeneidad entre middleware de invocación dinámica de métodos, finalmente en marcaremos los sistemas middleware entre la gran tipología de sistemas distribuidos existentes.

6.1 Definiciones

Un sistema distribuido es aquel en el que los componentes localizados en computadores, conectados en red, comunican y coordinan sus acciones únicamente mediante el paso de mensajes. [Gelemter85]. Una definición más formal sería:

"Un sistema distribuido es aquel en el que los componentes hardware o software, localizados en computadores unidos mediante red, comunican y coordinan sus acciones sólo mediante paso de mensajes."

Otra definición, dada por Tanenbaum en [Tane02] es:

"Un sistema distribuido es una colección de ordenadores independientes que el usuario ve como un sistema único coherente"

Esta definición tienen las siguientes consecuencias significativas: concurrencia, inexistencia de reloj global e independiente a fallos.

Compartir recursos es uno de los motivos principales para construir sistemas distribuidos.

Los desafíos que surgen en la construcción de sistemas distribuidos son la heterogeneidad de sus componentes, su carácter abierto, que permite que se puedan añadir o reemplazar componentes, la seguridad y la escalabilidad, que es la capacidad para funcionar bien cuando se incrementa el número de usuario, el tratamiento de los fallos, la concurrencia de sus componentes y la transparencia

6.2 Características

6.2.1 Compartición de recursos

El objetivo básico y fundamental de los sistemas distribuidos es permitir compartir recursos entre los equipos del sistema. Antes de nada debemos aclarar que un *recurso* caracteriza el rango de cosas que pueden ser compartidas de forma útil en un sistema de computadoras conectadas en red. Este se extiende desde los componentes hardware, como son, impresoras, discos, fax, etc., a componentes software, como son, aplicaciones, bases de datos, objetos. Esto nos permite, ahorrar en recursos, al utilizar por ejemplo una impresora entre varios equipos, aumentar la capacidad del sistema, al compartir varios discos, el rendimiento, al compartir un procesador o por último potencia la comunicación colaborativa, al compartir el escritorio del sistema.

Para permitir la compartición de recursos necesitaremos una serie de prestaciones como son: el nombrado de recursos que nos permitirá poder utilizarlo, una correspondencia entre el nombre del recurso y su dirección de comunicación, y una coordinación del acceso concurrente a recursos.

6.2.2 Apertura

Un sistema distribuido abierto es un sistema que ofrece servicios de acuerdo a reglas estándar que describe la sintaxis y semántica de estos [Alv98]. Con esto obtenemos sistemas concurrentes, independientemente extensibles y que permite a componentes heterogéneos conectarse o desconectarse del sistema. Esto implica que los sistemas abiertos son de naturaleza evolutiva y que la vida de sus partes es más corta que la del propio sistema. Esta característica que a primera vista es una ventaja, conlleva una serie de inconvenientes, como son: la difícil gestión del mismo y de sus partes, la falta de visión global, la dificultad para mantener la seguridad y confidencialidad de los mensajes intercambiados, la heterogeneidad de los sistemas interconectados. Esto ha dado lugar a nuevos modelos de programación que tratan de solucionar y adaptarse a estas problemáticas. Se ha visto que el paradigma de Orientación a Objetos no es capaz de abarcar o minimizar estas, por lo que ha evolucionada a un nuevo paradigma, el de Orientación a Componentes que trata de adecuarse a los entornos abiertos y solucionar su problemática asociada. Debemos observar que un sistema abierto no tiene por que ser distribuido, mientras que un sistema distribuido, por naturaleza es abierto. Esto nos lleva a que en la programación orientada a componentes estos pueden ser o no distribuidos.

Interoperabilidad caracteriza el alcance por el cual dos implementaciones de sistemas o componentes de diferentes fabricantes pueden coexistir y trabajar juntos

simplemente dependiendo en los servicios de cada uno como ha sido especificado en un estándar común [Alv98].

La interoperabilidad va más allá de la mera adecuación de estándares e interfaces. Por Interoperabilidad entendemos la capacidad que tienen dos o mas entidades para comunicarse y cooperar de forma compatible entre sí: Existen varios niveles de interoperabilidad: Sintáctico, protocolario y semántico [Valle99].

6.2.3 Portabilidad

La portabilidad caracteriza hasta que punto una aplicación desarrollada para un sistema distribuido A puede ser ejecutada, sin modificación en un sistema distribuido diferente B que implementa los mismos interfaces que A.

6.2.4 Flexibilidad

Nos permitirá una fácil modificación del sistema de tal forma que sea sencillo cambiar o sustituirlas debido a errores o a ampliaciones. Por otro lado, debe ser fácil introducir nuevas funcionalidades, incluso diferentes implementaciones simultáneas para el mismo servicio.

6.2.5 Concurrencia

Los sistemas distribuidos deben gestionar la concurrencia de tal forma que al acceder varios equipos diferentes a un mismo recurso no de lugar a errores o datos ambiguos. Hay dos tipos de concurrencia: la real, debida a que existen múltiples computadores en la red y la aparente que enlaza la ejecución de programas en el mismo computador. Una solución sencilla consistiría en serializar todos los accesos a todos los recursos, pero esto penalizaría el ritmo de producción del sistema. Por lo que se deben de utilizar mecanismos de sincronización en los recursos que se consideren críticos.

6.2.6 Rendimiento

El rendimiento de un sistema distribuido aumenta conforme se interconectan equipos al mismo, aunque este aumento viene penalizado y determinado por muchos factores como son la velocidad de comunicación o el grado de paralelismo. Un paralelismo de grano fino aumentará menos el rendimiento que un paralelismo de grano grueso, el tamaño y frecuencia de los mensajes intercambiados. Aunque la ejecución en un sistema distribuido de un programa es más rápida que en equipo único, si es verdad que dicho aumento no es lineal al número de computadores que forman el sistema.

6.2.7 Escalabilidad

Se dice que un sistema es escalable si conserva y aumenta su efectividad cuando ocurre un incremento significativo en el número de recursos y número de usuarios que pertenecen a él. Un sistema escalable debe permitir:

- Controlar el coste de los recursos físicos, cuando un sistema crece debido a demanda en algún tipo de recurso.

- Control de la pérdida de prestaciones

- Prevención del desbordamiento de recursos software

- Evitar cuellos de botella de prestaciones

6.2.8 Fiabilidad y tolerancia a fallos

Los sistemas distribuidos son más fiables que los centralizados. Se dice que los fallos en los sistemas distribuidos son parciales, es decir, algunos componentes fallan mientras otros siguen funcionando. A continuación enumeramos algunas técnicas para el tratamiento de fallos:

- Detección de fallos

- Enmascaramiento de fallos

- Tolerancia a fallos

- Recuperación frente a fallos

- Redundancia

6.2.9 Disponibilidad

Disponibilidad, mide la proporción de tiempo en que un sistema está utilizable. Esta propiedad tiene que ver con la tolerancia a fallos, a menor número de fallos, mayor disponibilidad. Hay que maximizar el grado de disponibilidad de todos los elementos que forman parte de un sistema distribuido.

6.2.10 Transparencia

La transparencia será la ocultación al usuario y programadores de las separaciones de los componentes o recursos utilizados en un sistema distribuido, de forma que perciba el sistema como una unidad en vez de verla como una colección de recursos. En otras palabras se pretende ocultar la naturaleza distribuida del sistema.

En diversas bibliografías ANSA Reference Manual [ANS89] y el International Standars Organization's Reference Model for Open Distribued Processing (RM-ODP) [ISO92] se realizan las siguientes clasificaciones de transparencia:

6.2.10.1 Transparencia de acceso

Nos permite acceder a recursos locales y remotos de idéntica forma. El típico ejemplo son los sistemas de archivos distribuidos que nos permiten acceder de forma idéntica, a través de las instrucciones read y write, al contenido de los mismos, ya sean estos locales o remotos.

6.2.10.2 Transparencia de ubicación/localización

Nos permite acceder a los recursos sin tener conocimiento de su ubicación física. Las bien conocidas URL nos permiten conseguir este tipo de transparencia.

6.2.10.3 Transparencia de concurrencia

Dada la naturaleza concurrente de los sistemas distribuidos, esta característica nos permitirá acceder a varios usuarios a la vez a recursos compartidos, sin que estos sean conscientes de ello. Para llevar esto a cabo se puede utilizar el mecanismo de transacciones, aunque estos son difíciles de implementar y gestionar.

6.2.10.4 Transparencia de replicación

Oculta el hecho de que hay varias copias de un mismo recurso dentro del sistema, por motivos de eficiencia y seguridad. Para obtener transparencia de replicación también deberemos tener transparencia de localización ya que el usuario no debe saber tampoco la ubicación de cada una de las copias.

6.2.10.5 Transparencia frente a fallos

Una informal definición de sistemas distribuidos, realizada por Leslie Lamport:

> "Sabes cuando estas trabajando en un sistema distribuido, cuando el bloqueo de un ordenador del cual nunca has oído hablar te interrumpe el trabajo que estas realizando"

Esta definición nos lleva a la necesidad de gestionar los fallos e intentar que estos permanezcan lo más ocultos posibles a los usuarios, cosa que no siempre es posible.

6.2.10.6 Transparencia de movilidad

Permite ocultar la que los programas o usuarios cambian su ubicación, un típico ejemplo de este tipo de transparencia la obtenemos con el sistema de telefonía móvil, donde los usuarios cambian continuamente su ubicación.

6.2.10.7 Transparencia de rendimiento

Permitirá al sistema reconfigurarse o ser reconfigurado, para mejorar las prestaciones según varía la carga.

6.2.10.8 Transparencia al escalado

Permitirá al sistema expandirse en tamaño de forma transparente al usuario, esto suele llevar implícito un aumento del rendimiento del sistema o de su capacidad.

6.2.10.9 Transparencia de persistencia

Oculta si el contenido de un recurso se encuentra en memoria (volátil) o en disco (no volátil) de tal forma que siempre tengamos disponible el recurso y este mantenga su

estado. Esta característica se pretende tanto en sistemas distribuidos, como en sistemas no distribuidos.

6.2.10.10 Transparencia de migración

La transparencia de localización nos permite identificar recursos sin importarnos su ubicación física, un claro ejemplo son las URL en la Web, pero si las páginas migran a otro servidor, con diferente nombre, este hecho no se ocultará al usuario. Con transparencia de migración, esto se oculta al usuario.

Existe un compromiso entre el rendimiento del sistema y el grado de transparencia que adquiere este. Por ejemplo, dos procesos deben saber que están en diferentes redes (eliminan la transparencia de ubicación) si deseamos realizar llamadas muy rápidas de uno a otro.

6.2.11 Heterogeneidad

La heterogeneidad, que quiere decir variedad y diversidad, de un sistema distribuido, se aplica a los siguientes elementos del mismo: Hardware, Redes, Sistemas operativos, Lenguajes de programación y Programas de diferentes proveedores.

Para conseguir esta heterogeneidad, es necesario cumplir alguno de los siguientes requisitos: transformar los formatos o representaciones de datos o protocolos de un sistema a otro o cumplir con formatos o protocolos estandarizados que permitan la comunicación. Así por ejemplo, Internet permite heterogeneidad a nivel de sistemas operativos, redes y hardware debido a que está basado en un protocolo de comunicación estandarizado y ampliamente aceptado como es TCP/IP. De tal forma que todo dispositivo conectado a Internet, tiene en común que se comunica utilizando el protocolo TCP/IP. Sobre este protocolo hay un conjunto de servicios soportados por otros protocolos que permiten realizar un gran número de tareas como son HTTP para la Web, FTP para transferencia de archivos, TELNET para terminales remotos, H323 para videoconferencias, POP3, MAPI y SMTP para correo electrónico, etc. Estos protocolos están ampliamente aceptados y estandarizados. Todos estos protocolos han sido ideados para una comunicación humano/computador, computador/humano, si un programa necesita comunicarse con otro debe utilizar una capa, protocolo intermedio proporcionado por lo que se denominan **middleware**. Un middleware es una abstracción de comunicación a nivel de lenguaje de programación de tal forma que un programa puede comunicarse con otro, realizando llamadas a procedimientos, funciones, métodos de otro a través de él, independientemente del sistema operativo, del lenguaje de programación destino, del hardware y de los protocolos de red subyacentes. Existen varias plataformas middleware que pretenden solucionar el problema de la abstracción de la programación en los sistemas abiertos distribuidos, como son: RPC, CORBA, DCOM, RMI, SOAP. A diferencia de Internet, que se ha estandarizado su protocolo de comunicación TCP/IP, a nivel middleware no hay aun un consenso y nos ha surgido una torre de babel a nivel middleware, es decir hay muchos sistemas implementados en diferentes plataformas que no pueden interoperar entre sí. Realmente existen un conjunto de especificaciones y sistemas desarrollados en el mercado denominados puentes que convierten a los sistemas abiertos distribuidos con middleware en un conjunto de islas conectadas con estos puentes. El objetivo de esta tesis es solucionar este problema construyendo un único sistema intermedio que permite

la interoperabilidad a nivel de middlewares heterogéneos. Si bien es cierto que posteriormente este sistema puede permitir la comunicación con otros muchos sistemas de comunicación, incluso con sistemas heredados, con lo que puede resultar una agradable arma para la gran lucha de la integración de sistemas, como veremos a lo largo de la misma.

Código móvil es el código que puede ser enviado desde un computador a otro y ejecutarse en éste. Se puede conseguir heterogeneidad de código móvil con la utilización e maquinas virtuales, como la de Java, la de Oviedo3 o la de .NET

6.2.12 Extensibilidad

La extensibilidad se determina en un sistema distribuido y por tanto abierto, en el grado en el cual se pueden añadir nuevos servicios de compartición de recursos y ponerlos a disposición para el uso de una variedad de programas clientes. La extensibilidad es inherente de los sistemas abiertos, aunque además es necesario disponer de la especificación y la documentación del las interfaces software públicas que pueden ser utilizadas por el nuevo componente que extiende el sistema. Esto nos da lugar a los lenguajes de definición de interfaces y al concepto de reflectividad estructural o introspección, véase sección 5.2.1.

6.3 Tipología

Cumpliendo la definición y características anteriores, existe una gran tipología de sistemas distribuidos como son los: sistemas de comunicación entre procesos, archivos distribuidos, sistemas de seguridad distribuida, servicios de nombres, transacciones, replicación, sistemas multimedia, sistemas de computación masiva, sistemas de coordinación y acuerdo, sistemas de memoria compartida distribuida, arquitecturas de integración de aplicaciones, etc. A continuación se expondrá en qué consiste cada una de ellas, para en el siguiente capítulo centrarnos dentro de las comunicaciones entre procesos, en las plataformas RMI que analizaremos más detenidamente.

6.3.1 Comunicación entre procesos

Para poder construir sistemas distribuidos, necesitamos por un lado computadores, con todo su software y hardware asociado, redes de computadoras para interconectarlos y técnicas de abstracciones de comunicación entre ellos de más alto nivel que los utilizados en las comunicación a nivel de transporte. Con motivo de esta necesidad de abstracciones de comunicación, ha surgido una nueva capa software que se encuentra entre medias de las aplicaciones distribuidas y los servicios de red, los sistemas operativos y el hardware de comunicaciones. Esta nueva capa se denomina *middleware* y pretende resolver el problema de la comunicación entre procesos de forma independiente del lenguaje y de cualquier plataforma hardware o software subyacente. En los últimos años una gran variedad de sistemas middleware han surgido, pero los más ampliamente extendidos son los denominados RMI (Remote Method Invocation) o LII (Location Independent Invocation).

6.3.2 Sistema Operativo Distribuidos

Básicamente se pueden considerar dos grandes tipos de sistemas operativos distribuidos, DOS. Los sistemas operativos multiprocesador que gestionan los recursos de un multiprocesador y sistemas operativos multicomputadores que son sistemas operativos que están desarrollados para varios ordenadores homogéneos. La funcionalidad para sistemas operativos distribuidos es esencialmente la misma que para los sistemas operativos tradicionales monoprocesador, excepto que permiten gestionar varias CPUs. Por otro lado están los sistemas operativos de red, NOS, ideados para entornos de computadores heterogéneos débilmente acoplados entre sí, a diferencia de los sistemas DOS que están formado por elementos, ya sean procesadores o computadoras homogéneos y fuertemente acoplados [Tane02].

6.3.3 Sistemas de Archivos Distribuidos

Los sistemas de archivos distribuidos permiten la compartición de información en forma de archivos a través de redes de ordenadores. Un servicio de archivos bien diseñado proporciona acceso a los archivos almacenados en un servidor con prestaciones y fiabilidad semejantes a la de los archivos almacenados en discos locales. Un sistema de archivos distribuidos permite a los programas almacenar y acceder a archivos remotos del mismo modo que si fueran locales, permitiendo a los usuarios que accedan a archivos desde cualquier computador de la red. Estos sistemas cumplen con las características de transparencia, los usuarios no diferencian entre archivos locales o distribuidos, hetegoreneidad de hardware y de sistemas operativos, se pueden acceder a archivos que se encuentran en máquinas, redes y sistemas operativos diferentes al de los clientes, tolerancia a fallos, los servicios de archivos continúan funcionando en caso de fallos en los clientes y mantienen los datos consistentes en caso de fallos de los servidores, seguridad, a través del control de acceso se puede controlar el acceso a los diversos recursos del sistemas de archivos en función de los usuarios conectados, este servicio tiene que ver con el concepto de seguridad distribuida.

6.3.4 Seguridad distribuida

En todos aquellos sistemas de computadores, donde haya objetivos potenciales para ataques maliciosos o con fines de diversión, habrá que incluir medidas de seguridad. En los sistemas distribuidos surge la necesidad de garantizar la privacidad, integridad y disponibilidad de recursos. Los ataques contra la seguridad toman las formas de escuchas, suplantación, modificación y denegación de servicio. La criptografía es la base de la autentificación de mensajes así como del secreto y la integridad, para explotarla es preciso utilizar protocolos de seguridad diseñados cuidadosamente. Los requisitos de seguridad principales son, la identificación y autentificación de usuario, la protección de servicios, seguridad robusta para transacciones comerciales y por último el control de acceso para objetos individuales.

6.3.5 Servicios de nombres

Cualquier proceso que necesite acceder a un recurso específico debe poseer su nombre o un identificador. Needham [Need93] distingue entre nombre puro y otros, los primeros son patrones de bits sin interpretara que representan un recurso, mientras que

los segundos incluyen información acerca del objeto al que nombran, como su ubicación. Los nombre puros deben de buscarse antes de poder ser utilizados. Existen varios sistemas que pretenden ofrecer estos servicios, como son los servicios de nombres en Internet, DNS, que son la base para la construcción de nombres no puros denominados URI, los servicios de directorios de información X.500 y los servicios ligeros de directorios como son LDAP.

6.3.6 Transacciones distribuidas

Una transacción define una secuencia de operaciones que se realiza como una unidad y pueden ser realizadas o revocadas. Para formalizar el concepto de transacción, en 1980 Harder an Reuter definieron las famosas reglas ACID [Box98], que debe cumplir todo sistema de transacciones distribuidas:

1. *Atomicidad*, todos los cambios de estado dentro de una transacción son atómicos, es decir son indivisibles con lo que todos los cambios ocurren o no ocurre.

2. *Consistencia*, toda transacción es una transición correcta de estado. Por lo que los cambios realizados en una transacción nunca deben dejar el estado del sistema corrupto o inconsistente.

3. *Aislamiento*, cuando se ejecutan dos o más transacciones de forma concurrente dentro de un sistema, los cambios de estado ocurren como si las transacciones se ejecutasen de forma secuencial. Es decir, están perfectamente sincronizados.

4. *Durabilidad*, una vez se ha realizado una transacción correctamente, y se ha pasado a un estado correcto del sistema, este estado sobrevivirá a cualquier fallo posterior del sistema.

Los sistemas basados en transacciones distribuidas deben permitir las anteriores propiedades. Actualmente en el mercado existen varios sistemas distribuidos transaccionales, como MTS de Microsoft, CCM de CORBA, si bien es en el mundo de las bases de datos donde encuentra su nicho idóneo, como en SQL Server, Oracle Server, DB2, etc.

6.3.7 Replicación

La replicación es la clave para proporcionar alta disponibilidad y tolerancia a fallos en sistemas distribuidos. Se consideran dos sistemas para proporcionar servicios de alta disponibilidad, las arquitecturas de cotilleo y Bayou, las actualizaciones se propagan de forma diferida entre las réplicas de los datos compartidos. Con los sistemas de replicación conseguimos mejorar el rendimiento de los sistemas distribuidos se incrementa la disponibilidad y aumenta la tolerancia a fallos.

6.3.8 Sistemas multimedia distribuidos

Las aplicaciones distribuidas multimedia generan y consumen caudales de datos continuos en tiempo real. Estos contienen grandes cantidades de auto, vídeo y otros datos dependientes del tiempo. Resulta esencial el procesamiento y la entrega a tiempo de los elementos individuales. Los elementos entregados tarde no tienen interés y son desechados normalmente. Conceptos como el ancho de banda, la latencia, la tasa de pérdida de paquetes, la reserva y planificación de recursos de red, toman crucial importancia, como respuesta a las solicitudes de la solicitud de la garantía de la calidad de servicio entre aplicaciones (QoS). Dentro de estos tipos de sistemas, tenemos los sistemas de multimedia basada en Web, que actualmente están emergiendo y posiblemente en un futuro como los sistemas líderes dentro de los sistemas de multimedia distribuidos, otros sistemas son los de telefonía en red, videoconferencias, sistemas colaborativos multimedia y por último los servicios de video bajo demanda.

6.3.9 Memoria compartida distribuida

Los sistemas de memoria compartida distribuida (Distributed Shared Memory DSM) se basan en una abstracción utilizada para la compartición de datos entre procesos en computadores que no comparten memoria física común. De esta forma se pueden utilizar modelos de programación basados en memoria compartida, que tienen alguna ventaja sobre otros modelos, por ejemplo, no es necesario empaquetar los datos para su envío. La memoria distribuida se muestra como memoria en el espacio de direcciones del proceso. El programa de este tipo de sistemas es mantener las prestaciones debido a la grandísima diferencia entre las memorias locales y las comunicaciones entre redes, utilizados para construir estos sistemas. El sistema XDMF es un sistema de integración heterogénea de aplicaciones que utiliza el mecanismo de memoria compartida como elemento de comunicación entre estas [Clarke00].

6.3.10 Computación masiva (Grid/Rejilla)

La computación en rejilla o informática en rejilla, es un nuevo modelo para resolver problemas de computación masiva utilizando gran número de ordenadores organizados en racimo incrustados en una infraestructura de telecomunicaciones distribuida.

La computación en rejilla ha sido diseñada para resolver problemas demasiado grandes por cualquier simple superordenador, mientras mantiene la flexibilidad de trabajar en múltiples problemas más pequeños. Por tanto, la computación en rejilla es un entorno multiusuario.

Debido a esta razón, las técnicas de autorización segura son esenciales para permitir que los recursos informáticos sean controlados por usuarios remotos (distantes).

La computación en rejilla consiste en compartir recursos heterogéneos (basadas en distintas plataformas, arquitecturas de equipos y programas, lenguajes de programación), situados en distintos lugares pertenecientes a diferentes dominios de administración sobre una red que utiliza estándares abiertos. Dicho brevemente consiste en virtualizar los recursos informáticos.

En términos de funcionalidad, las rejillas se clasifican en rejillas computacionales y en rejillas de datos.

La herramienta Globos ha emergido como el estándar de facto para la capa intermedia (middleware) de la rejilla. Globos tiene recursos para manejar:

a) La gestión de recursos (Protocolo de Gestión de Recursos en Rejilla o Gris Resource Management Protocol – GRAM).

b) Servicios de Información (Servicio de Descubrimiento y Monitorización o Monitoring and Discovery Service – MDS).

c) Gestión y Movimiento de Datos (Acceso Global al Almacenamiento Secundario o Global Access to Secundary Storage (GASS) y FTP en rejilla GridFTP.

La mayoría de rejillas que se expanden sobre las comunidades académicas y de investigación de Norteamérica y Europa están basadas en las herramientas Globos Toolkit como núcleo de la capa intermedia.

Los Servicios Web basados en XML ofrecen una forma de acceder a diversos servicios/aplicaciones en un entorno distribuido. Recientemente, el mundo de la informática en rejilla y los Servicios Web caminan juntos para ofrecer la Rejilla como un Servicio Web (Servicio de rejilla). La arquitectura está definida por la Open Gris Services Architecture (OGSA). Serán ofrecidas varias funcionalidades, adheridas a la semántica de los Servicios de Rejilla.

La versión 3.0 de Globos Toolkit, que actualmente se encuentra en fase alfa, será una implementación de referencia acorde con el estándar OGSA.

La rejilla ofrece una forma de resolver los problemas de Gran Reto como el plegamiento de las proteínas y descubrimiento de medicamento, modelización financiera, simulación de terremotos, inundaciones y otras catástrofes naturales, modelización del clima/tiempo, etc. Ofrecen un camino para utilizar los recursos de las tecnologías de la información de forma óptima en una organización.

6.3.11 Coordinación y acuerdo

La mayoría de los sistemas distribuidos surgen como resultado de tomar un determinado objeto, de naturaleza no distribuida, como elemento básico para la distribución. Como ejemplo: tenemos los procesos distribuidos, los documentos distribuidos, los ficheros distribuidos, la memoria distribuida, los objetos distribuidos, etc. Estos elementos se han adaptado de forma transparente para el usuario para se utilizados de forma distribuida. Ahora se consideran sistemas distribuidos donde los componentes que lo forman son inherentemente distribuidos y el problema real de desarrollo nos surge en coordinar las actividades de los diferentes elementos que la componen. Es decir, en vez de centrar el trabajo en ofrecer distribución de forma transparente a los elementos del sistema, ahora nos centramos en coordinar las actividades de estos elementos que de por sí son implícitamente distribuidos.

La aproximación clave de los sistemas basados en coordinación es la separación entre computación y coordinación. Consideramos computación como los diversos procesos que se ejecutan en los diversos elementos distribuidos, llevando cada uno de ellos una actividad concreta e independiente. Mientras que la parte de coordinación la desempeñan el sistema que gestiona toda la comunicación y cooperación entre los procesos. Los sistemas basados en coordinación se centran en ver las diversas formas de llevar a cabo la coordinación de estos procesos independientes.

A continuación se muestra una taxonomía sencilla de los modelos de coordinación:

		TEMPORAL	
		ACOPLADO	DESACOPLADO
REFERENCIAL	ACOPLADO	**DIRECTO**	**BUZÓN**
	DESACOPLADO	**ORIENTADO A CITA**	**COMUNICACIÓN GENERATIVA**

Cuando los procesos están temporal y referencialmente acoplados se dice que la coordinación tiene lugar de forma **directa**. El acoplo referencial ocurre cuando los procesos tienen referencias explicitas a los otros procesos con los que se quieren comunicar. Por ejemplo, cuando un proceso tiene el identificador o nombre del otro proceso con el que quiere comunicar. El acoplo temporal significa que ambos procesos están ejecutándose durante el mismo instante de tiempo. En la comunicación **orientada a mensajes** o asíncrona, ocurre que los procesos no están acoplados temporalmente, pero si referencialmente, en este caso se dice que tenemos una coordinación orientada a buzones. La coordinación **orientada a cita** ocurre cuando los sistemas están acoplados temporalmente, pero no se conocen, desacoplo referencial, son los típicos sistemas de comunicación distribuida de *publicación/suscripción,* como son InfoBus de Saun [Sun98] o MundoCore [Aiten03], para computación ubicua. Pero cuando se habla de coordinación, el modelo más utilizado es la combinación de desacoplo temporal y desacoplo referencial, es decir, los procesos no se conocen y además no están activos al mismo tiempo, es lo que se denomina **comunicación generativa**, fue introducida en el sistema de programación *Linda* [Gelemter85].

6.3.12 EAI Enterprise Architecture Integration

La aproximación típica a la integración de sistemas por parte de EAI es utilizar adaptadores para convertir todo el tráfico a un formato canónico. Para dar acceso a un servicio, se escribe un adaptador que convierta el formato del sistema canónico del sistema EAI y el formato del sistema. De esta forma se puede evitar las N2 conversiones posibles entre N formatos de protocolo y permite construir y mantener los adaptadores de forma sencilla. Sin embargo, esta aproximación penaliza el rendimiento del sistema ya que se realiza la conversión de cada transacción se necesite esta o no. Además los formatos canónicos normalmente cambian, de tal forma que no solucionan el problema de las N2 conversiones a largo plazo.

6.3.12.1 ¿Qué es EAI?

Para el propósito de este portal (EAI.ITtoolbox.com), ITtoolbox ha definido la Integración de Aplicaciones de Empersa o EAI, como la combinación de procesos, software, estándares y hardware dando como resultado la integración homogénea de dos o más sistemas de empresa permitiendo que estas operen como una. Aunque EAI se asocia a menudo con la integración de sistemas dentro de una empresa concreta, EAI puede referir también a la integración de sistemas de empresas de entidades corporativas dispares o lo que es también llamado B2Bi cuando el objetivo es permitir transacciones sencillas entre múltiples sistemas.

6.3.12.2 Breve historia de EAI

Las aplicaciones para empresas, desde principios de los 60 hasta finales de los 70, eran sencillas en estructura y funcionalidad, desarrollando principalmente tareas repetitivas. “No había pensamientos acerca de la integración de los datos corporativos. El objetivo consistía en replicar las actividades manuales con el ordenador”

Sobre los 80, varias empresas se dieron cuenta de la necesidad y el valor que conlleva la integración de aplicaciones. Comenzaron a surgir retos, conforme las empresas de tecnología de la información intentaban rediseñar aplicaciones ya implementadas para añadirles la funcionalidad de integrabilidad. Las típicas tareas consistían en realizar proceso de transacciones operacionales (asociadas con planificación de recursos de empresa ERP) en sistemas diseñados para procesado de datos (data warehousing) [Inmon01].

Conforme las aplicaciones ERP eran más profusas en los 90, surgió una necesidad por parte de las empresas para permitir aplicaciones y datos ya existentes dentro de los sistemas ERP, esto sólo se podía hacer incluyendo EAI [Ren01]. Frances Ren escribió que el paso a EAI es un resultado lógico “Unas compañías una vez que usaron la tecnología cliente/servidor para construir sus aplicaciones, se dieron cuenta después que necesitaban en lazar los múltiples procesos de la empresa ”Otro tema que potenció el mercado de EAI fue la proliferación de aplicaciones empaquetadas, aplicaciones que solucionaban el problema potencial del año 2000, integración de cadenas de suministro gestionadas entre empresas (B2B), integración de aplicaciones Web y los avances tecnológicos generales dentro del desarrollo de EAI [Hey00].

6.3.12.3 Estructura de EAI

La estructura de EAI es compleja y afecta a cada uno de los niveles de un sistema empresarial: su arquitectura, hardware, software y procesos. En ITtoolbox se definen los siguientes niveles de integración:

- Integración de Procesos de Negocios (BPI): Se deben definir, activar y gestionar los procesos de intercambio de información en la empresa dentro de los diversos sistemas de negocio. Esto permite a las empresas optimizar las operaciones, reducir costos y mejora la capacidad de respuesta a las demandas de los clientes [Yee01]. Los elementos principales implicados son: gestión de procesos, modelado de procesos y flujos de trabajo que lleva consigo las tareas, procedimientos, organizaciones,

entradas y salidas de información y las herramientas necesarias en cada paso dentro de los procesos de negocios.

- Integración de Aplicaciones: A este nivel de integración el objetivo es llevar datos o funcionalidad de una aplicación a otra en tiempo real. La integración de aplicaciones se utiliza para: integración entre empresas (B2B Integration), integración de aplicaciones de relación con los clientes (Customer relationship management CRM) con las aplicaciones general de una empresa, integración de sitios Web que gestionan diferentes sistemas de procesos de las empresas. La integración a medida suele ser necesaria, principalmente cuando se intenta integrar una aplicación heredada, ya existente, con alguna aplicación de gestión de recursos reciente o ERP (Enterprise Resources Planning).

- Integración de Datos: Para conseguir Integración de aplicaciones e Integración de negocios de forma adecuada, se debe atacar el problema de la integración de sistemas de bases de datos. Antes de realizar la integración de datos, se deben identificar los datos, catalogarlos y construir un metamodelo para la gestión de varios almacenes de datos. Una vez hecho esto, los datos pueden ser distribuidos o compartidos entre los diferentes sistemas de bases de datos.

- Estándares de Integración: Antes de conseguir una completa Integración de datos, se deben seleccionar los formatos estándares, estos permiten la compartición y distribución de información y datos de la empresa, estos estándares son el núcleo de la Integración de las Aplicaciones de las Empresas EAI. Algunos de estos son: DCOM, CORBA, EDI, Java RMI, Servicios Web.

- Integración de Plataformas: Para completar la integración de sistemas, la arquitectura subyacente, el software y el hardware, es necesario integrar las diversas necesidades en redes heterogéneas. La Integración de plataformas está relacionado con los procesos y herramientas necesarias para que la comunicación entre sistemas tenga lugar, de forma óptima y segura, de tal forma que los datos pasen por diferentes aplicaciones sin dificultades. Por ejemplo, descubrir como una máquina Windows pasa información de forma fiable a una máquina UNIX es una compleja tarea cuando se desea integrara una sistema corporativo.

6.3.12.4 El futuro de EAI

IDC Research espera que los servicios del mercado EAI, sean los más importantes y de rápido crecimiento dentro del creciente sector de las tecnologías de la información. Los beneficios de este mercado se estiman que serán superiores a 5 mil millones de dólares este año. Esto representa un crecimiento anual del 30%, se espera

que se mantenga durante los próximos años. Mientras que el resto de sectores tecnológicos solamente crecerán un 11%. IDC indica además que Norte América y la comunidad económica europea generarán el 90% de la demanda global [Ren01].

Capítulo 7:
Plataformas LII

Una vez identificada la problemática que surge debido a la heterogeneidad de los sistemas middleware basados en invocación dinámica de métodos. En éste capítulo estudiaremos en detalle algunas de las plataformas middleware RMI más representativas como son: CORBA, DCOM y WEB Services. Esto lo haremos con el objetivo de identificar cualidades comunes que permitan abstraerlas para poder generalizarlas y tratarlas a todas, de forma homogénea, a pesar de su heterogeneidad. Al final del capítulo veremos que esto es posible conseguirlo y que, además, hay un gran parecido con las estructuras esenciales de las máquinas abstractas identificadas en la sección 4.5.

7.1 CORBA

7.1.1 Introducción

La Common Object Request Broker Architecture (CORBA) está estructurada de tal forma que nos permite la integración de una gran variedad de objetos dentro de diferentes sistemas.

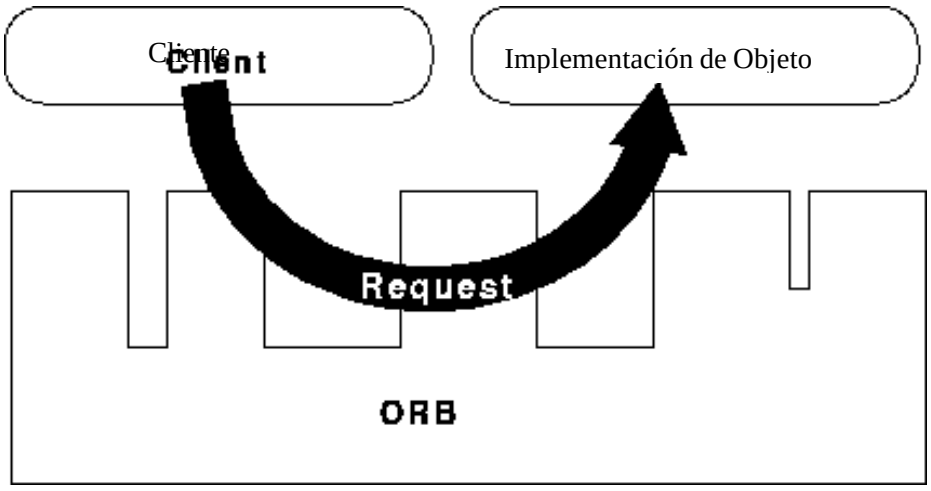


Ilustración 7.1: Object Request Broker

Consta de cuatro elementos o partes fundamentales: 1) Object Request Broker (ORB) define el bus de objetos de CORBA, 2) Common Object Services, define los servicios de objetos, 3) Common Facilities define los marcos de trabajo (frameworks) de aplicaciones horizontales y verticales que se utilizan directamente por objetos de negocios y 4) Aplication Object, son los objetos de negocios y aplicaciones. En la Ilustración 7.1 se muestran estos cuatro elementos.

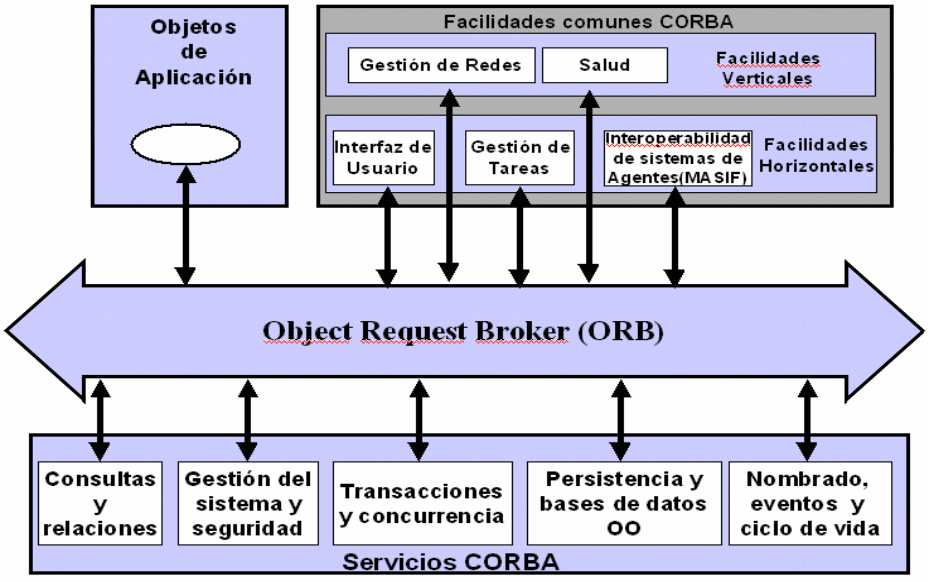


Ilustración 7.2: Arquitectura de gestión de objetos OMG.

Los siguientes apartados van a trata comentar estos cuatro elementos comenzando por el ORB, seguido de los Common Services y Common Facilities y por último se describe como construir un Application Object.

7.1.2 Estructura de ORB (Object Request Broker)

En la figura anterior podemos ver de modo esquemático la utilización de servicios ofrecidos por un objeto, el servidor, por parte de otro objeto, el cliente, a través del bus ORB o gestor de peticiones a objetos (Object Request Broker), este es el responsable de todos los mecanismos necesarios para realizar esta petición de recursos de objeto, que aunque a simple vista resulta muy simple, son necesario un conjunto muy sofisticado de mecanismos para conseguir esto.

El ORB es el responsable de todos los mecanismos necesarios para encontrar el objeto al que se desea hacer la petición de servicios, prepararlo para que reciba esta petición e indicarle los datos necesarios que necesita en dicha petición. La interfaz que el cliente ve es completamente independiente de donde se encuentre el objeto, así como del lenguaje de programación en el que este halla sido programado.

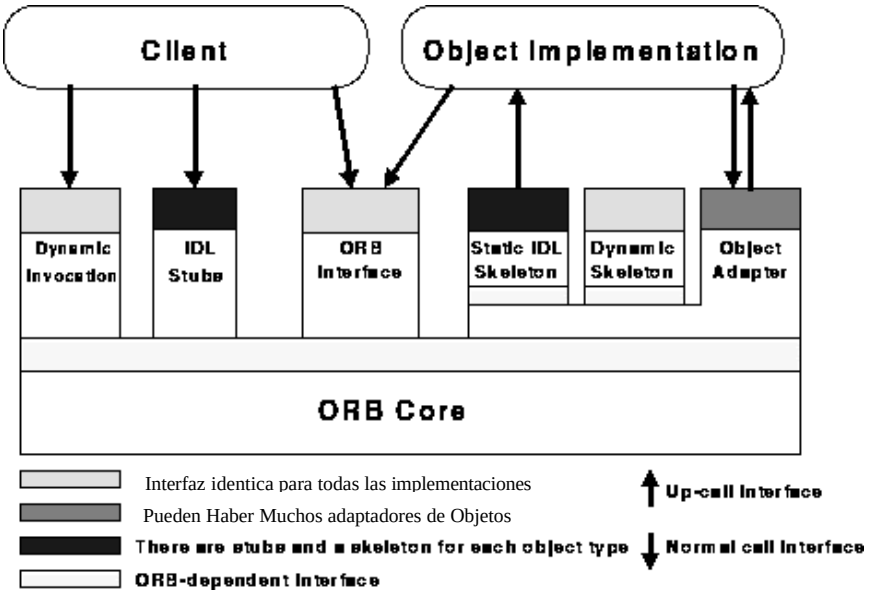


Ilustración 7.3: La estructura de la Interfaz ORB.

La Ilustración 7.3 nos muestra la estructura de un ORB. Las interfaces se muestran como las cajas con recuadros y las flechas indican si se llama al ORB o es este el que llama a los objetos a través de las interfaces dependiendo de donde apunten.

Para realizar una petición, el cliente puede usar el interfaz dinámico de invocación o DII que es una misma interfaz independientemente de la interfaz del objeto al que se dirige la petición o por otro lado se puede usar un 'stub' descrito mediante IDL (Lenguaje de Descripción de Interfaces), además el cliente también puede interactuar directamente con el ORB para realizar algunas tareas determinadas.

La implementación del objeto recibe la petición a través de un 'skeleton' general definido mediante IDL o a través de un 'skeleton' dinámico.

La interfaz de un objeto puede definirse de dos formas, una es estáticamente mediante el mencionado anteriormente IDL, este lenguaje describe los tipos de objetos de acuerdo a las operaciones que puede realizar y a los parámetros que dichas operaciones acepta; la otra forma es añadiendo la interfaz a un repositorio o almacén de

interfaces, de esta forma se puede realizar acceso en tiempo de ejecución a la información de las interfaces de los objetos. En todas las implementaciones de ORB el almacén de interfaces debe ser igualmente de potente a nivel de expresividad de la semántica que cualquier definición IDL.

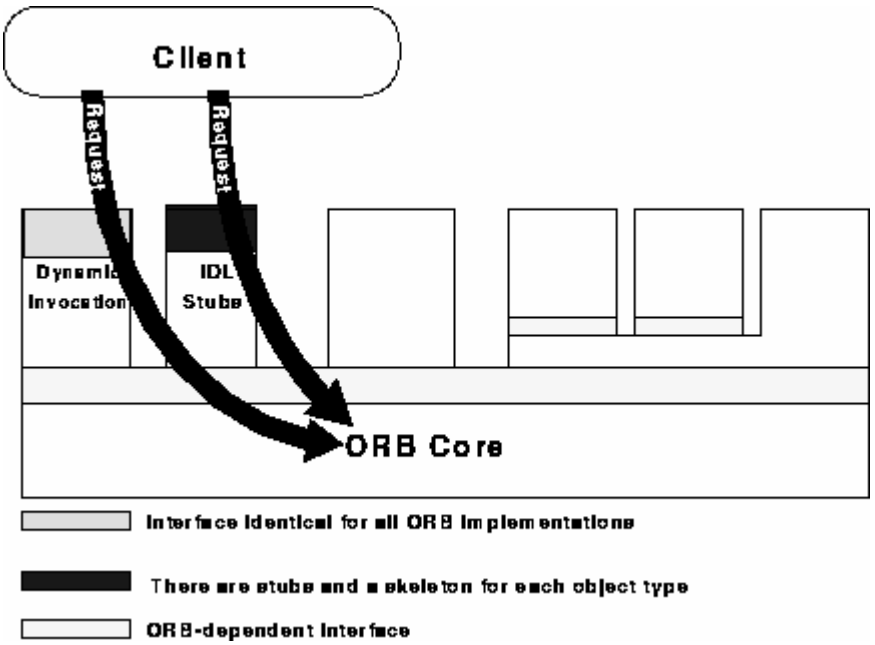


Ilustración 7.4: Un cliente usando el Stub estático o el DII dinámico.

Un objeto debe realizar una petición a un objeto a través de una referencia al mismo además debe conocer el tipo de objeto y la operación que desea realizar. El cliente inicia la petición llamando a las rutinas 'stub' que son específicos para cada objeto o construyendo una petición dinámicamente a través del DII.

Las interfaces dinámicas y 'stub' que nos permiten invocar a un método tienen la misma potencia semántica, y el receptor del mensaje no necesita saber de qué forma se ha realizado la petición.

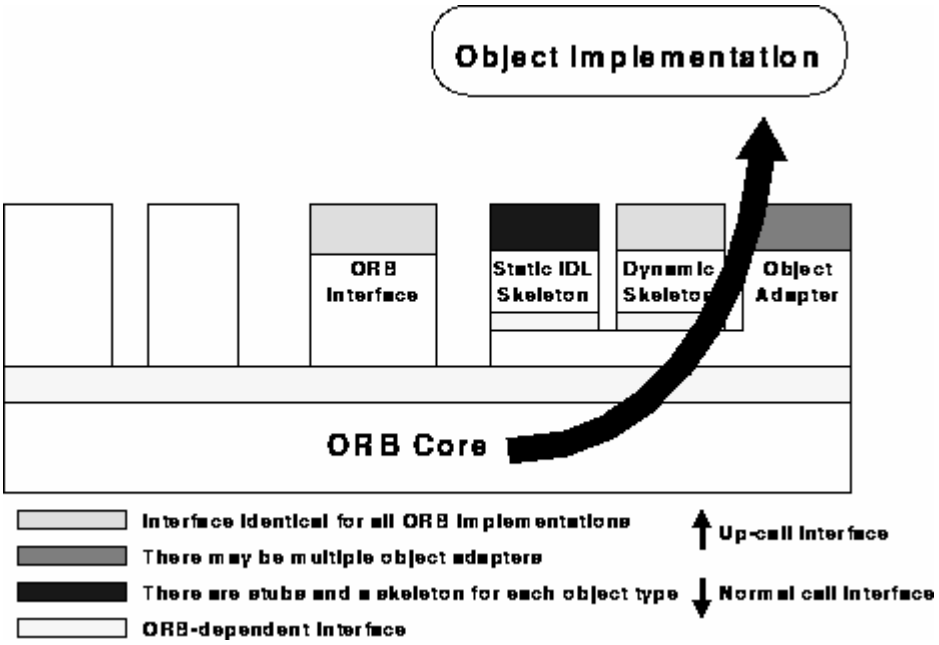


Ilustración 7.5: El objeto recibe la petición de servicios.

El ORB se encarga de localizar el objeto adecuado, transmitirle los parámetros y transferir el control al objeto ya sea a través del 'skeleton' general o del dinámico. Los 'skeletons' son específicos de la interfaz y del objeto adaptador. Una vez que la petición ha sido completada, tanto el control como los valores de salida son devueltos al cliente.

El objeto servidor debe decidir que adaptador de objeto usar si el dinámico o el estático dependiendo de los tipos de servicios que este objeto quiera ofrecer

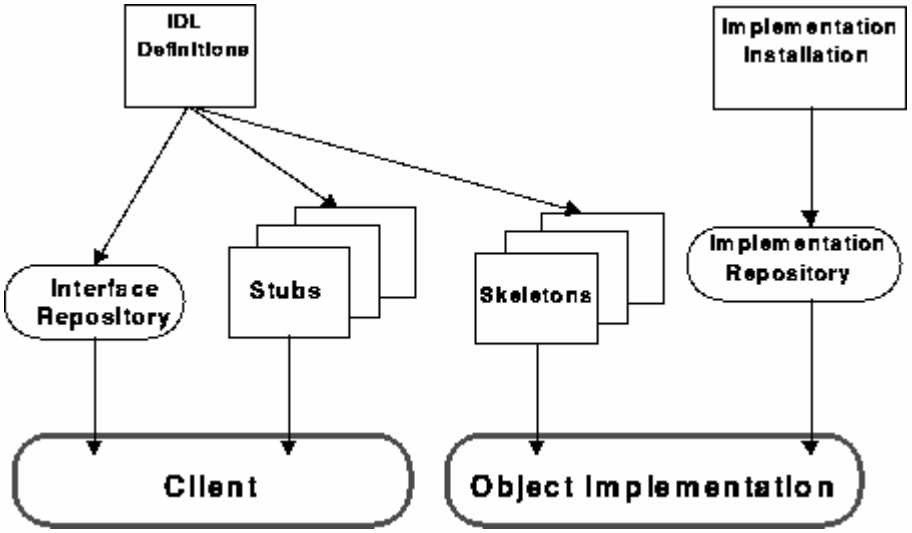


Ilustración 7.6: Repositorio de Interfaces y de Implementaciones.

La Ilustración 7.6 nos enseña como se nos muestra la información tanto de la interfaz como de la implementación tal y como el objeto servidor y cliente la ven. La interfaz está definida en IDL o en el Almacén de Interfaces, estas definiciones se usan para general tanto el 'stub' ya sea dinámico o estático en la parte del cliente como del

'skeleton' ya sea estático o dinámico en la parte del objeto servidor o en la implementación del objeto.

La información sobre la implementación del objeto está disponible en tiempo de instalación y se guarda en el Almacén de Implementaciones para usarlo durante el envío de la petición.

7.1.2.1 El ORB

La arquitectura CORBA no obliga a que la implementación del ORB sea realizada mediante un componente, sin embargo se definen sus interfaces. Cualquier implementación de ORB que proporciones estas interfaces se considera apropiado, estas están organizadas en tres categorías:

Las operaciones que son la misma para todas las implementaciones del ORB.

Las operaciones que son específicas para un tipo particular de objetos.

Las operaciones que son específicas para un estilo particular de implementación de objeto.

Los ORB se pueden implementar de diferentes formas por los distintos fabricantes, que junto con los compiladores de IDL, los almacenes o repositorios y los objetos adaptadores, nos proporcionaran un conjunto de servicios a los clientes y a las implementaciones de los objetos que tendrán diferentes propiedades y calidades.

Puede haber múltiples implementaciones de ORB con tienen diferente representación de referencias a los objetos y diferentes formas de realizar la invocaciones a estos, puede que un cliente tenga acceso a dos referencias a objetos gestionadas por diferentes implementaciones de ORB, pero estos deben estar diseñados para trabajar juntos y distinguir entre sus propias referencias, en ningún momento el cliente debe ser responsable o debe tener información de cómo las referencias a los objetos en los distintos ORB funcionan esto es responsabilidad de los propios ORB.

El núcleo del ORB es la parte del ORB que proporciona la representación básica de los objetos y permite el envío de peticiones. CORBA está diseñada para permitir diferentes mecanismos de objetos y para ellos estructura el ORB con las partes que van sobre el núcleo de ORB que proporcionan las interfaces que enmascaran las diferencias entre los diferentes núcleos dentro de diferentes implementaciones de ORB.

7.1.2.2 Los Clientes

Un cliente de un objeto tiene acceso a una referencia de ese objeto y puede invocar operaciones de dicho objeto. Un cliente solo conoce la estructura lógica de un objeto de acuerdo con sus interfaces y el comportamiento del objeto tras realizar alguna invocación. Es importante darse cuenta de que cuando hablamos de cliente, como un programa o proceso, que inicia peticiones en otro objeto ,siempre va ha ser relativo a ese objeto al que se realizan las peticiones porque puede que el objeto cliente sea a su vez un servidor de otro cliente que le realiza peticiones a él.

Los clientes generalmente ven los objetos y las interfaces ORB desde el punto de vista de una adaptación del lenguaje de programación en el que se están programando, de tal forma que estos adaptadores acercan el ORB al nivel del programador. Los clientes son portables al máximo y funcionan sin necesidad de realizar cambios en las fuentes aunque cambie el ORB al que se realicen las peticiones e incluso este no acepte adaptación para el lenguaje en el que se programó el cliente. Los clientes no tienen conocimiento sobre, la implementación de los objetos, que objetos adaptadores es utilizado por la implementación o que ORB es usado para acceder a ellos.

7.1.2.3 Implementaciones de Objetos

La implementación de un objeto o el objeto servidor proporcionan la semántica de dicho objeto, generalmente mediante la definición de datos para la instancia del objeto y código para los métodos del objeto. A menudo la implementación usará otros objetos o programas adicionales para implementar el comportamiento de este. En algunos casos, la función primaria del objeto es realizar operaciones sobre otras cosas que no son objetos.

La implementación de un objeto puede proporcionar varios tipos de servicios como, un servidor separado, librerías, un programa por método, una aplicación encapsulada, una base de datos orientada a objetos, etc. A través del uso de diversos adaptadores de objetos es posible realizar cualquier estilo de implementación de objeto.

Normalmente el objeto servidor no depende del ORB o de cómo el cliente realiza la invocación, si es dinámica o estáticamente, el objeto servidor puede seleccionar interfaces para servicios dependientes del ORB para elegir que Adaptador de Objetos va a utilizar.

7.1.2.4 Referencias a Objetos

Una referencia a un objeto es la información necesaria para identificar un objeto dentro de un ORB. Tanto el cliente como el servidor tienen una noción no transparente de las referencias de objetos según el adaptador de lenguaje. Dos implementaciones de ORB pueden diferir en la elección de la representación de las referencias, pero eso al cliente no debe importarle.

La representación de la referencia de un objeto que gestiona un cliente es solo válida durante el tiempo de vida del cliente.

Todos los ORB deben proporcionar la misma adaptación para los lenguajes a referencias. Esto permite a un programa escrito en un lenguaje particular acceder a las referencias independientemente del ORB. El adaptador de lenguaje puede a su vez proporcionar una forma de acceder a las referencias de una forma adecuada para la conveniencia del programador.

Las referencias de los objetos nos proporcionan la información que necesitamos para identificar unívocamente a un objeto dentro de un ORB a un objeto. En la definición de CORBA no se indica como hacerlo, con lo que es dependiente de la implementación, por lo que deben crearse mecanismo para interconectar objetos de diferentes ORB. CORBA 2.0 define la Interoperable Objet References (IOR) que deben usar los vendedores para pasarse referencias entre objetos.

Para llamar a referencias de objetos que están en diferentes ORB, CORBA nos da dos funciones: object_to_string y string_to_object que nos ayudan a gestionar esto de tal forma que un programa puede utilizar estas dos funciones para obtener el nombre en forma string de un objeto y obtener una referencia a este y viceversa.

7.1.2.5 El IDL de OMG

El lenguaje de definición de interfaz de OMG o IDL define los tipos de objetos especificando sus interfaces. Una interfaz consiste en un conjunto de operaciones y parámetros para dichas operaciones. Véase que, aunque IDL proporcional el marco conceptual para describir los objetos manipulados por el ORB, no es necesario para que este use los fuentes en IDL para trabajar correctamente.

El lenguaje IDL es la forma por la cual un objeto servidor puede decirle a sus potenciales clientes que operaciones están disponibles y como deben invocarse. Con las definiciones de IDL, es posible adaptar objetos CORBA dentro de lenguajes particulares de programación o sistemas de objetos.

7.1.2.6 Adaptación de IDL a los lenguajes de programación

Se pueden acceder a los objetos CORBA de diferentes formas, independientemente de si el lenguaje de programación es orientado a objetos o no. Para estos últimos, puede ser deseable ver los objetos CORBA como objetos del lenguaje de programación. Incluso para los lenguajes no orientados a objetos, es una buena idea ocultar la representación de las referencias, los nombres de los métodos, etc. Una adaptación particular de IDL para un lenguaje de programación debe ser el mismo para todas las implementaciones del ORB. La adaptación de lenguajes incluye definición de tipos de datos específicos del lenguaje e interfaces a procedimientos para acceder a los objetos a través del ORB. Incluyendo la estructura de la interfaz ‘stub’ del cliente, que no es necesaria para los lenguajes orientados a objetos, la interfaz de invocación dinámica, el skeleton del objeto servidor, los adaptadores de objetos y la interfaz directa al ORB.

7.1.2.7 Los Stubs

Para la adaptación de un lenguaje no orientado a objetos deberá haber una interfaz de programación para el ‘stub’ de cada tipo de interfaz. Normalmente, el ‘stub’ realizará accesos a las operaciones definidas mediante IDL en el objeto de tal forma que sea fácil para el programador. El ‘stub’ realiza llamadas al resto del ORB usando interfaces privados y posiblemente optimizados para cada núcleo particular de cada ORB. Si hay disponible más de un ORB pueden existir diferentes ‘stubs’ correspondientes con diferentes ORB, en tal caso se hace necesario para el ORB y el mapeador del lenguaje cooperaren para asociar el ‘stub’ correcto con la referencia del objeto apropiado.

Los lenguajes de programación orientados a objetos tales como C++ y Smalltalk no necesitan interfaces stubs.

7.1.2.8 Las Interfaces de Invocación Dinámicas (DII)

Otra cosa que podemos hacer es construir dinámicamente la invocación a un objeto, es decir, en lugar de llamar a una rutina ‘stub’ que es específica para una

operación determinada en un objeto determinado, un cliente puede especificar el objeto a ser invocado, la operación a realizar y el conjunto de parámetros para la operación y una llamada o una secuencia de llamadas. El código del cliente debe proporcionar información sobre la operación a realizar y los tipos de parámetros pasados, quizás obtenidos de un Almacén de Interfaces u otra fuente en tiempo de ejecución. La naturaleza de la interfaz de invocación dinámica puede variar substancialmente de un lenguaje de programación a otro.

7.1.2.9 El Skeleton

Para un lenguaje de adaptación particular, y posiblemente, dependiendo del objeto adaptador habrá, una interfaz para los métodos que implementa cada tipo de objeto. La interfaz normalmente será una interfaz que llama a estos métodos, de tal forma que, el objeto servidor debe tener las rutinas apropiadas, que cumplan con esta interfaz y que el ORB las llama a través del 'skeleton'.

La existencia de un 'skeleton' no implica la existencia de su correspondiente 'stub', los clientes pueden realizar peticiones a través de DII.

Es posible escribir un objeto adaptador que no use 'skeletons' para invocar a métodos de la implementación del objeto, por ejemplo puede ocurrir que se creen las implementaciones dinámicamente para lenguajes como Smalltalk.

7.1.2.10 El Skeleton Dinámico

Existe una interfaz que nos permite la gestión de la invocaciones a objetos de forma dinámica, es decir, en lugar de acceder a través de un 'skeleton' que especifica una operación particular, la implementación de un objeto es alcanzada a través de una interfaz, que proporciona el acceso a las operaciones y a los parámetros, de una forma análoga a la parte cliente, mediante una Interfaz de Invocación Dinámica. Se puede usar conocimiento puramente estático, para realizar las invocaciones, pero también, se puede adquirir conocimiento dinámico, posiblemente a través del Almacén de Interfaces.

El objeto servidor debe proporcionar la descripción de todos los parámetros de todas las operaciones al ORB, y este proporciona los valores de cualquier parámetro de entrada, para realizar la operación, además debe proporcionar los valores de cualquier parámetro de salida o excepción al ORB después de haber realizado la operación. La naturaleza de la interfaz 'skeleton' dinámica, puede variar sustancialmente de un lenguaje de programación a otro, o de un objeto adaptador a otro, pero será generalmente una interfaz que realiza llamadas al objeto.

Las invocaciones a los 'skeletons' dinámicos pueden realizarse a través tanto de objetos dinámicos como estáticos o 'stubs', ambos van a proporcionarnos idénticos resultados.

7.1.2.11 Adaptadores de Objetos

Un adaptador de objetos es la forma primaria que tiene un objeto servidor para acceder a los servicios proporcionados por el ORB. Se espera que haya unos cuantos objetos adaptadores que estarán ampliamente disponibles con las interfaces apropiadas para tipos específicos de objetos. Los servicios proporcionados por el ORB a través de los Objetos Adaptadores a menudo incluyen: generación e interpretación de las

referencias a objetos, invocación de métodos, seguridad de interacción, activación y desactivación de objetos e implementaciones, adaptar referencias de objetos a las implementaciones y el registro de las implementaciones.

El amplio margen de granularidades de objetos, tiempo de vida, políticas, estilos de implementación y otras propiedades hacen difícil para el núcleo del ORB proporcionar interfaces simples para todos los objetos, de esta forma a través de los Objetos Adaptadores, es posible para el ORB asociar grupos particulares de objetos servidores que tienen requisitos similares con interfaces asociadas a ellos.

7.1.2.12 La interfaz ORB

La interfaz ORB es la que va directamente a el ORB, que además es la misma para todas las diferentes implementaciones de ORB y no dependen, ni de la interfaz del objeto, ni del adaptador de objetos, debido a que la mayoría de la funcionalidad de el ORB es proporcionado a través de los objetos adaptadores, stubs, skeleton o invocaciones dinámicas. Hay solo unas cuantas operaciones que son comunes a todos los objetos, estas son útiles para el cliente como la implementación de los objetos.

7.1.2.13 El repositorio de Interfaces

El repositorio de interfaces o almacén de interfaces es un servicio que proporciona objetos persistentes, que representan la información especificada mediante el lenguaje IDL en tiempo de ejecución. La información del almacén de interfaces puede ser usada por el ORB para realizar peticiones. Además, usando esta información, es posible para un programa, encontrar el objeto cuya interfaz no se conocía cuando se compiló el programa, determinar que operaciones son válidas en el objeto y realizar la invocación a sus operaciones.

Además del papel que desempaña en el funcionamiento del ORB, el Almacén de Interfaces, es un lugar común para almacenar información adicional asociada con las interfaces de los objetos ORB, por ejemplo se podrían asociar con los almacenes de interfaces: la información de depuración, librerías de stubs y skeletons, rutinas que pueden formatear o visualizar determinados tipos de objetos, etc.

7.1.2.14 El repositorio de Implementaciones

El repositorio de implementaciones o almacén de implementaciones contiene información que permite al ORB localizar y activar las implementaciones de determinados objetos, aunque la mayoría de la información en el Almacén de Implementaciones es específica a un ORB o entorno de funcionamiento, este es el sitio para poner este tipo de información, generalmente la instalación de implementaciones y el control de políticas de activación y ejecución de implementaciones de objetos se realiza a través de operaciones sobre el Almacén de Implementaciones.

Al igual que el Almacén de Interfaces el Almacén de Implementaciones puede ser usado para tener un lugar común donde poder localizar todo tipo de objetos como: información de depuración, control administrativo, localización de recursos, seguridad. etc.

7.1.3 La estructura de un cliente

El cliente de un objeto tiene una referencia que sirve para localizar e identificar a ese objeto, esta referencia puede ser usada como un representante del objeto de tal manera que puede ser invocada o pasada como parámetro en la invocación de una operación sobre un objeto diferente. La invocación de un objeto lleva consigo especificar el objeto a ser invocado, la operación que debe ser realizada y los parámetros que deben pasarse a la operación.

El ORB gestiona la transferencia de control y de datos hacia el objeto servidor y se lo devuelve al cliente. Si el ORB no puede realizar la invocación, una excepción surge indicando el tipo de error producido. Normalmente un cliente llama a una rutina en su programa que realiza la invocación y devuelve cuando la operación ha sido completada.

Los objetos acceder a los stubs específicos de cada objeto como a librerías de rutinas en su programa, ver la Ilustración 7.7. El cliente de esta forma ve simplemente rutinas normales en su lenguaje de programación. Todas las implementaciones proporcionaran unos tipos de datos específicos al lenguaje a usar, para hacer referencia a objetos, generalmente un puntero. El cliente entonces pasa esa referencia de objeto a la rutina stub que inicia una invocación. Los stubs tienen acceso a la representación de las referencias de los objetos e interactúan con el ORB para realizar la invocación.

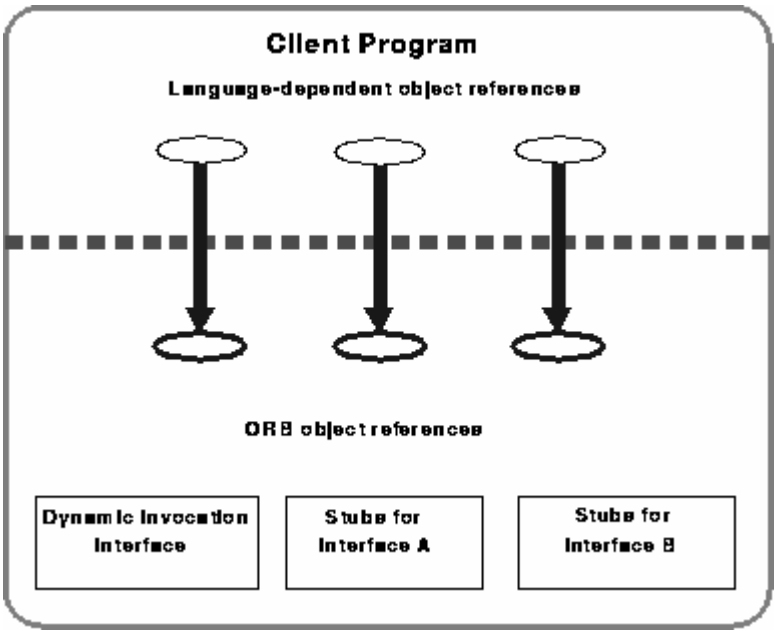


Ilustración 7.7: La estructura de un cliente típico.

Existe un conjunto de librerías disponibles para realizar invocaciones a objetos de una forma alternativa, por ejemplo, cuando un objeto no se había definido aun en tiempo de compilación. En este caso, el cliente proporciona información adicional nombre del objeto y del método a ser invocado y realiza una serie de llamadas para especificar los parámetros e inicia la invocación.

Los clientes generalmente obtienen referencias a objetos a través de parámetros de salida de invocaciones a otros objetos para los que tienen referencia. Cuando un

cliente es también una implementación, recibe las referencias de objetos como parámetros de entrada en la llamada a los objetos que implementa. Una referencia a un objeto puede ser convertida a una string, que puede ser almacenada en un fichero o enviada por diferentes medios, y después vuelta a convertir en una referencia a un objeto por el ORB que ha proporcionado la string.

7.1.4 La Estructura de la Implementación de un Objeto

La implementación de un objeto proporciona el estado y comportamiento actual de un objeto, esta puede estar estructurada de varias formas. Definen los métodos para que realicen las operaciones, una implementación definirá procedimientos para activar y desactivar objetos y usará otras facilidades de objetos o no para conseguir la persistencia en el estado de un objeto, para controlar el acceso al objeto así como para implementar los métodos.

La implementación del objeto, ver Ilustración 7.8, interactúa con el ORB de varias formas para establecer su identidad, crear nuevos objetos y obtener servicios dependientes de ORB, esto lo realiza principalmente mediante el Adaptador de Objetos, que proporciona una interfaz a los servicios ORB.

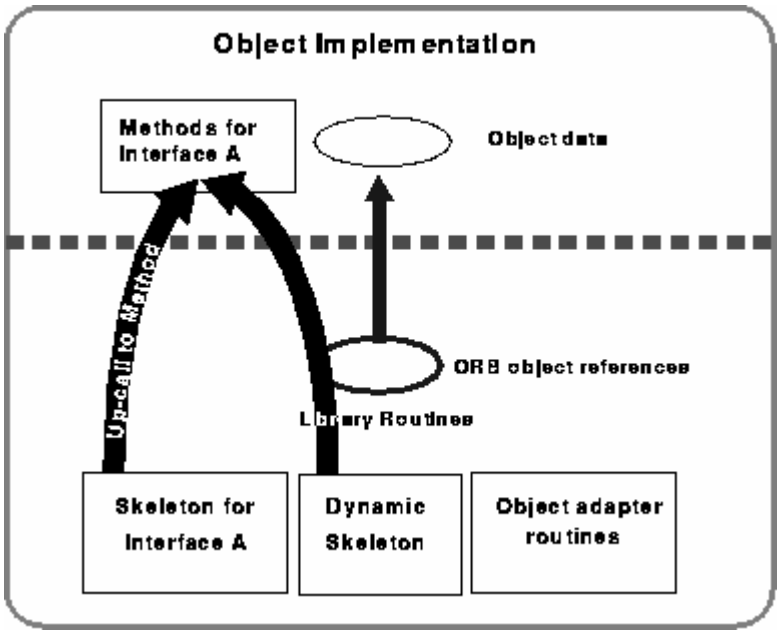


Ilustración 7.8: Estructura de la implementación típica de un objeto.

Debido a la gran cantidad de posibles implementaciones es difícil definir cómo, en general, un objeto está estructurado.

Cuando una invocación ocurre, el núcleo del ORB, el adaptador de objetos y el skeleton son los encargados de que la llamada se realice sobre el método apropiado de la implementación. Un parámetro para ese método especifica el objeto a ser invocado, que puede ser usado para localizar los datos para el objeto. Además, los parámetros se proporcionan de acuerdo a la definición del skeleton. Cuando el método acaba, vuelve, causando que los parámetros de salida o la excepción en caso de error, se transfieran de nuevo al cliente.

Cuando un nuevo objeto se crea, el ORB debe ser notificado para que sepa donde encontrar su implementación. Generalmente, la implementación se registra así misma, como el objeto implementación de una interfaz particular y especifica, cómo iniciar la implementación en el caso de que no está ejecutándose.

La mayoría de las implementaciones de los objetos consiguen su funcionalidad a través de facilidades, además, a través del ORB y el Adaptador de Objetos. Por ejemplo, aunque el Adaptador de Objeto Básico (BOA) proporciona persistencia a los datos de un objeto, esta pequeña cantidad de datos se usa como un identificador para el objeto actual en el servicio de almacenamiento que el objeto elija. De esta forma, no solo es posible para diferentes objetos usar el mismo servicio de almacenamiento, si no que además permite a los objetos elegir el servicio que les sea más apropiado para sus necesidades.

7.1.5 La Estructura de un objeto adaptador

El objeto adaptador es el medio principal por el que un objeto implementación accede a los servicios ORB, tales como generación de referencia de objeto. Un objeto adaptador exporta una interfaz pública a la implementación del objeto y una interfaz privada al skeleton. Se construye en una interfaz dependiente de ORB privada.

Los adaptadores de objetos son los responsables de las siguientes funciones:

- Generación e interpretación de las referencias a objetos.
- Invocación de los métodos.
- Seguridad de interacción.
- Activación de objetos e implementaciones.
- Adaptar las referencias de los objetos a sus correspondientes objetos implementaciones.
- Registro de las implementaciones.

Estas funciones se realizan usando el núcleo de ORB y algunos componentes adicionales. A menudo un adaptador de objetos podrá mantener su propio estado para realizar una tarea. Puede ocurrir que un adaptador de objetos particular delegue una o más de sus responsabilidades al núcleo sobre el que está construido.

Como se muestra en la Ilustración 7.9, el adaptador de objetos se ve envuelto implícitamente, en la invocación de los métodos, aunque la interfaz directa es a través del skeleton. Por ejemplo el adaptador de objetos puede ser usado para activar la implementación o autentificar la petición.

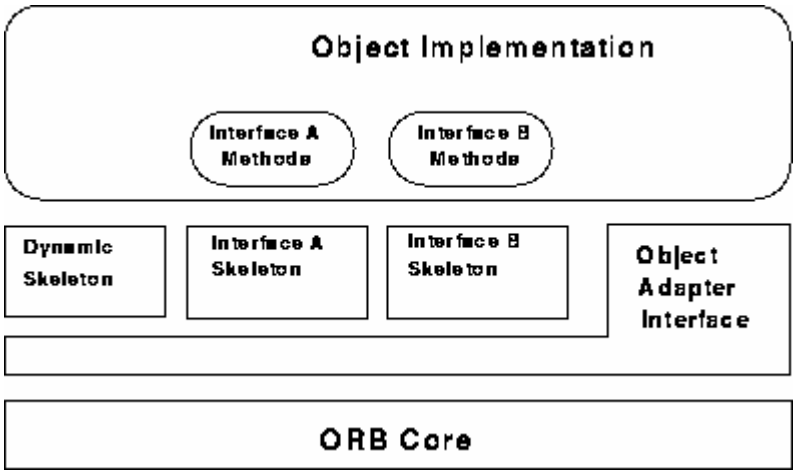


Ilustración 7.9: La Estructura de un objeto adaptador típico.

El adaptador de objetos define la mayoría de los servicios de ORB de los que la implementación de Objetos suele depender. Diferentes ORB proporcionarán diferentes niveles de servicio y, diferentes entornos de funcionamiento, pueden proporcionar algunas propiedades implícitamente y necesitar otras para ser añadidas por el adaptador de objetos. Por ejemplo, es normal para el objeto implementación, desear almacenar ciertos valores en la referencia a un objeto, para identificar fácilmente al objeto en una invocación. Si el adaptador de objetos permite a la implementación especificar estos valores, cuando un nuevo objeto es creado, entonces puede ser capaz de guardarlos en la referencia al objeto para los ORB que lo permitan. Si el núcleo del ORB no proporciona estas características, el adaptador de objetos debería recordar este valor en su propio almacén y proporcionarlo a la implementación en una invocación. Con los adaptadores de objetos es posible para una implementación de objeto tener acceso a los servicios estén o no implementados en el núcleo del ORB, si este los proporciona este simplemente proporciona una interfaz para ello, si no el adaptador debe implementarlo sobre el núcleo del ORB. Cada instancia de un adaptador particular debe proporcionar la misma interfaz y los mismos servicios para todos los ORB donde esté implementado.

También es necesario para todos los adaptadores de objetos proporcionar la misma interfaz o funcionalidad. Algunas implementaciones de objetos necesitan requisitos especiales, por ejemplo una base de datos orientada a objetos podría registrar implícitamente sus miles de objetos sin necesidad de realizar las llamadas individuales al objeto adaptador. En tal caso, sería imposible e innecesario para el adaptador de objetos mantener ningún estado por objeto. Usando una interfaz de objeto adaptador que está diseñada especial mente para estos tipos de objetos sería posible obtener ventaja de un particular núcleo ORB para obtener el acceso más efectivo posible al ORB.

7.1.6 Integración de sistemas externos

CORBA está diseñado para permitir la interoperación en con un amplio rango de sistemas de objetos, véase la Ilustración 7.10. Debido a la gran cantidad de sistemas de objetos, una idea común es permitir a todos estos objetos de estos sistemas ser accesibles a través de ORB, para los sistemas de objetos que son ellos mismos ORB, pueden ser conectados a otros ORB, a través del mecanismo descrito de intercomunicación llamado GIOP o protocolo de interoperabilidad entre ORB genérico

o más concretamente a través de Internet con el IIOP protocolo de interoperabilidad entre objetos a través de Internet.

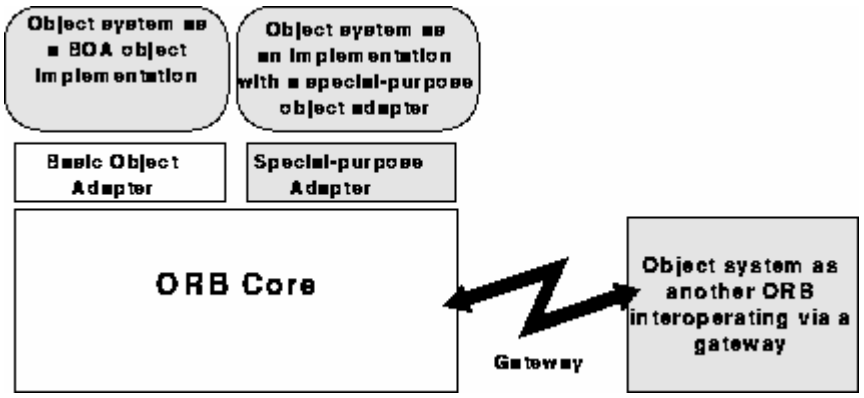


Ilustración 7.10: Diferentes formas de integrar Sistemas de Objetos externos.

Para los sistemas de objetos que simplemente quieren adaptar sus objetos a objetos ORB y recibir invocaciones a través de ORB, una posible forma es que estos sistemas de objetos aparenten ser implementaciones de objetos ORB, comportándose como un ORB ante el resto de sistemas externos pero sin ser un sistema CORBA.

7.1.7 Servicios comunes de CORBA

El **servicio de nombres** de objetos de CORBA, permite a los objetos que están sobre el bus localizar a otros objetos mediante un nombre, también soporta federación de contextos, es decir, agrupación de diferentes servidores para ofrecer un servicio común, de tal forma que si el nombre no se encuentra en un ORB ir a buscar a otro. Este servicio permite a los objetos, ser en lazados con los directorios actuales de red o con los contextos de nombres actuales como ISO X500, OSF Dce, Sun NIS+ y el LDAP par Internet.

El **servicio de comercio** de objetos de CORBA proporciona un servicio de páginas amarillas para los objetos CORBA, de tal forma que se facilitara la búsqueda y localización de objetos determinados en función de palabras claves, contenido dentro de la descripción dentro de un contexto determinado.

El **servicio de ciclo de vida** de objetos de CORBA nos permite crear, copiar, mover y borrar objetos y grupos de objetos relacionados. Para la creación es necesario el uso de objetos fábrica que se usara para crear nuevos objetos de alguna clase determinada.

El **servicio de eventos** de CORBA permite, que los objetos interesados en determinados eventos sean avisados cuando estos ocurran, un evento no es más que un suceso que ocurre a alguno de los objetos del sistema. Para llevar a cabo esto, el objeto registra su interés en un determinado evento y cuando este ocurre, un objeto evento es enviado a este. Los eventos se organizan en modos de objetos canales. Estos sirven para que los objetos se registren, para distribuir los eventos y para filtrarlos cuando sea necesario de tal forma que el objeto consumidor y el objeto creador del evento no necesitan conocerse el uno al otro. Este servicio permite eventos empujados (push) y eventos tirados (pull), en el primer caso, el productor de eventos llama a un método del

canal de eventos el cual reacciona y llama a los objetos registrados como consumidores, en el segundo caso, el consumidor llama a un método del canal de eventos y pregunta a los proveedores de eventos si alguno de estos tiene algún evento.

El **servicio de transacción** de objetos de CORBA es posiblemente el servicio más importante, ya que, es el que nos garantiza la construcción de aplicaciones distribuidas de forma robusta. Permite transacciones simples y opcionalmente anidadas, es un coordinador de dos fases. Las operaciones de transacción son begin, commit y rollback que se definen en el contexto actual de tal forma que, un objeto podrá realizar transacciones simplemente heredando de la clase TransactionalObject. Con las transacciones podremos, desechar el conjunto de operaciones invocadas después de begin si ocurre algún fallo mediante rollback, sin afectar a la integridad del sistema, o por el contrario, si todo sale correctamente podremos acabar la transacción con commit.

El **servicio de control de concurrencia** de CORBA provee las interfaces para bloquear y liberar bloques que permitan a clientes, múltiples coordinar su acceso a recursos compartidos, estos bloqueos pueden ser tanto en el contexto de una transacción como en el contexto de recursos independientes. En el primer caso se produce una liberación cuando se ejecuta el método rollback. Se puede definir un *lockset* que es una colección de bloqueos asociados con un recurso simple. A su vez se define un coordinador de bloqueos que gestiona la liberación de los locksets relacionados.

El **servicio de persistencia** de objetos de CORBA proporciona una interfaz sencilla para almacenar objetos persistentemente en varios servicios de almacenamiento, desde bases de datos orientadas a objetos (ODBMS), a Bases de datos relacionales (RDBMS), pasando por archivos simples, o sistemas estructurados de almacenamiento, usados en los documentos compuestos. El POS puede soportar almacenamientos de un nivel y de dos niveles. En el primero, el cliente no está al tanto de si un objeto está en memoria o en disco. En contraste, en uno de dos niveles separa la memoria del almacenamiento persistente. El objeto debe ser explícitamente colocado de una base de datos a memoria y viceversa.

El **servicio de consultas** de CORBA ayuda a buscar objetos a través de sus atributos. Es similar al servicio de comercio, pero en lugar de localizar servidores, localiza instancias. Las consultas están basadas en los atributos que un objeto hace públicos o accesible a través de sus operaciones. Se han definido dos lenguajes de consulta: el Object Database Management Group (ODMG-93) definió el OQL y el SQL con extensiones de objetos.

El **servicio de colección** de CORBA sirve para estandarizar las interfaces a las colecciones de objetos que se usan normalmente como listas, colas, conjuntos, sacos, árboles. Está basado en la librería de clases de Smalltalk llamada collection.

El **servicio de relación** de CORBA proporciona una forma de crear asociaciones dinámicamente entre objetos que no se conocen, así como mecanismos para recorrer estos enlaces. Se puede usar este mecanismo para garantizar las restricciones de integridad referencial, mantener un seguimiento de la relación de contención o para cualquier tipo de enlaces entre componentes.

El **servicio de externalización** de CORBA define interfaces para exteriorizar un objeto a un flujo e interiorizar un objeto de un flujo. Un flujo es un área que mantiene

datos que pueden estar en memoria, en un fichero de disco, o a través de una red. Se exterioriza un objeto a un flujo para transportarlo a diferentes procesos, maquinas, o ORB. Se interioriza un objeto cuando se requiere regresarlo a la vida en su nuevo destino. Este proceso de dos pasos permite exportar el objeto fuera de su entorno en el ORB. Es equivalente al servicio de serialización de Java.

El **servicio de licencias** de CORBA provee de operaciones para medir el uso de un objeto para posteriormente poder cobrar por él. Este servicio permite cualquier modelo de control de uso en cualquier punto de la vida del objeto como cobro por sesión, por estación, por creación de una instancia o por localización.

El **servicio de propiedades** de objetos de CORBA permite asociar pares de la forma nombre valor a cualquier objeto, el servicio de propiedades no interpreta el valor de las propiedades, son útiles cuando se necesita pegar información a un objeto arbitrario cualquiera par por ejemplo identificar su estado, ponerle un titulo a un objeto, añadirle fecha de creación, etc.

El **servicio de tiempo** del objeto de CORBA proporciona interfaces para sincronizar el tiempo en un sistema distribuido, así como, las operaciones necesarias para gestionar eventos disparados por tiempo.

El **servicio de seguridad** del objeto de CORBA provee de un completo marco de seguridad para objetos distribuidos permitiendo: Autentificación, control de acceso, confidencialidad y auditorías.

El **servicio de gestión de cambio de versión** de objetos de CORBA la idea es permitir una evolución de los sistemas de forma segura, es decir poder seguir la evolución de las diferentes versiones que un objeto va tomando, pero que a su vez los objetos que usaban versiones antiguas de ese objeto no se vean afectados.

7.1.8 Facilidades comunes de CORBA

Las facilidades comunes de CORBA son colecciones de jerarquías de clases bien definidas que proveen servicios directamente útiles a los objetos de negocios. Las dos categorías de facilidades comunes que son la horizontal y vertical definen las reglas de juego que son necesitadas por los componentes de negocios, para colaborar efectivamente. Las facilidades horizontales realizan la gestión de diversos aspectos, comunes a un amplio campo de objetos diversos, mientras que las verticales proveen IDL definidas y marcos de trabajo para objetos concretos, de un segmento del mercado específico como salud o ventas.

7.1.8.1 Las facilidades horizontales de CORBA

La facilidad de Interfaz de usuario trabaja con la tecnología de documentos compuestos y con servicios de edición apropiados similares a los que provee OpenDoc y OLE. La idea es proveer un marco de trabajo para compartir y subdividir la visualización de una ventana de tal forma que diferentes componentes puedan compartir el estado visual real de la pantalla de forma integra.

Un componente debe ser capaz de contener otros componentes dentro de él y también debe ser capaz de editar sus contenidos en el lugar sin abandonar esa ventana.

Esta facilidad trabaja con programación. Los lenguajes de programación usan mecanismos de enlace retardado lo que permite crear y unir programas en tiempo de ejecución a componentes en documentos compuestos. De igual forma usando un lenguaje de programación es posible invocar un servicio de CORBA IDL específico.

La facilidad de manejo de información incluye almacenamiento de documentos compuestos y facilidades para el intercambio de información. También define estándares que los componentes usan para codificar y representar sus datos, para definir e intercambiar metadatos, y para modelar información.

Los metadatos son particularmente importantes en un entorno de componentes distribuidos ya que son usados para describir servicios e información. La información en metadatos, puede ser un suplemento al repositorio de interfaces, describiendo la estructura y procedimientos de acceso a los datos. Ayuda a los componentes a buscar información efectivamente, y a descubrir como son recuperados sus contenidos.

La facilidad de gestión del sistema incluye interfaces y servicios para la gestión, instrumentación, configuración, instalación, operación y reparación de componentes de objetos distribuidos. La jerarquía de clases de gestión del sistema define las siguientes interfaces:

1. *Instrumentation* permite recoger información acerca de la sobrecarga, responsabilidad y consumo de recursos de un componente.
2. *Data collection* reúne información acerca de eventos históricos relacionados con una componente.
3. *Quality of service* selecciona el nivel de servicio que un componente provee en áreas como disponibilidad y rendimiento.
4. *Security* permite gestionar la seguridad del sistema.
5. *Event management* genera, registra, filtra y envía notificaciones de eventos a las aplicaciones en gestión.
6. *Scheduling* programa tareas repetitivas y asocia gestores de eventos con eventos.
7. *Instance tracking* asocian objetos con otros objetos gestionados que están sujetos a políticas comunes.

La facilidad de gestión de tareas provee un esquema de trabajo para la gestión de flujos de trabajo, grandes transacciones, agentes, programación, reglas y automatización de tareas. Incluye una facilidad semántica de mensajes usada para comunicar peticiones orientadas a tareas.

Los agentes consisten en objetos de información y un programa asociado que sabe que hacer con la información y como trabajar con el entorno, existen agentes estáticos y móviles. La infraestructura de agentes de CORBA provee capacidades para que los agentes publiquen sus servicios y para que los subscriptores se registren a estos servicios.

La gestión de reglas provee facilidades para especificar reglas evento/condición/acción en un lenguaje declarativo y para manejar y ejecutar estas reglas. OMG provee un lenguaje de especificación de reglas basadas en IDL.

La automatización es la tecnología usada por sistemas de programación o cualquier cliente que necesite, dinámicamente, acceder a las funciones de los componentes o datos contenidos. Se puede usar la automatización para gestionar objetos visibles por el usuario, que son más grandemente granulados que los objetos típicos de un ORB. La automatización requiere enlaces retardados, el cual puede ser proporcionado usando la invocación de métodos dinámicos de CORBA y la interfaz de esqueletos dinámica.

7.1.8.2 Las facilidades verticales de CORBA

Las facilidades verticales proveerán interfaces IDL definidas estándares para la interoperabilidad de componentes para segmentos del mercado tales como salud, ventas y finanzas. Algunos de los marcos de trabajo que están en desarrollo son los siguientes.

- Proceso de imágenes establece interoperabilidad entre objetos imagen, información relacionada con la imagen y servicios de proceso de imágenes para aplicaciones.
- *Supercarreteras de información* usa a CORBA como un medio para un gran flujo de información.
- Fabricación integrada proporciona el marco necesario para realizar aplicaciones distribuidas que gestionen el proceso de fabricación de una empresa.
- Simulación distribuida, permite la interoperabilidad entre diferentes objetos de simulación en un entorno virtual.
- Gestión de la exploración de gas y petróleo.
- Proceso de contabilidad distribuido que incluye transacciones de negocios, cambio de monedas, ventas, etc.
- Marcos para las aplicaciones de desarrollo de informático.

7.1.9 Construción de un programa CORBA

Antes de comenzar veamos unos conceptos básicos sobre IDL, básicamente este lenguaje usa casi la misma sintaxis que C de tal forma que va a resultar muy sencillo comprenderlo. Las interfaces definidas en IDL se dividen en módulos, que a su vez están compuestos por, interfaces, y dentro de cada interfaz, se definen las operaciones y los atributos de la interfaz así un claro ejemplo seria:

```
module Contador{
   interface Cuenta{
   attribute long sum;
   long incrementa();
   }
 }
```

En el ejemplo anterior hemos definido un modulo `Contador` que dispone de una interfaz llamada `Cuenta` donde hay un atributo sum de tipo long y un método incrementa que no recibe ningún parámetro de entrada y devuelve un long. Básicamente los módulos representan a los paquetes en y las interfaces representan a los objetos en orientación a objetos.

CORBA nos permite llamar a operaciones que se conocen en tiempo de compilación, y que definen su interfaz, mediante IDL y también nos permite llamar a operaciones, que no se conocen en tiempo de compilación, pero si en tiempo de ejecución mediante DII, veamos la diferencia práctica entre ambas:

En ambos casos se hace una petición a la referencia de un objeto a través de su ID y se invoca el método que deseamos. El servidor no diferencia entre las invocaciones estáticas y dinámica.

La implementación del objeto, su adaptador y el ORB usados son independientes y totalmente transparentes a clientes tanto dinámicos como estáticos.

La interfaz estática se genera directamente en forma de stubs por el IDL preprocesador. Esto es perfecto para los programas que saben en tiempo de compilación las características de las operaciones que necesita invocar. Las interfaces estáticas se enlazan en tiempo de compilación y tienen las siguientes ventajas respecto a las dinámicas:

- Son más fáciles de programas.
- Proporcionan un sistema de comprobación de tipos mejor.
- Tienen mejor rendimiento.
- Están autodocumentadas, es decir puedes saber lo que hacen leyendo el código.

7.1.9.1 Pasos para crear una aplicación CORBA con invocación estática

1. Definir los objetos mediante la IDL.
2. Compilar los archivos IDL mediante un precompilador a un determinado lenguaje
3. Añadir el código de implementación de nuestro resultado al resultado de la compilación que son dependientes del lenguaje de compilación.
4. Compilar el código anterior.
5. Guardar las definiciones de clases en el Almacén de Interfaces, para que los programas puedan acceder a esta información en tiempo de ejecución.
6. Guardar el código ejecutable del objeto dentro del Almacén de Implementaciones.
7. Instanciar los objetos del servidor, en tiempo de ejecución el adaptador de objetos servidores puede instanciar a los objetos que tienen los métodos que el cliente está invocando en cada momento.

A continuación vamos a ver un ejemplo de un programa CORBA en Java que realiza invocación estática, este ejemplo se basa en el IDL comentado anteriormente. Primero realizamos la compilación de IDL a Java esta compilación nos va a generara un paquete llamado Contador y dentro de este varias clases y una interfaz, pasamos a comentarlas:

Por el lado del cliente tenemos:

1. `_st_Cuenta.class`, esta clase implementa el stub del cliente.
2. `Cuenta_var.class`, esta clase proporciona el método bind que se usa para localizar a los objetos de esta clase. En este ejemplo se usa para encontrar objetos remotos que implementan la interfaz Cuenta.

Por el lado del servidor tenemos:

1. `_sk_Cuenta.class`, esta clase implementa el skeleton del servidor.
2. `Cuenta.class`, este archivo define una interfaz, debemos implementar esta interfaz.
3. `_example_Cuenta`, muestra un ejemplo para la implementación del objeto Cuenta.

Comencemos por el servidor, cada uno debe tener una función principal que inicialice el entorno ORB y ejecute los objetos, un posible ejemplo sería:

```
public class Servidor {
  public static void main(String[] args) {
    // Inicializamos el ORB.
    CORBA.ORB orb = CORBA.ORB.init();
    // Inicializamos el BOA.
    CORBA.BOA boa = orb.BOA_init();
    // Creamos el objeto servidor cuenta.
    Contador.Cuenta cuenta =  new CuentaImpl("Mi cuenta");
    // Exportamos el nuevo objeto al ORB.
    boa.obj_is_ready(cuenta);
    // Esperamos las peticiones de los clientes.
    boa.impl_is_ready();
  }
}
```

Primero creamos un objeto ORB con `CORBA.ORB.init()` y a partir de este tenemos acceso a todas las operaciones del ORB, luego creamos un adaptador básico para este ORB con `orb.BOA_init()` ahora ya disponemos de un objeto adaptador del que podemos obtener toda su funcionalidad tras esto creamos la implementación del objeto que queremos poner de servidor en este caso pertenece a la clase Cuenta del modulo Contador, lo llamaremos cuenta tras esto le indicamos al BOA que nuestro objeto cuenta esta listo mediante `boa.obj_is_readi(cuenta)` y que espere para llamadas de los clientes mediante `boa.impl_is_ready()`.

Hemos de comentar que para crear el objeto cuenta que pertenece a la clase `Contador.Cuenta` hemos llamado al constructor de la clase `CuentaImpl` que deriva de la anterior, como comentemos en toda la introducción anterior hemos de diferencias entre el objeto `Cuenta` y el objeto implementación de `Cuenta` llamado `CuentaImpl`. La principal diferencia es que `Cuenta` deriva directamente de `CORBA.Object` que es la clase raíz de CORBA mientras que `CuentaImpl` deriva de la clase `_sk_Cuenta` e implementa la interfaz `Cuenta`. `_sk_Cuenta` deriva de `CORBA.portable.Skeleton` que a su vez deriva de `CORBA.Object` con lo que, gracias al polimorfismo, se puede realizar la asignación anterior. La idea general es que la clase que deseamos poner como servidora es Cuenta, pero esta se nos genera automáticamente a través del precompilador de IDL con lo que debemos utilizar otra clase para crear la implementación de esta clase que en este caso es `CuentaImpl` que deriva del skeleton que ha generado el precompilador y que debemos de construir nosotros, veamos un ejemplo sencillo:

```
    class CuentaImpl extends Contador._sk_Cuenta implements
Contador.Cuenta
   {
```

```
     private int sum;
     //Constructor
    CuentaImpl(String nombre)
    {
       super(nombre);
       sum=0;
    }
    //get sum
    public int sum() throws CORBa.SystemException
    { return sum; }
    //set sum
    public void sum(int val) throws CORBA.SystemException
    { sum=val; }
    // método incrementa
    public int incrementa() throws CORBA.SystemException
    { sum++; return sum; }
   }
```

El constructor crea un objeto con un nombre para poder hacer referencia a el desde los clientes el resto de métodos son la implementación de la interfaz Cuenta que viene de la definición hecha en el archivo IDL.

Vemos que con estas dos clases por un lado tenemos un servidor y por el otro la implementación de nuestro objeto, veamos ahora como se realizaría un cliente.

Primero inicializaremos el ORB como en el caso del servidor, y una vez hecho esto crearemos una instancia de nuestro objeto a través del método `bind()` de la clase `Cuenta_var`. Veamos un ejemplo sencillo:

```
public class Cliente {
  public static void main(String[] args) {
    // Inicializamos el ORB.
    CORBA.ORB orb = CORBA.ORB.init();
    // Localizamos un objeto Cuenta
    // vemos que el nombre que le pasamos al metodo bind en
este caso “Mi cuenta”
    // debe coincidir con el que le dimos en el servidor
    Contador.Cuenta cuenta= Contador.Cuenta_var.bind(orb,
"Mi cuenta");
    // A partir de ahora ya podemos usar el objeto cuenta
como otro mas
    // dentro de nuestro programa en Java.
    //Ponemos el valor de sum a 0
    cuenta.sum(0);
    // Llamamos al metodo incrementa
    cuenta.incrementa();
  }
}
```

En un cliente básicamente lo importante es localizar a nuestro objeto dentro del ORB a través de su nombre, que debe coincidir con el que le dimos al crearlo en el servidor, esto se consigue con el método bind de la clase `Cuenta_var`, una vez obtenido este objeto y apodemos manipularlo igual que si fuese un objeto normal, pero

ahora este objeto puede encontrarse en otra maquina distinta localizada en otro sitio diferente de una forma copletamente transparente para nosotros.

7.2 DCOM

La primera encarnación de OLE, Objetos incrustados y enlazados, fue un mecanismo para crear y manipular documentos compuestos. Para sus usuarios, un documento compuesto aparece como un conjunto simple de información, pero de hecho, contiene elementos creados por dos o más aplicaciones diferentes. Con OLE 1, por ejemplo, un usuario podía combinar una hoja de cálculo creada usando Microsoft Excel con un documento de texto creado con Microsoft Word, como se muestra en la Ilustración 7.11. La idea fue darle al usuario una visión centrada en los documentos de la informática, para permitirles pensar más en la información y menos en las aplicaciones que usaron para trabajar con ella. Como sugiere el nombre, los documentos compuestos podían ser creados enlazando dos documentos juntos o incrustando completamente un documento en otro.

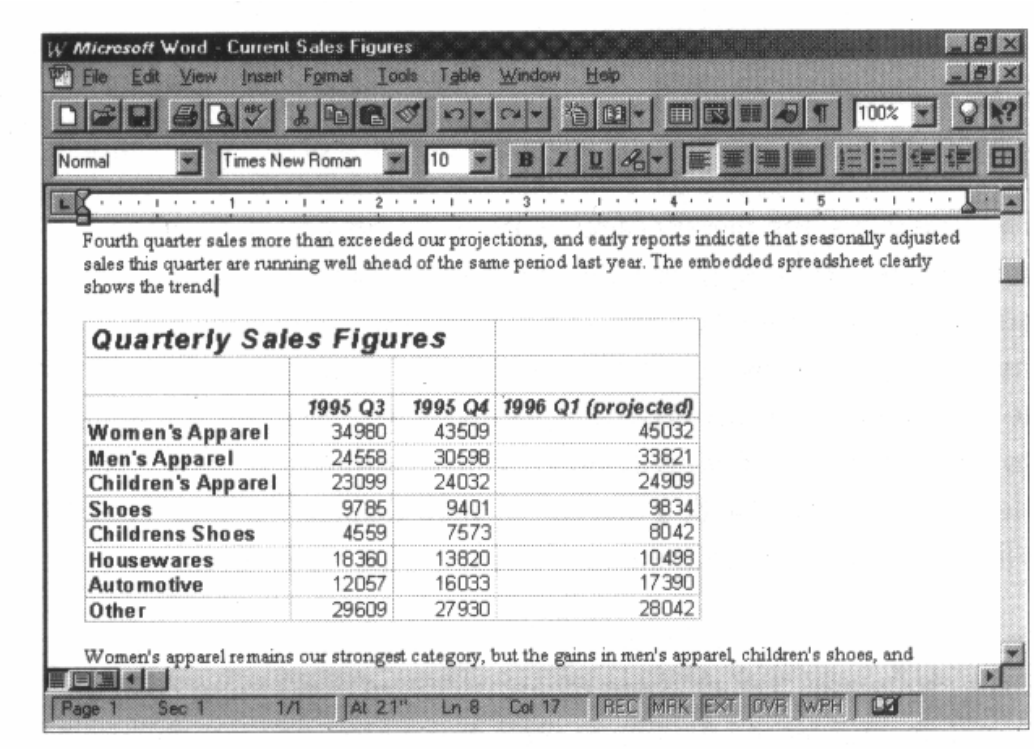


Ilustración 7.11: Documento compuesto en COM

Como muchas versiones primeras, OLE 1, no fue perfecta. Los arquitectos de la siguiente versión se propusieron mejorar el diseño original. Pronto se dieron cuenta de que el problema de los documentos compuestos era un caso especial de un problema más general: ¿Cómo deben proporcionarse varios componentes software servicios entre sí? Para solucionar este gran problema, los arquitectos de OLE crearon un conjunto de técnicas que fuesen aplicables no solo a documentos compuestos. La tecnología principal de entre todas las que surgieron fue COM, que sirve como base para OLE 2. Esta nueva versión de OLE permitió documentos compuestos incluso mejor que la

primera versión, pero muchas más cosas surgieron de aquí, además de la posibilidad de combinar documentos creados con diferentes aplicaciones. OLE 2 ofrecía el potencial necesario para crear una nueva forma de pensar en como los programas de todo tipo deberían interactuar.

Este potencial fue en gran medida resultado de COM, este establece un paradigma común para la interacción entre todo tipo de programas, librerías, aplicaciones, software de sistema y más. En consecuencia virtualmente cualquier tipo de tecnología software puede ser implementada usando esta aproximación definida por COM, y hacerlo así ofrece algunos beneficios muy tangibles.

Como consecuencia de estos beneficios, COM pronto se convirtió en una tecnología a parte que no tenía nada que ver con documentos compuestos. Microsoft, sin embargo, quería seguir teniendo un nombre común para referirse a todas las tecnologías basadas en COM como un grupo. La empresa decidió reducir el nombre de Objetos enlazados e Incrustados a OLE- estas letras no fueron tratadas nunca más como un acrónimo - y eliminar el número de versión [CHA1996].

Bajo esta nueva idea, él término OLE fue aplicado a cualquier cosa que se construyese usando COM (aunque COM fue usado en productos que no tienen nada que ver con OLE). OLE ya no significaba solo documentos compuestos sino ahora esta etiqueta se asignaba a cualquier cosa basada en la tecnología COM. De algún modo, agrupar bajo un solo nombre todo el software escrito usando COM no tenía más sentido que, decir, agrupemos junto todo el software escrito en C++. Ambos COM y un lenguaje de programación como C++ son herramientas generales que pueden ser usadas para crear todo tipo de software. Aun ambos por razones históricas y para indicar el origen de esta nueva y inalcanzable tecnología, el término OLE fue usado para identificar muchas (pero no todas) tecnologías basadas en COM. A principios de 1996, Microsoft sacó otro término en disputa: ActiveX. En su primera apariencia, este nuevo término fue asociado con tecnologías relacionadas con Internet y con aplicaciones que crecían con Internet, como la WWW. Debido a que la mayoría del esfuerzo realizado por Microsoft en esta área estaba basada en COM, ActiveX fue directamente conectada con OLE. Pronto, sin embargo, este nuevo término comenzó a usurpar más y más el territorio tradicional de OLE y hoy en día las cosas han dado la vuelta. Ahora el término OLE una vez más se refiere solo a la tecnología usada para los documentos compuestos. Los diversos conjuntos de tecnologías construidas usando COM estaban agrupadas con la etiqueta OLE, ahora están agrupadas con el cartel de ActiveX. En varios casos, las tecnologías que tenían en su nombre OLE han sido rebautizadas como tecnologías ActiveX. Nuevas tecnologías basadas en OLE que una vez pudieron tener la etiqueta OLE ahora son denominadas frecuentemente en su lugar como ActiveX.

¿Es este el final de la historia de los nombres de las tecnologías basadas en COM? Mirando la historia de lejos la respuesta es posiblemente no. Lo que el departamento de marketing de Microsoft pueda inventar no lo puede adivinar nadie. Pero a pesar de esta aventura en la nomenclatura, lo realmente importante no ha cambiado. Lo realmente importante es COM.

7.2.1.1 Entendiendo COM

Todas las tecnologías OLE y todas las ActiveX descritas en este capítulo están basadas en los fundamentos proporcionados por COM. Así que ¿Qué es COM? Para

contestar a esta pregunta, pensemos primero en otra: ¿Cómo debería un trozo de código acceder a los servicios proporcionados por otro trozo de código? Hoy, como se muestra en la Ilustración 7.12, la contestación de pende del trozo de software. Una aplicación podría, por ejemplo, enlazarse con una librería y por lo tanto acceder a los servicios de esta llamando a las sus funciones. O una aplicación podría usar los servicios proporcionados por otra que corre en un proceso completamente separado. En este caso, los dos procesos locales se comunican usando un mecanismo de intercomunicación de procesos, que generalmente requiere definir un protocolo entre las dos aplicaciones (un conjunto de mensajes que permiten a las aplicaciones especificar las peticiones a una y a la otra responder apropiadamente). Un tercer ejemplo es una aplicación que podría usar los servicios proporcionados por el sistema operativo. Aquí la aplicación generalmente hace llamadas al sistema, las cuales son gestionadas por el sistema operativo. O finalmente, una aplicación podría necesitar los servicios de software que se esta ejecutando en una máquinas completamente diferentes, accesibles vía red. Muchas aproximaciones diferentes pueden ser usadas para acceder a estos servicios, tales como intercambio de mensajes con la aplicación remota o realizando llamadas a procedimientos remotos.

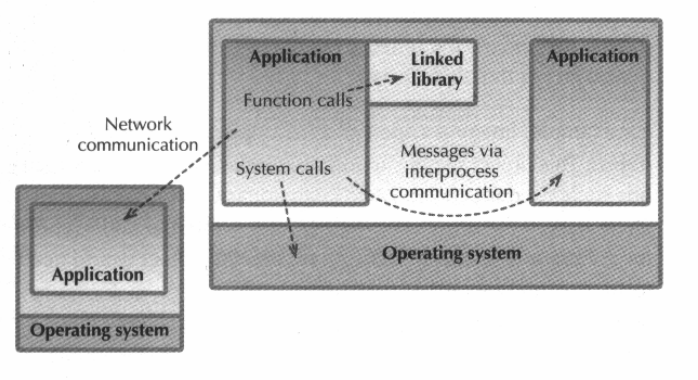


Ilustración 7.12: Tecnología COM

La necesidad fundamental en todas estas relaciones es la misma: un trozo de software debe acceder a servicios proporcionados por otro. Pero el mecanismo para conseguir que esto se diferencia en la forma de hacerlos en cada caso - llamadas a funciones locales, paso de mensajes vía intercomunicación de procesos, llamadas al sistema o algún tipo de comunicación a través de red. ¿Por que todo esto? ¿No debería ser más sencillo definir una forma común de acceso a todos los tipos de servicios sin tener en cuenta como nos son proporcionados? Esto es exactamente lo que COM hace. Define una aproximación estándar mediante la cual trozos de software proporcionan servicios a otros, una aproximación que funciona en todos los casos descritos anteriormente. Aplicando esta arquitectura de servicio común independientemente de librerías, aplicaciones, software de sistema y redes, COM transforma la forma en la que se construye el software.

7.2.1.2 Como funciona COM

Con COM, cada trozo de software implementa sus servicios como uno o más objetos COM. Cada objeto COM proporciona una o más interfaces, cada una de las cuales incluye una cantidad de métodos. Un método es una función o un procedimiento que realiza una acción específica y que puede ser llamada por los programas usando el

objeto COM (el cliente es el que realiza esa llamada). Los métodos que tiene cada interfaz se relacionan entre sí de alguna forma. Los clientes pueden acceder a los servicios proporcionados por un objeto COM solo invocando el método en la interfaz del objeto, no pueden acceder directamente a ninguno de los datos del objeto. Por ejemplo, supongamos un corrector ortográfico que esta implementado como un objeto COM. Este objeto podría tener una interfaz con métodos tales como MiraUnaPalabra, AñadaAlDiccionario y BorraDelDiccionario. Si el programador de objetos posteriormente quiere añadir soporte para tesauros a este mismo objeto COM, este necesitaría otra interfaz (quizá con un solo método como DevuelveSinonimo). Los métodos de cada interfaz proporcionan servicios relacionados ya sea comprobación ortográfica o acceso a tesauros.

O imagine un objeto COM representando su cuenta bancaria. Podría soportar una interfaz a la que usted accede directamente con métodos tales como Ingreso, Reembolsos y ComprobaciónDeBalances. Este mismo objeto podría soportar una segunda interfaz conteniendo métodos tales como CambiarNumeroDeCuenta y CierraCuenta que pueden ser invocados solo por empleados bancarios. De nuevo cada interfaz contiene métodos que se relacionan entre sí. La Ilustración 7.12 ilustra un objeto COM. La mayoría de los objetos COM proporcionan más de una interfaz, y el objeto de la Ilustración 7.13 no es una excepción: tiene tres interfaces, cada una representada por un pequeño círculo pegado al objeto. El objeto esta siempre implementado dentro de un servidor, que se muestra como un rectángulo alrededor del objeto. Este servidor puede ser una librería DLL que se carga cuando se necesita o un proceso separado.

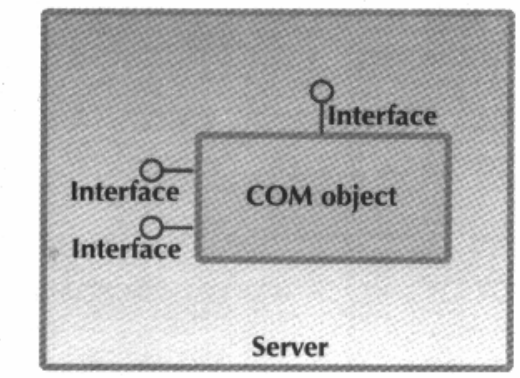


Ilustración 7.13: Interfaces COM

La Ilustración 7.14 muestra un detalle de una interfaz soportada por este objeto COM. Esta interfaz permite el acceso al comprobador ortográfico y contiene los tres métodos comentados anteriormente. Si otra de las interfaces del objeto permitiera acceso a los servicios del tesauro descrito anteriormente, una muestra de él contendría solo el método DevuelveSinonimo. (De hecho este diagrama esta un poco simplificado, todas los interfaces suelen incluir unos pocos métodos estándares más, que no se muestran aquí.)

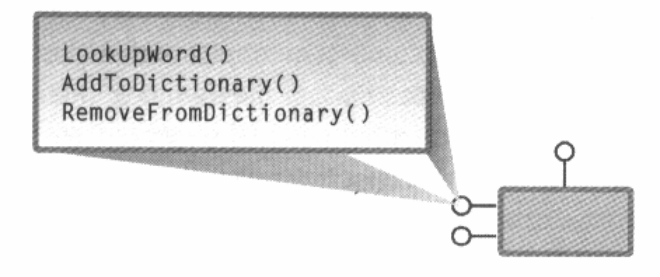


Ilustración 7.14: Métodos de una interfaz COM

Para invocar a un método dentro de la interfaz de un objeto COM, un cliente debe conseguir un puntero a esa interfaz. Un objeto COM típicamente proporciona sus servicios a través de varias interfaces cuyos métodos se plantea invocar. Por ejemplo, un cliente de nuestro objeto ejemplo necesitaría un puntero a la interfaz y otro puntero para invocar el método en la interfaz de tesauro del objeto. La Ilustración 7.15 muestra un cliente con punteros a las dos interfaces en un mismo objeto COM.

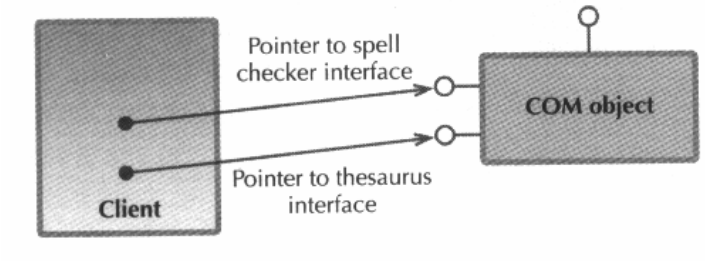


Ilustración 7.15: Cliente COM

Cada objeto COM es una instancia de una clase específica. Una clase, por ejemplo, podría contener objetos que proporcionan comprobación ortográfica y servicios de tesauros, mientras otro podría contener objetos representando cuentas bancarias. Normalmente, debes conocer la clase del objeto del que quieres ejecutar una instancia, para hacer esto se usa la librería COM. Esta librería esta presente en todos los sistemas que soportan COM y accede a un directorio de todas las clases de objetos COM del sistema, más concretamente el registro del sistema. Un cliente puede, por ejemplo, llamar a una función en la librería COM especificando la clase del objeto COM que quiere y la primera interfaz proporcionada por él a la que se quiere apuntar (la librería COM proporciona esos servicios como llamadas ordinarias, no a través de métodos en interfaces COM). La librería COM entonces provoca que un servidor que implementa un objeto de esa clase comience a funcionar. La librería devuelve al el cliente un puntero a la interfaz requerida en el objeto recién instanciado. El cliente puede entonces preguntar al objeto directamente por punteros a otras interfaces que el objeto soporta.

Una vez que un cliente tiene un puntero a una interfaz en un objeto puede comenzar a usar los servicios que el objeto implementa llamando los métodos de la interfaz. Para un programador, invocar un método parece como invocar un procedimiento o función local. De hecho, sin embargo, el código que se ejecuta podría estar ejecutándose en una librería o en un proceso separado o como parte del

sistema operativo o incluso en otro sistema separado. Con COM, el cliente no necesita preocuparse de estos detalles - todo se accede de la misma forma. Como se muestra en la Ilustración 7.16 un modelo común es usado para acceder a los servicios proporcionados por todo tipo de programas.

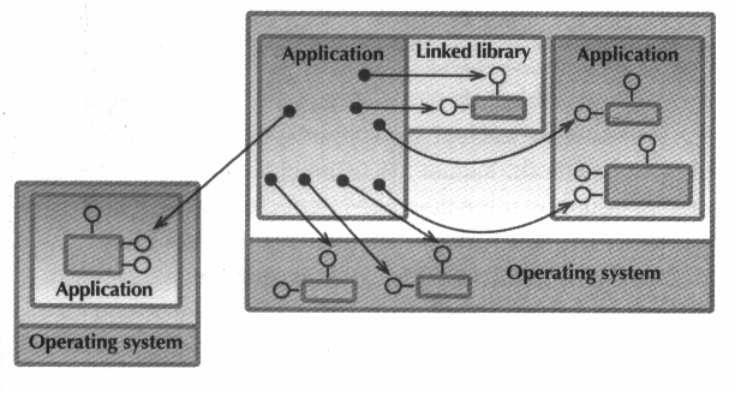


Ilustración 7.16: Modelo común para COM

7.2.1.3 Orientación a Objetos y COM

En COM los objetos son la idea central. Pero la forma en que COM define y usa los objetos difiere de la forma en la que los objetos son usados por otras tecnologías orientadas a objetos. Para comprender como COM se relaciona con otras tecnologías orientadas a objetos, es útil describir que significa el término orientado a objetos y después ver como COM encaja en él.

Definición de objeto, el término objeto ha sido enturbiado por el marketing intentando asignarle a este sus últimos caprichos, pero en la mente de la mayoría, las tecnologías orientadas a objetos están formadas por una serie de características claves. La principal de todas es la noción de lo que es un objeto. Hay un amplio acuerdo en que un objeto consiste en dos elementos: un conjunto de datos (también llamado, estado o atributos) y un conjunto de métodos. Esos métodos, comúnmente implementados como procedimientos o funciones, permiten al cliente del objeto pedir a este que realice varias tareas. La Ilustración 7.17 muestra un diagrama de un objeto.

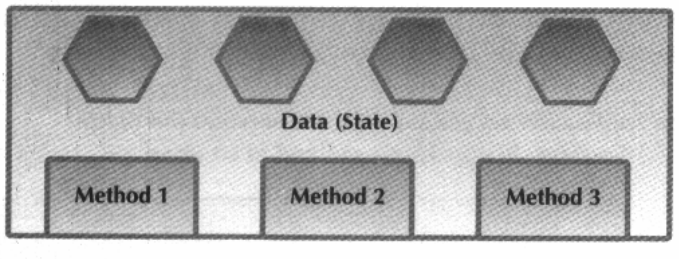


Ilustración 7.17: Objetos y métodos

Hasta aquí todo correcto, los objeto COM son exactamente como estos. En la mayoría de las tecnologías orientadas a objetos, cada objeto solo puede tener una interfaz con un conjunto de métodos. Por contra, los objetos COM pueden, y casi siempre lo hacen, soportan más de una interfaz. Un objeto en C++, por ejemplo, tiene

solo una interfaz que incluye todos los métodos del objeto. Un objeto COM, con sus múltiples interfaces, podría perfectamente ser implementado usando varios objetos en C++, uno para cada interfaz que el objeto necesite (aunque C++ no es el único lenguaje que puede ser construido usando objetos COM). Otra idea familiar en la tecnología de objetos es la noción de clase. Todos los objetos representan cuentas bancarias, por ejemplo, podrían ser de la misma clase. Algún objeto de una cuenta bancaria determinada, como la que representa tu cuenta, es una instancia de esta clase.

Los objetos COM, también tienen clases, como ya se describió. En COM, una clase identifica una implementación específica de un conjunto de interfaces. Varias implementaciones diferentes del mismo conjunto de interfaces pueden existir, cada una de las cuales es una clase diferente. Desde el punto de vista del cliente, lo que le interesan son las interfaces. Como se implementan, que es lo que la clase realmente indica, no le interesa al cliente. Esta capacidad de trabajar igual con diferentes tipos de objetos, cada uno soportando la misma interfaz pero implementando la de forma distinta, se llama polimorfismo. Se describe un poco más en la próxima sección.

Encapsulación, polimorfismo y herencia si una tecnología modela las cosas como un grupo de métodos y datos organizados en grupos dentro de clases, ¿Es suficiente para clasificarla de orientada a objetos? Aunque esta cuestión está llena de debate, la contestación de la mayoría de los profesionales será no. En general, la orientación a objetos requiere soportar tres características más: encapsulación, polimorfismo y herencia.

Encapsulación significa que los datos de un objeto no están directamente accesibles a los clientes del objeto. En su lugar, los datos se encapsulan, se ocultan del acceso directo. La única forma de acceder a los datos del objeto es usando los métodos de este. Estos métodos muestran un interfaz bien definido al mundo exterior, y solo a través de esta interfaz el usuario del objeto puede leer o modificar sus datos. La encapsulación protege los datos del objeto de un acceso inapropiado y permite controlar al objeto mismo de como los datos son accedidos. Para prevenir que cambios involuntarios e incorrectos sean realizados accediendo directamente a los datos del objeto, la encapsulación puede ayudar enormemente para la creación de un mejor software.

C++ proporciona soporte directo para la encapsulación. Si un programador intenta modificar los datos de un objeto directamente, el compilador puede indicar el intento como un error. Aunque COM no es un lenguaje de programación, la misma idea se mantiene. Un cliente puede acceder a los datos de un objeto COM solo a través de los métodos de la interfaz del objeto. Los datos de los objetos COM están encapsulados.

La segunda característica que define la tecnología orientada a objetos es polimorfismo. Significa que un cliente puede tratar diferentes objetos como si fuesen los mismos y cada uno se comportara de forma adecuada. Por ejemplo, consideremos un objeto que represente tu cuenta corriente. Este objeto probablemente tiene un método Reintegro, que llamarás cada vez que escribes un cheque. Podrías tener un objeto que represente tu cartilla de ahorros y un método Reintegro. Para un cliente estos dos métodos aparentan el mismo y cuando cada uno es invocado, las mismas cosas ocurren: el balance del objeto se reduce.

Sin embargo la implementación de estos dos métodos podría ser muy diferente. La implementación en la cartilla de ahorros simplemente comprueba que la cantidad pedida es menor que el dinero que tienes en la cuenta, en tal caso la petición se realiza correctamente sino esa petición falla. En el objeto cuenta corriente generalmente ofrece una cantidad de crédito con lo que esto tendría que tenerse en cuenta.

Para un cliente, estos dos métodos de Reintegro parecen el mismo, la diferencia es su implementación. Esta habilidad de tratar a diferentes cosas como si fuesen las mismas, es la esencia del polimorfismo. Este ejemplo demuestra el beneficio del polimorfismo: los clientes pueden recordar sin preocuparse las diferencias que no le interesan con lo que simplifica el desarrollo de software al cliente.

Los objetos COM soportan completamente esta idea. Es perfectamente posible que dos objetos de diferente clase presente la misma interfaz o quizás solo un método común a sus clientes, aunque cada objeto los métodos relevantes diferentes. La característica final de las tecnologías orientadas a objetos es la herencia. La idea es simple: cuando un objeto, puede crear nuevos objetos que automáticamente incluye alguna o todas las características de un objeto ya existente. De la misma forma que una persona podría sin ningún esfuerzo por su parte heredar la calvicie de su padre un objeto puede automáticamente heredar las características de otro.

Hay varios tipos de herencia. Una distinción que nos interesa aquí es herencia de implementación y herencia de interfaces. Con **herencia de implementación** un objeto hereda el código de sus padres. Cuando un cliente del hijo de otro objeto llama un método que ha heredado el hijo, el código del padre es el que se ejecuta. Con la *herencia de interfaces*, el hijo solo hereda la definición de los métodos. Cuando un cliente del objeto hijo llama a uno de estos métodos, el hijo debe proporcionar el código para manejar la petición ya que este no ha sido heredado solo se ha heredado la implementación.

La herencia de implementación es un mecanismo que nos permite reusar el código, es ampliamente usada en lenguajes como C++ y SmallTalk. La herencia de interfaces por contra, tiene que ver con la reutilización de especificaciones, la definición de métodos que un objeto soporta. Una razón importante para usar herencia de interfaces es que facilita la implementación del polimorfismo. Definir una interfaz heredando de un objeto ya existente garantiza que un objeto que soporta la nueva interfaz puede ser tratado como un objeto que soporta la antigua.

Los lenguajes de programación como C++ soportan ambos métodos de herencia. Los objetos COM sin embargo, solo soportan la herencia de interfaces. Los creadores de COM consideraron que proporcionar a COM la herencia de implementación era una forma inapropiada y potencialmente peligrosa para que un objeto reuse otro. Por ejemplo, debido a que la herencia de implementación a menudo expone detalles de implementación de los padres, esto puede romper la encapsulación de los padres. Con herencia de interfaces como hace COM, permite el reuso de una parte clave de otro objeto como es su interfaz mientras que evita este problema.

Pero sin implementar herencia ¿Cómo puede un objeto reusar el código de otro? En COM esto se hace mediante dos mecanismos incrustación y agregación. Con incrustación un objeto simplemente llama a otro como ayuda para implementar sus funciones. Con agregación un objeto presenta una o más de las interfaces de otro como

si fuesen suyas, lo que un cliente ve como un objeto que proporciona un grupo de interfaces es de echo uno o más objetos agregados juntos. Como puede imaginarse, la agregación es un poco más difícil de implementar que la incrustación pero ambos proporcionan una forma eficiente de construir a partir de objetos COM ya existentes.

Pero ¿Es COM orientado a objetos? Hemos visto que las nociones de una colección de datos y métodos juntos así como las características de encapsulación, polimorfismo y herencia de interfaces se cumplen en COM pero la herencia de implementación no la cumple y realiza el reuso a través de incrustación y agregación, además COM permite múltiples interfaces cosa que no se contempla como una característica de las tecnologías orientadas a objetos. Por lo que debemos contestar a esta pregunta que COM no es realmente orientada a objetos. Pero si hacemos la pregunta ¿Proporciona COM todas las características claves y beneficios de los objetos? La respuesta es si y lo que realmente importa es esta segunda pregunta. El objetivo es conseguir escribir mejor software y esto COM lo consigue.

7.2.1.4 Componentes Software y COM

En los últimos 35 años los ingenieros de hardware han ido de construir computadoras del tamaño de una habitación a crear ligeros portátiles basados en pequeños y potentes microprocesadores. En los mismos 35 años, el desarrollo de programas ha ido de construir grandes sistemas en ensamblador y COBOL a construir sistemas incluso más grandes en C y C++. Vemos que el mundo del software no avanza a la misma velocidad que el mundo hardware. ¿Qué tienen los diseñadores de hardware que los programadores no tienen?

La contestación son los componentes. Si los ingenieros de hardware tuviesen que comenzar desde la arena cada vez que ellos quieren construir un nuevo dispositivo, si su primer paso fuese siempre extraer el silicio para hacer un chip, ellos no progresarían tan rápido. Pero, por supuesto, eso no es lo que hacen. En su lugar, un diseñador de hardware generalmente construye un sistema a partir de componentes, cada uno de los cuales realiza una función determinada y un conjunto de servicios a través de unas interfaces bien definidas. Los diseñadores de hardware pueden simplificar sus trabajos reusando el trabajo de otros.

Reusar es también un camino para crear software mejor. Los programadores de hoy en día a menudo comienzan con algo que no va muy lejos de la arena y después proceden a repetir los pasos de cientos de programadores anteriores a ellos. El resultado es a menudo muy bueno, pero podría ser mejor. Creando aplicaciones nuevas a partir de los componentes ya comprobados que existen es posiblemente producir código de mayor fiabilidad. Y más importante aun, se podría hacer más rápido y barato.

Esta idea de definir partes reusables, cada una ofreciendo unos servicios a través de una interfaz bien definida es exactamente lo que COM nos ofrece. Los objetos COM nos proporciona un mecanismo eficiente para reusar software permitiendo la creación de componentes reusables y discretos. Estos componentes pueden actuar como los chips que los diseñadores de hardware usan, cada uno soportando una función específica. Quizás gracias a esta analogía, esta aproximación ha sido conocida como componentes software.

Los programadores han reconocido la necesidad de reusar software desde antes de los compiladores. Alguna de las críticas al reuso son culturales, incentivando en muchas organizaciones la reinvención mejor que el reuso, por ejemplo. Pero tecnológicamente el potencial de reuso también estaba restringido. Los mecanismos de reuso existentes no van muy lejos. Para entender porque esto es así, puede servirnos de ayuda examinar los dos esquemas de reutilización más comunes: librerías y objetos.

Las librerías tienen mucho que ofrecer como mecanismos para el reuso. Esto es especialmente cierto cuando hablamos de librerías de enlace dinámico, que pueden ser usadas bajo demanda, están típicamente compartidas en lugar de las estáticamente enlazadas dentro de una sola aplicación. Las librerías nos son familiares y son fáciles de usar. Como se pueden distribuir en forma binaria, no hay riesgo de revelar el código fuente a ojos fisgones. Y con un poco de cuidado, un programa escrito en un lenguaje puede comunicarse con rutinas de otro lenguaje diferente. Las librerías no carecen de problemas el más significante es la dificultad para añadir funcionalidad: ¿Cómo instalar una nueva versión de una librería sin dañar las aplicaciones que usan la antigua? y ¿Cómo puede haber de una forma fácil y segura más de una implementación de una misma librería en el sistema, que podría ser necesario en algunas circunstancias? Vemos que solo con las librerías no tenemos suficiente.

Encapsulando datos y métodos los objetos pueden proporcionar una forma limpia de empaquetar trozos reusables de funcionalidad. Como las librerías tradicionales, los objetos que resuelven problemas específicos pueden crearse una vez y usarse muchas veces. Los objetos tienen a menudo más que ofrecernos que las librerías. A través de la herencia, un objeto reusar la definición de interfaz o su código o ambas cosas de otro objeto. Y el polimorfismo simplifica el reuso ocultando diferencias irrelevantes a los clientes de los un objeto.

A pesar de estas ventajas, la tecnología de objetos no ha conseguido su potenciarse de tal forma permita el reuso. Para ver porque, consideremos lo siguiente: ¿Por que no puede una organización que quiere escribir una nueva aplicación comenzar el proceso visitando el almacén de programas, buscando en un catálogo o buscando en la WWW el objeto que necesita? ¿Por que no hay un mercado grande de objetos reusables para la empresa? Los desarrolladores de hardware se benefician de este tipo de mercado, así que ¿Por qué los creadores de software no tienen uno también? ¿Por que no hay tiendas de objetos ricos en variedades?

La respuesta es que esta en la raíz de la tecnología de objetos que se usan actualmente. Los lenguajes orientados a objetos como C++ fueron diseñados para permitir el reuso dentro de un grupo de trabajo o como mucho de una organización. Mientras puedes muy posiblemente encontrar algún objeto reusable en C++ en venta, el tipo de tienda de objetos a nivel mundial que se expone aquí no es posible con la tecnología existente. Hay tres grandes problemas para esto.

La primera y quizás la más importante es que no hay un estándar que permita enlazar el código binario de los objetos. Aunque se puede compilar un objeto en C++ y usar ese binario compilado desde una librería, esto funciona solo cuando el mismo compilador es usado para ambas librerías y la aplicación que las usa. C++ no tiene un estándar de compilación para los objetos en formato binario. Como resultado de esto, los objetos que están disponibles en C++ casi siempre incluyen la fuente. Un punto relacionado: reusar código a través de la herencia de implementación tiende a enlazar

objetos padres e hijos muy estrechamente. El creador de el objeto hijo tendría acceso al fuente del padre, para saber lo que ocurre cuando un método heredado es llamado.

¿Es razonable esperar que el creador del software de nuestra hipotética tienda este preocupado por su código fuente, porque revela los secretos de programación? La contestación parece ser no, ya que dicho bazar no existe. Aunque el reuso a nivel de fuente es viable en grupos de desarrollo e incluso empresas, para una distribución a nivel mundial tiendas de binario son esenciales.

El segundo problema que, a pesar de ser el dominante en el mundo de los lenguajes orientados a objetos es que C++ no es el único lenguaje. Un objeto escrito en C++ no puede ser reusado en Smalltalk. Una tienda de objetos debe ofrecer objetos que pueden ser usados y reusados en varios lenguajes y en varios entornos de desarrollo, pero actualmente es difícil reusar un objeto escrito en un lenguaje dentro de una aplicación escrito en otro lenguaje.

El tercer problema es este: si creas una aplicación con objetos escritos en C++ y decides cambiar uno de los objetos, debes reenlazar y quizás incluso recompilar la aplicación. Si varias aplicaciones en un sistema usan este objeto cambiado debes reenlazar o recompilar todas ellas. Idealmente, debería haber una forma de poner una nueva versión de un objeto y todas las aplicaciones que lo usan se actualicen automáticamente. Y por supuesto sin necesidad de reenlazar o recompilar las aplicaciones.

Todos estos problemas se resuelven con COM. Los objetos COM pueden ser empaquetados en librerías o ficheros ejecutables y después ser distribuidos en formato binario. Como COM define el estándar binario para acceder a estos objetos binarios, los objetos COM pueden ser escritos en un lenguaje y usados en otro. Y como los objetos COM se instancian cuando se necesitan, cuando una nueva versión se instala en el sistema, todos los clientes automáticamente tendrán la nueva versión la siguiente vez que el objeto sea usado. COM ofrece los beneficios del reuso de las librerías y los objetos, junto con otros beneficios que no ofrecen ni las librerías ni los objetos, la más importante es una aproximación común al acceso a todo tipo de servicio software.

COM proporciona los beneficios del reuso, ya usado durante mucho tiempo en hardware, para la creación de software. De echo existen sitios en Internet llenos de componentes basados en COM ya disponibles, donde puedes verlos e incluso usarlos y revistas llenas de anuncios de componentes. La tienda de objetos está comenzando a ser realidad, de tal forma que los programadores pueden crear aplicaciones que están construidas con partes reusables. La arquitectura general de los servicios de COM es muy útil en muchas cosas, quizás la creación de componentes software fuese el objetivo más importante de sus creadores.

7.2.1.5 Los beneficios de COM

Todo lo que simplifique el complejo esfuerzo de la creación de grandes trozos de software es bueno. La terminología definida por COM nos permite esto de varias formas.

COM ofrece caminos para estructurar los servicios proporcionados por un trozo de software. Los programadores pueden diseñar su implementación primero

organizando lo en objetos COM y luego definiendo las interfaces de cada objeto. Este es uno de los beneficios tradicionales de una aproximación basada en componentes al diseño e implementación. Y como se acaba de comentar, COM va más allá permitiendo a los programadores crear componentes que pueden ser distribuidos y reusados de varias formas.

Un segundo beneficio de COM ya ha sido mencionado: consistencia. Proporciona una aproximación simple para acceder a todo tipo de servicios software, COM simplifica el problema con el que se enfrentan los programadores. Tanto si el software en cuestión esta en una librería, en otro proceso o parte del sistema software puedes siempre acceder a él de la misma forma. Un efecto lateral de esta consistencia es que COM tiende a difuminar la distinción entre aplicaciones y sistema software. Si puedes acceder a todos como objetos COM percibirás muy poca diferencia entre estos dos tipos de software, que han tenido tradicionalmente una gran diferencia. Por el contrario, puedes desarrollar aplicaciones que se construyen sobre servicios software en tu entorno, estén donde estén y los proporcione quien los proporcione. Además, COM es independiente del lenguaje de programación usado. Como el COM define una interfaz binaria que los objetos deben soportar puedes escribir objetos COM en cualquier lenguaje capaz de hacer llamadas a través de éste interfaz binario para invocar métodos de la interfaz de estos objetos. Un objeto y sus clientes ni conocen ni se preocupan del lenguaje en el que otros han programado el objeto. Mientras se debe comentar que hay lenguajes que encajan mejor con COM que otros, también hay que tener en cuenta que COM es independiente de cualquier lenguaje. Otro beneficio de COM es la aproximación a un problema persistente en el desarrollo de software y es la gestión de versiones, es decir reemplazar una versión existente de software por otra con nuevas características, sin que se corrompa ningún cliente. Los objetos COM proporcionan una solución sencilla basada en la habilidad de los objetos para tener mas de una interfaz. Como se comentó anteriormente, un objeto cliente COM debe adquirir un puntero a una interfaz especifica que necesite usar. Para añadir características en una nueva versión de un objeto COM, simplemente puedes ofrecer nuevas características a través de una nueva interfaz en el objeto. Las interfaces existentes no se cambian, de hecho COM prohíbe cambios a las interfaces ya existentes, de tal forma que los clientes que usan esas interfaces no se ven afectados. Y los antiguos clientes no pregunta por las interfaces que no le afectan. Y estos clientes antiguos nunca preguntan por nuevas interfaces. Solo los nuevos clientes sabrán suficiente para usar los nuevos servicios y solo los nuevos clientes ser verán afectados por la nueva versión.

COM también resuelve otro tipo de problema de gestión de versiones: ¿Que ocurre si un cliente espera de cierto objeto cierta funcionalidad, pero el objeto aun no ha sido actualizado para ofrecerla? El cliente pide un puntero a una interfaz a través de la que sus servicios estarán disponibles pero no le devuelve ninguno. COM proporciona una forma de aprender que un objeto no es todo lo que el cliente esperaría, los programadores pueden escribir clientes para manejar estas situaciones adecuadamente en lugar de romperse. Esta simple aproximación a la gestión de versiones que permite a los clientes y objetos independientemente actualizar su uso y de las contribuciones más grandes que ha hecho COM.

7.2.1.6 Disponibilidad de COM

COM, se desarrollo inicialmente para Windows pero ahora proporciona soporte para Macintosh. Aunque Microsoft no de soporte a COM en otro SO. Varias empresas,

grandes y pequeñas, proporcionan implementación de COM y varias tecnologías basadas en este en un amplio rango de sistemas operativos. Programas desarrollados usando objetos COM estarán disponibles en todos los sistemas operativos desde Workstation que ejecutan Windows y NT a maiframes IBM que usan MVS. Y como se verá después, DCOM permite interactuar a objetos COM de todo tipo de sistema.

OpenCOM [Lanc01] está formado por la construcción de un subconjunto ligero, eficiente y reflectivo de las librerías utilizadas en COM, lo desarrolla la Universidad de Lancaster, es una aversión abierta y libre de COM.

7.2.1.7 Las Interfaces Estándar de COM

COM proporciona el mecanismo básico necesario para que un trozo de software proporcione servicios a otro a través de la implementación de una interfaz COM bien definida. Pero ¿Cómo definir esas interfaces? A no ser que COM y sus clientes se pongan de acuerdo en que interfaces existen, que métodos contienen estas interfaces y que hace cada método, no será posible para ellos realizar nada útil.

En algunos casos, el programador puede definirse sus interfaces específicas para una aplicación concreta. Por ejemplo, un banco puede crear su propio software para llevar a cabo sus negocios y decidir que objetos usar. El programador define sus propias interfaces e implementar objetos que los soporten, no es necesarios ningún tipo de consentimiento o aprobación por parte de Microsoft.

Pero supongamos que todos los bancos tienen similares necesidades de objetos e interfaces. ¿Porque no poner a todos a definir una interfaz para estos objetos y definir un estándar industrial en este tipo de software? Esto permitiría la creación de un mercado para componentes estándares producido por las competencias entre compañías. OLE Industry Solutions, es un programa financiado por Microsoft para definir este tipo de interfaces, tiene precisamente ese objetivo. A través de este programa, grupos desde compañías financieras, organizaciones de la salud, proveedores de equipos de venta punto a punto, y muchos otros han definido interfaces estándares para componentes útiles en cada área.

Hay otros tipos de servicios donde los nuevos estándares de interfaces podrían ser incluso mejor conocidos. Por ejemplo, supongamos que el propietario de un sistema operativo decide realizar servicios de su sistema de ficheros, vía COM. Tendrían que definir uno o más objetos COM cada uno con una clase específica y proporcionando un determinado conjunto de interfaces. Después tendría que publicar estas interfaces para los usuarios potenciales de este servicio, para que los programadores puedan usar este sistema de ficheros.

El problema original solucionado por OLE, los documentos compuestos, es otro ejemplo de la necesidad de estándares en interfaces. Un documento compuesto contiene elementos de dos aplicaciones que comparten una ventana en la pantalla del usuario, permitiendo al usuario trabajar con la información de ambas aplicaciones. Claramente, ambas aplicaciones deben cooperar para hacer esto posible, proporcionando servicios el uno al otro que les permitan presentar una interfaz de usuario uniforme.

7.2.1.8 La tecnología ActiveX y OLE

Esta sección proporciona una breve introducción a las tecnologías basadas en COM más importantes.

7.2.1.9 Automatización

Las hojas de cálculo, los procesadores de texto y otro software personal permiten que los usuarios obtengan la máxima funcionalidad. ¿Por qué no permitir a otros programas acceder a esta funcionalidad? Para que esto sea posible las aplicaciones deben exponer su funcionalidad a los programas a demás de a los usuarios. En otras palabras, estas aplicaciones deben ser programables. Proporcionar esta programabilidad es el objetivo de la automatización.

Una aplicación puede ser programable exponiendo todos sus servicios a través de las interfaces normales de COM, sin embargo esto rara vez se hace, en vez de eso las aplicaciones exponen su funcionalidad a través de dispinterfaces. Una dispinterfaz es una interfaz normal como se ha comentado anteriormente, pero se diferencia en que estos métodos son más fáciles de llamar desde clientes escritos en lenguajes como Visual Basic, realmente la pispinterafaz nos proporciona metainformación sobre el conjunto de interfaces que queramos dejar disponibles dentro de nuestra aplicación además de ofrecernos métodos para poder llamar a otros métodos de otras interfaces en tiempo de ejecución y no en tiempo de compilación como podría ocurrir con las interfaces normales. De esta forma una aplicación podría ver todos los métodos que nos ofrece una hoja de cálculo como Excel que permite automatización ver que parámetros necesita y llamarlos, permitiendo así obtener la funcionalidad de Excel. Esto por otra parte le permite a Excel u otras aplicaciones con automatización ampliar su sistema de macros a cualquier lenguaje que pueda utilizar automatización y así aumentar su productividad, de hecho cualquier cosa que se puede hacer desde el menú de Excel se puede hacer con automatización.

7.2.1.10 Persistencia

Los objetos, como sabemos tienen métodos y datos y muchos objetos necesitan alguna forma de guardar sus datos cuando dejan de ejecutar para volver a recuperarlos cuando vuelvan a la ejecución, con palabras más técnicas un objeto necesita hacer sus datos persistentes que al final de cuentas significa guardar sus datos en el disco. COM permite varias formas de realizar esto una de las más comunes es mediante lo que se conoce como Almacenamiento Estructurado.

Para comprender mejor el funcionamiento veamos como guardan los datos las aplicaciones normales, los sistemas tradicionales de ficheros permiten que las aplicaciones compartan un disco sin cada programa maneja sus propios ficheros e incluso su propio directorio de trabajo independientemente de lo que otras aplicaciones puedan hacer. Una aplicación no necesita cooperar para almacenar sus datos, ya que cada una tiene asignado su propia área de almacenamiento.

Sin embargo con COM la cosa es un poco más complicada, ya que este permite a todo tipo de aplicaciones trabajar juntas usando un modelo simple. Objetos COM independientemente desarrollados pudrían ser parte de una aplicación pero aun necesitar guardar sus datos en disco separadamente. Mientras cada objeto COM podría usar su

propio fichero, a los usuarios de la aplicación el objeto es invisible pero deberían seguir la pista de múltiples ficheros.

Lo que se necesita es una forma de que varios objetos COM compartan un simple archivo. Esto es lo que nos permite el Almacenamiento Estructurado permitiéndonos tener un sistema de ficheros dentro de cada archivo. De esta forma para el usuario solo existe un archivo pero la aplicación guarda los datos de cada componente en áreas separadas.

Para permitirnos esto el Almacenamiento Estructurado nos proporciona dos tipos de objetos COM, cada uno con sus interfaces adecuadas, uno se llama Almacén y el otro se llama Secuencia estos objetos son análogos a los directorios y los archivos respectivamente.

Pero esto no es todo sobre persistencia, una objeto puede permitir salvar sus datos en otros formatos como en ficheros ordinarios o incluso en la WEB. Además un objeto puede proporcionar una forma para que los clientes le indiquen cuando debe cargar o guardar sus datos, para todo esto se definen otros tipos de interfaces.

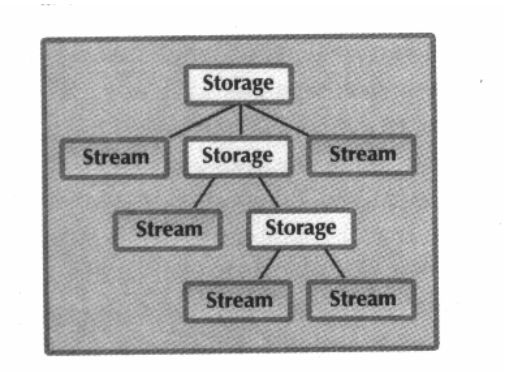


Ilustración 7.18: Persistencia en COM

7.2.1.11 Moniker

Imagina una instancia de un objeto COM que representa a tu banco. Para acceder a tu cuenta un cliente necesita ejecutar el objeto y decirle al objeto que cargue sus datos. COM proporciona una forma de realizar esto, es decir proporciona mecanismos que permiten a un cliente instanciar e iniciar a un objeto COM.

Para realizar esto el cliente necesita saber bastante debe saber por ejemplo donde se encuentran localizados los datos para tu cuenta y como decirle al objeto COM apropiado como cargar estos datos. Es razonable que el cliente sepa hacer esto, pero sería mucho más cómodo simplemente decir “Créame la instancia del objeto” y que el resto ocurriera automáticamente.

Esto es lo que los monikers hacen, no son más que simples objetos COM pero cuyo propósito es saber como crear e inicializar instancias de otros objetos. Si yo tengo un moniker para tu cuenta bancaria yo podría pedirle que cree, inicialice y me conecte a tu cuenta. Todos los detalles necesarios para realizar esto están ocultos. Si quiero dos cuentas bancarias necesitare dos monikers. En general no son necesarios en el entorno COM simplemente hacen que las cosas sean más fáciles al cliente.

7.2.1.12 Transferencia Uniorme de Datos y Objetos Conectables

El intercambio de datos es una operación fundamental en el mundo de la informática. Las aplicaciones copian datos para arriba y para abajo, por ejemplo cuando usan el portapapeles para compartir datos. Hay un montón de formatos y formas de almacenar e intercambiar datos pero todos son diferentes esta tecnología nos va a permitir una forma uniforme y estándar de intercambiar datos.

Como siempre esto se realiza en COM a través de una serie de métodos que están dentro de unas determinadas interfaces, estos definen formas estándares de describir los datos que se van a intercambiar, donde se encuentran los datos y donde moverlos o a quien pasarse los, incluso se define un mecanismo simple que permite a una aplicación informar a otra cuando un determinado dato está disponible. Esta tecnología es llamada Transferencia Uniforme de Datos y juega un papel muy importante en el papel que realizan las aplicaciones basadas en COM. Este mecanismo sencillo de aviso a veces es insuficiente, con lo que se ha creado una tecnología conocida como Objetos Conectables para subsanar estas deficiencias.

7.2.1.13 Documentos Compuestos

Las aplicaciones cada vez son más complicadas. Los procesadores de texto añaden capacidades gráficas, hojas de cálculo y funciones de gráficos de barras parece que el objetivo es llegar a una gran aplicación que lo realice todo. Pero lo que realmente ocurre es lo que se está consiguiendo es realizar la integración de diferentes aplicaciones. Así un procesador de textos no necesita funciones gráficas, simplemente puede usar las de las aplicaciones gráficas que ya existen para usarlas dentro de sus propios documentos. Un usuario vería lo que aparentemente es un simple documento pero lo que realmente esta viendo es un conjunto de aplicaciones cooperando para trabajar en varios trozos del documento.

Las tecnologías OLE son las que nos permiten resolver este problema. Permitiendo que con los objetos COM apropiados cada un con sus interfaces, las aplicaciones puedan cooperar para presentar documentos compuestos al el usuario. Estas interfaces son completamente genéricas, nadie sabe nada del otro, ninguna aplicación sabe nada de la otra. Pero estas interfaces no están definidas solo para procesadores de texto en realidad cualquier tipo de interacción está permitida por ejemplo puedes incluir sonido dentro de gráficos, crear presentaciones con video clips integrados. Muchas aplicaciones soportan OLE como la forma de interacción con otras aplicaciones.

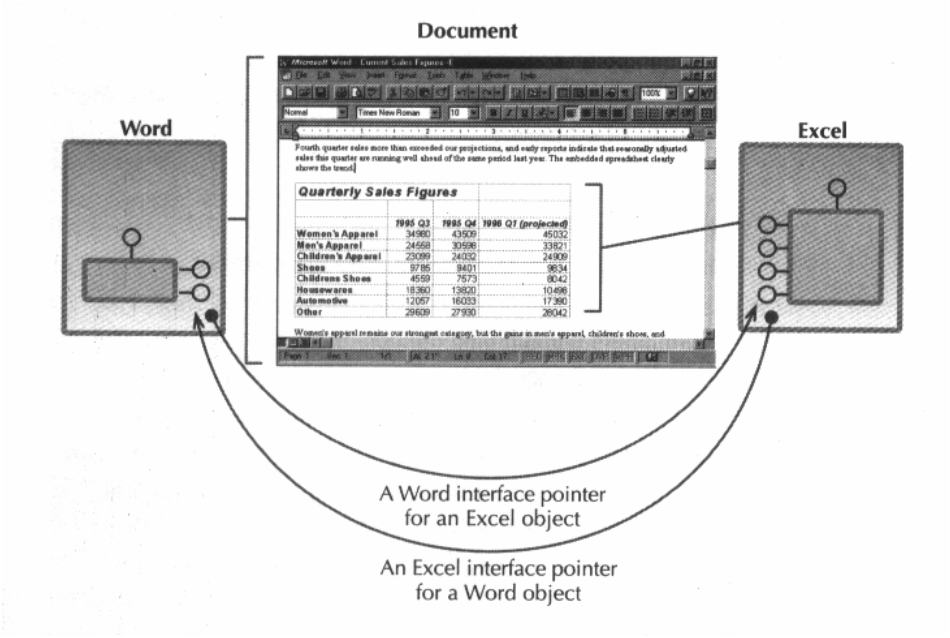


Ilustración 7.19: Documentos compuestos. OLE en COM

Cuando se crea un documento compuesto OLE, una aplicación siempre actúa como contenedora, como el nombre implica un contenedor define el documento exterior y contiene a todo lo demás.

Usando OLE un documento servidor puede ser incrustado o enlazado en el documento contenedor. Si el documento es enlazado este se almacena en un documento separado y solo un enlace a ese fichero se almacena en e documento si por el contrario un documento servidor es incrustado ese documento se almacena en el mismo archivo que el documento contenedor. Las dos aplicaciones comparten un mismo fichero, esto es posible gracias al Almacenamiento Estructurado como vimos anteriormente.

La creación de documentos compuesto fue el problema que condujo hasta la creación de COM, aunque este es usado ahora mucho más ampliamente las huellas dactilares de los documentos compuestos se observan aun en muchas de las tecnologías basadas en COM. Este reto motivó el diseño de una gran cantidad de tecnologías basadas en COM. Particularmente creo que con excepción de los Controles y algunas nuevas tecnologías de servicios como transacciones o por otro lado DCOM, lo demás es fruto del esfuerzo por conseguir OLE.

7.2.1.14 Controles ActiveX

Un control ActiveX es un componente software que realiza algunas cosas específicas siguiendo una forma estándar. Los programadores pueden conectar uno o más controles dentro de una aplicación creada por ejemplo en Builder C++ y obtener todas las ventajas de las funcionalidades de este software sin necesidad de tener que rehacerlo. El resultado es la realización de programas a partir de partes prefabricadas es decir de componentes software.

Por ejemplo un control ActiveX puede almacenar sus datos en algún sitio de la red o puede ser descargado desde un servidor de la red y después ser ejecutado en un

cliente. El contenedor donde se ejecuta este control no tiene porque ser un entorno de programación, puede ser un navegador.

Un control ActiveX no es una aplicación separada sino como se ve en la Ilustración 7.20 son servidores que se conectan dentro de un contenedor de controles. Como siempre las interacciones entre un control y sus contenedores se especifican a través de varias interfaces en objetos COM. Esta tecnología usar muchas otras tecnologías OLE y ActiveX como la interfaz para incrustar o acceso a sus métodos vía dispinterface o generación de eventos mediante Objetos Conectables.

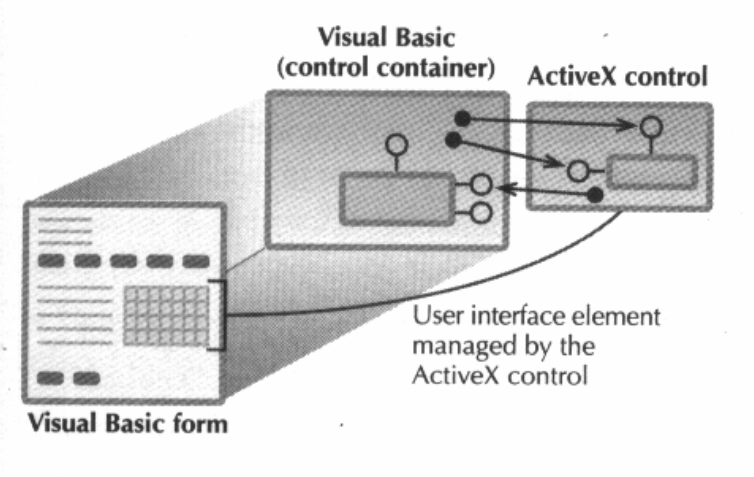


Ilustración 7.20: Controles ActiveX

7.2.1.15 COM Distribuido

Aunque se diseño para soportar distribución desde el principio la implementación original de COM solo funcionaba en un sistema. Los objetos COM pueden ser implementados en DLL o en procesos separados dentro de la misma máquina, pero no pueden residir en otras máquinas de la red. Con DCOM esto cambia, ahora los objetos pueden proporcionar sus servicios en otras máquinas como se muestra en la Ilustración 7.21.

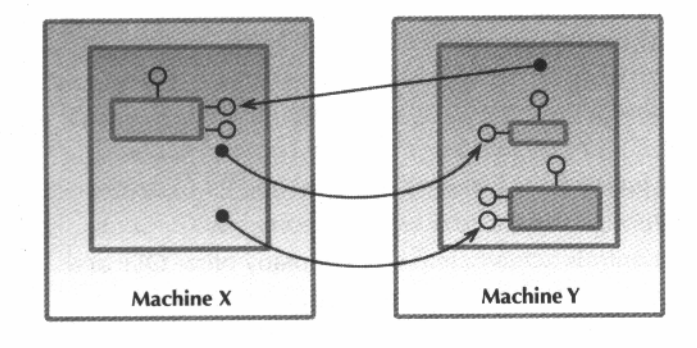


Ilustración 7.21: COM distribuido

Para realizar esto DCOM usa RPC que permite realizar llamadas aparentemente locales pero que se ejecutan en máquinas diferentes. DCOM incorpora un servicio de seguridad que controla que clientes puede usar que objetos, y un mecanismo para indicar en que máquina debe crearse un objeto.

7.2.1.16 Interfaces de Servicios basados en COM

Es a menudo interesante tener una interfaz común para acceder a diferentes implementaciones de un servicio. Por ejemplo Open Database Conectivity (ODBC) de Windows y Windows Server define un conjunto de funciones en C que pueden ser usadas para acceder a cualquier sistema de gestión de bases de datos. Con la llegada de COM, este tipo de interfaces puede ser especificado en un objeto común. Microsoft ha definido varias interfaces de este tipo incluyendo esta de bases de datos, de transacciones y servicios de directorios

7.2.1.16.1 Bases de datos

Un sistema de gestión de bases de datos proporciona una forma de organizar, almacenar y obtener información. Las DBMS se usan ampliamente en muchas aplicaciones, el acceso local es a través de una librería que se enlace con el proceso cliente.

Si se definen objetos e interfaces estándares que se usen ampliamente, un cliente puede acceder a varias DBMS de la misma forma o incluso a la mejor parte de cada uno de ellos, incluso no hay ninguna razón por la que esta misma interfaz puede ser usada de una forma más general para acceder a datos aunque estos no se encuentren en una DBMS.

La tecnología basada en COM para acceso a bases de datos soluciona este problema definiendo estándares de objetos e interfaces para acceder a datos, esta tecnología proporciona una medio común para que los clientes puedan acceder a los datos almacenados de diversas formas. En realidad OLE Database va más allá del estándar comentado anteriormente y lo ve todo como objetos COM, de tal forma que una fuente de datos puede modelar un objeto DataSource que a su vez tiene un objeto Command definido. Este objeto Command puede especificar una consulta SQL u otro tipo de comando que permita manejar los datos. Cada objeto Command tiene una interfaz que contiene un método Execute que ejecuta el comando. El resultado es otro objeto llamado Rowset que contiene el resultado de la ejecución del comando. Este objeto está provisto de una interfaz con métodos que permiten examinar los datos contenidos en el objeto. Todos estos objetos se definen usando COM y ofrecen sus servicios a través de interfaces COM. El resultado es una visión abstracta del acceso a los datos que puede ser implementada de muchas formas y para una gran cantidad de mecanismos de acceso.

7.2.1.16.2 Transacciones

En el acceso a datos, especialmente a datos distribuidos, la noción de transacción puede ser muy útil. Supongamos que queremos modificar dos bases de datos pero que o se modifican las dos o no se modifica ninguna de tal forma que no se puede modificar una y la otra no. Un claro ejemplo es una transferencia de dinero entre dos cuentas bancarias se debe quitar dinero de una y ponerlos en la otra, en este caso se ve

claramente que deben realizarse las dos operaciones o ninguna de las dos no cabe la posibilidad de que se sustraiga el dinero de una cuenta y no se añada a la otra.

Para realizar este tipo de operaciones indivisibles se debe definir una transacción que incluya ambas modificaciones. Este servicio puede ser incluido en el mecanismo de acceso a datos comentado anteriormente pero tenerlo separado para poder utilizarlo con diferentes servicios de este tipo puede resultar una mejor idea. Una vez más tenemos un trozo de software que ofrece servicios a otro, el servicio de transacciones a una aplicación cliente, entonces ¿Por qué no crear esta interacción usando COM?

Así como la tecnología de bases de datos de COM lo modela todo usando solo objetos COM el modelo de transacciones basado en COM ve las transacciones como objetos COM. Los objetos definidos incluyen gestores de recursos por ejemplo DBMS, coordinadores de transacciones y transacciones.

7.2.1.16.3 Servicios de Directorios

> Como las guías o las páginas amarillas de telefónica, un servicio de directorios es un entorno distribuido que permite a sus usuarios buscar información, se puede buscar tanto información referente a una máquina como referente a una persona o cosa el concepto y la idea es la misma, se puede buscar el nombre de una máquina y podemos obtener información sobre su dirección en la red o su procesador; o por el contrario podemos darle el nombre de una persona y nos devolverá su email o su número de teléfono.

Debido a que no hay un solo directorio que guarde toda la información que un usuario pueda necesitar existen diferentes servicios de directorio y diferentes tecnologías por ejemplo Windows NT usa X.500 el más estándar pero el menos utilizado, Novell definió su propio servicio llamado NDS y para Internet se ha creado un protocolo de servicio de directorios llamado LDAP.

Siguiendo con la filosofía de los dos servicios anteriores OLE Directory Services u OLE DS proporciona una interfaz común para acceder a todo tipo de servicio de directorios independiente de su implementación y siguiendo la misma línea se crea una abstracción a partir de objetos COM. De esta forma se puede crear un cliente que pueda funcionar con todos los servicios de directorios.

7.2.1.16.4 Internet y COM

La mayoría de las tecnologías creadas por Microsoft en Internet se han basado en COM. Por ejemplo el navegador de Microsoft, Internet Explorer, está basado en COM y en una extensión de OLE llamada documentos ActiveX, de esta forma un navegador puede visualizar varios tipos de información además de las páginas HTML. Por otro lado la tecnología de controles ActiveX ha sido mejorada para permitir que se pueda descargar controles dentro de un navegador. Los scripts ActiveX proporcionan una forma genérica de ejecutar scripts en cualquier lenguaje script mientras que el estilo de hiperenlaces del Web, basando se en monikers, permite la creación de hiperenlaces pero no solo entre paginas HTML sino entra cualquier tipo de documentos.

7.2.2 El Modelo de Objeto Componente

COM especifica las reglas necesarias para definir objetos e interfaces. Comprender COM es un prerrequisito para entender ActiveX. Aunque COM no es grande o complicado difiere tanto de otras arquitecturas para proveer servicios como de la tradicional aproximación a los tradicionales objetos[CHA1996].

7.2.2.1 Interfaces

Cada interfaz del objeto es realmente un contrato entre el este y sus clientes. El objeto promete mantener los métodos de la interfaz exactamente como se han definido, y el cliente debe invocar los métodos correctamente. Para hacer que este contrato funcione, el objeto y sus clientes deben ponerse de acuerdo en lo siguiente: una forma para identificar cada interfaz; una forma común para describir o definir los métodos de cada interfaz; y una definición concreta de como implementar cada interfaz.

7.2.2.1.1 Identificar una Interfaz

Cada interfaz COM tiene dos nombres. Uno que tiene sentido para las personas es una cadena de caracteres. El otro es más complicado y está destinado para ser usado por los programas. El nombre que entiende el ser humano puede no ser único, puede ocurrir, aunque no es normal, que dos interfaces tengan el mismo nombre. El nombre para los programas, es único.

Por convenio, los nombres para los humanos de las interfaces comienzan por la letra I de interfaz. Diferentes tecnologías basadas en COM definen interfaces con varios nombres, pero los nombres típicamente comienzan con I, e intentan ser un poco descriptivos de la función que desempeñan.

El creador de cada interfaz debe asignar un nombre único, llamado identificador único global GUID. Un GUID para una interfaz se llama IID. Un GUID son 16 bytes que se generan mediante un programa que cada vez genera una diferente.

7.2.2.1.2 Describir una Interfaz

La forma de describir una interfaz a nivel de fuente es independiente, y de hecho COM no especifica nada, de la especificación binaria estándar que define COM.

Es conveniente aunque no hay una herramienta estándar definir las interfaces a nivel de fuente, para ello se utiliza IDL que una extensión del IDL del Open Software Foundation Distributed Computing Environment (OSF DCE) que también definió un identificador único como el GUID que lo llamo UUID. Veamos un ejemplo de especificación IDL:

```
[ object,
  uuid(E7CD0D00-1827-11CF-9946-444553540000) ]
interface ISpellChecker : IUnknown {
     import "unknwn.idl";
     HRESULT LookUpWord([in] OLECHAR word[31],
                        [out] boolean *found);
     HRESULT AddToDictionary([in] OLECHAR word[31]);
     HRESULT RemoveFromDictionary(
                        [in] OLECHAR word[31]);
}
```

Ilustración 7.22: IDL de COM

Una cosa importante a sabes es que cuando se ha implementado una interfaz COM indica expresamente que jamás debe modificarse, es decir si queremos añadir un nuevo método, deberemos definir una nueva y nunca modificar la antigua. Las interfaces son inmutables.

7.2.2.1.3 Implementar una Interfaz

COM describe un estándar binario para las interfaces de tal forma que todos los objetos COM deben cumplirlo obligatoria mente, esto permite que un cliente pueda invocar los métodos de otro objeto sin preocuparse del lenguaje de programación solo debe preocuparse de cumplir con el estándar binario. La Ilustración 7.23 muestra un ejemplo del formato binario de una interfaz.

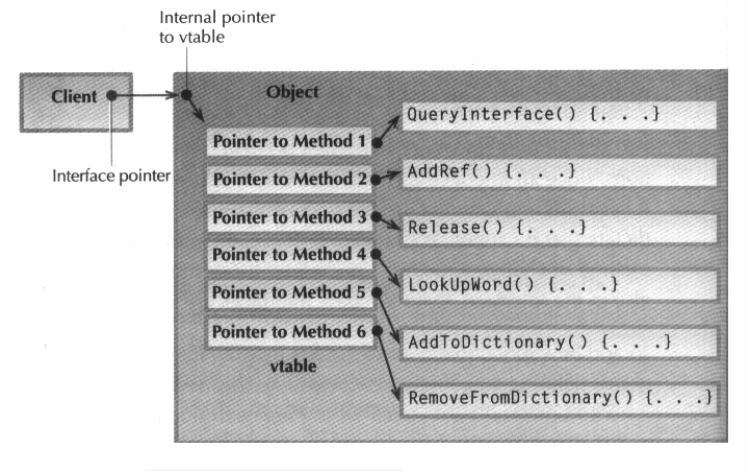


Ilustración 7.23: Puntero a interfaz COM

Como podemos observar en la Ilustración 7.23 el puntero a la interfaz del cliente apunta a una dirección donde se encuentra una tabla de punteros a los diferentes métodos, esta tabla se suele llamar vtable, los tres primeros punteros deben ser siempre los punteros a los métodos de la interfaz IUnknown, ya que toda interfaz en COM siempre hereda de IUnknown. Cada interfaz cumple esta estructura por ejemplo si el objeto tuviese otra interfaz que se llamase ITesaurus esta tendría su propia vtable y los tres primeros punteros apuntarían a los métodos de IUnknown.

Esta forma de estructuración procede de cómo el compilador de C++ de Microsoft compila las clases virtuales, de tal forma que en C++ es relativamente muy sencillo escribir objetos COM, aunque estos se pueden crear en cualquier lenguaje que permita seguir este estándar binario.

7.2.2.1.3.1 La interfaz fundamental, IUnknown

Cada objeto COM debe soportar la su propia interfaz y la interfaz IUnknown, esta interfaz contiene tan solo tres métodos: QueryInterface, AddRef y Release. Como todos los objetos deben implementar esta interfaz todos los objetos pueden acceder a estos tres métodos.

7.2.2.1.3.2 Usando IUnknown::QueryInterface

Normalmente un cliente obtiene un puntero al objeto cuando lo crea, una vez con ese puntero ¿Qué debe hacer el cliente para obtener un puntero a una interfaz determinada? La respuesta es sencilla, simplemente debe llamar al método IUnknown::QueryInterface, Ilustración 7.24 ilustra el funcionamiento esta función.

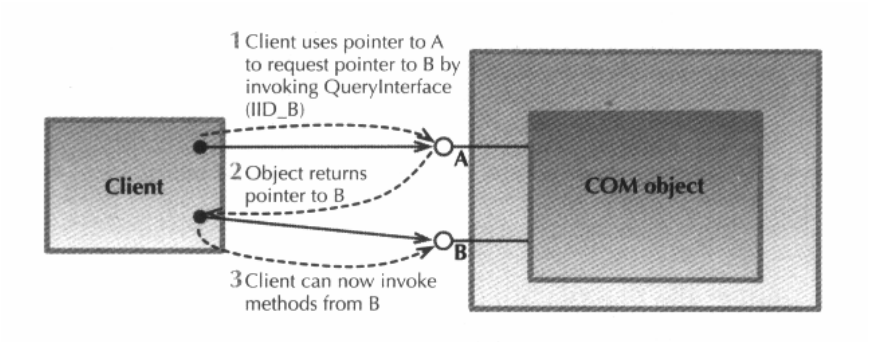


Ilustración 7.24: QueryInterface

El método QueryInterface junto con la restricción impuesta de que nunca jamás se debe cambiar la funcionalidad de una interfaz son consideradas como los elementos más importantes de COM ya que nos permite hacer modificaciones a nuestros objetos sin que se vean afectados nuestros clientes. Un ejemplo es que tenemos una aplicación que utiliza la interfaz A de un objeto servidor, si queremos actualizar nuestro objeto servidor y añadir nueva funcionalidad debemos crear otro interfaz B de tal forma que nuestra aplicación seguirá usando la interfaz A sin verse afectada para nada, si por el contrario adquirimos una versión más actualizada de nuestra aplicación también podremos tener acceso a los servicios de la interfaz B.

7.2.2.1.3.3 Contar Referencias

Para controlar el ciclo de vida de un objeto se utiliza un método denominado 'contador de referencias' es decir cada vez que se crea una referencia a un objeto se incrementa el contador llamando a el método addRef y cuando se deja de utilizar se llama al método Release que cuando el contador llega a cero borra el objeto.

7.2.2.2 Clases

Cada objeto COM es una instancia de una clase y a cada clase se le puede asignar una GUID llamada CLSID aunque esto no es necesario así como tampoco es obligatorio crear una nueva CLSID cuando se añaden interfaces a esa clase aunque puede ser recomendable. Realmente una clase identifica a un conjunto de implementaciones de interfaces y no solo a las interfaces de tal forma que una clase puede implementar las interfaces A y B mientras que otra clase diferente también las puede implementar.

7.2.2.3 Servidores para Objetos COM

Cada objeto COM está implementado dentro de un servidor. Este servidor contienen el código que implementa los métodos de las interfaces y sigue la pista de los datos del objeto mientras está activo. Un servidor puede contener más de un objeto de una determinada clase e incluso puede contener a más de una clase. La Ilustración 7.25 muestra los tres tipos principales de servidores:

1. Servidor In-process, cuando se implementa el servidor dentro de una DLL y comparte el mismo espacio de direcciones del cliente.
2. Servidor Local, cuando el servidor se implementa en un proceso separado en la misma máquina.
3. Servidor Remoto, cuando el servidor se implementa en una DLL o en un proceso separado pero en una máquina diferente.

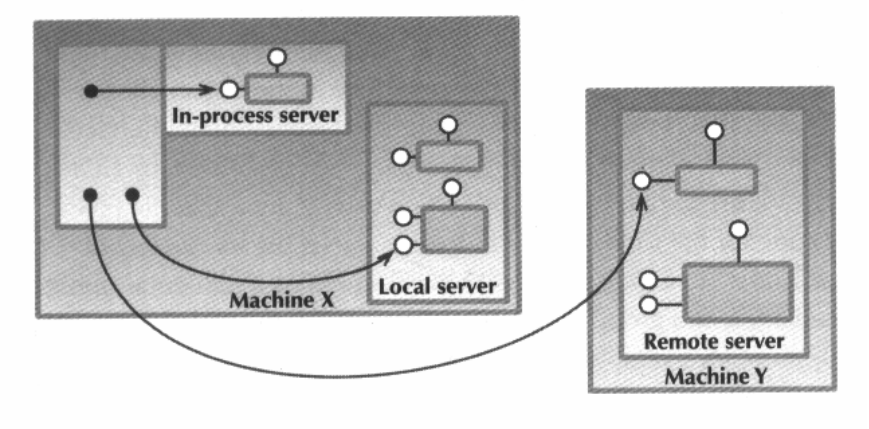


Ilustración 7.25: Servidores COM

Para un cliente la forma de acceder a un objeto es independiente del tipo de servidor, siempre va a manejar punteros.

7.2.2.4 COM y multiprocesos

El problema de los multiprocesos dentro de un entorno de programación siempre crea complejidad, los programadores deben de tener cuidado para evitar posibles conflictos entre procesos sobre todo cuando se desea acceder a una variable común. COM soluciona este problema de dos formas: la primera y original fue usando el modelo de ‘apartamento’, la idea es que aunque un proceso puede tener varias líneas de proceso los objetos individuales no pueden. Cada línea de proceso actúa como un

apartamento y cada objeto COM vive dentro de un apartamento es decir una línea de proceso, las llamadas desde otras líneas de proceso se encolan y gestionan secuencialmente por el objeto; un segundo modelo es el llamado 'líneas de proceso libres' o 'multiproceso' de esta forma varias líneas de proceso pueden ser activadas a la vez dentro de un objeto COM.

7.2.2.5 Crear Objetos COM

Para obtener un puntero a un objeto siempre se lo podemos pedir a otro objeto, pero llegara un momento en el que habrá que obtener el puntero al primer objeto, para esto está la librería COM que nos proporciona los funciones básicas para comenzar a usar los objetos.

7.2.2.6 La librería COM

Todos los sistemas que soporten COM deben incluir una implementación de la librería COM. Las funciones de esta librería son funciones normales no son métodos ni están agrupadas en interfaces, los nombres de estas funciones generalmente comienzan con la palabra Co-.[CHA1996]

7.2.2.6.1 Encontrar servidores

Cuando un cliente hace una petición a la librería COM para crear un objeto le pasa el identificador de clase que la función usará para localizar el servidor de esta clase de objetos. Para llevar a cabo esto existe un registro del sistema donde hay una entrada para cada CLSID indicando donde se encuentra el servidor y las clases de todos los objetos COM deben registrarse en este registro.

7.2.2.6.2 Clases e instancias

Crear un objeto es instanciar un objeto de una determinada clase, sin embargo COM no proporciona ningún medio de inicializar los objetos con determinados datos, para ellos debemos usar los servicios de los monikers.

7.2.2.7 Como crea un simple objeto

Para crear un objeto se llama a la función CoCreateInstance de la librería COM especificando la CLSID del objeto a crear y la IID de una interfaz que soporte el objeto. Entonces la función utiliza la CLSID para localizar dentro del registro del sistema la localización y tipo de servidor a ejecutar y lo ejecuta, este crea una instancia del objeto deseado y devuelve un puntero a la interfaz requerida, la librería COM devuelve este puntero como valor de retorno de la función CoCreateInstance. Este proceso se lleva a cabo independientemente del tipo de servidor el cliente no debe preocuparse en absoluto de eso. La Ilustración 7.26 muestra la secuencia de funcionamiento anterior.

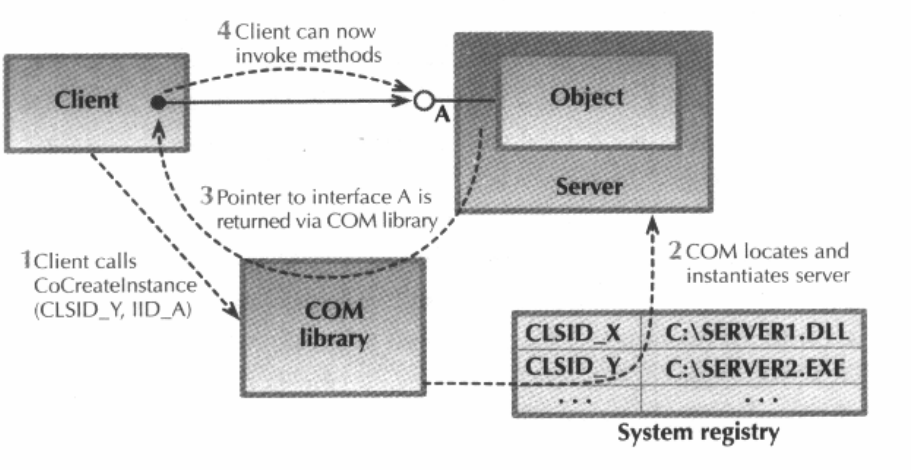


Ilustración 7.26: Creación de un objeto COM

Hay que destacar que este objeto no ha sido inicializado y normalmente el cliente debe pedir al objeto que se inicialice, aunque puede que no sea necesario.

7.2.2.8 Como crea múltiples objetos de la misma clase: Fábricas de objetos

Si un cliente necesita solo una instancia a un objeto particular la solución más simple es crear la una instancia con CoCreateInstance. Es posible sin embargo que un cliente necesite varias instancias de la misma clase, para crearlas de una forma eficiente el cliente puede acceder a la objeto fabrica que puede crear objetos de una determinada clase. Las fabricas de objetos son objetos COM, como cualquier otro, solo que están programadas para crear objetos de una determinada clase.

Lo cierto es que los objetos que se crean con CoCreateInstance se crean con una fábrica de objetos lo que ocurre es que esto es transparente al usuario pero en realidad se usan los métodos de la interfaz IclassFactory que implementan todos las fábricas de objetos y que pasamos a comentar.

7.2.2.8.1.1 La interfaz IclassFactory

Solo contiene dos métodos.

1. CreateInstance que crea una nueva instancia de la clase del objeto que la fábrica puede instancias. El cliente no necesita especificar la CLSID como parámetro ya que la clase viene implícita dentro del objeto fábrica, sin embargo si se especifica la IID de la interfaz y el tipo de servidor que se desea.
2. LockServer permite que un cliente mantenga el servidor en memoria, esto garantiza que el servidor se mantiene funcionando.

Hay otra versión de la interfaz IClassFactory que se denomina IclassFactory2 con esta se pueden crear objetos con licencias de uso.

7.2.2.8.1.2 Usar una fabrica de objetos

Para acceder a un objeto fábrica el cliente invoca a una función de la librería COM llamada CoGetClassObject. En esta llamada el cliente especifica la CLSID de la clase de objetos para la que desea la fábrica y no una CLSID de la clase del objeto fábrica. El cliente también especifica la IID de la interfaz que el cliente necesita usar para acceder a la fábrica, normalmente será la IID de IclassFactory. La Ilustración 7.27ilustra el proceso de crear una instancia con una fábrica de objetos.

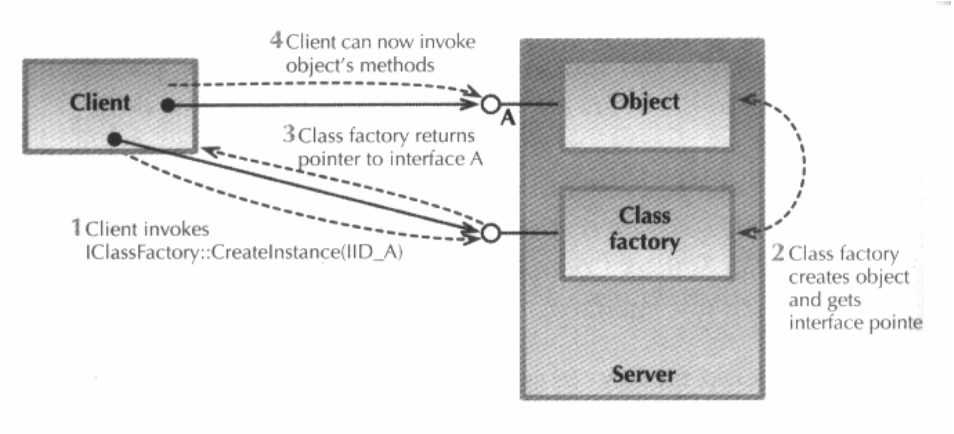


Ilustración 7.27: Fábrica de objetos COM

7.2.2.9 Emulación

Puede ocurrir que se sustituya una clase por otra con nuevas interfaces, entonces todos los objetos que utilizaban la antigua clase no funcionarán, para solucionar esto se crea lo que se denomina 'emulación' que es que mediante llamadas a la función CoTreatAsClass con dos parámetros uno el antiguo CLSID y el nuevo CLSID crea una relación de emulación dentro del registro de tal forma que cuando la aplicación que usaba la antigua clase quiera usarla, usará la nueva sin verse afectada. Ha de tenerse en cuenta que la nueva clase debe implementar todas las interfaces de la antigua clase.

Otro caso típico es cuando se pueden tener varias clases que implementa las mismas interfaces, en tal caso se debe definir una clase genérica que represente la funcionalidad de ambas dos y establecer emulación a la que más no interese, por ejemplo supongamos que tenemos clases traductoras C1 y C2 y tenemos un procesador de textos, sería conveniente definir una clase llamada por ejemplo CTraductora de tal forma que el procesador de textos use esta clase y mediante la emulación podremos seleccionar entre el traductor C1 o el C2.

7.2.2.10 Inicializar los objetos COM

Generalmente CoCreateInstance simplemente crea una instancia genérica de una clase pero sabemos que los objetos son más que simple métodos también están formados por datos.

En COM un cliente normalmente debe pedirle al objeto que se inicialice el mismo, para ello los datos deben ser almacenados de forma persistente. Uno de las primeras interfaces que se solicitan al crear un objeto son las que contienen los métodos para inicializar tales como IPersisFile, IPersistStorage o IPersistStream.

7.2.2.11 Reusar Objetos COM

Uno de los principales objetivos de COM es permitir el amplio reuso de forma efectiva de código ya existente. COM permite el reuso a través de dos mecanismos delegación y agregación. Se definen los objetos externos los objetos que reusan los servicios de los objetos internos.

7.2.2.12 Delegación

Con este mecanismo los objetos externos contienen métodos que llaman a los objetos internos directamente. La implementación de este mecanismo es muy sencilla no es necesario escribir el objeto interno de ninguna forma especial. Vea se la Ilustración 7.28.

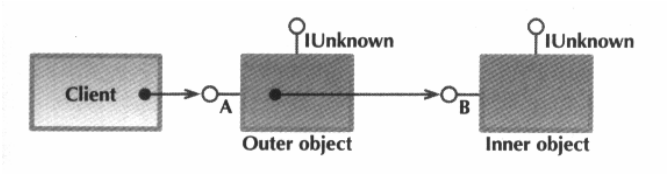


Ilustración 7.28: Delegación en COM

7.2.2.13 Agregación

Supongamos que simplemente queremos poner a disposición de nuestro cliente las interfaces del objeto interno además de las del objeto externo, con delegación deberíamos redefinir todas las interfaces del interno y para cada método simplemente hacer una rellamada al objeto interno, eso cuando se tienen varios objetos anidados resulta muy ineficiente, para solucionar esto se crea la agregación que simplemente expone directamente las interfaces del objeto interno en el externo, pero para esto el objeto interno debe estar preparado de una forma especial ya que cuando se llama a la interfaz IUnknown a partir de una interfaz del objeto interno, si este no está preparado devolverá un puntero a la interfaz del objeto interno, con lo que se el cliente ya no podría acceder al objeto externo o a sus interfaces. Vea se la Ilustración 7.29.

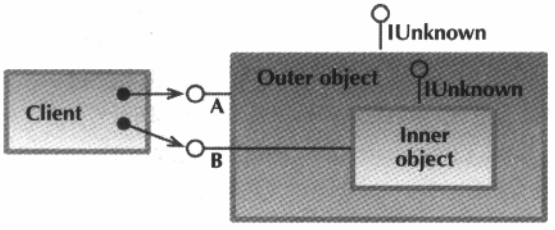


Ilustración 7.29: Abrogación en COM

7.2.3 Documentos Compuestos OLE

En un principio OLE fue una tecnología enfocada a resolver un único problema: la creación de documentos compuestos. Sin embargo el término OLE acabó siendo

asignado a cualquier cosa construida usando COM. Durante esta fase la tecnología original de documentos compuestos basados en COM fue conocido como Documentos OLE. Actualmente ambos términos y el uso de la palabra OLE para referirse a un rango de tecnologías basadas en COM esta obsoleta. El nombre ha vuelto a sus raíces, y OLE una vez mas hace referencia sólo a la tecnología de crear documentos compuestos.

Como este capítulo demuestra, la tecnología para documentos compuestos OLE utiliza muchos otras tecnologías basadas en COM como Almacenaje Estructurado, Monikers y Transferencia Uniforme de Datos. Mientras que el resultado final es un sencillo y natural interfaz de usuario conseguir esa simplicidad no es tan simple. Para comprender como es esta tecnología, es necesario primero examinarla desde el punto de vista del usuario. Una vez se tenga esto claro podremos ver los mecanismos que lo hacen posible.

7.2.3.1 Desde el punto de vista del Usuario

La Ilustración 7.11 ilustra un documento compuesto como lo ve un usuario. Una vez más el ejemplo muestra una Hoja de Calculo Microsoft dentro de un documento Word. Desde el punto de vista del usuario es un documento simple. En realidad aunque Word y Excel deben cooperar para presentar esta continuidad. Esta cooperación es debida, naturalmente, a que cada aplicación proporciona los apropiados objetos COM con sus interfaces estándares.

Cuando dos trozos de software se combinan para formar un documento compuesto, un actúa como contenedor y el otro como servidor. OLE permite al servidor o incrustar sus datos en el contenedor o vincular los.

Incrustar significa que los datos del servidor se almacenan en el mismo fichero que los datos del contenedor. Con vinculación, sin embargo, los datos del servidor quedan en su propio fichero; solo se almacena una referencia a los datos dentro del contenedor.

El programador del trozo de software decidió si el software va a actuar como contenedor, como servidor o como ambas cosas. Excel y Word, por ejemplo pueden actuar de ambas formas. En nuestro ejemplo de documento compuesto, Word actúa como el contenedor y Excel como el servidor. Pero esta organización puede ser a la inversa. Para hacer la situación incluso más complicada, un programador puede escribir una aplicación contenedora que permita al servidor solamente incrustar sus datos, solo vincular o ambas cosas. De la misma forma el creador del servidor puede hacer sus datos accesibles mediante vinculación o incrustación.

7.2.3.2 Creando un Documento Compuesto

En Windows y Windows Server un usuario tiene varias opciones para crear un documento compuesto. Para crear nuestro ejemplo el usuario podría abrir ambos documentos en Word y Excel en sus respectivas aplicaciones, copiar todo o parte de la hoja de cálculo en el portapapeles y usar pegar especial del Word para pegar el trozo de hoja copiada en el documento Word. El pegado especial de Word pregunta si se desea incrustar o vincular el documento.

Otra opción es usar el servicio arrastrar y soltar de COM. El usuario selecciona el trozo de hoja que desee y simplemente arrastra sobre el documento Word y soltar lo, automáticamente este se inserta dentro del contenedor.

7.2.3.3 Editando Documentos Compuestos

La edición de objetos incrustados se puede hacer en el contenedor o en la aplicación servidora.

7.2.3.4 Desde el punto de vista del Programador

Para el usuario, la tecnología OLE que crea documentos compuestos difumina la distinción entre aplicaciones diferentes, permitiendo una visión centrada sobre el documento. Un programador, sin embargo, sabe muy bien que el contenedor y el servidor son aplicaciones diferentes y que cada una tiene sus propias obligaciones.

7.2.3.5 Un inciso: Complejidad versus Herramientas

Antes de adentrarnos en una discusión más detallada de los documentos compuestos OLE, hay un importante punto del que hablar: no es tan difícil como parece. Es cierto que si cada programador de aplicaciones tuviese que escribir cada método en cada uno de los interfaces necesarios seria realmente difícil. Pero las herramientas de programación nos facilitan esta tarea. Mientras es interesante y útil sabes que es lo que realmente ocurre, el nivel de conocimiento necesario para el implementador es menor cada día y cada vez las herramientas de programación nos ayudan más.

7.2.3.6 Contenedores

Los ejemplos de contenedores más comunes son las aplicaciones independientes que el usuario puede ejecutar para trabajar con ellas como Word o Excel. Construir un contenedor OLE básico no es muy difícil - requiere solo dos interfaces descritos más adelante. Gracias a que las interfaces son completamente genéricas, un contenedor no necesita saber nunca que tipo de servidor le proporciona los datos a sus documentos. El contenedor simplemente soporta los interfaces estándar para contener, mientras el servidor tiene los interfaces estándares para servir. Ninguno de los dos se preocupa de que tipo de aplicación que es la otra. En nuestro ejemplo Word solo sabe que está interactuando con un servidor que cumple con las reglas OLE para servidores; no sabe ni tiene en cuenta si el servidor es Excel.

7.2.3.7 Servidores

Los servidores no son tan sencillos como los contenedores. Necesitan mas interfaces que estos y algunos de estas interfaces son un poco complicados. Los servidores vienen con diferentes configuraciones y el creador del servidor debe decidir cual es la correcta para su aplicación.

Para realizar un servidor hay que ver si lo hacemos in-process o como una aplicación independiente. Independientemente de la decisión que tomemos siempre debemos proporcionar un in-process parte para que interactúe con el contenedor ya que la aplicación servidora no puede acceder al documento contenedor a menos que sea un in-process.

7.2.3.8 Como cooperan el servidor y el contenedor

El problema principal es hace que los cambios del usuario se reflejen dentro del contenedor, para esto es necesario un in-process handles del servidor que esta dentro del contenedor y que se encargue de esto.

7.2.3.9 Como funciona la incrustación

Se crea un almacenamiento estructurado dentro del documento contenedor que llamada ObjectPool donde se almacenan todos los elementos incrustados para cada objeto se guarda sus datos, una copia de su representación y el CLSID de la aplicación servidora para cuando haya que editar los datos.

7.2.3.10 La copia de representación

La idea es que cuando en un documento haya algún objeto incrustado, no sea necesario llamar al servidor para que nos muestre una representación de este, sino que tenemos una copia de la representación del objeto cuando se le hizo la última modificación.

7.2.4 COM Distribuido

Desde el principio, COM fue diseñado para permitir la distribución, la habilidad de que un cliente cree e invoque métodos de un objeto en otra máquina. Esta promesa arquitectural fue cumplida con la realización de DCOM en 1996. Con DCOM un cliente puede crear y usar objetos en otros sistemas así como objetos ejecutando se localmente. Incluso mejor, el cliente no necesita distinguir los objetos locales de los remotos. De la misma forma que acceden transparentemente a objetos de una DLL o de un EXE. De echo lo más difícil de todo es permitir la transparencia a la hora de que un proceso use un método de un objeto ejecutando se en otro proceso dentro de la misma máquina. En este sentido, DCOM es una pequeña mejora del original COM.

La capacidad para iniciar e invocar objetos remotos es una importante ventaja pero no es suficiente. En particular, debe haber alguna forma de controlar quien tiene acceso a los objetos de una determinada máquina, dentro de una red que puede estar abarrotada de potenciales intrusos. Para hacer esto posible, DCOM se ha construido sobre un conjunto de servicios seguros. Las aplicaciones, incluso las que existían antes de DCOM, pueden usar DCOM y trabajar seguras sin necesidad de añadir ningún código para trabajar con seguridad. Por otro lado, las aplicaciones que necesiten explícitamente la seguridad de DCOM pueden usarla.

Aunque presenta algunas complicaciones, DCOM es en su mayor parte simple de entender. Añade solo tres elementos a los de COM: técnicas para crear y eliminar objetos, un protocolo para invocar esos métodos y mecanismos de seguridad para los objetos.

7.2.4.1 Crear un Objeto Remoto

Entre los más importantes servicios que COM proporciona están los que permiten crear objetos. Como este libro ha demostrado, los clientes generalmente crean objetos a través de llamadas a la librería de COM o usando monikers. Ambas

aproximaciones también funcionan con DCOM, aunque con unas pocas puntualizaciones como era de esperar. Esta sección las varias formas que un objeto cliente tiene para crear uno remoto.

7.2.5 Uso de CoCreateInstance

Independientemente de donde se ejecuta un objeto, un cliente típicamente lo crea y después adquiere un puntero a su interfaz ya que lo necesita para usarlo. Para la mayoría de los objetos descritos en este libro, estos están implementados en una DLL o en un servidor local, esto puede hacerse llamando a CoCreateInstance y luego usando QueryInterface para pedir el puntero a la interfaz que nos interese. Un cliente puede crear un objeto en un servidor remoto usando la misma llamada, el cliente incluso no necesita preocuparse de que el objeto esta ejecutando se en otra máquina simplemente pasando un CLSID junto con una IID.

Sin embargo para un objeto remoto, se debe especificar en que máquina debe ser creado el objeto. Como se describió anteriormente, para un objeto que se crea en la misma máquina que el cliente, el registro del sistema asigna a la CLSID el nombre de la DLL o EXE donde se implementa. Para un objeto creado en otra máquina el registro del sistema debe asignar el CLSID al nombre de la máquina donde se debe crear el objeto. Para la creación del objeto se contacta con la máquina, se busca en su registro la CLSID y el servidor se ejecuta en esa máquina. Si el objeto remoto esta implementado en una DLL, un proceso se ejecuta que carga la DLL y desaparece completamente. En otro caso, el proceso del objeto se inicia como en un servidor local. La Ilustración 7.25 muestra un esquema simplificado del uso de CoCreateInstance para crear un objeto remoto. En la Ilustración 7.26, el cliente pide a la librería COM crear un objeto con el CLSID X, junto con un puntero a su interfaz A. La entrada del registro para X en la máquina cliente contiene el nombre de la máquina. DCOM permite varias formas de identificar a un ordenador, dependiendo de qué protocolo de red se esté usando. DCOM permite los nombres de dominios de TCP/IP también valores de IP, nombres NetBIOS y nombres usados en las redes Novell. Cuando la máquina remota se ha contactado, el objeto se crea allí usando la información de X encontrada en el registro.

7.3 Servicios Web

En los últimos años, una tecnología ha cambiado de forma definitiva el panorama del desarrollo de aplicaciones más que ninguna otra: Internet. Las organizaciones dependen cada vez más de los recursos digitales y de los canales de comunicación que proporcionan Internet y las tecnologías relacionadas. Como consecuencia, en el diseño y desarrollo de la mayoría de las aplicaciones se tiene en cuenta cómo incorporar las tecnologías de Internet para aprovechar plenamente las ventajas de los sistemas conectados.

Cuando no pretenden desarrollar aplicaciones completamente nuevas, las organizaciones pueden considerar interesante la idea de agregar distintas aplicaciones tradicionales, orientadas a tareas en una sola aplicación compuesta. En ocasiones, esto implica integrar aplicaciones que se encuentran dentro de los límites de una entidad independiente, como otra compañía o un proveedor de servicios. Sin embargo, el dilema

es mayor cuando se intenta integrar aplicaciones antiguas que fueron creadas con diversas tecnologías, modelos de objetos, sistemas operativos y lenguajes de programación. ¿Cómo se puede conseguir que esta combinación funcione? La respuesta se encuentra en una Internet programable.

XML (eXtensible Markup Language, Lenguaje de marcado extensible), como formato abierto de descripción de datos, ha permitido el desarrollo de una Internet programable. Del mismo modo que TCP/IP proporcionó conectividad universal para Internet y HTML brindó un lenguaje estándar para presentar la información en una gran diversidad de plataformas para el uso humano, XML proporciona un lenguaje normalizado que permite intercambiar datos destinados a un consumo automatizado. Proporciona la posibilidad de representar datos en un formato ampliamente aceptado que hace posible que los equipos envíen y reciban datos en un estilo previsible, lo que posibilita una programación no limitada a sistemas cerrados y controlados. XML ofrece una mayor libertad, ya que su sencillez y capacidad de extensión permiten definir casi cualquier elemento, además de posibilitar la expansión. Una de las principales unidades de creación de la Internet programable son los Servicios Web XML.

7.3.1 Información general acerca de Servicios Web XML

Un servicio Web XML es una entidad programable que proporciona un elemento de funcionalidad determinado, como lógica de aplicación, al que se puede tener acceso desde diversos sistemas potencialmente distintos mediante estándares de Internet muy extendidos, como XML y HTTP. Los Servicios Web XML dependen en gran medida de la amplia aceptación de XML y otros estándares de Internet para crear una infraestructura que posibilite el funcionamiento conjunto de aplicaciones, de modo que se solucionen muchos de los problemas que antes dificultaban estos intentos.

Un servicio Web XML puede ser utilizado internamente por una aplicación o bien ser expuesto de forma externa en Internet por varias aplicaciones. Dado que a través de una interfaz estándar es posible el acceso a un servicio Web XML, éste permite el funcionamiento de una serie de sistemas heterogéneos como un conjunto integrado.

En vez de centrarse en las posibilidades genéricas de portabilidad del código, los Servicios Web XML proporcionan una solución viable para habilitar la interoperabilidad de datos y sistemas. Los Servicios Web XML utilizan mensajería basada en XML como medio fundamental de comunicación de datos, para contribuir a reducir las diferencias existentes entre entornos que utilizan distintos modelos de componentes, sistemas operativos y lenguajes de programación. Los programadores pueden crear aplicaciones que entrelacen Servicios Web XML de una diversidad de orígenes, de modo similar a como utilizan tradicionalmente componentes en la creación de aplicaciones distribuidas.

Una de las características básicas de un servicio Web XML es el alto grado de abstracción existente entre la implementación y el consumo de un servicio. Al utilizar mensajería basada en XML como mecanismo de creación y acceso al servicio, el cliente del Servicio Web XML y el proveedor de Servicios Web XML no necesitan más conocimiento mutuo que el relativo a las entradas, las salidas y la ubicación.

Los Servicios Web XML posibilitan una nueva era para el desarrollo de aplicaciones distribuidas. Quedan así relegadas las competiciones entre modelos de objetos y entre lenguajes de programación. Cuando los sistemas se acoplan estrechamente mediante infraestructuras patentadas, esto se realiza a expensas de la interoperabilidad entre las aplicaciones. Los Servicios Web XML ofrecen interoperabilidad en un nivel completamente nuevo que no deja lugar para esas rivalidades contraproducentes. Como siguiente avance revolucionario de Internet, los Servicios Web XML se convierten en la estructura fundamental que vincula a todos los equipos y dispositivos.

7.3.2 Infraestructura de Servicios Web XML

Los Servicios Web XML deben ser independientes en lo que respecta a la selección de sistema operativo, modelo de objetos y lenguaje de programación con el fin de funcionar correctamente en la heterogeneidad del Web. Asimismo, para facilitar una adopción de los Servicios Web XML tan generalizada como en el caso de otras tecnologías, estos servicios deben caracterizarse por lo siguiente:

Correspondencia imprecisa: se considera que dos sistemas mantienen correspondencia imprecisa si la única condición impuesta a ambos consiste en comprender los mensajes de texto autodescriptivos mencionados con antelación. Por su parte, los sistemas que mantienen una correspondencia precisa imponen una notable carga de personalización para habilitar la comunicación y precisan una mejor comprensión entre los sistemas.

Comunicación ubicua: es poco probable que alguien cree, ahora o en el futuro, un sistema operativo que no incorpore la posibilidad de conexión a Internet, que por tanto equivale a un canal de comunicación ubicua. La posibilidad de conectar casi cualquier sistema o dispositivo a Internet garantizará que esos sistemas y dispositivos estén disponibles para cualquier otro sistema o dispositivo conectado a Internet.

Formato de datos universal: mediante la adopción de estándares abiertos sobre métodos de comunicaciones patentados de bucle cerrado, cualquier sistema compatible con esos mismos estándares abiertos puede comprender los Servicios Web XML. El uso de mensajes de texto autodescriptivos que pueden compartir los Servicios Web XML y sus clientes sin necesidad de conocer los sistemas subyacentes permite la comunicación entre sistemas autónomos y heterogéneos. Los Servicios Web XML obtienen esta capacidad mediante XML.

Los Servicios Web XML emplean una infraestructura que proporciona lo siguiente: un mecanismo de descubrimiento para localizar Servicios Web XML, una descripción de servicio para definir cómo se deben utilizar esos servicios y formatos de conexión estándar para la comunicación. En la siguiente ilustración se muestra un ejemplo de esta infraestructura.

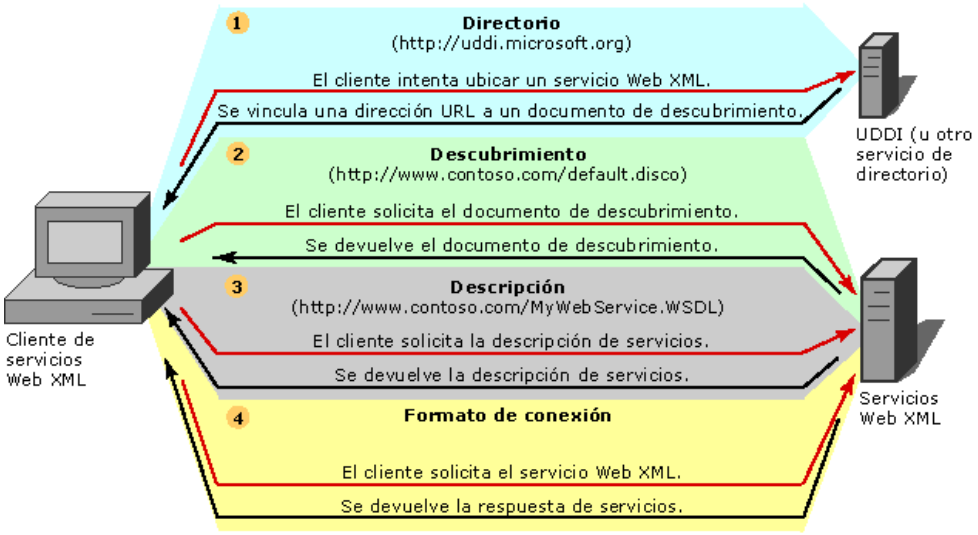


Ilustración 7.30: Infraestructura de Servicios Web XML

7.3.3 Funciones de las partes de la infraestructura

Directorios de Servicios Web XML: Al igual que con cualquier otro recurso de Internet, no se puede encontrar un Servicio Web XML en particular sin ayuda de algún medio que permita buscarlo. Los directorios de Servicios Web XML proporcionan ubicaciones centralizadas en las que los proveedores de estos servicios pueden publicar información acerca de los servicios de que disponen. Esos directorios pueden ser incluso Servicios Web XML, a los que se pueda tener acceso mediante programación y que proporcionen resultados de búsqueda en respuesta a consultas realizadas desde clientes potenciales de Servicios Web XML. Quizá sea necesario utilizar un directorio de Servicios Web XML para localizar una organización que proporcione un servicio con una función determinada o para especificar qué Servicios Web XML ofrece una organización concreta.

Las especificaciones UDDI (Universal Description, Discovery, and Integration, Descripción, Descubrimiento e integración universales) definen un medio estándar para publicar y descubrir información acerca de los Servicios Web XML. Los esquemas XML asociados con UDDI definen cuatro tipos de información que permitirían a un programador utilizar un servicio Web XML publicado: información comercial, información de servicios, información de enlace e información acerca de especificaciones para servicios.

Como componente básico del proyecto UDDI, el registro de empresas UDDI permite a las compañías localizar mediante programación datos acerca de Servicios Web XML expuestos por otras organizaciones. Los programadores pueden utilizar el registro comercial UDDI para localizar documentos de descubrimiento y descripciones de servicio. Para obtener más información, vea el sitio Web de UDDI (http://uddi.microsoft.com).

Descubrimiento de Servicios Web XML: El descubrimiento de Servicios Web XML es el proceso consistente en localizar, o descubrir, uno o varios documentos relacionados que describen un Servicio Web XML determinado mediante Lenguaje de descripción de Servicios Web (WSDL). A través del proceso de descubrimiento, los clientes de Servicios Web XML conocen la existencia de un Servicio Web XML y dónde encontrar el documento de descripción del mismo.

Un archivo .disco publicado, que es un documento XML donde se incluyen vínculos a otros recursos que describen el Servicio Web XML, permite descubrir mediante programación un Servicio Web XML. A continuación se muestra un ejemplo de la estructura de un documento de descubrimiento:

```
<?xml version="1.0" encoding="utf-8" ?>
<discovery xmlns:xsd="http://www.w3.org/2001/XMLSchema"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns="http://schemas.xmlsoap.org/disco/">
  <contractRef ref="http://www.contoso.com/Counter.asmx?wsdl"
               docRef="http://www.contoso.com/Counter.asmx"
               xmlns="http://schemas.xmlsoap.org/disco/scl/" />
    <soap address="http://www.contoso.com/Counter.asmx"
        xmlns:q1="http://tempuri.org/"
        binding="q1:CounterSoap"
        xmlns="http://schemas.xmlsoap.org/disco/soap/" />
</discovery>
```

El documento de descubrimiento es un contenedor de elementos que suelen incluir vínculos (direcciones URL) a recursos que proporcionan información de descubrimiento para un Servicio Web XML. Si las direcciones URL son relativas, se supone que lo son con respecto a la ubicación del documento de descubrimiento.

Sin embargo, un sitio Web que implementa un Servicio Web XML no necesariamente tiene que permitir el descubrimiento. Otro sitio podría encargarse de la descripción del servicio; por ejemplo, un directorio de Servicios Web XML. También existe de la posibilidad de que no haya un medio público para buscar el servicio; por ejemplo, cuando se crea para uso privado.

Descripción de Servicios Web XML: La infraestructura de Servicios Web XML se fundamenta en la comunicación por medio de mensajes basados en XML que cumplen con una descripción de servicio publicada. La descripción del servicio es un documento XML escrito en una gramática XML denominada WSDL (Lenguaje de descripción de Servicios Web) que define el formato de mensaje comprensible para el Servicio Web XML. La descripción del servicio sirve como acuerdo que define el comportamiento de un Servicio Web XML e indica a los clientes potenciales cómo interactuar con él. El comportamiento de un Servicio Web XML está determinado por modelos de mensajería que el servicio define y admite. Estos modelos dictan conceptualmente lo que el consumidor del servicio puede esperar que ocurra cuando se envía un mensaje con formato correcto al Servicio Web XML.

Por ejemplo, el modelo de solicitud y respuesta asociado a un servicio del estilo de RPC (Remote Procedure Call, llamada a procedimiento remoto) definiría qué esquema de mensajes SOAP debe utilizarse para invocar un método en concreto. Este modelo determinaría igualmente el formato que debería seguir el mensaje SOAP de respuesta.

Otro ejemplo de modelo de mensajería, representa interacciones unidireccionales. Este modelo se emplea cuando una comunicación se va a establecer en un solo sentido. En esta situación, el emisor no recibirá ningún mensaje del Servicio Web XML, ni siquiera los mensajes de error. Existe una limitación cuando se establece una comunicación unidireccional mediante un protocolo que es tradicionalmente de solicitud y respuesta, donde se puede devolver un mensaje de error.

Los esquemas que definen los formatos de mensajes SOAP pueden definirse internamente para la descripción del servicio o bien externamente e importarse en la descripción.

Además de las definiciones de formato de mensaje y los modelos de mensajería, la descripción del servicio puede contener la dirección asociada a cada punto de entrada del Servicio Web XML. El formato de esta dirección se ajusta al protocolo utilizado para el acceso al servicio, como una dirección URL para HTTP o una dirección de correo electrónico para SMTP.

Para obtener información acerca de la especificación WSDL, vea el sitio Web del W3C (World Wide Web Consortium), http://www.w3.org/TR/wsdl.

Formatos de conexión de Servicios Web XML: Los protocolos binarios como DCOM se componen de un nivel de solicitudes de método sobre un protocolo de comunicaciones patentado. Estos protocolos no son convenientes para la creación de Servicios Web XML que puedan estar disponibles universalmente. Esto no le impide utilizarlos en un escenario de Servicios Web XML, pero el inconveniente que supone es que dependen de las arquitecturas específicas de sus sistemas subyacentes y, por lo tanto, limitan el espectro de posibles clientes.

Como alternativa, puede construir Servicios Web XML que trabajen con uno o varios protocolos abiertos, como una combinación de HTTP y SOAP. Lógicamente, la infraestructura necesaria para admitir diferentes protocolos variará.

Los Servicios Web XML no se limitan a proporcionar acceso de llamada a procedimientos remotos (RPC). También se pueden crear para intercambiar información estructurada, como órdenes de compra y facturas, y se pueden utilizar para automatizar y conectar procesos comerciales internos y externos.

HTTP-GET y HTTP-POST son protocolos estándar que utilizan verbos de HTTP (Hypertext Transfer Protocol, Protocolo de transferencia de hipertexto) para la codificación y el análisis de parámetros como pares de nombre y valor, junto con la semántica de solicitudes asociada. Cada uno se compone de una serie de encabezados de solicitud HTTP que, entre otras características, definen lo que el cliente solicita al servidor, el cual responde con una serie de encabezados de respuesta HTTP y los datos solicitados, si procede.

HTTP-GET pasa sus parámetros en forma de texto no codificado mediante el tipo aplicación MIME/x-www-form-urlencoded, que se anexa a la dirección URL del servidor que trata la solicitud. Urlencoding es una forma de codificación de caracteres que garantiza que los parámetros que se pasan consten de texto con un formato compatible; por ejemplo, codificar un espacio como %20. También se hace referencia a los parámetros anexados como cadena de consulta.

Al igual que HTTP-GET, los parámetros HTTP-POST también se codifican con Urlencoding. Sin embargo, en vez de pasarse como parte de la dirección URL, los pares de nombre y valor se pasan dentro del mensaje de solicitud HTTP.

SOAP es un protocolo simple y ligero basado en XML para el intercambio en el Web de información estructurada de tipos. El objetivo general del diseño de SOAP consiste en conseguir la mayor simplificación posible y proporciona un mínimo de funcionalidad. El protocolo define un marco para la mensajería que no contiene semántica de aplicaciones ni transporte. Como resultado, el protocolo es modular y muy extensible.

Al viajar sobre protocolos de transporte estándar, SOAP puede aprovechar la arquitectura abierta de Internet existente y obtener fácilmente la aceptación por parte de cualquier sistema arbitrario compatible con los estándares de Internet más básicos. Podría interpretarse que la infraestructura necesaria para un Servicio Web XML compatible con SOAP es más bien simple, aunque muy eficaz, ya que agrega relativamente poco a la infraestructura de Internet y aún así facilita el acceso universal a los servicios creados con SOAP.

La especificación del protocolo SOAP se compone de cuatro partes principales. La primera define un sobre extensible y obligatorio para encapsular los datos. El sobre SOAP define un mensaje SOAP y constituye la unidad básica de intercambio ente procesadores de mensajes SOAP. Ésta es la única parte obligatoria de la especificación.

La segunda parte de la especificación del protocolo SOAP define reglas de codificación de datos opcionales para representar tipos de datos definidos por cada aplicación y gráficos dirigidos, así como un modelo uniforme para serializar modelos de datos no sintácticos.

En la tercera parte se define un modelo de intercambio de mensajes (de solicitud y respuesta) del estilo de RPC. Cada mensaje SOAP es una transmisión en un sentido. Aunque las raíces de SOAP se encuentran en RPC, no se limita a un mecanismo de solicitud y respuesta. Los Servicios Web XML suelen combinar mensajes SOAP para implementar esos modelos, pero SOAP no impone un modelo de intercambio de mensajes y esta parte de la especificación también es opcional.

En la cuarta parte de la especificación se define un enlace entre SOAP y HTTP. Sin embargo, también es opcional. Se puede utilizar SOAP en combinación con cualquier protocolo de transporte o mecanismo que pueda transportar el sobre SOAP, incluido SMTP, FTP o un disco.

Para obtener información acerca de la especificación SOAP, vea el sitio Web del W3C (World Wide Web Consortium), http://www.w3.org/TR/soap.

7.3.4 Ciclo de vida de los Servicios Web XML

El proceso que se produce cuando se realiza una llamada a un Servicio Web XML es similar al que tiene lugar cuando se llama a un método regular. La diferencia principal es que, en vez de llamar a un método que se encuentra en la aplicación cliente, se genera un mensaje de solicitud sobre el transporte especificado, como HTTP. Dado que el método del Servicio Web XML se puede encontrar en un equipo diferente, la

información que tiene que procesar el Servicio Web XML debe pasar por la red al servidor que aloja el Servicio Web XML. El Servicio Web XML procesa la información y devuelve el resultado, a través de la red, a la aplicación cliente.

En la ilustración siguiente se muestra el proceso de comunicación entre un cliente y un Servicio Web XML.

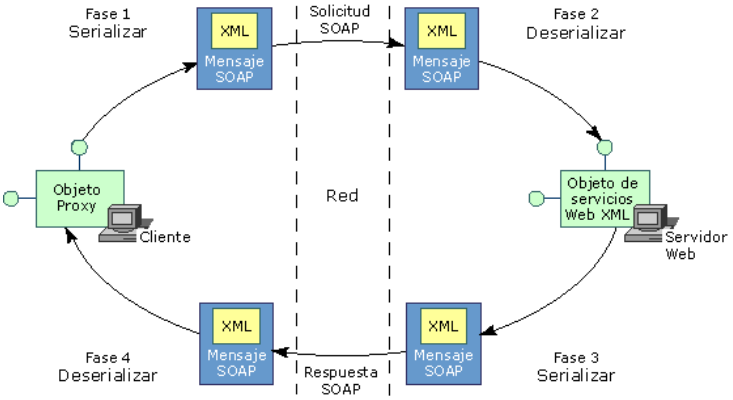


Ilustración 7.31: Ciclo de vida de los Servicios Web XML

A continuación se describe la secuencia de eventos que se producen cuando se llama a un Servicio Web XML:

1. El cliente crea una nueva instancia de una clase de proxy de Servicio Web XML. Este objeto reside en el mismo equipo que el cliente.

2. El cliente invoca un método en la clase de proxy.

3. La infraestructura del equipo cliente serializa los argumentos del método del Servicio Web XML en un mensaje SOAP y lo envía a través de la red al Servicio Web XML.

4. La infraestructura recibe el mensaje SOAP y deserializa el XML. Crea una instancia de la clase que implementa el Servicio Web XML e invoca el método de Servicio Web XML; el XML deserializado se pasa como argumentos.

5. El método de Servicio Web XML ejecuta el código y establece finalmente el valor devuelto y los parámetros out.

6. La infraestructura del servidor Web serializa en un mensaje SOAP el valor devuelto y los parámetros de salida, y lo devuelve al cliente a través de la red.

7. La infraestructura de Servicios Web XML, en el equipo cliente, recibe el mensaje SOAP, deserializa el XML en el valor devuelto y los parámetros out, y los pasa a la instancia de la clase de proxy.

8. El cliente recibe el valor devuelto y los parámetros out.

Actualmente los Servicios Web están siendo utilizados para adaptarse a la tecnología de computación masiva del Grid [Midliardi01], de igual modo, prácticamente todos los lenguajes de programación disponen de un API para la gestión de los mismos, como Java Axis, Phiton, Perls y todos los lenguajes adaptados al framework .NET.

7.4 Características comunes

El objetivo de análisis detallado de las plataformas middleware LII anteriores, es el de poder extraer estructuras comunes a todas, de tal forma que sea posible, abstraer su comportamiento y de esta forma homogeneizarlo para así poder construir un sistema que permita la comunicación entre todos ellos.

Dichas estructuras comunes que caracterizan todas las plataformas anteriormente estudiadas son:

- Servicio de nombres y repositorios, de esta forma, en todas las plataformas se pueden localizar los elementos del sistema a través de alguna técnica de nombrado, ya sea DNS, registro del sistema, repositorio de objetos, directorio de objetos tipo LDAP.

- Mecanismos de introspección de estructuras internas a las plataformas, ya sean descripción de tipos, descripción de interfaces, utilización de lenguajes de descripción de interfaces, etc. de tal manera que podemos obtener información descriptiva de los servicios a utilizar.

- Técnicas de invocación dinámica, de modo que se definen protocolos de comunicación dinámica del tipo DII, invocación dinámica, o protocolo de invocación remota.

En próximos capítulos veremos que existe un isomorfismo entre estas características comunes y las estructuras esenciales identificadas en las máquinas abstractas, con lo que resultará relativamente sencillo construir un nuestro middlebus.

A continuación expondremos, clasificaremos y compararemos las diversas técnicas de tratamiento de la heterogeneidad a nivel de sistemas distribuidos que actualmente están siendo utilizadas, investigadas y desarrolladas.

Capítulo 8:
Heterogeneidad en sistemas distribuidos

Para estudiar la panorámica de soluciones actuales que pretenden resolver el problema de la interoperabilidad heterogénea, vamos a definir varias categorías de soluciones, siguiendo la analogía con las soluciones de interconexión a nivel de red enumeradas en las siguientes secciones. Posteriormente realizaremos una comparativa y estudiaremos las características y deficiencias de cada uno de ellos, para de esta forma identificar los rasgos que deseamos tengan los sistemas creados con nuestro modelo arquitectónico.

8.1 Tipos de interoperabilidad entre plataformas

Heterogeneidad es la propiedad que tienen los elementos que permite la variedad y la diferencia. Estos elementos pueden ser:

- Redes
- Hardware de ordenadores
- Sistemas Operativos
- Lenguajes de programación
- Implementaciones de diferentes desarrolladores

Uno de los retos con los que se enfrentan todos los sistemas distribuidos es la heterogeneidad. Vemos que Internet es un claro ejemplo de éxito en la resolución de este problema, a nivel de protocolos de red en las capas de red y transporte. Internet

enmascara todas la diferencias tanto de las redes interconectadas entre si, que pueden ser: Ethernet, ATM, líneas telefónicas, cable-modem, H25, RS232; así como las diferencias aportadas por diversas plataformas hardware y software. De esta forma, dos sistemas completamente diferentes pueden comunicarse sobre Internet. Por ejemplo, un teléfono móvil puede acceder a las imágenes de una cámara IP a través del protocolo HTTP sobre Internet. Este éxito es debido a que se ha conseguido una globalización de la estandarización de los protocolos de Internet comenzando por TCP/IP, seguida por sus protocolos de aplicación más importantes HTTP, FTP, SMTP, POP, IMAP, TELNET.

Java es otro ejemplo de éxito en el problema de la construcción de código portable a plataformas heterogéneas. A través del concepto de máquina virtual, se pueden construir programas para cualquier plataforma de ordenadores. Se compila, una vez y se ejecuta en cualquier sitio. Si bien esta idea viene sujeta a un lenguaje específico como es Java, en este punto la plataforma .NET, pretende solucionar el problema de la coexistencia de varios lenguajes diferentes y la portabilidad de código definiendo una máquina virtual CLR, pero en este caso con una arquitectura independiente de cualquier lenguaje de programación. En las próximas secciones veremos que se puede ver el CLR como un bus para la interoperabilidad de lenguajes.

Sin embargo, a otros niveles, por ejemplo, a nivel de enlace o a nivel físico de red, no se ha conseguido definir un estándar global. Aunque muchos son los motivos, ya que no existe un medio de transferencia físico universal que permita rapidez, economía, y simplicidad. Disponemos de líneas de fibra óptica, cables telefónicos analógicos, cables telefónicos digitales, cables analógicos coaxiales, conexiones inalámbricas por radio, por infrarrojos, etc. A nivel de enlace, tampoco existen protocolos ideales, se dispone de Ethernet, con sus sabores 10BT, 10B2, 10B5, 100BT, GIGAEthernet, redes TokenRing, redes ATM que permiten el control de caudal y enrutamiento, RS232, el protocolo H25, protocolo FrameRelay, para redes inalámbricas los más populares son Wi-fi y Bluetooth. A este nivel para solucionar la heterogeneidad se han construido una gran variedad de mecanismos como son:

- Adaptadores, suelen ser muy simples y se utilizan principalmente en el nivel físico, por ejemplo, si tenemos una tarjeta de red 10BT y queremos conectarla a una red Wi-fi, podemos conectarle un adaptador que convierta la señal 10BT del cable en señal de radio Wi-fi. Un ejemplo más sencillo es la conversión de una señal RS232 en señal infrarrojos.

- Puentes (Bridges), estos dispositivos nos permiten comunicar dos redes heterogéneas a nivel de enlace. Por ejemplo, si disponemos de una red Ethernet y una Tokenring, con un puente pueden comunicarse ambas. Igualmente, podemos utilizar un puente para conectar dos redes Ethernet a través de una línea ATM.

- Buses y hubs o concentradores, nos permiten distribuir las señales a nivel de enlace entre diferentes dispositivos con un mismo protocolo de red. Así, existen hubs para Ethernet o para TokenRing. El problema de los buses es que cuando un recurso está emitiendo el resto sólo puede escuchar, es decir sólo un recurso puede enviar a la vez, el resto debe esperar a que este acabe para realizar sus envíos.

- Conmutadores (Swiches) son dispositivos inteligentes diseñados para realizar las tareas de un bus, pero permitiendo que varios recursos emitan a la vez, esto lo hace realizando conmutaciones directas entre el emisor y sus receptores.

- Túneles, son protocolos con el PTPP o en LL2 para la creación de redes privadas virtuales, permiten encapsular un protocolo dentro de otro o incluso el mismo protocolo con diferentes valores dentro de otro. De tal forma que se pueden hacer túneles (tunneling) de protocolos de enlace como Ethernet sobre Internet o túneles de protocolos de Internet sobre el propio Internet, este es el caso de comunicar dos Intranet separadas, con IPs locales a través de Internet con IPs públicas. El objetivo es proporcionar una línea de comunicación virtual sobre una red existente y dentro de esa línea transmitir la información en un protocolo que puede ser diferente al de la red sobre la que se envían los paquetes reales.

Middleware es un término que se aplica al estrato software que provee una abstracción de programación, así como un enmascaramiento de la heterogeneidad subyacente de las redes hardware, sistema operativo y lenguaje de programación. Entre ellas tenemos CORBA, RMI, DSOM, DCOM, SOAP. Proporcionando así, un modelo computacional uniforme al alcance de los programadores de servidores y aplicaciones distribuidas. Los posibles modelos incluyen, como ya hemos visto en secciones anteriores, invocación sobre objetos remotos, notificación de eventos remotos, acceso remoto mediante SQL y procesamiento distribuido de transacciones. Como se indicó en la sección 6.2.11, algunos autores han observado que todos los problemas de heterogeneidad que se pretenden solucionar con el middleware pasa a este nivel, de tal forma que ahora la problemática de la interoperabilidad surge entre los diferentes middleware, además de aparecer el problema de la no completitud, es decir, que las especificaciones e implementaciones incluso de un mismo middleware no parecen ser compatibles.

A continuación examinaremos las diversas soluciones que se pretenden dar al problema de la interoperabilidad entre sistemas programables heterogéneos, incluyendo los middleware, realizando una clasificación en función de los mecanismos utilizados para resolverlas. Dicha clasificación se basará en la analogía que existe entre la resolución del problema de la heterogeneidad a nivel de redes comentara anteriormente y a nivel de sistemas software heterogéneos. De tal forma que definiremos cinco categorías: adaptadores, puentes, buses, conmutadores y túneles. Si bien alguna de las propuestas encajan en varias de las categorías incluso a diferentes niveles de comunicación y otras no encajan en ninguno.

8.2 Panorámica de soluciones a la heterogeneidad

Seguidamente se enumeran un conjunto de sistemas empleados tanto a nivel científico, como comercial, para resolver diversas facetas de la heterogeneidad en aplicaciones distribuidas. Iremos clasificando dichos sistemas al mismo tiempo que justificamos la clasificación y comentamos los rasgos que se consideran más representativos de dichos sistemas. En la siguiente sección se realizará una comparativa de dichos características.

8.2.1 Adaptadores

Consideraremos adaptadores como aquellos que aportan una solución puntual a un determinado lenguaje o sistema, para que interopere con otra plataforma concreta. Son soluciones, poco flexibles, y no profundizaremos en su estudio debido a que no nos aportan gran información.

Existen varios niveles de adaptación, por ejemplo a nivel de leguaje de programación podemos hablar de el Java Native Methods Interfaces de Java que permite la interoperabilidad entre Java y lenguajes de la plaforma nativa, Cecil de Eiffel que permite la interoperabilidad del lenguaje Eiffel con los objetos DCOM, Axis de Java que permite la interoperabilidad entre Java y los Servicios Web, Smalltalk también lo permite a través de su proyecto WinInt, y otros que permiten realizar llamadas especificas a las plataformas.

El concepto de adaptador también se utiliza en buses universales cuando se conectan a través de sus mecanismos plug-ing software que permite la interoperabilidad con el sistema de integración del bus, ver sección 6.3.12.

8.2.2 Puentes

Permiten interoperabilidad bidireccional entre dos plataformas diferentes. Se caracteriza por su bidireccionalidad y por ser siempre dos a dos. A continuación exponemos dos puentes, uno entre dos plataformas middleware ActiveX y CORBAplus y otro entre a nivel de máquinas virtuales entre la máquina de Java y la de .NET.

8.2.2.1 Expersoft's CORBAplus, ActiveX Bridge

CORBAplus, ActiveX Bridge [Expersoft053] de Expersoft, es una solución propietaria para permitir obtener todas las ventajas de la arquitectura distribuida de CORBA mientras se mantiene compatibilidad con los objetos ActiveX de Windows. Este puente cumple completamente con las especificaciones COM-CORBA de la OMG. Utiliza la tecnología de Object Bridge de Actional para proporcionar una interoperabilidad bidireccional completa entre la tecnología ActiveX de Microsoft (COM, DCOM, COM+, Automation, Dual Interface) y los sistemas de objetos de CORBA. Se utiliza el protocolo IIOP para la comunicación de las aplicaciones distribuidas. Esta herramienta también proporciona herramientas visuales para interactuar con el Repositorio de Interfaces de CORBA, de tal forma que resulte fácil ojear, seleccionar y editar los componentes del repositorio y generar stubs, skeletons, así como puentes estándar COM/CORBA. Se utiliza una técnica de puenteado dinámico de alta velocidad, para ello emplean técnicas de compilación JIT, con esta técnica se consigue un alto rendimiento. También se incluye un explorador de servicios de nombres de CORBA, de esta forma se pueden añadir, de una manera sencilla, nombres a objetos y contextos CORBA. Por último incluye una herramienta de administración visual que permite exponer instancias, fábricas e interfaces de CORBA a ActiveX y viceversa. Permite seleccionar y registrar objetos ActiveX y CORBA tan solo con hacer un clic.

8.2.2.2 JNBridgePro

JNBridgePro [Jnbrid04], es un puente a nivel de máquina virtual, utiliza un motor de invocaciones y generadores de envoltorios para ambas plataformas. Entre otras características podemos:

- Acceder a objetos y clases Java desde .NET y acceder a objetos y clases .NET desde Java.

- Crear objetos, invocar métodos, acceder a objetos y devolver objetos.

- Capturar excepciones y arrojar excepciones entre ambas plataformas.

- Extender las clases de Java extendiendo clases escritas en lenguajes soportados por .NET y viceversa.

- Interoperar con librerías de clases compiladas en Java, o con ensamblados de .NET.

- El código Java y .NET puede estar ejecutándose en: el mismo proceso, utilizando memoria compartida, en la misma máquina pero en diferente proceso o incluso en diferentes máquinas conectadas a través de red.

A continuación se muestra un ejemplo de cómo JNBridgePro trabaja, éste permite que una aplicación Swing de Java utilice ventanas Froms de .NET.

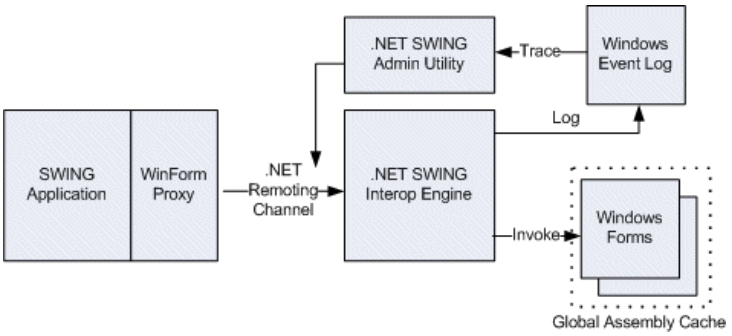


Ilustración 8.1: Ejemplo JNBridgePro

La aplicación Swing utiliza un envoltorio (Proxy) para construir los mensajes enviados a través del canal de Remoting .NET, al motor de interoperabilidad .NET para Swing. Dicho envoltorio se debe crear a mano con una herramienta denominada Ja.NET que está incluida dentro de JNBridge.

Podemos observar que es la típica aplicación de puente entre dos plataformas a nivel de máquinas virtuales, con este tipo de puentes, se obtienen una gran integración entre ambas plataformas así como un alto nivel de rendimiento. Sin embargo es necesario un alto grado de intervención de los desarrolladores, los envoltorios no se generan automáticamente y no están abiertos a otras plataformas.

8.2.3 Conmutadores

Esta categoría difiere con su análoga en redes, consideraremos sistemas conmutadores como aquellos que permiten una comunicación unidireccional entre una plataforma y varias. Suele ser una comunicación de uno a muchos unidireccional. Expondremos varios sistemas, tanto comerciales como científicos.

8.2.3.1 ReMMoC

ReMMoC [Grace03] es una arquitectura dinámica que pretende resolver el problema de heterogeneidad resultante de los diferentes paradigmas de comunicación: Invocación Remota, Publicación-Suscripción, Compartición de datos, Agentes móviles y Espacio de tuplas. Para ello esta arquitectura utiliza el concepto de IDL utilizado en Servicios Web, WSDL, como mecanismos reflectivo para proporcionar una abstracción de alto nivel que permita realizar las reconfiguraciones necesarias entre las diferentes implementaciones de diferentes paradigmas, permitiendo así su interoperabilidad heterogénea.

Esta arquitectura dinámica está formada por dos marcos (framework) de componentes, dentro de la plataforma de componentes OpenCOM [Lanc01]: Binding CF, para interoperar con los servicios implementados en diferentes middleware y Service Discovery CF para localizar diversos servicios contenidos en diferentes protocolos de localización de servicios.

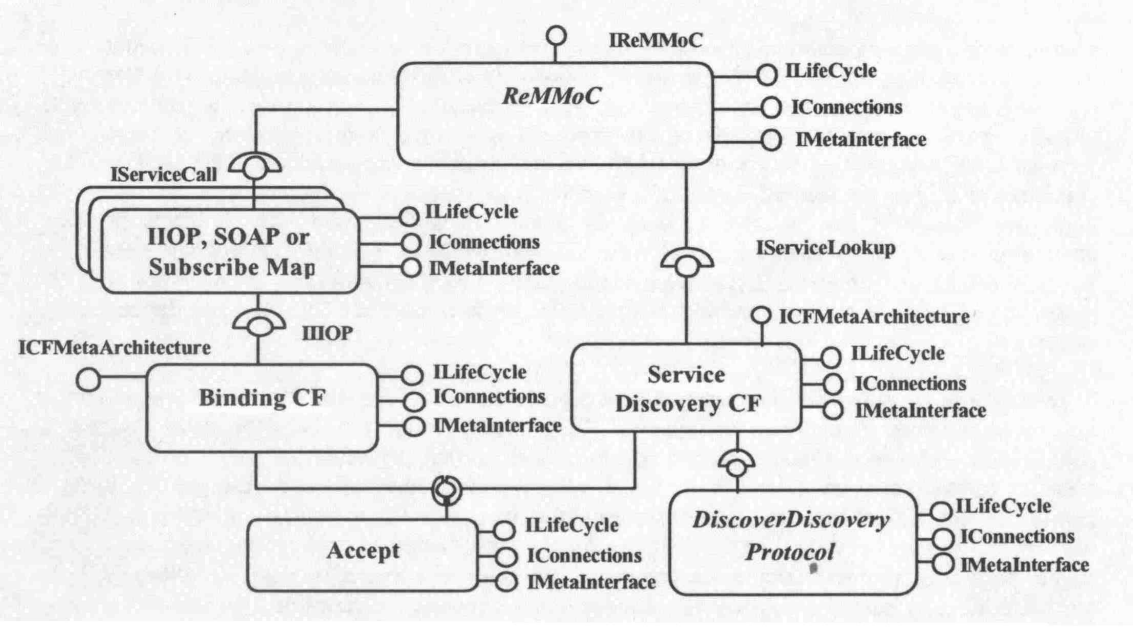


Ilustración 8.2: Modelo ReMMoC

Para cada plataforma middleware debe existir un componente con el cual interactuará Binding CF de tal forma que este puede configurarse para una plataforma de invocación

dinámica como puede ser IIOP, SOAP o DCOM o para otro paradigma diferente como suscripción-publicación como puede ser InfoBus. Todo estos middlewares se accederá de forma unidireccional y siempre a través de Binding CF.

La interfaz principal de ReMMoC viene dada por las siguientes operaciones:

```
Interface ReMMoc_ICF:IUnknown{
HRESULT WSDLGet(
      WSDLService* ServiceDescription,
      char*             XML
);
HRESULT FindandInvokeOperation(
      WSDLService ServiceDescription,
      ServiceReturnEvent ReturnedLookupEvent,
      char* OperationName,
      int Iterations,
      ReMMoCOPHandler Handler
);
HRESULT AddMessageValue(
      WSDLService *ServiceDescription,
      char *OperationName,
      char* elementName,
      ReMMoC_TYPE type,
      char* direction,
      VARIANT value
);
HRESULT GetMessageValue(
      WSDLService *ServiceDescription,
      char* OperationName,
      char* ElementName,
      ReMMoC_TPE type,
      char* direction,
      VARIANT value
);
}
```

Estas operaciones las invocarán los clientes a través de Servicios Web y vienen documentadas como WSDL. Las operaciones vienen definidas de cuatro tipos y la asignación de llamadas a estas operaciones con respecto al tipo de paradigma utilizado podría ser como se muestra en la figura.

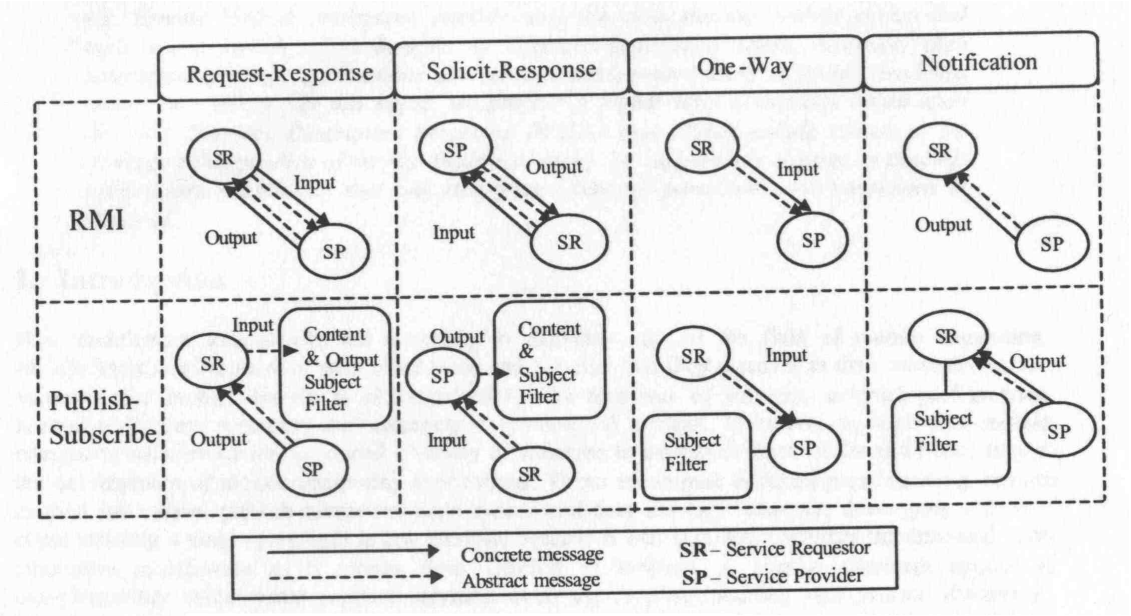


Ilustración 8.3: Mapeo de WSDL a diferentes middleware

Este servicio tiene la gran ventaja de permitir que una aplicación interopere con diferentes paradigmas de comunicación, pero los inconvenientes de su unidireccionalidad y que los clientes deben estar escritos explícitamente para este entorno.

8.2.3.2 RMIX

RMIX [Kursz03] establece una extensión de la API común para la invocación de métodos remotamente, esta es la API RMI de Java. Define un RMI extendido, RMIX que permite realizar invocaciones a diferentes plataformas CORBA, SOAP y Java RMI. Adicionalmente define mejoras en la semántica RMI, permitiendo invocaciones unidireccionales y asíncronas, así como la personalización del acceso de clientes a través de interceptores de invocaciones.

RMI elimina la necesidad de generar clases stubs, para ello utiliza las clases proxy que se realizan conforme se van necesitando, de esta forma se va a conseguir la propiedad de generación automática de envoltorios.

Desde el punto de vista de la extensibilidad, resulta muy sencillo añadir nuevos proveedores al sistema, tan solo hay que implementar las clases indicadas por la arquitectura, insertarlas en un archivo JAR y ponerla en su directorio correspondiente. Actualmente se han desarrollado dos proveedores para este entorno uno es XSOAP para Servicios Web y el otro JRMPX para Java RMI (véase la Ilustración 8.4).

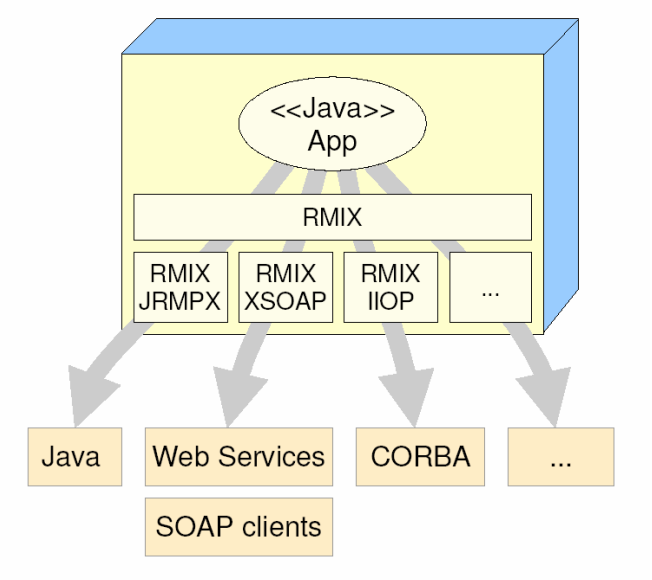


Ilustración 8.4: Interoperabilidad de RMIX

8.2.3.3 IIOP.NETJava .NET CORBA over IIOP

A continuación presentamos IIOP.NET, un sistema que permite interoperar a objetos .NET, Java y CORBA utilizando como mecanismo de envío de mensajes IIOP.

Según Microsoft, la próxima generación de sistemas distribuidos se comunicará a través de Servicios Web. Estos parecen apropiados cuanto se intentan integrar sistemas heterogéneos débilmente acoplados, pero tienen sus limitaciones: no tienen soporte directo para referencias a objetos remotes, no tienen estados y se asemeja más a una técnica para invocación de métodos remotos que a una plataforma completa para sistemas distribuidos. Por otro lado SOAP y XML generan paquetes de intercambio de información muy voluminosos.

.NET y J2EE son muy similares pero también tienen sus diferencias: ambos pueden interactuar a través de Servicios Web. Ambas plataformas ofrecen mecanismos para construir sistemas distribuidos de objetos: Remoting de .NET y Java RMI, pero por desgracia ambos están basados en estándares incompatibles. Afortunadamente, el servicio de remoting de .NET es altamente configurable, se puede incluir fácilmente un nuevo canal de transporte, junto con un serializador y deserializador de objetos.

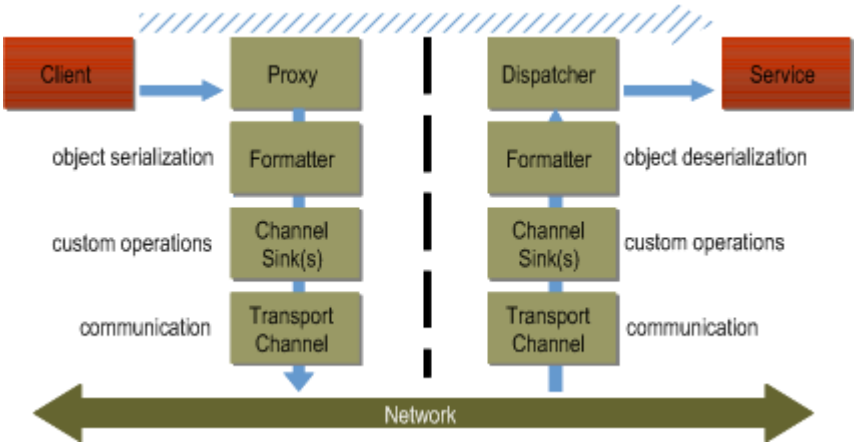


Ilustración 8.5: Canales de Remoting .NET

IIOP.NET es a .NET canal de remoto basado en el protocolo IIOP de la OMG. IIOP es el protocolo definido por el estándar de CORBA, que a su vez es usado por Java RMI/IIOP. IIOP.NET actúa como un ORB (CORBA object request broker). Convierte los sistemas de tipos de .NET a los tipos de CORBA y viceversa, de tal forma que los objetos definidos en ambas aplicaciones están disponibles a otros ORBs. RMI/IIOP implementa solamente un subconjunto de la funcionalidad de ORB debido a algunas limitaciones de sistema de tipos de Java, proporcionando así las mismas características para la plataforma J2EE que IIOP.NET aporta para la plataforma .NET.

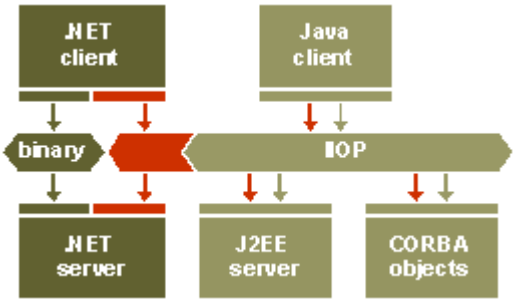


Ilustración 8.6: Arquitectura de IIOP.NET

Utilizar IIOP.NET es tan sencillo como utilizar el servicio Remoting, propio de .NET, es un proyecto open-source hospedado por sourceforge [ELCA04] (http://iiop-net.sourceforge.net/). Fue desarrollado por Dominic Ullmann como parte de su diploma de tesis en ETH-Z. Contienen un generador de IDLs a partir de dll.

8.2.3.4 WSIF Apache Web Services Invocation Framework

El objetivo de WSIF [Wsif03] es proporcionar una API de Java que oculta los detalles de acceso a los Servicios Web multiprotocolo. En vez de enfocarse en SOAP, WSIF se centra en el Lenguaje de Definición de Servicios Web (WSDL). Una definición WSDL consta de dos partes:

- Una parte lógica, denominada el tipo puerto o la interfaz, que define los tipos independientes del protocolo del Servicio Web y los mensajes de entrada y salida.

- Una parte física que define los enlaces específicos a los protocolos de comunicación que van a ser usados por el Servicio Web. Un enlace a un protocolo determinado especifica los detalles necesarios para que las aplicaciones compartan mensajes con una determinada instancia del servicio usando dicho protocolo.

La separación de la abstracción del servicio de su enlace es la clave fundamental en la que se basan los bloques de construcción de WSIF. Así se pueden realizar llamadas a las interfaces lógicas y automáticamente se realizará el envío y recepción de los mensajes oportunos en la parte física. De esta forma SOAP no es más que uno de los posibles protocolos físicos a utilizar, se pueden acceder a otros servicios implementados usando Enterprise Java Beans, Java Message Service, la J2EE Connector Architectura o incluso llamadas a objetos Java locales. Esta solución permite a los servicios compartir una definición de interfaz y permitir múltiples en laces físicos, lo cual es precisamente lo que necesitamos para permitir a las aplicaciones tener un adecuado acceso a sistemas con múltiples middlewares.

Definitivamente WSIF va en la dirección adecuda, pero aún tiene algunas desventajas, la primera y fundamental es que es una solución solo para Java. Esto significa que todos estos protocolo, solo pueden ser invocados desde esta plataforma.

8.2.3.5 SOAPswitch Web Services from iWay Software

Ha pasado ya por varias manos, comenzó siendo ObjectBridge de Visual Edge, evolucionó hasta convertirse en SOAPswitch, esta empresa desapareció y el producto se incorporo en Actional [Actional04], quien a su vez volvió a desaparecer y ahora la podemos encontrar en IWay [Iway05]. Desde su aparición en Visual Edge, el producto no ha sufrido muchas modificaciones, básicamente se trata de un adaptador manual unidireccional de sistemas tanto de aplicación como de información a Servicios Web. Permitirá ofrecer todas las aplicaciones y todos los datos con capacidad de integración como Servicios Web, de una forma sencilla y rápida, simplemente utilizando un asistente y pasando por nueve sencillos formularios. Este sistema es modular con lo que puede ser ampliado con nuevos adaptadores, conversores de formatos, para futuras plataformas. Actualmente se disponen de una amplia variedad de adaptadores: para arquitecturas SAP, Oracle, IBM MQ Series, COM++, CORBA, JEE, XML, etc.

Con respecto a la arquitectura y estructura interna se dispone de muy poca información debido a que es un sistema propietario comercial. No obstante podemos decir que se basa en un Motor de Traducción de Mensajes, que está capacitado para realizar custering y permite convertir sistemas complejos software en servicios.

SOAPswitch proporciona una interfaz dirigida por asistente que permite a los usuarios inspeccionar las aplicaciones del sistema y permitir que estén disponibles como Servicios Web en tan sólo nueve pulsaciones de botón.

Los Servicios Web ayudan a los usuarios a ensamblar nuevos procesos de negocios de forma sencilla, pero la efectividad depende del número de elementos de

nuestro sistema que tengamos disponibles como Servicios Web. Con SOAPswitch podemos, crear, gestionar y exponer como Servicios Web, los procesos de negocios que soportan otras plataformas como puede ser CORBA, DCOM, Java.

Permite a los usuarios exponer nuevas aplicaciones de forma rápida, utilizando los sistemas ya existentes sin el coste y retraso de métodos más complejos de integración de aplicaciones.

Disminuye el coste de forma drástica, en la integración del software existente con las nuevas aplicaciones que se desarrollan e implantan.

Proporciona un complete framework de seguridad que incluye seguridad a nivel de transporte, basado en roles y servicio de seguridad basado en Servicios Web.

8.2.4 Buses

Los sistemas buses consisten en un elemento común donde se conectan el resto de sistemas externos y permite que estos interoperen entre sí, de forma bidireccional y muchos a muchos. Seguidamente, mostraremos varios buses a nivel de lenguaje y posteriormente expondremos buses a nivel de middleware. En este último sentido existe actualmente una tendencia que es la de utilizar los Servicios Web como nexo de unión entre las diversas plataformas.

8.2.4.1 Polylith Software Bus

En [Purtillo94] se describe un sistema llamado Polylith que ayuda a los programadores a preparar e interconectar componentes software formados por varios lenguajes diferentes para su ejecución en entornos heterogéneos. El principal beneficio de Polylith es que una vez que una aplicación ha sido desarrollada para ser ejecutada en un entorno determinado por ejemplo en una red distribuida, se puede adaptar para su reutilización en otro entorno, por ejemplo en un sistema con memoria compartida, de forma automática. No obstante esta flexibilidad se lleva a cabo a costa de perdida en el rendimiento.

Para llevar a cabo esto es necesario una organización software en tiempo de ejecución, un agente desacoplado abstracto llamado software bus, es introducido entre los componentes del sistema. La heterogeneidad en lenguajes y arquitectura se consigue ya que las unidades de programas deben estar preparadas para interactuar directamente con el bus y no con otras unidades de programa directamente. Los programadores deben definir la estructura de sus aplicaciones través de un lenguaje de interconexión de módulos MIL. Polylith utiliza esta especificación para llevar a cabo el empaquetado, que consiste en la generación de stubs, programas fuente para la adaptación, compilación y enlace de las mismas. En tiempo de ejecución una implementación del bus ayuda en el envío de mensajes, servicio de nombres y configuración del sistema.

8.2.4.2 PolySPIN

El sistema PolySPIN [Cho98] (Ayuda para la persistencia, interoperabilidad y nombre multilenguajes) resuelve el problema de interoperabilidad surgido en sistemas escritos en diferentes lenguajes de programación sin necesidad de que el programador escriba ninguna especificación adicional. Está compuesto por cuatro subcomponentes

que permiten una interoperabilidad homogénea: Localizador que resuelve las referencias a objetos, Arbitro de lenguajes que gestiona información específica de cada lenguaje, Comunicador que gestiona la comunicación entre los diversos lenguajes, Emparejador de tipos que realiza la adaptación de tipos de un lenguaje a otro. Todas estas actividades se realizan sin que tenga que intervenir el programador.

PolySPIN está creado para lenguajes orientados a objetos. Mientras en otros sistemas de comunicación entre procesos es necesario algún tipo de IDL, en PolySPIN no es necesario, ya que la información se extrae de los propios módulos, de esta forma los programadores no tienen que preocuparse de ningún tipo de especificación adicional a parte del los módulos y los parámetros. En los sistemas basados en IDL, si una aplicación cambia, debe cambiarse su archivo IDL asociado, es mas todos los programas que utilizaban a este cambien deberán cambiar su IDL, esto ocurre en el caso de MIL de Polylith, o su implementación. Sin embargo el precio a pagar por esta transparencia es su alto nivel de complejidad, de tal forma que si necesitamos trabajar con un nuevo lenguaje al sistema PolySPIN con otros tantos N lenguajes ya existentes, es necesario una N implementaciones para cada uno de los N lenguajes. Mientras que el caso de los sistemas IDL solo es necesario una implementación adicional para el lenguaje determinado.

8.2.4.3 .NET como Bus de Lenguaje

En [Meyer02b], Bertrand Meyer expone y define un Bus de Lenguaje, basándose en estudios realizados en Eiffel para adaptarlo a .NET, considerando que Eiffel tiene herencia múltiple y el modelo de objetos de .NET soporta solamente herencia simple e interfaces. Una primera solución es dejar el trabajo de la adaptabilidad al diseñador del compilador, para que en tiempo de compilación se resuelvan las discrepancias y se genere el modelo que adapta la herencia múltiple a herencia simple e interfaces. Esto nos lleva a replantearnos el concepto de Modelo de Objetos de .NET como un tipo **Bus de Lenguaje** que permite que todos los lenguajes cooperen acordando un modelo básico de mecanismos.

Meyer también indican que hay mecanismos como la genericidad que no pueden ser idealmente gestionados por los desarrolladores de compiladores. En este caso todos los usuarios de lenguajes sin genericidad verán los tipos genéricos como herederos del tipo Object.

En este trabajo, se muestra como resolver de forma adecuada la interoperabilidad entre diferentes lenguajes de programación, utilizando los conceptos de máquinas abstractas. Nuestro objetivo es siguiendo esta idea, permitir además la interoperabilidad de diferentes plataformas middleware LII.

8.2.4.4 UAN Universal Application Network

Universal Application Network [Siebel05] es una arquitectura propietaria orientada a servicios basada en estándares de comunicación que ayuda a las empresas a resolver el problema de la integración de negocios dentro y fuera de la misma. La arquitectura de UAN define procesos que son independientes de las aplicaciones software y por lo canto pueden ser utilizado para integrar cualquier aplicación. Está

desarrollada y comercializada por Siebel. Dentro de la arquitectura UAN Siebel define todo un modelo de integración de aplicaciones de negocio.

Siebel Business Integration Applications incluye tres componentes principales: Siebel Business Integration Processes, Siebel Business Integration Common Objects y Siebel Business Integration Transformations. La siguiente ilustración muestra la arquitectura formada por dichos componentes:

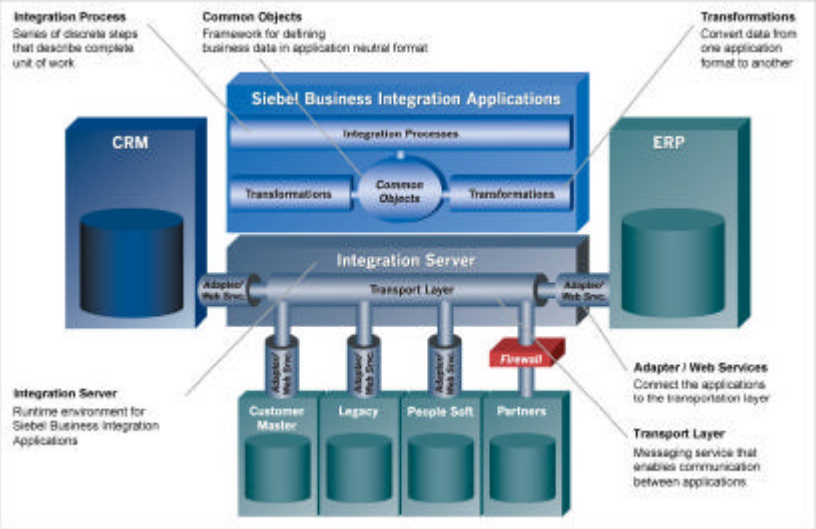


Ilustración 8.7: Siebel Universal Application Network

- Siebel Business Integration Processes, define una serie de pasos coordinados para llevar a cabo tareas de integración, por ejemplo para sincronizar una cuenta contable sobre múltiples aplicaciones, enviar ordenes de trabajo del departamento de compras al departamento de producción, definir ordenes de pago o generar facturas, entre otras muchas.

- Siebel Business Integration Common Objects, proporciona un marco de trabajo que definen datos y formatos de datos independientes de las aplicaciones. Dado que cada aplicación tienen un modelo diferente de datos, se elimina la necesidad de mapear datos entre diferentes aplicaciones directamente. Ejemplos de estos objetos comunes son: cuenta contable, actividad, contacto, empleado, factura, oportunidad, orden, producto, artículo o solicitud de servicios, entre otras muchas.

- Siebel Business Integration Transformations, reconcilia las diferencias entre los diferentes modelos de datos de las diversas aplicaciones de la empresa. Cada aplicación tiene un modelo de datos diferente y un diferente formato de los mismos datos y cualquier comunicación entre ellas necesita algún tipo de conversión. Un simple ejemplo, una aplicación puede utilizar las dos letras finales de los dominios, para identificar los países, mientras que otra puede utilizar el nombre completo.

Todos estos componentes funcionan sobre el entorno de ejecución definido por la aplicación Integration Server, donde a través de su capa de transporte y los diversos

adaptadores / Servicios Web que pueden conectarse a el, permite la comunicación entre aplicaciones.

Está basado en Servicios Web y en XML, los adaptadores realizan las transformaciones oportunas para que se lleve a cabo la comunicación.

8.2.4.5 ESB Enterprise Service Bus

Sonia ESB [Sonic05] es un bus de servicios de empresa que permite a las compañías integrar las aplicaciones informáticas se utilizan dentro y fuera de ella, usando el estándar definido por la Arquitectura Orientada a Servicios (SOA). El ESB es altamente distribuible, con gestión centralizable y permite eliminar las limitaciones de los servidores de aplicaciones monolíticos, creando una Arquitectura Orientada a Servicios dirigida por eventos de negocio, que pueden adaptarse a los, siempre cambiantes, requisitos de las empresas.

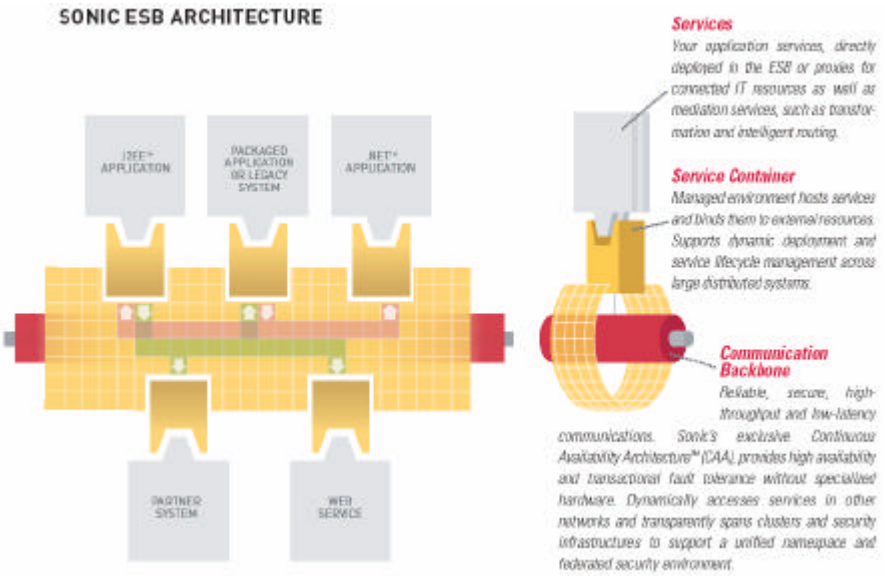


Ilustración 8.8: Sonic ESB

Su arquitectura está formada por los servicios, *Services*, que pueden conectarse al bus, los contenedores de servicios, *Service Container*, que definen el entorno para que puedan conectarse estos al centro de comunicación, *Communication Backbone*. Utiliza XML como mecanismo de transporte, manipulación y transformación de los datos que son transferidos entre los diversos servicios. El centro de comunicación está formado por un servicio de mediación, *Service Mediation*, y un servicio de comunicación, *Service Communication*, el primero, permite los modelos de interacción asíncrona, síncrona y publicación/suscripción. Una avance notorio que añade Sonic ESB es el de realizar enrutamientos inteligentes entre diversos servicios externos o internos, a través de un espacio de nombres global, *Global Service Name*, bajo una arquitectura de enrutado dinámica, *Dynamic Routing Architecture® (DRA)*.

Ya existen disponibles para esta plataforma más de doscientos cincuenta adaptadores, para diversos sistemas tanto internos como externos. Adicionalmente también se puede utilizar en conjunción con servicios de bases de datos, *Database*

Service, permitiendo un acceso y reuso uniforme de las fuentes de datos relacionales existentes y heredadas a través de una arquitectura orientada a servicios.

8.2.4.6 IONA Artix Tech Zone

Artix [Iona05] es un producto de integración de Servicios Web desarrollado por IONA, líder mundial en el desarrollo de soluciones distribuidas. De todos es conocido su versión de CORBA Orbix. Artix es un conjunto de productos independientes de la plataforma y fieles a los estándares que permite construir Servicios Web en Java, C/C++ y mainframe. Es una herramienta que permite integrar los subsistemas informáticos en los que está organizada una empresa, a través de aplicaciones preparadas para ser integradas y creando una arquitectura orientada a servicios (SOA).

- Es independiente de la plataforma
- Independiente del lenguaje: Java, C++ y mainframe
- Compatible 100% con el estándar
- Amplia la interoperabilidad
- Seguridad avanzada e integración de la administración
- Rendimiento y escalabilidad a nivel de empresa
- Permite multitransporte

El la siguiente ilustración se muestra la arquitectura de Artix:

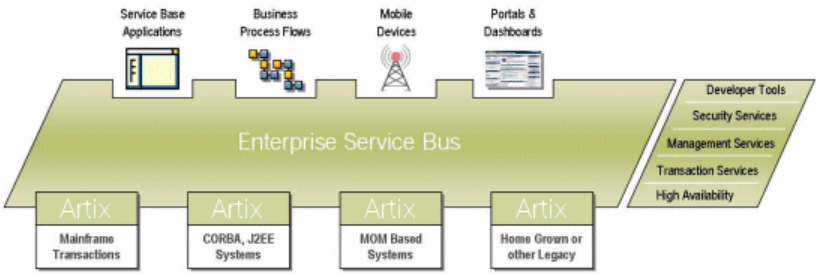


Ilustración 8.9: Artix

A propósito de dicha arquitectura, es posible definir una gran diversidad de plugins, de tal forma que resulte sencillo extenderla a diversos sistemas externos. Está diseñada para poder ser utilizada con las herramientas de desarrollo más populares. Al mismo tiempo aporta, alta disponibilidad, servicios de gestión, de transacciones y de seguridad.

8.2.5 Túneles

Los túneles son los sistemas que utilizando una técnica de comunicación entre procesos subyacenten definen técnicas para resolver la interoperabilidad heterogénea

8.2.5.1 BizTalk

La aplicación propietaria BizTalk de Microsoft, permite unificar aplicaciones separadas de forma coherente, como un todo, con el objetivo de permitir crear procesos de negocios. Dispone de una interfaz gráfica para crear procesos de negocios a partir de los servicios que ofrecen las aplicaciones. Además permite la monitorización y gestión de las aplicaciones, así como mecanismos para especificar reglas de negocios.

La Ilustración 8.10 muestra los componentes principales que forman la arquitectura del BizTalk. Básicamente está formado por adaptadores de entrada y salida de información, unidos mediante tuberías de conversión de datos, que sirven como interfaces de entrada y adaptación de información, una vez realizada esta tarea, mediante el sistema de gestión de mensajes y el de planificación de actividades se procesan y adaptan los datos antes de ser llevados a su destino.

BizTalk manipula toda la información mediante XML, la herramienta de planificación sirve para coordinar las diversas actividades de los procesos de negocios y pueden realizarse de forma visual a través de reglas o utilizando lenguajes convencionales de programación como son Visual Basic, C#, C++ u otros.

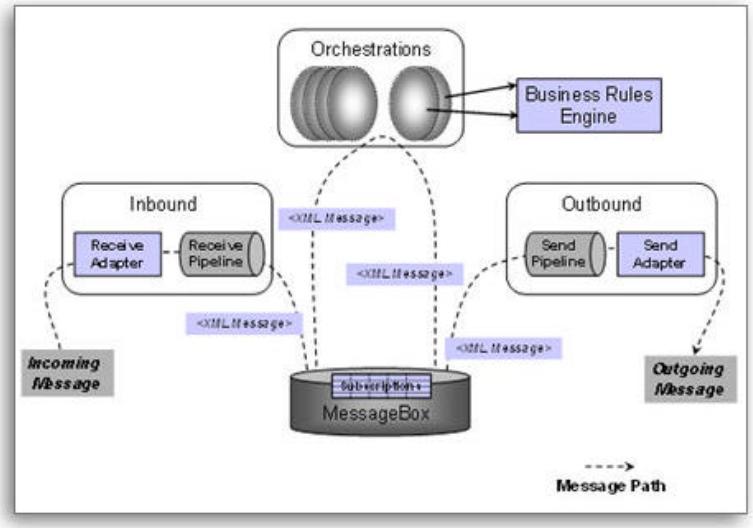


Ilustración 8.10: Arquitectura de BizTalk

La información que permite manipular BizTalk va desde documentos EDI, documentos formados por archivos planos, información en formato XML, acceso a bases de datos, comunicación con aplicaciones CRM, ERP. Todo este tipo de información permite el intercambio de información con procesos de negocios tanto internos de la empresa como externos de otras empresas con las que se mantiene relación.

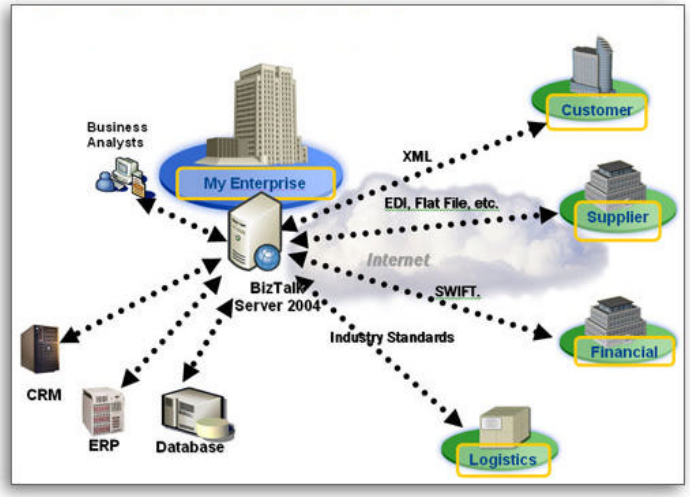


Ilustración 8.11: Tipo de información gestionada por BizTalk

8.2.5.2 EII Enterprise Information Integration

A continuación se muestra un sistema, también propietario, especializado en la integración de información a partir de fuentes de datos. Esta herramienta se denomina Attunity Connect [Attunityy05]. Está principalmente pensado para la integración de información, en contraposición del resto de sistemas que hemos estudiado que están pensadas para la integración de aplicaciones. A pesar de que la integración de información no es el objetivo de esta tesis, hemos querido estudiar este modelo para poder comprobar la semejanza que hay entre la integración de información y la integración de aplicaciones.

Con Attunity Connect vamos a poder conectarnos a diversas y heterogéneas fuentes de datos como son, bases relacionales, sistemas heredados, aplicaciones, a través de una arquitectura común y homogénea.

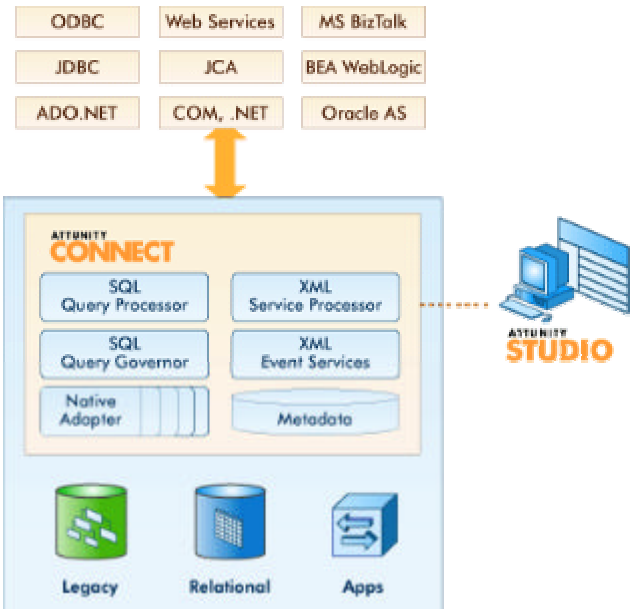


Ilustración 8.12: Attunity Connect

A continuación se enumeran algunas de sus características principales:

- Interfaces orientados a servicios como SQL, XML y Servicios Web
- Una amplia librería de adaptadores preconstruidos para prácticamente todas las plataformas.
- Integración con lectura y escritura transaccional
- Alto nivel de escalabilidad, fiabilidad y rendimiento.
- Fácil instalación y rápida configuración a través de interfaces gráficos basados en asistentes.

De esta forma podemos:

- Acelerar los proyectos de integración
- Reducir el riesgo en las implementaciones
- Reducir el costo de producción
- Maximizar la utilización de sistemas heredados

8.2.5.3 XDMF

XDMF [Clarke00] proporciona motores computacionales con las herramientas necesarias para integrar, en entornos computacionales heterogéneos, mecanismos de intercambio de datos flexibles, con mínimas modificaciones, en vez de imponer un nuevo paradigma en Computación de Alto Rendimiento también denominado HPC. Estos mecanismos nos permitirán integrar modelos consistentes de datos, formateo de datos, gestión de datos, movimiento de datos continuo y transparente, así como la inclusión de algoritmos escalables, modulares y robustos para la manipulación de estos datos de forma distribuida.

Se ha elegido el uso de Scripts para que los usuarios de XDMF puedan realizar la integración de los diversos componentes, considerando que se utilizarán para la gestión de funcionalidades de grano grueso, dejando las de grano fino para los lenguajes de programación de sistemas como pueden ser C++, C o Fortran. De esta forma los lenguajes como Java, Pithon y Tcl ya están soportados, además se ha definido un generador de wrapper (envoltorios) de interfaces sencillo para permitir a estos y otros sistemas interactuar con las clases C y C++ subyacentes dentro de XDMF de manera que se pueda acceder a toda la funcionalidad del sistema a partir de una gran variedad de lenguajes heterogéneos y de una forma orientada a objetos.

Esta basado en Network Distributed Global Memory NDGM, Hierarchical Data Format versión 5 HDF5 y XML. Estos dos últimos han sido elegidos como el estándar para el intercambio y formateo de los datos y, NDGM ha sido elegido como el mecanismo para la gestión de memoria distribuida en red.

Existen varias herramientas tanto comerciales como de código abierto (open source) que utilizan XDMF, del primer caso tenemos EnSight y del segundo OpenDX, ambas permiten realizar aplicaciones que manipulan y usan diferentes modelos de datos desde diferentes entornos de programación.

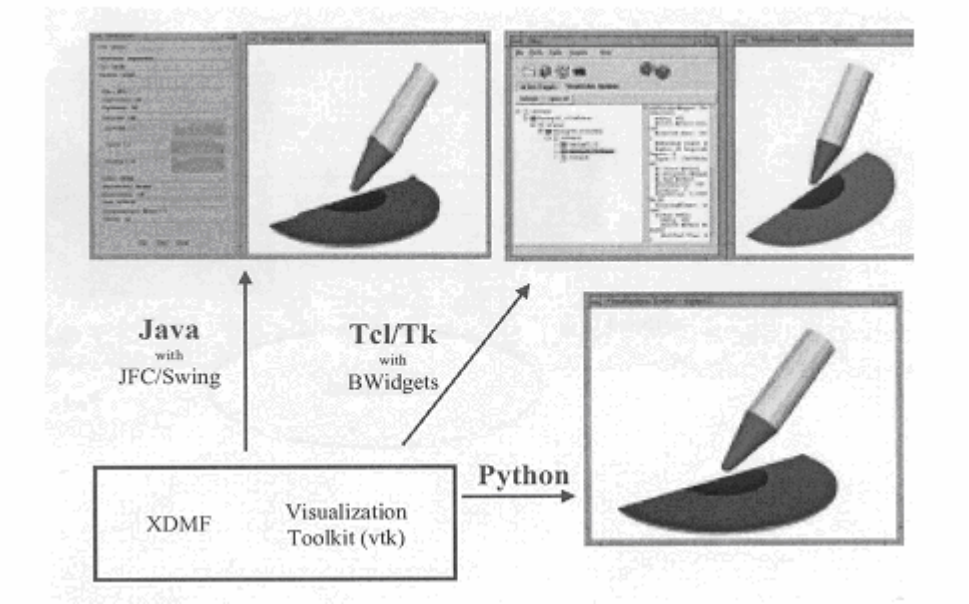


Ilustración 8.13: Heterogeneidad con XDMF

Este mecanismo resuelve la heterogeneidad a nivel de transferencia e intercambio de datos, utilizando como mecanismos de comunicación la gestión distribuida de memoria. Es multiplataforma e independiente del lenguaje, si bien no es un sistema abierto a interoperar con otros sistemas diferentes.

8.2.5.4 HARNESS

Harness [Midliardi01] es un sistema de metacomputación basado en la noción de contenedor software donde se conectan componentes modulares. Estos componentes se coordinan unos con otros para llevar a cabo varias funciones dentro de un sistema distribuido débilmente acoplado, permitiendo mejorar la flexibilidad y funcionalidad del sistema, así como reconfigurar las capacidades del sistema según las necesidades que vayan surgiendo. El corazón del modelo arquitectónico está basado en una máquina virtual distribuida (DVM) que proporciona un contexto de ejecución concurrente de los programas. Cada DVM está asociado con un nombre dentro de un único espacio de nombres, que está formado por varios nodos o máquinas. Dentro de cada máquina hay un conjunto de componentes conectados. Cada componente dentro de cada nodo puede ser específico o copiado. Normalmente todo nodo tiene un conjunto de componentes copiados que son los que proporcionan la funcionalidad básica del sistema en cada nodo, como puede ser el paso de mensajes, gestión de procesos, etc. Los usuarios de Harness pueden instalar sus propios componentes, crearlos o incluso instalar componentes que emulan a otros entornos, actualmente hay disponibles PVM [Geist94], MPI [Mpi05] y JavaSpace, de tal forma que permitirá crear un marco de trabajo (framework) donde puede funcionar los códigos heredados. Harness está actualmente implementado en Java, con lo que está cerrado a los servicios que puede proporcionar esta tecnología. Se pretende construir Harness II que es un sistema más abierto y

estándar con respecto a descripción de servicios, registro de los mismos, técnicas de directorios y búsqueda de componentes e invocación de servicios. En Harness II se pretende obtener una mayor interoperabilidad heterogénea a partir de la infraestructura de los Servicios Web, argumentando para ello que la mayoría de los sistemas están tendiendo a esta tecnología. Si analizamos detenidamente Harness y los Servicios Web nos daremos cuenta que tienen una mecánica de funcionamiento muy parecida.

Pretenden pasar del uso de Java a Servicios Web por que consideran que los Servicios Web van a ser ubicuos y por eso van a ser más heterogéneos, además pretende mejorar los Servicios Web, debido a que son muy ineficientes en procesamientos y los procesos de Harness requieren un alto rendimiento. El objetivo de Harness es ser un marco de trabajo más para ser utilizado en entornos Grid, ver sección 6.3.10.

Este sistema el concepto de máquinas virtuales para solucionar el problema de la heterogeneidad de los diversos elementos que lo componen, además de para facilitar la extensión a otros sistemas emulándolos.

8.2.5.5 DOTS

Sistema de Threads Orientado a Objetos Distribuidos DOTS [Bloch01] es un plataforma de paralelización que permite a los programadores construir aplicaciones paralelas distribuidas usando un paradigma de programación de alto nivel y fácil de usar, entre clusters heterogéneos de alto rendimiento. Proporciona un servicio de llamadas remotas asíncronas orientada a objetos para programas en C++, donde el kernel de comunicación es altamente adaptable. La idea principal del modelo de programación de DOTs es hacer que el paradigma de programación con threads utilizado en máquinas de memoria compartida esté disponible en entornos con memoria distribuida. El paradigma no añade nada nuevo para el programador de threads con memoria local, ya que disponemos de dots_fork y dots_joing cuyo funcionamiento es análogo a fork y join de manejo de threads pero para sistemas distribuidos.

8.2.6 Otros

Existen multitud de tecnologías que se enfrentan al problema de la heterogeneidad y no encajan en las descripciones anteriores, además de estar enfocados a otros niveles de interoperabilidad o distribución. Tal es el caso de los sistemas coordinados. A continuación mostramos dos sistemas que están enfocados desde esta perspectiva.

8.2.6.1 AaLaadin

En [Ferber98] se define un metamodelo llamado Aalaadin, basado en grupos, roles y agentes para construir sistemas multiagente permitiendo heterogeneidad a nivel de lenguajes, aplicaciones y arquitectura. Define dos modelos conceptuales:

- el modelo concreto, donde se definen los elementos principales como agentes, grupos y roles

- el nivel metodológico abstracto, donde se definen los elementos necesarios para llevar a cabo el diseño del sistema de agentes y cuyos elementos principales son las estructuras del grupo, la estructura organizacional y la organización.

Así mismo utiliza el concepto de reflectividad, cosificando los elementos principales del sistema y permitiendo así añadir propiedades o servicios a los agentes de forma dinámica. De manera que, un agente puede ser móvil o no dependiendo si se agrega a los grupos Movilidad y Gestión de Movilidad.

Se realiza una instanciación de este modelo basado en Java llamado MadKit, en el se definen los siguientes elementos de diseño:

- Arquitectura de micro-kernel
- Agentificación de servicios
- Modelo común de componentes gráficos.

De esta forma se consigue construir y utilizar componentes en aplicaciones Java, en applets Java y en un entorno denominado G-Box, que podrán correr tanto en computadoras, como en PDAs con Java o incluso en dispositivos móviles con arquitectura Java.

Es un sistema flexible multiplataforma, que permite la adaptabilidad, extensibilidad, si bien está pensado de momento para Java. Sería necesario definir un metamodelo abstrayendo los servicios de comunicación.

8.2.6.2 MANIFOLD

En el artículo [Arbad94] se utiliza la coordinación dirigida por control para gestionar la reconfiguración de sistemas, a través de la plataforma MANIFOLD. Esta basada en Ideal Worker Ideal Manager IWIM, donde los trabajadores son los que generan información y consumen, a través de puertos independientemente de quien este conectados en ellos. Estos procesos trabajadores pueden ser generados en diferentes lenguajes. Por otro lado, los controladores son los encargados de gestionar las conexiones que permiten la comunicación del sistema, son los encargados de coordinar los diferentes trabajadores para que el flujo de información sea el adecuado. En este articulo se presenta un ejemplo donde dos trabajadores, monitor y enfermera, intercambian información para llevar a cabo la tarea de monitorización de enfermos y este flujo de información es gestionada por el proceso controlador llamado supervisor, que es el encargado de, en caso de que alguna enfermera se retire, reconfigurar el reparto de monitores para que el proceso de monitorización siga siendo lo más seguro y efectivo posible.

8.3 Comparativa

Una vez estudiadas y clasificadas, algunas de las técnicas utilizadas para solucionar la heterogeneidad en sistemas distribuidos. Desarrollamos una tabla comparativa de las mismas, desde el punto de vista de las siguientes características comunes identificadas en este tipo de tecnologías:

- Multilenguaje, el sistema permite que elementos desarrollados en diferentes lenguajes de programación puedan interoperar entre sí.

- Multiplataforma, los sistemas podrá ser utilizado en diversas plataformas, tanto hardware y de red como software, a nivel de sistema operativo.

- Multimiddleware, permitirá que sistemas desarrollados para diferentes middleware RMI interaccionen.

- Multiprotocolo, el sistema facilita la interoperabilidad con plataformas donde la comunicación entre procesos puede utilizarse con diferentes protocolos de envío de mensajes o con middlewares diferentes al RMI como pueden ser, publicación/suscripción, eventos, etc.

- Abierto, obtenemos sistemas independientemente extensibles y que permite a componentes heterogéneos conectarse o desconectarse del sistema. Con esta característica los sistemas adquieren un grado más de flexibilidad, pudiendo adaptarse o ser adaptados a nuevos paradigmas o dominios de problema.

- Bidireccional, la tecnología permite la invocación entre los elementos en ambos sentidos.

- Extensible, el sistema puede ampliarse para ser utilizado con diferentes tipos de elementos software con lo que interoperar. Estos elementos dependerán de si el sistema es multiprotolo, multimiddleware o multilenguaje.

- Automatizado, la realización de los proxies o envoltorios de los objetos a utilizar se realiza de forma automática, sin intervención ni configuración manual previa.

- Escalable, conserva y aumenta su efectividad cuando ocurre un incremento significativo en el número de recursos y número de usuarios que pertenecen a él.

- No Intrusivo, tanto los elementos cliente como los servidores no necesitan ser modificados para ser utilizados con esta tecnología.

En dicha tabla podemos destacar los buses a nivel middleware como lo que más características cumplen, si bien adolecen de poco abiertos, al ser propietarios y estar basados en lenguajes cerrados y compilados, se reduce su flexibilidad y su capacidad de extensión a las interfaces definidas a priori para los plugins que se pueden conectar en estos; y no automatizados, de tal forma que hay que hacer uso de técnicas manuales de configuración para indicar determinados parámetros necesarios a la hora de construir un envoltorio de un determinado objeto o sistema. Por otro lado están diseñados para ser muy generales y poder adaptarse tanto a heterogeneidad de comunicación entre procesos como integración de datos. Son escalables, no intrusivos, bidireccionales y permiten interaccionar con sistemas multiprotocolo.

En cuanto a las tecnologías de conmutación, reseñar que permiten comunicación de uno a muchos, aunque con excepción de SoapSwitch suelen ser intrusivos, unidireccionales, aunque IIOP.NET permite la bidireccionalidad a través del protocolo IIOP y con excepción de WSIFApache, están preparados solamente para protocolos middleware RMI. Una característica a resaltar es la posibilidad de RMIX de ser automatizado, de tal forma que se generan los proxies de forma dinámica, conforme son necesarios.

La característica más interesante que aportan los puentes es su capacidad bidireccional, de tal forma que los sistemas de ambos extremos se ven entre sí de forma transparente y pueden tanto recibir invocaciones como iniciarlas.

Los sistemas clasificados como túneles son bastante diferentes entre sí, tenemos Biztalk, que nos permite la mayoría de las características enumeradas, salvo que es específico de plataforma ya que es un producto Windows, y la adaptación a objetos no es automática. Attunity Connect al estar basada en integración de datos, no nos aporta información destacable, salvo que es automatizado dada la facilidad para automatizar sistemas basados en datos y es intrusivo ya que los sistemas que deben atacar esos datos deben ser conscientes de la existencia de los mismos en el formato indicado por la plataforma a la que pertenecen. XDMF está basado en memoria compartida con lo que la bidireccionalidad resulta inmediata, está preparada para varios lenguajes. Harness está basado en máquinas virtuales, la interoperabilidad se consigue a través de componentes añadidos a nivel de máquina.

Tabla 1: Comparativa de tecnología para abordar la heterogeneidad

	PUENTES		CONMUTADORES					BUSES					TÚNELES				
	JNBRIDGEPR	EXPERTSOFT	REMMOC	RMIX	IIOP.NET	WSIFAPACH	SOAPSWITC	POLYSPIN	POLYLITH	UAN	ESB	ARTIX	BIZTALK	ATTUNITY	XDMF	HARNESS	DOTS
MULTILENGUAJE	-	*X*	*X*	*X*	*X*	-	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*
MULTIPLATAFORMA	-	-	-	*X*	-	*X*	*X*	*X*	*X*	*X*	*X*	*X*	-	*X*	*X*	*X*	*X*
MULTIMIDDLEWARE	*X*	*X*	*X*	*X*	*X*	-	*X*	-	-	*X*	*X*	*X*	*X*	*X*	-	-	-
MULTIPROTOCOLO	-	-	*X*	-	-	*X*	-	-	-	*X*	*X*	*X*	*X*	*X*	-	-	-
ABIERTO	-	-			-	-	-	-		-	-	-	*X*	-	-		-
BIDIRECCIONAL	*X*	*X*	-	-	*X*	-	-	*X*	*X*	*X*	*X*	*X*	*X*	-	*X*	-	*X*
EXTENSIBLE	-		*X*	-	-	-	-	*X*		*X*	*X*	*X*	*X*	*X*	-	*X*	
AUTOMATIZADO	-	*X*	-	*X*	-	-	-	*X*	-	-	-	-	-	*X*	-	*X*	-
ESCALABLE			*X*		-	-	-			*X*	*X*	*X*	*X*	*X*	*X*	*X*	
NO INTRUSIVO			-	-	-	-	*X*			*X*	*X*	*X*	*X*	-	-		-

EL SÍMBOLO "X" INDICA QUE EL SISTEMA PRESENTA LA CARÁCTER, EL SIMBOLO "-" INDICA QUE NO LA PRESENTA Y VACIO INDICA QUE NO HAY SUFICIENTE INFORMACIÓN PARA DETERMINAR SI LA CARACTERÍSTICA ESTÁ PRESENTE O NO.

8.4 Conclusiones

De lo expuesto anteriormente observamos que no existen sistemas bidireccionales generales que se centren en middlewares LII de muchos a muchos. Por un lado, tenemos soluciones parciales como los puentes o los conmutadores y, por otro, soluciones muy generales como los buses universales o los túneles. Estas últimas suelen ser arquitecturas propietarias cerradas, orientadas a la integración de aplicaciones, muy genéricas. Permiten tanto la comunicación entre procesos como a nivel de datos, y necesitan de sofisticadas técnicas de administración y configuración, tanto de los adaptadores que pueden contener como de los traductores a nivel de datos. Son poco flexibles y están basadas en "motores de invocaciones", con arquitectura desconocida, construidas en lenguajes compilados y, algunos de ellos, con la posibilidad de extensión a través de plugins específicos y concretos.

Partiendo de estas consideraciones, en el siguiente capítulo expondremos el planteamiento que pretendemos solucionar, en esta tesis, con un modelo arquitectónico basado en máquinas abstractas.

Capítulo 9:
Planteamiento del problema

En este capítulo se expone la motivación y planteamiento del problema descubierto en los capítulos anteriores, que es el propósito de esta tesis, y cuya solución se expondrá a lo largo de los siguientes capítulos. Para ello, repasaremos el contexto en el que se enmarca el problema, describiremos el problema concreto, posteriormente comentaremos las soluciones al problema que han planteado terceros, para llegar finalmente, a nuestro planteamiento del propósito de esta tesis.

9.1 Contexto del problema

A lo largo del Capítulo 6 hemos posicionado nuestra diserción en el entorno de los *sistemas distribuidos*. Con este fin, hemos identificado qué es un sistema distribuido y qué características debe tener.

Hemos visto que un sistema distribuido es una *colección de ordenadores independientes que colaboran unidos por una red de comunicación* y que debe permitir compartir recursos de todo tipo de forma abierta, flexible, concurrente, movible, eficiente, fiable y transparente. De este modo, formará un sistema que será escalable, seguro, de alta disponibilidad y extensible.

Cumpliendo la definición y características anteriores, existe una gran tipología de sistemas distribuidos, a saber: sistemas de comunicación entre procesos, archivos distribuidos, sistemas de seguridad distribuida, servicios de nombres, transacciones, replicación, sistemas multimedia, sistemas de computación masiva, sistemas de coordinación y acuerdo, sistemas de memoria compartida distribuida, arquitecturas de integración de aplicaciones, etc.

Para poder construir sistemas distribuidos, necesitamos tanto computadores, con todo su software y hardware asociado, como redes de computadoras para interconectarlos y técnicas de abstracciones de comunicación, de más alto nivel que las utilizadas en las comunicación a nivel de transporte. Con motivo de esta necesidad de abstracciones de comunicación, ha surgido una nueva capa software que se encuentra entre medias de las aplicaciones distribuidas y los servicios de red, los sistemas operativos y el hardware de comunicaciones. Esta nueva capa se denomina m*iddleware* y pretende resolver el problema de la comunicación entre procesos de forma independiente del lenguaje y de cualquier plataforma hardware o software subyacente. En los últimos años una gran variedad de sistemas middleware han surgido, pero los más ampliamente extendidos son los denominados RMI (Remote Method Invocation) o LII (Location Independent Invocation).

9.2 Identificación del problema

Los middlewares se definieron con la intención de permitir una máxima interoperabilidad entre aplicaciones heterogéneas a nivel de lenguajes de programación, sistemas operativos y redes de comunicación. De tal forma que aplicaciones de diferentes fabricantes puedan interoperar y compartir recursos entre sí. A partir de estos, surgieron una gran cantidad de servicios independientes de las aplicaciones, como son, los servicios horizontales de transacciones, nombrados, seguridad, etc. y los servicios verticales enfocados a determinados dominios de aplicaciones, también denominados *facilities*.

Diversos middleware RMI han surgido, de entre los que cabe destacar: CORBA desarrollado por la OMG, la RMI de Java, que actualmente está adaptada para interoperar con CORBA, DCOM de Microsoft, que surgió de la filosofía de desarrollo de aplicaciones abiertas a través de componentes y la recién llegada, y cada vez más predominante, Servicios Web, promovida por la W3G y Microsoft.

Decidir cuál de las tecnologías es la más apropiada no es fácil. Actualmente, todas estas tecnologías están conviviendo entre sí, pero son incompatibles; de manera que aplicaciones desarrolladas en CORBA no puede compartir recursos con aplicaciones desarrolladas en otras plataformas como DCOM y viceversa.

De esta forma lo que pretendía ser una solución se ha convertido en un problema: **El problema de la interoperabilidad entre plataformas middleware heterogéneas de invocación remota de métodos**. Tanenbaum en [Tanen02] dice:

"seguramente que no tardará en surgir alguna otra capa denominada 'upperware' que solucione este problema".

9.3 Revisión de Soluciones

Si bien no ha sido desarrollada ninguna solución ideal "upperware", como la denomina Tanenbaum, sí ha surgido un gran abanico de soluciones parciales para

resolver el problema de la heterogeneidad de sistemas tanto RMI como de cualquier otro tipo.

En el Capítulo 8 se expusieron y compararon una variedad de ellos. Llegamos, incluso, a idear una pequeña clasificación, siguiendo la analogía existente con las soluciones propuestas a la heterogeneidad en el mundo de las redes de ordenadores, como son: adaptadores, puentes, conmutadores, buses, enrutadores y túneles. Si bien la clasificación no resulta tan sencilla, ya que, de hecho, algunos sistemas no han podido ser clasificados, mientras que otros tienen cabida en más de una categoría.

No existen sistemas bidirecionales generales que se centren en middlewares LII. Por un lado, tenemos soluciones parciales como los bridges o los switches y, por otro, soluciones muy generales como los buses universales o los túneles. Estas últimas suelen ser arquitecturas propietarias cerradas, orientadas a la integración de aplicaciones, muy genéricas. Permiten tanto la comunicación entre procesos como a nivel de datos, y necesitan de sofisticadas técnicas de administración y configuración, tanto de los adaptadores que pueden contener como de los traductores a nivel de datos. Son poco flexibles y están basadas en "motores de invocaciones", con arquitectura desconocida, construidas en lenguajes, compilados y, algunos de ellos, con la posibilidad de extensión a través de plugins específicos y concretos.

9.4 Planteamiento del propósito

De todo lo visto anteriormente podemos plantear como problema identificado a resolver que:

No existe actualmente un modelo arquitectónico general que permita construir sistemas que solucionen el problema de la interoperabilidad, bidireccional de muchos a muchos, entre middlewares heterogéneos de invocación remota de métodos, que aporten el máximo de características necesarias en un sistema distribuido heterogéneo.

De lo que se deduce el propósito de nuestra tesis que es:

Definir un modelo arquitectónico general que permita construir sistemas que permitan solucionar el problema de la interoperabilidad, bidireccional de muchos a muchos, entre sistemas middleware heterogéneos de invocación remota de métodos, tratando de aportar el máximo de características necesarias en un sistema distribuido heterogéneo además de otras características como la automatización, la facilidad de aprendizaje, uso y administración.

El objetivo del resto de este trabajo es describir dicho modelo y comprobar su efectividad a partir de la construcción de varias instancias del mismo, así como un prototipo basado en una máquina virtual concreta.

En el siguiente capítulo describiremos detalladamente los requisitos que deseamos que tengan los sistemas construidos a partir de nuestro modelo. Para posteriormente, exponer varios criterios de diseño que nos ayudaran y facilitarán en la construcción de sistemas interoperables heterogéneos. A partir de ahí, en los cinco

capítulos siguientes, se expondrá el contenido del modelo arquitectónico a un nivel alto de abstracción.

Capítulo 10:
Requisitos del sistema

El objetivo principal e inicial de esta tesis es el desarrollo de un modelo arquitectónico basado en máquinas abstractas, que dé lugar a sistemas software que permitan la interoperabilidad entre diversas y heterogéneas plataformas de middleware. A lo largo de este trabajo llamaremos a estos sistemas *Buses Middleware* o *middlebuses*. Este nombre surge debido a que su comportamiento es similar al de un bus hardware pero aplicado a middleware, es decir, si dentro del bus están conectados varios elementos y se añade uno nuevo, este puede comunicarse con los elementos anteriores y viceversa. De igual modo ocurre con los middlebuses, si en uno determinado hay conectados varios middleware y se conecta uno más, este podrá interoperar con todos los anteriormente conectados y, a su vez, estos también podrán hacerlo con el recién llegado. Todo ello se consigue de forma transparente o no intrusiva en la capa de middleware, es decir no hay que hacer ninguna modificación en las aplicaciones existentes. Así, por ejemplo, un objeto CORBA puede utilizar un objeto RMI Java de forma completamente transparente. Este modelo arquitectónico permitirá que los sistemas construidos a partir de él tengan las siguientes características.

10.1 Heterogeneidad

Nuestro sistema permitirá interoperar sistemas interconectados con una gran diversidad de hardware o software. Uno de los objetivos de los sistemas distribuidos es superar la barrera de la heretogeneidad. Para ello se han definido mecanismos de comunicación entre procesos independientes de los lenguajes de programación, los sistemas operativos, estructuras de red y estructuras hardware. Los más representativos de estos mecanismos son los middleware. Nuestra tesis pretende romper una barrera adicional surgida a partir de plataformas middleware heterogéneas. Así pretendemos

definir un modelo arquitectónico que permita construir sistemas independientes de los sistemas operativos, los lenguajes de programación, los sistemas de comunicación o los sistemas de interoperabilidad middleware.

10.2 Transparencia

Pretendemos ocultar la naturaleza heterogénea de las plataformas middleware que interoperan entre sí, de tal forma que los elementos de una plataforma origen vean a los elementos de las demás plataformas destinos como si fuesen elementos de la misma plataforma origen. Es decir, los Servicios Web podrán ver y manipular a todos los objetos COM, CORBA, JAVA RMI, etc. como Servicios Web. Por otro lado, los objetos CORBA podrán crear e interoperar con objetos de todas las otras plataformas conectadas en el sistema, como DCOM, JAVA RMI, Servicios Web, etc. de igual forma que crea e interopera sus propios objetos CORBA. Esto mismo ocurrirá para el resto de plataformas distribuidas, es decir, todas podrán interoperar entre sí, sin necesidad de realizar ningún cambio ni en los servidores ni en los clientes. Todo el trabajo de transformación y adaptación lo lleva a cabo el middlebus.

Dentro de la propiedad de transparencia conseguiremos los siguientes tipos de transparencia:

10.2.1 Transparencia de acceso

Las aplicaciones cliente accederán a los objetos locales, remotos de la misma plataforma y remotos de diferentes plataformas, de la misma manera.

10.2.2 Transparencia de ubicación/localización

A la hora de localizar un determinado objeto, se seguirán utilizando los mismos servicios de localización. No se añaden mecanismos de localización diferentes para diferentes plataformas.

10.2.3 Transparencia de concurrencia

Dada la naturaleza concurrente de los sistemas distribuidos, nuestro sistema permitirá acceder de forma concurrente a los diversos objetos de las otras plataformas, definiendo sus propios hilos de ejecución para cada uno de ellos.

10.2.4 Transparencia de escalado

El sistema podrá expandirse en tamaño para poder afrontar de forma transparente, un mayor número de transacciones o para obtener un mejor rendimiento, si es necesario, entre las diversas plataformas distribuidas conectadas.

10.2.5 Transparencia heredada

Todas las propiedades de transparencia de las plataformas que interoperan, serán transparentes además para los objetos de las otras plataformas. De esta forma podremos tener:

1- Transparencia de replicación

2- Transparencia de fallo

3- Transparencia de movilidad

4- Transparencia de persistencia

5- Transparencia de migración

Ya comentadas anteriormente en el estado del arte, sección 6.2.10.

10.3 Tecnología no intrusiva

No hay que hacer cambios ni en las aplicaciones ya existentes que utilicen los middleware heredados ni las nuevas aplicaciones se deben diseñar para utilizar este nuevo sistema. Esto es debido a todas las propiedades de transparencia mencionadas anteriormente. De nuevo, todo el trabajo de transformación y adaptación lo lleva a cabo el middlebus.

10.4 Multiplataforma

El sistema, al estar basado en máquinas abstractas, hereda todas las características de estas, de tal forma que será multiplataforma, permitiendo, aparte de la interoperabilidad entre plataformas diferentes, instalarse en todas las plataformas para las que haya implementación de la máquina abstracta que hayamos elegido para la construcción del sistema.

10.5 Sencillo y rápido de instalar

La implementación del sistema será tan sencillo y rápido de instalar como cualquier otra aplicación desarrollada para una determinada máquina abstracta. Así, si utilizamos la máquina virtual de Java, la instalación del sistema será ten sencillo como la instalación de Java y de sus correspondientes archivos JAR. Si elegimos CLI de .NET será tan rápido como instalar el framework de .NET y los ensamblados de la aplicación.

10.6 Sin curva de aprendizaje

Los programadores no tienen que aprender nada sobre los middleware con los que van a interactuar, solo de la tecnología middleware ya aprendida con la que siempre han trabajado. Es decir, si los programadores están acostumbrados a utilizar CORBA,

van a continuar programando bajo esta arquitectura, pero podrán interoperar con las otras plataformas interconectadas. De igual modo, no necesitarán aprender nada del nuevo sistema middlebus, no será necesario interaccionar con objetos CORBA, ni con ningún otro, para configurar, abrir, cerrar o activar los servicios del bus, a nivel de desarrollo de aplicaciones.

10.7 Extensibilidad

El sistema podrá ampliarse fácilmente con nuevos sistemas de comunicación futuras, incluso aunque no sean middleware. Si bien nuestro modelo arquitectónico está pensado para estos últimos, en cuyo caso la extensibilidad queda reducida a refinar o construir clases que heredan de tan sólo cuatro clases. Es decir, con tan solo redefinir algunos métodos de cuatro clases conseguiremos añadir una nueva plataforma al middlebus.

Se puede construir una extensión de un plataforma en otra plataforma diferente a la original, con esto conseguiremos la propiedad de escalabilidad, pero sólo será posible si la máquina abstracta está preparada para la distribución transparente. Esto ocurre en la máquina Oviedo3, en JAVA y en .NET.

10.8 Escalabilidad

Como se ha mencionado en el apartado anterior, el sistema construido podrá escalarse para aumentar el rendimiento, dentro de las restricciones propias de las conversiones necesarias en las transformaciones y las invocaciones, o con el fin de soportar una mayor carga. Para ello, se debe elegir una máquina abstracta que internamente permita la distribución, de forma transparente dentro de sus servicios.

10.9 Eficiencia

No se realiza ningún tipo de conversión temporal ni ningún canal intermedio; aunque, debido a que está basado en máquinas abstractas, dependerán mucho de la eficiencia de esta. Por ejemplo, si utilizamos .NET u otra qué permita compilación JIT, será un sistema muy eficiente. Si bien siempre habrá una pérdida en las adaptaciones y en los reenvíos de mensajes de una plataforma a otra, aunque esta queda minimizada.

10.10 Automatizado

La creación de proxy de los objetos a utilizar se realiza de forma automática. En otros sistemas es necesario saber de ante mano con qué objetos vamos a interoperar y definir los wrappers o proxys oportunos de forma manual, como ocurre en SOAPSwitch [Actional04], [Ren01], para posteriormente activar el sistema.

10.11 Fácil de gestionar

Debido a la característica de automatización, no se plantea ningún tipo de gestión en el modelo arquitectónico, si bien a niveles prácticos será necesaria. En tal caso, será muy sencillo crear los módulos que se consideren, utilizando las capacidades scripts de la máquina abstracta que se desee utilizar.

10.12 Flexibilidad

El sistema permite fácilmente ampliarse, no solamente para añadir nuevas plataformas distribuidas middleware, sino para la integración de datos, monitorizar, administrar, auditar. Con el modelo arquitectónico propuesto y al estar basado en máquinas abstractas se consiguen sistemas con un alto grado de flexibilidad.

10.13 Uniformidad

La uniformidad la obtenemos desde el punto de vista del paradigma de la orientación a objetos. Desde dentro de la máquina tenemos una uniformidad de objetos, todo son objetos, hasta los sistemas externos se manipulan también como objetos. Dispondremos de objetos primitivos, reflectivos, de usuario y externos, y todos ellos se manipulan y tratan de igual forma, consiguiendo así una uniformidad completa. De manera que los sistemas externos orientados a objetos siguen viendo el resto de sistemas como orientados a objetos.

10.14 Integrabilidad

Se podrán integrar elementos de otras plataformas o sistemas a nuestros sistemas de forma sencilla. Nuestros desarrolladores no tendrán que tener conocimientos sobre las plataformas heredadas que integramos y seguirán desarrollando en la plataforma de siempre, pero, además, podrán acceder a los objetos integrados.

10.15 Interoperabilidad completa

Dentro de la propia máquina se consigue tener una plataforma de desarrollo con interoperabilidad completa con todas las plataformas externas. Supongamos que desarrollamos una instancia de nuestro modelo arquitectónico utilizando la máquina virtual de Java, a partir de ese momento los programadores de Java podrán hacer uso de todos los objetos CORBA, Servicios Web y DCOM que estén disponibles y podrán ser tratados como objetos por cualquier otro objeto de Java.

De acuerdo con los requisitos anteriormente expuestos, desarrollaremos un modelo arquitectónico general que modele nuestro sistema y, posteriormente, veremos varias instanciaciones concretas de este modelo aplicadas a varias plataformas específicas.

Construiremos un prototipo del middlebus, basándonos en la máquina abstracta orientada a objetos Oviedo3, definida dentro del sistema integral orientado a objetos, que llamaremos OviBus.

A continuación, veremos que se puede generalizar este modelo arquitectónico para poder ser aplicado a sistemas no middleware de tal forma que resulte una herramienta útil en la integración de sistemas. Incluso puede dar lugar a paradigmas para la solución de determinados dominios de problemas basados en sistemas intermediarios abiertos y flexibles.

Si bien nuestro modelo arquitectónico tiene las anteriores propiedades, los sistemas construidos con él, inicialmente, solo pueden ser utilizados con mecanismos de comunicación entre procesos de tipo invocación independiente de la localización LII o invocación a métodos remotos que permitan localizar los elementos del sistema a través de alguna técnica de nombrado, ya sea DNS, registro del sistema, repositorio de objetos, directorio de objetos tipo LDAP. Por otro lado, debe tener algún mecanismo de introspección de tal forma que podamos obtener información descriptiva del servicio a utilizar y, así, poder crear las clases envoltorios, necesarias de forma dinámica y, por último, un protocolo de comunicación dinámica del tipo DII, invocación dinámica o protocolo de invocación remota. Además, restringiremos el dominio del problema considerando solamente los tipos básicos de objetos. También se evita la manipulación de referencias, tanto para objetos ya creados como para nuevos objetos. Todo esto lo comentaremos en detalle en el siguiente capítulo, donde expondremos las restricciones iniciales necesarias llevadas, a cabo la para construcción de nuestro modelo arquitectónico.

Capítulo 11:
Cuestiones de diseño

En el capítulo anterior se expusieron las características deseadas para los sistemas construidos a partir de nuestro modelo arquitectónico. En este capítulo expondremos las características y restricciones del propio modelo arquitectónico y de los sistemas construidos con él.

Se pretende definir un modelo arquitectónico que permite la interoperabilidad entre plataformas middleware heterogéneas. Para ello vamos a ver que, utilizando como pilar de desarrollo máquinas abstractas reflectivas orientadas a objetos, la construcción de este tipo de sistemas se simplifica considerablemente y se heredan todas las ventajas que estas aportan: multiplataforma, escalabilidad, independencia del lenguaje y muchas otras que comentamos en la sección 12.2. Por otro lado, vamos a limitar los tipos de middleware sobre los que se puede aplicar el sistema, en concreto, nos centraremos en middleware con Invocación Independiente de la Ubicación (Location Independent Invocation) que denominaremos LII, también llamados RMI, Invocación a Métodos Remota (Remote Methods Invocation). Elegiremos estos sistemas ya que ofrecen una serie de servicios que son homomórficos con los que encontramos dentro de la mayoría de diseños de máquinas abstractas.

En el Capítulo 4, hemos hecho un repaso general a los conceptos básicos de máquinas abstractas. Posteriormente, hemos visto qué beneficios nos aporta el uso de máquinas abstractas. A continuación, hemos revisado cinco máquinas abstractas, tanto comerciales como de investigación, y, finalmente, hemos analizado las características comunes que tienen entre sí, desde el punto de vista de los objetivos que queremos conseguir en nuestra tesis, que son, un área de memoria común, un mecanismo de invocación de métodos y una estructura de representación de objetos.

Estas estructuras de representación de objetos pueden ser consultadas y manipuladas a partir de los mecanismos reflectivos comentados en el Capítulo 5, añadiendo, por un lado la capacidad descriptiva que necesitaremos y, por otro, un grado adicional de flexibilidad a los sistemas basados en máquinas abstractas.

En el Capítulo 6, definimos los sistemas distribuidos, identificamos los objetivos para los que están construidos, vimos las características que deben cumplir, encontramos el problema de la heterogeneidad a nivel de middleware y enumeramos los diversos tipos de sistemas distribuidos que existen. Posteriormente, en el Capítulo 7, analizamos en detalle varias de las plataformas middleware basadas en invocación remota de métodos con la intención de identificar sus características comunes, a saber: servicios de localización de objetos, servicios de descripción de objetos y servicios de invocación dinámica de objetos. Vimos que estas tienen un comportamiento análogo a las de las estructuras esenciales identificadas en las máquinas abstractas reflectivas estudiadas en el Capítulo 4 y Capítulo 5.

A continuación, revisaremos las características que nos ofrecen las máquinas abstractas junto con los requisitos de los sistemas con los que podemos interoperar y comprobaremos que existe una similitud entre ambos. Posteriormente restringiremos el problema a la utilización de tipos básicos y eliminaremos el paso de referencias externas, dejando la solución de todo ello a los desarrolladores concretos del modelo para plataformas específicas. Nuestro modelo no pierde genericidad y se simplifica considerablemente si eliminamos en su diseño estos elementos. El motivo es que dichos elementos simplemente amplían las tareas de *marshaling* o conversión de tipos y suelen incluir muchos criterios de diseño y conversión arbitrarios y dependientes del ámbito de utilización de los sistemas construidos.

11.1 Máquina abstracta como pilar básico de la arquitectura

Nuestro modelo arquitectónico está basado en la extensión o modificación de una determinada máquina abstracta. Con esto añadiremos mecanismos que permitan la interoperabilidad de los objetos de la máquina con los objetos externos y de los objetos externos con los objetos de la máquina. Por lo tanto, nuestro sistema se puede considerar como una máquina abstracta ampliada. Tal es el caso, que la propia máquina se podrá seguir utilizando para la ejecución de los programas desarrollados para ella además de servir como middlebus. Esto siempre dependerá de las características concretas de la misma.

Consideraremos que la máquina es reflectiva, es decir, se puede manipular e inspeccionar su contenido a nivel de programación de usuarios. En el Capítulo 13, se describe cómo añadir la reflectividad a una máquina que no la tenga.

Las características básicas de las que vamos a hacer uso y que posee toda máquina abstracta reflectiva son:

- Área de clases, toda máquina abstracta contiene una estructura de datos con la descripción de las clases que contiene.

- Mecanismos de invocación a métodos, todas las máquinas disponen de alguna instrucción para la invocación a métodos tanto primitivos como de usuario.
- Mecanismos de introspección, las máquinas reflectivas disponen de mecanismos que permiten inspeccionar la estructura de los elementos que contienen e incluso algunas permiten su modificación.

Vamos a construir el modelo arquitectónico para que interopere con sistemas middleware que nos ofrezcan estos tres servicios de tal forma que exista un isomorfismo entre las máquinas y dichos sistemas. En la siguiente sección, veremos qué tipo de sistemas cumplen con estas características y cuales son los servicios que equivalen a las características básicas comentadas anteriormente.

11.2 Requisitos de los sistemas a interoperar

En el Capítulo 6 se describieron los sistemas distribuidos, dentro de los cuales se hallaban los que permitían la comunicación entre procesos, se describieron varias formas de llevar a cabo esta comunicación. El objetivo de este apartado es identificar los sistemas de comunicación entre procesos que mejor se adaptan al modelo que deseamos construir basado en máquinas abstractas. Para ello, vamos a intentar que cumplan con las características que se indicaron en la sección anterior: área de clases, invocación a métodos e introspección.

A continuación describiremos los servicios que deseamos obtener de las plataformas de comunicación distribuida que pueden adaptarse a nuestro middlebus.

11.2.1 Servicio de nombres

Localizar los elementos del sistema a través de alguna técnica de nombrado, ya sea DNS, registro del sistema, repositorio de objetos o directorio de objetos tipo LDAP. Estos servicios tienen isomorfismo con las áreas de clases que utilizan las máquinas abstractas.

11.2.2 Descripción de tipos

Debe tener algún mecanismo de introspección, ya sea descripción de tipos, descripción de interfaces, utilización de lenguajes de descripción de interfaces, etc. De tal forma que podamos obtener información descriptiva del servicio a utilizar y, así, poder crear las clases envoltorios necesaria de forma dinámica. Vemos que este servicio es isomórfico con los mecanismos reflectivos en las máquinas abstractas.

11.2.3 Invocación dinámica

Es necesario un protocolo de comunicación dinámica del tipo DII, invocación dinámica o protocolo de invocación remota. Estos mecanismos son isomórficos con las invocaciones a métodos dentro de una máquina abstracta.

Como estudiamos en la sección 7.4 vemos que los anteriores servicios los ofrecen las plataformas RMI, con lo cual dichas plataformas serán idóneas para adaptarlas a nuestro sistema. Si bien, puede resultar relativamente sencillo adaptar otros mecanismos de comunicación entre procesos como los orientados a eventos o punto a punto. En este caso, muy posiblemente, el desarrollador deba desviarse de nuestro modelo arquitectónico y realizar algún tipo de modificación u adaptación concreta del mismo.

11.3 Manipulación y conversión de tipos

Consideraremos la interoperabilidad a nivel de tipos básicos: enteros, reales, booleanos, cadenas, etc. Durante el diseño de modelo arquitectónico no haremos prácticamente referencia a ellos, dejando la elección de los mismos a los implementadores de los sistemas concretos. La manipulación de todos estos tipos lleva consigo la conversión de representación de unas plataformas a otras, lo que se conoce en la literatura anglosajona como *marshaling*. La realización de dichas conversiones está contemplada en el modelo, si bien los criterios sobre cómo realizar las mismas se dejan a la elección de los desarrolladores. Dichos criterios no afectan a nuestro modelo y si los tratásemos podrían, por un lado, complicar y aumentar el modelo y, por otro, crear conflicto de criterios con implementaciones concretas, mientras que arquitectónicamente no aportan información representativa. El desarrollador será libre de ampliar los tipos utilizados por el sistema, pudiendo este utilizar tipos estructuralmente más sofisticados como clases, vectores, árboles, etc. Nuestro modelo no deberá en ningún caso verse afectado por estas ampliaciones.

11.4 Referencias externas

La utilización o manipulación de referencias sería un caso concreto del apartado anterior, considerando las referencias como un tipo más. Si bien en el diseño de nuestro modelo hemos optado por no manipularlas y adopta el criterio de que no se considera la utilización de objetos ya existentes, sino que se crearán instancias de las clases dentro del propio sistema durante las invocaciones.

Este criterio lo hemos adoptado para tener una mayor compatibilidad con los sistemas middleware basados en Servicios Web, donde no existe el concepto de referencia ni de contextos de ejecución de forma estandarizada, si bien cada implementador puede utilizar sus mecanismos para conseguir esto. Muy posiblemente, esta ausencia sea debida a la falta de madurez de esta tecnología y a la carencia de una capa superior, más abstracta, que no esté tan cerca de la comunicación de paquetes basada en XML, como lo están SOAP, WSDL y el resto de protocolos en los que se sustenta.

No obstante, queda en manos del desarrollador seguir estos criterios o utilizar los que el considere más oportuno. Aunque, en este último caso, al contrario de la manipulación de tipos, sí sea necesario realizar pequeñas modificaciones al modelo arquitectónico.

Estas son las restricciones y características de los sistemas en las que nuestro modelo arquitectónico está basado. En las siguientes secciones describiremos más detenidamente tanto la máquina abstracta como el modelo arquitectónico en sí.

Capítulo 12:
Una máquina abstracta orientada a objetos

En el Capítulo 10 hemos visto los requisitos que deseamos que cumplan los sistemas construidos a partir de nuestro modelo arquitectónico para crear middlebuses. En el Capítulo 11 hemos restringido el dominio del problema y hemos tomado determinadas decisiones sobre la forma de llevarlo a cabo. Hemos optado por sistemas con una serie de características, a saber, invocación dinámica, introspección y localización, llegando a la conclusión de que los sistemas distribuidos RMI, también denominados LII, son los más adecuados, ya que dichos sistemas tienen una relación isomórfica con las máquinas abstractas orientadas a objetos. Hemos restringido el problema a los tipos básicos y decidido no considerar directamente las refencias.

En este capítulo comentaremos las consecuencias que conllevan tales decisiones, como son la orientación a objetos y las máquinas abstractas. Definiremos una máquina abstracta básica de alto nivel que nos servirá de pilar sobre el que construir nuestro modelo arquitectónico. Esta máquina es una simplificación que recopila las propiedades principales de las máquinas abstractas y de orientación a objetos.

A continuación, analizaremos las ventajas aportadas por la orientación a objetos. Posteriormente, mostraremos el diseño de nuestra máquina abstracta (MA) básica de alto nivel. En el Capítulo 13 comentaremos la necesidad de añadir reflectividad a la máquina y realizaremos el diseño para llevar a cabo esta tarea. A partir de ahí estaremos listos para comenzar con el diseño de nuestro modelo arquitectónico, ya que tendremos los pilares necesarios para ello.

12.1 Aporte de la orientación a objetos

Son muchos los paradigmas de desarrollo que actualmente se utilizan, tales como imperativos, funcionales, lógicos, concurrentes, orientados a objetos, además de otros emergentes, como el paradigma orientado a componentes. Pero de todos estos es el orientado a objetos el que, hoy por hoy consigue una mayor aceptación y tiene una mayor madurez, debido a las ventajas que ofrece con respecto a otros paradigmas. Además, la orientación a objetos se puede integrar dentro de otros modelos, por ejemplo el paradigma funcional permite perfectamente que se use dentro de él el orientado a objetos, tal es el caso de Haskell con su versión orientada a objetos O'Haskell [165].

A pesar de que el paradigma orientado a componentes pretende solventar las limitaciones que actualmente tiene la orientación a objetos, aún no ha madurado lo suficiente como para considerarse una disciplina estable, fiable y madura.

Veamos las ventajas que nos ofrece la orientación a objetos:

- Permite un proceso homogéneo y continuado de desarrollo, es más, se puede extraer el modelo de objetos del mundo del software al mundo real. De hecho, existe todo un proceso de desarrollo de objetos orientado a negocios [Ivar95] que utiliza técnicas de análisis, diseño y de modelado orientado a objetos de empresas dentro de la que se incluye el propio proceso de desarrollo orientado a objetos software.
- Avanza un paso en la definición de estructuras de datos, aumentando el nivel de abstracción conseguido. Para ello utiliza mecanismos como la herencia, la encapsulación, y el polimorfismo.
- Nos permite definir jerarquías de elementos con los que podemos atacar los problemas complejos, descomponiéndolos y reutilizándolos en los niveles superiores (herencia).
- Nos permite descomponer el problema en varios módulos que interaccionan débilmente entre ellos y que internamente están fuertemente acoplados. Este es otro mecanismo para poder afrontar problemas complejos (modularidad).
- La orientación a objetos nos permite identificar patrones repetidos, abstraerlos y reutilizarlos, para ello podemos utilizar la herencia, el polimorfismo o la genericidad.
- Nos permite manejar la evolución de los diferentes elementos software, si bien, este paradigma no lo soluciona por completo y es la orientación a componentes la que pretende solventarlo.

Los elementos que componen el modelo de un sistema OO están formados por combinaciones de modelos estáticos *vs.* dinámicos y lógicos *vs.* físicos. Estos conceptos no son tenidos en cuenta en otros paradigmas.

12.2 Aporte de las máquinas abstractas

A continuación, enumeraremos las características aportadas, por las máquinas abstractas, extraídas de [Ortin01], viendo como estas se adecuan a los requisitos deseados de los sistemas construidos con nuestro modelo arquitectónico:

- Independencia del lenguaje de programación, esto nos va a permitir extender nuestro modelo en cualquier lenguaje, independientemente del lenguaje base que se eligió para construir la máquina, así como del lenguaje utilizado para la compilación de todas las clases iniciales y fundamentales en el modelo arquitectónico.
- Portabilidad de su código, de tal forma que podremos ejecutar cualquier instancia de nuestra arquitectura en otras plataformas, sin necesidad de ajustes adicionales, lo único que necesitaremos será un intérprete o un compilador JIT de la máquina que utilicemos.
- Independencia de la plataforma, tanto el modelo arquitectónico como los sistemas construidos a partir de él se podrán ejecutar en toda plataforma para la que exista la posibilidad de portar el código, es decir para las plataformas para las que se implemente la máquina abstracta.
- Elección del nivel de abstracción base en función del paradigma seleccionado para el sistema global. Este nivel de abstracción suele se más alto, facilitando así la compilación y eliminando desajustes de impedancias.
- Interacción única entre aplicaciones, utilizando un único modelo de computación, de tal forma que podemos construir aplicaciones paralelas que se apoyen en nuestro sistema, para la interoperabilidad sin necesidad de ningún mecanismo común de comunicación entre procesos complicado de utilizar e ineficiente. Lo ideal sería que todas las aplicaciones de todos los entornos se pudieran comunicar como lo hacen las aplicaciones dentro de una misma máquina abstracta, por supuesto, con los mecanismos de seguridad apropiados [Diaz00]: de forma sencilla, transparente y eficiente. Esto es lo que se pretende conseguir de forma global y es donde esta tesis aporta su granito de arena.
- Programación interactiva y continua, que nos va a permitir poder manipular los sistemas construidos de tal forma que podamos añadir nuevos elementos o modificar los existentes en cualquier momento. Así, podremos añadir elementos de monitorización de determinadas tareas o de filtrado de determinados mensajes, de registro de sucesos, de corrección o ampliación de mensajes, de compresión de mensajes, cifrado de mensajes, etc. No obstante, en algunos casos para llevar a cabo esto, deberemos ampliar la máquina con algún mecanismo reflectivo como se comenta en el Capítulo 13.
- Distribución de aplicaciones, los sistemas construidos con máquinas abstractas son susceptibles de ser movidos de un entorno a otro, de tal forma que su instalación consistirá simplemente en descargarse el código que los contiene.

- Interoperabilidad nativa de aplicaciones distribuidas, con las máquinas abstractas se facilita la construcción de sistemas interoperables distribuidos, aportando así una capacidad para la escalabilidad del sistema desarrollado. Esta característica no se contempla en nuestra máquina básica debido a su complejidad [Alvarez00], si bien las máquinas reales sí la aportan y puede ser aprovechada.
- Desarrollo de servicios operativos mediante código único, portable y distribuible, esta característica no aporta mucho a los sistemas construidos con nuestra arquitectura, si bien es muy interesante para la construcción de sistemas operativos distribuidos, persistentes y seguros.

A continuación mostramos el diseño y detalles de implementación de una máquina abstracta básica que pretende ser una simplificación de las máquinas comerciales, sin perder las características y aportes comentados en las dos secciones anteriores, y que nos servirá de pilar fundamental de nuestro modelo arquitectónico.

12.3 Máquina abstracta básica

12.3.1 Introducción

Nuestro modelo arquitectónico pretende construir sistemas que permitan la interoperabilidad heterogénea a partir de máquinas abstractas, para, de esta forma, obtener todas las ventajas que estas aportan, como se comentó en la sección 12.2, así como de la orientación a objetos, como se indicó en 12.1. En este apartado pretendemos describir y mostrar el diseño de una máquina abstracta básica que, si bien no resulta eficiente y carece de muchas otras características necesarias para ser una máquina comercial, cumple con las características mínimas y básicas comentadas anteriormente, además de aportar un alto grado de abstracción y, por consiguiente, simplicidad, tanto de comprensión, como de utilización.

A continuación, indicaremos las características concretas de muestra máquina abstracta, que denominaremos de forma abreviada MA, y en las restantes secciones describiremos cada una de sus partes.

12.3.2 Características

Para construir nuestro modelo arquitectónico de patrones basado en máquinas abstractas orientadas a objetos, necesitamos definir un modelo básico y fundamental de nuestra máquina. Para ello, considerando la analogía entre teoría de lenguajes y máquinas lógicas, nos basaremos en los modelos básicos que estos definen para construir interpretes orientados a objetos, que no deja de ser una simplificación de una máquina abstracta de alto nivel. Las características de esta máquina básica están presentes de forma más o menos explícita en todas las máquinas lógicas y virtuales comentadas en el Capítulo 4, dedicado a máquinas abstractas.

Si bien toda máquina abstracta está basada en una máquina lógica definida por su lenguaje, en nuestro caso y con intención de conseguir una independencia máxima de cualquier lenguaje de programación, para así obtener una máxima genericidad y un alto nivel de abstracción, hemos considerado no asociarla directamente con ningún lenguaje en concreto, de hecho se han implementado minicompiladores de varios lenguajes, como Java, C++, Smalltalk e, incluso, un nuevo lenguaje ideado en español TranquiTal, específicos para esta máquina.

A continuación, se describen las características más importantes del lenguaje abstracto que interpreta y, posteriormente, describiremos la máquina a través de su modelo arquitectónico de diseño orientado a objetos mediante diagramas UML.

Veamos las características que cumple nuestra máquina abstracta general básica de alto nivel de abstracción:

- Comprobación débil de tipos, no se verifican tipos en tiempo de compilación, de tal forma que si en tiempo de ejecución algún objeto recibe un mensaje cuya clase o ancestros de la misma no tienen implementado se producirá el correspondiente error de ejecución. Esta es una característica que hace la máquina poco robusta a la hora de construir sistemas, sin embargo, la flexibilidad y sencillez que aporta nos ha hecho decantarnos en este sentido. Hay que destacar que estas propiedades de comprobación de tipos estricta las pueden añadir los compiladores de lenguajes con tipado estricto tanto a nivel de compilación como a nivel de ejecución.
- Debido a la comprobación débil de tipos, se puede prescindir de la declaración de variables temporales, sólo se declararán las variables pertenecientes a la instancia o los atributos de la clase, incluso se les asignará un tipo y, si es una clase primitiva, se le podrá dar un valor inicial. En el caso de las variables de los parámetros de los métodos sólo se declarará su nombre.
- Permite herencia simple explícita, polimorfismo y herencia de interfaces implícita.
- Debido al concepto de herencia implícita de interfaces no se ha considerado necesario una clase común `OBJECT` explícita, de la que derivan todas las clases y de la que heredan un comportamiento básico, como ocurre con otras máquinas.
- Enlace tardío de mensajes, no se sabe a qué clase pertenece el mensaje hasta el momento de su ejecución.
- Esta máquina no dispone de sobrecarga de métodos dentro de una misma clase, si bien no sería difícil implementarla realizando la búsqueda de los métodos además de por su nombre por los tipos de los parámetros. Aunque sí se pueden redefinir métodos heredados dentro de una jerarquía de clases. Esta última capacidad es una característica de los sistemas orientados a objetos que permite especializar las clases descendientes o hijas redefiniendo los métodos de los ascendentes o padres.

- No existen instrucciones en sí, todo son expresiones que devuelven un valor u objeto. Estrictamente hablando, todo son objetos es decir instancias de una clase. Más adelante se comentarán las expresiones que implementa esta máquina. La invocación a métodos es un tipo de expresión, cuando se invoca un método de usuario, este devolverá lo que devuelva la última instrucción que se evalúe. No existiendo instrucción o expresiones de ruptura de flujo como *return*, *exit*, *break* y demás que otros sistemas aportan.
- No se diferencia entre tipos primitivos y tipos derivados de clases. Todos los tipos son derivados de clases, de tal forma que no hay que realizar *boxing*, *typecasting* o adaptación de tipos.
- Todas las variables son por referencia. Es decir, todos los nombres utilizados son referencias a instancias internas. En nuestra implementación estas referencias vienen dadas y gestionadas por el propio lenguaje de implementación Smalltalk, que, a su vez, está basado en referencias.
- Al igual que en la CLI de .NET [Nutt03], se implementan hilos de ejecución aprovechando las características de multitarea del sistema subyacente donde se implemente la máquina.
- Al igual que en Smalltalk, todos los atributos son privados y todos los métodos son públicos, por lo que habrá que construir los oportunos métodos de acceso a miembros.

El diseño y posterior implementación de un prototipo sencillo de esta máquina, han sido realizados en Smalltalk dentro del entorno Squeak con la imagen 3.7 [Squeak05], por lo que el código que aparecerá a lo largo de la descripción de la misma estará en este lenguaje. Se ha elegido Smalltalk y Squeak por varios motivos:

- Es un entorno de libre distribución y de domino público, diseñado y construido por el grupo liderado por el reciente premio Turing y líder del grupo de investigación de los laboratorios de Xerox en Palo Alto que inventaron Smalltalk, Alan Kay [Gold83].
- Es multiplataforma, Squeak está implementado para más de 27 plataformas diferentes, todas ellas compatibles entre sí, desde PDA, Unix, Linux, Solaris, Windows, PalmOS, hasta sistemas operativos de móviles como Symbían.
- El lenguaje de programación Smalltalk es un lenguaje orientado a objetos puro, es decir, todos los elementos del mismo son objetos, al contrario que Java o C++ que son híbridos, con lo que resulta muy simple poner en práctica los conceptos de orientación a objetos y desarrollar sistemas con este paradigma.
- Es un lenguaje interpretado dentro de un entorno interactivo, de tal forma que el prototipo estaba continuamente compilado y listo para ejecutarse, probarse y depurarse.
- Es un entorno reflectivo, con lo que se puede ver en todo momento la estructura y contenido de la máquina. Los objetos, *inspector* y *browser*, son los más utilizados con tal fin.

- Smalltalk es sencillo y fácil de aprender, está basado en el envío de mensajes, de los que existen tres tipos: monarios, mensajes sin parámetros; binarios, utilizados para los operadores aritméticos; y basados en claves, donde cada palabra clave lleva asociado un parámetro.
- Squeak es uno de los entornos de Smalltalk. El entorno gráfico de Squeak se denomina Morphic y fue diseñado por [Maloney00] cuando trabajaba en el proyecto Self [Ungar91]. Morphic es abierto, fácil de aprender y utilizar. Además lleva implícito un entorno de desarrollo rápido y muy sencillo, debido a la potencia expresiva y capacidad reflectiva de Smalltalk.

Toda máquina consta de una jerarquía básica de clases que se puede abstraer de distintas formas, pero que, al final, tratan de modelar lo mismo que nuestro modelo básico, es decir, los conceptos básicos de la orientación a objetos, a saber, clases, instancias, métodos, herencia, polimorfismo, encapsulación, genericidad y abstracción. Además de otras abstracciones típicas de todo lenguaje de programación como son las estructuras de control, las variables y las subrutinas.

12.3.3 Clases e Instancias

El siguiente diagrama de clases, Ilustración 12.1, modela la estructura interna de la máquina, ideada para la gestión de instancias y clases.

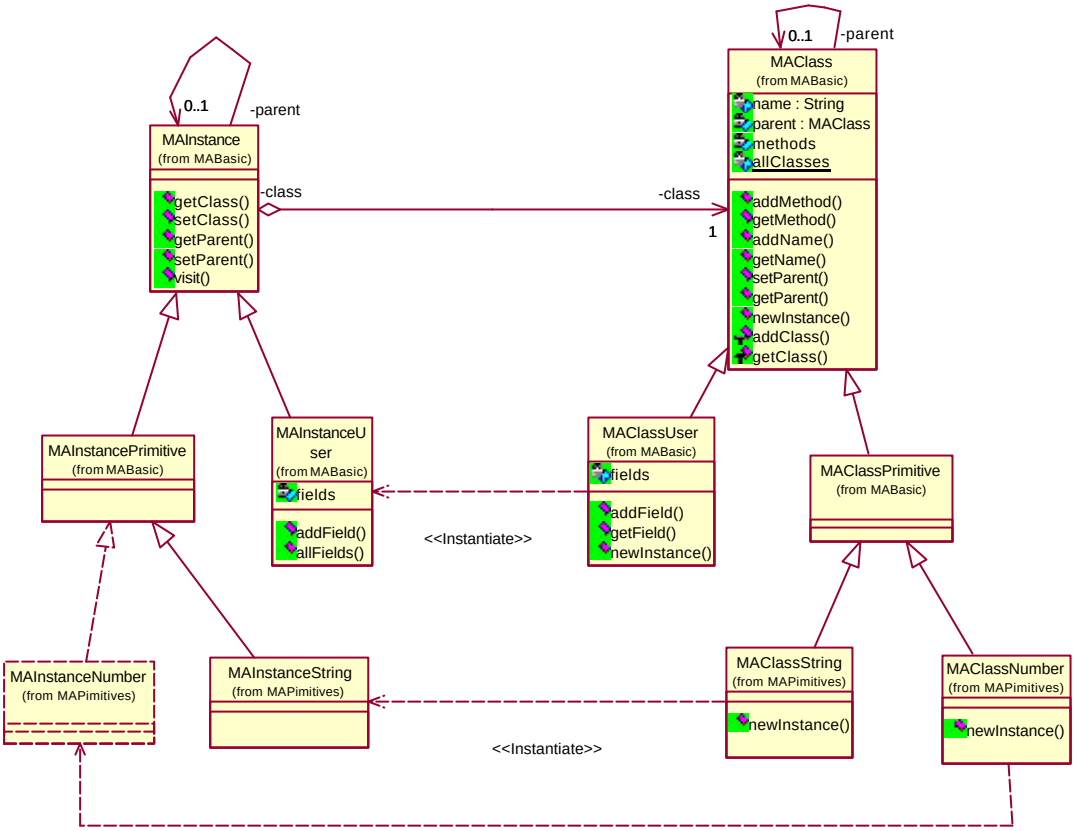


Ilustración 12.1: Clases e Instancias

Se pretende modelar el hecho de que toda clase está formada básicamente por un conjunto de métodos y debido a la herencia está relacionada con su clase padre. De ahí los atributos `parent` y `methods`, además de tener un nombre, atributo `name`.

Desde el punto de vista de las instancias, una instancia viene determinada por la clase a la que pertenece, de ahí el atributo `class`, que en este caso viene dado por el rol `class` en el diagrama.

Al igual que en todo sistema, va a haber dos tipos tanto de instancias como de clases, estas son:

- Las primitivas, es decir, las que están en la máquina nada más esta comienza a ejecutarse y
- Las definidas por el usuario, que son las que pueden ser creadas, ampliadas y modificadas de forma explícita a partir de las instancias y clases primitivas.

De todos los métodos que se exponen en el diagrama, el más importante es `newInstance`, de la clase `MAClass` que debe ser redefinido por toda clase derivada de esta. Este método se encarga de construir la instancia apropiada de la clase a la que pertenece. Este método implementa el patrón *método fábrica* [Gamma00] y con él se

pueden crear instancias de una clase sin necesidad de conocer la clase o la instancia concreta a la que se hace referencia.

En todo sistema orientado a objetos debe existir como mínimo un contenedor de todas las clases que se han definido y pueden ser utilizadas. Cada sistema lo implementa de una manera, tanto de forma pública como de forma privada. En nuestra máquina este contenedor es la propia clase `MAClass` a través de un atributo de clase o atributo estático denominado `allClases`. En nuestro caso es un diccionario de *Smalltalk* con todas las clases del sistema. Sería el equivalente al diccionario `Smalltalk` que contiene Squeak en el que es posible encontrar todas las clases definidas en el sistema, el registro del sistema en entorno Windows o el repositorio de interfaces en CORBA. En este último caso, este es público. Para acceder a ese atributo se utilizan dos métodos de clase `addClass` y `getClass`.

De todas las clases derivadas de `MAClass`, con excepción de `MAClassUser`, debe haber una única instancia que representa una clase de la máquina. Para conseguir esto hemos seguido las indicaciones del patrón de diseño *Singleton* o Único [Gamma00]. Lo primero que hace el sistema al comenzar e inicializar la clase `MAClass` es crear una instancia de cada una de las clases primitivas: `MANumber`, `MAString`, etc. y añadirlas al contenedor `allClases`. De esta manera están disponibles dentro del sistema. Así, si deseamos crear una instancia de un número entero escribiremos:

```
i := (MAClass getClass: 'NUMBER') newInstance setValue: 5
```

A través de `MAClass getClass: 'NUMBER'` obtenemos la instancia única de la clase `MAClassNumber`, posteriormente llamamos a `newInstance` para que internamente se llame al constructor de `MAInstanceNumber` y, por último, inicializamos esa instancia al valor 5 enviándole el mensaje `setValue`. El entero definido lo dejamos en la variable `i`.

12.3.4 Métodos

Veamos otra jerarquía de clases, Ilustración 12.2, quizás la más extensa de todas las que contiene la máquina. Nos permite representar los métodos pertenecientes a una determinada clase. Al igual que en el caso de las instancias y las clases, también se definen métodos primitivos y de usuario, `MAMethodPrimitive` y `MAMethodUser` respectivamente, estando ambos derivados de una clase abstracta común denominada `MAMethod`.

Se pretende modelar un método como un elemento que representa el nombre del mismo, otro que representa la clase a la que pertenece y, por último, una colección de los nombres de los parámetros que se le pueden pasar a ese método. En nuestro caso, los parámetros no tienen tipo debido a la característica de la máquina, que es de comprobación débil de tipos. Por otro lado, cada método puede ser invocado, ejecutado

o evaluado a través de su mensaje `invokeInstance`, del que hablaremos más detenidamente a continuación.

La diferencia fundamental entre los métodos primitivos y de usuario es que estos últimos tienen un atributo adicional `seq` que representa una expresión que, a su vez, hace referencia al programa interno, que se ejecuta cuando lo invocamos a través del método mencionado anteriormente. Veamos en el diagrama de clases siguiente la relación jerárquica de la que hablamos:

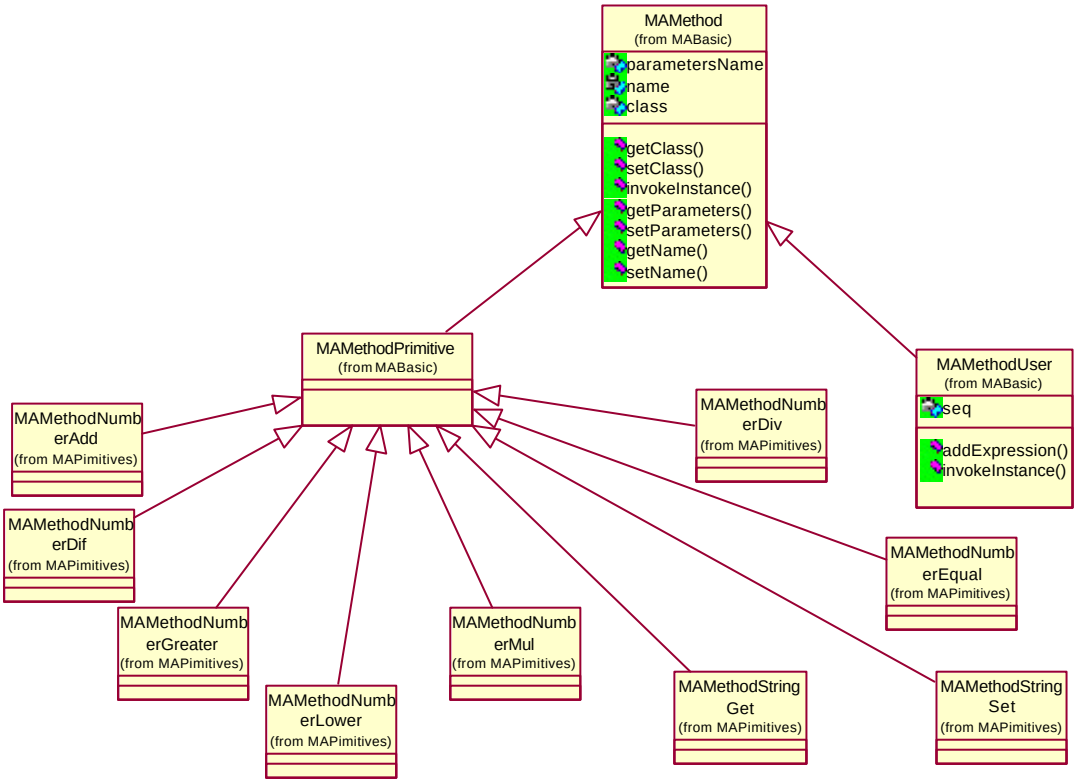


Ilustración 12.2: Métodos

De la clase `MAMethodPrimitive` derivan directamente todos y cada uno de los métodos que pertenecen a las clases primitivas. Vemos que en el diagrama tenemos los métodos de la clase `Number` que realizan las operaciones aritméticas `MAMethodNumberADD`, `MAMethodNumberDif`, `MAMethodNumberDiv`, `MAMethodNumberMul`, los que realizan las operaciones de comparación `MAMethodNumberEqual`, `MAMethodNumberLower`, `MAMethodNumberGreater`. Para la clase `String` y para el resto de clases primitivas, ocurrirá lo mismo. El análisis de cómo se implementan estas clases se verá más adelante, ya que va a resultar muy útil en la construcción de fábricas dinámicas de clases primitivas dentro de la propia máquina, que realizarán un *wraper* de las clases externas con las que deseamos interoperar de forma dinámica, es decir, en tiempo de ejecución.

Veamos en más detalle el método principal de esta jerarquía, para ello mostramos su signatura:

```
MAMethod>>invokeInstance: i withParameters: p
```

Observamos que recibe dos parámetros `i` y `p.` El primero representa la instancia sobre la que se desea evaluar el método en cuestión y el segundo es una colección de instancias que servirán como parámetros asociados a ese método. Un aspecto a tener en cuenta es que tanto el método que deseamos invocar como la instancia `i` sobre la que queremos realizar los cálculos deben pertenecer a la misma clase o a alguna clase ascendente.

Una implementación especial de este método es la correspondiente a la clase de usuario, veamosla un poco más en detalle:

```
MAMethodUser>>invokeInstance: i withParameters: p
      "Creamos el evaluador/visitor/contexto a partir de la instancia
y los parametros "
      | e |
      e := MAEvaluator new.
      "Definimos todas las variables de la instancia y de sus padres"
      i allFields: e symbolTable.
      "Luego pasamos de parametros formales a parametros reales"
      parametersNames
            keysAndValuesDo: [:j :n | e symbolTable
                        at: n
                        put: (p at: j)].
      "Añadimos la variable this y super"
      e symbolTable at: 'THIS' put: i.
      e symbolTable at: 'SUPER' put: i parent.
      "Finalmente evaluamos las expresiones del
      método con el evaluador/visitor/contexto e"
      ^ seq visit: e
```

Para poder ejecutar un método de usuario es necesario crear un contexto de ejecución. En el caso de nuestra máquina abstracta básica, este se ha simplificado a su mínima expresión, lo que detallaremos en la siguiente sección, de momento, nos quedaremos con que es la instancia `e` perteneciente a la clase `MAEvaluator.` Lo que resulta interesante ver ahora es que dentro de él se deben crear todos los atributos de la instancia en la que se ejecuta el método así como el de sus ancestros con sus actuales valores. Esto se realiza con una simple llamada a `allFiels` en la instancia `i`, posteriormente, se añaden todos los parámetros, nombrándolos según al colección `parametersNames`. Se suman dos variables adicionales, `this` y `super,` la primera representa la instancia sobre la que se está ejecutando el método y la segunda referencia a su ancestro. Una vez definidas todas las variables, está listo el método para ser evaluado o ejecutado, lo que se realiza llamando al método `visit` de la expresión `seq` del método de usuario con el contexto `e` como parámetro.

En la siguiente sección veremos en qué consisten las expresiones y cómo se evalúan. Nos queda comentar, por último, que `seq` es una expresión especial, denominada *secuencia*, que está formada por una colección de expresiones y que consistirá en el programa que hay construido dentro del método de usuario que la contiene.

12.3.5 Expresiones

Como ya comentamos anteriormente, esta máquina carece de instrucciones en sí. En su lugar todo son expresiones que devuelven algún valor u objeto, estrictamente hablando se trata de valores, todo son objetos. No existen instrucciones para acceso a miembros de las clases ya que dichos miembros son privados, solo podemos invocar métodos o enviar mensajes. Todas las expresiones derivan de una clase abstracta `MAExpression` que define un único método `visit`. Este método permitirá evaluar la expresión o realizar otro tipo de acciones sobre las expresiones. Realmente, lo que hemos hecho es implementar el patrón *Visitor* o *Visitante* [Gamma00]. Se podría haber simplificado el diseño de la máquina sin utilizar este patrón, definiendo el método `evaluate` en vez de `visit`, no obstante, debido a la gran flexibilidad que nos aporta, decidimos que era la mejor solución. No obstante a lo largo de esta sección consideraremos la implementación de `visit` como evaluación.

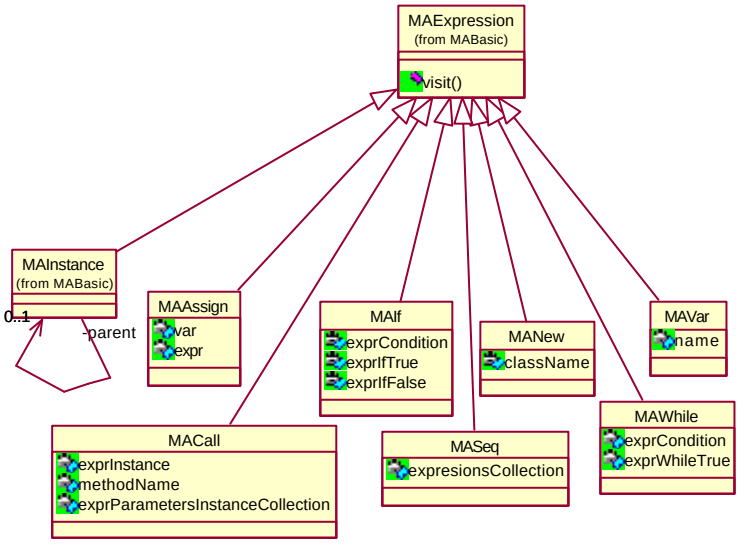


Ilustración 12.3: Expresiones

Las clases que modelan las expresiones que pueden ser utilizadas en esta máquina abstracta son las que se muestran en el diagrama y que procedemos a comentar a continuación:

- `MAVar` nos permite manipular variables o referencias dentro de la máquina a partir de un nombre. Su evaluación nos devolverá la instancia a la que representa o hace referencia.
- `MAAssign` nos permitirá asignar un objeto resultado de evaluar una expresión a una variable dada. Nos devuelve el objeto asignado.

- `MAIf` es una instrucción de control típica de comparación de todo lenguaje de programación de alto nivel. Se evaluará la expresión de condición `exprCondition` y en función de su valor, si es cierto se evaluará la expresión `exprIfTrue` y, en caso contrario, la expresión `exprIfFalse`. Devolverá el objeto resultado de la expresión evaluada.

- `MAWhile` representa la clásica instrucción de iteración. Se irá evaluando la expresión `exprCondition` y mientras el resultado de esta evaluación se considere cierto se evaluará `exprWhileTrue`, devolviendo como resultado el objeto devuelto en la última iteración en que la condición era verdadera.

- `MASeq` es una colección de expresiones. Su evaluación consiste en la evaluación de todas las expresiones que hay dentro de la colección `expresionsCollection`, devolviendo como resultado el valor devuelto por la última expresión.

- `MANew` dado el nombre de una clase crea una instancia perteneciente a ella. Esta instrucción encapsula una invocación implícita a un método new. Se podría haber evitado expresión, simplemente añadiendo este comportamiento a la clase raíz OBJECT o a una instancia global denominada `Área de Clases` como ocurre en Oviedo3 [Cueva96]. No obstante por motivos de simplicidad y sencillez inicialmente la hemos construído como una instrucción completamente independiente. Veremos, cuando estudiemos la sección de reflectividad, que es una operación reflectiva y que se podrá implementar realizando llamadas reflectivas a alguna clase cosificada, en nuestro caso, `MAClass`.

- `MACall` realiza una invocación a un método dado sobre una instancia determinada, pasándole los parámetros necesarios. Devuelve como resultado el valor devuelto por `seq` del método invocado, que, como ya hemos comentado más arriba, corresponde al último valor evaluado dentro de la secuencia. Observemos que la implementación de esta función consiste, simplemente, en el envío del mensaje `invokeInstance` del método sobre el que queremos realizar la llamada. En nuestro caso, esta instrucción no guarda un acceso al método, sino su nombre. El método en sí lo obtendremos buscándolo dentro de la clase a la que pertenece la instancia sobre la que se invoca, de tal forma que nos garantizamos que método e instancia pertenecen a la misma clase. Por último, resaltar que esta implementación, junto con la comprobación débil de tipos nos va a permitir llamar a métodos de instancias no pertenecientes a la misma jerarquía de clases, cosa que en los lenguajes fuertemente tipados no se puede hacer. Esto nos ayuda a solventar la limitación de diseño impuesta por la herencia simple, permitiéndonos la máquina implementar interfaces implícitas y así como la cooperación con otras clases.

- `MAInstance` en sí no es una instrucción, sino que representa los valores constantes y resultado de evaluaciones de las otras instancias. Destaquemos también que esta clase pertenece a la jerarquía de instancias `MAInstance`. Desde el punto de vista de su implementación

resaltar que a la máquina abstracta de *Smalltalk* le ocurre lo mismo que a nuestra máquina básica, es decir, se pueden llamar a métodos de clases que no pertenecen a la jerarquía. Aunque en el diagrama indique que `MAInstance` hereda de `MAExpression`, en realidad no es así, simplemente implementa el mensaje `visit`, no es necesario que pertenezca a la misma jerarquía para permitir el polimorfismo. Con esto se solucionan las restricciones impuestas con la herencia simple en *Smalltalk* que comentamos en el apartado anterior.

Un programa consiste en la anidación de unas expresiones dentro de otras, formando un árbol semántico, que será evaluado a través de un recorrido realizado por un objeto evaluador, derivado de una clase abstracta principal denominada `MAVisitor`. A continuación, mostramos una jerarquía de clases visitantes del árbol semántico. De los posibles programas construidos a partir de las expresiones de la máquina, la utilidad más importante es `MAEvaluator`, que veremos en detalle en el siguiente apartado; pero también tenemos `MADebuger` que permite una ejecución paso a paso, `MAPrinter,` que nos permite imprimir nuestro programas, `MAFile,` que nos permite guardar el programa en un fichero, y `MASocket,` que nos permite enviar el programa a través de una conexión de red, de tal forma que sería muy sencillo construir sistemas móviles. Se podrían crear otras clases visitantes que permitieran realizar comprobaciones estrictas de tipos, conversión del programa al lenguaje de otra máquina, filtrado de la ejecución, etc.

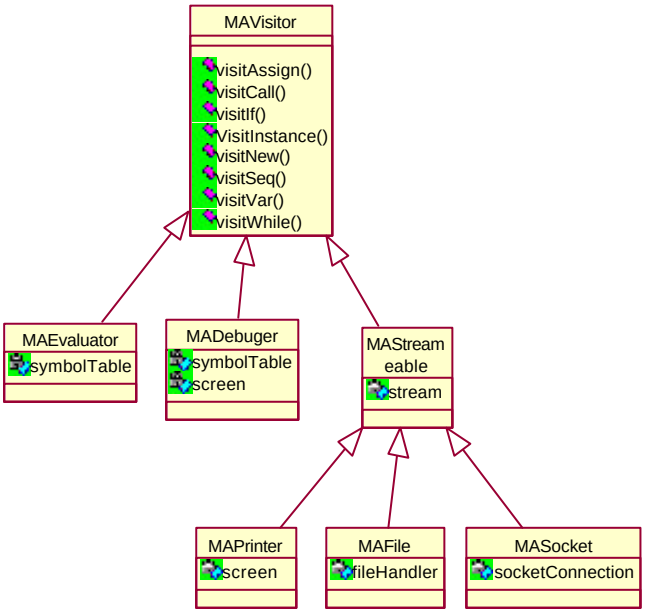


Ilustración 12.4: Jerarquía Visitador

Este árbol semántico lo construyen los compiladores de determinados lenguajes. Veamos un ejemplo de árbol semántico, Ilustración 12.5, correspondiente aun método llamado `CalculaTotal`.

Esta forma de representar los programas internos dentro de la máquina es de alto nivel y nos permite tener construcciones con un alto grado de expresividad y sencillez

como, por ejemplo, `MAIf` y `MAWhile`, que es lo que se pretendió cuando se diseño. Cierto es que esto redunda en pérdida de eficiencia, pero nuestro objetivo es la claridad, simplicidad y generalidad de la máquina.

Por cada clase dentro de la jerarquía de expresiones debe existir un método que lo visite dentro de las clases de la jerarquía de visitantes. La implementación de los métodos `visit` dentro de las clases derivadas de `MAExpression` será una simple llamada al método que la visita, de tal forma que cada visita puede ser una acción independiente sobre la expresión.

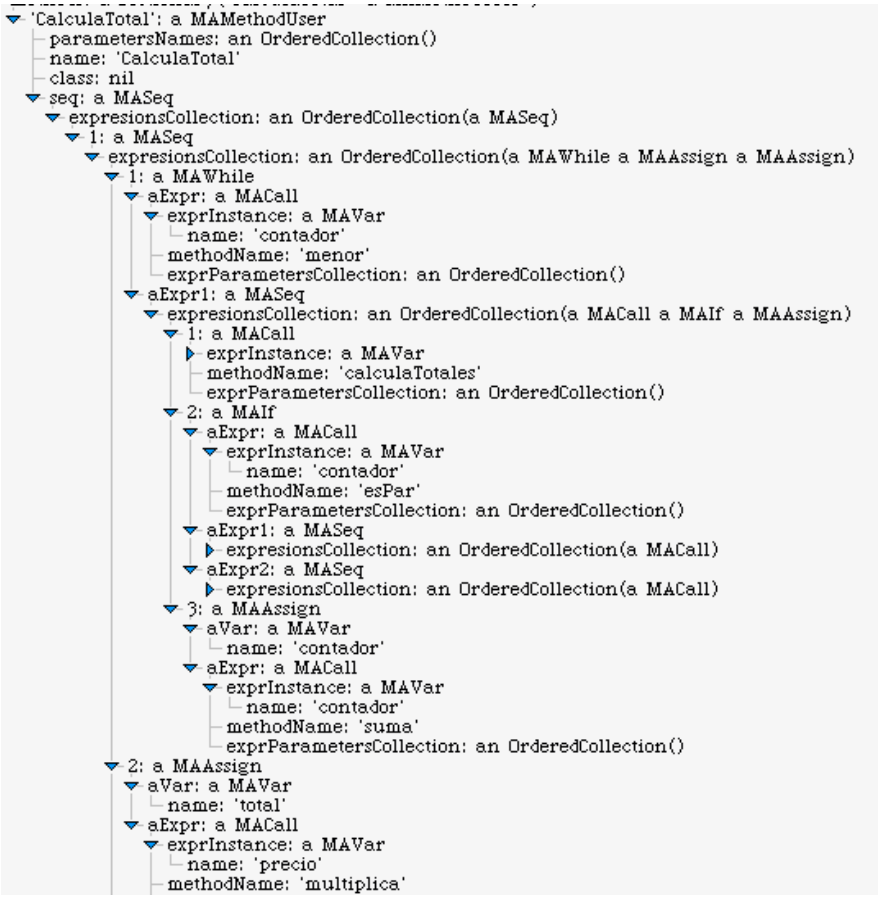


Ilustración 12.5: Árbol Semántico

Por ejemplo, para cada clase derivada de `MAExpression` existe un método en la clase `MAEvaluator` derivada de `MAVisitor` que la evalúa, interpreta o ejecuta a través del método `visit`. Para la clase `MACall` la implementación, en *Smalltalk*, de `visit` es:

```
MACall>>visit: aVisitor
 ^aVisitor visitCall: self.
```

El método `visit` recibe como parámetro una instancia de la clase que realizará la visita. En este caso, `aVisitor` va a ser una instancia de `MAEvaluator`, que

tendrá un método llamado `visitCall`, que recibe a su vez, como parámetro la propia instancia de la expresión que estamos evaluando, que pertenece a la clase `MACall` y se denomina, en *Smalltalk*, `self`.

De igual modo ocurrirá con el resto de clases derivadas de expresión: `MAVar`, `MASeq`, etc. Con ello, lo que estamos haciendo es delegar el comportamiento de esta clase a las clases derivadas de `MAVisitor`. Así, una posible implementación del método `visitCall` dentro de la clase `MAEvaluator` sería:

```
MAEvaluator>>visitCall: aCall
      | instance parameters method |
      instance := aCall exprInstance visit: self.
      "Look for the method in the class of instance"
      method := instance getClass getMethod: aCall methodName.
      parameters := aCall exprParametersCollection
                        collect: [:p | p visit: self].
      ^ method invokeInstance: instance withParameters: parameters
```

En este caso, `aCall` hace referencia al `self` del código anterior, es decir, a la instancia de la clase `MACall` que estamos evaluando.

El procesamiento de esta clase consiste en evaluar la expresión que nos devolverá la instancia sobre la que se desea invocar el método al que estamos llamando, que en el código se denomina `instance`. Para ello, visitamos `exprInstance` con nuestro evaluador, en este caso, `self`. Una vez obtenida la instancia vamos a buscar un método dentro de la clase a la que pertenece cuyo nombre coincida con el que queremos invocar. Con este fin, le enviaremos el mensaje `getMethod` a la clase a la que pertenece la instancia, `instance getClass`, con el nombre del método que deseamos buscar `aCall methodName.` Una vez encontrado el método, lo dejaremos en la variable `method`.

Por último, evaluaremos todas las expresiones que forman los parámetros, creando una colección que dejaremos en `parameters`. Ahora ya estamos listos para realizar la invocación del método que queremos llamar enviando el mensaje `invokeInstance` al método method con la instancia `instance` y los parámetros `parameters`. Finalmente, devolvemos el resultado de esa invocación como resultado de la evaluación de la expresión `MACall`.

En el siguiente diagrama de clases, Ilustración 12.6, se pretende representar la relación entre la jerarquía `MAExpression` y `MAVisitor`, si bien de esta última sólo se muestra la clase `MAEvaluator`, que tiene una relación de dependencia con las clases que visita estereotipada como `visit`. En este diagrama, lo más destacable es que dentro de la clase evaluadora hay un atributo denominado `symbolTable`. Representa la tabla de símbolos que está utilizando el método en estos momentos. Se trata de una simplificación del concepto de contexto de ejecución y consiste en una tabla con todas las variables implicadas en el método, como son las variables de instancia de la instancia y de sus ancestros, los parámetros y las variables especiales `this` y `super`, que no dejan de ser una agregación nombrada de instancias. La construcción

del contenido de `symbolTable` se realiza en la implementación de `invokeInstance` de la clase `MAMethodUser`, como se describió en la sección anterior.

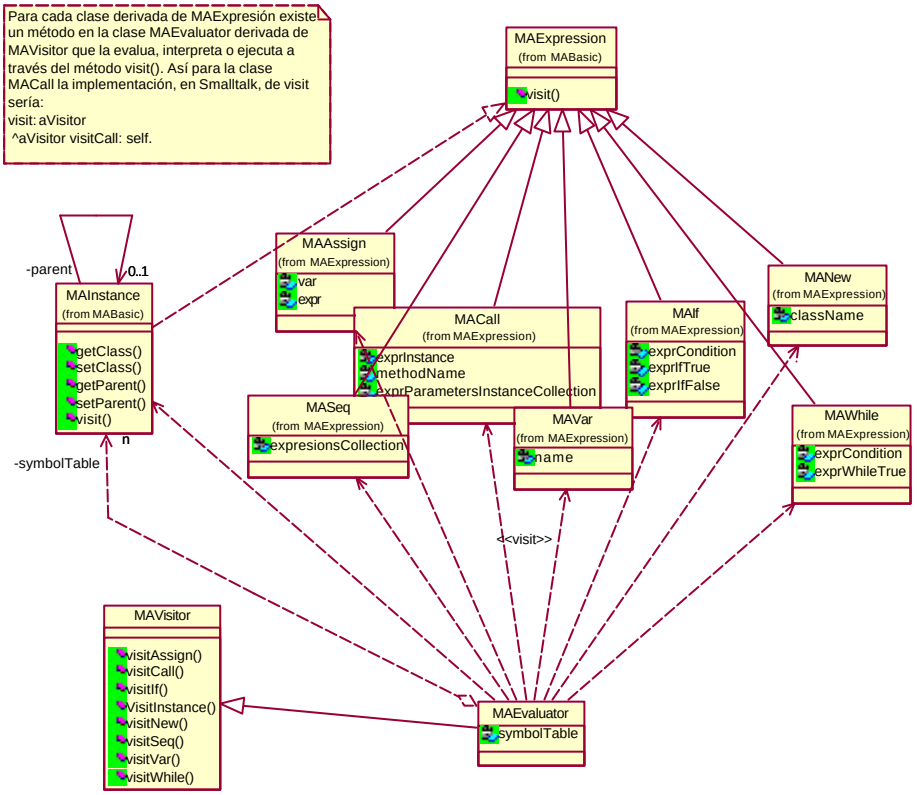


Ilustración 12.6: Relación entre las expresiones y sus evaluadores

A continuación, veremos cómo todas estas clases se relacionan e interaccionan entre sí para dar vida a nuestra máquina abstracta básica general.

12.3.6 Arquitectura

Tras una primera aproximación a la modelización estructural de los conceptos de clase, instancia, método y expresión de nuestra máquina básica, pasemos a un nuevo diagrama de clases en el que hay una visión más global de la arquitectura de la máquina y donde se muestran las relaciones más importantes que tienen entre sí.

Como vemos, una clase está formada por varios métodos y todo método pertenece a una única clase. Dentro de la jerarquía de métodos, hay clases especiales que son los métodos de usuario, que, además de tener un nombre, pertenecer a una clase y ser invocados con unos determinados parámetros, están formados por un conjunto de expresiones almacenadas en una instancia de la clase expresión secuencia `seq`. A su vez, las expresiones pueden ser de varios tipos, como ya vimos, siendo la expresión de llamada la que se va encargar de invocar los métodos tanto de usuario como primitivos a través del mensaje `invokeInstance`. Si bien para llevar esto a cabo es necesario asociarle a las expresiones un visitante que las evalúe y realice esta acción, este visitante

es un evaluador que internamente tiene definido un contexto de ejecución formado por todas las variables que intervienen en la ejecución del mismo y que viene representado por el atributo de instancia `systemTable`.

Se han definido dos estereotipos *nameRerefence* para indicar que una llamada se relaciona con el método sobre el que se realiza la invocación. De forma indirecta a través de su nombre, que servirá para buscar dentro de la clase de la instancia sobre la que se realiza la invocación y, además, por medio de *visit* que viene para estereotipar la relación de dependencia que tienen las clases visitadoras con los elementos que visitan. Si cambian las clases visitadas, las clases visitadoras también lo tendrán que hacer, dado que es una relación de dependencia, y se estereotipa con *visit* porque la dependencia hace de la abstracción visita.

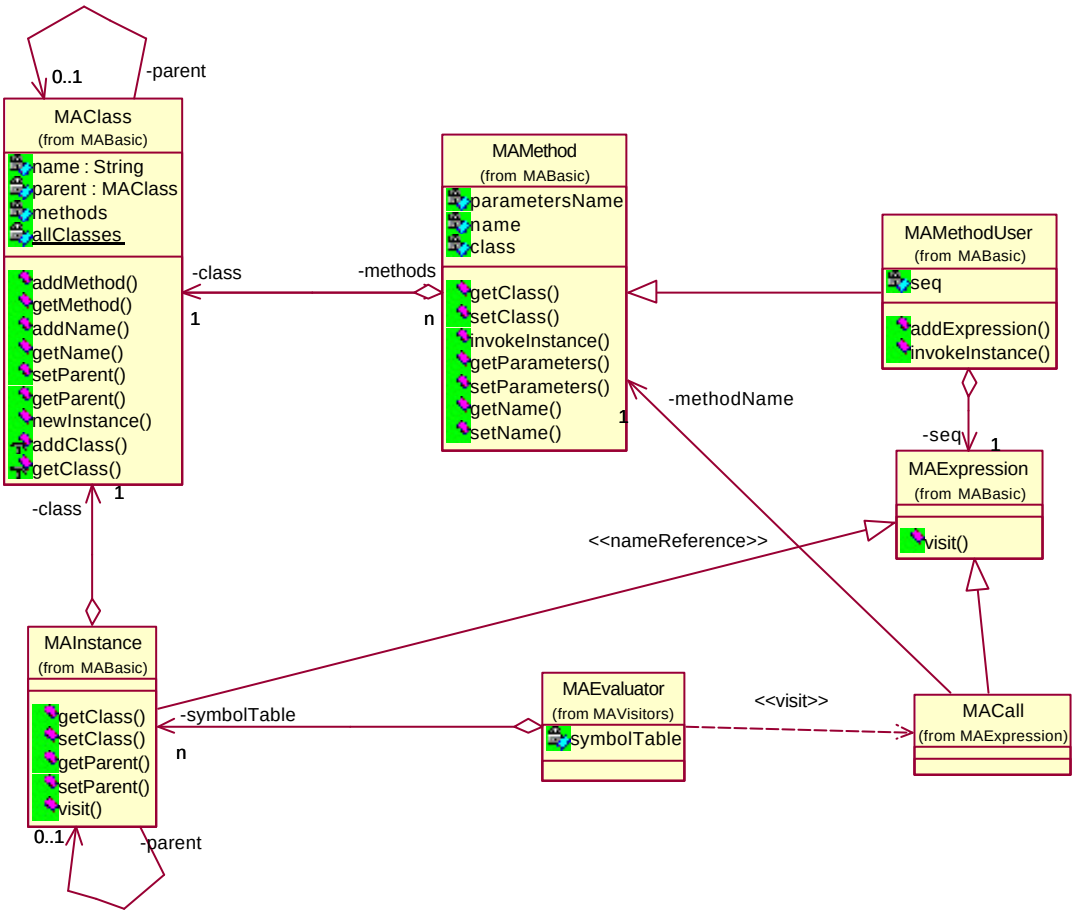


Ilustración 12.7: Arquitectura de MA

Quedan por comentar las relaciones recursivas que tienen las clases `MAInstance` y `MAClass` a través del atributo `parent`. Debido a b extenso de su explicación, dedicaremos la próxima sección completa a ello, aunque conviene indicar que son la base en las que nos sustentamos para añadir los conceptos de herencia y polimorfismo a la máquina.

12.3.7 Herencia y polimorfismo

En nuestra máquina, las clases pueden heredar de forma simple de otra clase superior sus atributos y métodos. Para lograrlo se utiliza el atributo `parent`, que es una referencia a la clase padre de la clase actual. A través de ella recorreremos toda la jerarquía de clases en busca tanto de atributos como de métodos. Así, cuando buscamos un método y este no pertenece nuestra clase, se buscará en la clase `parent` de forma recursiva hasta llegar a una clase que tenga el atributo parent nulo o `nil`, en el caso de *Smalltalk*. Lo mismo sucede con los atributos de una clase. El siguiente código muestra la implementación del método `getMethod` de la clase abstracta `MAClass`:

```
MAClass>>getMethod: paramName
      "the method paramName into this Class or its ancestors"
      | aMethod |
      aMethod := methods at: paramName.
      aMethod ~= nil
            ifTrue: [^ aMethod]
            ifFalse: [parent <> nil
                        ifTrue: [^ parent getMethod: paramName]
                        ifFalse: [^ nil]]
```

Lo primero que hace `getMethod` es buscar el nombre del método, `paramName`, dentro de la colección de métodos de la clase actual `methods`. Si lo encuentra, lo devuelve y finaliza. En caso contrario, realiza una llamada al mismo método `getMethod` de la clase `parent` buscando el mismo nombre `paramName`. Este proceso se repite hasta que o bien se encuentra el método entre los ancestros de nuestra clase o bien el atributo `parent` del último elemento de la jerarquía es nulo y se devuelve un valor nulo.

Una cosa que debemos observar es que, dado el mecanismo de implementación de métodos y de herencia, pueden redefinirse métodos dentro de una jerarquía, no así dentro de una misma clase. De forma que, al buscar, se encontrará el método más cercano de la jerarquía.

A nivel de instancias, esta relación de parentesco se debe mantener de tal forma que cuando se construye la instancia de una determinada clase, además de crear e inicializar los atributos básicos de esta clase, hay que construir e inicializar las instancias de los ancestros y repetir el proceso de forma recursiva, hasta llegar a una clase sin ancestros. Esto es lo que realiza el método `newInstance` de la clase `MAClassUser`:

```
MAClassUser>>newInstance

      | theInstance |
      theInstance := MAInstanceUser new class: self.
      fields
            do: [:aField | theInstance addField: aField].
      parent ~= nil
            ifTrue: [theInstance setParent: parent newInstance].
      ^ theInstance
```

12.3.8 Análisis Sintáctico

Como se comentó al inicio de esta sección, la máquina abstracta no ha sido diseñada para ningún lenguaje en concreto, es independiente del lenguaje de programación. No obstante resulta más cómodo programarla a través de algún lenguaje en vez de construir de forma manual los programas, como se muestra en la siguiente figura:

```
MATest>>makeMethodADDAB
      | m eCall eAssign |
      m := MAMethodUser new name: 'ADDDAB'.
      eCall := MACall new exprInstance: (MAVar new name: 'B').
      eCall addParameter: (MAVar new name: 'A').
      eCall methodName: 'ADD'.
      eAssign_MAAssign new aVar: (MAVar new name:'C');aExpr: eCall.
      m addExpresion: eAssign.
      m addExpresion: (MAVar new name:'C').
      ^ m
```

El código equivalente en Java sería:

```
Integer ADDAB(){
Integer a,b,c;
c=a.ADD(b);
Return c
}
```

Se han construido varios compiladores pequeños de diversos sabores de lenguajes populares como Java, C++, Smalltalk e, incluso, se ha diseñado un lenguaje en castellano aprovechando las características de la máquina llamado TranquiTal, en el Apéndice A se puede encontrar una descripción de sus sintaxis y semántica. Para ello, se ha utilizado un generador automático de compiladores denominado *SmaCC* [Brant03] que está basado en gramáticas LR(1) y L1RL(1).

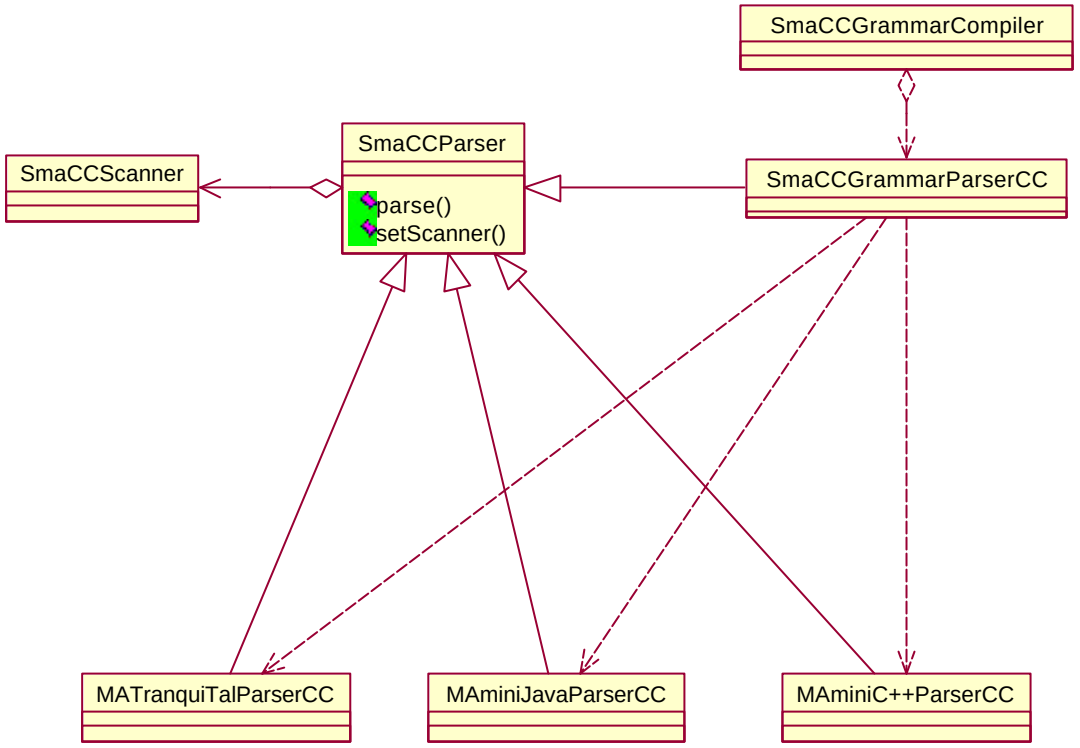


Ilustración 12.8: Arquitectura de SmaCC

En la Ilustración 12.8 se muestra la arquitectura estructural sobre la que se basa el generador de compiladores SmaCC. La clase principal de todos los compiladores de los lenguajes construidos con esta herramienta hereda de `SmaCCParser` que dispone de un método principal que genera el *código objeto* a partir de un *Sream* que contienes el *código fuente.* El propio compilador de la herramienta también hereda de dicha clase, `SmaCCGrammarParserCC`, que el usuario manipula a través de otra clase agregada y definida por la interfaz de usuario, `SmaCCGrammarCompiler`.

SmaCCParser está basado en un lenguaje de descripción sintáctica valuada que permite definir acciones semánticas para construir el análisis semántico y la generación de código [Cueva98].

Capítulo 13:
Reflectividad en máquinas abstractas

Como se vio en el Capítulo 5, la reflectividad nos hace posible, a nivel básico, no solo la consulta sino incluso la modificación de la organización interna del sistema de computación que lo soporta o metasistema. De modo que, facilita el acceso a esta estructura interna como si fuesen datos, ofrece los mecanismos necesarios para razonar acerca de su estado actual y posibilita la modificación de dicho estado para adecuarse a las condiciones actuales del problema a tratar. Se consiguen así una mayor flexibilidad, extensibilidad y modificabilidad de nuestros sistemas. Todas estas propiedades se las podemos asociar a nuestro sistema siempre que definamos el modelo arquitectónico pensando en una máquina que lo permita.

Veamos cómo ampliar de forma elemental la máquina básica con el fin de permitir la reflectividad estructural y cosificar determinadas partes internas de su implementación para, así, obtener reflectividad en su comportamiento. Sin embargo esta última técnica no es la mejor forma de obtener reflectividad del comportamiento, puesto que solo conseguimos un resultado parcial. Con el modelo Nitro obtendríamos una reflectividad total de comportamiento, pero su complejidad, la poca disponibilidad en el mundo real de máquinas de este tipo y su ineficiencia[‡] nos hacen descartarla.

El modelo de ejecución sigue intacto, salvo en que hay determinadas acciones que no pueden y no deben ser realizadas, como, por ejemplo, que una clase método se inspeccione o se modifique a sí misma, lo que podría dar resultados inesperados. Este tipo de acciones se solucionarían con un sistema de seguridad [Diaz00].

[‡] Con respecto a la eficiencia hay que hacer un apunte, indicando que las últimas investigaciones llevadas a cabo que permiten realizar compilación JIT con máquinas tipo Nitro están consiguiendo mejoras relativamente importantes en cuanto al rendimiento

A continuación, expondremos cómo realizar la cosificación de un determinado elemento interno. A este modelo ya estamos habituados, ya que en el capítulo anterior se ilustró. Ahora lo veremos con más detenimiento debido a que es el factor determinante sobre el que se sustenta la reflectividad, tanto estructural como de comportamiento. Posteriormente, veremos cómo añadir reflectividad estructural y comentaremos cómo se podría añadir, en caso de necesitarse, reflectividad del comportamiento.

13.1 Cosificación básica

Como ya se indicó en el apartado de reflectividad, todo proceso reflectivo consta de dos partes: una cosificación del metasistema sobre el que se quiere trabajar y, posteriormente, una reflexión de las acciones resultantes sobre ese metasistema.

Si bien todo proceso de construcción de un sistema consiste en cosificar parte de sus elementos de tal forma que sean fácilmente manipulables, a un nivel de abstracción superior, por ejemplo, cuando un programa de facturación representa una factura, se está exponiendo al exterior un conjunto de elementos que, todos juntos y cooperando, dan la sensación al usuario de un modelo de una factura del mundo real. Sin embargo internamente, este modelo está constituido por varios elementos del lenguaje en el que se ha programado la aplicación como pueden ser registros, vectores, tipos básicos, etc.

En una máquina abstracta gran parte de los elementos que lo componen son reificaciones de elementos externos o representaciones primitivas. Así, una instancia de la clase `Number` viene a ser la cosificación de un conjunto de bits.

Lo que se pretende con la reflectividad es cosificar las representaciones conceptuales o las que se utilizan internamente a la hora de utilizar un sistema. Así, lo que se pretende con la reflectividad de una máquina abstracta es cosificar los conceptos de clase, instancia y método para poder manipularlos desde el propio sistema base.

Este proceso es análogo al de la cosificación de cualquier otro elemento. Por ello, estudiaremos detenidamente el proceso de cosificación de la clase `Number` de *Smalltalk* y, posteriormente, procederemos a ver la realización del proceso de cosificación de los diferentes elementos de la máquina. Obtenemos, de esta forma, una máquina reflectiva, con lo que a nivel de programación de la propia máquina básica se podrán crear y manipular nuevas clases, instancias y métodos.

En la Ilustración 13.1 se muestra un diagrama de colaboración donde se puede observar como interaccionan las tres instancias de las clases que dan vida en el sistema base a un elemento de tipo `Number` a partir de la clase Number de Smalltalk. En este caso concreto, vemos el envío de mensajes que tiene lugar al tratar de comparar dos enteros a través del método primitivo `Equal` cuando es invocado mediante la llamada a `invokeInstance`. El objeto `instanceA` es el objeto sobre el que se realiza la invocación, `instanceB` es el objeto con el que se compara y `instanceR` es el resultado de la comparación que tendrá el valor uno si son iguales o cero si son diferentes.

Como comentamos en la sección anterior, las instancias de la clase `Number` de nuestra máquina abstracta vienen representadas por la clase `MAInstanceNumber` que tiene agregad el objeto que representa, en este caso, un Number de Smalltalk. El comportamiento de esta instancia viene determinado por su clase `MAClassNumber` que contiene, entre otras cosas, los mensajes que puede recibir dicha instancia a través de sus métodos. Así, la invocación del método `MAMethodNumberEqual` a través de `invokeInstance: instanceA withParameters: instanceB` toma el valor numérico de `instanceA` a través del mensaje `getValue` que, a su vez llama a `getValue` de `numbreA` agregado. Posteriormente, llama a `getValue` para instanceB que hace lo propio con `numberB`. En el quinto mensaje se llama a `add` de la clase `Number` sobre la instancia `numberA`. A continuación llama al mensaje `add` y con el mensaje número seis crea una nueva instancia `instanceR` con su correspondiente `number` agregado `numberR` y deposita el valor de la suma a través del método `setValue` en él, devolviéndolo como resultado.

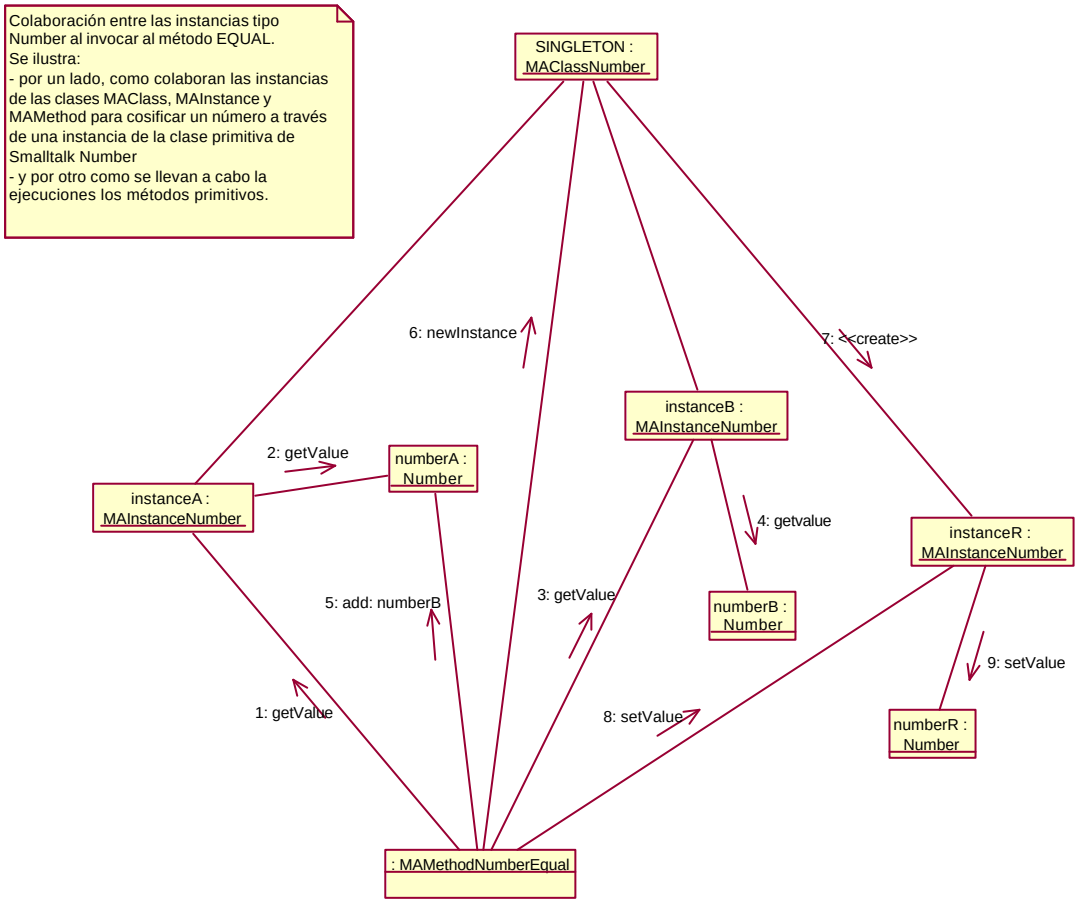


Ilustración 13.1: Colaboración para la cosificación

Una colaboración parecida a esta tiene lugar con cada una de las invocaciones de los métodos primitivos. Podemos deducir que para cada objeto a cosificar necesitamos una instancia de su clase `MAClassNumber`, una para todas las instancias, su instancia

`MAInstanceNumber`, una instancia de esta clase por cada instancia del objeto a cosificar, y, para cada método, una instancia de su clase que se agregará a la instancia de su clase `MAClassNumber`, en nuestro caso `MAMethodNumberAdd`.

Considerando una clase como una agregación de métodos, el siguiente diagrama de clases nos muestra la jerarquía de clases implicadas en la cosificación de una instancia Number de Smalltalk, para construir la clase `Number` de nuestra máquina. Esta misma jerarquía está implicada en la cosificación de cualquier elemento del metasistema, en nuestro caso Smalltalk, incluidas las propias clases `MAClass`, `MAInstance`, `MAMethod` y `MAExpression`.

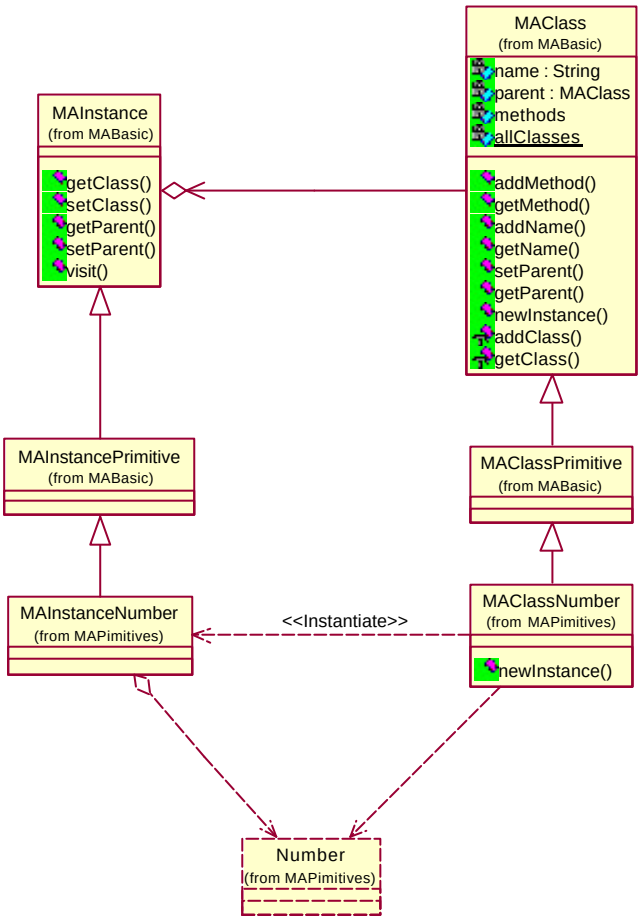


Ilustración 13.2: Cosificación de Number

13.2 Reflectividad estructural

De esta forma y siguiendo la misma filosofía de cosificación, en el diagrama de la Ilustración 13.3 se muestra el proceso de cosificación de las clases conceptuales utilizadas en nuestra máquina básica. En dicho diagrama vemos que necesitamos dos

clases para cada cosificación, pero, en nuestro caso, el objeto agregado en la clase `MAInstance` correspondiente son las propias clases definidas para dar vida a nuestra máquina. De tal forma que nos encontramos que la clase `MAInstanceInstance` agrega a su abuela la clase `MAInstance`, o que la clase `MAInstanceClass` agrega a la abuela, de la clase `MAClassClass`, es decir, a `MAClass`. Esto puede dar lugar a operaciones o estados inestables que trataremos de evitar. Así como, por ejemplo, impediremos que un método se modifique a sí mismo o que una instancia modifique la clase a la que pertenece.

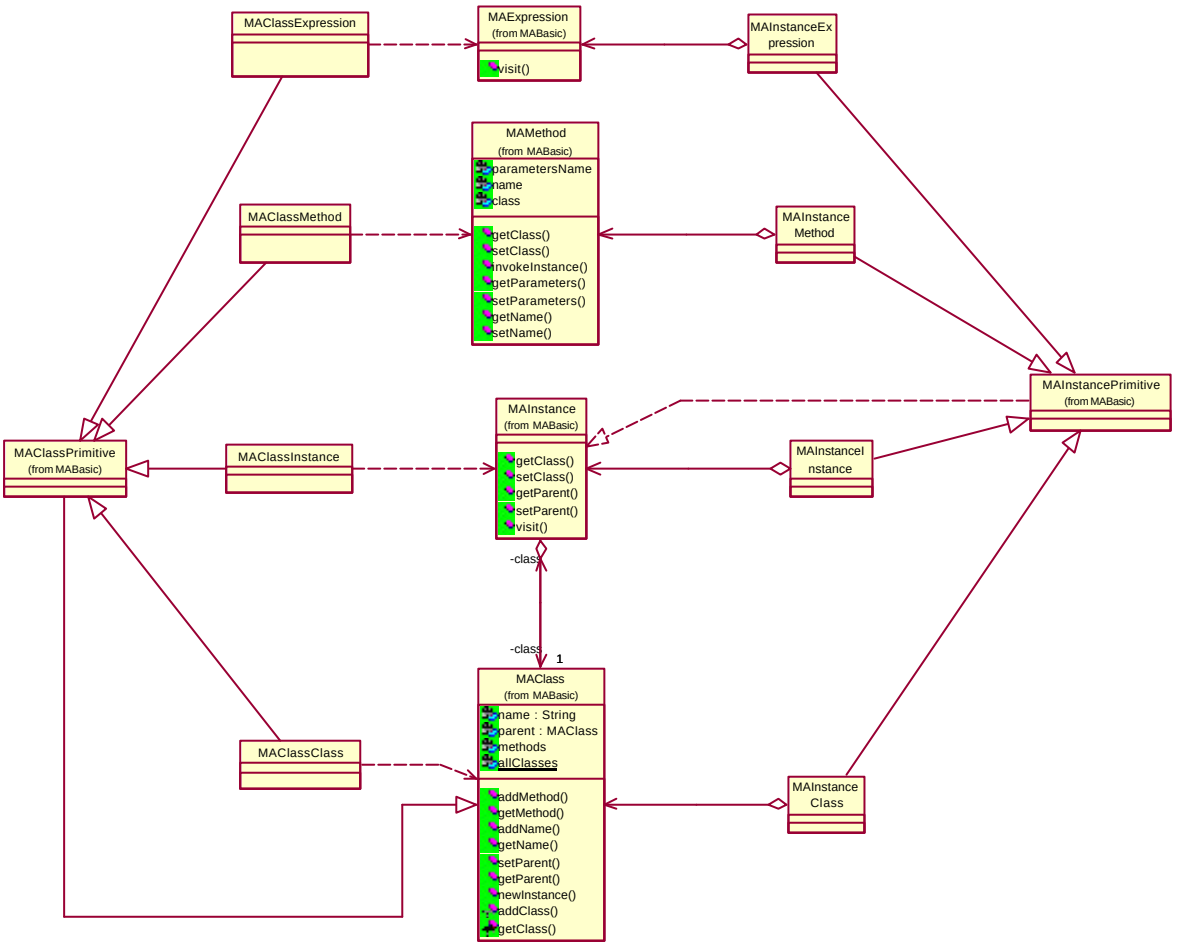


Ilustración 13.3: Reflectividad estructural

Una vez implementadas estas nuevas clases, disponemos de ellas de forma primitiva, dando vida a las clases de la máquina abstracta `Class`, `Instance`, `Method` y `Expression`. Así, lograremos crear una instancia de una clase a partir de una cadena que representa su nombre o a partir de otra instancia. Podremos acceder a cualquier método de una clase e invocarlo de forma dinámica, añadir nuevas clases al sistema en tiempo de ejecución, sumar nuevos métodos o atributos a una clase o añadir expresiones a un método.

13.3 Reflectividad computacional

Todo lo mencionado anteriormente hace referencia a la llamada reflectividad estructural, en la que lo que se modifica es la estructura del propio programa. Es decir, el programa puede cambiar su propia estructura, ampliándola, reduciéndola o modificándola. Si bien hay otro tipo de reflectividad, denominada reflectividad del comportamiento, computacional o semántica. En este caso se pretende modificar el comportamiento de la propia máquina, para así, por ejemplo, poder cambiar el comportamiento de las instrucciones, una invocación, el acceso a una variable, la herencia, etc. En resumen, modificar la propia implementación de la máquina. Para ello, hay soluciones, parciales tales como, cosificar y definir un MOP para acceder a determinadas partes de la máquina, definir MOP dinámicos como propone Ortín en su máquina Nitro o cosificar el propio sistema en el que se implementa la máquina, lo que se puede hacer si dicho, sistema permite reflectividad estructural. En el caso de nuestro prototipo, esto sería posible, aunque no lo consideramos necesario. Además de ser complicado y peligroso. En caso de necesitar reflectividad computacional, nos definiremos un MOP utilizando la técnica de cosificación parcial mencionada anteriormente.

Capítulo 14:
Diseño arquitectónico

Hasta aquí hemos esbozado los requisitos generales que deseamos que cumplan los sistemas que se construyan con nuestro diseño arquitectónico (véase el Capítulo 10), las cuestiones de diseño que restringen el dominio del problema sobre el que se aplicara el diseño (véase el Capítulo 11) y, por último, el diseño del modelo general de una máquina básica de alto nivel de abstracción, sobre la que se ha aplicado reflectividad estructural y que nos servirá de pilar fundamental para poder afrontar el diseño (véanse el Capítulo 12 y el Capítulo 13). Estamos ahora en condición de comenzar con el diseño arquitectónico de nuestro sistema, que permitirá resolver el problema de la interoperabilidad heterogénea de middlewares utilizando máquinas abstractas reflectivas orientadas a objetos.

Dicho diseño cumple con las pautas básicas indicadas en el Proceso de Desarrollo Unificado [PUD00] y viene expresado en UML, el Lenguaje de Modelado Unificado [UML99], que se ha convertido en el estándar para el modelado de sistemas orientados a objetos. Se han seguido un proceso iterativo e incremental, realizando varias iteraciones sobre los *flujos fundamentales*: Requisitos, Análisis, Diseño, Implementación y Pruebas. Estos dos últimos flujos no se han llevado a cabo dado que se trata de un diseño arquitectónico donde no hay ninguna implementación final, aunque en el desarrollo del prototipo basado en este diseño, que se expone a continuación, sí se han abordado ambas fases. Por otro lado, solo hemos realizado iteraciones sobre las *fases* de inicio y elaboración ya que se pretende obtener un diseño arquitectónico que, precisamente, es el hito fundamental de la fase de elaboración. En la implementación del prototipo se han realizado dos iteraciones sobre la fase de construcción, como veremos en el Capítulo 17. Nuestro diseño arquitectónico se centra en el artefacto de UML denominado colaboración. Dicho diseño viene expresado, principalmente, en forma de colaboración genérica, dentro de la vista de diseño. Contemplando siempre, como toda colaboración, una parte estructural y otra de comportamiento.

Este diseño consta principalmente de dos vistas, la vista de casos de uso y la vista de diseño. La vista de proceso hemos decidido incluirla dentro de la vista de diseño. Las vistas de despliegue e implementación se dejan para las instanciaciones del diseño.

El modelo de diseño arquitectónico pretende definir una colaboración genérica que podrá ser **instanciada** para construir modelos concretos arquitectónicos de sistemas reales. Básicamente, se deberá parametrizar/instanciar sobre la elección de máquina abstracta concreta y sobre las plataformas middleware que se desea que interoperen entre sí. En el Capítulo 16, describiremos varias de estas instanciaciones de la colaboración genérica de nuestro modelo, aplicado a varias plataformas middleware reales, en concreto DCOM, CORBA y Servicios Web.

La colaboración genérica resultante es tan representativa y amplia que hemos decidido dedicarle el Capítulo 15 íntegramente. Esta colaboración se subdivide en dos colaboraciones genéricas que se describirán detenidamente en cada subapartado de dicho capítulo.

El objetivo de este capítulo es realizar una primera aproximación al modelo arquitectónico y en el siguiente se detallará dicho modelo. Inicialmente expondremos a alto nivel el objetivo que se pretende conseguir. Seguidamente, mostraremos la vista de casos de uso, donde delimitaremos el sistema a partir de los actores que intervienen y expondremos los requisitos fundamentales a través de casos de uso. El resto del capítulo se centra el la vista de diseño, ya sea a nivel de subsistemas y estructural como de comportamiento.

14.1 Visión de alto nivel

Como se ha venido indicando en los capítulos anteriores, el objetivo de esa tesis es definir un modelo arquitectónico que nos permita construir, de forma sencilla y estructurada, un bus para plataformas middleware o middlebus, al que se puedan añadir nuevas plataformas y todas interaccionen entre sí automáticamente, tanto para invocar servicios de otras plataformas como para recibir invocaciones de las mismas. Como indicamos en el Capítulo 11, restringimos las plataformas middleware a las que pertenecen al tipo LII o RMI.

En la siguiente ilustración se esboza el resultado de los sistemas que podremos construir a partir de nuestro modelo arquitectónico. Observamos que hay varios middleware conectados al bus que hemos denominado LIIBUS, a saber DCOM, RMI Java, CORBA, RPC y SOAP. Los objetos de cada una de estas plataformas van a ver al resto de elementos de las otras, como propios, pudiéndolos utilizar sin necesidad de que se realice ningún tipo de adaptación o puente, como ocurre con otras técnicas de resolución de heterogeneidad (ver Capítulo 8). Una vez instalada una plataforma, esta ya esta lista para interaccionar con todas las otras plataformas que están conectadas, incluso pueden interaccionar a la vez con varias. Quizás, por esta característica, debería denominarse middleswitch en vez de middlebus. El sistema LIIBUS está basado en una máquina abstracta reflectiva concreta y se aprovecha de sus características de invocación, introspección y localización.

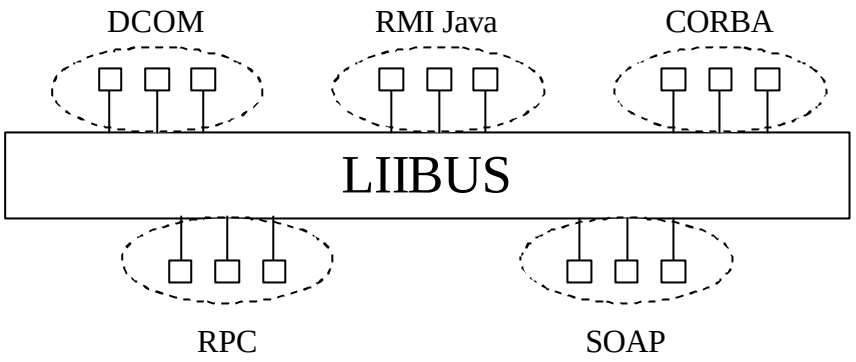


Ilustración 14.1: Vista esquemática de LIIBUS

Para conseguir esta comunicación de unos sistemas con otros de forma transparente, podemos utilizar el concepto de *wrapper*. Pasamos a explicar en términos generales y abstractos de alto nivel, sin ver detalles, cómo se consigue esta comunicación.

Disponemos de un sistema intermediario, nuestro sistema, que está basado en una máquina virtual orientada a objetos reflectiva. Para cada uno de los objetos de los sistemas externos se crea un envoltorio, *wrapper* o representante, *proxy*, dentro de la misma, de forma automática. A su vez, se exponen cada uno de los objetos internos de la máquina a los sistemas externos a través de despachadores de invocaciones. De esta forma, conseguimos que, por un lado, los objetos internos puedan ver a todos los objetos de nuestra máquina como propios y, por otro, que los objetos de nuestra máquina vean los objetos de los sistemas externos como internos de la máquina. Es decir, logramos una comunicación bidireccional entre nuestro sistema, formado por los elementos de la máquina abstracta, y los elementos del sistema externo. Si disponemos de dos sistemas externos con los que comunicarnos, estos podrán comunicarse entre sí a través de los objetos de nuestra máquina.

La Ilustración 14.2 nos muestra lo comentado anteriormente de forma gráfica. Los círculos de los extremos representan sistemas externos, mientras que el círculo central es nuestra máquina abstracta. Vemos que el objeto Ob1 de la izquierda tiene un wrapper dentro de la máquina denominado Obj1 Wrapper. Igual ocurre con el sistema de la derecha. A través de los envoltorios ambos sistemas podrán comunicarse con los internos objetos que haya dentro de la máquina; pero resulta que los propios envoltorios son, a su vez, objetos internos igual que el resto. Por lo que el Obj puede realizar invocaciones al objeto envoltorio del segundo sistema de forma transparente, y podemos comunicar de manera virtual el sistema uno con el dos.

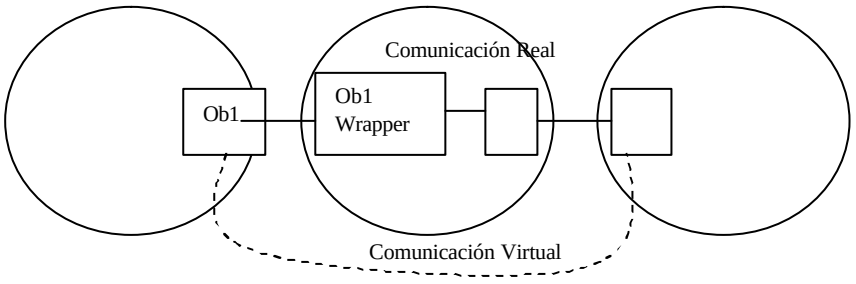


Ilustración 14.2: Comunicación real y virtual

Una vez esbozado el funcionamiento general de los sistemas que deseamos construir, el diseño de nuestro modelo arquitectónico pretende abstraer los conceptos identificados en este escenario, proceder a generalizarlos y aplicarlos. Esto lo conseguiremos definiendo la vista de casos de uso que está descrita en los dos apartados siguientes. Después de identificar los elementos, tanto funcionales como externos, representativos del sistema los utilizaremos para guiar la construcción del modelo arquitectónico.

14.2 Actores

Vamos a considerar dos tipos de actores. En primer lugar, los seres humanos que interoperan, gestionan y administran nuestro sistema y que lo representaremos con `administrator`. Por otra parte los sistemas externos de invocación independiente de la ubicación que interoperan con nuestro sistema, cuyo representante será `External RMI` y del que todos derivan. Hemos considerado los más importantes: `RMI Java`, `WebService`, `CORBA`, `DCOM`, `Oviedo3`, así como un actor especial que representa cualquier otro sistema de invocación independiente de la ubicación que ya exista o pueda aparecer en un futuro: `Future RMI.`

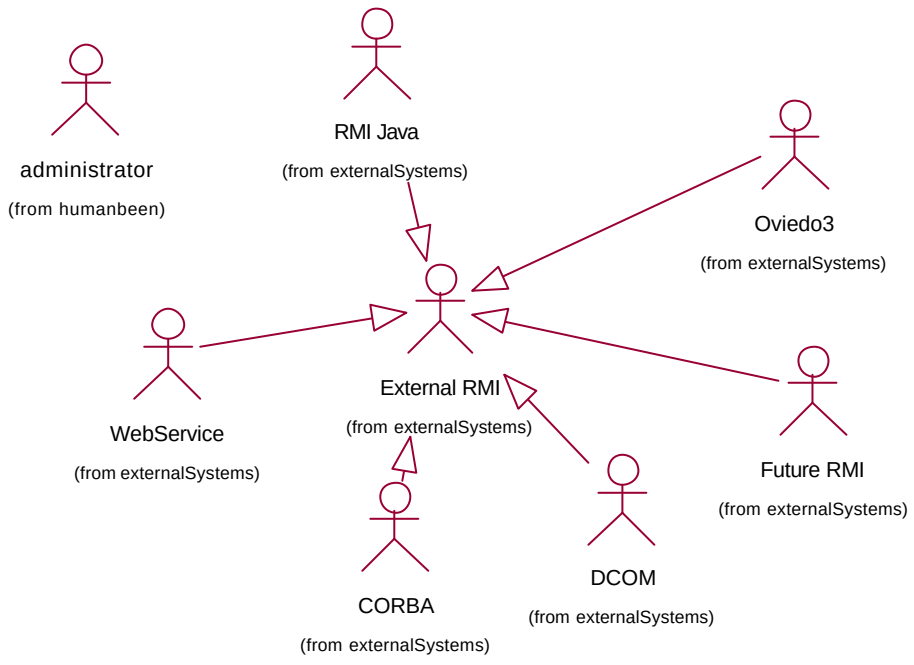


Ilustración 14.3: Actores, Vista Arquitectónica

`External RMI` representa los sistemas que cumplen con las condiciones de diseño comentadas en el Capítulo 11, es decir, deben ofrecer servicios de invocación dinámica de métodos, introspección y localización. Todos los actores que heredan de él tendrán que cumplir las mismas con estas condiciones.

14.3 Casos de uso

En la siguiente figura se ilustra el caso de uso principal, así como su efecto indirecto sobre los sistemas distribuidos externos. Este caso de uso se denomina `interoperate` y representa la funcionalidad aportada por nuestro sistema para interactuar con los sistemas externos, en nuestro caso, los actores derivados de `External RMI`. Esta comunicación bidireccional da como resultado que los propios actores puedan establecer entre sí una comunicación bidireccional virtual, como si nuestro sistema intermediario no existiese.

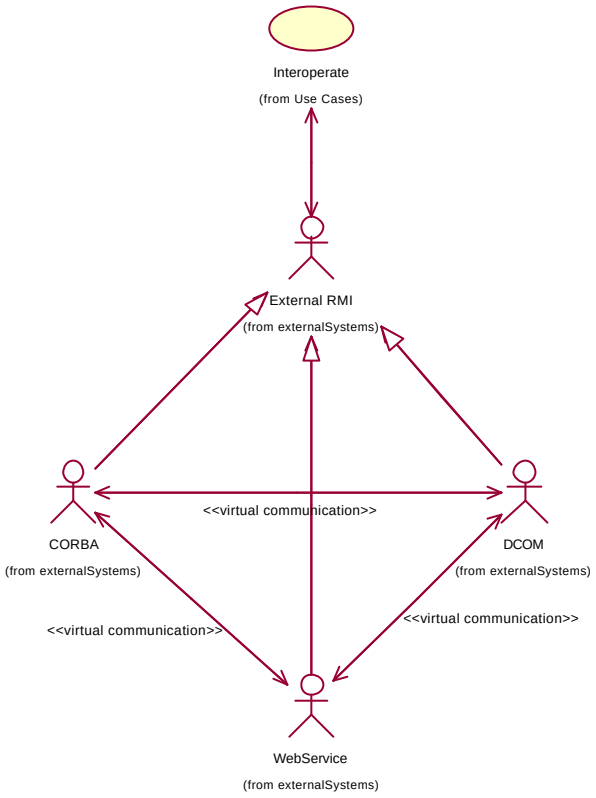


Ilustración 14.4: Caso de Uso, Vista Arquitectónica

La Ilustración 14.5 muestra los tres casos de uso más importantes:

- `admin`, que consiste en la interacción del actor administrador con el sistema para gestionarlo, administrarlo y utilizarlo.
- `interoperate`, que, como hemos comentado más arriba, representa la interoperabilidad bidireccional entre nuestro sistema y los sistemas externos representados por los actores derivados de `External RMI`. Este caso incluye dos subcasos de uso, uno para cada una de las direcciones de la interoperabilidad. De tal forma que, realmente, los sistemas externos interactúan con cada uno de ellos en función de la dirección de los mensajes.

- `outdoor interoperate`, que representa la comunicación desde nuestro sistema hacia los sistemas externos. Con su realización conseguiremos ver todos los elementos de los sistemas externos como propios y podremos utilizarlos e interactuar con ellos como si estuviesen dentro de nuestro propio sistema.
- `indoor interoperate`, que representa la comunicación hacia fuera de nuestro sistema, recibiendo invocaciones de los sistemas externos y simulando todos los objetos internos como sistemas pertenecientes a los sistemas externos.

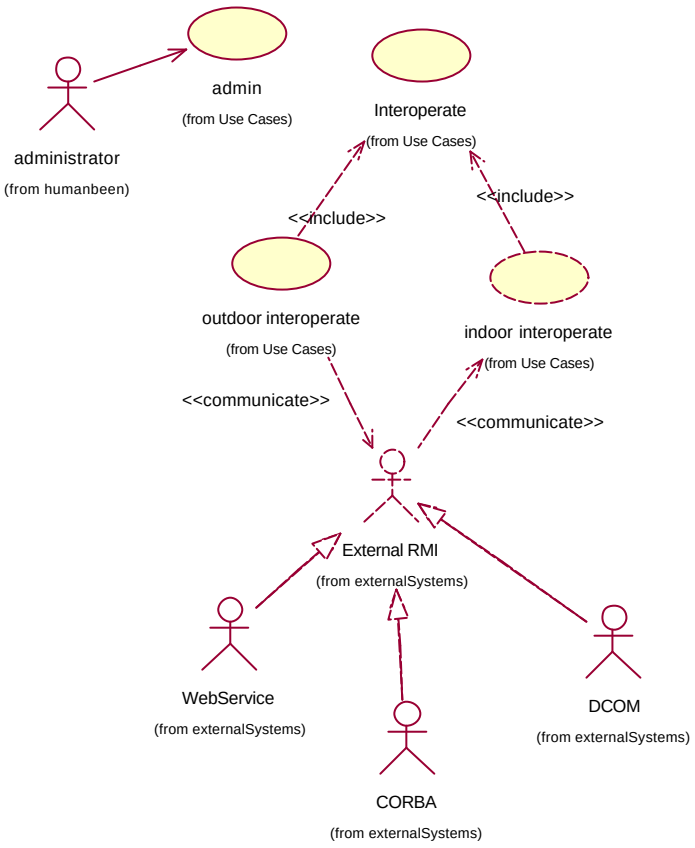


Ilustración 14.5: Casos de Uso detallados, Vista Arquitectónica

El caso de uso `admin` no lo vamos a contemplar a lo largo de este trabajo. Consideramos que las actividades de administración y gestión van a ser diseño propio de cada arquitectura concreta y se deja a libre elección y construcción del desarrollador del sistema. Nos centramos en resolver problema de la interoperabilidad a nivel de modelo de diseño general e independiente de plataformas concretas.

14.4 Descripción de subsistemas

Realmente nuestro sistema va a ser una ampliación, ya sea esta interna o externa, a través de clases de usuario, de la propia máquina abstracta. Es por ello que nuestro sistema va a estar formado por el sistema constituido por la propia máquina junto con los diversos subsistemas, que no dejan de ser ampliaciones de la máquina. A nivel arquitectónico, vamos a considerar dos subsistemas: el reflectivo `MAReflective` y el propio del bus `LIIBUS.` Este último a su vez contiene dos subsistemas, uno para cada una de las direcciones de la comunicación. La primera, que denominaremos `<-MA->,` que representa las llamadas de los objetos internos de la máquina hacia objetos externos y la otra, que denominaremos `->MA<-,` que representa las llamadas de los objetos internos hacia los objetos de la propia máquina. Ambas se corresponden con las realizaciones de los casos de uso `outdoor interopreate` e `indoor interoperate` respectivamente.

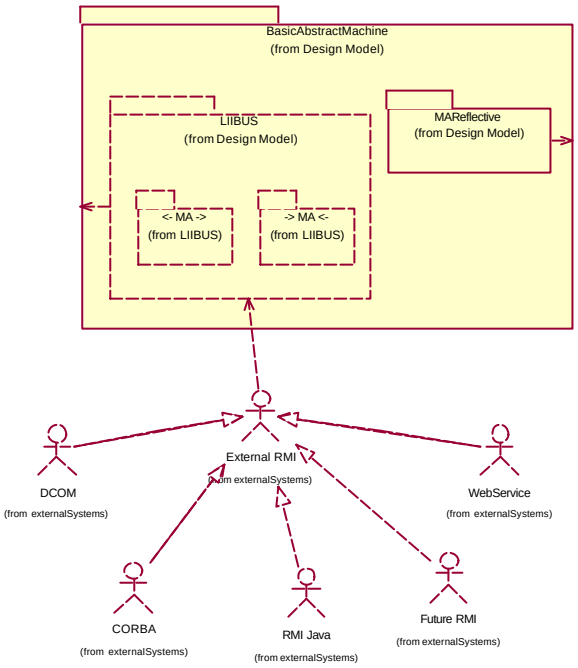


Ilustración 14.6: Modelo de Subsistemas

La Ilustración 14.6 nos muestra los subsistemas, la relación que existe entre ellos y cómo interaccionan con los sistemas externos representado por el actor `External RMI.`

14.5 Jerarquía de clases

La representación estructural de nuestra arquitectura viene reflejada en la Ilustración 14.7, donde se muestra un diagrama de clases con las clases más representativas del sistema que deseamos conseguir. Muchas de estas clases ya las conocemos y pertenecen de forma primitiva a la nuestra máquina básica. Nos van a

servir para dar vida o cosificar las clases que nos permitirán construir los wrappers o envoltorios de forma dinámica.

Las dos clases más importantes, desde el punto de vista arquitectónico, van a ser `MAExternalSystem` y `MAExternalDispatcher`.

La primera nos va a servir para exponer los sistemas externos dentro de la máquina y así poder interactuar con ellos. Es la clase principal del subsistema <-MA->. Para añadir un nuevo sistema externo a nuestro middlebus, básicamente, necesitaremos refinar esta clase e implementar sus métodos de forma que interaccionen con el sistema externo. Si bien para representar los objetos de estos sistemas externos son necesarias las clases `MAInstanceExternal` y `MAClassExternal`, que vienen a envolver las instancias y las clases de los sistemas externos respectivamente.

La segunda nos permite poner a disposición de los sistemas externos todos los objetos internos de la máquina. Para ello debemos interceptar las llamadas entrantes, descomponerlas, ajustar los tipos, crear los objetos necesarios y realizar las invocaciones internas oportunas.

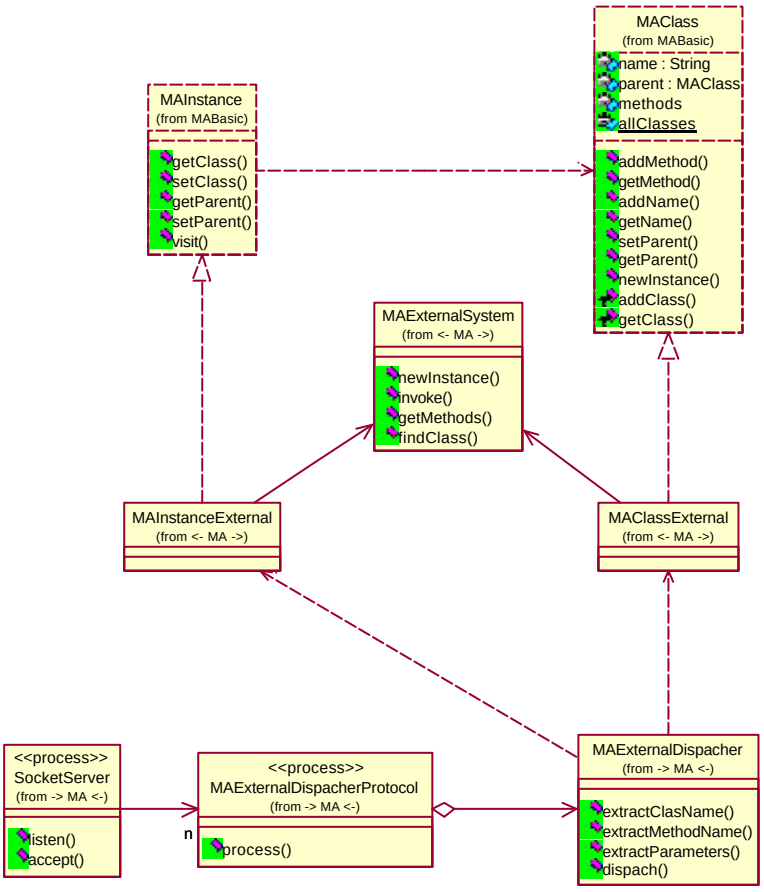


Ilustración 14.7: Modelo Arquitectónico Estructural

Cada llamada recibida externa va a ser atendida de forma concurrente. Es por ello que se define la clase activa `MAExternalDispatcherProtocol`, que viene creada

por la clase activa `SocketServer`, cuya función es actuar de objeto *listener* en los puestos especificados en cada protocolo para la comunicación.

En el siguiente apartado vamos a ver cómo interaccionan todas estas clases para llevar a cabo la comunicación virtual entre sistemas externos.

14.6 Colaboración genérica

El diagrama de clases anterior muestra una vista estructural de nuestra arquitectura. Ahora vamos a ver la vista de comportamiento de la misma. En el siguiente diagrama de secuencia se exponen las invocaciones que tienen lugar cuando un sistema externo desea interaccionar con un elemento de nuestro sistema interno que resulta ser, a su vez, un envoltorio de otro sistema externo, con lo que se produce la antes citada comunicación virtual.

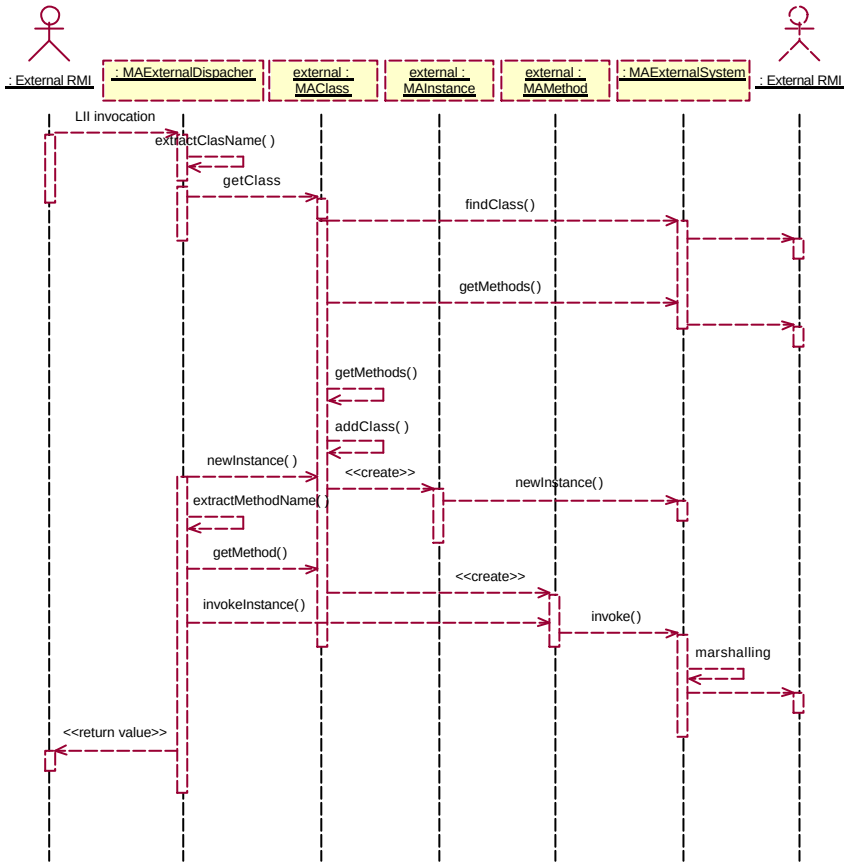


Ilustración 14.8: Modelo Arquitectónico de Comportamiento

El sistema externo inicial realiza invocaciones que son interceptadas por `MAExternalDispatcher`. Por medio de algún protocolo de interacción a nivel de paquetes de redes. Este extrae de los paquetes recibidos la información necesaria para identificar la clase hacia la que va destinada la invocación. Una vez identificada, trataremos de localizarla dentro de la máquina llamando a `getClass`, que, a su vez,

llamará a `findClass` de `MAExternalSystem`. Una vez localizada se buscarán todos sus métodos para, a partir de ellos, construir una nueva clase que la envuelva. Cuando está construida, la añadiremos al sistema con `addClass` y ya estará disponible.

Ahora necesitamos crear una instancia de la clase para realizar la invocación sobre esa instancia[§]. Con este fin, llamaremos a `newInstance` de una instancia de `MAExternalClass`, que representa la clase anteriormente creada. Esta, a su vez, redirige la llamada a `newInstance` de `MAExternalSystem` para crear una instancia en el sistema externo, que será realmente la instancia que recibirá la invocación.

Tras los pasos anteriores, estamos preparados para realizar la invocación al método indicado en el mensaje. Para ello deberemos identificar el nombre del método con `getMethod` y posteriormente, realizar la invocación con `invokeInstance` que pasará la llamada a `invoke` de `MAExternalSystem`, que realizará las conversiones de tipos oportunas y procederá a realizar la invocación sobre el sistema externo. El valor resultante de esa invocación se devolverá como un paquete de red, cumpliendo con las normas específicas de la plataforma de middleware determinada.

14.7 Escalabilidad

La propiedad de escalabilidad la tratamos aparte, ya que viene determinada, además de por el propio modelo de diseño, por la capacidad que tenga la máquina o plataforma sobre la que se construye para escalar. Así, en el caso de Oviedo3 y gracias a su sistema de distribución implícito, sería muy sencillo construir un sistema con varios procesadores aumentando de este modo el rendimiento. De esta forma, el propio sistema puede instalarse en diversa máquinas e interoperar unos con otros, consiguiendo así una escalabilidad completa. Vamos a necesitar subsistemas de comunicación distribuida transparente que nos permitan tener los elementos de nuestra máquina en diversos procesos, incluso, en diversa máquinas sin que deje de funcionar la máquina como un único sistema.

[§] Como se indicó en el Capítulo 11, no consideramos el uso de referencia por lo que todas las invocaciones crearán una instancia sobre la que se realiza la invocación y no se consideran los casos en las que las instancias ya estén creadas y nos indiquen su ubicación a través de una referencia.

Capítulo 15:
Colaboración genérica

La comunicación bidireccional entre nuestra máquina y los sistemas externos se puede diferenciar en función de la dirección de la comunicación.

Tendremos un primer subsistema que se encargará de la comunicación de nuestra máquina con el exterior, es decir, de enviar mensajes fuera del sistema y de ver todos los elementos externos al sistema como propios. Como ya hemos indicado en secciones anteriores, lo denominaremos <- MA -> para reflejar que los mensajes salen de nuestro sistema.

Por otra parte, un segundo subsistema que se encargará de la comunicación de los sistemas exteriores con el nuestro, de tal forma que los sistemas exteriores vean los elementos de nuestro sistema como propios. Utilizaremos la notación **-> MA <-** para indicar que los mensajes entran en nuestro sistema.

Si unimos los dos subsistemas, surge un sistema bidireccional de comunicación, que denotaremos como **<-> MA <->,** para indicar que nuestro sistema envía mensajes al exterior considerando los sistemas externos conectados a él como propios, y por otro lado, recibe mensajes de tal forma que los sistemas externos ven los elementos de nuestro sistema como propios. Todo esto tiene como resultado transitivo la comunicación bidireccional virtual de los sistemas externos interconectados al nuestro. Es decir, si tenemos dos sistemas externos E1 y E2 que se comunican con nuestro MA, podremos afirmar que nuestro sistema permite la comunicación bidireccional con ambos y lo denotaremos como E1 <-> MA <-> E2. No obstante, y como resultado de la propiedad transitiva de la comunicación bidireccional, tendremos que E1 <-> E2, es decir, que los elementos del sistema E1 ven a los elementos de E2 como propios y viceversa. Si bien, en realidad, siempre existirá MA como sistema intermediario o BUS.

Considerando la disección anterior, pasamos a describirlo de forma detallada.

15.1 <- MA ->

15.1.1 Introducción

En este apartado describiremos pormenorizadamente cómo conseguir que los elementos internos de nuestra máquina vean los objetos externos como objetos de la propia máquina. Con este fin a profundizar en el concepto de cosificación. Posteriormente, veremos que necesitaremos un generador de clases primitivas y pasaremos a generalizar este generador que nos permite adaptarnos a cualquier plataforma de comunicación distribuida. Para ello haremos uso del patrón de diseño *puente* o *bridge*. A continuación veremos cómo federar el proceso de localización de clases y haremos uso de patrón de diseño *cadena de responsabilidades.* Por último, veremos varias técnicas que podemos utilizar para obtener la propiedad de extensibilidad.

15.1.2 Cosificación de elementos LII externos

La idea, aquí, es realizar una cosificación de los elementos de los sistemas RMI externos. En principio, para un elemento concreto resulta relativamente fácil. Se trataría, prácticamente de construir las mismas clases que hemos utilizado para cosificar los elementos primitivos de la máquina o los elementos reflexivos. De esta forma, sabiendo el nombre de la clase y los métodos que contiene junto con sus parámetros, se podrían incluir manualmente todos los objetos con los que deseamos interoperar. Con ello conseguiremos que la máquina abstracta los considere como objetos propios, sin ningún tipo de diferencia con los primitivos o los de usuario. A continuación mostramos un diagrama de clases, análogo al que vimos anteriormente, que mostraba la cosificación de la clase `Number` de *Smalltalk* dando lugar a la clase `number` de nuestra máquina.

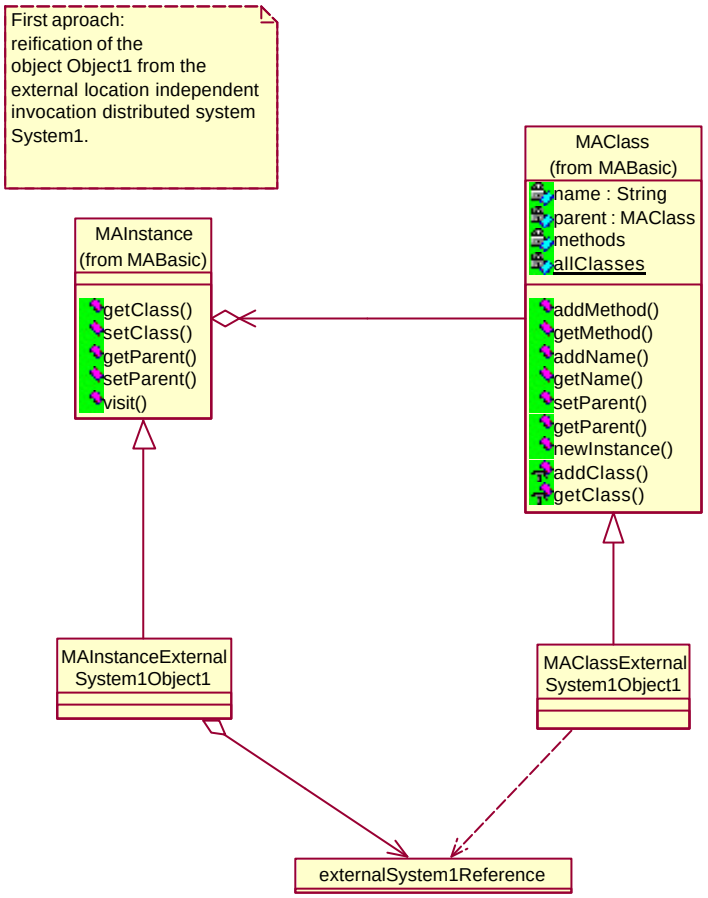


Ilustración 15.1: Cosificación básica

En este caso, consideramos un objeto `Object1` que está dentro del sistema distribuido con invocación independiente de la ubicación de `System1` y que pertenece a la clase representada por `MAClassExternal1Object1`. A partir de este momento, se pueden crear tantas instancias de este tipo de objetos como deseemos.

Ahora bien, si queremos construir instancias de otra clase dentro de `System1` deberemos construir otro cosificador de objetos de esa clase, por ejemplo, objetos de la clase a la que pertenecen los objetos de tipo `Object2`. Para otro sistema externo `System2` se podría hacer lo propio. Es más, se podría hacer derivar las clases `MAInstanceExternalsystem1Object1` y `MAClassExternalsystem1Object1` de `MAInstancePrimitive` y `MAClassPrimitive`, ya que prácticamente son clases primitivas a todos los efectos.

15.1.3 Generador de clases primitivas

Aunque la técnica comentada anteriormente da como resultado una integración totalmente transparente y homogénea de objetos y clases de sistemas distribuidos externos, se puede avanzar un paso más sacando partido de los servicios ofrecidos por la mayoría de los servicios de invocación independiente de la ubicación. Algunas de estas

prestaciones son el repositorio de clases y la introspección e invocación dinámica. Con la primera accederemos a instancias de todas las clases pertenecientes al sistema de distribución, ya que podremos ir cosificándolo conforme los vayamos necesitando. Con las segundas conseguiremos crear todas estas clases de forma automática. Considerando lo anterior, podemos construir para cada sistema un generador de cosificador de objetos y de clases que llamaremos *fábrica de clases internas*. De esta forma, cuando queramos acceder a un objeto perteneciente a alguna clase dentro de `System1`, con sólo declarar una variable de ese tipo o con invocar al sistema reflectivo para que cree una instancia, el sistema de fabricación de clases se encargará de construirlo a partir de la información obtenida de la introspección.

En este caso ocurre como con las clases de usuario, es decir, a partir de la clase `MAClassUser` se pueden controlar muchas clases diferentes de usuario. En nuestro caso, las clases pertenecientes a nuestro sistema de distrubución `System1`. Las nuevas clases de usuario las va generando el compilador conforme va analizando los códigos fuente que responden a una determinada sintaxis. Ahora las clases del sistema externo `System1` se irán formando a partir de la información obtenida de la introspección de cada clase externa.

En la siguiente figura podemos observar cómo ha aparecido la clase `MAClassExternalSystem1` que tiene una relación de dependencia, estereotipada con `<<instanciate>>` , con `MAClassExternalSystem1Object1` que, a su vez, está estereotipada con `<<base system>>`, indicando que es una clase a nivel del sistema base, es decir, a nivel de nuestra máquina, aunque a nivel del diagrama es una instancia de `MAClassExternalSystem1`. Existe una clara analogía entre la clase `MAClassUser` y `MAClassExternalSystem1`. Las instancias de `MAClassUser` representan las clases de usuario y cada una de ellas representa una clase en el sistema base que viene determinada por su campo `name`, Igual ocurre con las instancias de `MAClassExternalSystem1`, para cada clase externa dentro del sistema distribuido `System1` habrá una instancia de `MAClassExternalSystem1`, que en el diagrama de, clases hemos denominado `MAClassExternalSystem1Object1`, que sería equivalente a una instancia de `MAClassExternalSystem1` con el atributo `name` indicando su clase. Para gestionar las instancias del sistema distribuido externo `System1` se utiliza la clase `MAInstanceExternalSystem1` que tiene agregadas una referencia a la instancia real para poder manipularla a través de ella, a una asociación con la clase a la que pertenece, es decir, a la instancia de la clase `MAClassExternalSystem1` que, en el diagrama mostrado, correpondería con `MAClassExeternalSystem1Object1`.

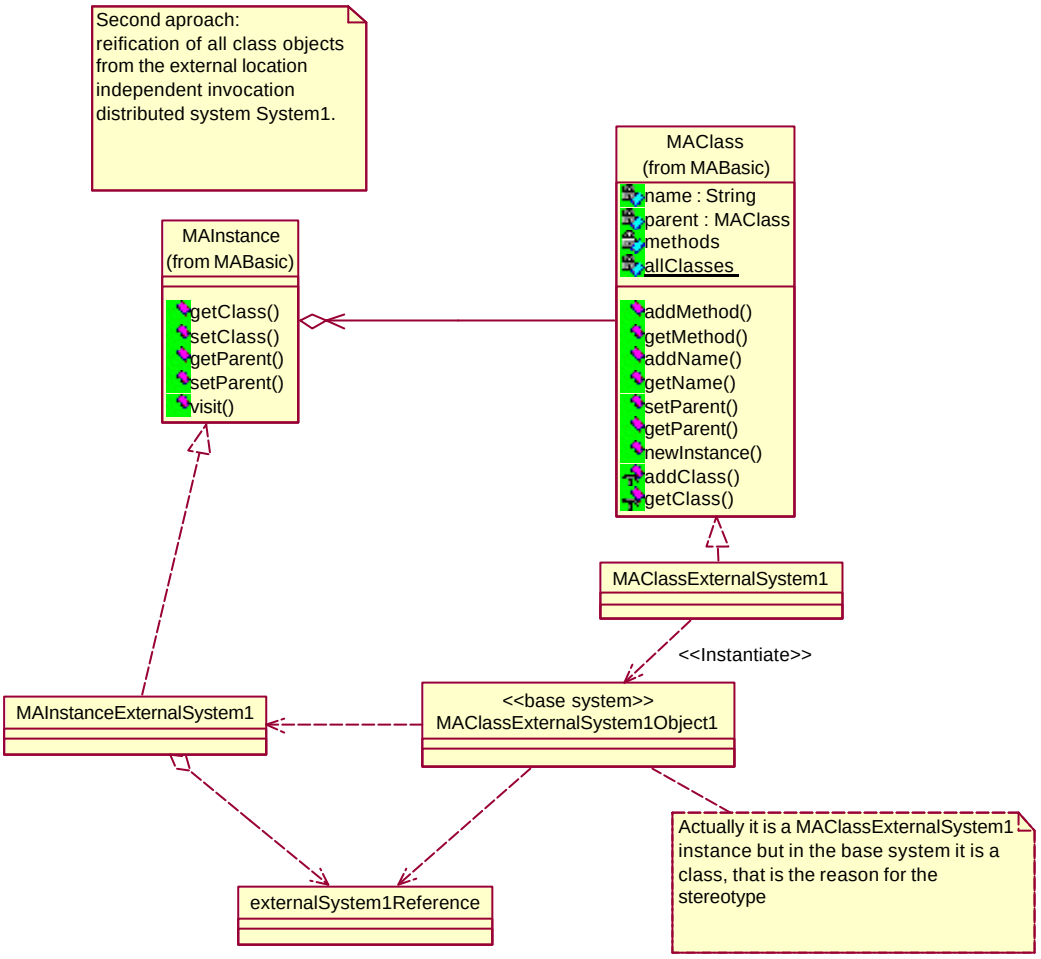


Ilustración 15.2: Generador de Clases

15.1.4 Generalización del generador

El generador de clases primitivas puede ser generalizado y abstraido a través del *patrón puente (Bridge)*. De esta forma definiremos las operaciones generales que deseamos abstraer pertenecientes a todo sistema LII, utilizando de una clase base común que denominaremos `MAExternalSystem` de la cual heredarán todos los sistemas distribuidos externos LII. Esta clase base se corresponde con la clase *implementador* en el patrón puente. Por otro lado, las abstracciones que utilizará esta clase implementador, la llamada clase *abstracción* en la plantilla puente, corresponde, con varias clases de nuestro sistema. Dichas clases son `MAClassExternal`, `MAInstanceExternal` y `MAExternalSystem`. Cada instancia de las clases anteriores asociará una instancia que hereda de implementador, en nuestro caso `MAExetrnalSystem`, diferente, consiguiendo, así, un puente hacia cada una de las plataformas externas. Veamos el siguiente diagrama de clases, donde se muestra nuestra instanciación concreta del patrón puente.

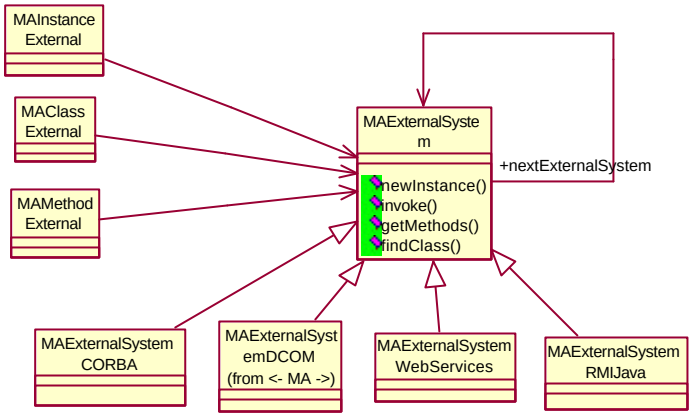


Ilustración 15.3: Generalización del generador

A partir de ahí, podemos definir una colaboración genérica correspondiente a la creación, instanciación e invocación de cualquier elemento de los sistemas externos. Motramos dicha colaboración mediante el siguiente diagrama de secuencia.

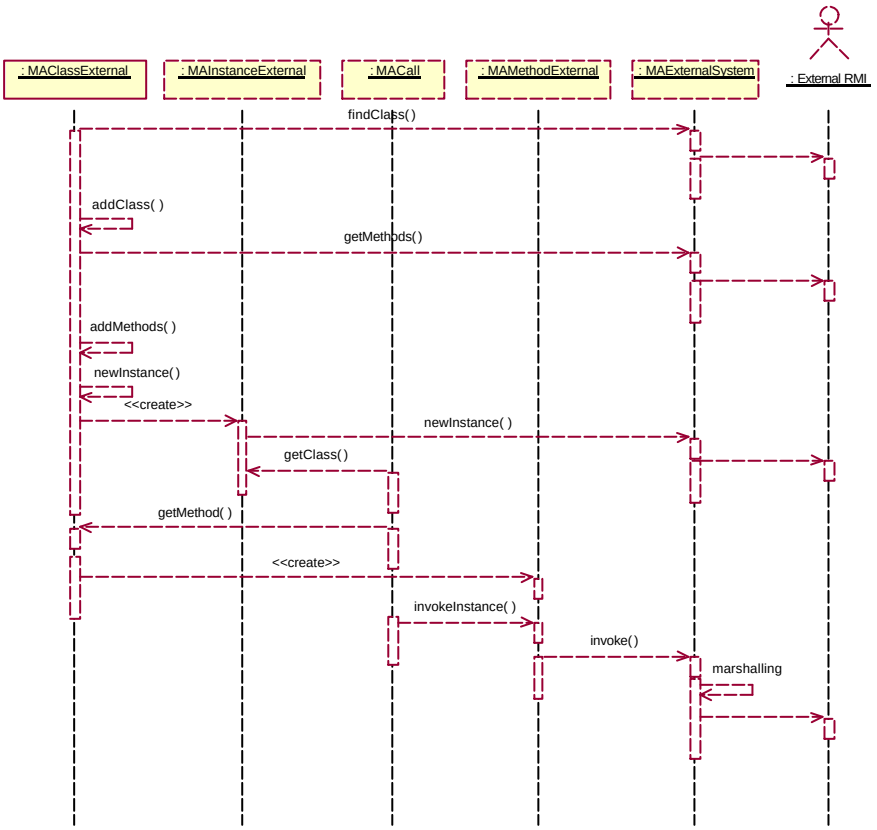


Ilustración 15.4: <-MA-> Diagrama de Secuencia

Los primeros mensajes se corresponden con la creación de nuestra clase a través de una llamada al método `findClass`. Con él, la clase `MAExternalSystem` inspecciona el sistema externo y devuelve una clase construida y lista para ser añadida al sistema a través del método `addClass`.

Para la creación de una instancia se realiza una llamada al método `newInstance`, que se encargará de crear una instancia de la clase `MAInstanceExternal`, que será el envoltorio del objeto externo y a su vez se llama a `newInstance` de `MAExternalSystem` para crear el objeto en el sistema externo.

A continuación, se obtiene el método sobre el que queremos realizar la llamada, se crea una instancia de `MAMethodExternal` y se procede a realizar la invocación mediante el método `invokeInstance` que, a su vez, redirige la invocación a `invoke` de `MAExternalSystem`. Este último, tras realizar las conversiones de tipos oportunas, procede a realizar la invocación al sistema externo.

De esta forma, si deseamos añadir un nuevo sistema externo con el que interoperar, simplemente debemos construir una clase que herede de `MAExternalSystem` e implementar sus métodos, en el Capítulo 16 estudiaremos varios ejemplos concretos.

15.1.5 Federación de búsquedas de clases

Como se comentó con anterioridad, se pretende construir un sistema que almacene de forma virtual todas las clases de todos los sistemas externos con los que se puede interoperar. Para conseguir esto, se irán creando las clases proxy, o envoltorio, apropiadas dentro de nuestra máquina conforme se vayan necesitando. O lo que es lo mismo, a medida que se vayan realizando las declaraciones oportunas. Toda máquina tiene algún almacén de clases dentro del cual se buscan las clases que se desean utilizar. En el caso de nuestra máquina básica, este almacen es un atributo de clase dentro de `MAClass`, denominado `allClasses`. Cuando necesitamos una clase, la buscamos ahí a través del método de clase `getClass`.

Lo que se pretende es ampliar la búsqueda de clases para que, en caso de no estar la clase solicitada en `allClases`, la busque de forma sucesiva dentro de los diversos sistemas externos a los que se tiene acceso. Para llevar a cabo esto, el patrón *cadena de responsabilidad (Chain Of Responsibility)* nos define un modelo de diseño que se ajusta perfectamente. En él se define una clase base denominada *manejador* de la cual derivan el resto de clases sobre las que se desea delegar responsabilidades. En nuestro caso, la busqueda y creación de la clase externa sobre la que deseamos manipular.

Por todo ello vamos a añadir un atributo de clase a `MAClass` que corresponda con el primer m*anejador* dentro de la cadena de responsabilidades, que denominaremos `externalClasses`. De esta forma, dentro de la implementación de `getClass`, si no se encuentra la clase buscada en `allClasses`, se devolverá lo que encuentre dicho manejador. Además, necesitaremos un método nuevo de `MAClass`, que denominaremos `addExternalSystem`, que se encargará de añadir un nuevo sistema externo a la cadena.

En nuestro caso concreto, la propia clase `MAExternalClass` actúa de manejador a través de su atributo `nextExternalSystem`, como se puede observar en la Ilustración 15.3. Observemos que esta forma de federación lleva implícita una prioridad, al igual que ocurre en la resolución de prioridades *Daisy Chain* en la gestión de buses. Es decir, que si en varios sistemas existe la misma clase, se busca primero en `allClasses` y, si se encuentra, se devuelve directamente. Demonos cuenta que esta clase puede ser una clase externa previamente creada en una busqueda anterior. Sino se encuentra, se busca, en `externalClasses`, comenzando la busqueda en el primero de la cadena. Si no está, se buscará en la siguiente y así sucesivamente. De esta forma, los nuevos sistemas tendrán prioridad sobre los antiguos.

15.1.6 Extensibilidad

En esta sección estudiamos las posibilidades que tiene nuestro sistema de ser extendido con nuevos sistemas externos sobre los que deseemos actuar. Una aproximación, ya comentada, consiste en crear, para el nuevo sistema externo, su clase correspondiente encargada de puentear el sistema externo y que, como vimos, hereda directamente de `MAExternalSystem`. Esto se puede llevar a cabo de varias formas, en tiempo de compilación o en tiempo de ejecución, a nivel del metasistema de *Smalltalk* en nuestro caso, o a nivel del metasistema de MA, programándolo nosotros en cualquier lenguaje que acepte nuestra máquina. Todas estas posibilidades las vamos a ir comentando en cada una de las subseciones siguientes, tanto sus ventajas y desventajas como los requisitos que deben cumplir los sistemas subyacentes y modificaciones que debemos realizar, siempre que sea preciso, a nuestro sistema para obtener los resultados deseados.

15.1.6.1 Con clases primitivas

Esta aproximación consiste en añadir una clase que herede directamente de `MAExtenalSystem` y que implemente sus operaciones. Esta técnica sería la más eficiente y dependerá de la forma en que se ha creado la máquina abstracta. Si se ha creado en C++, para añadir la nueva clase es necesario recompilar todo el sistema de nuevo, si bien la eficiencia es la máxima. Algo parecido ocurre si se ha programado el sistema sobre Java, como ocurre con nuestra máquina abstracta. Si bien es más ineficiente, al estar sobre un sistema interpretado como es *Smalltalk*, resulta más sencilla su extensibilidad, ya que habría que crear la nueva clase y añadirla con `addExternalSystem` a la máquina en tiempo de ejecución sin necesidad de compilar más que la nueva clase e, incluso, sin necesidad de parar la ejecución de la máquina en sí.

15.1.6.2 Con componentes de la plataforma

Si el sistema sobre el que se desarrolla la máquina está basado en componentes, se puede modelar el sistema para que la clase `MAExternalSystem` sea un componente de la plataforma. En este caso solo habría que añadir un nuevo componente al sistema o contemplar la posibilidad de añadirlo. Esto se podría hacer si se programa la máquina en C++ sobre Windows con COM o en Java sobre J2EE con Javabeans. Este planteamiento de extensibilidad fue utilizado en la máquina de Nitro que se desarrolla sobre Windows con sistema de componentes COM en Phyton. En el caso de nuestra máquina básica al estar desarrollada sobre un sistema de objetos puro, no hay diferencia con la sección

anterior, son equivalentes. Este sistema aporta el máximo rendimiento con una muy buena flexibilidad, si bien para obtener una optima flexibilidad debería poderse añadir sistemas externos creados en el propio sistema base de la máquina, es decir, programado por el propio usuario de la máquina. Este tipo de sistema es el que se considera a continuación, los usuarios podrán crear sus propias clases derivadas de la clase interna `MAExternalSystem`.

15.1.6.3 Con reflectividad semántica

Para conseguir que el propio usuario de la máquina pueda definir y construir puentes a sistemas externos sobre los que se desea interoperar vamos a necesitar modificar la semántica del sistema. Veamos cómo poder llevarlo a cabo a continuación.

15.1.6.3.1 Construyendo cosificadores internamente

Si el sistema dispone de reflectividad computacional completa, habría que añadir dicha posibilidad a la máquina con las técnicas que se comentaron en la sección 13.3 Reflectividad computacional. En esta sección consideraremos que nuestro sistema dispone de este mecanismo reflectivo. Siendo esto así, podríamos construir mediante programas de usuario las clases derivadas de `MAExternalSystem`. En nuestro caso, tendríamos definido un MOP que nos permitiría interaccionar con la reflectividad estructural propia de *Smalltalk* y podríamos añadir clases, métodos y crear instancias del propio *Smalltalk*, construyendo así las clases que necesitemos. Como ya comentamos en su momento, para obtener una reflectividad computacional completa, uno de los métodos es por medio de la reflectividad estructural del propio metasistema. En este caso, a través de la reflectividad estructural de *Smalltalk*. Otro sistema que permitiría utilizar reflectividad de comportamiento completa sería el sistema Nitro. Si bien las máquinas actuales carecen de este mecanismo y su diseño arquitectónico hace muy complicada su implementación. Actualmente hay estudios para implantar reflectividad estructural en las máquinas abstractas más utilizadas como son Java con OpenJava o en el CLI de .NET. Esta sería la solución más flexible y versátil, aunque no sencilla. En cuanto al rendimiento, los sistemas que ofrecen algún nivel de reflectividad computacional suelen ser poco rápidas.

15.1.6.3.2 Un MOP para los cosificadores a nivel de máquina abstracta

Una solución parcial y más asequible a la mayoría de sistemas, consistiría en cosificar la semántica de la parte del sistema que necesitemos, definiendo con este fin un MOP. De esta forma expondríamos a nivel de usuario los elementos necesarios para añadir nuevos sistemas externos. En el caso de nuestro diseño arquitectónico, tendríamos que cosificar la clase `MAExternalSystem` de tal manera que el usuario pueda heredar de ella e implementar sus propios puentes a sistemas externos. Esta sería una solución estática y para cada sistema externo habría que construir manualmente la clase correspondiente.

Si hacemos uso de la capacidad de reflectividad estructural, podremos, incluso, construir esas clases que heredan de `MAExternalSystem` de forma dinámica, utilizando programas que construyan esas clases en función de las necesidades o las características de los sistemas con los que queramos interoperar. Esta sería una solución dinámica.

15.2 -> MA <-

15.2.1 Introducción

Siguiendo con la analogía con los sistemas reflectivos, en el apartado anterior definimos mecanismos para cosificar los elementos externos dentro del nuestro. En concreto, para cosificar elementos orientados a objetos, como clases, instancias, métodos, de sistemas distribuidos con invocación independiente de la ubicación. Ahora, pretendemos realizar el proceso contrario, que llamaremos el reflejo, reflejar o exponer los elementos de nuestro sistema dentro de los sistemas externos sin necesidad de modificar estos últimos.

Es obvio que, si realizáramos la modificación mencionada anteriormente en los sistemas externos, la comunicación -> MA <- sería inmediata. Es decir, si modificáramos el sistema externo E1 de tal forma que cumpla que MA <- E1 -> MA, vemos que es equivalente a E1 -> MA <- E1, sin embargo el objetivo es construir un sistema que no sea intrusivo, es decir, en el que, independientemente del sentido de la comunicación, no haya que modificar nunca los sistemas externos. Esto es posible siempre a costa de construir nuestro sistema completamente adaptable a ambos sentidos de la comunicación.

Para conseguirlo necesitaremos exponer los cuatro servicios básicos en todas nuestras plataformas: nombres, instanciación, introspección e invocación.

15.2.2 Exposición de la invocación

Como en el caso anterior iremos estudiando las posibles soluciones tratando de resolver la exposición de cada uno de los servicios anteriores, comenzaremos por la invocación.

Para cada sistema externo necesitaremos exponer un servicio de invocación hacia nuestro sistema. Para ello precisaremos de una clase *despachadora* de llamadas a invocaciones entrantes. Veremos que podemos definir una clase abstracta de la cual se heredará para cada sistema externo concreto.

El despachador de llamadas para las diversas plataformas se puede realizar de dos formas. Bien a través de la DSI de CORBA o alguna equivalente bien a nivel IIOP o SOAP, a nivel de protocolo de red, si la plataforma no ofrece DSI. En ambos casos, necesitaremos localizar la clase, crear una instancia e invocar al método indicado. La diferencia radica en la forma de obtener la información de llamada. En el caso de nivel SOAP, se hará procesando directamente los paquetes. En el caso DSI, será en la implementación del `invoke` de la plataforma indicada.

Todas las acciones anteriormente comentadas vienen abstraídas por la raíz de la nueva jerarquía y empaquetadas en el método `dispatch` que cumple con el *Patrón de Diseño Método Plantilla (Template Method)*. En el siguiente diagrama de secuencia se ilustran las llamadas que se realizan dentro de este método hasta llevar a cabo una invocación. Este método es el único que no tienen que redefinir las clases que heredan

de su clase, el resto de métodos deben ser redefinidos y servirán para aportar la información necesaria para llevar a cabo la colaboración.

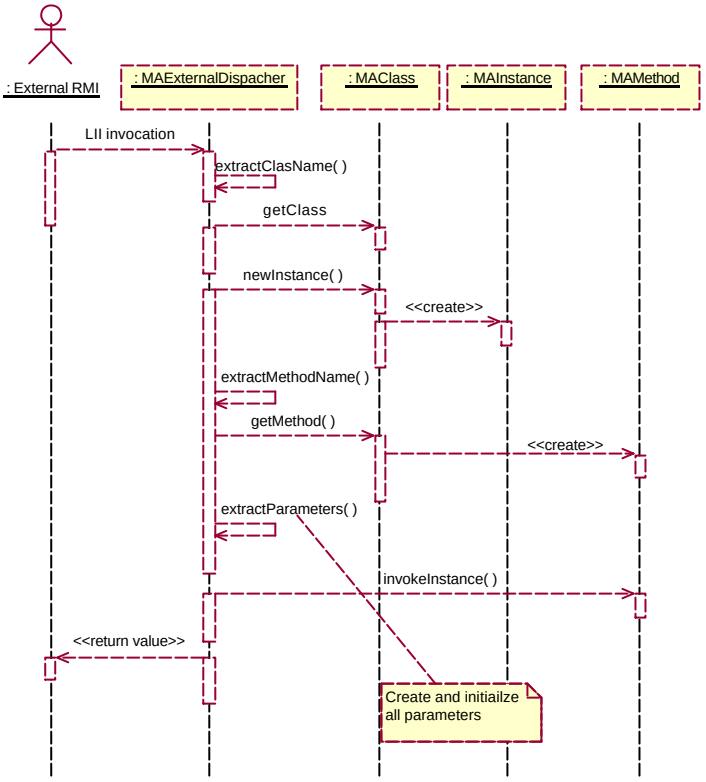


Ilustración 15.5: ->MA<- Diagrama de Secuencias

Con este fin, también es necesario realizar conversiones de tipos o *marshalling*. En este caso, esta tarea viene realizada por el método `extractParameters`.

Todas las llamadas van a ser atendidas de forma concurrente, por lo que habrá que definir objetos o clases estereotipadas con `<<process>>` para indicar que tienen su propio hilo de ejecución.

15.2.2.1 Mediante DSI

Determinadas plataformas como CORBA, permiten la creación e invocación dinámica de *servants*. Así, es posible implementar, a partir de este servicio, los servicios de aceptación de invocaciones.

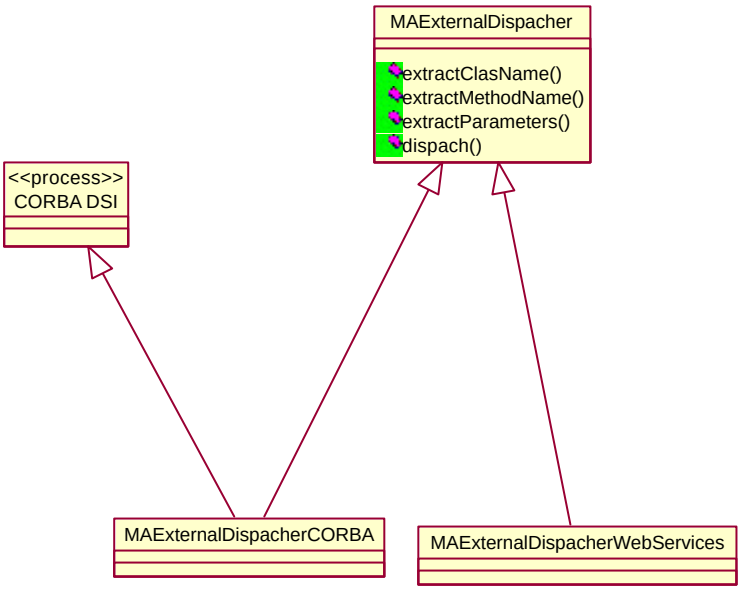


Ilustración 15.6: Mediante DSI

15.2.2.2 A nivel de protocolo de transporte

En las plataformas para las que no exista un servicio de aceptación de invocaciones DSI, se puede crear, a nivel de protocolo de transporte, un gestor de mensajes de invocación. De todos es sabido que para la comunicación entre objetos ORB subyace un protocolo de invocación de Internet denominado IIOP. De igual forma, bajo DCOM existe otro protocolo, y, con Servicios Web, se utiliza el sencillo protocolo de acceso a objetos, SOAP.

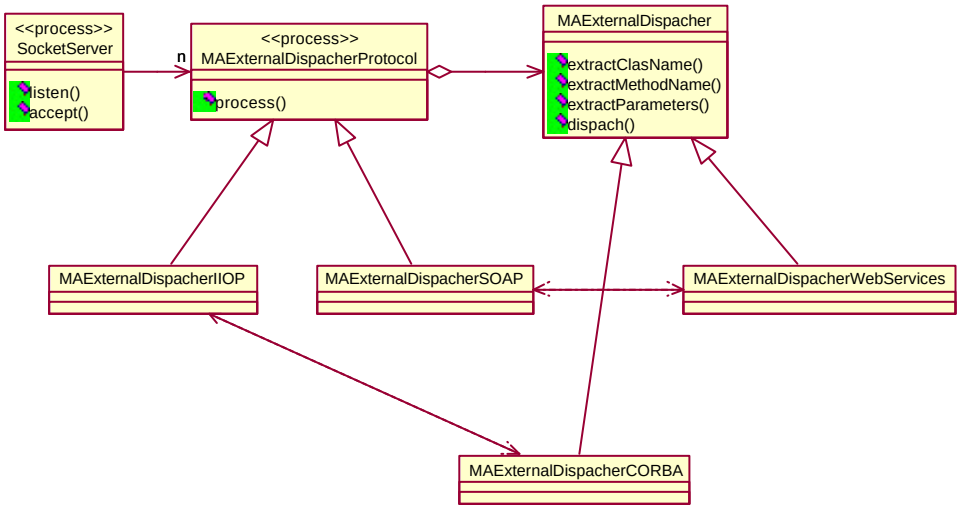


Ilustración 15.7: A nivel de transporte

Una vez más, hacemos uso del patrón puente. En este caso el implementador es `MAExternalDispatcher` y la abstracción viene realizada por `MAExternalDispatcherProtocol`, que se encarga de llamar al método plantilla

dispatch, citado anteriormente. El mensaje viene determinado por un mensaje a nivel de paquete de transporte de protocolo de red.

15.2.3 Exposición de la introspección

Gracias a las capacidades reflectivas de nuestra máquina, va resultar relativamente sencillo exponer los servicios de introspección a los sistemas externos. Básicamente, deberemos solucionar el problema de la comunicación merced a dichos servicios. Para ello, hemos definido una clase abstracta denominada `MAExternalInstrospector` de la cual heredarán las diversas clases que interaccionen con los sistemas externos. En la Ilustración 15.8, podemos observar cómo de ella derivan dos clases activas `MAIntrospectionCORBA` y `MAIntrospectionWSDL`, que serán las encargadas de ofrecer el servicio de introspección a CORBA y a los Servicios Web. La implementación de estas clases deberá realizarse con la tecnología de los sistemas externos con los que interopera, de tal forma que `MAIntrospectionCORBA` debe ser un objeto CORBA que implemente un repositorio de interfaces y que permita federarse con otros repositorios, según especifican los estándares de la OMG. Igual ocurrirá con la clase `MAIntrospectionWSDL`, que deberá implementar un servidor de lenguaje de descripción de Servicios Web basado en XML. La peculiaridad de estas clases es que la información la van a construir de forma dinámica a través de los mecanismos reflectivos de la máquina. Como vemos este diagrama de clases cumple con el patrón adaptador para cada cliente. Dado que vamos a necesitar muchos clientes, uno por plataforma, tendremos a tener que definir este patrón tantas veces como sea necesario.

La asociación que hay entre `MAExternalIntrospection` y `MAClass` realmente no existe, se ha incluido a modo ilustrativo, ya que la clase `MAExternalInstrospection` necesita consultar la información de la clase `MAClass` a través del método de clase `getClass`. Pero esta clase está globalmente accesible con lo que no hace falta ningún atributo dentro de `MAExternalInstrospection` que referencie a `MAClass`.

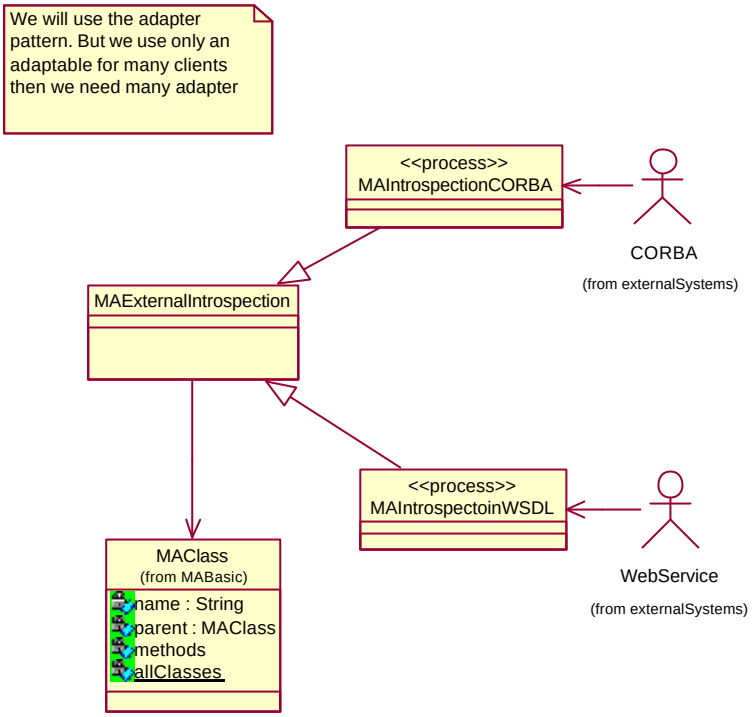


Ilustración 15.8: Introspección Externa

15.2.4 Localización de clases internas

Vamos a seguir de nuevo el patrón *adapter*. La filosofía a utilizar es la misma que la de la exposición de la introspección. Sin embargo en este caso, en vez de describir su contenido, vamos a localizarlas, para, posteriormente, poder crear instancias de ellas. En este caso, rompiendo con la especificación del apartado 11.4, donde indicamos que no utilizaremos referencias, sí que se utilizarán, aunque de forma implícita, sin que el usuario tenga conciencia de ello.

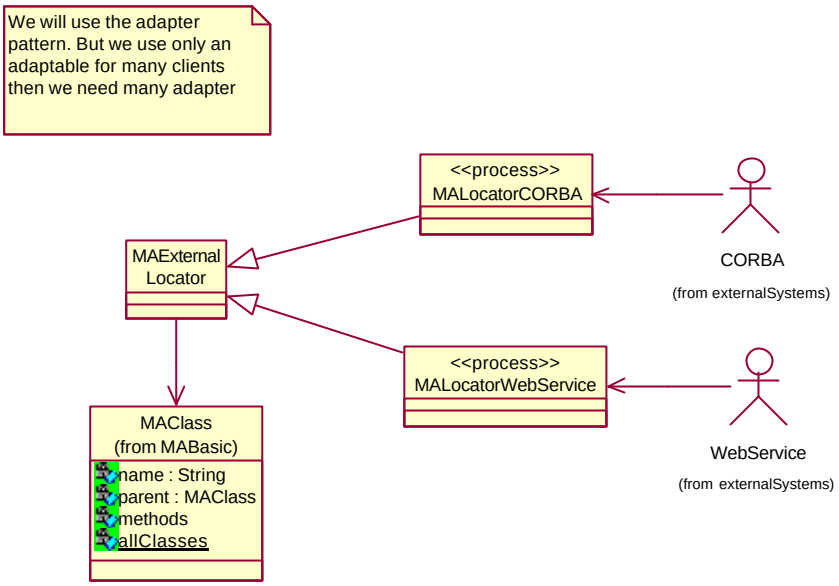


Ilustración 15.9: Localización de clases

15.2.5 Extensibilidad

Las técnicas para conseguir extender nuestra máquina con más sistemas externos serían las mismas que las comentadas en la sección 15.1.6, si bien, en este caso, se complican debido a que hay más clases en juego y que todas ellas deben ejecutarse en diversos hilos de ejecución.

Comentaremos las diversas posibilidades de extensión de esta parte del modelo. Y, a pesar de que habría que aplicarlo a las clases `MAExternalInstrospector`, `MAExternalLocator` y `MAExternalDispatcher`, haremos los comentarios para esta última, considerando que para las otras dos los cambios son similares.

15.2.5.1 Con clases primitivas

Esta aproximación consiste en añadir una clase que herede directamente de `MAExternalDispatcher` y que implemente sus operaciones. Esta técnica sería la más eficiente y dependerá de la forma en que se ha creado la máquina abstracta. Si se ha creado en C++, para añadir la nueva clase es necesario recompilar todo el sistema de nuevo, aunque la eficiencia es la máxima. Algo parecido ocurre si se ha programado el sistema sobre Java, el caso de nuestra máquina abstracta. A pesar de que es mas ineficiente, al estar sobre un sistema interpretado como es *Smalltalk*, resulta más sencilla su extensibilidad ya que habría que crear la nueva clase y añadirla con `addExternalSystem` a la máquina en tiempo de ejecución, sin necesidad de compilar más que la nueva clase e, incluso, sin necesidad de parar la ejecución de la máquina en sí.

15.2.5.2 Con componentes de la plataforma

Si el sistema sobre el que se desarrolla la máquina está basado en componentes, se puede modelar el sistema para que la clase `MAExternalDispatcher` sea un

componente de la plataforma. En este caso, solo habría que crear un nuevo componente al sistema o contemplar la posibilidad de añadirlo. Esto se podría hacer si se programa la máquina en C++ sobre Windows con COM o en Java sobre J2EE con Javabeans. Este planteamiento de extensibilidad fue utilizado en la máquina de Nitro que se desarrolla sobre Windows con sistema de componentes COM en Phyton. En el caso de nuestra máquina básica, al estar desarrollada sobre un sistema de objetos puro, no hay diferencia con la sección anterior. Este sistema aporta el máximo rendimiento con una muy buena, si bien para obtener una flexibilidad óptima se pueden añadir sistemas externos creados en el propio sistema base de la máquina, es decir, programado por el propio usuario de la máquina. Este tipo de sistema son los que se consideran a continuación, donde los usuarios podrán crear sus propias clases derivadas de la clase interna `MAExternalDispatcher`.

15.2.5.3 Con reflectividad semántica

El objetivo es conseguir que el propio usuario de la máquina pueda definir y construir puentes a sistemas externos sobre los que se desea interoperar. Por lo que vamos a necesitar modificar la semántica del sistema, veamos como poder realizarlo a continuación.

15.2.5.3.1 Construyendo cosificadores internamente

Si el sistema dispone de reflectividad computacional completa, habría que añadir dicha posibilidad a la máquina, con las técnicas que se comentaron en la sección 13.3 (Reflectividad computacional). En esta sección consideraremos que nuestro sistema dispone de este mecanismo reflectivo. Siendo esto así, podríamos construir mediante programas de usuario las clases derivadas de `MAExternalDispatcher`. En nuestro caso, tendríamos definido un MOP que nos permitiría interaccionar con la reflectividad estructural propia de *Smalltalk* y podríamos añadir clases, métodos y crear instancias del propio *Smalltalk*, construyendo así las clases que necesitemos. Como ya comentamos en su momento, para obtener una reflectividad computacional completa, uno de los métodos para conseguirlo es a través de reflectividad estructural del propio metasistema, en este caso, a través de la reflectividad estructural de *Smalltalk*. Otro sistema que permitiría utilizar reflectividad de comportamiento completa sería el sistema Nitro, aunque las máquinas actuales carecen de este mecanismo y su diseño arquitectónico hace muy complicada su implementación. Actualmente, hay estudios para implantar reflectividad estructural en las máquinas abstractas más utilizadas como son Java con OpenJava o en el CLI de .NET . Esta sería la solución más flexible y versátil, aunque no sencilla. En cuanto a rendimiento, los sistemas que ofrecen algún nivel de reflectividad computacional suelen ser poco rápidos.

15.2.5.3.2 Un MOP para los cosificadores a nivel de máquina abstracta

Una solución parcial y más asequible a la mayoría de los sistemas consistiría en cosificar la semántica de la parte del sistema que necesitemos definiendo un MOP. De esta forma expondríamos a nivel de usuario los elementos necesarios para añadir nuevos sistemas externos. En el caso de nuestro diseño arquitectónico, tendríamos que cosificar la clase `MAExternalDispatcher`, de tal forma que el usuario pueda heredar de ella e implementar sus propios puentes a sistemas externos. Esta sería una solución estática y, para cada sistema externo, habría que construir a mano la clase correspondiente.

Si hacemos uso de la capacidad de reflectividad estructural, podremos, incluso, construir esas clases que heredan de `MAExternalDispatcher`, de forma dinámica, generando programas que construyan esas clases en función de las necesidades o las características de los sistemas con los que queramos interoperar. Esta sería una solución en tiempo de ejecución.

En todas las consideraciones anteriores siempre debemos tener presente que las tres clases son clases activas, con lo que deberán ejecutarse en su propio hilo de ejecución. Además, se debe permitir la reentrancia, con lo que habrá que utilizar los mecanismos necesarios más adecuados en cada situación para conseguir una adecuada sincronización y administración de los diversos recursos sobre los que se desee operar.

15.3 <-> MA <->

15.3.1 Introducción

En esta sección trataremos de demostrar los resultados que obtenemos cuando ponemos en común las dos colaboraciones expuestas en las secciones anteriores. Por un lado, obtenemos una bidireccionalidad en la interoperabilidad de los sistemas externos con los elementos de nuestra máquina. Por otro lado, gracias a la transitividad en la comunicación de los servicios construidos, obtenemos como consecuencia una comunicación virtual o indirecta entre los propios sistemas externos, cumpliéndose así el propósito de esta tesis. A continuación, estudiaremos ambas propiedades por separado, desde el punto de vista de los diversos servicios utilizados.

15.3.2 Bidireccionalidad

En este apartado analizamos la comunicación obtenida desde el punto de vista de los elementos propios de la máquina abstracta, es decir, desde las clases y objetos creados dentro de la máquina. Estos podrán utilizar objetos de otros sistemas externos a través de la colaboración llamada <- MA ->, y por otro lado, con la colaboración <- MA -> los objetos de la máquina pueden verse en sistemas externos. Para ello, comprobemos la bidireccionalidad de los tres servicios utilizados en la colaboración, a saber, la invocación, la introspección y la localización.

15.3.2.1 En la invocación

Veamos que, por un lado, los objetos externos pueden invocar los métodos de los objetos internos de la máquina. Por otro lado, nuestros objetos internos pueden invocar los métodos de los objetos externos. Para el primer caso, en la Ilustración 15.4, vemos cómo se lleva a cabo. Los objetos externos son envueltos de forma automática gracias a la introspección que ofrecen y podemos referenciarlos de tal forma, que con la instrucción `MACall`, podemos invocar sus métodos. Para el segundo caso, es decir, la exposición de los objetos internos al exterior, hemos visto que hay dos aproximaciones. La denominada DSI, véase la Ilustración 15.6, y la llamada "A nivel de protocolo", véase la Ilustración 15.7. En ambos casos, interceptamos las invocaciones externas para invocar los objetos internos de nuestra máquina.

15.3.2.2 En la introspección

Para poder realizar la invocación necesitamos obtener una descripción de los métodos que contiene una determinada clase o instancia. Por ello es necesario el mecanismo de la introspección, tanto para exponerlo como para cosificarlo. Desde el punto de vista de la máquina abstracta, los mecanismos de reflectividad nos aportan la capacidad de introspección de la misma, que se pueden realizar como se indicó en el Capítulo 13 y que, junto con las clases mostradas en la Ilustración 15.8, permite que los sistemas externos puedan construir dinámicamente o estáticamente llamadas a los objetos internos. Desde otro punto de vista, utilizando los mecanismos de introspección de los sistemas externos y la clase `MAExternalSystem` junto con las mostradas en la Ilustración 15.3, podemos construir los envoltorios comentados anteriormente, en la sección 15.3.2.1, para, después realizar la invocación sobre los mismos.

15.3.2.3 En la localización

Antes de poder inspeccionar una determinada clase o instancia es necesario identificarla y localizarla. Por esto, necesitamos cosificar los servicios externos de localización y exponer los servicios de localización de la máquina abstracta. En nuestro caso hemos considerado como requisito de diseño, Capítulo 11, localizar solamente la clase de los objetos a manipular, dejando para posteriores estudios la localización de clases, en cuyo caso habría que utilizar mecanismos para envolver servicios de directorios de recursos.

15.3.3 Comunicación virtual

Ahora realizamos el análisis de la comunicación desde el punto de vista de los propios sistemas externos, de tal forma que veremos cómo unos se pueden comunicar con otros de forma transparente, a través de nuestros mecanismos, siempre utilizando la máquina abstracta como motor de envío y gestión de mensajes.

15.3.3.1 En la invocación

Una vez obtenida la bidireccionalidad en la invocación a métodos, es posible que un elemento perteneciente a un sistema externo, tras localizar una clase y obtener sus descripciones, desee invocar alguno de sus métodos. En tal caso el sistema recibirá la invocación a dicho método. Si el método pertenece a una clase interna de la máquina, se ejecutará en la propia máquina. En caso de que la clase pertenezca a una clase interna, que es un envoltorio de otra clase externa, esta última será la que reciba la invocación de la llamada. Esto se ilustra al final del diagrama de secuencia de la vista arquitectónica de nuestro modelo, véase la Ilustración 14.8. De esta forma, la clase externa que comenzó la invocación se comunica con otra clase externa de forma transparente y no intrusiva.

15.3.3.2 En la introspección

Para poder realizar la invocación necesitamos obtener una descripción de los métodos que contiene una determinada clase o instancia. Con este fin, es necesario el mecanismo de introspección, tanto para exponerlo como para cosificarlo. Desde el punto de vista de la máquina abstracta los mecanismos de reflectividad aportan la

capacidad de introspección de la misma, que se pueden ejecutar como se indicó en el Capítulo 13 y que junto a las clases mostradas en la Ilustración 15.8, permite que los sistemas externos puedan construir dinámica o estáticamente llamadas a los objetos internos. Desde el otro punto de vista, utilizando los mecanismos de introspección de los sistemas externos y la clase `MAExternalSystem` junto con las mostradas en la Ilustración 15.3, Podemos construir los envoltorios comentados anteriormente, en la sección 15.3.2.1, para a continuación, realizar la invocación sobre los mismos.

15.3.3.3 En la localización

El problema de la localización bidireccional se comentó en la sección 15.3.2.3. Sin embargo, desde el punto de vista de los sistemas externos, debe existir un punto o nexo de unión entre todos los sistemas de localización existentes. Para ello se define, utilizando el patrón c*adena de responsabilidades* (Chain of Responsability), la federación de busqueda, en nuestro caso, para nuestra máquina abstracta básica MA, aplicado al atributo de clase `allClasses` dentro de la clase global `MAClass`, como se muestra en la Ilustración 15.3 y se describe en la sección 15.1.5. En caso de utilizar una máquina virtual concreta, habrá que instanciar este patrón para utilizarlo con su área de clases. Gracias a este mecanismo, un elemento de un sistema externo buscará dentro de su repositorio y recibirá descripciones de objetos que están dentro de la máquina y que pueden ser, a su vez, elementos de sistemas externos. En el diagrama de secuencias de la colaboración general del modelo arquitectónico, se ilustran los pasos de mensajes que se llevan a cabo, véase la Ilustración 14.8.

15.4 Conclusiones

Hemos visto en detalle la vista arquitectónica de la colaboración genérica que forma nuestro modelo arquitectónico y comprobado de forma teórica que nos permite tanto comunicación bidireccional, desde el punto de vista de la máquina abstracta en cuestión, como comunicación virtual, desde el punto de vista de los sistemas externos. A continuación, en el siguiente capítulo, vamos a aplicar estas colaboraciones genéricas a plataformas concretas y reales, con la intención de comprobar que el diseño se adapta bien a ellas y qué problemáticas nos surgen en cada una de ellas.

Capítulo 16:

Varias instanciaciones de la colaboración

Una vez definido el modelo arquitectónico basado en máquina abstractas que permite interoperar a plataformas distribuidas heterogéneas, vamos a ver cómo encaja este al aplicarlo en varias de las plataformas más representativas, a saber DCOM, COBRA y Servicios Web. Estudiaremos cómo aplicar el modelo, qué modelos y servicios de cada plataforman se adaptan mejor. Comprobaremos que, aún considerando las hipótesis de trabajo, cada plataforma aporta sus propias características e impone sus propias restricciones. En esta sección consideraremos como máquina abstracta nuestra máquina básica, sin pretender llegar a ninguna implementación concreta, nos quedaremos en el nivel de diseño. Será en el Capítulo 17 donde aplicaremos varias de las instancias del modelo arquitectónico estudiadas aquí sobre una máquina abstracta y dos plataformas distribuidas concretas, logrando a la construcción completa de un prototipo completamente operativo.

16.1 Con DCOM

16.1.1 Introducción

Utilizaremos el servicio de Automation para que todos los objetos y clases DCOM estén disponibles en nuestra MA como instancias y clases virtuales a las que todo objeto de nuestra máquina básica puede acceder de forma transparente.

Utilizaremos la interfaz IDispath para interceptar, gestionar y distribuir las llamadas realizadas hacia DCOM desde otros sistemas. De tal forma que todos los objetos y clases de la MA, tanto primitivas, como las definidas por el usuario y las virtuales pertenecientes a otras plataformas de invocación síncrona remota sean vistas como objetos DCOM nativos.

A continuación, veremos una pequeña descripción detallada del servicio Automation de DCOM para, seguidamente, ver como poder adaptarlas a nuestro sistema. Obtendremos, así, una comunicación bidireccional de los objetos de nuestra máquina con DCOM.

16.1.2 Dispatch interfaces y Automatización (Automation)

Un cliente COM ha usado hasta ahora un interface COM para comunicarse directamente con un componente COM. *Automation*, formalmente, *OLE Automation* es otra forma para el cliente de controlar el componente. Este servicio es utilizado no solo por aplicaciones como Microsoft Word y Microsoft Excel sino también por lenguajes como Visual Basic y Java.

Automation no está separado de COM pero está construido en su cima. Un a*utomation server* es un componente COM que implementa el interface *IDispatch*. Un a*utomation controller* es un cliente COM que se comunica con el *automation server* a través de su interface *IDispatch*. Un *automation controller* no llama directamente a las funciones implementadas por el *automation server*. En su lugar, el *automation controller* usa las funciones miembros del interface *IDispatch* para, indirectamente, llamar a las funciones del *automation server*.

16.1.2.1 El interface *IDispatch*

IDispatch toma el nombre de una función y la ejecuta. La descripción en IDL del interface *IDispatch* se muestra a continuación:

```
interface IDispatch : IUnknown
{
 HRESULT GetTypeInfoCount (
[out] UINT* pctinfo ) ;
 HRESULT GetTypeInfo (
```

```
[in] UINT iTInfo,
       [in] LCID lcid,
       [out] ITypeInfo** ppTInfo ) ;

 HRESULT GetIDsOfNames (
       [in] const IID& riid,
       [in, size_is(cNames)] LPOLESTR* rgszNames,
       [in] UINT cNames,
       [in] LCID lcid,
       [out, size_is(cNames)] DISPID* rgDispId ) ;

 HRESULT Invoke (
       [in] DISPID dispIdMember,
       [in] const IID& riid,
       [in] LCID lcid,
       [in] WORD wFlags,
       [in, out] DISPPARAMS* pDispParams,
       [out] VARIANT* pVarResult,
       [out] EXCEPINFO* pExcepInfo,
       [out] UINT* puArgErr ) ;
} ;
```

Las dos funciones más interesantes de `IDispatch` son `GetIDsOfNames` e `Invoke`. La función `GetIDsOfNames` lee el nombre de una función y devuelve su *dispatch ID* o `DISPID`. Un `DISPID` no es un GUID, pero es también un entero largo (`LONG`). El `DISPID` identifica una función aunque no es único sino para una implementación particular de `IDispatch`. Cada implementación de `IDispatch` obtiene su propio IID.

Para ejecutar la función, el *automation controller* pasa el DISPID a la función miembro `Invoke`. La función `Invoke` puede usar el `DISPID` como un índice en un array de punteros a funciones, muy parecido a los interfaces normales de COM. Sin embargo, un *automation server* no necesita implementar `Invoke` de esta manera. Un *automation server* puede usar un *Case* que ejecute diferente código dependiendo del `DISPID`.

16.1.2.1.1 Dispinterfaces

El conjunto de funciones implementadas por una implementación de `IDispatch::Invoke` se llama *dispatch interface* o *dispinterface*,de forma abreviada.

Una implementación de `IDispatch::Invoke` constituye un conjunto de funciones a través de las cuales un *automation controller* y un *automation server* se comunican. Por tanto, las funciones implementadas por `Invoke` constituyen un interface, pero no un interface COM.

16.1.2.1.2 Interfaces duales (Dual interfaces)

Un interface dual (*dual interface*) es un *dispinterface* que hace todas las funciones que están disponibles a través de `Invoke` accesibles también de forma directa a través de `vtbl`.

16.1.2.2 Uso de IDispatch

El poder de `Invoke` es que puede ser usada polimórficamente. Cualquier componente que implemente `Invoke` puede ser llamado usando el mismo código. Sin embargo, una de las tareas de `IDispatch::Invoke` es pasar parámetros a las funciones que ejecuta.Los tipos de los parámetros que `Invoke` puede pasar a las funciones de un dispinterface están limitados en número.

16.1.2.2.1 Los argumentos de Invoke

El primer parámetro de la función `Invoke` es el `DISPID` de la función que el *controller* quiere llamar. El segundo parámetro está reservado y debe ser siempre `IID_NULL`. El tercer parámetro contiene información de localización.

Los atributos IDL `propget` y `propput` significan que una función COM ha sido tratada como una propiedad. Cuando el MIDL genera el fichero de cabecera para funciones marcadas como `propget` o `propput`, adjunta el prefijo *get_* o *put_* respectivamente a la función. Esto guarda relación con el cuarto parámetro de la función `Invoke`. Un mismo nombre puede estar asociado con cuatro funciones diferentes: una función normal, una función para poner una propiedad, una función para poner una propiedad por referencia y una función para obtener una propiedad. Todas estas funciones con el mismo nombre tendrán igualmente el mismo `DISPID`, pero podrán estar implementadas de formas completamente diferentes. Por lo tanto, `Invoke` necesita conocer a qué tipo de función está llamando. El cuarto parámetro suministra la información necesaria con uno de los siguientes valores:

- `DISPATCH_METHOD`
- `DISPATCH_PROPERTYGET`
- `DISPATCH_PROPERTYPUT`
- `DISPATCH_PROPERTYPUTREF`

El quinto parámetro de `IDispatch::Invoke` contiene los parámetros de la función que está siendo invocada. Este parámetro es una estructura `DISPPARAMS`. La definición de la estructura `DISPPARAMS` se muestra a continuación:

```
typedef struct tagDISPPARAMS {
VARIANTARG*  rgvarg ;
       DISPID* rgdispidNamedArgs ;
       unsigned int cArgs ;
       unsigned int cNamedArgs ;
} DISPPARAMS ;
```

El primer campo de la estructura `DISPPARAMS`, `rgvarg`, es un array de argumentos. `cArgs` es el número de argumentos de ese array. Cada argumento es de tipo `VARIANTARG` y esto es así porque el número de tipos que se pueden pasar entre un *automation controller* y su *server* es limitado. Sólo los tipos que puedan ser puestos en una estructura `VARIANTARG` pueden ser pasados por medio del dispinterface o dual interface.

Una estructura `VARIANTARG` es lo mismo que una `VARIANT`. La definición de la estructura `VARIANT` se muestra a continuación:

```
typedef struct tagVARIANT {
VARTYPE vt;
WORD wReserved1;
WORD wReserved2;
WORD wReserved3;
union {
  BYTE             bVal;       // VT_UI1
  SHORT            iVal;       // VT_I2
  LONG             lVal;       // VT_I4
  FLOAT            fltVal;     // VT_R4
  DOUBLE           dblVal;     // VT_R8
  VARIANT_BOOL     boolVal;    // VT_BOOL
  SCODE            scode;      // VT_ERROR
  CY               cyVal;      // VT_CY
  DATE             date;       // VT_DATE
  BSTR             bstrVal;    // VT_BSTR
  IUnknown*        punkVal;    // VT_UNKNOWN
  IDispatch*       pdispVal;   // VT_DISPATCH
  SAFEARRAY*       parray;     // VT_ARRAY
  BYTE*            pbVal;      // VT_BYREF|VT_UI1
  SHORT*           piVal;      // VT_BYREF|VT_I2
  LONG*            plVal;      // VT_BYREF|VT_I4
  FLOAT*           pfltVal;    // VT_BYREF|VT_R4
  DOUBLE*          pdblVal;    // VT_BYREF|VT_R8
  VARIANT_BOOL*    pboolVal;   // VT_BYREF|VT_BOOL
  SCODE*           pscode;     // VT_BYREF|VT_ERROR
  CY*              pcyVal;     // VT_BYREF|VT_CY
  DATE*            pdate;      // VT_BYREF|VT_DATE
  BSTR*            pbstrVal;   // VT_BYREF|VT_BSTR
  IUnknown**       ppunkVal;   // VT_BYREF|VT_UNKNOWN
  IDispatch**            ppdispVal;  // VT_BYREF|VT_DISPATCH
  SAFEARRAY**            pparray;    // VT_BYREF|VT_ARRAY
  VARIANT*         pvarVal;    // VT_BYREF|VT_VARIANT
  PVOID            byref;      // Generic ByRef
  CHAR             cVal;       // VT_I1
  USHORT           uiVal;      // VT_UI2
  ULONG            ulVal;      // VT_UI4
  INT              intVal;     // VT_INT
  UINT             uintVal;    // VT_UINT
  DECIMAL*         pdecVal;    // VT_BYREF|VT_DECIMAL
  CHAR*            pcVal;      // VT_BYREF|VT_I1
  USHORT*          puiVal;     // VT_BYREF|VT_UI2
  ULONG*           pulVal;     // VT_BYREF|VT_UI4
  INT*             pintVal;    // VT_BYREF|VT_INT
  UINT*            puintVal;   // VT_BYREF|VT_UINT
 } ;
```

Una estructura `VARIANT` es solo una gran unión de tipos diferentes que sirve para almacenar diferentes variables de una forma común. Lo más importante es que tanto los dispinterfaces como los dual interfaces pueden pasar solamente tipos que puedan ser expresados como `VARIANT`.

El sexto parámetro de `Invoke`, `pVarResult`, es un puntero a una estructura `VARIANT` que contiene el resultado del método o `propget` ejecutado por `Invoke`. Este parámetro puede ser `NULL` para métodos que no retornen valores como, por ejemplo, `propputs` y `propputrefs`.

El penúltimo parámetro de `Invoke`, `pExcepInfo`, es un puntero a una estructura `EXCEPINFO`. Si el método o la propiedad ejecutada por la llamada a `Invoke` produjo como resultado una excepción, esta estructura `EXCEPINFO` se rellenará con información sobre la situación. Las estructuras `EXCEPINFO` se usan en el mismo tipo de situaciones en las que C++ utiliza excepciones.

La definición de la estructura `EXCEPINFO` se muestra a continuación:

```
typedef struct tagEXCEPINFO {
       WORD        wCode ;           // Código de error
 WORD        wReserved ;
      BSTR         bstrSource ;      // Fuente de la excepción
 BSTR        bstrDescription ; // Descripción del error
 BSTR        bstrHelpFile ;
 DWORD       dwHelpContext ;
 ULONG       pvReserved ;
 ULONG       pfnDeferredFillIn ;
 SCODE       scode ;                 // Valor de retorno
} EXCEPINFO ;
```

Tanto el código de error (`wCode`) como el valor de retorno (`scode`) deben contener un valor que identifique el error.

Si el valor retornado por `IDispatch::Invoke` es bien `DISP_E_PARAMNOTFOUND` bien `DISP_E_TYPEMISMATCH`, el índice del argumento correspondiente al error es devuelto en el parámetro final, `puArgErr`.

16.1.2.2.2 El tipo VARIANT

Una estructura `VARIANT` es inicializada usando `VariantInit` que pone el campo `vt` a `VT_EMPTY`. Después de la llamada a `VariantInit`, el campo `vt` indica el tipo de dato almacenado en la unión `VARIANT`.

El uso de `VARIANT` permite eliminar casi completamente el chequeo estático de tipos en favor del chequeo de tipos en tiempo de ejecución. Retrasar el chequeo de tipos hasta el tiempo de ejecución requiere que los métodos y propiedades *dispatch* chequeen los tipos de los argumentos que reciben.

Aunque los dispinterfaces pueden retornar error para los tipos que no manejan, es preferible que sean capaces de hacer conversiones de tipos por el programador. Para hacer estas conversiones de una forma estándar, `Automation` proporciona una función llamada `VariantChangeType`.

La declaración de la función `VariantChangeType` se muestra a continuación:

```
HRESULT VariantChangeType (
 VARIANTARG* pVarDest,
 VARIANTARG* pVarSrc,
 unsigned short    wFlags,
 VARTYPE           vtNew
) ;
```

Un método de un dispinterface puede también tener argumentos opcionales. Si no se quiere proporcionar un valor para un argumento opcional, se pasa una estructura

VARIANT con los campos vt a VT_ERROR y scode a DISP_E_PARAMNOTFOUND. El método debe proporcionar su propio valor por defecto.

16.1.2.2.3 El tipo de dato BSTR

Un BSTR (*Basic String* o *Binary String*) es un string que mantiene una cuenta de sus caracteres. Esta cuenta de caracteres se almacena antes que el propio string, por lo que no se puede declarar un BSTR y asignarle un array de caracteres de esta forma:

```
BSTR bstr = L“Esto no es correcto” ; // Mal!
```

ya que entonces la cuenta de caracteres no podría obtener su valor apropiadamente. Se debe usar la funcion Win32 API SysAllocString. Un BSTR necesita ser liberado de memoria con SysFreeString. Un BSTR también puede contener múltiples carácteres ‘\0’ dentro del string.

16.1.2.2.4 El tipo de dato SAFEARRAY

Otro tipo de dato especial que puede ser pasado al dispinterface es el SAFEARRAY. La definición del tipo SAFEARRAY se muestra a continuación:

```
typedef struct tagSAFEARRAY {
 unsigned short cDims;         // Número de dimensiones
 unsigned short fFeatures;
 unsigned long cbElements;     // Tamaño de cada elemento
 unsigned long cLocks;
 BYTE* pvData;                 // Puntero a los datos
 [size_is(cDims)] SAFEARRAYBOUND rgsabound[ ];
} SAFEARRAY ;

typedef struct tagSAFEARRAYBOUND {
 ULONG cElements;
 LONG lLbound;
} SAFEARRAYBOUND ;
```

El campo *fFeatures* describe el tipo de dato almacenado en el SAFEARRAY. Los valores permitidos son los siguientes:

- FADF_BSTR Un array de BSTRs
- FADF_UNKNOWN Un array de *IUnknown**
- FADF_DISPATCH Un array de *IDispatch**
- FADF_VARIANT Un array de VARIANTs

La *Automation Library*, OLEAUT32.DLL, incluye varias funciones para manipular SAFEARRAY. Estas funciones empiezan todas con el prefijo SafeArray.

16.1.2.3 Librerías de tipos (*Type Libraries*)

El chequeo de tipos en tiempo de ejecución consume mucho tiempo y puede ocultar errores en el programa. La solución para este problema en COM son las librerías de tipos (*type libraries*), que proporcionan información sobre componentes, interfaces,

métodos, propiedades, argumentos y estructuras. El contenido de una *type library* es el mismo que un fichero de cabecera de C++. Una *type library* es una versión compilada de un fichero IDL a la que se puede acceder desde un programa. La *Automation Library* proporciona componentes estándar para la creación y lectura de este fichero binario.

16.1.2.3.1 Creación de una Type Library

La función de la *Automation Library* `CreateTypeLib` crea una *type library*. `CreateTypeLib` devuelve un interface `ICreateTypeLib`, que puede ser usado para rellenar la *type library* con información. Sin embargo, no suele utilizarse este interface. En su lugar, se usa IDL y el compilador MIDL.

La clave para construir una *type library* usando IDL es la declaración *library*. Todo lo que aparezca dentro del bloque *library* -el bloque de código que sigue a la palabra *library*- será compilado dentro de la *type library*. Cuando el compilador MIDL encuentra la declaración *library* en un fichero IDL, automáticamente genera una *type library*.

Una vez que se ha creado la *type library*, esta se puede enviar a un fichero separado o incluirla en el EXE o DLL como recurso. La mayoría de los programadores prefieren esto último porque simplifica la instalación y configuración.

16.1.2.3.2 Uso de las Type Libraries

El primer paso para usar una *type library* es cargarla. Existen varias funciones para cargar una *type library*. Primero, ha de intentarse con `LoadRegTypeLib`, que intenta cargar la *type library* desde el Registro de Windows. Si esta función falla, se debe usar `LoadTypeLib`, que carga la *type library* desde disco a partir de un nombre de fichero que recibe como parámetro, bien `LoadTypeLibFromResource`, que carga la *type library* desde un EXE o una DLL. `LoadTypeLib` registra la *type library* cuando la carga, aunque si se le proporciona el *pathname* entero a `LoadTypeLib`, entonces no la registra. En ese caso, hay que llamar a `RegisterTypeLib` después de `LoadTypeLib` para completar el proceso con éxito.

Una vez que se ha cargado la *type library*, ya se puede usar. `LoadTypeLib` y las demás funciones devuelven un puntero al interface `ITypeLib`, que se usa para manipular la *type library*. En una *type library* habitualmente se encuentra la información sobre un interface o un componente, es decir una *type library* es básicamente un conjunto de información sobre interfaces y componentes. Para obtener datos sobre un interface o un componente se debe pasar su CLSID o IID a la función `ITypeLib::GetTypeInfoOfGuid`, la cual retorna un puntero al interface `ITypeInfo` del ítem deseado.

Usando el puntero `ITypeInfo`, se puede obtener cualquier información deseada sobre componentes, interfaces, métodos, propiedades, estructuras, etc. Pero la mayoría de los programadores de C++ nunca suelen hacer uso de este recurso, excepto cuando implementan `IDispatch`. `ITypeInfo` puede implementar automáticamente `IDispatch`.

16.1.2.3.3 Type Libraries en el Registro

Las *type libraries* se registran por sí mismas. En *HKEY_CLASSES_ROOT\TypeLib* se encuentra una lista de LIBID, que son GUID que identifican *type libraries*.

16.1.2.4 Elección del tipo de interface

Hay tres opciones para la comunicación entre un cliente y un componente:

- Vtbl interfaces

- Dual interfaces

- Dispinterfaces

Si el componente va a ser accedido solo desde lenguajes compilados tales como C++, se deben usar interfaces `vtbl` normales, que son mucho más rápidos que los dispinterfaces. Además, es más fácil para los programadores de C++ acceder a ellos.

Si el componente va a ser accedido a través de Visual Basic o Java, debe implementar un dual interface.

A no ser que se necesite realmente construir componentes en tiempo de ejecución, no se deben implementar dispinterfaces puros. Los dual interfaces son mucho más versátiles que los dispinterfaces.

Otro factor para tomar una decisión puede ser la velocidad. Si se tiene un *in-proc component*, un interface `vtbl` es 100 veces más rápido que un dispinterface. Si se tiene un *out-of-proc component*, un interface `vtbl` es sólo un poco más rápido que un dispinterface. Si el componente es remoto, el interface usado no produce diferencia.

16.1.3 DCOM <- MA -> DCOM

16.1.3.1 Introducción

En esta sección veremos la instanciación de la colaboración de invocaciones outdoor, (hacia fuera). Como se mostró en la sección 15.1.4 gracias al patrón *bridge*, solamente vamos a necesitar refinar la clase MAExternalSystem y definir sus métodos. Para ello utilizaremos los *servicios automation* de DCOM vistos anteriormente y en concreto la interfaz `IDispatch`.

En la Ilustración 16.1 se muestra un diagrama de clases donde se define una nueva clase, `MAExternalSystemDCOM`, que hereda de `MAExternalSystem` y está asociada a través del atributo `theIDispatch` que, como su nombre sugiere, es un puntero a la interfaz `IDispath` de DCOM, que será nuestro enlace con la plataforma distribuida externa DCOM.

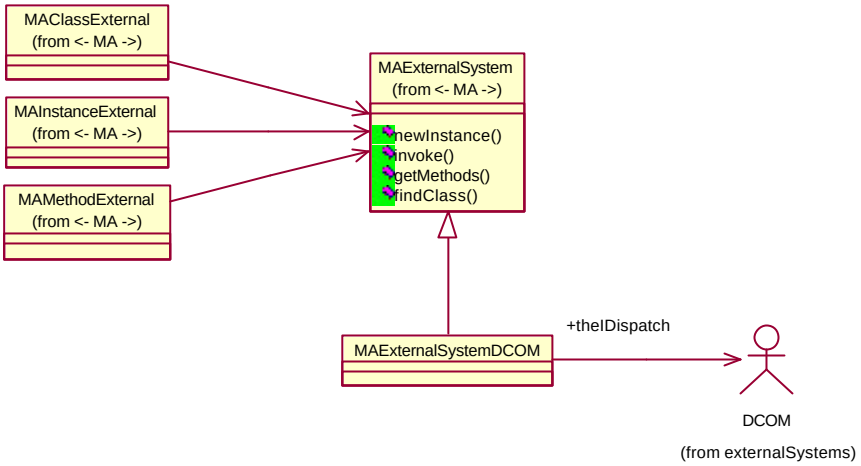


Ilustración 16.1: DCOM <- MA -> DCOM

Comenzaremos viendo cómo localizar las clases externas.Luego veremos como nos ocuparemos de obtener su información estructural para construir, a partir de dicha información, las clases, instancias y métodos oportunos de tal forma que todos los objetos de la máquina vean los objetos DCOM como propios a través del servicio *automation*.Posteriormente, veremos como construir la invocación a los métodos externos DCOM. En realidad todas estas tareas serán realizadas por los diversos métodos redefinidos de la clase `MAExternalSystem`, como son `invoke()`, `newInstance()`, `getMethods()`y `findClass()`.

16.1.3.2 Introspección

Una vez obtenido un puntero a la interfaz `IDispatch` de la clase con la que deseamos comunicarnos, procederemos a utilizar sus métodos de gestión de librerías de tipos para realizar la introspección y obtener información acerca de los métodos y los párametros necesarios.De tal manera que, a su debido tiempo, se puedan realizar las invocaciones y convesiones/adaptaciones de tipos o *marshalling*.

16.1.3.3 Localización

A través del método nativo de DCOM `CLSIDFromProgID` obtendremos el CLSID de la clase a la que deseamos acceder.Una vez obtenida esta, mediante `CoCreateInstance` crearemos una instancia de dicha clase a través de un puntero a la interfaz `IDispatch`.

La identificación de componentes en DCOM está formada por dos partes separadas por '.'. La parte izquierda consiste en el nombre el programa que contiene la clase o componente.En el lado derecho tendremos el nombre de la clase concreta sobre la que queremos crear la invocación.

Vamos a seguir la sencilla regla la de sustituir el '.' por un '_' y toda la cadena completa será el nombre de una clase dentro de nuestra máquina. Esta nomenclatura la puede cambiar el instanciador del modelo arquitectónico. Así, el corrector ortográfico

de la aplicación Word que tiene como ProgID Word.Application se convertirá en una clase dentro de nuestro sistema denominada Word-Application.

Para localizar e importar las clases externas dentro de nuestra máquina vamos a utilizar, como se comentó en la sección 15.1.5 (Federación de búsquedas de clases), el patrón cadena de responsabilidades y redefinir la clase MAExeternalSystem que, utilizando CLSIDFromProgID y siguiendo el diagrama de secuencia de la Ilustración 15.2, localizará la clase, la introspeccionará y construirá una clase envoltório con sus métodos apropiados. Ahora, ya estÁ lista esta clase dentro de nuestra máquina para crear instancias de ellas e invocar sus métodos. La proxima vez que se haga referencia a esta clase, ya se encontrará en allClasses y se devolverá el envoltorio previamente construido.

16.1.3.4 Instanciación

Una vez creada la clase dentro de nuestra máquina, podremos crear instancias de ella y manipularla exactamente igual que cualquier otra clase. Las instancias creadas vendrán gestionadas por MAInstanceExternal, que se asocia con una instancia de la clase MAExternalSystemDCOM que contiene un único atributo theIDispatch que es un puntero a la instancia que hemos creado.

Para crearla se llamará al método newInstance donde se realizará una llamada a newInstance de MAExternalSystemDCOM, quien se encargará de construir el IDispatch apropiado, crear una instancia de MAExternalSystemDCOM que la contenga, crear una instancia de MAInstanceExternal y asociarle la instancia previamente creada de MAExternalSystemDCOM que encapsula el puntero a IDIspatch.

16.1.3.5 Invocación

Para cada método de la clase externa se creará una instancia de la clase MAMethodExternal, que como atributo, contiene simplemente el nombre heredado de MAMethod y cuyo método invokeInstance realiza una llamada al método invoke de la clase MAExternalSystemDCOM, pasándole tanto las instancias como los parámetros originales que recibe Es en este último método, desde donde se debe sacar el puntero a IDispacth de la instancia sobre la que se desea ejecutar el método, construir la invocación, realizando las diversas conversiones tanto de los parámetros como de los valores de retorno, realizar la invocación y gestionar los posibles errores que surjan.

16.1.4 DCOM -> MA <- DCOM

16.1.4.1 Introducción

En esta sección veremos la instanciación de la colaboración de invocaciones indoor. Tal y como se mostró en la sección 15.2.2, tenemos dos opciones: mediante DSI y a nivel de protocolo de transporte. En este caso, como se introdujo en la sección anterior de este capítulo vamos a utilizar el equivalente a DSI que proporciona DCOM, es decir, la interfaz IDispatch. En la Ilustración 16.2, se muestra la la jerarquía de clases básica que vamos a necesitar, en concreto MAExternalDispatcherDCOM que hereda de MAExternalDispacher y de la interfaz primitiva IDispatch encapsulada en la clase activa DCOM IDispatch.

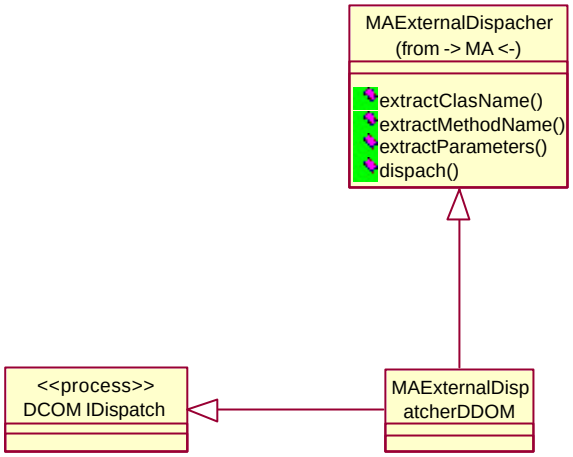


Ilustración 16.2: DCOM -> MA <- DCOM

16.1.4.2 Invocación

Un mismo objeto puede implementar varios `IDispatch` a través de varios IID diferentes, lo que nos permite, con un solo componente, recibir llamadas hacia varias clases.Estos IID pueden ser consultados a través de la función estándar `QueryInterface`. De esta forma, con una sola instancia de la clase activa `MAExternalDispathcerDCOM` podremos atender todas las peticiones externas procedentes de objetos DCOM, siempre a través del servicio *automation*. Por medio de un IID se le indicará sobre que clase se desea realizar la invocación. El resto de parámetros se pasarán a través del método `Invoke` de la interfaz `IDispatch` comentada anteriormente, en la sección 16.1.2.2.1.

16.1.4.3 Introspección

Nuestro diseño para gestionar la instrospección de clases externas sobre nuestro sistema, visto en la sección 15.2.3, encaja muy bien con el modelo de librería de tipos de automation. Para ello, crearemos, una clase `MAIntrospectionDCOM` que herede de `MAExternalIntrospection`, vease la Ilustración 15.8, y que permita a los clientes de automation obtener la descripción de las clases internas de MA expresadas según la estructura de información indicada por la especificación de la librería de tipos de COM (*Type Library*). Esta clase se encargará de recibir las peticiones realizadas al método `GetTypeInfo` de la interfaz `IDispatch` para, así, inspeccionar el metasistema y proporcionar la información solicitada.

16.1.4.4 Localización

Cada vez que se añada una clase al área de clases de MA, esta, como ya vimos en la sección 15.1.5, además se añadirá al resto de sistemas de localización externos o área de clases externas. En el caso de DCOM, este servicio viene dado por el *Registro del Sistema Operativo Windows* y está basado en GUID. Así, cuando se añada una clase al sistema también se añadirá al registro del sistema a través de una GUID creada para la nueva clase y un puntero al proceso de gestión de invocación `MAExternalDispathcerDCOM`, explicado en la sección anterior, Invocación.

Hasta este punto el servicio de coherencia de los sistemas de localización externos se mantiene; pero, como se comentó en la sección 15.2.4, para obtener un sistema completamente coherente en cuanto a localización es necesario que los sistemas externos de localización puedan federarse entre sí. En nuestro caso es preciso que los registros de los sistemas Windows puedan federarse, cosa que por el momento resulta imposible. Es por ello que no encontramos con la imposibilidad de poder mantener todos los objetos externos accesibles desde DCOM. Solamente se podrán localizar acceder desde DCOM a sistemas externos cuyas clases previamente se encuentren en el área de clases.

16.1.5 Conclusiones

Para la comunicación *outdoor* hemos visto que resulta ideal el servicio de *automatión* si bien sería más eficiente utilizar la invocación nativa de DCOM es decir las vtables. Al tratarse de comunicaciones remotas,la diferencia no es apreciable frente al retraso incluido por las líneas de comunicación. La comunicación de información se ve restringida a los tipos representables mediante VARIANT, aunque resulta un poco escaso para nuestro caso de estudio, es suficiente ya que por el momento tomamos como hipótesis de trabajo la utilización de tipos simples.

Por otro lado, para las comunicaciones *indoor*, observamos el inconveniente de la utilización del Registro del Sistema como herramienta de localización y almacenamiento. Este registro no ha sido pensado para poder federarse o extenderse, su rigidez presenta el inconveniente de que solo estarán accesibles en los servicios de localización de las plataformas externas las clases que previamente hayan sido instaladas dentro del área de clases de MA.El motivo de esto es que al no poder federarse el registro del sistema en Windows, no se puede controlar o ampliar la búsqueda dentro del mismo.

Debemos notar que, para realizar la instanciación comentada aquí, es necesario que o la máquina o la plataforma sobre la que se desarrolla soporten el sistema DCOM. Actualmente, esto implica que el sistema operativo subyacente debe ser alguna de las versiones de Windows.

16.2 Con CORBA

16.2.1 Introducción

La siguiente plataforma sobre la que vamos a aplicar nuestro modelo arquitectónico será CORBA. Esta plataforma, como se vió en la sección 7.1, se creó como un modelo arquitectónico para la construcción de aplicaciones distribuidas principalmente orientadas a objetos. Este modelo pretende ser independiente tanto del sistema operativo, como de la plataforma. Define varios elementos y servicios. Nuestro modelo se podría aplicar a nivel IIOP o a nivel de servicios ORB. Hemos considerado esta última opción por su sencillez, adaptabilidad e idoneidad con nuestro modelo. En contraste con el último caso de estudio, sobre Servicios Web en el que veremos que resulta más adecuado aplicarlo a un nivel de abstracción más bajo, por su sencillez a ese nivel y la falta de especificación a nivel abstracto superior.

Veremos cómo se comporta nuestro modelo con CORBA y qué ventajas y restricciones nos impone. Los servicios ORB que vamos a considerar van a ser la invocación de interfaces dinámica DII. La invocación a esqueletos dinámicos DSI, el repositorio de interfaces y los servicios de localización.

Primero, veremos cada uno de los servicios detenidamente y, posteriormente, consideraremos como encajan cada uno de ellos dentro del modelo.

La diferencia fundamental de CORBA con respecto a otros modelos es que está orientado a objetos o, más precisamente, orientado a referencias a objetos. Así para acceder a los objetos, primero deben existir como instancias dentro del propio servidor.

16.2.2 Servicios dinámicos de CORBA

16.2.2.1 DII

Antes de llamar a un método de un objeto dinámicamente, se debe localizar el objeto y obtener una referencia a él. Con esta referencia podemos obtener la interfaz y construir la llamada dinámicamente.

1. Obtener el nombre de la interfaz.
2. Obtener la descripción de métodos del *Almacén de Interfaces.*
3. Crear una lista de argumentos.
4. Crear una llamada.
5. Realizar la llamada.

Las interfaces para la invocación dinámica se encuentran dentro del núcleo de CORBA pero están repartidas en cuatro grupos:

1. `Object` es una interfaz de seudo-objeto que nos define operaciones que todos los objetos CORBA deben soportar.

2. `Request` es una interfaz de seudo-objeto que nos define las operaciones que podemos realizar sobre los objetos remotos.
3. `NVList` es una interfaz de seudo-objeto que nos ayuda a construir listas de parámetros para realizar las llamadas a operaciones. Una lista `NVList` está formada por un conjunto de datos autodescriptivos llamado `NamedValues`, veamos la IDL que describe su estructura:

```
Struct NamedValue
{ Identifier name;        //nombre del artumento
  any          argument;  //artumento
  long         len;       //longitud del valor del argumento
  Flags        arg_modes; //in, out, o inout
 };
```

4. `ORB` es una interfaz de seudo-objeto que define métodos de propósito general para el `ORB`.

A continuación, describimos los métodos que nos interesan a la hora de realizar llamadas dinámicas de las interfaces anteriormente citadas.

`Corba.Object.get_interface`

Se llama para obtener las interfaces que tiene un determinado objeto, devuelve una referencia a un objeto que se encuentra en el Almacén de Interfaces que se llama `InterfaceDef`. Este objeto describe las interfaces en las que estamos interesados.

`Corba.Object.create_request`

Se llama para crear un objeto `Request.` Este es el objeto que se envía cuando deseamos realizar una llamada. En él guardamos el nombre del método y los parámetros.

`Corba.Object._request`

Este método es una versión reducida del anterior se usa cuando el método al que queremos llamar no necesita parámetros.

`Corba.Request.add_arg`

Añade argumentos incrementalmente a un objeto `Request`.

`Corba.Request.invoke`

Realiza una llamada al método indicado en el objeto `Request` y espera el resultado.

`Corba.Request.send_oneway`

Realiza la llamada al método indicado en el objeto `Request` cuando no se espera respuesta, simplemente se envía un datagrama al servidor.

`Corba.Request.send_deferred`

Realiza una llamada al método indicado en el objeto `Request` pero no espera el resultado. Para obtener el resultado, se utilizan los métodos que siguen.

`Corba.Request.poll_response`

Se utiliza para ver si se hay algún mensaje de resultado para el objeto `Request`.

`Corba.Request.get_response`

Si el método anterior nos indica que hay un resultado disponible con este método podemos leerlo.

`Corba.Request.delete`

Nos permite eliminar el objeto `Request` de la memoria de nuestro ordenador.

`Corba.NVList.add_item`

Añade un parámetro a la lista de parámetros.

`Corba.NVList.add_value`

Añade un valor a un parámetro determinado.

`Corba.NVList.get_count`

Obtiene el número de parámetros insertados.

`Corba.NVList.remove`

Permite eliminar un parámetro.

`Corba.NVList.free`

Elimina la lista completamente, llamando sucesivamente a `free_memory`.

`Corba.NVList.free_memory`

Elimina cualquier argumento asociado con la lista.

`Corba.ORB.create_list`

Crea una `NVList` vacía, que deberemos rellenar con los métodos anteriores.

`Corba.ORB.create_operation_list`

Crea una `NVList` y la rellena.

Un ejemplo de cómo sería el código en Java de un programa que llama a una función 'incrementa' podría ser el siguiente:

```
// tenemos el objeto CORBA llamado 'cuenta'
try{
 //Obtenemos la interfaz de Cuenta
 CORBA.InterfaceDef InterfaceCuenta = cuenta.get_interface();
 //obtenemos la descripción completa de la interfaz de Cuenta
 CORBA._InterfaceDef.FullInterfaceDescription intDesc =
Interfacecueta.describe_interface();
 //Vemos si se encuentra el método "incrementa"
 If (intDesc.operations[0].name.compareTo("incrementa")==0){
 //Creamos un objeto Request para el método "incrementa"
 CORBA.Request request=cuenta._request("incrementa");
 //Indicamos que el valor que devuelve es un long
 Request.result().value().form_long(0);
 //Llamamos al método "incrementa"
 Request.invoke();
}
else{
 //El método que buscamos no lo encontramos
```

```
}
}
catch(CORBA.SystemException e){
   //Excepción en el sistema CORBA
}
```

16.2.2.2 DSI

La interfaz de esqueleto dinámico permite recibir llamadas de forma dinámica con objetos CORBA. Este servicio es muy dependiente de la adaptación o mapeo con lenguajes determinados. Enicialmente, está pensado para la construcción de *bridges*, de tal forma que con una sola implementación de la interfaz se puedan gestionar todos los objetos. Veamos un ejemplo en C++ para omniORB2:

```
void MyDynImpl::invoke(CORBA::ServerRequest_ptr request,
CORBA::Environment& env) throw()
{
 try{
 if( strcmp(request->op_name(),"echoString"))
  throw CORBA::BAD_OPERATION(0,CORBA::COMPLETED_NO);
CORBA::NVList_ptr args;
orb->create_list(0,args);
CORBA::Any a;
a.replace(CORBA::_tc_string,0);
args->add_value("",a,CORBA::ARG_INT);

request->params(args);

CORBA::Any& input_any= *(args->item(0)->value());
CORBA::String_var input;
input_any >>= input.out();

CORBA::Any* result= new CORBA::Any();
*rsult<<= CORBA::Any::from_string(input._return(),0);
request->result(result);
}
catch(CORBA::Exception& ex){
 CORBA::Any* v=new CORBA::Any;
::operator<<=(*v,ex);
request->exception(v);
}
}
```

Este ejemplo implementa la invocación dinámica del método `echoString`. La instrucción `if` se puede sustituir con un `case` y, de esta forma, dar servicio a todos los métodos que se desee. En la variable `request`, del tipo `ServerRequest_ptr`, están incuidos todos los parámetros necesarios para la invocación, así como los valores que devolverá. En la variable `env`, del tipo `Environment`, se define el contexto, de ejecución y el objeto al que va dirigido el mensaje, de esta forma inspeccionándolo podremos diferenciar llamadas a objetos diferentes.

16.2.2.3 Interface Repository

El repositorio de interfaces de CORBA (IFR) es esencial para toda aplicación que necesite utilizar las características dinámicas de CORBA, tales como la interfaz de invocación dinámica o el tipo dinámico `Any`. Dicho repositorio mantiene las

definiciones de tipos al igual que las IDL y puede ser consultada y recorrido en tiempo de ejecución.

Aunque se puede interoperar con el repositorio de interfaces como con cualquier otro servidor CORBA, se recomienda el uso de la operación `get_interface()` definida dentro de la interfaz general de `Object`. A continuación, mostramos un sencillo ejemplo en Java que ilustra el uso del repositorio de interfaces con la operación estándar comentada anteriormente:

```
// Java
import org.omg.CORBA.*;
…
org.omg.CORBA.ORB= … //initialize the ORB
org.omg.CORBA.Object obj=… // get object reference somehow
org.omg.CORBA.Object defObj=obj._get_interface_def();
if(defObj==null)
{
 System.err.println("No Interface Repository available");
 System.exit(1);
}
InterfaceDef def=InterfaceDefHelper.narrow(defObj);
org.omg.CORBA.InterfaceDefPackage.FullInterfaceDescription desc=
  def.describe_interface();

int i;
System.out.println("name = " + desc.name);
System.out.println("id = " + desc.id);
System.out.println("defined_in = " + desc.defined_in);
System.out.println("version = " + desc.version);
System.out.println("operations:");
for(i=0;i<desc.operations.length;i++)
{
 System.out.println(i+": "+desc.operations[i].name);
}
```

A pesar de que en el ejemplo anterior utilizamos la operación estándar y el servicio de repositorio del propio `ORB`, para el caso de las llamadas entrantes *indoor* veremos que resulta necesario implementar los objetos servidores directamente, con el fin de construir dinámicamente las consultas que se realicen sobre la máquina.

16.2.2.4 Localización de objetos

En todo sistema distribuido, en algún momento, se debe obtener la primera referencia al primer objeto. De este modo el resto de los objetos pueden ser obtenidos a partir de él, si es necesario. Así, el código típico podría ser:

```
// Java
B b= …. // Get a B object reference somehow
A a = b.getA();
```

Vemos que la referencia a una instancia de `A` ya se obtiene a través de `B`, pero necesitamos algún mecanismo para obtener esa primera referencia a `B`. En nuestro caso, CORBA define una gran cantidad de manera de obtener referencias a objetos.

16.2.2.4.1 Referencias convertidas en string

Se puede convertir una referencia en un `string` a través de las operaciones `object_to_string()` y `string_to_object()`. Una vez obtenida esta cadena podemos utilizarla de varias maneras:

- Directamente, como parámetro al ejecutar el programa distribuido.
- Mediante un fichero donde se ha guardado la referencia previamente convertida en `string`.
- A través de la URL donde se encuentra la referencia de un objeto, por ejemplo, http://www.mywebserver/object.ref. La referencia, de nuevo, viene representada como un `string`.
- Utilizando las referencias como un parámetros `string` en un `applet`.

16.2.2.4.2 Referencias URL a objetos

Antes de la adopción del Servicio de Nombres Interoperable INS (Interoperable Name Service) la única forma de acceder a un objeto era a través de `string`, como se mostró en la sección anterior. Los INS introducen dos nuevas formas de acceder a una referencia utilizando la sintaxis de URL.

corbalock:URL, se denomina a las referencias a objetos que cumplen con la siguiente sintaxis, cuando se utiliza el protocolo `iiop`:

```
corbaloc:[iiop]:[version@] host [:port] / object-key
```

Donde `object-key` es la conversión a string de una referencia. En caso de utilizarse el protocolo `rir`, que proviene de la abreviatura de `resolve_initial_refrence`, se utiliza la sintaxis:

```
corbaloc: rir: [/id]
```

Donde `id` sirve para identificar el nombre del servicio demandado a resolver. Si no se especifica nada, se considera el Servicio de Nombres (*NameService*).

corbaname:URL, proporciona flexibilidad adicional al incorporar el uso del servicio de nombres en la operación `string_to_object`. Ahora se permite que el `object-key` identifique a un nombre dentro de un Servicio de Nombres. Veamos el siguiente ejemplo:

```
corbaname::ns1:5001//NameService#ctx/MyObject
```

Se accederá al servicio de nombres conectado en el puerto 5001 y, una vez que el contexto ha sido resuelto se buscará un enlace a un objeto con nombre `MyObject`.

file:URL y **refili:RUL**, nos proporcionan mecanismos para obtenre una referencia a través de caminos a ficheros, absolutos y relativos respectivamente.

16.2.2.4.3 Servicios de inicio

CORBA permite una forma estándar de localizar determinados objetos especiales que ofrecen servicios tales como los Sevicios de Nombres (*NameService*) o los Servicios de Tratos (*TradingService*). Para ello, solo hay que realizar una llamada a la operación estándar de CORBA `resolve_initial_reference()`. Una vez obtenida una referencia a estos servicios es posible acceder al resto. En el caso de los servicios de nombres, se podrá acceder al resto a través de los nombres simbólicos registrados en el servicio concreto, siempre dentro de un contexto.

Estos servicios deben ser iniciados dentro del sistema donde se ejecuta CORBA con unos nombres y valores predeterminados y que el `ORB` debe conocer para poder realizar correctamente la resolución.

16.2.3 CORBA <- MA -> CORBA

16.2.3.1 Introducción

En esta sección veremos la instanciación de la colaboración de invocaciones *outdoor*. Como se mostró en la sección 15.1.4, gracias al patrón *bridge*, solamente vamos a necesitar refinar la clase `MAExternalSystem` y definir sus métodos. Para ello, utilizaremos servicios de invocación dinámica de CORBA (DII), vistos anteriormente, y en concreto, el objeto `CORBA::Request`.

En la Ilustración 16.3 se muestra un diagrama de clases donde se define una nueva clase, `MAExternalSystemCORBA,` que hereda de `MAExternalSystem` y está asociada a través del atributo `theDII` que, como su nombre sugiere, es un puntero a un objeto `Request` de CORBA y que será nuestro enlace con la plataforma distribuida externa CORBA.

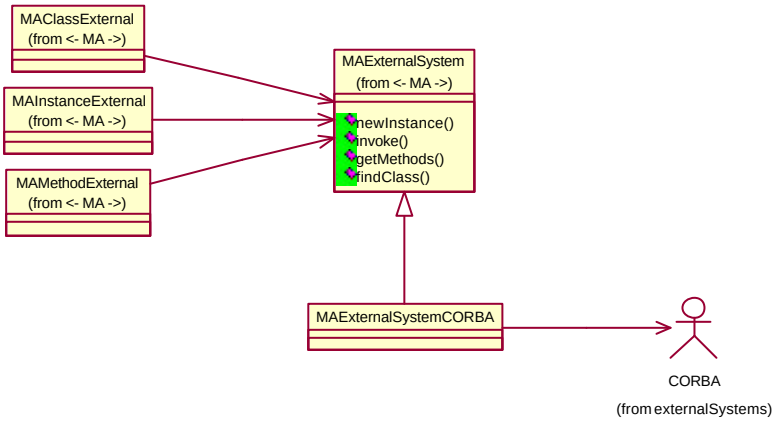


Ilustración 16.3: CORBA <- MA -> CORBA

Comenzaremos viendo cómo localizar las clases externas. Luego contemplaremos cómo podemos obtener su información estructural para construir a partir de dicha información, las clases, instancias y métodos oportunos. De esta forma, todos los objetos de la máquina verán los objetos CORBA como propios a través de los

servicios dinámicos. Posteriormente. Hablaremos de como construir la invocación a los métodos externos CORBA. En realidad, todas estas tareas serán realizadas por los diversos métodos redefinidos de la clase `MAExternalSystem`, a saber, `invoke()`, `newInstance()`, `getMethods()` y `findClass()`.

16.2.3.2 Introspección

Para consultar en tiempo de ejecución de forma dinámica la información necesaria para la construción de las llamadas dinámicas, obteniendo, así, información acerca de los métodos y parámetros necesarios y poder realizar las conversiones de tipos pertinentes, utilizaremos el repositorio de interfaces (RI) así como el conjunto de operaciones que incluye. Como se comentó en las secciones anteriores, se recomienda el uso de la operación propia del `ORB`, `_get_description()`, por encontrarse directamente disponible en el sistema.

16.2.3.3 Localización

De los varios servicios de localización que ofrece CORBA en sus últimas especificaciones, como se citó en la sección 16.2.2.4, vamos a basar nuestro diseño arquitectónico en la técnica basada en URL de nombres de objetos denominada *corbaname*. Tomaremos esta técnica por su sencillez, su alto nivel de abstracción y su parecido al nombrado utilizado en el resto de sistemas, que es basado en nombres. En CORBA para localizar un objeto a través de su nombre, primero debemos comunicarnos con el servicio de nombres, posteriormente, ubicar en qué contexto se encuentra el nombre y, finalmente, buscar en dicho contexto el que deseamos utilizar. De todas estas tareas se encargará la clase `MAExternalSystemCORBA` a través de su método `getClass` principalmente.

Al igual que hicimos con los nombres de objetos DCOM, donde se diferencia la aplicación y la clase, vamos a utilizar el símbolo "_" para diferenciar entre el contexto y el nombre del objeto. De esta forma `TranslatioService_Checkspeller` indicará el objeto `Checkspeller` dentro del contexto `TranslationService`.

En cuanto a la federación de búsqueda, la búsqueda a través de servicio de nombres se adapta perfectamente al modelo mostrado en la Ilustración 15.9.

16.2.3.4 Instanciación

Inicialmente, CORBA está más orientado a instancias e interfaces que a clases. De tal forma que, como ya hemos visto, la localización y descripción se realiza en función de objetos y de interfaces. Así, el criterio de diseño considerado para la instanciación es que todos los envoltorios instancia de CORBA, se considerarán que son clases que a su vez heredan de su propia clase. De este modo la localización en el área de clases de la máquina MA valdrá tanto para las clases como para las instancias.

16.2.3.5 Invocación

Para cada método de la clase externa se creará una instancia de la clase `MAMethodExternal` que como atributo, contiene símplemente el nombre heredado de `MAMethod` y cuyo `invokeInstance` realiza una llamada al método `invoke` de la clase `MAExternalSystemCORBA` pasándole tanto las instancias como los parámetros que

recibe originalmente. Desde este último método es de donde se debe sacar el puntero al objeto `Request` de la instancia sobre la que se desea ejecutar el método, construir la invocación, realizar las diversas conversiones, tanto de los parámetros como de los valores de retorno, realizar la invocación llamando al método `invoke` del objeto `Request` (como se mostró en 16.2.2.1), y gestionar los posibles errores que surjan.

16.2.4 CORBA -> MA <- CORBA

16.2.4.1 Introducción

De nuevo volvemos a tener dos opciones para diseñar esta dirección de la comunicación y uan vez más, volvemos a elegir la opción DSI, ya que CORBA se adapta perfectamente a este perfil. de hecho CORBA dispone de un servicio con identicos nombre y finalidad, el servicio DSI o interfaz de esqueleto dinámica. En la Ilustración 16.4 se muestra la jerarquía de clases básicas donde definimos `MAExternalDispatcherCORBA`, que hereda de `MAExtenalDispatcher` y de la clase activa CORBA DSI que encapsula los objetos de la interfaz `DynamicImplementation` e implementa su método principal `invoke`.

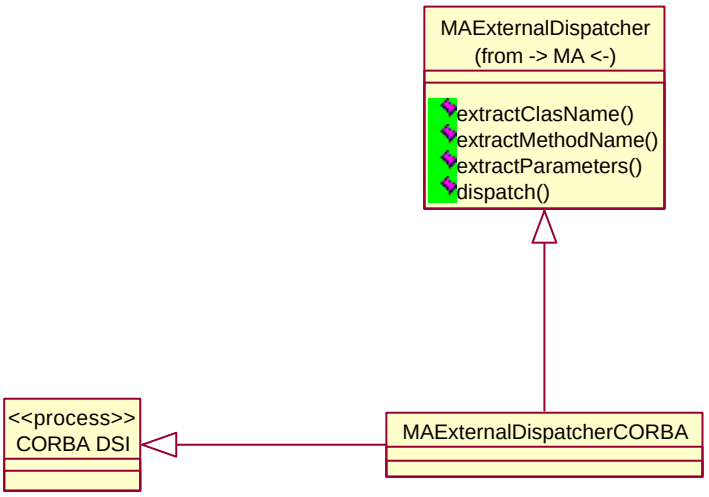


Ilustración 16.4: CORBA -> MA <- CORBA

16.2.4.2 Exposición de la invocación con DSI

Utilizaremos el DSI para interceptar, gestionar y distribuir las llamadas realizadas hacia CORBA desde otros sistemas. De este modo, todos los objetos y clases de MA, tanto primitivas definidas por el usuario o virtuales, pertenecientes a otras plataformas de invocación síncrona remota seran vistos como objetos CORBA nativos. Para ello, utilizaremos la interfaz `DinamicImplementation` y, más concretamente, el método `invoke`. Esta interfaz está pensada para recibir llamadas de forma dinámica con objetos CORBA. Inicialmente, se emplea para la construción de *bridges* entre ORB. Esto significa que con una sola implementación del método `invoke` se pueden gestionar todos los objetos que se deseen a través de la variable `env` de tipo `CORBA::Environment` y los parámetros indicados en `request` del tipo `ServerRequest_ptr`. Todo ello nos va a permitir, como en el caso de DCOM, con una

sola instancia de la clase activa `CORBA DSI`, gestionar todas las peticiones externas de los objetos CORBA.

16.2.4.3 Exposición de la introspección de un repositorio de interfaces

Nuestro diseño de gestión de introspección de clases externas encaja muy bien con el modelo de introspección dinámico de CORBA a través de los repositorios de interfaces (RI). Solamente necesitamos crear una clase `MAInstrospectionCORBA` que herede de `MAExternalInstrospection`, vease la Ilustración 15.8, y que permitirá a los clientes CORBA obtener la descripción de las clases internas de MA, expresadas según la estructura especificada por los RI. De esta forma, se podrán construir llamadas dinámicas bien a otros sistemas externos bien a los propios elementos de la máquina MA.

16.2.4.4 Localización

Como se indicó en la sección anterior, nuestro diseño arquitectónico para CORBA va a basarse, simplemente, en los Servicios de Nombres para localizar objetos. Diseños alternativos pueden considerar las otras opciones de localización e incluso, todas. Al contrario de lo que ocurria con el Registro del Sistema de Windows, en este, caso sí resulta factible la federación de Servicios de Nombres. De tal forma que ahora sí vamos a poder mantener la coherencia de localización con el resto de sistemas externos. Esto significa que si un objeto CORBA desea acceder a otros objetos externos, aunque este no esté en el área de clases de MA podrá hacerlo. Para ello, habrá que implementar la clase activa `MALocatorCORBA` que hereda de `MAExternalLocator`, que actuará como un servicio de nombres a los ojos de los sistemas CORBA externos. Para conseguir la coherencia, bastará con federar este servicio de nombres con otros servicios que se deseen utilizar en el entorno.

16.2.5 Conclusiones

Para las comunicaciones *outdoor*, los servicios DII resultan ideales y todos las colaboraciones de nuestro modelo arquitetónico son fácilmente instanciables para llegar a una arquitectura concreta. El único punto crítico es que nos encontramos en un sistema orientado a objetos e interfaces, y no a instancias, de tal forma que la búsqueda está basada en objetos ya creados. Esto ha sido solucionado considerando cada objeto como una instancia de una clase con el mismo nombre que la instancia y que hereda de una clase con el nombre de su interfaz. De este modo, en el área de clases de la máquina MA está accesible tanto la instancia a través de su clase directa, como la clase relacionada con su interfaz.

Desde el punto de vista *indoor*, los resultados son mejores que en el caso de los sistemas DCOM. Ahora, los servicios de localización pueden ser federados y podemos conseguir sistemas de localización coherentes o completos, con lo que todos los objetos CORBA van a poder acceder a todos los objetos de todas las plataformas conectadas al middlebus.

16.3 Con Servicios Web

16.3.1 Introducción

Como última instanciación del modelo arquitectónico, nos centraremos en los Servicios Web. En este caso veremos que resulta más adecuado a nivel de protocolo de comunicación, son WSDL, UDDI y SOAP. La diferencia de los Servicios Web con respecto a las otras plataformas de distribución consiste en que no existe, en las especificaciones, un concepto de referencia, objeto o estado, aunque se puede hacer una analogía con respecto a los conceptos de clase y servicio. La especificación de Servicios Web contempla una capa superior no estandarizada sobre la cual se pueden definir estados o referencias a objetos y contextos de ejecución, pero siempre con el inconveniente de que no está estandarizado. Esto indica un nivel de inmadurez de esta tecnología.

Como en los casos anteriores, instanciaremos la arquitectura en una dirección y, posteriormente, en la otra. En el primer caso estudiaremos cómo crear invocaciones SOAP y HTTP GET/POST de forma dinámica, cómo inspeccionar los Servicios Web a través de WSDL y cómo localizarlos. Todas estas acciones, a nivel de red, son muy sencillas ya que simplemente es preciso un *parser* XML y con un pequeño cliente web. Para la comunicación *indoor*, necesitaremos implementar varios servidores web y utilizar XML para construir las diversas estructuras de comunicación. Por un lado, necesitaremos un servidor web para recibir las invocaciones SOAP o HTTP GET/POST, otro para recibir solicitudes de consultas WSDL y, por último, un tercer servidor que permitirá localizar elementos dentro del sistema.

Teniendo en cuenta la no existencia de estados y contextos dentro de los Servicios Web, consideraremos que todas las llamadas internas a un servicio crearán una instancia de la clase que representa el servicio. Se invocará su método y, posteriormente, se elimina a la clase. De esta forma, no podremos mantener estados en las llamadas entrantes. Por otro lado, al realizar llamadas externas, consideraremos que todas las variables definidas en la máquina de una determinada clase que representa un servicio externo no contienen datos fijos y todos representarán información y métodos de clase. De este modo las invocaciones serán todas al mismo Servicio Web. Aunque dentro de la máquina sean variables o instancias diferentes de una misma clase, se comportarán como una única. Estas restricciones pueden mejorarse para que no sea así, de tal forma que tengamos estados y contextos, pero nos saldríamos del estándar, por lo que nuestra recomendación es esperar a que madure esta tecnología y, de momento, considerar estas restricciones.

Nos centraremos en el protocolo de comunicación HTTP, si bien los Servicios Web podrán utilizarse con otros protocolos SMTP, TCP/IP, etc.

16.3.2 Servicios Web: SOAP, WSDL, UDDI

16.3.2.1 SOAP

Históricamente, el primer, y más importante protocolo que se definió fue SOAP, que es una simplicación de la invocación a métodos. Utiliza XML para encapsular la información transmitida en los paquetes a nivel de transporte, a modo de IIOP. Esta pareció ser una muy buena idea, sencilla, flexible y de alto nivel. Posteriormente, se definieron nuevos servicios de descripción de operaciones e interfaces a modo de IDL basado en XML. Tuvo varias versiones y nombres hasta que, finalmente, se definió WSDL, que comentaremos en la siguiente sección anterior. Este nuevo protocolo añadió a los Servicios Web nuevas técnicas de invocación, incluso más sencillas que las de SOAP, como, por ejemplo HTTP GET/POST.

SOAP es un protocolo basado en mensajes de texto formados por un sobre (*Envelope*) dentro del cual hay dos partes una cabecera (*Head*) y un cuerpo (*Body*), como se muestra en la Ilustración 16.5.

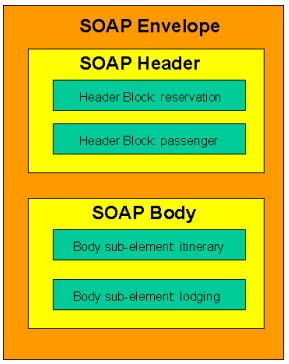


Ilustración 16.5: Estructura SOAP

A continuación, se muestra un sencillo ejemplo de mensaje utilizando el protocolo HTTP 1.1, con el que se pretende invocar un método denominado `GetlasTradePrice`:

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
       <m:GetLastTradePrice xmlns:m="Some-URI">
           <symbol>DIS</symbol>
       </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

El siguiente código muestra la respuesta producida por el proceso que recibió y procesó la invocación anterior:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
       <m:GetLastTradePriceResponse xmlns:m="Some-URI">
           <Price>34.5</Price>
       </m:GetLastTradePriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

16.3.2.2 WSDL

Con estas siglas se denomina al lenguaje de descripción de Servicios Web. Básicamente, es información XML en la que se aporta una descripción de las interfaces que intervienen en determinados servicios junto con los protocolos de invocación que soporta. Inicialmente, se consideró que sería SOAP; pero actualmente, se pueden utilizar otros servicios de invocación, como HTTP GET/POST, SMTP e incluso, IIOP de CORBA. De hecho, hay tecnologías de resolución de la heterogeneidad de servicios definiendo descripciones WSDL con diversos protocolos de invocación, como se comentó en la sección 8.2.

A continuación se muestra un ejemplo modelo donde se identifica la estructura de los documentos WSDL. Como se ve, se trata tan solo de un conjunto de definiciones anidadas, donde se definen seis tipos de elementos:

- `types`, define los tipos de datos usados en el intercambio de mensajes.
- `message`, representa una definición abstracta de los datos que van a ser transmitidos. Un mensaje está formado por varias partes, cada una de las cuales está asociada con algún tipo definido previamente.
- `portType`, es un conjunto de operaciones abstractas. Cada operación hace referencia a un mensaje de entrada (*input*) y un mensaje de salida (*output*).
- `binding`, que especifica protocolos y especificación de formatos de datos concretos para las operaciones y mensajes definidos en un `porType` concreto.
- `port`, especifica una dirección para un enlace (`binding`), de tal forma que queda definido un punto final de comunicación.
- `service`, se utiliza como un conjunto de puertos (`port`) relacionados lógicamente.

Compruebese dicha estructura en la siguiente ilustración:

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>

    <import namespace="uri" location="uri"/>*

    <wsdl:documentation .... /> ?

    <wsdl:types> ?
        <wsdl:documentation .... />?
        <xsd:schema .... />*
        <-- extensibility element --> *
    </wsdl:types>

    <wsdl:message name="nmtoken"> *
            ....
    </wsdl:message>

    <wsdl:portType name="nmtoken">*
            ....
     </wsdl:portType>

    <wsdl:binding name="nmtoken" type="qname">*
            ....
     </wsdl:binding>

    <wsdl:service name="nmtoken"> *
            ....
     </wsdl:service>

    <-- extensibility element --> *

</wsdl:definitions>
```

16.3.2.3 UDDI

Inicialmente, se diseñó un servicio de descubrimiento de Servicios Web denominado DISCO que permita buscar Servicios Web en una determinada computadora. Posteriormente, surgió UDDI como una ampliación y federación de servicios DISCO donde se pueden buscar máquinas que ofrezcan un determinado servicio así como el servicio en sí o una descripción técnica detallada. En [Skon02] se discute detenidamente el paso de DISCO a UDDI.

La especificación UDDI consiste en una API de programación junto con una definición Schema XML de las estructuras de datos y mensajes soportados. Una implementación de una especificación UDDI permite a sus ususarios publicar sus Servicios Web y consultar en busca de otro. Toda implementación UDDI es accesible a través de Servicios Web, de tal forma que todas las operaciones de publicaciones y consultas son definidos en términos de mensajes SOAP. Los servicios UDDI se encuentran federados y comparten su información a través de técnicas de replicación.

Los repositorios UDDI contienen información a cerca de negocios, servicios, enlace a servicios y metainformación para categorizar. La forma de organizar esta información es mediante páginas blancas, páginas amarillas y páginas verdes,de forma similar a los listines telefónicos.

- Las **págians blancas** incluyen información de los negocios, direcciones y datos de contacto.
- Las **páginas amarillas** incluyen categorías basadas en taxonomías estándar.
- Las **páginas verdes** incluyen especificaciones técnicas y referencias.

Los tipos principales de información contenidos en un registro UDDI son: negocios, servicios, plantillas de enlace y tModel. Cada registro de negocios expone información básica sobre el servicio e información técnica relacionada con él. Cada servicio consiste en información técnica y de clasificación, además de exponer las plantillas de enlace que exponen como acceder a los Servicios Web descritos.

A la hora de comprobar la compatibilidad entre sistemas, no resulta suficiente con los servicios y las plantillas de enlace, es necesario conocer las interfaces contractuales y su comportamiento asociado. Un tModel es, esencialmente, una técnica para describir estos elementos, como las categorías de componentes en DCOM. Se utilizan para comprobar si determinados servicios son compatibles con los sistemas de un cliente determinado.

La **API de programación** UDDI tiene dos grandes grupos de operaciones que se invocan como Servicios Web. Por un lado, los de publicación y por otro los de consulta. Veamos un ejemplo de una consulta de los servicios proporcionados por un determinado negocio a través de la invocación al método `find_service`:

```
<?xml version='1.0' encoding='utf-8'?>
<s:Envelope
  xmlns:s='http://schemas.xmlsoap.org/soap/envelope/'>
  <s:Body>
    <find_service generic='1.0'
      xmlns='urn:uddi-org:api'
      businessKey='0076B468-EB27-42E5-AC09-9955CFF462A3'>
      <name>UDDI Web Services</name>
    </find_service>
  </s:Body>
</s:Envelope>
```

16.3.3 Servicios Web <- MA -> Servicios Web

16.3.3.1 Introducción

Como en los casos anteriores, en esta sección vamos a introducir todos los Servicios Web como clases dentro de nuestro sistema, con la salvedad de que hay que considerar las restricciones comentadas anteriormente de no estado y no contexto. Básicamente, diseñaremos la clase que crearemos aquí `MAExternalSystemWebServices`, que hereda de `MAExternalSystem`. Cáda método consistirá en un cliente HTTP que construirá la llamada XML correspondiente al protocolo indicado, véase la Ilustración 16.6.

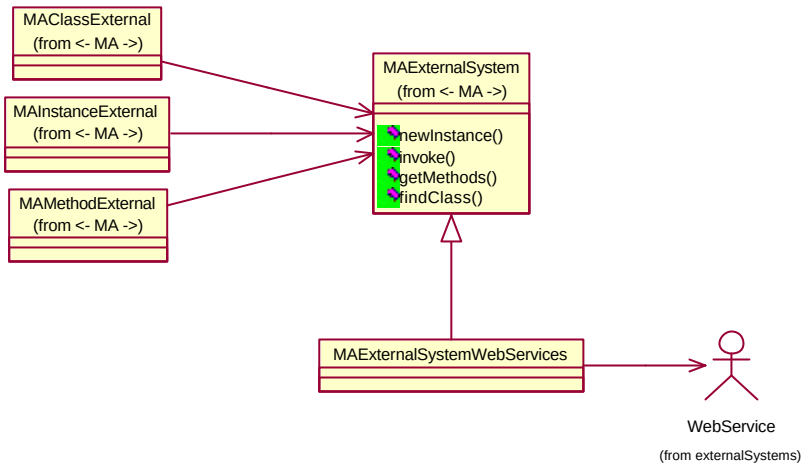


Ilustración 16.6: Servicios Web <- MA -> Servicios Web

16.3.3.2 Realización de las consultas con SOAP o HTTP GET/POST

Implementaremos el método `invoke` dentro de la clase `MAExternalSystemWebServices`. Este método tomará los parámetros y construirá una invocación XML SOAP o HTTP GET/POST en función de la descripción WSDL del servicio al que pertenezca la instancia. Debido a que no existen instancias dentro de lo Servicios Web, todos los objetos de `MAInstance` contendrán solamente la referencia a su clase, que identificará el Servicio Web. Por lo demás, la invocación será realizada por el método `invoke()`, tras haber adaptado y convertído los parámetros, y devolverá el resultado, una vez recibida la contestación a la llamada `remote.`

16.3.3.3 Introspección con WSDL

Una vez localizado el servidor WSDL del servicio que queremos inspeccionar, la consulta sobre la estructura de datos será tan sencilla como el analisis de un archivo XML. Si bien, habrá que asociar a la instancia del sistema externo `MAExternalSystemWebService` el protocolo de invocación. Por lo demás, los servicios WSDL se adaptan muy bien a nuestro modelo arquitéctónico.

16.3.3.4 Localización con UDDI

El registro de localización UDDI permite localizar diversas empresas proveedoras de Servicios Web y consultar los servicios que ofrecen. En nuestro caso, todas estas tareas deberán ser realizadas de forma automática. A la hora de buscar una determinada “clase” (léase Servicio Web), habrá que localizar de entre todas las empresas, cuáles ofrecen Servicios Web con el nombre de la clase. Para facilitar la búsqueda, y siguiendo con los criterios establecidos en las otra paltaformas, el nombre de la clase vendrá separada por ‘_’ donde la primera parte indica la categoria UDDI donde se encuentra el Servicio Web y la segunda el nombre del Servicio Web. Una vez localizadas las clases, habrá que seguir algún criterio de selección que se deja a elección

del diseñador del sistema detallado. Una vez seleccionada una de ellas, se realizará una invocación de descripción pormenorizada como la mostrada a continuación:

```
<?xml version='1.0' encoding='utf-8'?>
<s:Envelope
  xmlns:s='http://schemas.xmlsoap.org/soap/envelope/'>
  <s:Body>
    <get_serviceDetail generic='1.0'
      xmlns='urn:uddi-org:api'>
      <serviceKey>D2BC296A-723B-4C45-9ED4-494F9E53F1D1
      </serviceKey>
    </get_serviceDetail>
  </s:Body>
</s:Envelope>
```

Como se muestra en el ejemplo anterior, una vez identificado el Servicio Web hay que solicitar su clave (*serviceKey*) para, posteriormente, en función de ella solicitar la descripción detallada.

16.3.4 Servicios Web -> MA <- Servicios Web

16.3.4.1 Introducción

En esta sección pretendemos exponer todos los elementos de la máquina como Servicios Web, siguiendo con las restricciones de no estado y no contexto. Continuando con la analogía con los casos anteriores, esta vez necesitamos definir servidores HTTP, uno para cada servicio requerido: invocación, introspección y localización. En este caso de estudio se ha tomado el criterio de diseño de exposición de elementos de la máquina a nivel de protocolo de transporte, tal y como se describe en la sección 15.2.2.2.

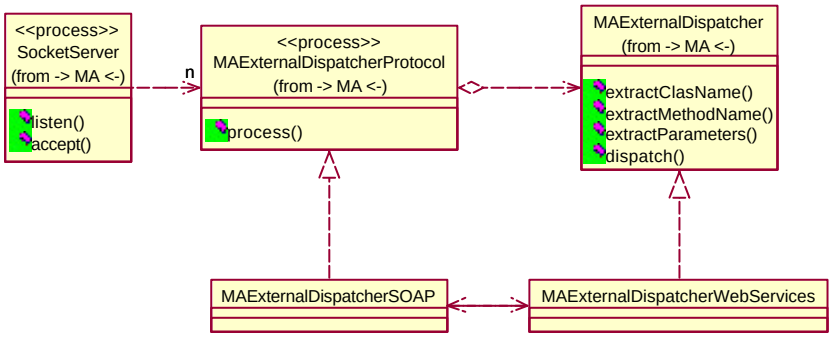


Ilustración 16.7: Servicios Web -> MA <- Servicios Web

En la Ilustración 16.7 se muestra la estructura arquitectónica estática de sistema de interceptación de llamadas SOAP.

16.3.4.2 Interceptacion de las llamadas SOAP.

Ahora vamos a necesitar una clase activa que actúe de intérprete del protocolo concreto, en nuestro caso SOAP. Esta se denomina `MAExternalDispatcherSOAP` y consiste, básicamente, en un servidor web que procesa las peticiones enviadas desde los clientes de Servicios Web. Habría que definir una nueva clase para cada protocolo de enlace que deseemos que ofrezcan nuestros Servicios Web y será necesario indicar a

la clase `MAInstrospectionWebServices`, que veremos en detalle más adelante, que genere en la información del WSDL los enlaces con los protocolos indicados.

Una vez procesada la petición a nivel de protocolo de enlace, la clase `MAExternalDispatcherWebService` que, tal y como sucedía en las plataformas anteriores, DCOM o CORBA, hereda de `MAExternalDispatcher`, se encargará de construir la invocación a la clase interna apropiada y adaptar los datos a los modelos internos de la máquina MA. En este caso, al ser una plataforma sin estados y sin contextos, las instancias no tendrán asociado ningún valor y se realizarán las llamadas directamente, como si de métodos de clase se tratase.

16.3.4.3 Exposición del sistema a través de WSDL

Se expondrá un servidor web que aprovechando las capacidades reflectivas de la máquina construirá los mensajes WSDL descriptivos del servicio inspeccionado.

El diseño mostrado en la sección 15.2.3 encaja muy bien en el modelo de lenguaje de descripción de Servicios Web WSDL. Ahora crearemos una clase `MAInstrospectionWebServices`, que heredará de `MAExtenalIntrospection` y que permitirá a los clientes de Servicios Web obtener la información descritiva necesaria para realizar invocaciones dinámicas a los elementos existentes dentro del middlebus. Lo único a destacar es que la información referente al protocolo de enlace utilizado vendrá determinada por cuántos de estos protocolos van a ser utilizados. Esta será una decisión de diseño a la hora de construir el modelo arquitectónico del sistema. En nuestro caso hemos considerado que solamente vamos a utilizar el protocolo SOAP, con lo que, en la descripción que genere la clase implicada, deberá, además, añadir que el protocolo utilizado es este.

16.3.4.4 Exposición de la localización a través de UDDI

En este apartado deberemos crear un servidor UDDI que permita ser federado y, por consiguiente, admita replicaciones de otros servidores UDDI. Este será implementado en la clase `MALocatorWebService` que hereda de `MAExternalLocator`. Al igual que en el caso de CORBA, y a diferencia de DCOM, ahora si podremos mantener la coherencia de localización y todos los clientes de Servicios Web podrán realizar consultas de búsqueda de todos los elementos de las plataformas externas conectadas al middlebus así como de las clases internas de la máquina MA, que serán vistas como Servicios Web. Habrá que definir algún parámetro de configuración para indicar la empresa en la que se pueden localizar los Servicios Web del middlebus.

16.3.5 Conclusiones

Los Servicios Web se adaptan bastante bien al modelo arquitectónico propuesto y nos permiten realizar instanciaciones del mismo para realizar arquitecturas concretas de forma bastante cómoda. La única pecualiaridad es la de que no están contemplados de forma estandar los estados, por lo que no se puede utilizar el concepto de instancia de objeto. Nuestra solución es ver y ofrecer los Servicios Web como clases sin instancias.

Se han tenido que tomar criterios de diseño como qué protocolos de enlace vamos a utilizar. Hemos decidido utilizar SOAP. Con respecto al nombrado interno de las clases hemos decidido que los nombres van a estar formados por la categoría UDDI y el nombre del servicio separado por '_'. En caso de encontrar en un servidor UDDI más de una ocurrencia del servicio que estamos buscando, se deja abierta al diseñador específico el criterio de elección y su flexibilidad.

En cuanto a las comunicaciones *indoor*, hemos elegido el desarrollo de recepción de invocaciones a nivel de protocolo de transporte, ya que se adapta muy bien a los Servicios Web.

Por lo demás, tanto en las comunicaciones *indoor* como *outdoor*, nuestro modelo se adapta muy bien y las arquitecturas resultantes son muy parecidas a las de los sistemas anteriormente estudiados.

Capítulo 17:
Implementación de un prototipo

En la sección anterior, estudiamos casos concretos a nivel de diseño de instanciación del modelo arquitectónico propuesto en esta tesis. Vimos la adaptabilidad del modelo a las diversas plataformas y qué restricciones impone cada una de ellas, así como las soluciones consideradas para solventar tales restricciones. Hemos visto que nuestro modelo se adapta de forma sencilla y homogénea a todas las plataformas estudiadas, simplificando, en gran medida, la construcción y adaptación de nuevas plataformas al bus y manteniendo la transparencia, homogeneidad y resto de requisitos deseados. En esta sección proponemos un caso concreto de implementación. Hemos decidido utilizar la máquina Oviedo3 como máquina abstracta sobre la que crear nuestro sistema prototípico. Pretendemos crear comunicación unidireccional entre Servicios Web y objetos DCOM. El diseño queda prácticamente adaptado a la nueva máquina, comentaremos su arquitectura y observaremos su similitud con la de nuestra máquina básica MA, si bien Oviedo3 tiene un nivel de abstracción de código más bajo y además, aporta su propio control concurrente independiente del sistema operativo subyacente y un completo conjunto de servicios sobre los que se basa el sistema operativo SO4.

Los pasos necesarios para construir nuestro prototipo será, inicialmente, una especificación del contexto en el cual definiremos arquitectónicamente la máquina abstracta de Oviedo3. Posteriormente, veremos como hemos introducido la reflectividad dentro de la máquina siguiendo las indicaciones vistas en la sección 13.2. Seguidamente, mostraremos las dos implementaciones, una de Oviedo3 hacia DCOM y la otra de Servicios Web hacia Oviedo3. Por último, expondremos un ejemplo sencillo de interacción entre los Servicios Web con DCOM atravesando nuestro sistema conectado en Oviedo3.

Para nuestro caso concreto, nos hemos basado en trabajos anteriores realizados sobre máquinas abstractas y hemos decidido utilizar la máquina abstracta carbayonia para construir nuestro sistema por varias razones:

- Adaptación continua al paradigma de orientación a objetos pues se trata de una máquina orientada a objetos desde sus orígenes.

- Robustez y calidad de la máquina Carabayonia al haber sido el pilar fundamental para el sistema integral orientado a objetos Oviedo3, que ha servido de soporte para la realización de numerosas tesis doctorales.

- Alto nivel de abstracción, al no ser una máquina de pila como la máquina de Java o la de Smalltalk, que nos permitirá centrarnos en conceptos de más alto nivel, como hilos de ejecución, clases, herencia o polimorfismo.

- Similaridad con la máquina abstracta básica. Su arquitectura guarda un gran parecido con la máquina abstracta básica general utilizada para realizar la descripción arquitectónica en esta tesis, con lo que la adaptación de la misma será mucho más sencilla que si se utilizan máquinas menos similares.

No obstante, se puede utilizar cualquier otra máquina abstracta, realizando las adaptaciones oportunas. De hecho, la más flexible de todas sería Nitro ya que incorpora la reflectividad estructural y de comportamiento de base.

17.1 Estructura de la máquina orientada a objetos Carbayonia

Para describir la estructura de Carbayonia vamos a mostrar una Vista arquitectónica estructural, donde se comentarán los elementos más importantes que participan en ella. Posteriormente, describiremos la jerarquía de clases `TClass`, que resulta ser la más importante de la máquina, donde podremos apreciar las similitudes con nuestra máquina general básica. Por último, se describirá su comportamiento a través de la Vista arquitectónica dinámica, formada por los diagramas de colaboración más representativos de Carabayonia.

17.1.1 Vista Arquitectónica Estructural

En la Ilustración 17.1 se muestra el diagrama de clases de las clases más representativas existentes en la máquina abstracta así como las relaciones entre ellas. A continuación, se describe y comenta cada una de dichas clases.

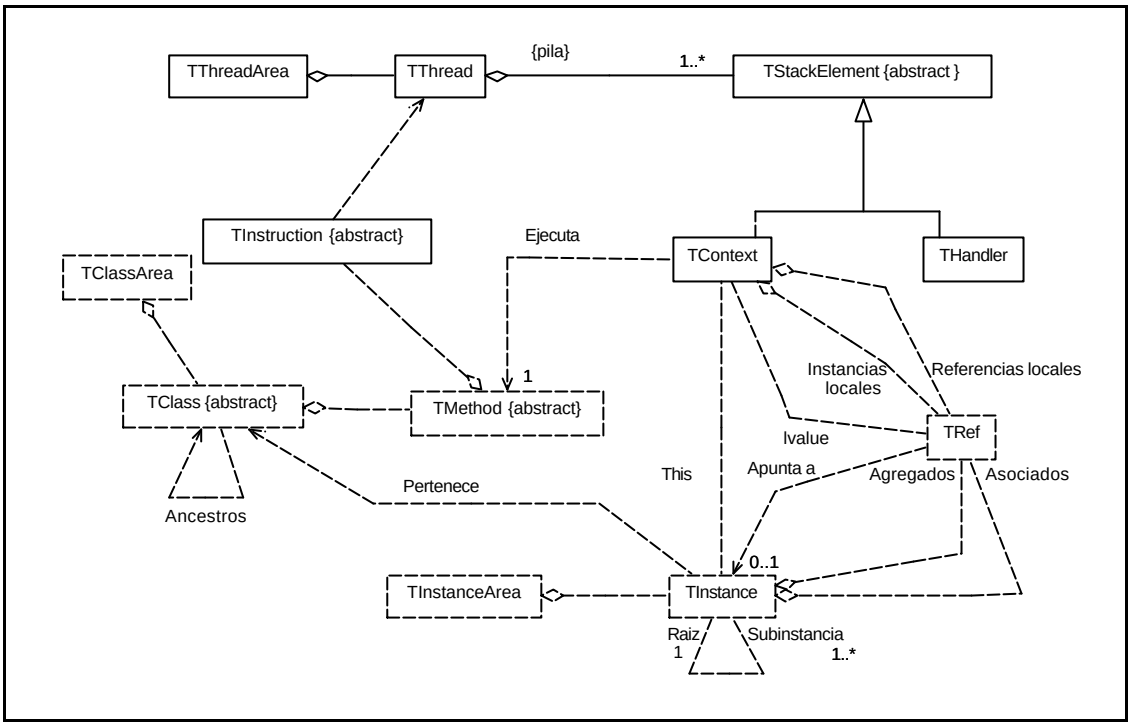


Ilustración 17.1: Vista Arquitectónica Estructural de Carbayonia

17.1.1.1 TClass

`TClass` es una clase abstracta que representa a una clase en tiempo de ejecución. Mediante su derivación podrá ser tanto una clase de usuario (definida mediante el lenguaje) como una clase primitiva. De esta manera, permite que el resto de las clases del simulador puedan tratar de forma homogénea a las clases de usuario y a las primitivas. Una clase tiene una o más clases ancestro (o clases base inmediatas) y uno o más métodos o mensajes. Todas las clases tienen un nombre que las identifica de forma única.

17.1.1.2 TClassArea

`TClassArea` es una clase concreta que representa al área de clases de la máquina. Contiene `P` o más clases, siendo `P` el número de clases primitivas que siempre deben estar en ella. Es la responsable de la introducción de nuevas clases en el sistema y de mantener la información necesaria sobre ellas. Cualquier elemento que necesita información de una clase se lo solicitará al área de clases, que funcionará como intermediario.

17.1.1.3 TMethod

Esta clase abstracta representa un método o mensaje asociado a una clase. Mediante derivación podrá concretarse tanto en un método de usuario como en un método primitivo (es decir, perteneciente a una clase primitiva). De esta manera permite que el resto de las clases del simulador puedan tratar de forma homogénea a los métodos de usuario y a los primitivos. Un método tiene cero o más instrucciones. Posee un nombre único que lo identifica dentro de su clase.

17.1.1.4 TInstruction

Clase abstracta que representa a una instrucción de las disponibles en el repertorio de instrucciones de la máquina abstracta. De aquí derivarán todas las instrucciones de Carbayonia. Esta clase permite que se puedan introducir nuevas instrucciones en la máquina abstracta sin afectar a ninguna otra clase del simulador ya que todas ellas tratan con esta clase base y no con las implementaciones concretas de cada instrucción. De la misma forma, se pueden eliminar o modificar las instrucciones existentes. Como se ve en el diagrama de clases, las instrucciones actúan sobre un `thread` ya que cambian el estado de aquel sobre el que se ejecutan.

17.1.1.5 TInstance

La clase abstracta `TInstance` representa a una instancia en tiempo de ejecución. Una instancia pertenece a una clase y tiene cero o más referencias a instancias agregadas o asociadas.

Además tiene una relación con cero o más instancias de las que es la raíz. Cuando se crea una nueva instancia de una clase (instancia *raíz*), se crean también instancias de todas las clases de las que deriva (*subinstancias*). Estas subinstancias, a su vez, posiblemente tendrán que crear sus propia subinstancias. La raíz de estas nuevas subinstancias seguirá siendo la raíz inicial y no de la subinstancia que las generó.

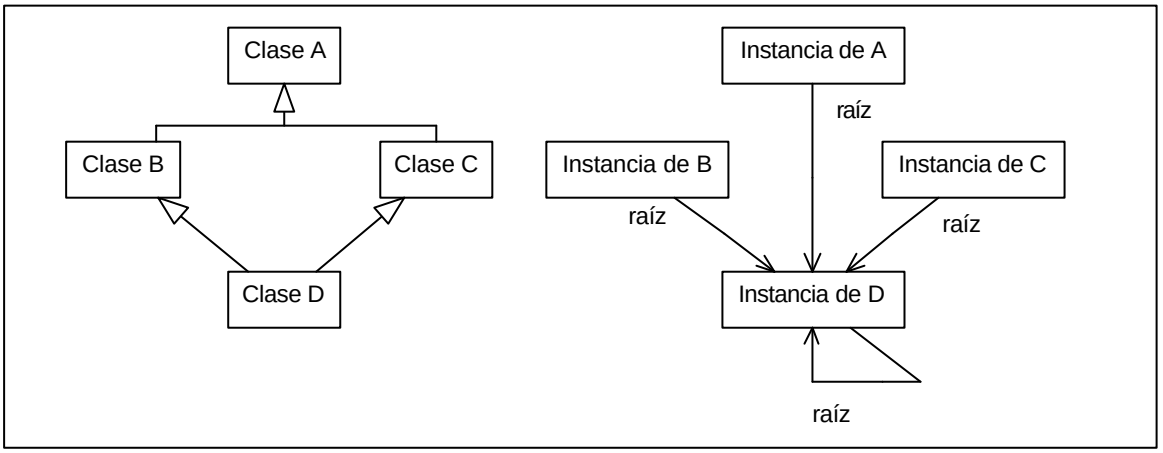


Ilustración 17.2: Representación interna de la herencia

17.1.1.6 TInstanceArea

`TInstanceArea` es una clase concreta que representa el área de instancias de la máquina abstracta. Su comportamiento es el equivalente a la labor que realiza el área de clases con las clases.

17.1.1.7 TRef

`TRef` es una clase concreta que representa una referencia. Una referencia tiene un nombre, un tipo (clase) y, opcionalmente, una instancia asociada.

17.1.1.8 TThread

Un *thread* es una clase concreta que representa a un hilo de ejecución de la máquina abstracta. Cada *thread* contiene una pila de `TStackElement` (elementos de la pila) que define su estado interno. Los hilos se crean bien cuando se inicia la ejecución del programa bien cuando se envía un mensaje. Se destruyen bien cuando se le acaba la pila de llamadas bien cuando se produce una excepción que no es atrapada. Tiene un atributo de estado que indica la situación actual del *thread*. Este atributo puede tomar los valores:

- *Ready*, si el thread está disponible para ejecutarse.
- *Killed*, si el thread terminó normalmente al retroceder mediante exit hasta el final de la pila.
- *UnHandled*, si el thread finalizó por no haberse producido una excepción y no haber sido atrapada mediante un *handler*.

17.1.1.9 TThreadArea

`TThreadArea` es el contenedor de los *threads* de la máquina abstracta. Su comportamiento es el equivalente a la labor que realiza el área de clases con las clases.

17.1.1.10 TStackElement

El `TStackElement` es una clase abstracta que representa los elementos que pueden estar en la pila de un thread. Estos dos tipos de elementos son los handler y los contextos.

17.1.1.11 THandler

Un *handler* es un `TStackElement` concreto que registra la activación de un handler de excepciones realizado mediante la instrucción *handler*. Este objeto se crea cuando se ejecuta la instrucción y se destruye cuando se produce una excepción y es el último handler en haberse introducido en la pila. También se destruye cuando se produce un exit y su posición en la pila es posterior al último contexto introducido, es decir, que el handler se sale de ámbito.

17.1.1.12 TContext

Un `TContext` es una `TStackElement` concreto que define el entorno en que se ejecuta un determinado método. Un contexto está asociado con:

- Un método, sobre el que se ejecuta.
- Una instancia, sobre la que se invocó el método.
- Cero o más referencias, que apuntarán a los parámetros y las referencias locales.
- Cero o más referencias, que apuntarán a las instancias locales.

- Una referencia en la cual es donde deberá dejar el valor de retorno al salir del contexto. Es decir, una referencia sobre la que se copiará el valor de la referencia del sistema `RR`.

Un `TContext` se crea cuando se ejecuta una instrucción de invocación a un método o mensaje sobre una instancia. Se destruye cuando se ejecuta la instrucción *exit* y es el último contexto que se ha introducido en la pila del *thread*. También se destruye cuando se produce una excepción y no hay ningún *handler* posterior a él en la pila. En este caso libera todas sus instancias locales.

17.1.2 Jerarquía de TClass

Una `TClass` es una clase abstracta que define las operaciones que debe ofrecer una clase del simulador en tiempo de ejecución. La funcionalidad de cada una de estas operaciones tienen que darla cada una de sus clases derivadas de acuerdo con sus características.

Las clases concretas derivadas de `TClass`, y, por tanto, las que realmente responden a los mensajes enviados a la misma se representan en las Ilustración 17.3.

Aunque puede que cada una redefina más de una función de la clase base `TClass`, las dos principales misiones que tienen encomendadas los diferentes tipos de clases son:

a) Redefinir `TClass::newInstance` para crear la instancia adecuada cuando les llegue la petición de crear una nueva instancia.

De esta manera se libera al área de instancias de saber las interioridades de cada uno de los tipos de instancias que puede haber para crearlos correctamente. Son los derivados de `TClass` los que conocen la estructura de sus instancias ya que, al fin y al cabo, una clase es la descripción de la estructura común de un grupo de objetos (sus instancias) y dichos derivados son los únicos que necesitan conocer estos detalles.

Así, el área de instancias pasa el mensaje de crear una nueva instancia a la clase solicitada y, debido al polimorfismo, unas veces se conseguirá una instancia de usuario, otras un integer, un bool, etc. cosa que solo sabe la clase a la que se invoca.

La ventaja de este reparto de responsabilidades es que se pueden crear fácilmente nuevos tipos de clases primitivas sin afectar al resto de la arquitectura. Simplemente hay que derivar de `TClass` y redefinir `TClass::newInstance` para devolver el tipo de instancia asociado a la nueva clase.

b) Redefinir `TClass::findMethod` para indicar si la clase tiene disponible un método con un determinado nombre.

De esta manera, una clase primitiva puede añadir o quitar métodos de la definición de su clase de forma transparente al resto de la arquitectura, por lo que se facilitan las posteriores evoluciones del lenguaje. Este método es el único que necesita estar al corriente de los distintos métodos que tiene una clase.

Para las clases de usuario su función sería comprobar la información que almacena sobre la clase y verificar si se realizó la definición de un método con ese nombre.

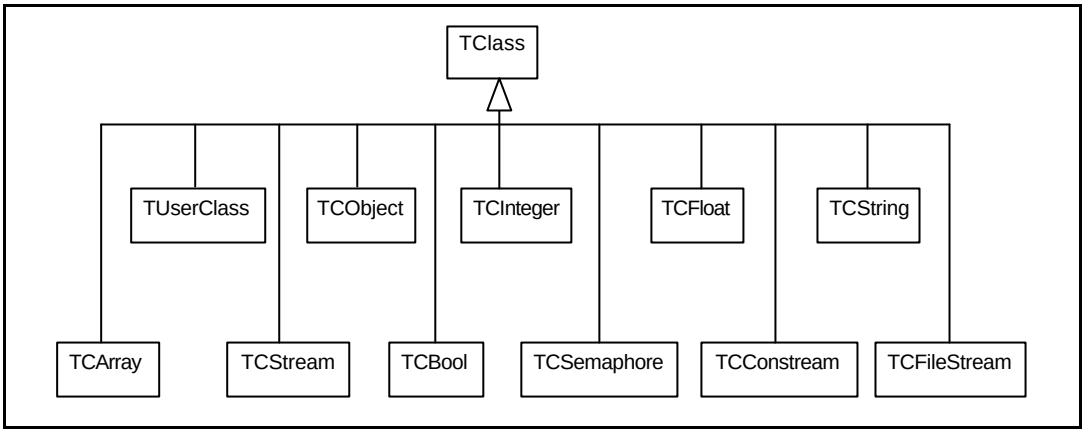


Ilustración 17.3: Jerarquía de TClass (clases básicas)

17.1.2.1 TBasicClass

Clase abstracta que representa a toda clase que tiene como único antecesor el `TCObject` exceptuando `TUserClass`. Su única utilidad es agrupar todas las funciones comunes de este tipo de clases.

17.1.2.2 TUserClass

Representa a una clase de usuario. Encapsula las estructuras de datos que representan la definición de una clase dada leída de un fichero objeto. Por lo tanto, guarda cuántos y cuáles son sus agregados, asociados, nombres, signaturas de sus métodos, etc.

Cada una de las clases de usuario será una instancia de esta clase. De tal manera que, si en un programa se definen dos clases `A` y `B`, en tiempo de ejecución habrá dos instancias de `TUserClass`, cada una con la definición de la clase en el formato adecuado para su consulta.

Cuando se le envía la petición de crear una nueva instancia, devuelve un puntero a una `TUserInstance`, la cual, a su vez, representa una instancia de una clase de usuario.

En la petición de devolución de la dirección de un determinado método (`TClass::findMethod`) contrasta, el nombre del mismo con sus estructuras de datos, para comprobar si está definido en él o en las clases de sus ancestros.

17.1.3 Vista arquitectónica dinámica

En este apartado se muestran los escenarios más importantes de interacción entre las clases del simulador.

17.1.3.1 Creación de una clase

El primer escenario a mostrar es la introducción de una clase en el área de clases. Esta petición puede nacer bien de una petición externa desde el programa principal (para cargar las clases iniciales del programa) bien de la instrucción `newClass` de la máquina abstracta (Ilustración 17.4) bien del método `AddNewClass` de la clase `ClassArea` del módulo de reflectividad.

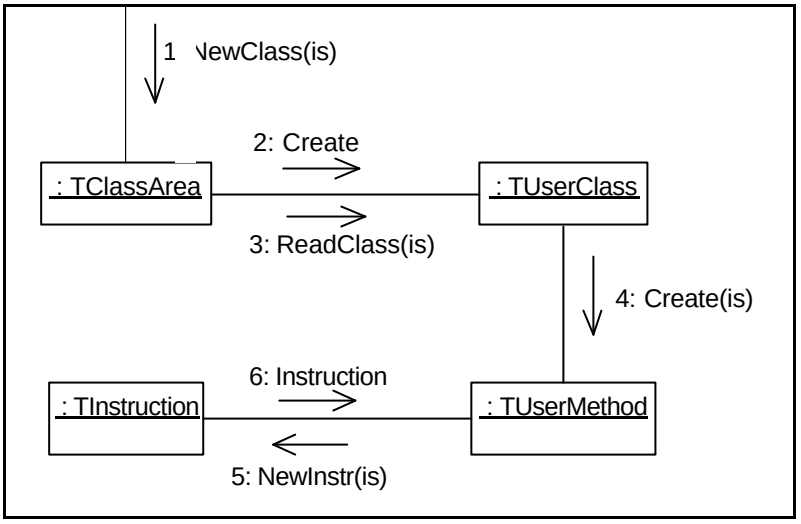


Ilustración 17.4: Creación de una clase

La petición llega al área de clases identificando un *stream* del que leer. Este crea una instancia de `TUserClass` la cual será la encargada de representar a la nueva clase.

Una vez creada la clase de usuario, el área de instancias le indica que lea la clase del *stream* mediante `readClass`. Como parte del proceso de lectura de la definición la instancia de `TUserClass` va creando una instancia de `TUserMethod` por cada método o mensaje que pertenezca a la clase, pasándole como parámetro la posición actual del stream para que sea el mismo método el que lea su definición del fichero.

A su vez, el `TUserMethod`, como parte del proceso de su creación, por cada instrucción que encuentre solicita a la función estática `TInstruction::newInstr` que le cree y le devuelva una instrucción del tipo que se encuentra en el fichero.

La razón de que no sea el `TUserMethod` el que cree las instrucciones es porque no sabe qué instrucciones hay ni cuáles son sus operandos (ni necesita saberlo). Dicho conocimiento está centralizado en la clase `TInstruction` por lo que el añadir o eliminar instrucciones no afecta al resto de la aplicación. `NewInstr` crea una instancia de la clase derivada de `TInstruction` adecuada y le pasa el stream para que sea esta, a su vez, la que lea sus operandos.

17.1.3.2 Creación de una instancia de usuario

En la Ilustración 17.5 se muestra un diagrama de objetos que representa el proceso de creación de una instancia de usuario, que es un caso más general que la creación de una instancia de un tipo primitivo.

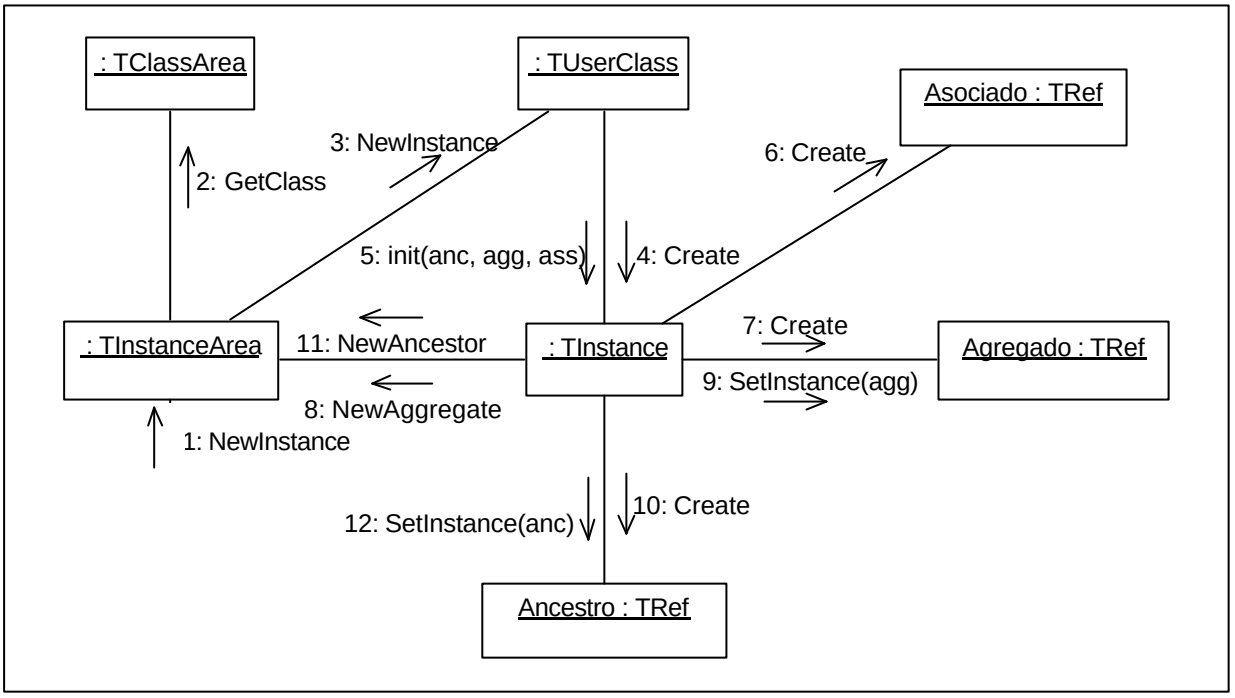


Ilustración 17.5: Creación de una instancia de usuario

Todo comienza con la petición de una nueva instancia de una determinada clase al área de instancias. Lo primero que hace el área de instancias es localizarla en el área de clases para poder transmitirla la petición (flecha 3).

Si esta clase hubiese sido una clase primitiva, crearía una instancia de su tipo y el escenario finalizaría. Sin embargo, dado que se supone que es una clase de usuario (y por tanto el mensaje newInstance que se recibe es el de la versión redefinida de `TClass::newInstance`) el proceso continúa.

Lo primero que hace la `TUserClass` es crear una `TUserInstance` y a continuación le envía las estructuras de datos necesarias para informar a la nueva instancia cual es su composición (agregado, asociados y ancestros).

Con esta información la `TUserInstance` realiza los siguientes pasos:

- Crea una referencia, envía una petición de crear un nuevo agregado al área de instancias y a continuación asigna el nuevo objeto agregado a la referencia (pasos 6, 7 y 8). Este proceso se repite tantas veces como agregados tenga la instancia.
- Crea tantas referencias libres como asociados tenga las instancias (paso 9).

- Crea una referencia, envía una petición de crear un nuevo ancestro al área de instancias y a continuación asigna el nuevo objeto a la referencia (pasos 10, 11 y 12). Este proceso se repite tantas veces como ancestros directos tenga la instancia.

Finalmente se van devolviendo hacia atrás la dirección de la instancia de usuario (paso 3) y el identificador de éste (paso 1) hasta el cliente que inicio la petición.

Nótese la recursividad implícita de este proceso, ya que cada petición de crear un nuevo agregado o un nuevo ancestro generó la misma secuencia de pasos aunque sobre otros participantes.

17.1.3.3 Ejecución de una instrucción

El siguiente escenario es el de la ejecución de una instrucción (Ilustración 17.6).

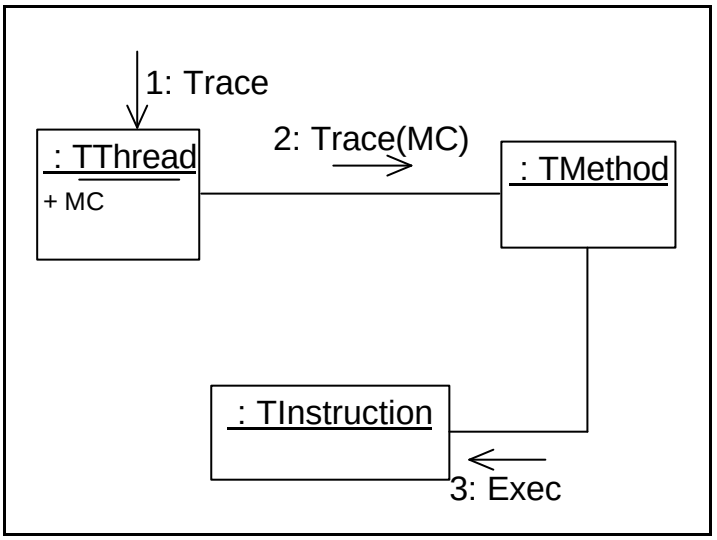


Ilustración 17.6: Ejecución de una Instrucción

El thread recibe una petición de que ejecute la instrucción siguiente del contexto actual.

Éste recupera el `MC` actual (contador de método) el cual indica cuál es el número de la instrucción siguiente a ejecutar dentro del método del contexto actual.

Se indica al método que ejecute dicha instrucción y éste se limita a retransmitirle la petición a la instrucción que ocupe dicho lugar mediante la función exec. Este objeto será una instancia de una de las clases derivadas de `TInstruction` que corresponde con una de las instrucciones de la máquina abstracta. Por tanto a partir de este momento la secuencia de llamadas del escenario depende de cual fuera dicha instrucción (ya que debido al polimorfismo la llamada a exec puede acabar en distintos sitios).

Como ejemplo se mostrarán dos escenarios que sirven para dar una idea del comportamiento de todos los demás: la ejecución de una llamada a un método y la ejecución de una instrucción de salto condicional.

17.1.3.4 Llamada a un método

Se plantea ahora el otro escenario en el que la `TInstruction` a la que le llega el mensaje exec sea esta vez una instancia de `TInstrCall`, la cual es la encargada de realizar las llamadas a los métodos (Ilustración 17.7).

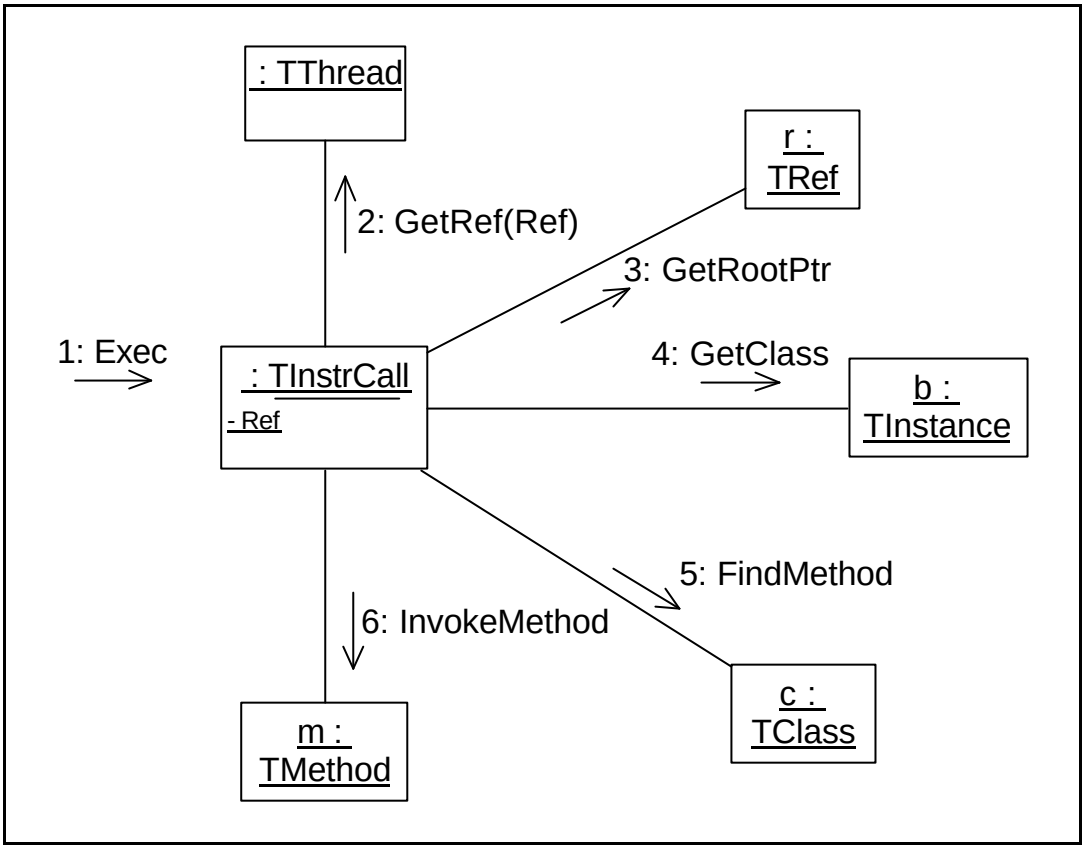


Ilustración 17.7: Ejecución de la instrucción Call

Cuando la `TInstrCall` recibe la petición de ejecutarse solicita al thread que le dé acceso a la referencia sobre la que se ha realizado la invocación.

Una vez obtenida la referencia le solicitamos que nos dé la dirección de la instancia raíz de la instancia a la que apunta. La razón de solicitar la instancia raíz es por que necesitamos mirar primero en las clases derivadas por si la clase de la instancia raíz ha redefinido el método solicitado.

Se le solicita a la instancia que identifique la clase a la que pertenece para de esta manera solicitarla que nos de un puntero al método requerido (paso 5).

Finalmente le decimos al método que un thread le ha invocado mediante el mensaje `invokeMethod`.

De nuevo, dependiendo de a qué clase concreta derivada de `TMethod` corresponda el método requerido se producirán dos subescenarios:

17.1.3.5 Invocación de un método de usuario

El primero de ellos es si el método corresponde a una instancia de `TUserMethod` (definido por el usuario). Entonces la situación corresponde al diagrama de la Ilustración 17.8.

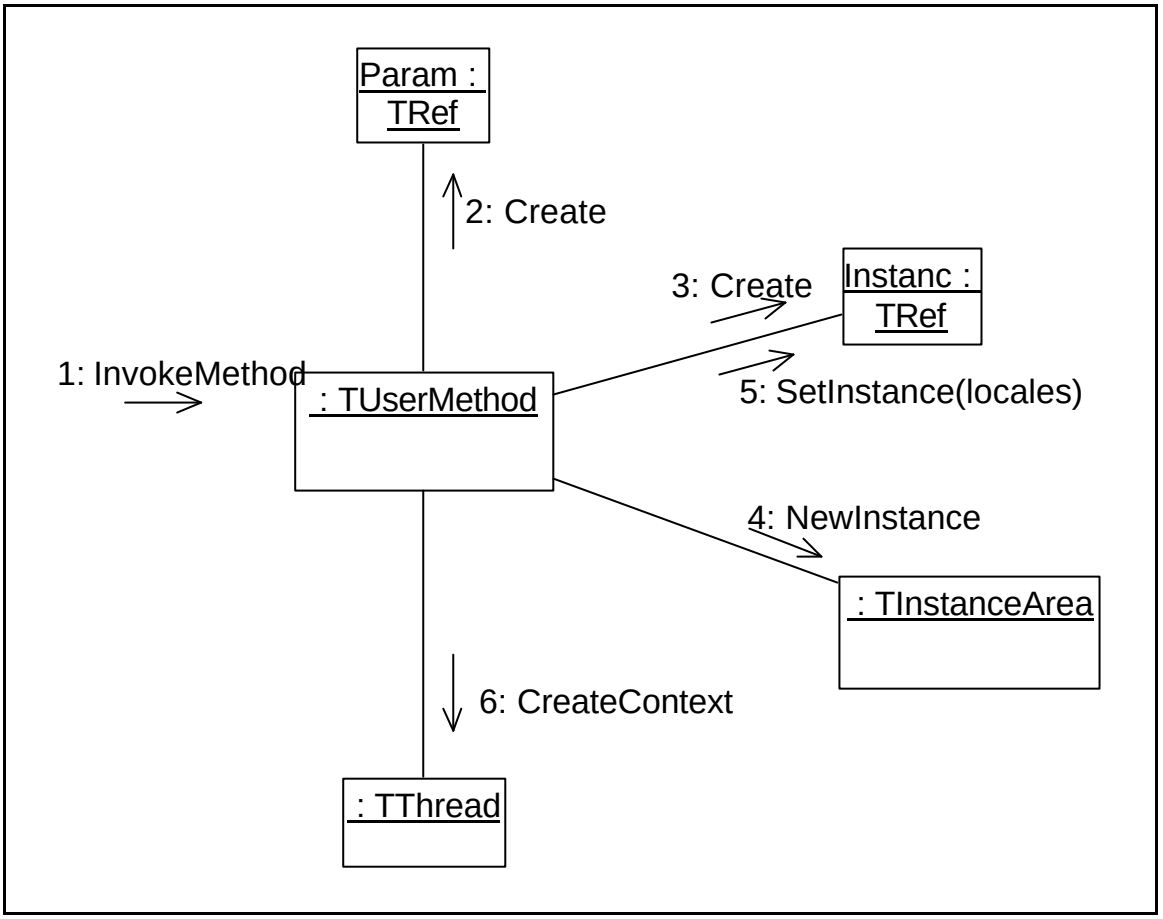


Ilustración 17.8: Invocación de un método de usuario

Cuando el `TUserMethod` recibe el mensaje lo primero que hace es crear todas las referencias que le hagan falta para representar sus argumentos, referencias e instancias locales. En el caso de estas últimas además creará las instancias a las que apunten solicitándoselo al área de instancias (pasos 4 y 5).

A continuación, en el caso de que el método sea de tipo mensaje (concurrente) solicita al área de threads que construya un nuevo thread sobre el que ejecutar el método (paso 6).

En cualquier caso finaliza indicando al thread nuevo o al actual que cree un nuevo contexto que ejecute el método actual y con las referencias locales que le pasa como parámetro.

Por tanto la siguiente instrucción a ejecutar será la primera de este método.

17.1.3.6 Invocación de un método primitivo

Como escenario final, y como representante de todos los métodos primitivos, se muestra ahora el escenario en el caso de que el método hubiese sido un método primitivo, concretamente la suma de números enteros (Ilustración 17.9).

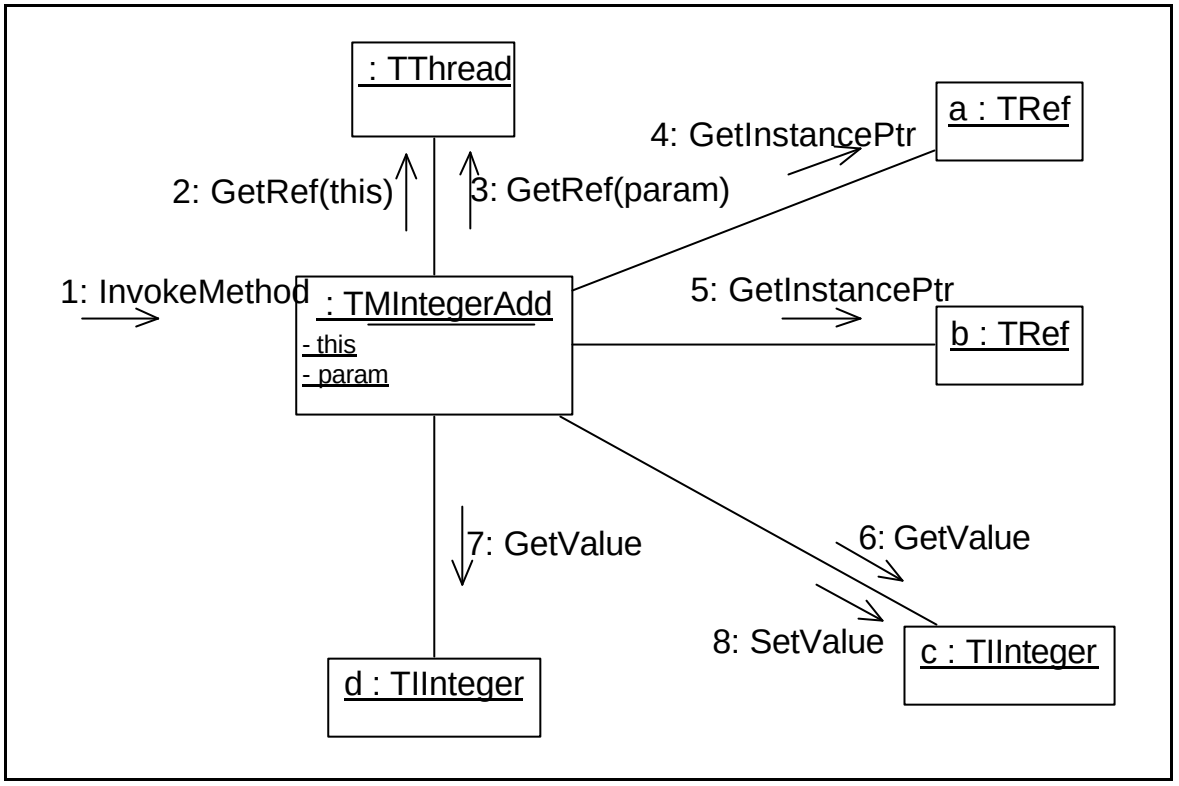


Ilustración 17.9: Ejecución de un método primitivo

El método de suma recibe la notificación de que le han invocado mediante `invokeMethod`.

Lo primero que hace es solicitar al thread sobre el que se está ejecutando el método que le dé acceso a la referencia sobre la que se ha invocado el método y la que se ha enviado como parámetro (pasos 2 y 3).

Una vez localizadas las referencias se les solicita las direcciones de los objetos a los que apuntan (pasos 4 y 5).

Una vez que se tiene acceso directo a las instancias de tipo `TIInteger` se les pide el valor de cada una y se establece en la primera el resultado de sumar ambas.

Nótese que no se realiza ningún cambio en el contexto del thread, por lo que la siguiente instrucción a ejecutarse es la que sigue a la invocación del método `Add`. Se puede considerar como que el método crea un contexto sobre el que se ejecuta y al finalizar su ejecución sale de él, que en la práctica todos estos pasos serían redundantes.

17.2 Extensión reflectiva de la máquina abstracta

A continuación se comenta en detalle como funciona y se ha construido el sistema reflectivo de Oviedo3. Básicamente se sigue el diseño comentado en el Capítulo 13, pero adaptándolo a la máquina Carbayonia. De esta forma el Subsistema de Reflectividad Estructural de Oviedo3 permite a los objetos del sistema:

- Obtener información sobre todas las clases del área de clases o solamente una clase concreta que deseemos.
- Ver los métodos de una clase, sus relaciones de herencia, sus atributos, etc.
- Obtener una instancia e invocar los métodos de su clase.
- Modificar un método y añadirle nuevas instrucciones o variables. Esto sería de utilidad a la hora de desarrollar un IDE activo, es decir un entorno de desarrollo integrado que nos permita programar en tiempo de ejecución, eliminando así los tiempos previos de compilación e interpretación.
- Otra característica interesante es que podemos consultar los hilos (threads) que están ejecutándose en el sistema y cambiar la siguiente instrucción a ser ejecutada. Con esto podremos desarrollar depuradores de código.

Pero todas estas acciones que podemos realizar con nuestro subsistema reflectivo, no están fuera de control, ya que están monitorizados por el mecanismo de control de acceso uniforme [DIA99]. De tal forma que solo usuario y/o objetos autorizados pueden hacer uso de ellas.

Las clases que se describirán a continuación son las que permiten al sistema razonar acerca de sí mismo y actuar sobre sí mismo modificando su comportamiento.

17.2.1 Method

Esta clase es la que permite el acceso, análisis y modificación de los métodos (primitivos y de usuario, objetos de tipo `TPrimitiveMethod` y `TUserMethod` respectivamente) del sistema en tiempo de ejecución. Algunos de los métodos de esta clase no pueden actuar sobre métodos primitivos, pero serán ellos mismos los que comprueben si el método sobre el que se está actuando es o no de usuario.

La declaración de esta clase es:

```
      Class Method
      Isa object
      Methods
            getMethodClassName():string;
            getMethodName():string;
            getInstructions():array;
            isUserMethod():bool;
            setInstrucions(array);
            newMethod(string, string);
            setReturnType(string);
            setMessage(bool);
            setConcurrent(bool);
            setArgs(array);
            setLocalRefs(array);
            setLocalInst(array);
            getLocalInst():array;
            getArgs():array;
            getLocalRefs():array;
            getReturnType():string;
            newLocalInst(string, string);
            isMessage():bool;
            isConcurrent():bool;
      // Los siguientes 4 métodos tendrán valor de retorno o no
dependiendo de si lo tiene el método a invocar.
            invoke(...);
            invoke_R(...);
            invokeWithParamsArray(object,array);
            invokeWithParamsArray_R(object,array);
EndClass
```

El método `getMethodClassName` permite conocer cuál es el nombre de la clase a la que pertenece el método (primitivo o de usuario) del sistema al que se está accediendo a través del objeto de la clase method, y el método `getMethodName` devuelve el nombre de dicho método del sistema.

Los métodos `getInstructions` y `setInstructions` son los que permiten acceder y/o modificar las instrucciones de un método de usuario. Para averiguar en tiempo de ejecución si un método es o no de usuario se puede utilizar la función `isUserMethod`.

El método `newMethod` permite crear un nuevo método de usuario en tiempo de ejecución que no existía en tiempo de compilación, dándole un nombre y asignándolo a una clase de usuario, de tal manera que el objeto que invoca a este método será el que lo represente. Por supuesto la clase a la que pertenecerá el método deberá de existir en el área de clases.

Existen también una serie de métodos en la clase method que permiten modificar o acceder a las características de un método de usuario, como por ejemplo al valor de retorno del método (para el acceso el método `getReturnType`, para la modificación `getReturnType`), los argumentos del método (`getArgs` y `setArgs`), las referencias locales del método (`getLocalRefs`, `setLocalRefs`), las instancias locales (`getLocalInst`, `setLocalInst`). Para el caso de la modificación de las instancias locales, argumentos locales y referencias locales, estos últimos métodos comentados

anteriormente, lo que hacen es modificar el conjunto de estas referencias en el método para poder utilizarlas la próxima vez que se invoque al método.

Para la creación en tiempo de ejecución de una instancia local se ha definido el método `newLocalInst`.

Este último crea una instancia local con el nombre y el tipo especificados. Si esa instancia local se está creando en el método que se está ejecutando actualmente, además de añadir la instancia al conjunto de instancias locales, prepara dicha instancia para que pueda ser utilizada en ese contexto.

El método permite una construcción especial para las instancias de las clases primitivas integer, float y string. Consiste en indicar en el segundo parámetro el nombre de la clase a la que pertenece la instancia local, seguida del carácter de subrayado "_" y a continuación el valor de inicialización para los objetos de estos tipos.

Definición instancias en el simulador	**Definición de instancias en tiempo de ejecución en un método**
Class Demostracion Methods Ejemplo() Instances I:integer(10); C:Constream; Code c.write(i); Endcode Endclass	Class Demostración Methods Ejemplo() Instances Ac:classArea; Cl:clase; Mejemplo:method; Sejemplo:string('EJEMPLO'); Sinteger:string('INTEGER_10'); Si:string('I'); Sc:string('C'); SConstream:string('CONSTREAM'); Code Ac.getClase():cl; Cl.getMethod(sejemplo):mejemplo; Mejemplo.newLocalInst(si, sinteger); Mejemplo.newLocalInst(sc,sconstream); c.write(i); Endcode EndClass

Tabla 2: Definición de instancias en tiempo de ejecución en un método

En el ejemplo anterior se puede comprobar como a pesar de no haber definido las instancias `C` e `I`, éstas pueden crearse en tiempo de ejecución añadiéndolas a un método determinado (y en este caso también al contexto actual para poder ser utilizada dicha instancia inmediatamente después de la definición).

También se puede conocer en tiempo de ejecución si un método de usuario es concurrente o exclusivo y modificar sus valores (métodos `isConcurrent` y

`setConcurrent` respectivamente) o si es un método o un mensaje y modificar esta característica (métodos `isMessage` y `setMessage`).

Además la clase method ofrece la posibilidad de invocar métodos tanto primitivos como de usuario en tiempo de ejecución, aun siendo este desconocido en tiempo de compilación, esto es posible gracias a los métodos:

- `invoke:` Deberá indicarse el objeto sobre el que se realiza la invocación o el objeto de tipo Reference asociado al objeto que realiza la invocación, así como los parámetros que requiera el método a invocar.

  Este método podrá tener o no un valor de retorno dependiendo de si lo tiene el método a invocar.

- `invoke_R:` Este método proporciona la funcionalidad del método `invoke`, pero con la diferencia de que si existe un valor de retorno en el método que se va a invocar, al método `invoke_R` deberá asignársele como retorno un objeto de tipo string cuyo valor es el nombre de la instancia donde se recogerá el valor de retorno.
- `invokeWithParamsArray:` Este método proporciona la funcionalidad del método `invoke,` pero con la diferencia de que los parámetros del método a invocar se deben indicar a través de un array.
- `invokeWithParamsArray_R:` Este método proporciona la funcionalidad del método `invokeWithParamsArray` comentado anteriormente, pero con la diferencia de que si existe un valor de retorno en el método que se va a invocar, al método `invokeWithParamsArray_R` deberá asignársele como retorno un objeto de tipo string cuyo valor es el nombre de la instancia donde se recogerá el valor de retorno.

A continuación se mostrará un ejemplo de cómo invocar al método `less` de la clase `integer`.

Invocación de métodos en el simulador	Invocación de métodos a través del módulo de reflectividad

Class Demostracion
Methods
Ejemplo()
Refs
B:bool;
Instances
uno:integer(1);
dos:integer(2);
 	
Code
uno.less(dos):B;
 	
Endcode
EndClass
	Class Demostracion
	Methods
	Ejemplo()
	Refs
	B:bool;
	Instances
	uno:integer(1);
	dos:integer(2);
	
	Ac:classarea;
	Cl:Class;
	mLess:method;
	sLess:string('LESS');
	sInteger:string('INTEGER');
	Code
	Ac.getClase(sInteger):Cl;
	Cl.getMethod(sLess):mLess;
	mLess.invoke(uno,dos):B;
	EndCode
	EndClass

Tabla 3: Invocación de métodos a través del módulo de reflectividad

17.2.2 Clase

Esta clase es la que permite el acceso, análisis y modificación de clases (primitivas y/o de usuario, objetos de tipo `TBasicClass` y `TUserClass` respectivamente) del sistema en tiempo de ejecución. Algunos de los métodos de esta clase no pueden actuar sobre clases primitivas, pero serán ellos mismos los que comprueben si la clase sobre el que se está actuando es o no de usuario.

```
Class clase
Isa object
Methods
      getMethodsNames():array;
      getClaseName():string;
      getMethod(string):method;
      getMethods():array;
      gettype():string;
      getAncestors():array;
      getDirectAncestors():array;
      setAgg(array);
      setAss(array);
      setIsa(array);
      getAgg():array;
      getAss():array;
      getIsa():array;
      isUserClass():bool;
EndClass
```

El método `getClaseName` permite conocer el nombre de la clase del sistema a la que se está accediendo a través del objeto de la clase `clase` y el método `getMethodsNames` permite acceder al nombre de los métodos de dicha clase del sistema.

A través del método `getMethod` se obtiene el método (objeto de tipo method) de la clase cuyo nombre se haya especificado como parámetro. Para obtener todos los métodos de la clase se define el método `getMethods`.

Para averiguar si una clase es de usuario o primitiva, se pueden utilizar los métodos:

- `Gettype`, que devuelve un objeto de tipo string indicando si la clase es primitiva o de usuario.
- `IsUserClass`, que devuelve un objeto de tipo bool que valdrá *true* si la clase es de usuario o *false* si la clase es primitiva.

La función `getAncestors` indica cuáles son los ancestros de la clase (primitiva o de usuario), tanto los directos como los indirectos, `getDirectAncestors` indica sólo los ancestros directos de la clase (primitiva o de usuario) y `getIsa` permite obtener el conjunto `Isa` (que se obtuvo a partir la cláusula `Isa` de la definición de una clase en Carbayonia o de `setIsa`) con los ancestros directos de una clase de usuario.

Los métodos `setAgg`, `getAgg` permiten acceder y modificar a los agregados de una clase de usuario, los métodos `setAss`, `getAss` a los asociados de la clase y los métodos `setIsa`, `getIsa` al conjunto de ancestros (`Isa`) de la clase de usuario.

17.2.3 ClassArea

Esta clase es la que permite el acceso y modificación del área de clases de la máquina abstracta orientada a objetos Carbayonia.

```
Class classArea
Isa object
Methods
      getClassesNames():array;
      getClase(string):clase;
      addNewClass(string);
      createNewClass(string);
EndClass
```

A través de esta clase se podrá averiguar todos los nombres de las clases que contienen, concretamente con el método `getClassesNames`, así como hacer visibles y manipulables las clases con el método `getClase`.

`AddNewClass` permite añadir en tiempo de ejecución al área de clases la clase de usuario cuya definición se encuentra en un determinado archivo con tan sólo indicar el path de dicho archivo.

Y por último, el método `createNewClass` que crea en el área de clases una nueva clase de usuario. Éste será el primer paso en la creación de clases de usuario en

tiempo de ejecución. Posteriormente se podrá empezar a definir sus métodos, instrucciones, agregados de los métodos, etc.

17.2.4 Instance

Permite el acceso a las instancias (primitivas y de usuario, objetos de tipo `TobjectDerivedInstance` y `TIObject` las primeras y `TUserInstance` las segundas) de la máquina abstracta. Algunos de los métodos de esta clase no pueden actuar sobre instancias primitivas, pero serán ellos mismos los que comprueben si la instancia sobre la que están actuando es de o no de usuario.

```
Class Instance
Isa object
Methods
      isAggregate():bool;
      getAggregates():array;
      getAssociates():array;
      isUserInstance():bool;
      getClass():string;
      isSubInstance():bool;
      getAncestors():array
EndClass
```

El método `isAggregate` devolverá un valor booleano que valdrá true si la instancia es agregada, false en caso contrario. `IsUserInstance` devolverá true si la instancia es de usuario.

Los métodos `getAggregates`, `getAssociates`, `getAncestors` permiten acceder a los agregados, asociados y ancestros respectivamente de una instancia de usuario.

Y finalmente, `getClass` devuelve el nombre de la clase a la que pertenece la instancia.

17.2.5 InstanceArea

Esta clase es la que permite el acceso y modificación del área de instancias de la máquina abstracta orientada a objetos Carbayonia.

```
      Class InstanceArea
      Isa  Object
      Methods
            getInstances():array;
            createInstance(string):instance;
      EndClass
```

El método `getInstances` devuelve las instancias del área de instancias a través de un array de objetos de tipo instance.

`CreateInstance` crea en el área de instancias una nueva instancia del tipo indicado. Este método deberá ser utilizado cuando se esté creando una clase en tiempo de ejecución para crear una instancia de usuario de dicha clase.

17.2.6 Context

Esta clase permite el acceso y modificación de determinadas características de un contexto (objeto de tipo `TContext`) de la máquina.

```
Class Context
Isa object
Methods
      newLocalInst(string, string);
      getLocalInst():array;
      getLocalRefs():array;
      getPreviousContext():context;
      getCurrentMethod():method;
EndClass
```

Puesto que los contextos se crean y se destruyen en la máquina de Carbayonia, cada vez que se invoque algunos de los métodos de la clase `Context`, habrá que comprobar que el objeto de tipo `TContext` aún existe en la pila de contextos del thread que lo creó.

Los métodos de que consta la clase `Context` son:

- `NewLocalInst`: Crea una instancia local en el contexto.

El método permite una construcción especial para las instancias de las clases primitivas `integer`, `float` y `string`. Consiste en indicar en el segundo parámetro el nombre de la clase a la que pertenece la instancia, seguida del carácter de subrayado "_" y a continuación el valor de inicialización para los objetos de estos tipos.

Definición instancias en el simulador	**Definición de instancias en tiempo de ejecución en un contexto**
Class Demostracion Methods Ejemplo() Instances I:integer(10); C:constream; Code C.write(I); Endcode Endclass	Class Demostración Methods Ejemplo() Instances Th:thread; Contexto:context; Sconstream:string('CONSTREAM'); Sinteger:string('INTEGER_10'); Si:string('I'); Sc:string('C'); Code Th.setCurrentThread(); Th.getContext():contexto; Contexto.newLocalInst(Sc,Sconstream); Contexto.newLocalInst(si, sinteger); c.write(i); Endcode EndClass

Tabla 4: Definición de instancias en tiempo de ejecución en un contexto

Una vez que se sale del contexto la instancia deja de existir ya que no se añade al conjunto de instancias locales del método.

- `getLocalInst:` Devuelve las instancias locales del contexto.
- `getLocalRefs:` Devuelve las referencias locales y argumentos del método de usuario que se está ejecutando en el contexto.
- `GetPreviousContext:` Devuelve un objeto de tipo context que hace visible y manipulable el contexto anterior al actual.
- `getCurrentMethod:` Devuelve un objeto de tipo method que hace visible y manipulable el método que se está ejecutando en el contexto actual.

17.2.7 Reference

```
Class Reference
Isa object
Methods
      getRefName():string;
      getRefType():string;
      isFree():bool;
      newReference(string, string);
EndClass
```

Para conocer el nombre de la referencia se utilizará el método `getRefName` y para saber el nombre de la clase a la que pertenece la referencia el método `getRefType`.

`IsFree` devolverá *true* si la referencia está libre, *false* en caso contrario.

El método `newReference` permite crear nuevas referencias en tiempo de ejecución. Estas nuevas referencias se podrán utilizar más adelante para definir y obtener los agregados, asociados y ancestros de una clase, las referencias locales, argumentos y referencias locales de un método, etc...

Se permite una construcción especial para referencias que se creen de las clases primitivas `integer`, `float` y `string`. Consiste en indicar en el segundo parámetro el nombre de la clase a la que pertenece la referencia, seguida del carácter de subrayado "_" y a continuación el valor de inicialización para los objetos de estos tipos.

17.2.8 Thread

Esta clase es la que permite el acceso y modificación de threads (objetos de tipo Tthread) en la máquina abstracta de Carbayonia.

```
Class Thread
Isa object
Methods
      getState():string;
      killThread();
      getContext():context;
      setCurrentThread();
      getRef(string):reference;
EndClass
```

La forma de obtener el estado en el que se encuentra un thread (el estado de un thread puede ser Ready, Killed o UnHandled) es a través del método `getState`.

Si se desea asignar a un determinado thread el estado de Killed y por tanto finalizar su ejecución, se utilizará el método `KillThread`. Este último también elimina los contextos y handlers de la pila del thread eliminado.

El método `getContext` el contexto actual del thread, último contexto en la pila de contexto del thread.

`SetCurrentThread` asigna al objeto de tipo `Thread` que realizó la invocación de este método, el thread que está ejecutando dicho método.

`getRef` devuelve un objeto de tipo reference que representa a la referencia cuyo nombre sea el especificado.

17.2.9 ThreadArea

Esta clase es la que permite el acceso al área de threads de la máquina abstracta orientada a objetos Carbayonia.

```
Class ThreadArea
Isa Object
Methods
      getThreads():array;
      getNumThreads():integer;
      getThread(integer):thread;
EndClass
```

Los métodos de que consta la clase son los siguientes:

- `GetThreads`: Devuelve los threads del área de threads.
- `GetNumThreads`: Indica el número de threads que hay en el área de threads.
- `GetThread:` Devuelve del conjunto de threads del área de threads aquél que ocupa la posición indicada por el objeto de tipo integer que se especifica como parámetro.

17.2.10 Instruction

Esta clase es la que permite el acceso y creación de nuevas instrucciones (objetos `TInstruction`) de un método de usuario del sistema en tiempo de ejecución.

```
Class Instruction
Isa object
Methods
      getInstrType():string;
      getString():string;
      newTInstrCall(sring, string,array, string);
      newInstrExit();
      newInstrNew(string);
      newInstrAssign(string, string);
      newInstrDelete(string);
      newInstrThrow();
      newInstrJmp(instruction);
      newInstrHandler(instruction);
      newInstrJcc(string, string, instruction);
      newInstrJcNull(string, string, instruction);
      newInstrJmpJC(integer);
      newInstrHandlerJC(integer);
      newInstrJccJC(string, string, integer);
      newInstrJcNullJC(string, string, integer);
EndClass
```

Es posible, en tiempo de ejecución, crear nuevas instrucciones que más tarde, tras ser asignadas a algún método de usuario, podrán ser ejecutadas. Los métodos que proporcionan esta capacidad son:

- `newTInstrCall:` Asigna al objeto instruction una nueva instrucción de tipo `TInstrCall`. El método deberá recibir el nombre de la instancia a la que se le invoca un método, el nombre del método que se invoca, un array con el nombre de los parámetros del método a invocar y por último el valor de retorno, que valdrá "VOID" si el método que se quiere invocar no tiene valor de retorno.
- `NewInstrExit:` Asigna al objeto de tipo instruction una nueva instrucción de tipo `TinstrExit`.
- `NewInstrNew:` Asigna al objeto de tipo instruction una nueva instrucción de tipo `TinstrNew`. Deberá indicarse el nombre de la referencia que va asociada a la sintaxis de esta instrucción `new` (ver el juego de instrucciones del simulador para más información sobre ésta).
- `NewInstrAssign:` Asigna al objeto de tipo instruction una nueva instrucción de tipo `TinstrAssign`. Deberá indicarse el nombre de la referencias destino y origen asociadas a la sintaxis de la instrucción `Assign` (ver el juego de instrucciones del simulador para más información sobre ésta).
- `NewInstrDelete:` Asigna al objeto de tipo instruction una nueva instrucción de tipo `TinstrDelete`. El parámetro deberá especificar el nombre de la referencia que va asociada a la sintaxis de esta instrucción `delete` (ver el juego de instrucciones del simulador para más información sobre ésta).
- `NewInstrThrow`: Asigna al objeto de tipo instruction una nueva instrucción de tipo `TinstrThrow`.
- `NewInstrJmp:` Asigna al objeto de tipo instruction una nueva instrucción de tipo `TinstrJmp`. Este método recibirá como parámetro un objeto de tipo instruction que represente a la instrucción a la que se deberá saltar.

- `NewInstrHandler:` Asigna al objeto de tipo instruction una nueva instrucción de tipo `TinstrHandler`. Este método recibirá como parámetro un objeto de tipo instruction que represente a la instrucción a la que se deberá bifurcar el flujo de control en caso de que se produzca una excepción.
- `NewInstrJcc:` Asigna al objeto de tipo instruction una nueva instrucción de tipo `TinstrJcc`. Deberá indicarse al método el tipo de instrucción `TinstrJcc` (ver tabla 5), el nombre de la referencia de tipo `bool` que se debe evaluar y por último la instrucción a la que se desea saltar en caso de que dicho salto deba producirse.
- `NewInstrJcNull:` Asigna al objeto de tipo instruction una nueva instrucción de tipo `TInstrJcNull`. Deberá indicarse al método el tipo de instrucción `TinstrJcNull`, el nombre de la referencia de tipo `bool` que se debe evaluar y por último la instrucción a la que se desea saltar en caso de que dicho salto deba producirse.
- `NewInstrJmpJC:` Igual que el método `newInstrJmp` pero en vez de recibir como parámetro un objeto de tipo instruction que representa a la instrucción a la que se desea saltar, deberá recibir un objeto de tipo integer que indique el salto relativo hacia delante o hacia atrás que se debe hacer.
- `newInstrHandlerJC(integer):` Igual que el método `newInstrHandler` pero en vez de recibir como parámetro un objeto de tipo instruction, deberá recibir un objeto de tipo integer que indique el salto relativo hacia delante o hacia atrás que se deba producir en caso de excepción.
- `NewInstrJccJC:` Igual que el método `newInstrJcc` pero en vez de recibir como parámetro un objeto de tipo instruction que representa a la instrucción a la que se desea saltar, deberá recibir un objeto de tipo integer que indique el salto relativo hacia delante o hacia atrás que se debe hacer en caso de que se cumpla la condición para ejecutar el salto.
- `NewInstrJcNullJC:` Igual que el método `newInstrJcNull` pero en vez de recibir como parámetro un objeto de tipo instruction que representa a la instrucción a la que se desea saltar, deberá recibir un objeto de tipo integer que indique el salto relativo hacia delante o hacia atrás que se debe hacer en caso de que se cumpla la condición para ejecutar el salto.

Los métodos que permiten averiguar características sobre las instrucciones son:

- `GetInstrType:` Devuelve el tipo de instrucción que representa el objeto instruction. Los posibles tipos de instrucción son: `TinstrCall`, `TinstrExit`, `TinstrNew`, `TinstrAssign`, `TinstrDelete`, `TinstrThrow`, `TinstrJmp`, `TinstrHandler`, `TinstrJcc`, `TinstrJcNull`.
- `GetString:` Devuelve un objeto de tipo string que reconstruye la instrucción que representa el objeto de tipo instruction.

17.2.11 Concurrencia

La concurrencia en la máquina abstracta define un sistema básico que nos da la posibilidad de crear aplicaciones multihilo.

El enfoque elegido es que sea la persona encargada del diseño de la clase la que decida para cada servicio si se deberá esperar a que finalice o si se podrá seguir ejecutando el hilo cliente. Por tanto los servicios que ofrecen las clases se clasificarán en dos tipos:

- *Métodos*. Son los que se han venido utilizando y los que se comportan como las subrutinas de los lenguajes tradicionales. Cuando se invoca a un método la ejecución se traslada a él hasta que finalice y sigue por el punto donde fue llamado. Nótese que no se crea un nuevo hilo, sino que es un único hilo que va avanzando a través de distintos objetos.
- *Mensajes*. Crean un nuevo hilo que nace en el método invocado. Se les llama así porque se les podría interpretar como que el cliente le pasa un mensaje al objeto destino (con sus correspondientes parámetros) para que le realice, cuando pueda, una determinada labor. Pero, al igual que cuando se le envía un mensaje a alguien, uno no se queda parado esperando.

Esta es una declaración de una propiedad de la clase. Se supone que la declaración de un mensaje dentro de una clase, es realizada por el propio creador de la clase puesto que se puede ejecutar de forma concurrente con el resto de los métodos. De forma similar, cada vez que se invoca a un mensaje de una clase desde otra clase externa, no se sabrá si es o no concurrente.

A la hora de la declaración de las clases el diseñador dispondrá de una nueva palabra reservada para especificar el tipo de cada servicio: **Messages**. Así tendremos pues la declaración de una clase:

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

Por tanto, se seguirá utilizando la sintaxis habitual para invocar un servicio, y mediante la declaración del mismo que se halla en el área de clases se sabrá, si se debe esperar a que finalice o bien continuar.

Lógicamente los mensajes no pueden declararse con valor de retorno, ya que no hay nadie esperando para recogerlo.

17.2.12 Un método que se invoca a sí mismo

A través de la funcionalidad que proporciona la reflectividad es posible que un método se invoque a sí mismo.

Suponiendo que esto ocurriera, la ejecución del método no terminaría hasta que se consumieran los recursos del sistema o bien se generaran en el área de instancias tal número de instancias que se produjera una excepción del tipo SegmtOvfl (por rebose de segmento).

17.2.13 Modificación de las instrucciones de un método

Para modificar las instrucciones de un método en tiempo de ejecución, es necesario utilizar el método `setInstructions` de la clase `Method`. Este método sustituye las instrucciones actuales del método por otro conjunto de instrucciones, eliminando de memoria las anteriores.

Podría hacerse un uso indebido de la función `setInstructions` en el caso de que se tratara de asignar al método una instrucción que había sido eliminada. Pongamos el siguiente ejemplo:

```
...
instances
      m:method;
      arr:array;
      uno:integer(uno);

code

...
// m representa un método de usuario cualquiera.
m.getInstructions():arr; // Paso 1
arr.setSize(uno); // Paso 2
m.setInstructions(arr); // Paso 3
...

endcode
```

Primero se obtienen las instrucciones de un método (cualquiera) a través del método `getInstructions` (Paso 1). Ahora supongamos que lo que se desea es reprogramar el método de tal manera que su código estuviera formado únicamente por la primera instrucción del código antiguo. Para ello le indicaríamos al array su nuevo tamaño (paso 2) y finalmente asignaríamos al método el array con una única instrucción.

Este procedimiento es incorrecto. El método `setInstructions` elimina de memoria las instrucciones antiguas y por tanto, la instrucción que está tratando de asignarse no existe realmente, se estaría accediendo a una posición de memoria con datos no válidos y más adelante al acceder a esa instrucción se produciría un error en el sistema.

17.2.14 Un método que modifica sus propias instrucciones

Puede ocurrir que dentro del código de un método, se establezcan las instrucciones necesarias para que un método modifique su propio código haciendo uso de la capacidad reflectiva del sistema.

El problema en concreto surge cuando se está ejecutando el método que va a autoprogramarse.

Para que las instrucciones del método se modifiquen habrá que hacer uso del método `setInstructions` de la clase `Method`. Este método sustituye las instrucciones actuales del método por otro conjunto de instrucciones, eliminando de memoria las anteriores.

No es el objetivo del método `setInstructions` modificar el `MC` (method counter) del contexto en el que se está ejecutando el método, por tanto una vez ejecutada la instrucción `setInstructions` el Thread tratará de ejecutar la instrucción siguiente a `setInstructions` puesto que la última instrucción no fue `exit` y no ha se ha eliminado de la pila de contextos del thread dicho contexto.

Podría ocurrir que la siguiente instrucción que está tratando de ejecutar el thread, una vez se ha modificado el conjunto de instrucciones del método no existiera, en cuyo caso se produciría un error del tipo `TInvalidMC`.

Suponiendo que la instrucción siguiente a `setInstructions` existiera, habría que tener en cuenta que las nuevas instrucciones del método comenzarían a ejecutarse en el contexto donde se asignaron al método (contexto actual) y por tanto su ejecución no comenzaría en la primera instrucción del conjunto de instrucciones, sino en aquella instrucción que ocupara el lugar de la siguiente instrucción a la antigua instrucción `setInstructions`. Por tanto, habría que ser muy cuidadoso a la hora de reprogramar el método, y más si se tiene en cuenta que dicho método puede volver a ser invocado, en tal caso la ejecución de sus instrucciones sí comenzaría por la primera de ellas.

17.2.15 Resultados de la reflectividad en el sistema

A continuación se muestra una ilustración que resume gráficamente la actuación del módulo de reflectividad en el sistema.

En el área de instancias se muestran únicamente para una mejor comprensión, las clases básicas que representan a las instancias del módulo de reflectividad.

Vemos cómo a través de la reflectividad es posible que desde una instancia de tipo "CLASE" se pueda acceder a las clases del área de clases, tanto a las clases primitivas como de usuario, y modificar o incluso crear clases de usuario. Asimismo, desde las instancias de tipo "THREADAREA" es posible acceder al área de threads, desde las instancias de tipo "THREAD" acceder a los threads existentes y modificarlos.

Se muestra también gráficamente la forma de cosificar los objetos del sistema base a través de los objetos del módulo de reflectividad.

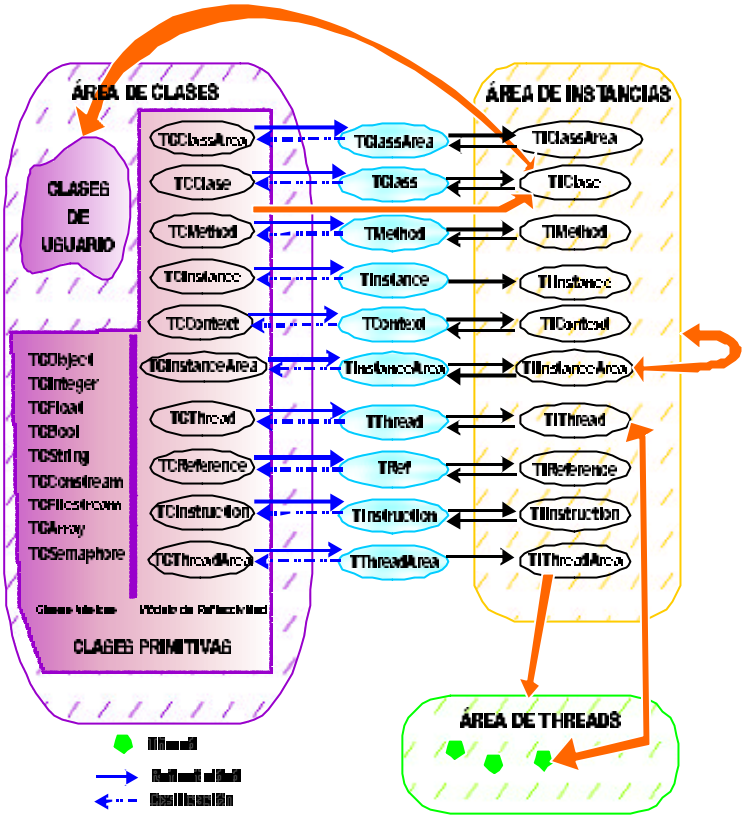


Ilustración 17.10: Resultados del módulo de reflectividad en el sistema

17.2.16 Un sencillo pero potente Shell

Para ilustrar la simplicidad y potencia de la reflexión hemos implementado un sencillo pero potente Shell. Lo único que hace este es leer cadenas de caracteres de la forma:

```
objeto.metodo(param1,param2,..):valretorno;
```

A continuación se separa la cadena en objeto, metodo, param#, valretorno y se lleva a cabo la invocación correspondiente. Esto tan sencillo que no tiene más de 40 líneas de código, nos permite por ejemplo:

- Cargar clases desde un fichero en tiempo de ejecución
- Usar archivos de comandos por lotes
- Inspeccionar todo el sistema en busca de clases, instancias, valores de atributos.
- Crear instancias de una determinada clase
- Crear clases de forma dinámica en tiempo de ejecución
- Invocar métodos de una clase para una determinada instancia
- Añadir y borrar instrucciones a un determinado método
- E incluso adaptar, cambiar y personalizar el propio objeto Shell.

Un ejemplo interesante es que podemos de una forma muy sencilla personalizar el `prompt` del Shell. Para ello sabemos que existe un atributo de tipo string llamado prompt dentro de la clase a la que pertenece nuestro Shell. A continuación se muestra el código carbayon necesario para cambiar el prompt del símbolo ‘>’ a ‘Hola>’:

```
>
>SHELL.NEWLOCALINST(NP,STRING_Hola>);
>PROMPT.SET(NP);
Hola>
```

Creamos un nuevo atributo dentro de la clase a la que pertenece el objeto SHELL llamado `NP`, que es una `STRING` con el contenido Hola.

Antes de seguir démonos cuenta que es una cadena que cumple el formato anteriormente comentado y el propio Shell lo único que ha hecho es realizar la llamada al método `NewLocalInst` del objeto Shell.

A continuación simplemente se llama al metodo `Set` de la variable `Prompt` que es un atributo interno de la clase a la que pertenece el Shell. Este atributo es de tipo `String` y el método `Set` es el método típico para cambiar el contenido de una cadena de caracteres en el lenguaje Carbayón.

17.3 Servicios Web -> Oviedo3 -> DCOM

Una vez expuesto en detalle el contexto de implementación de nuestro prototipo, formado por la arquitectura interna de la máquina abstracta Carbayonia, junto con la arquitectura de su extensión reflectiva, es el momento para comenzar con la descripción detallada de la implementación del mismo, que permitirá la interoperabilidad de plataformas de Servicios Web con plataformas de objetos DCOM de forma transparente.

Dividiremos la realización de nuestro prototipo en dos partes, según el sentido de las invocaciones realizadas, es decir, desde Oviedo3 a DCOM y de Servicios Web hacia Oviedo3. En cada una de estas partes se describirá en detalle, por un lado, una instanciación de nuestro modelo arquitectónico dando lugar a una arquitectura concreta, adaptada a Carbayonia, por otro, la posterior implementación de dicha arquitectura en el lenguaje base de Carbayonia que es C++ y por último mostraremos cómo se lleva a cabo la comunicación desde el punto de vista de la máquina Carabayonia.

En la sección 17.6, uniremos ambas partes y describiremos como trabajan de forma conjunta para obtener una comunicación virtual y transparente entre los Servicios Web y los objetos DCOM.

17.4 DCOM <- Oviedo3 -> DCOM

En esta sección realizaremos el diseño e implementación que permitirá a los objetos de Oviedo3 utilizas los objetos DCOM de forma transparente. La hemos estructurado en las tres siguientes subsecciones:

- Diseño arquitectónico, donde se describe la arquitectura obtenida como resultado de una instanciación de nuestro modelo arquitectónico adaptado a la máquina virtual Carbayonia y aplicada a una determinada plataforma, en nuestro caso a DCOM.

- Implementación, dentro de la que vamos a describir en detalle la implementación de la arquitectura obtenida anteriormente, comentando la dificultades, restricciones y problemas obtenido a nivel de programación en C++ que es el lenguaje en el que se ha desarrollado Carbayonia.

- Por último, en la sección de validación se mostrarán programas escritos en Carbayón, que es el lenguaje ensamblador de la máquina Carbayonia, donde se interacciona con objetos DCOM como si de objetos propios de la máquina se tratase. Validando así los resultados obtenidos del diseño arquitectónico y de su posterior implementación.

17.4.1 Diseño Arquitectónico

El diseño arquitectónico mostrado a continuación viene como resultado de la instanciación de la colaboración genérica <- MA -> descrita en nuestro modelo arquitectónico en la sección 15.1, pero en este caso aplicado a la máquina abstracta de Oviedo3, obteniendo así una colaboración concreta que denominaremos DCOM <- Oviedo3 -> DCOM. En nuestro caso y debido a que sólo vamos a interactuar con una plataforma externa no hemos implementado la clase `MAExternalSystemDCOM` que hereda de `MAExternalSystem`.

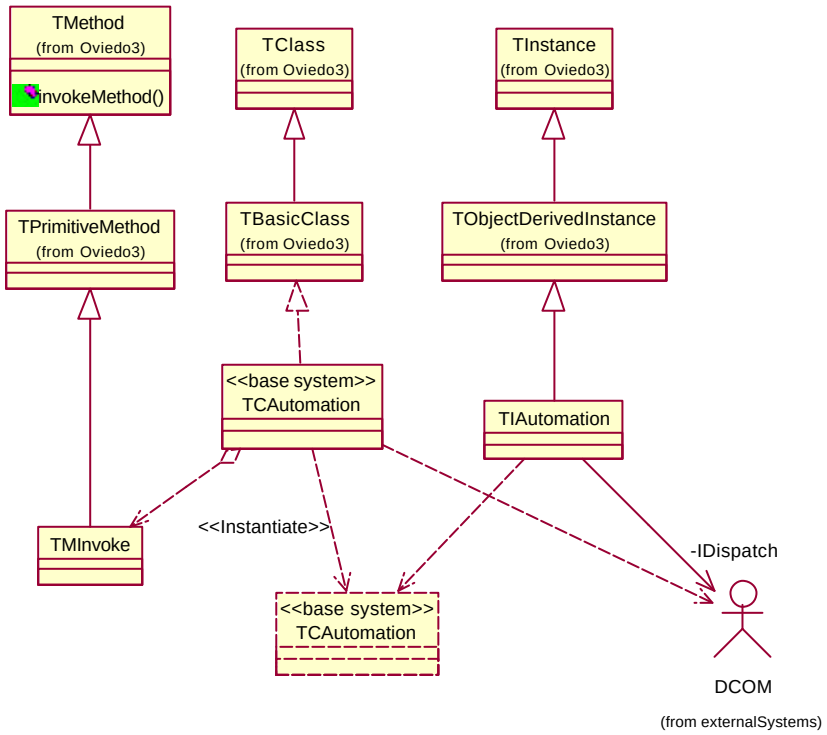


Ilustración 17.11: DCOM <- Oviedo3 -> DCOM

La ilustración superior muestra la vista estática del diseño arquitectónico resultante. El resultado es prácticamente el diseño esbozado en la Ilustración 16.2 (véase

página 245) salvo pequeños detalles en los nombres de las clases y algún nivel de generalización adicional que aporta Oviedo3. Por ejemplo, la clase `MAClassExternal` la hemos denominado `TCAutomation`, la clase `MAIntanceExternal` la hemos denominado `TIAutomation`, la clase estereotipada con <<*base system*>> son instancias de `TCAutomation`, como se indicó en su momento, salvo que el nombre de la clase (instancia) antes era `MAClassExternalSystemObject1` y ahora es `TCAutomation` estereotipado con <<*base system*>>. Obsérvese además que `TMInvoke`, `TCAutomation` y `TIAutomation` no heredan directamente de `TMethod`, `TClass` y `TInstance` sino que Oviedo3 añade un nivel adicional de generalidad e incluye las clases `TPrimitiveMethos`, `TBasicClass` y `TObjectDerivedInstance`. Por lo demás y dada la gran similitud entre MA y Oviedo3, no se ha realizado ningún tipo de adaptación o modificación al patrón de diseño inicial.

En cuanto a la introspección, invocación e instanciación se aplica lo comentado en la sección 16.1.3 (véase página 242) cuando se estudio la instanciación de la colaboración genérica <- MA -> para el sistema DCOM.

Un pequeño matiz a comentar es que en el caso inicial `MAMethod` tiene un método denominado `invoke` para recibir invocaciones a métodos y en Oviedo3 el método equivalente, en este caso de la clase `TMethod` tiene de nombre `invokeMethod`().

Para el tema de la localización, el modelo no se ajusta tanto, ya que mientras por un lado MA define un diccionario de clases como un atributo de clase denominado `allClasses` dentro de `MAClass`, con posibilidad de federación a través de patrón de diseño *Cadena de Responsabilidades*, en el caso de Oviedo3, como se vio en la sección 17.1.1, contienen una clase especial con este propósito denominada `TClassArea`. Por otro lado, en este prototipo al interactuar con una sola plataforma externa no se ha considerado la federación de clases de tal forma que la búsqueda de la clase dentro de DCOM se realiza directamente dentro del método `getClass()` de `TClassArea`, como se describirá con detalle en la sección siguiente.

17.4.2 Implementación

A continuación mostraremos los detalles de implementación más representativos en cada una de las clases diseñadas. Posteriormente se expondrán los detalles y criterios utilizados en las conversiones de tipos y localización de clases.

Utilizaremos el servicio de Automation para que todos los objetos y clases DCOM estén disponibles en Oviedo3 como instancias y clases virtuales a las que todo objeto Oviedo3 puede acceder de forma transparente.

Utilizaremos la interfaz IDispath para interceptar, gestionar y distribuir las llamadas realizadas hacia DCOM desde otros sistemas, de tal forma que todos los objetos y clases de Oviedo3, tanto primitivas, definidas por el usuario, como virtuales, pertenecientes a otras plataformas de invocación síncrona remota, sean vistas como objetos nativos.

17.4.2.1 TCAutomation

Como se comentó en la sección 15.1.3 y hemos adaptado para Oviedo3 en la sección anterior, la clase `TCAutomation` no es una clase propia de la máquina, si no más bien es un generador de clases. Donde sus instancias ya si son clases disponibles en la máquina que sirven para envolver las clases externas de la plataforma DCOM a través del servicio Automation comentado en detalle en la sección 16.1.2.

La clase `TCAutomation` recibe como parámetro en su constructor un objeto de tipo string que indica el nombre de la clase accesible a través de *automation*. El cometido de esta clase es, al igual que en el resto de las clases de Carbayonia, añadir a la colección de métodos de la clase los métodos que dicho objeto *automation* expone a través del mecanismo de introspección de su interfaz *IDispatch*. Veamos su código a continuación.

```
TCAutomation::TCAutomation(const std::string &nombre)
{
// Guardamos el nombre de la clase
name = nombre;

// Obtenemos el CLSID de la aplicación
// Quitamos el '_' y ponermos un '.' en el nombre de la clase
string aux=nombre;
SetDot(aux);

// Convertimos a unicode
wstring progid;
cstows(progid,aux);

DWORD clsctx = CLSCTX_SERVER ;
CLSID clsid ;
HRESULT hr = ::CLSIDFromProgID(progid.c_str(), &clsid) ;

// Creamos el componente
IDispatch* pIDispatch = NULL ;
hr = ::CoCreateInstance(clsid,
                        NULL,
                        clsctx,
                        IID_IDispatch,
                        (void**)&pIDispatch) ;

// Tratamos de conseguir una ITypeInfo
ITypeInfo *pITypeInfo = NULL;
hr = pIDispatch->GetTypeInfo(0,0,&pITypeInfo);

// Tratamos de conseguir los atributos
TYPEATTR *pTypeAttr = NULL;
hr = pITypeInfo->GetTypeAttr(&pTypeAttr);

// Tratamos de conseguir las funciones
unsigned int n;
MEMBERID idMember;
FUNCDESC *pFuncDesc = NULL;

BSTR bstrName[1] ;
string methodName;

for (int i=0; i<pTypeAttr->cFuncs; i++) {
      hr = pITypeInfo->GetFuncDesc(i,&pFuncDesc);
```

```
      // Obtenemos los nombres de las funciones a partir de su ID
      idMember = pFuncDesc->memid;
      hr = pITypeInfo->GetNames(idMember,bstrName,1,&n);

      wstocs(methodName,bstrName[0]);

      // Pasamos a mayusculas los nombres de los métodos
      for (int j=0;j<methodName.length();j++)
            methodName[j]=toupper(methodName[j]);

      // Comprobamos el tipo de invocación
      if (pFuncDesc->invkind == DISPATCH_PROPERTYGET) {
            string prefijo="GET";
            methodName= prefijo.append(methodName);   }

      else if (pFuncDesc->invkind == DISPATCH_PROPERTYPUT) {
            string prefijo="PUT";
            methodName= prefijo.append(methodName);   }

      // Añadimos los métodos al conjunto de métodos
      methods.Add(new TMInvoke(this, methodName));
}
```

La clase `TCAutomation` dispone de un único método que, a su vez, permite la invocación de cualquier método miembro de cualquier clase *automation*.

El método constructor de `TMInvoke` recibe como parámetros un puntero a un objeto de tipo `TCAutomation`, otro de tipo string que indica un nombre de método y produce como instancia un objeto, que permitirá realizar las invocaciones de ese método para esa clase, a través del método de invocación `invokeMethod`.

La clase `TCAutomation` redefine la función `TClass::newInstance` para devolver un puntero a una instancia de tipo automation (`TIAutomation`). En `TClass::findMethod` devuelve un puntero a un derivado de `TMethod`, si el nombre del método es alguno de los expuestos a través de la interfaz *IDispatch* del objeto automation.

17.4.2.2 TIAutomation

La clase `TIAutomation` derivada de `TObjectDerivedInstance` y representa a una instancia de una clase automation. Añade funciones miembro para permitir la manipulación de su valor interno por los métodos de su clase. Las instancias de esta clase son creadas exclusivamente por `TCAutomation` a través de su método `newInstance()`. Contiene, como muestra el siguiente código, el nombre de la clase dentro de Oviedo3 a la que pertenece y un puntero IDispatch al objeto DCOM que envuelve.

```
class TIAutomation: public TObjectDerivedInstance
{
public:
      TIAutomation(TCAutomation *c);
      ~TIAutomation(void);
```

```
      std::string getClass(void) {return className;}

      IDispatch* getValue(void) {return valor;}

private:
      std::string className;
      IDispatch* valor;
};
```

17.4.2.3 TMInvoke

Clase concreta derivada de `TMethod` que representa al método `Invoke` de la clase automation. `TMInvoke`, permite llamar a cualquier método automation. `TMInvoke` tiene dos campos privados que contienen el nombre de la clase a la que pertenece y el nombre del método a invocar en ese objeto automation. Tanto el método a invocar como el objeto sobre el que se ejecuta dicho método se pasan como parámetros al constructor de `TMInvoke`.

```
class TMInvoke : public TPrimitiveMethod
{
public:
      TMInvoke(TCAutomation* c, string _name);
      string getClass(void) { return className; }
      string getName(void) { return name; }
      void invokeMethod(TThread &th, instanceID id, const CDSet<string
*> &params, const string &lvalue);
private:
      string className,name;
};

TMInvoke::TMInvoke(TCAutomation* c, string _name) : TPrimitiveMethod()
{
      className=c->getName();
      name=_name;
}
```

La complejidad de esta clase se centra en el método `TMethod::invokeMethod`, cuyo funcionamiento pasamos a describir[*]:

- Obtener el IID del método automation a invocar.
- Obtener el tipo de invocación del método automation.
- Determinar el número y tipo de los parámetros del método automation.
- Comprobar la validez de los parámetros introducidos por el usuario.
- Comprobar la validez del valor de retorno.
- Preparar la llamada al método de la Librería COM *IDispatch::Invoke* con los parámetros convenientemente tratados para que sean compatibles con los esperados por dicho método.
- Comprobar el resultado retornado por la función *IDispatch::Invoke.*
- Adaptar los valores retornados a Carbayonia.

[*] Dada su extensión, el código completo puede ser consultado en el Apéndice B.

Todo esto se realiza sobre el contexto actual del thread, por lo que la ejecución continuará en la siguiente instrucción a la que invocó este método.

17.4.2.4 Conversión de tipos, parámetros y valor de retorno

Los tipos elegidos para la interoperabilidad han sido los primitivos básicos, enteros, reales, cadenas y boléanos. Automation basa la comunicación de parámetros en un registro especial denominado VARIANT como se mostró en la sección 16.1.2.2. Para cada tipo se realizan dos conversiones, una en cada sentido, así, inicialmente tenemos las funciones:

```
void IntegerToVariant (TIInteger* entero, VARIANTARG* pvar)
void VariantToInteger (VARIANTARG* pvarSrc, TIInteger* entero)
void FloatToVariant (TIFloat* flotante, VARIANTARG* pvar)
void VariantToFloat (VARIANTARG* pvarSrc, TIFloat* flotante)
void StringToVariant (TIString* cadena, VARIANTARG* pvar)
void VariantToString (VARIANTARG* pvarSrc, TIString* cadena)
void BoolToVariant (TIBool* booleano, VARIANTARG* pvar)
void VariantToBool (VARIANTARG* pvarSrc, TIBool* booleano)
```

Para la conversión de parámetros en la llamada de la invocación definimos dos funciones `param2Variant` y `transferParams2VariantArgs`. La primera convierte un parámetro normal de Carbayonia a un tipo `Variant`, veamos un extracto de su código.

```
VARIANTARG param2Variant(TThread &th, const CDSet<string *> &params,
const CVector<PARES> *argTypes, int i)
{
VARIANTARG varg;
::VariantInit(&varg);

std::string aux1 = (*argTypes)[i].tipoEnCarbayonia;
VARTYPE           aux2 = (*argTypes)[i].tipoEnAutomation;

switch (aux2) {

case VT_I4: case VT_UI1: case VT_I2: case VT_I1:
case VT_UI2: case VT_UI4: case VT_INT: case VT_UINT:
      {
      if (aux1 == "INTEGER") {
      TRef &p = th.getRef(*params[i]);
      TIInteger* entero =
                  dynamic_cast<TIInteger*> (p.getInstancePtr());
      IntegerToVariant(entero,&varg);
      }
      else {
            OleUninitialize();
                  throw TParamError;
            }
      }
      break;

case VT_R4: case VT_R8:
                  . . . . . .
```

La segunda función es un bucle de llamadas a la primera para cada uno de los parámetros y prepara la estructura denominada DISPPARAMS que contiene los

Variants a ser enviados en la invocación. A continuación se muestra un resumen de su código.

```
void transferParams2VariantArgs(TThread &th, const CDSet<string *>
&params, const CVector<PARES> *argTypes, DISPPARAMS &param, DISPID*
mydispid)
{
      VARIANTARG rgvarg[numParam];

      for (int i=0; i<numParam; i++) {
            ::VariantInit(&rgvarg[i]);
            rgvarg[i]=param2Variant(th,params,argTypes,i);
      }
      param.cArgs = numParam ;      // Número de argumentos
      param.rgvarg = rgvarg ;       // Argumentos

      // Si los párametros corresponden a una propiedad de escritura
      if (mydispid[0] == DISPID_PROPERTYPUT) {
            param.cNamedArgs = numParam ;
            param.rgdispidNamedArgs = mydispid; }
      else {
            param.cNamedArgs = 0 ;
            param.rgdispidNamedArgs = NULL;}
}
```

Una vez invocada la función anterior, en el parámetro `param` de tipo DISPPARAMS disponemos de los parámetros listos para ser utilizados en la invocación al método `invoke` de la interfaz IDispatch.

```
hr = (self->getValue())->Invoke(
                           dispid,
                           IID_NULL,
                           GetUserDefaultLCID(),
                           pFuncDesc->invkind,
                           &param,
                           &varResult,
                           &excepinfo,
                           NULL) ;
```

Una vez invocada la función deseada, en `varResult` tenemos el resultado, a continuación a través de una sección case se realiza la conversión del tipo de retorno. Veamos un extracto de su código:

```
 switch (varResult.vt) {

 case VT_I4: case VT_UI1: case VT_I2:
 {
       checkReturnValue(th, "INTEGER", lvalue);
       instanceID integerID =
areaInstancias.newInstance("INTEGER");
       TIInteger *pInteger = dynamic_cast<TIInteger *>
(areaInstancias.getInstance(integerID));
       VariantToInteger(&varResult,pInteger);
       th.getRef(lvalue).setInstance(integerID);
 }
       break;
```

17.4.2.5 Localización

En cuanto a la localización, como ya se comentó anteriormente, hemos ampliado la implementación del método `getClass()` de la clase `TClassArea`. A continuación exponemos el código completo autoexplicativo:

```
TClass *TClassArea::getClass(const std::string &nombre)
{
// Si el usuario define en lenguaje Carbayón una variable de un tipo
// Automation, el programa es capaz de tratar dicho tipo como si fuera
// uno más de los definidos en Carbayonia

// Buscamos si la clase existe en Carbayonia
for (int i=0; i<clases.NumItems(); i++)
 if (clases[i]->getName() == nombre)
  return clases[i];

// Si no existe en Carbayonia, buscamos si existe en Automation.
// Intentamos crear el objeto y obtener un puntero a su interfaz
IDispatch

if (isAutomation(nombre)) {  // Es un nombre Automation

      // Todo tipo de servidores
      DWORD clsctx = CLSCTX_SERVER ;

      // Obtenemos CLSID

      // Quitamos el '_' y ponermos un '.' en el nombre de la clase
      string aux=nombre;
      SetDot(aux);

      // Buscamos el Identificador Universal de Clase clsid
      wstring progid;
      cstows(progid,aux);
      CLSID clsid ;
      HRESULT hr = ::CLSIDFromProgID(progid.c_str(), &clsid) ;

      // Creamos el componente y pedimos un puntero a IDispatch
      IDispatch* pIDispatch = NULL ;
      hr = ::CoCreateInstance(clsid,
                              NULL,
                              clsctx,
                              IID_IDispatch,
                              (void**)&pIDispatch) ;

      // La clase existe en Automation
      TCAutomation *autom = new TCAutomation(nombre);

      // Incluimos la nueva clase en el area de clases
      clases.Add(autom);

      // Liberamos recursos
      pIDispatch->Release();

      // Devolvemos la clase
      return autom ;
}
// La clase no existe ni en Carbayonia ni en Automation
throw TClassNotFound;
return NULL;
```

```
}
```

17.4.3 Validación

El siguiente código en lenguaje Carbayón, ha sido utilizado para validar esta parte de la implementación del prototipo. En él se declara una variable de tipo `word_application` denominada `wd`, posteriormente se invoca a uno de sus métodos `CheckSpelling`, pasándole como parámetro la cadena `s` y obtenemos como resultado en `b` un valor booleano.

```
INSTANCES
         b:bool;
         wd:word_application;
         s:string('hola');
         sYES:string('SI');
         sNO:string('NO');
         c:constream;
CODE
        wd.CheckSpelling(s):b;
        jfd b, labelYES;
        c.write(sNO);
        exit;
labelYes:
        c.write(sYES);
        exit;
ENDCODE
```

La clase word_application es un envoltorio de la clase `Word.Application` de la plataforma DCOM. Se puede observar que se utiliza de forma transparente y sencilla, desde Carbayonia, en su uso no se realiza ningún tipo de manipulación especial, como, inicialización, apertura o cierre de la plataforma, o de algún objeto concreto. Simplemente es una clase más disponible en Oviedo3. Al igual que con esta clase de DCOM ocurre con el resto de las clases que pertenecen a la plataforma. Si bien, en nuestro prototipo concreto, la invocación a métodos queda restringida a los tipos básicos.

17.5 Servicios Web -> Oviedo3 <- Servicios Web

Nos encontramos con las mismas restricciones comentadas en la sección 16.3.4.2 de no estado y no contexto, esto significa que no mantendremos referencias a instancias de clase, sino que se realizarán llamadas directamente a las instancias recién creadas, como si de un método de clase se tratase. El protocolo que invocación que vamos a implementar es el de tipo POST/GET, donde las llamadas se pueden construir directamente a partir de las URL solicitadas al servidor Web y el resultado se obtiene en un formato XML sencillo. En el caso de nuestro prototipo dicho servidor Web se encuentra en la máquina *fdomingu.escet.utjc.es* en el puerto *8888*, así las peticiones de Servicios Web pueden ser de la forma:

```
http://fdomingu.escet.urjc.es:8888/paquete/clase/metodo?param1=valor1
```

De esta forma, para invocar el método setText de la clase window de Oviedo3 con el parámetro titulo=Prueba, se realizaría la siguiente invocación:

```
http://fdomingu.escet.urjc.es:8888/OVIEDO3/WINDOW/SETTEXT?TITULO=Prueba
```

Para el caso de Oviedo3, el nombre de los parámetros no tiene efecto, se realizará la invocación en función del orden, por otra parte, en otros sistemas si se tiene en cuenta. El paquete OVIEDO3, es un paquete especial que representa a todos los elementos del sistema integral orientado a objetos Oviedo3.

La organización de esta sección es análoga a la de la sección anterior, diseño arquitectónico, implementación y validación.

17.5.1 Diseño Arquitectónico

En este caso el diseño arquitectónico no sigue fielmente el modelo arquitectónico de la colaboración genérica, sin embargo, respeta su misma filosofía, en la que tenemos un servidor del protocolo utilizado, en este caso HTTP, que viene dado por el método `SirveSocket()` que es un hilo lanzado por el hilo principal asociado al método `InetServer()` y se encarga de escuchar peticiones TCP/IP en un determinado puerto. Cuando `SirveSocket` recibe un paquete a través del protocolo HTTP, lo convierte en una cadena string, crea una instancia de la clase Despachador y llama al método Procesar pasándole dicha cadena, el cual se encargará de gestionar la petición de invocación. En la sección de implementación describiremos este método más en detalle.

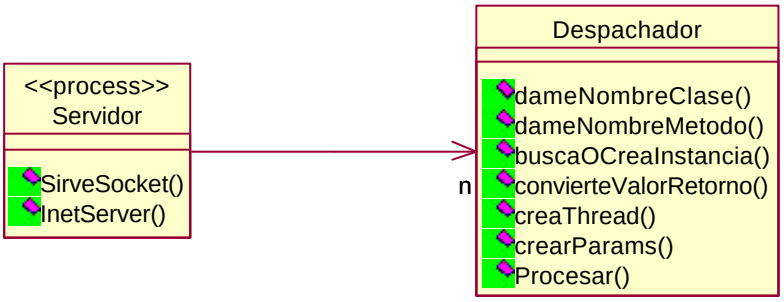


Ilustración 17.12: Servicios Web -> Oviedo3 <- Servicios Web

Hemos tomado la decisión de diseño de no definir las clases abstractas `MAExternalDispatcherProtocol` y `MAExternalDispatcher`, tal y como se aconseja en la colaboración genérica del modelo arquitectónico, ya que sólo vamos a interoperar con una plataforma, en este caso los Servicios Web.

Por otro lado los servicios de localización e introspección no se han implementado, con lo que los clientes de los Servicios Web deben conocer el nombre de las clases así como sus signaturas. Esto no se considera un problema, puesto que los conoceremos en todo momento y si son necesarios se podrán crear manualmente para casos puntuales. Además, con el objetivo de experimentar y valorar el prototipo, estos aportaban complejidad al mismo y poca información tanto cualitativa como cuantitativa.

17.5.2 Implementación

El método `Procesar()` se encarga de recibir la petición en forma de cadena de caracteres, extraer la información necesaria para construir la invocación, realizar la llamada y devolver el resultado de la misma, en el formato definido por el protocolo a través de una cadena de caracteres de retorno. Veamos el código que contiene dicho método y procederemos a describirlo detalladamente, para ello, lo separaremos en dos partes, una inicial que se encarga de extraer la información de la cadena de caracteres, tal como el nombre de la clase, del método, los parámetros, el tipo de retorno y una segunda que procesando la información anterior, se encarga de construir la invocación y finalmente devolver el valor de retorno del método interno invocado.

Comencemos por la primera parte, en ella, tal y como se muestra a continuación, hay cuatro métodos que toman como parámetro la cadena de caracteres `sPagina`, que contienen codificada la invocación realizada por el sistema externo. Estos son:

- `dameNombreClase`, nos devuelve el nombre de la clase sobre la que se desea realizar la invocación.

- `dameNombreMetodo`, que devuelve el nombre del método sobre el que se desea realizar la invocación

- `dameCadenaParametros`, devuelve un elemento del tipo `TParametros` que está formado por una colección de parámetros donde cada un contiene, un nombre, un tipo y su valor.

- `dameTipoValorRetorno`, nos indica de qué tipo es el valor que nos devuelve la función.

```
string despachador::Procesar(string sPagina)
{
// Obtenemos el nombre de la clase
string clase=dameNombreClase(sPagina);

// Obtenemos el nombre del método
string metodo=dameNombreMetodo(sPagina);

// Extraemos los parámetros de la invocación
TParametros p=dameCadenaParametros(sPagina);

// Obtenemos el tipo del valor de retorno
string lvalue=dameTipoValorRetorno(sPagina);
```

Una vez extraída la información de `sPagina`, se procede a construir la invocación. Estos pasos se podrían abstraer según el patrón de diseño *Metodo Plantilla*, ya que básicamente van a ser siempre los mismos para todos los sistemas externos que no tengan estados ni contextos, existiendo otro igual si se desea, para el caso de plataformas con estado. El siguiente código muestra en detalle tal proceso.

```
// Obtenemos un puntero a la clase
TClass *c= areaClases.getClass(clase);

// Creamos una instancia de la clase
instanceID id = areaInstancias.newInstance(clase);
```

```
// Creamos un thread sobre el que se ejecutará la invocación
// inicializamos todos los objetos de parametros
TThread *th = crearThread(p,lvalue);

// Definimos los parámetros para la invocación
CDSet <string *> params= crearParams(p);

// Buscamos el método dentro de la clase a invocar
TMethod *method = c->findMethod(metodo);

// Añadimos el thread al area de thread
// Solamente si es una clase de usuario
if (c->isUserClass()) {AreaThreads.Add(th);}

// Realizamos la invocación
method->invokeMethod(th, id, params, lvalue);

// Obtenemos el valor de retorno del thread
// que es una referencia de nombre "RR"
// la convertirmos a cadena de caracteres
// y lo devolvemos
return convierteValorRetorno(th.getRef("RR"));
}
```

Extraemos del área de clases un puntero a la clase sobre la que vamos a realizar la invocación `c`, posteriormente creamos una instancia, la depositamos en el área de instancias y dejamos su identificador en `id`. A continuación creamos el thread sobre el que deseamos que se realice la invocación, donde crearemos todas las instancias locales necesarias tanto para guardar los valores de los parámetros de `p`, como para guardar el valor de retorno de la función, denominaremos a esa variable con el nombre de `RR`. Seguidamente localizamos en el método dentro de la clase `c` y obtenemos un puntero a este que denominamos method. Llegado a este punto, debemos ver si se trata de una clase de usuario, en tal caso, debemos añadir el thread al área de thread para que sea procesado por el sistema como cualquier otro. En caso contrario, la invocación será inmediata y no será necesario añadirlo. Una vez inicializados todos los valores necesarios para realizar la invocación, procederemos a invocar la operación a través del método `invokemethod`, adjuntándole el thread donde debe ejecutarse, el identificador de la instancia, los parámetros previamente creados en el thread, así como el tipo del valor de retorno. Por último, una vez ejecutado el método, en la referencia `RR` del thread obtendremos el valor devuelto por la operación, dicha referencia se la enviamos a `convierteValorRetorno()` que nos devolverá una cadena con el valor devuelto codificado en XML.

Debe observarse que si se tratase de una invocación con estado, en tal caso se podría extraer del parámetro sPagina un identificador de la instancia sobre la que se desea realizar la invocación y en vez de llamar a la operación `newInstance(clase)` de areaInstancias se llamaría a `getInstance()`.

17.5.2.1 Conversión de tipos

Como se indicó en el Capítulo 11, la conversión de tipos entre las diversas plataformas se deja a elección del arquitecto del sistema. En el caso de nuestro prototipo, hemos definido conversión para los tipos básicos, como son, *booleanos*,

enteros, *reales* y *cadenas de caracteres*. Expresar dichos valores en XML para el protocolo POS/GET de los Servicios Web, resulta muy sencillo, vemos varios ejemplos.

Para devolver 166,386 se crea el siguiente código XML:

```
<?xml version="1.0" encoding="utf-8" ?>
<double xmlns="http://fdomingu/XmlWebServices/">166.386</double>
```

Para devolver un valor booleano cierto, sería este otro:

```
<?xml version="1.0" encoding="utf-8" ?>
<boolean xmlns="http://fdomingu/XmlWebServices/">true</boolean>
```

Y para devolver la cadena de caracteres “Hola”:

```
<?xml version="1.0" encoding="utf-8" ?>
<string xmlns="http://fdomingu/XmlWebServices/">Hola</string>
```

17.5.2.2 Creación del thread y los parámetros

Quizás la creación del thread donde se define el contexto de ejecución y las referencias a los parámetros para invocación sea la parte de código más específico de Carbayonia.

```
TThread * despachador::crearThread(TParametros &p,string &lvalue)
{
TThread *th = new TThread;
RefSet *localRefs=new RefSet(10,10);

// Inicializamos las referencias de los parametros
for (unsigned int i=0; i<p.NumItems();i++)
{
      // Creamos la referencia
      TRef *ref = new TRef(p[i].nombre,p[i].tipo);
      // Creamos la instancia
      ref->setInstance(areaInstancias.newInstance(p[i].tipo));
      // Ponemos el valor
      // De momento todos los parametros son de tipo STRING
      TIString *pString = dynamic_cast<TIString *>
                              (ref->getInstancePtr());
      pString->valor.erase();
      pString->valor.append(p[i].valor);
      // Añadimos la referencia
       localRefs->Add(ref);
}
// Definimos el valor de retorno
ref=new TRef(“RR”,lvalue);
// Lo añadimos al conjunto de referencias locales
// del contexto actual
localRefs->Add(ref);

// New context
th->createEmptyContext(localRefs, NULL);

return th;
}
```

La creación del therad de ejecución es necesaria puesto que debemos crear dentro de su contexto las referencias locales que apuntan a las instancias, donde se

guardarán los valores que deben tener los parámetros que se utilizan para la invocación. Además se crea una referencia adicional denominada `RR` donde se almacenará el valor de retorno de la función invocada.

Mientras que en campo params de la invocación, véase de nuevo la llamada a la invocación,

```
method->invokeMethod(th, id, params, lvalue);
```

se guardarán simplemente, los hombres de las referencias donde se encuentran los valores de los parámetros. Véase el código del método crearParams, que se encarga de ello.

```
CDSet <string *> *params despachador::crearParams(TParametros
&p)
{
CDSet <string *> *params=new DCSet(10,10);
// Parameters as an array of string with names of before
references
      for(unsigned int i=0;i<p.NumItems();i++)
      {
            string *s=new string(p[i].nombre);
            params->Add(s);
      }
return params;
}
```

17.5.3 Validación

Para comprobar que se realizan adecuadamente las invocaciones a los elementos internos de Oviedo3, hemos construido varias invocaciones. La primera, manualmente, a través de un navegador. La segunda, automáticamente definiendo un WSDL que describe la invocación. Y la tercera y última, en tiempo de programación en Java. Para comprobar el funcionamiento de la misma, hemos creado una clase en carbayón denominada `TempServer` con un método `ConvertTemperature` que recibe un valor real en grados centígrados y devuelve su conversión a grados Farenheit.

17.5.3.1 Invocación manual

Los Servicios Web son tan sencillos que pueden realizarse invocaciones a los mismos simplemente a través de un navegador Web, donde en la URL se indica, la clase, el método y los parámetros. Estos últimos siguiendo el formado de comunicación de parámetros CGI.

Para realizar una invocación con valor de parámetro 5, se escribiría la siguiente URL absoluta.

```
http://fdomingu.escet.urjc.es:8888/OVIEDO3/TEMPSERVER/CONVERTTEMPERATURE
?nFarenheit=5
```

La respuesta obtenida en el navegador en formato XML, en caso de no haber ningún error, sería:

```
  <?xml version="1.0" encoding="utf-8" ?>
```

```
<double xmlns="http://fdomingu/XmlWebServices/">-15</double>
```

Esta técnica es sencilla, rápida y útil para probar Servicios Web de forma manual, aunque no aporta ningún beneficio ya que pretendemos obtener comunicación entre procesos y no entre humanos y procesos como es el caso.

17.5.3.2 Invocación estática

Para realizar una invocación estática, simplemente habrá que añadir la referencia Web al entorno IDE a través de su descripción WSDL. En nuestro caso hemos utilizado el IDE de .NET y el lenguaje Visual Basic.NET, aunque se podría haber utilizado otro lenguaje cualquiera de dicha plataforma, como C++, C#, Eiffel.NET, Perl.NET, etc.

El siguiente ejemplo, a través de una descripción WSDL se ha importado dentro del IDE.NET un servicio denominado `Oviedo3.TempServer`, a partir de este momento se pueden crear instancias de dicho servicio e invocar sus métodos.

```
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Button1.Click
        Dim s As New Oviedo3.TempServer()
        Label1.Text = s.ConvertTemperature(5)
    End Sub
```

En el ejemplo anterior, al hacer click sobre el botón `Button1`, se realizará una llamada a `ConvertTemperature` con el valor `5` y el valor resultante aparecerá en el texto de la etiqueta `Laberl1`.

Quizás esta sea la forma más cómoda y sencilla de utilizar los Servicios Web, si bien, estos se pueden invocar de forma dinámica en tiempo de ejecución, como se muestra a continuación.

17.5.3.3 Invocación dinámica

A veces, no se conoce a priori el servicio sobre el que se va a realizar la invocación, en tal caso resulta útil poder construir la invocación de forma dinámica en tiempo de ejecución. Para realizar esta prueba, construiremos un sencillo programita en Axis de Java que realiza la misma invocación que en los ejemplos anteriores. Veamos el código.

```
String  endpoint=”http://localhost:8888/OVIEDO3/TempConvert”;
// Creamos el servicio
Service service=new Service();
// Creamos una llamada para dicho servicio
Call call=(Call) service.createCall();
// Definimos la URL destino
Call.setTargetEndpointAddress(new java.net.URL(endpoint));
// Definimos la operación del servicio a invocar
Call.setOperationName(“ConvertTemperature”);
// Indicamos los parámetros
Call.addParameter(“nFarenheit”,XMLType.XSD_INT,ParameterMode.IN);
// Definimos el tipo del valor de retorno
Call.setReturnType(XMLType.XSD_INT);
Double nFarenheit=new Double(5);
// Realizamos la invocación pasandole un vector de objetos
// inicializados como parámetros.
Double ret=(Double) call.invoke(new Object[]{nFarenheit});
```

Como podemos apreciar, en este caso, son necesarios una serie de pasos adicionales. Se define un servicio genérico denominado `service`. A partir de él se crea una llamada, `call`, se le indica una URL destino, `endpoint`, se definen los parámetros con el método `addParameter`, indicamos un tipo de retorno, inicializamos los parámetros y realizamos la invocación con `call.invoke`.

17.6 Servicios Web -> DCOM <- Servicios Web

A continuación vamos a combinar el resultado de las dos secciones anteriores con el fin de obtener una comunicación virtual entre los Servicios Web y los objetos DCOM de forma transparente.

En primer lugar vamos a utilizar un ejemplo que nos servirá como muestra del funcionamiento del proceso de interoperabilidad virtual entre ambas plataformas externas. Con este fin, realizaremos una invocación al método `CheckSpeller` de la clase DCOM denominada `Word.Application` con la cadena "Hola" como parámetro. Es posible realizar esta invocación de varias formas, de las cuales mostramos dos.

La primera sería lo que anteriormente denominamos de forma manual, y se lleva a cabo desde la línea de direcciones del navegador:

```
http://fdomingu.escet.urjc.es:8888/WORD/APPLICATION/CHECKSPELLING?word=Hola
```

La segunda responde a la técnica dinámica de invocación, donde utilizaremos la API Axis de Java para construir llamadas hacia Servicios Web.

```
String u="http://fdomingu.escet.urjc.es:8888/WORD/APPLICATION";
// Creamos el servicio
Service service=new Service();
// Creamos una llamada para dicho servicio
Call call=(Call) service.createCall();
// Definimos la URL destino
Call.setTargetEndpointAddress(new java.net.URL(u));
// Definimos la operación del servicio a invocar
Call.setOperationName("CheckSpelling");
// Indicamos los parámetros
Call.addParameter("sHola",XMLType.XSD_STRING,ParameterMode.IN);
// Definimos el tipo del valor de retorno
Call.setReturnType(XMLType.XSD_BOOL);
String sHola=new String("Hola");
// Realizamos la invocación pasandole un vector de objetos
// inicializados como parámetros.
Boolean ret=(Boolean) call.invoke(new Object[]{sHola});
```

Esta invocación es similar a la vista en la sección anterior, salvo en que se utiliza otro Servicio Web y otro método del mismo. En este último caso, tanto el método como el servicio son una clase y su método de la plataforma DCOM. Una de las ventajas de este sistema radica, en que no es necesario que el usuario, que realiza la llamada desde su sistema sea consciente de, que está utilizando una plataforma diferente a la de los Servicios Web.

En la Ilustración 17.13, pasamos a mostrar un diagrama de secuencia donde se detalla el mecanismo interno de funcionamiento del proceso adaptado a este ejemplo concreto.

La clase Servidor a través del método InetServer se encarga de interceptar la llama realizada de alguna de las formas expuestas anteriormente. Este envía la información de la invocación codificada en HTTP en una string, al método Procesar de la clase `Despachador` que se encarga de obtener el nombre de la clase, a la que va dirigido el mensaje, el nombre del método, los parámetros enviados y el tipo del valor de retorno. A partir de ahí se solicita al área de clases la clases `Word_Application`, la cual, al no localizarla, llamará al constructor `TCAutomation` con el nombre de la clase, a su vez y utilizando la introspección de DCOM, el constructor crea todos los métodos accesibles a esta clase y la nueva instancia creada, que hace de envoltorio del objeto DCOM a través, como se vio anteriormente del puntero IDispatch. Una vez acabada la construcción de la nueva clase, esta será añadida al área de clases. En estos momentos ya está disponible la clase dentro de Carbayonia. Seguidamente se llama a dicha clase para crea una instancia, del la misma, a través del método `newInstance` que a su vez llama al constructor de la `TIAutomation` con el nombre de la clase.

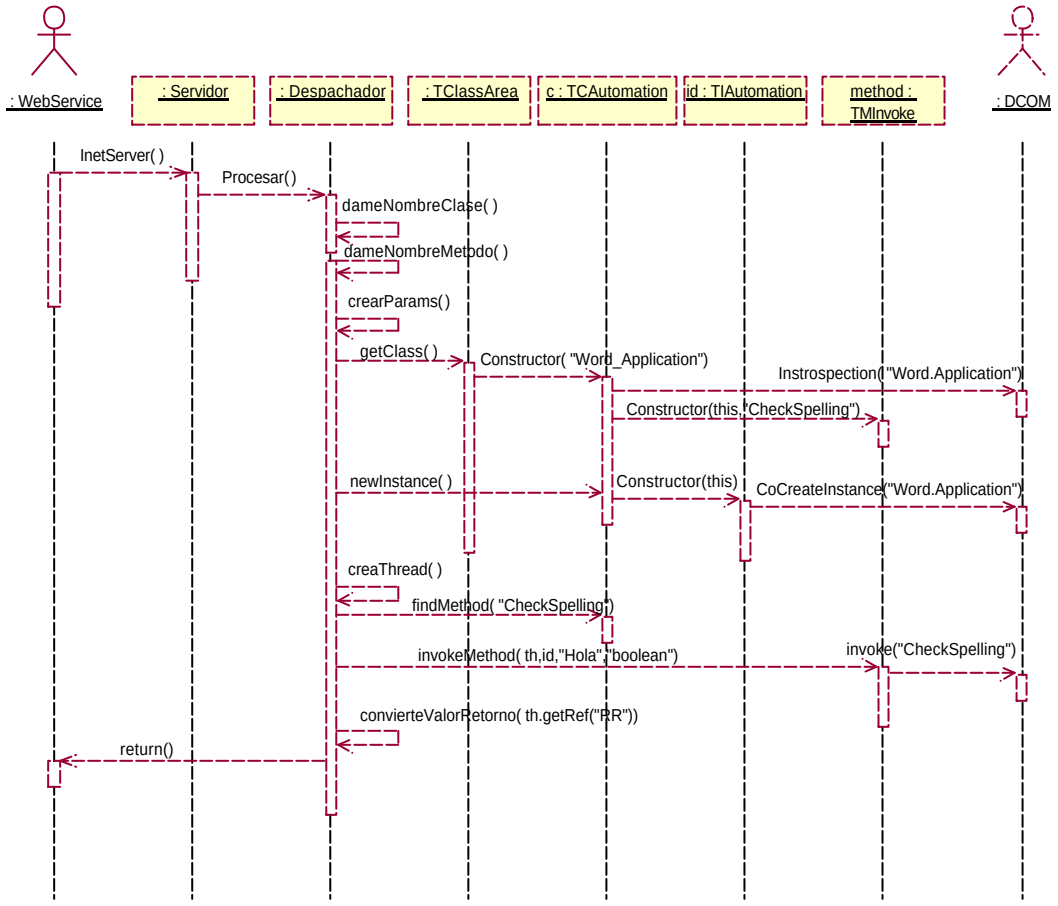


Ilustración 17.13: Servicios Web -> DCOM <- Servicios Web

El constructor de `TIAutomation` se encargará de crear la instancia en la plataforma DCOM a través de la función `CoCreateInstance()` y guardará un puntero a el que el permitirá en todo momento comunicarse e identificar a la instancia en DCOM. A continuación, dentro del método Procesar de Despachador se creará un contexto sobre el que se debe ejecutar la invocación del método, es decir los parámetros y los valores de retorno, para ello se llama a la función crea`Thread()`. Una vez creado el contexto se busca el método `CheckSpelling` dentro de la clase `c` a través del método `findMethod`, anteriormente creada y a continuación se realiza la invocación de dicho método, a través del método `invokeMethod()` pasándole como parámetro, el contexto de ejecución, los nombres de los parámetros y el tipo del valor de retorno. Este método recibe la invocación, convierte los parámetros y realiza la llamada a `invoke("CheckSpelling")` del objeto DCOM al cuál apunta la instancia sobre la que se desea ejecutar el método. Finalizada la invocación en la referencia `RR` del contexto `th`, creado previamente se encuentra el valor de retorno, lo convertimos a formato de datos XML y lo devolvemos como una string que a su vez será devuelto al cliente de Servicios Web como una respuesta HTTP.

17.7 Conclusiones

Llegado a este punto, podemos afirmas que el modelo arquitectónico planteado en esta tesis permite realizar, el diseño de arquitecturas de forma sencilla y rápida que dan lugar a implementaciones de sistemas software, donde dos o más plataformas middleware distribuidas heterogéneas puede comunicarse entre sí de forma transparente y no intrusiva, a través de máquinas abstractas reflectivas orientadas a objetos. Una vez validado nuestro modelo arquitectónico, comprobando que se pueden diseñar y construir sistemas concretos, procederemos a verificar que los sistemas obtenidos cumplen los requisitos deseados y comentados en el Capítulo 10, obedeciendo las restricciones de diseño expuestas en el Capítulo 11. En el siguiente capítulo nos encargaremos de esto, y de comparar los resultados obtenidos con otras técnicas, que se están aplicando actualmente, para resolver el problema de la heterogeneidad, que ya han sido estudiadas en el Capítulo 8.

Capítulo 18:
Evaluación del modelo

Una vez que hemos expuesto nuestro modelo, instanciando varios diseños de acuerdo con él y creado e implementado un prototipo, ha llegado el momento de la evaluación del modelo arquitectónico, que consiste, por un lado, en una verificación y validación de que todos los requisitos expuestos en el Capítulo 10 han sido considerados en el modelo arquitectónico y, por otro, en una comparación cualitativa con las soluciones aportadas por las otras tecnologías vistas en el Capítulo 8.

Contra lo que cabría esperar, el objetivo de esta tesis no está centrado en el rendimiento obtenido con los sistemas construidos a partir de los diseños arquitectónicos realizados siguiendo el modelo arquitectónico expuesto en la misma. Esto es debido a la gran variedad de máquinas abstractas y plataformas tanto software como hardware, sobre las que se puede aplicar el modelo arquitectónico. Por ejemplo, si elegimos como máquina abstracta la CLR de .NET, obtendremos, debido a su capacidad de compilación *Just in Time* (JIT), mucho más rendimiento que si utilizamos la máquina abstracta de Smalltalk, que utiliza un intérprete avanzado [Cueva98], o, como en el caso de nuestro prototipo la máquina abstracta Carbayonia de Oviedo3, que utiliza un intérprete puro [Cueva98].

18.1 Verificación

El proceso de verificación consiste en comprobar que los requisitos deseados, expuestos en el Capítulo 10, han sido considerados a la hora de construir el modelo arquitectónico, con el fin de que los sistemas diseñados e implementados a partir de él los contengan.

18.1.1 Heterogeneidad

Los principales obstáculos con los que se encuentran los sistemas distribuidos y que por tanto, deben salvar son los referentes a todo tipo de barreras heterogéneas. Nuestro modelo arquitectónico está definido para permitir heterogeneidad hardware y software de todo tipo, puesto que esta es una característica inherente a toda máquina abstracta, como se vio en el Capítulo 4. Ahora bien, este trabajo está centrado en superar el problema concreto de la heterogeneidad en sistemas middleware RMI, como se ha venido mostrando en los últimos cuatro capítulos.

18.1.2 Transparencia

Con este requisito se pretende ocultar la naturaleza heterogénea de las plataformas middleware que interoperan entre sí.

18.1.2.1 De acceso

Las aplicaciones clientes accederán a los objetos locales, remotos tanto de la misma plataforma como de diferentes plataformas de la misma manera. Análogamente, las aplicaciones servidoras serán accedidas uniformemente. Para conseguir esto, en los diseños del modelo, se ha creado la clase `MAExternalSystem`, sección 15.1, para la comunicación <- MA ->. Para la comunicación -> MA <-, se pueden utilizar las clases `MAExternalDispatcher` y `MAExternalDispatcherProtocol`, sección 15.2.

18.1.2.2 De ubicación/localización

El modelo ha sido diseñado para permitir de forma transparente por parte de todos los sistemas implicados, tanto externos como de la propia máquina, la localización de objetos. En este sentido, son las clases `MAExternalSystem`, sección 15.1, con su método `findClass` en la comunicación <- MA ->, y la clase `MAExternalLocator`, sección 15.2, para la comunicación -> MA <-, las encargadas de que dicha transparencia sea efectiva.

18.1.2.3 De concurrencia

Este requisito lo vamos a obtener de forma inseparable a partir de las técnicas concurrentes utilizadas por la máquina abstracta elegida. De tal forma que se puede acceder a los diversos objetos de otras plataformas concurrentemente, como la propia naturaleza de los sistemas distribuidos impone.

18.1.2.4 De escalado

Este requisito no siempre se cumplirá, como indicaremos más adelante. Aún así, en el caso de que se disponga de ella nos permitirá repartir la carga del sistema en varios equipos diferentes.

18.1.2.5 Heredada

El modelo permite que, hasta cierto punto, las propiedades de transparencia de las plataformas externas y de la máquina virtual elegida puedan ser utilizadas. De esta

forma podemos tener: transparencia de replicación de fallo, de movilidad, de persistencia y de migración.

18.1.3 Tecnología no intrusiva

El diseño de nuestro modelo arquitectónico ha sido ideado con el fin de que los sistemas externos, tanto clientes como servidores, no deban ser modificados para que interoperen entre sí.

18.1.4 Multiplataforma

Este requisito es inseparable de las máquinas abstractas, con lo que nuestro modelo lo recoge de forma inmediata.

18.1.5 Sencillo y rápido de instalar

La instalación de los sistemas desarrollados consistirá en instalar la máquina virtual, junto con los ensamblados de los programas para la misma que se hayan desarrollados.

18.1.6 Sin curva de aprendizaje

Los programadores no necesitarán aprender nada nuevo acerca de los sistemas distribuidos externos. Seguirán programando para los sistemas que eran habituales para ellos. Esta es una consecuencia del requisito de transparencia.

18.1.7 Extensibilidad

La extensibilidad elemental consistirá, simplemente, en heredar de varias clases definidas en el modelo. Desde el punto de vista <- MA -> dichas clases serán `MAExternalSystem`, sección 15.1. Desde el punto de vista -> MA <-, las clases de las que heredar serán `MAExternalDispatcher`, `MAExternalDispatcherProtocol`, `MAExternalLocator` y `MAExternalIntrospection`, sección 15.2.

18.1.8 Escalabilidad

El requisito de escalabilidad podrá superarse siempre que la máquina virtual subyacente consienta la distribución de sus servicios. Esto nos permitirá repartir la carga del sistema entre varios equipos diferentes y, así poder resistir demandas de rendimiento y recursos superiores.

18.1.9 Eficiencia

El modelo arquitectónico basa su propiedad de eficiencia en un modelo común de datos, el de la máquina abstracta, sobre el que se asienta para realizar las conversiones entre los diversos sistemas. De esta forma, se maximiza la eficiencia en

cuanto a transformación de datos. Por otro lado, la máquina elegida determinará cualitativamente en este requisito. Si se elige una máquina con compilación JIT, se obtendrán mejores resultados que si se utiliza una máquina interpretada.

18.1.10 Automatizado

Gracias a la capacidad de introspección de todos los sistemas participantes, el modelo permite que los sistemas construidos con el creen de forma automática los objetos envoltorios que actuarán de proxies de los objetos externos.

18.1.11 Fácil de gestionar

El modelo no plantea ninguna estructura para gestionar los sistemas desarrollados. Esto queda abierto al arquitecto encargado de construir el sistema. Aunque dada la característica de automatización, inicialmente no resulta necesario ya que los envoltorios se crearán automáticamente a partir de la información obtenida a través de la introspección, sin necesidad de ninguna intervención humana.

18.1.12 Flexibilidad

La flexibilidad es una consecuencia directa de los diseños de sistemas abiertos, como es el caso de nuestro modelo. De tal forma que, en función de los grados de reflectividad que apliquemos en la máquina elegida, a saber, introspección, estructural o de comportamiento, esta característica será mayor o meno. No obstante, nuestro modelo permite de una forma sencilla ser ampliado, no solo con nuevas plataformas de interoperabilidad sino con diferentes tecnologías de comunicación entre procesos, de integración de datos, técnicas de monitorización, administración y auditoria.

18.1.13 Uniformidad

Si la máquina virtual está diseñada para permitir la orientación a objetos, nuestro modelo nos permitirá obtener una uniformidad de objetos desde dentro de la misma. El motivo es que dispondremos de objetos primitivos, reflectivos, de usuario y externos, y todos ellos se manipularán uniformemente. Además, si los sistemas externos son orientados a objetos, se mantendrá dicha uniformidad externamente. Pero, por el contrario, y como ocurre con los Servicios Web, si la plataforma no es orientada a objetos, se romperá la uniformidad en dicha plataforma.

18.1.14 Integrabilidad

Determinados problemas de integración de aplicaciones de negocios pueden plantearse como cuestiones de interoperabilidad, de tal forma que los sistemas implementados con nuestro modelo permitirán integrar sistemas externos al interoperar con ellos.

18.1.15 Interoperabilidad completa

Nuestro modelo permite que dentro de la propia máquina se pueda disponer de un sistema de desarrollo, aportando interoperabilidad completa. Es decir, los desarrolladores podrán interoperar tanto con los elementos internos de la máquina como con los externos.

18.2 Validación

Para la validación de nuestro modelo arquitectónico, comprobaremos que este es lícito para construir diseños arquitectónicos a partir de él y, posteriormente, corroborar que con dichos diseños se pueden implementar sistemas completamente operativos y que permiten la interoperabilidad entre tecnologías middleware heterogéneas. Este proceo ya ha sido llevado a cabo en el Capítulo 16 y en el Capítulo 17, respectivamente.

18.2.1 Validación para diseño

En el Capítulo 16 se realizaron varias instancias del modelo arquitectónico para diferentes plataformas, a saber, DCOM, CORBA y Servicios Web. En todas ellas el modelo arquitectónico fue personalizado de forma sencilla, rápida y directa. Aunque el modelo se comportó como era esperado, no necesitando ningún tipo de adaptación o modificación significativa, cada una de las plataformas estudiadas aportaba sus propias idiosincrasias. De este modo, DCOM no permite la federación de localización, CORBA se adapta perfectamente y los Servicios Web, debido a su naturaleza de no estado y no contexto de ejecución, nos obliga a exponerle los objetos, sin dichas propiedades, con las consecuencias que esto conlleva. Esto se podría solventar de forma sencilla, pero en tal caso, nos saldríamos del estándar de Servicios Web.

18.2.2 Validación para implementación

En el Capítulo 17 se implementó una instanciación concreta utilizando la máquina virtual Carbayonia que se enmarca dentro del sistema integral orientado a objetos Oviedo3. Se ha comprobado que los diseños arquitectónicos instanciados a partir del modelo son perfectamente implementables en ambas direcciones, a saber, -> Oviedo3 <-, considerando como plataforma externa los Servicios Web, véase la sección 17.5, y <- Oviedo3 ->, considerando los objetos DCOM como sistema externo, véase la sección 17.4. Una vez validadas ambas comunicaciones, pasamos a comprobar la transitividad en el envío de mensajes, con la implementación del diseño arquitectónico creado. Así, en la sección 17.6, comprobamos y validamos Servicios Web -> DCOM <- Servicios Web.

18.3 Comparativa

Realizaremos una comparativa puramente cualitativa a partir de las características que cumplen todos los sistemas construidos con nuestro modelo

arquitectónico, enumeradas detalladamente en el Capítulo 8, y de la que nuestro prototipo no es más que una de las posibles realizaciones.

En este caso ampliaremos la tabla comparativa del Capítulo 8, añadiendo nuestro prototipo, y comentaremos la tabla resultante, vease Tabla 5: Comparativa con LIIBUS. Como se puede ver en ella, los sistemas construidos con el modelo arquitectónico expuesto en este trabajo, permiten todas las características identificadas en los sistemas estudiados, excepto la de multiprotocolo.

En fin, resuelve la heterogeneidad a nivel de multilenguaje, multiplataforma y multimiddleware RMI. Es abierto y flexible, permite la inteoperabilidad de forma bidireccional. Pueden extenderse nuevos sistemas externos y los envoltorios se realizan automáticamente a través de los mecanismos de introspección. Permite soportar un escalado en la demanda de recursos y, al ser no intruviso, permite que los sistemas heredados y futuros no necesiten tener en cuenta las plataformas externas con las que interoperan.

La característica de multiprotocolo no la resuelve ya que este trabajo está enfocado, principalmente a middlewares con invocación independiente de la ubicación. Sin embargo dada la flexibilidad del modelo, no resultará difícil ampliarlo con otras tecnologías de comunicación entre procesos, integración de datos, coordinación de sistemas, etc., quedando estas dentro de las líneas de investigación futuras.

De la tabla expuesta, los buses middleware son los que aportan una mayor cantidad de características, aunque se trata de sistemas propietarios, en los que la arquitectura detallada es prácticamente desconocida y están formados por sistemas muy complejos y generales que tratan de resolver la integración de aplicaciones en negocios. Además son poco abiertos y no son automatizados. Nuestro modelo, por el contrario, permite construir sistemas que, al ser más especializados y estar basados en máquinas abstractas, resultan más sencillos de manipular y elaborar. De hecho, una gran cantidad de propiedades se obtienen directamente a través de las características inherentes a las máquinas abstractas en las que se basa, como se ha podido apreciar en la sección de verificación de requisitos (18.1).

- Nuestro modelo reúne estructuras tecnológicas que pueden observarse en los sistemas estudiados.
- La propiedad de automatizado, que también utiliza el sistema RMIX a través de sus proxies dinámicos.
- La característica de no intrusivo, que se utiliza ampliamente en el conmutador SoapWitch.
- La comunicación muchos a muchos bidireccional, propia de PolySpin, aunque la forma de llevarla a cabo resulta muy costosa, pues, cada nueva plataforma requiere n implementaciones para los n sistemas previos.
- Finalmente, Harness es un sistema basado en máquinas virtuales al igual que nuestro modelo.

Las características principales a resaltar que no se encuentran de forma explícita dentro de la tabla, comparativa son, la sencillez del modelo, por tanto de los sistemas construidos con el, y su especiailzación en middlewares con invocación dinámica de métodos.

Tabla 5: Comparativa con LIIBUS

	PUENTES		CONMUTADORES					BUSES						TÚNELES				
	JNBRIDGEPR	EXPERTSOFT	REMMOC	RMIX	IIOP.NET	WSIFAPACH	SOAPSWITC	POLYSPIN	POLYLITH	UAN	ESB	ARTIX	LIIBUS	BIZTALK	ATTUNITY	XDMF	HARNESS	DOTS
MULTILENGUAJE	-	*x*	*x*	*x*	*x*	-	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*
MULTIPLATAFORMA	-	-	-	*x*	-	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	-	*x*	*x*	*x*	*x*
MULTIMIDDLEWARE	*x*	*x*	*x*	*x*	*x*	-	*x*	-	-	*x*	*x*	*x*	*x*	*x*	*x*	-	-	-
MULTIPROTOCOLO	-	-	*x*	-	-	*x*	-	-	-	*x*	*x*	*x*	-	*x*	*x*	-	-	-
ABIERTO	-	-			-	-	-	-		-	-	-	*x*	*x*	-	-		-
BIDIRECCIONAL	*x*	*x*	-	-	*x*	-	-	*x*	*x*	*x*	*x*	*x*	*x*	*x*	-	*x*	-	*x*
EXTENSIBLE	-		*x*	-	-	-	-	*x*		*x*	*x*	*x*	*x*	*x*	*x*	-	*x*	
AUTOMATIZADO	-	*x*	-	*x*	-	-	-	*x*	-	-	-	-	*x*	-	*x*	-	*x*	-
ESCALABLE			*x*		-	-	-			*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	
NO INTRUSIVO			-	-	-	-	*x*			*x*	*x*	*x*	*x*	*x*	-	-		-

Capítulo 19:
Conclusiones

En esta tesis se ha identificado el problema de la heterogeneidad entre plataformas middleware distribuidas del tipo invocación independiente de la localización. Se ha definido un modelo arquitectónico basado en máquinas abstractas reflectivas orientadas a objetos para construir arquitecturas a partir de las cuales se pueden desarrollar sistemas que solucionen el problema, cumpliendo una serie de requisitos expuestos en el Capítulo 10.

19.1 Resultados destacables

Para conseguir alcanzar los retos expuestos anteriormente, propondremos nuestro modelo arquitectónico que nos provee las siguientes novedosas contribuciones:

- Una taxonomía de tecnologías de comunicación utilizadas para resolver el problema de la interoperabilidad heterogénea.

- Definición de una máquina abstracta orientada a objetos básica y genérica, de alto nivel de abstracción.

- Una serie de técnicas para añadir reflectividad estructural y de comportamiento a máquinas abstractas.

- La contribución principal que supone LIIBUS, una arquitectura de Sistemas interoperables entre Middlewares heterogéneos usando Máquinas abstractas reflectivas orientadas a objetos.

- Tres instancias bidireccionales de este modelo para las siguientes plataformas: CORBA, DCOM y Servicios Web.

- Un prototipo basado en la máquina abstracta de Oviedo3, denominada Carbayonia, que permitirá la interoperabilidad de Servicios Web a objetos DCOM.

- La idoneidad de las máquinas abstractas como base para construir motores de transformaciones e interoperabilidad.

- Una proposición para un nuevo paradigma de desarrollo del software utilizando máquinas abstractas como base para la construcción de sistemas software.

19.2 Líneas de investigación futuras

Esta tesis sienta las bases de un modelo arquitectónico para la construción de arquitecturas y, posteriormente, los sistemas resultantes de su implementación que permiten la interoperabilidad entre middlewares heterogéneos, del tipo invocación dinámica de métodos, usando máquinas abstractas reflectivas orientadas a objetos. Se dejan abiertas diferentes líneas de investigación para completar y mejorar lo ya existente y desarrollar nuevos elementos. A continuación, se reseñan algunas de las líneas más inmediatas relacionadas con los temas discutidos en este trabajo.

19.2.1 Espacios de nombres

Cada una de las plataformas que interoperan utilizan una notación de nombrado específica para identificar los elementos estructurales que los componen. Cada una define su propio espacio de nombres de forma más o menos explícita. En el Capítulo 16, para cada una de las plataformas estudiadas se han identificado dichas estructuras de nombrado y se han definido mecanismos sencillos de unificación y simplificación. Por ejemplo, con DCOM se realiza la sencilla conversión de Categoria.Clase a Categoria_Clase, donde simplemente se sustituye el "." de DCOM por el "_" interno y expuesto externamente al resto de sistemas. En este sentido, queda abierta toda una línea de trabajos donde se puedan resolver los posibles problemas surgidos por la colisión de nombres de las diversas plataformas o por la utilización, acceso y nombrado de estructuras más complejas que las plataformas. Un buen punto de comienzo para esta línea podría ser la utilización los *Namespaces* de XML, definidos en W3C [W3C99], como mecanismos universales y de conversión entre las diversas plataformas, para la gestión de los diversos y heterogéneos espacios de nombres.

19.2.2 Añadir URI (referencias universales)

En el Capítulo 11 restringimos el ámbito de actuación del diseño de nuestro modelo, descartando los elementos estructurales menos importantes para así poder centrar nuestra atención en las estructuras más significativas de nuestro modelo. No obstante, dichos elementos pueden ser añadidos en futuros trabajos de investigación. De esta forma el uso de las referencias puede ser indagado, en concreto, una buena aproximación consistiría en utilizar las referencias universales URI, definidas por la W3C [W3C05], o cualquier otra estructura que permita y facilite la conversión entre diferentes formatos de las referencias de los diversos middleware que interoperan.

19.2.3 Tipos complejos

Otro criterio de diseño restrictivo descartado inicialmente en el Capítulo 11 consiste en el tratamiento de tipos complejos. Nuestro modelo se centra en los tipos básicos fundamentales, pasados por copia, dejando abierto al creador de las diferentes arquitecturas el uso de tipos complejos. Futuros trabajos de investigación pueden encauzarse en el estudio de los diversos tipos estructurales de las plataformas y cómo podrían unificarse para obtener un modelo de conversión y tratamiento homogéneo entre los varios sistemas middleware heterogéneos.

19.2.4 Ampliación para otras abstracciones de comunicación

Inicialmente, el modelo ha sido desarrollado para las plataformas middleware con invocación remota de métodos que ofrezcan tres servicios fundamentales, a saber, de nombrado, de introspección y de invocación dinámica. Actualmente, para la resolución del problema de la comunicación entre procesos existen una gran variedad de tecnologías emergentes. Quedan abiertas líneas futuras de investigación que estudien estas tecnologías y vean cómo pueden ser adaptadas al modelo desarrollado o, desde otro punto de vista, como adaptar el modelo arquitectónico para que permita la interacción de las tecnologías. Algunas de dichas tecnologías son Active Middleware [Botzer02], Medium [Matougui03], InfoPipes [Blac02, Koster01], Comunicación dirigida por eventos [Frei03, Zeiger01], etc.

19.2.5 Cosificación de la arquitectura para permitir extensiones desde el propio sistema

En el Capítulo 15 se describe la colaboración genérica que supone nuestro modelo arquitectonico, formada por dos partes, <-MA-> y ->MA<-. En ambas, se hacen una serie de indicaciones que nos guiarían en la obtención de diferentes formas de extensibilidad. En este sentido, queda una línea abierta en la que profundizar para la obtención de diversos tipos de extensibilidades. Un caso concreto sería permitir añadir nuevas plataformas middleware a través de clases de usuario de la propia máquina.

19.2.6 Optimización de rendimiento

El modelo arquitectónico proporciona un mecanismo implícito, eficiente, de conversión y equivalencia de tipos de datos entre las diversas plataformas externas. Para ello, considera como modelo común de datos los propios de la máquina abstracta seleccionada para implementar el diseño arquitectónico resultante. Quedan abiertas otras posibles líneas e investigación que permitan aumentar el rendimiento del sistema resultante, independientemente de la máquina abstracta seleccionada.

Un buen punto de partida en este sentido sería optar por la perspectiva de la eficiencia en los procesos de comunicación pues no se ha utilizado ningún mecanismo de optimización. Una línea abierta por investigar podría ser la utilización de cachés de puertos como los utilizados en el sistema DOTS [Bloch01].

19.2.7 Unificar la gestión de excepciones, fallos y errores

La cuestión es que cada máquina abstracta y cada plataforma de interoperabilidad dispone de sus propios mecanismos de gestión de excepciones, fallos y errores. En nuestro modelo queda abierta la forma en la que dicha gestión tiene lugar. Aunque se vislumbra una técnica implícita que consistiría, al igual que con los datos, en utilizar como modelo común de conversión de excepciones, fallos y errores los propios de la máquina seleccionada, quedan abiertos trabajos de investigación orientados a estudiar las propiedades comunes tanto de máquinas como de plataformas, además de añadir al modelo una colaboración genérica adicional que trate de forma explícita, uniforme y unificada dicha gestión.

19.2.8 Fiabilidad, tolerancia a fallos y disponibilidad

Las características de fiabilidad, tolerancia a fallos y disponibilidad no se contamplan directamente en nuestro modelo, si bien esta vendrán heredadas, de la máquina virtual y las plataformas que formen parte del sistema que interopera. Lineas de trabajos futuros pueden considerar explícitamente estas propiedades, homogeneizarlas y añadirlas al modelo arquitectónico.

19.2.9 Seguridad

Nuestro modelo arquitectónico se centra en la comunicación entre procesos y no considera la autentificación ni del remitente ni del receptor ni de los permisos asociados a ellos para, de esta forma, permitir o no dicha comunicación. Aun así, dada la flexibilidad y apertura del modelo, no resultaría dificil añadir cierto grado de seguridad. Por ejemplo, implementando protocolos seguros del tipo SSL o IPSec en la comunicación entre y hacia los Servicios Web. Aunque en futuros trabajos de investigación podrían ampliar el modelo o construir uno paralelo que homogeinice las diversas técnicas y normas de seguridad de las diversas plataformas heterogéneas.

19.3 Campos de investigación relacionados

Los resultados obtenidos con esta tesis pueden ser utilizados en otros campos de investigación donde, de algún modo, se precisa una comunicación entre elementos de diferente naturaleza. A continuación, enumeramos algunos de ellos.

19.3.1 Arquitecturas de integración

El campo tecnológico donde encaja perfectamente el trabajo desarrollado en esta tesis es el de arquitecturas de integración, tanto de negocios (EAI), visto en la sección 6.3.12, como de computación masiva (Grid), visto en la sección 6.3.10. En ambos casos se pretende la combinación de procesos, software, hardware, comunicaciones y estándares de tal forma que den como resultado la integración homogénea de dos o más sistemas, de manera que operen como un único sistema de mayor capacidad de computación o de gestión de recursos. Si bien sendos campos están formados por muchos más elementos estructurales que los encargados de resolver la comunicación

entre plataformas middleware heterogéneas, nuestro modelo arquitectónico puede, por un lado, servir de base para diseños e implementación de sistemas que traten con la interoperabilidad heterogénea a nivel de comunicación entre procesos, y, por otro, como fundamento de modelado para tratar de afrontar los problemas de heterogeneidad identificados en los diversos subsistemas de las arquitecturas con las que se pretenden integrar.

19.3.2 Agentes inteligentes heterogéneos

Se define agente como [Munindar98]:

"Un elemento de computación persistente que puede percibir su entorno, razonar y actuar sobre él, de forma autónoma o en colaboración con otros agentes."

Los conceptos claves de la anterior definición son interoperabilidad y autonomía. De manera que, los agentes se apartan de los objetos convencionales, diseñados para realizar una serie de tareas. Por el contrario, un agente puede negarse a realizar una determinada acción. De tal forma que un agente puede comunicar con otro y decidir qué información recabar o qué acción física llevar a cabo. Para permitir esto, es necesario un *lenguaje de comunicación entre agente*s (ACL), de modo que se pueda comunicar tanto la semántica como la pragmática del mensaje enviado entre agentes heterogéneos.

Inicialmente, surgió una especificación KQML que pretendió ser un estándar. Este dejó de utilizarse debido la cantidad ingente de dialectos que la utilizaban. Porteriormente surgió otro intento de comunicación para permitir la interoperabilidad de agentes, promovido por la organización sin animo de lucro FIPA [Fipa03]. Actualmente, esta especificación está en proceso de adopción por parte del IEEE, aunque la organización en sí ha dejado de funcionar.

Agentes de diferentes vendedores o de diferentes proyectos de investigación no pueden comunicarse entre sí. Esto es debido a dos razones fundamentales, por un lado, los agentes pueden no reconocer los términos utilizados por otro agente y, por otro, la comunicación entre los componentes que forman los propios agentes no se puede llevar a cabo debido a su naturaleza heterogénea. Es por ello que, con intención de resolver el segunda causa de incomunicación, se pretende utilizar los Servicios Web como medio de comunicación.

A nivel de comunicación entre componentes de agentes, nuestro modelo arquitectónico resulta prometedor ya que, en cierto modo, nos soluciona el problema de la heterogeneidad. En tal caso el problema de la comunicación se centra en la terminología tanto semántica como prágmatica utilizada por los agentes. Incluso a este nivel, se podría utilizar nuestro modelo basado en máquinas abstractas reflectivas como punto de inspiración para un sistema que lo resuelva.

19.3.3 Electrodomésticos inteligentes

A medida que los electrodomésticos que nos rodean en casa y en el trabajo aumentan sus capacidades de procesamiento y versatilidad, la posibilidad de interconectarlos ofrece un amplio abanico de retos. En un futuro próximo no solo se

interconectarán para intercambiar información y poder ser controlados sino que podrán colaborar de forma autónoma e independiente de modo asinérgico dando lugar a una nueva área de aplicaciones. Varios trabajos grupos de investigación de todo el mundo y congresos están surgiendo a propósito de estos nuevos retos. En [Mateos00a] se hace un avance de cómo se podría adaptar el modelo expuesto en esta tesis para ser aplicado a la comunicación de electrodomésticos inteligentes y en [Mateos00b] se estudian los servicios que deberían ofrecerse para permitir una relación adecuada entre dispositivos.

19.3.4 Sistemas ubicuos heterogéneos

La computación ubicua es un campo de la tecnología de la información emergente donde se pretende disponer de sistemas software de forma ubicua independientemente del equipo hardware desde el que se tenga acceso. La idea es que la comunicación tenga lugar de forma transparente y que el sistema sea adaptable al contexto del usuario que la manipula [Abowd00]. Esto conlleva implícita la heterogeneidad en hardware, software, comunicación y middleware. Vemos que el trabajo desarrollado en esta tesis resulta un punto de partida idóneo para encauzar estos problemas de interoperabilidad desde la perspectiva de la ubicuidad.

19.3.5 Monitorización y gestión estadística de redes de sistemas heterogeneos/homogéneos distribuidos

Dada la flexibilidad y apertura del modelo arquitectónico expuesto, nos permitirá no solo la construción de sistemas extensibles sino la posibilidad de ampliarlos con diferentes tecnologías de comunicación entre procesos, de integración de datos, tecnicas de monitorización, administración, auditoría, etc. En este sentido, queda abierta toda una linea de investigación para proporcionar todas estas posibilidades al modelo propuesto.

Apéndice A:
Especificación del lenguaje TranquiTal

```
programa:
declaracionClase {MAClass initialize. MAClass addClass: '1'.} |
declaracionClase programa{MAClass addClass: '1'.};

declaracionClase:
"clase" <IDENTIFIER> ["es" <IDENTIFIER>] "inicio" defReferencias*
defmensajes* 'dMen' "fin" {c_MAClassUser new name: '2' value.dMen
do:[:m|c addMethod: m].^c};

defReferencias:
"crea" <IDENTIFIER> "como" <IDENTIFIER> ["con" <INTEGERLITERAL>];

defmensajes:
"mensaje" <IDENTIFIER> "inicio" expresiones 'expr' "fin"
{MAMethodUser new name: '2' value; addExpresion: expr};

expresiones:
expresion {MASeq new initialize addExpresion: '1'. } | expresion
'expr'  "y" expresiones 'exprs' {exprs addExpresion: expr.};

expresion:
 exprAsignacion {'1'} |  exprIf {'1'} | exprNuevo {'1'} |
exprMientras {'1'} |  exprLlamada {'1'} | exprVariable {'1'} |
<INTEGERLITERAL> ;

exprAsignacion:
"pon" expresion "en" <IDENTIFIER>{MAAssign new aExpr: '2';aVar:
(MAVar new name:'4' value).};

exprIf:
"si" expresion "entonces" expresiones "sino" expresiones ","{MAIf
new aExpr:'2'; aExpr1: '4'; aExpr2: '6'};

exprMientras:
```

```
"mientras" expresion "haz" expresiones "," {MAWhile new aExpr:
'2'; aExpr1: '4'};

exprNuevo:
("un"|"una") <IDENTIFIER> 'nId' {MANew new className: nId value.};

exprLlamada:
"resultado de enviar" | "envia") <IDENTIFIER>'mId' ["con"
expresion ("y" expresion)*]  "a" expresion 'eInst' {MACall new
methodName: mId value; exprInstance: eInst.};

exprLlamada:
"dame" <IDENTIFIER> "de" expresion 'eInst' {MACall new methodName:
'2' value; exprInstance: eInst};

exprVariable:
<IDENTIFIER>{MAVar new name: '1' value};
```

Ejemplo de código en el lenguaje TranquiTal:

```
clase TPrueba
 inicio
crea miConexionTCP como conexionInternet
crea iva como real con 16
  mensaje que
   inicio
    envia abrir a miConexionTCP y
    pon algo * 5 + iva / 4 - venia en iva y envia abreSocket
a conexionTCP y
    pon dame velocidad de miConexionTCP en v y
    pon una factura en miFactura y envia calculaIRPF a
miFactura y
    pon resultado de enviar compruebaLujo con iva a
miFactura en esFacturaLujo y
    pon si esFacturaLujo>5 entonces lujoCalculado*20 sino
primeraNecesidad+5, en resultadoIRPF
   fin
fin
```

Apéndice B:
Código de TMethod::InvokeMethod()

```
void TMInvoke::invokeMethod(TThread &th, instanceID id, const CDSet<string
*> &params, const string &lvalue)
{
  // Obtenemos la instancia Automation
  TIAutomation *self = dynamic_cast<TIAutomation
*>(areaInstancias.getInstance(id));

  // Obtenemos el IID del método a invocar y el tipo de invocación

  wstring unicodeName;

  // En el caso de que estemos ante una propiedad de escritura (Put),
  // los argumentos que se pasen al IDispatch::Invoke habrán de ser
  // argumentos con nombre. Por ello, declaramos el array mydispid, que
  // indicará si el metodo a invocar es una propiedad de escritura o no.
  // Igualmente, en el caso de que estemos ante una propiedad, deberemos
  // quitar el "Get" o el "Put" que previamente hemos antepuesto al nombre

  DISPID mydispid[1];
  INVOKEKIND tipoInvocacion;

  if (name.substr(0,3) == "GET") {
        tipoInvocacion=INVOKE_PROPERTYGET;
        cstows(unicodeName,name.substr(3));
  }
  else if (name.substr(0,3) == "PUT") {
        tipoInvocacion=INVOKE_PROPERTYPUT;
        mydispid[0]=DISPID_PROPERTYPUT;
        cstows(unicodeName,name.substr(3));
  }
  else {
        tipoInvocacion=INVOKE_FUNC;
        cstows(unicodeName,name);
  }
```

```
 DISPID dispid ;
 OLECHAR* nameMethod = ::SysAllocString(unicodeName.c_str());
 HRESULT hr = self->getValue()->GetIDsOfNames(IID_NULL,
                                &nameMethod,
                                1,
                                GetUserDefaultLCID(),
                                &dispid) ;

 if (FAILED(hr)) {
       ErrorMessage("Fallo en la llamada a IDispatch::GetIDsOfNames", hr);
       OleUninitialize();
       throw TFailedHR;
 }

 // Necesitamos conocer el número y tipo de los parámetros del método

 // Tratamos de conseguir un puntero a ITypeInfo

 ITypeInfo *pITypeInfo = NULL;
 hr = self->getValue()->GetTypeInfo(0,0,&pITypeInfo);

 if (FAILED(hr)) {
       ErrorMessage("Fallo en la llamada a IDispatch::GetTypeInfo", hr);
       OleUninitialize();
       throw TFailedHR;
 }

 // Tratamos de conseguir la descripción de la función
 unsigned int n, i=0;
 FUNCDESC *pFuncDesc = NULL;

 TYPEATTR *pTypeAttr = NULL;
 hr = pITypeInfo->GetTypeAttr(&pTypeAttr);

 if (FAILED(hr)) {
       ErrorMessage("Fallo en la llamada a ITypeInfo::GetTypeAttr", hr);
       OleUninitialize();
       throw TFailedHR;
 }

 /* hr = pITypeInfo->GetFuncDesc(index,&pFuncDesc); */

 hr = pITypeInfo->GetFuncDesc(i,&pFuncDesc);

 if (FAILED(hr)) {
       ErrorMessage("Fallo en la llamada a ITypeInfo::GetFuncDesc", hr);
       OleUninitialize();
       throw TFailedHR;
 }

 while ( (pFuncDesc->memid != dispid) && (i<pTypeAttr->cFuncs) )
 {
       hr = pITypeInfo->GetFuncDesc(++i,&pFuncDesc);

       if (FAILED(hr)) {
             ErrorMessage("Fallo en la llamada a ITypeInfo::GetFuncDesc",
hr);
             OleUninitialize();
             throw TFailedHR;
       }
```

```
  }  // Del while

  while ( (pFuncDesc->memid == dispid) && (pFuncDesc->invkind !=
tipoInvocacion) )
  {
        hr = pITypeInfo->GetFuncDesc(++i,&pFuncDesc);

        if (FAILED(hr)) {
              ErrorMessage("Fallo en la llamada a ITypeInfo::GetFuncDesc",
hr);
              OleUninitialize();
              throw TFailedHR;
        }

  }  // Del while

  pITypeInfo->ReleaseTypeAttr(pTypeAttr);

  // En pFuncDesc tenemos un puntero a la FUNCDESC de la función a invocar

  // En pFuncDesc->cParams y pFuncDesc->cParamsOpt tenemos la cuenta del
  // número de parámetros y parámetros opcionales que recibe la función

  // Con pFuncDesc->lprgelemdescParam podemos conseguir un puntero a una
  // estructura ELEMDESC, que a su vez nos permite conseguir otra
estructura,
  // TYPEDESC, que define el tipo de un parámetro.
  // En el campo vt de esta última estructura tenemos el tipo del parámetro

  // Preparamos los argumentos para la llamada a la función

  // Estructura DISPPARAMS para métodos sin parámetros
  DISPPARAMS param = {NULL,NULL,0,0};

  // Número de parámetros obligatorios
  int numParOblig;

  switch (pFuncDesc->cParamsOpt) {

  case -1: {
  // A value of -1 specifies that the method's last parameter is a pointer
  // to a safe array of variants. Any number of variant arguments greater
  // than cParams -1 must be packaged by the caller into a safe array and
  // passed as the final parameter. This array of optional parameters must
  // be freed by the caller after control is returned from the call
        break;
              }

  case 0: {
        // No hay parámetros opcionales soportados
        numParOblig = pFuncDesc->cParams ;
        break;
              }

  default: {
        // Entonces cParamsOpt indica el número de parámetros opcionales
        numParOblig = (pFuncDesc->cParams) - (pFuncDesc->cParamsOpt) ;
              }

  }  // Del switch
```

```
 // Número de parámetros introducidos por el usuario
 int numParUser = params.NumItems();

 if (numParOblig>0) {  // La función tiene al menos un parámetro
obligatorio

       // Comprobamos si el número de parámetros es el correcto
       if (numParUser < numParOblig) {
             OleUninitialize();
             throw TNumParamError;
       }

       // El vector argTypes contendrá información sobre los tipos de
       // los parámetros mediante pares de la forma {"INTEGER", VT_I4}
       CVector<PARES> argTypes(numParUser);

       // Rellenamos el vector argTypes con información sobre los tipos

       for (int j=0; j<numParUser; j++) {

             PARES aux;
             TRef &p = th.getRef(*params[j]);

             aux.tipoEnCarbayonia = p.getType();
             aux.tipoEnAutomation = pFuncDesc->lprgelemdescParam-
>tdesc.vt;

             argTypes[j]= aux;

             pFuncDesc->lprgelemdescParam++;
       }

       // En el CVector argTypes tenemos toda la información sobre
       // el tipo y número de los parámetros de la función a invocar

       // Ahora tenemos que convertir los parámetros almacenados en el
       // CDSet<string *> params a los tipos requeridos por la función a
       // invocar a través del IDispatch::Invoke

       transferParams2VariantArgs(th,params,&argTypes,param,mydispid);

 }  // La función no tiene parámetros obligatorios

 // En param tenemos la estructura DISPPARAMS preparada
 // para la llamada a IDispatch::Invoke

 // Creamos un VARIANT para el valor retornado por la función

 VARIANT varResult ;
 ::VariantInit(&varResult) ;

 EXCEPINFO excepinfo ;

 // Hacemos la llamada a la función a través de IDispatch::Invoke
 hr = (self->getValue())->Invoke(dispid,
                          IID_NULL,
                                    GetUserDefaultLCID(),
                              pFuncDesc->invkind,
                                    &param,
                          &varResult,
                                    &excepinfo,
```

```
                          NULL) ;

 // Tratamiento del error

 if (FAILED(hr)) {

       ErrorMessage("Fallo en la llamada a IDispatch::Invoke", hr) ;
       switch (hr) {

       case DISP_E_EXCEPTION:
       {
             trace("Tenemos informacion del error desde el componente") ;

             if (excepinfo.pfnDeferredFillIn != NULL)
             {
                   (*(excepinfo.pfnDeferredFillIn))(&excepinfo) ;
             }

             strstream sout ;
             sout << "Fuente de la excepcion:  "
                  << excepinfo.bstrSource       << endl
                    << "Descripcion del error: "
                    << excepinfo.bstrDescription << endl
                    << ends ;
             trace(sout.str()) ;
             break;
       }

       case S_FALSE:
       {
             trace("La llamada a IDispatch::Invoke retorno un valor
booleano falso");
             break;
       }

       case E_UNEXPECTED:
       {
             trace("Error inesperado") ;
             break;
       }

       case E_NOTIMPL:
       {
             trace("Funcion miembro no implementada") ;
             break;
       }

       case E_FAIL:
       {
             trace("Error sin especificar") ;
             break;
       }

       }  // Del switch

       OleUninitialize();
       throw TFailedHR;
 }

 // Tratamiento del resultado
```

```
 // Comprobamos que el tipo retornado por la función es el esperado

/*      if (varResult.vt != pFuncDesc->elemdescFunc.tdesc.vt) {
        // Tipo retornado inesperado
        OleUninitialize();
        throw TReturnTypeError;
 } */

   switch (varResult.vt) {

 case VT_EMPTY:
        {
              checkReturnValue(th, "VOID", lvalue);
              break;
        }

 case VT_I4: case VT_UI1: case VT_I2:
        {
              checkReturnValue(th, "INTEGER", lvalue);
              instanceID integerID = areaInstancias.newInstance("INTEGER");
              TIInteger *pInteger = dynamic_cast<TIInteger
*>(areaInstancias.getInstance(integerID));
              VariantToInteger(&varResult,pInteger);
              th.getRef(lvalue).setInstance(integerID);
        }
        break;

 case VT_R4: case VT_R8:
        {
              checkReturnValue(th, "FLOAT", lvalue);
              instanceID floatID = areaInstancias.newInstance("FLOAT");
              TIFloat *pFloat = dynamic_cast<TIFloat
*>(areaInstancias.getInstance(floatID));
              VariantToFloat(&varResult,pFloat);
              th.getRef(lvalue).setInstance(floatID);
        }
        break;

 case VT_BOOL:
        {
              checkReturnValue(th, "BOOL", lvalue);
              instanceID boolID = areaInstancias.newInstance("BOOL");
              TIBool *pBool = dynamic_cast<TIBool
*>(areaInstancias.getInstance(boolID));
              VariantToBool(&varResult,pBool);
              th.getRef(lvalue).setInstance(boolID);
        }
        break;

 case VT_BSTR:
        {
              checkReturnValue(th, "STRING", lvalue);
              instanceID stringID = areaInstancias.newInstance("STRING");
              TIString *pString = dynamic_cast<TIString
*>(areaInstancias.getInstance(stringID));
              VariantToString(&varResult,pString);
              th.getRef(lvalue).setInstance(stringID);
        }
        break;

 default:
```

```
        {
              OleUninitialize();
              throw TReturnTypeError;
        }

  }  // Del switch

  // Liberamos recursos

  //pITypeInfo->ReleaseFuncDesc(pFuncDesc);
  pITypeInfo->Release();
}
```

Apéndice C:
Referencias

[Abowd00] Abowd, G.D. & Mynatt, E.D.. "Charting Past, Present, and Future Research in Ubiquitous Computing". ACM Transactions on Computer-Human Interaction, vol. 7, nº 1, Mar 2000, pp. 29-58. Julio de 2000.

[Abowd94] Gregory Abowd, Robert Allen, David Garlan. "Formalizing Style of understand descriptions of software architecture". ACM Transactions on Software Engineering and Methodology. Julio de 1995.

[Acebal02] César F. Acebal, et al.. "Good Design Principles in a Compiler University Course". ACM SIGPLAN Notices. Abril de 2002.

[Actional04] Actional. "Actional SOAPSwitch". http://www.actional.com/products/other_products/soapswitch/index.asp. Enero de 2004.

[Agarwal03] M. Agarwal. et al.. "AutoMate: Enabling Autonomic Applications on the Grid". AMS2003. Enero de 2003.

[Aguilar04] Alejandro Aguilar, et al.. "Modelos Computacionales". http://juanfc.lcc.uma.es/EDU/EP/trabajos/T102-ModelosComputacionales.pdf. Septiembre de 2004.

[Aho90] Alfred V. Aho, et al.. "Compiladores: Principios, técnicas y Herramientas". Addison Wesley Longman. ISBN: 968-444-333-1. Mayo de 1990.

[Aiten03] Erwin Aitenbichler, Jussi Kangasharju. "Communication Abstraction in MudoCore". CADS-ECCOP2003. Enero de 2003.

[Alvarez00] Fernando Álvarez García. "AGRA: Sistema de Distribución de Objetos para un Sistema Distribuido Orientado a Objetos soportado por una Máquina Abstracta". Universidad de Oviedo. Septiembre de 2000.

[Alvarez97] Álvarez D., Tajes L. et. al. "An Object-Oriented Abstract Machine as the Substrate for a Object-Oriented Operating System". 11th European Conference on Object Oriented Programming (Workshop). Julio de 1997.

[Alvarez98] Fernandez Alvarez Garcia, Lourdes Tajes Martinez. "AGRA: the Object Distribution Subsystem of the SO4 Object-oriented Operating System". Proceedings of the PDPTA98. International Conference on Parallel and Distributed Processing Techniques and Applications. Enero de 3000.

[Alvarez98a] Darío Álvarez Gutiérrez. "Persistencia Completa para un Sistema Operativo Orientado a Objetos usando una Máquina Abstracta con Arquitectura Reflectiva". Tésis Doctoral. Universidad de Oviedo. Marzo de 1998.

[ANS89] Architecture Project Management. "ANSA Reference Manual". Cambridge England. Enero de 1989.

[Arbad94] George A. Papadopoulos, Farhad Arbad. "Configuration and Dynamic Reconfiguration of Components Using Coordination Paradigm". CiteSeer. Enero de 1994.

[Attunityy05] Attunity Software Inc.. "Attunity Connect". http://www.attunity.com/Products/AttunityConnect.Asp. Abril de 2005.

[Avak05] Avaki Corporation. "Web services grid protocol". http://www.avaki.com/products/k2.html. Enero de 2005.

[Avaki05] Avaki Corporation. "Avaki 2.0 concepts and architecture". http://www.avaki.com/papers/AVAKI_cpncepts_architecture.pdf. Enero de 2005.

[Bailin97] Sidney C. Bailin. "Applying Multi-Media to the Reuse of Design Knowledge". Eighth Annual Workshop on Institutionalizing Software Reuse (WISR). Marzo de 1997.

[Bastide00a] Remi Bastide, Ousmane Sy, David Navarre, Philippe Palanque. "A Formal Specification of the Corba Event Service". 4th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000). Septiembre de 2000.

[Bastide00b] R. Bastide and Ousmane Sy. "Towards Componets that Plug and Play". Universidad de Extremadura. ISBN: 84-699-2538-5. Julio de 2000.

[Bastide99] Remi Bastide, Ousmane Sy, Philippe Palanque. "Formal Specification and Prototyping of CORBA Systems". 13 ECOOP 1999. Junio de 1999.

[Bernstein93] Philip A. Bernstein. "Middleware an Architecture for Distributed System Services". Digital Equipment. Enero de 1993.

[Blac02] Andrew P. Blac. "Distributed Object - The Next Then Years". Fundations of Objects Oriented Languages 2002. Enero de 2002.

[Blair98] G.S. Blair, G. Coulson, P. Robin, M. Papathomas. "An Arquietcture for Next Generation Middlware". Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distribuited Processing. Enero de 1998.

[Bloch01] Wolfgang Blochinger. "Distributed High Performance Computing in Heterogeneous Enviroments with DOTS". HCE. Abril de 2001.

[Bobrow91] G. Bobrow, et al.. "The Art of the Metaobject Protocol". MIT Press, Cambridge, Massachusetts.. Junio de 1991.

[Bobrow93] D.G.Bobrow, R.G.Gabriel y J.L.White. "CLOS in Context-The Shape of the Design Space. En Object Oriented Programming – The CLOS Perspective". MIT Press. Septiembre de 1993.

[Boll05] Norbert Bollow. "DotGNU Project ". http://www.gnu.org/projects/dotgnu/. Febrero de 2005.

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

[Booth04] David Booth. "Web Services Architecture". http://www.w3.org/TR/ws-arch/. Febrero de 2004.

[Borland91] Borland Corporation. "Back end Borland languages Specification". Borland Corporation. Junio de 1991.

[Bosch00] Jan Bosch. "Design & Use of Software Architectures". Pearson Education. ISBN: 0-201-67494-7. Enero de 2000.

[Botzer02] David Botzer . "Amit - Active Middleware Technology". IBM Research Lab in Haifa. Octubre de 2002.

[Box98] Don Box. "Q&A ActiveX/COM". Microsoft Systems Journals.

Marzo de 1998.

[Brant03] John Brant and Don Roberts. "SmaCC". http://www.refactory.com/Software/SmaCC/. Febrero de 2003.

[Brian82] Technical Report 272, MIT Laboratory of Computer Science. "Reflection and Semantics in a Procedural Language". Massachusetts Institute Technological. Junio de 1982.

[Briot89] Jean-Pierre Briot and Pierre Cointe. "Programming with Explicit Metaclasses in SmallTalk-80". In proceedings of OOPSLA89, volume 24 of Sigplan Notices, pages 419-43. ACM1. Octubre de 1989.

[Brown99] Alan W. Brown, Balbir Barn. "Enterprise Scale CBD: Building Complex Computer Systems From Components". 9th International Workshop on Software Technology and Engineering Practice. Septiembre de 1999.

[Budgen99] David Budgen, Amnart Pohthong. "Component Reuse in Software Design: An Observational Study". Software Technology & Engineering Practice (STEP 99). Agosto de 1999.

[Busch96] Frank Buschmann. "Pattern-Oriented Software Architecture". John Wiley & Sons. ISBN: 0471958697. Enero de 1996.

[Buyya01] Rajkumar Buyya, et al.. "An Economy Grid Architecture for Service-Oriented Grid Computing". 10th IEEE International Heterogeneous Computing Workshop. Septiembre de 2001.

[Campione99] Mary Campione, Kathy Walrath. "The Java Tutorial. Second edition. Object ORiented Programming for Internet". The Java Series. Sun Microsystems. Septiembre de 1999.

[Canal00] C. Canal, L.Fuentes, J.M. Troya and A. Vallecillo. "Adding Protocol Information to CORBA IDLs". Universidad de Extremadura. ISBN: 84-699-2538-5. Julio de 2000.

[Caron00] Rob Caron. "Develop a Web Service: Up and Running with the SOAP Toolkit for Visual Studio". MSDN Magazine. Septiembre de 2000.

[Cazzola98] Walter Cazzola. "Evaluation of Object-Oriented Reflective Models". ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS98). Junio de 1998.

[Chappel96] David Chappell. "Undestanding ActiveX and OLE". Microsoft Press. ISBN: ¿No tiene?. Enero de 1996.

[Chappell02] Dave Chappell, Tyler Jewell. "Java Web Services". OReilly. ISBN: 0-596-00269-6. Marzo de 2002.

[Chiba98] Shigeru Chiba, Michiaki Tatsubori. "Yet Another java.lang.Class". ECOOP98 Workshop on Reflective Object-Oriented Programming Systems. Julio de 1998.

[Cho00] I. Cho. "A framework for the Specification and Testing the Interopration aspect of Components". Universidad de Extremadura. ISBN: 84-699-2538-5. Julio de 2000.

[Cho98] Il-Hyung Cho, Lee Krause. "Interoperable Software Modules". CiteSeer. Julio de 1998.

[Clarke00] Jerry a. Clarke, Raju R. Namburu. "A Distributed Computing Environment for Interdisciplinary Applications". CiteSeer. Diciembre de 2004.

[Cointe87] Pierre Cointe. "MetaClasses are first class objects: the ObjVLisp model". In proceedings of OOPSLA87, volume 22 of Sigplan Notices, Orlando, Florida. Octubre de 1987.

[Costa00] Joao Costa Seco, Luis Caires. "A Basic Model of Typed Components". ECOOP-2000. Marzo de 2000.

[Coulouris01] George Coulouris, Jean Dollimore, Tim Kindberg. "Sistemas Distribuidos Conceptos y Diseño". Addison Wesley. ISBN: 84-7829-049-4. Enero de 2001.

[Cueva91] Juan Manuel Cueva Lovelle. "Lenguajes, Gramáticas y Autómatas". Universidad de Oviedo. ISBN: 84-600-7871-X. Diciembre de 1991.

[Cueva94] J. M. Cueva Lovelle, et al.. "Introducción a la Programación Estructurada y Orientada a Objetos con Pascal". ISBN: 84-600-8646-1. Enero de 1994.

[Cueva96] Cueva Lovelle J.M. et. al. "El sistema Integral Orientado a Objetos: Oviedo3". II Jornadas sobre Tecnologías Orientadas a Objetos. Julio de 1996.

[Cueva98] Juan Manuel Cueva Lovelle. "Conceptos Básicos de Procesadores de Lenguaje". Editorial Servitec. Enero de 1998.

[Diaz00] María Ángeles Díaz Fondón. "Núcleo de Seguridad para un Sistema Operativo Orientado a Objetos Soportado por una Máquina Abstracta". Tesis Doctoral. Universidad de Oviedo. Enero de 2000.

[Diaz99] María Ángeles Díaz Fondón, et al.. "Integrating Capabilities into the

Object Model to Protect Distributed Object Systems". International Symposium on Distributed Objects and Applications (DOA99). Edimburgo. UK.. Enero de 1999.

[Dormi00] Sebastián Dormido, et al.. "Estructura y Tecnología de Computadores". Sanz y Torres. ISBN: 84-88667-53-1. Febrero de 2000.

[DSouza98] Desmon F. D`Souza, Alan Cameron Wills. "Objects, Components and Frameworks with UML. The Catalysis Approach". Addison Wesley. ISBN: 0-201-31012-0. Enero de 1998.

[Egon93] Egon Borger, et al.. "Formal Definition of an Abstract VHDL93 Simulator by EA-Machines". Universita di Pisa. Enero de 1993.

[ELCA04] ELCA Informatique SA. " IIOP.NET". http://iiop-net.sourceforge.net/. Mayo de 2004.

[Emmerich01] Wolfgang Emmerich. "Engineering Distributed Objects". John Wiley & Sons, Ltd. ISBN: 0-471-98657-7. Agosto de 2001.

[Expersoft05] Expersoft Corporation . "CORBAplus, ActiveX Bridge". http://www.corba.org/vendors/pages/expersoft2.html. Enero de 2005.

[Fer99] Laura Fernández Lozano. "Reflectividad Estructural para Carbayonia". Proyecto Fin de Carrera. Escuela Universitaria de Ingeniería Técnica en Informática de Gijón de la Universidad de Oviedo. Septiembre de 1999.

[Ferber89a] Jacques Ferber. "Computational Reflection in Class Based Object Oriented Languages". OOPSLA89, volume 24 of Sigplan Notices, pages 317-326. Octubre de 1989.

[Ferber98] Jacques Ferber, Oliver Gutknecht. "A meta-model for the analysis and design of organizations in multi-agent systems". Proceedings of the 3rd International Conference on Multi Agent Systems . ISBN: 0-8186-8500-X. Julio de 1998.

[Fipa03] Fipa.org. "The Foundation for Intelligent Physical Agent". http://www.fipa.org. Enero de 2003.

[Foote89] B.Foote y R.E.Johnson. "Reflective Facilities in Smalltalk-80". En Proceedings of the Object-Oriented Programming, Languages, Systems and Applications (OOPSLA'89). ACM. Octubre de 1989.

[Forst96] Jim Frost. "BSD Sockets: A Quick And Dirty Primer". http://world.std.com/~jimf/papers/sockets/sockets.html. Diciembre de 1996.

[Foster99] Ian Foster, Carl Kesselman. "The Grid: Blueprint for a New Computing Infraestructure". Morgan kaufmann Publisher. ISBN: 1558604758. Enero de 1999.

[Frei03] Andreas Frei, Andrei Popovici, Gustavo Alonso. "Event based system as adaptative middleware platforms". CADS-ECOOP2003. Enero de 2003.

[Freier96] Alan O. Freier, et al.. "The SSL Protocol, version 3.0". Trasport Layer Security Working Group. Julio de 1996.

[Gamma00] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. "Design Patterns". Pearson Education. ISBN: 0-201-63361-2. Noviembre de 2000.

[Ganek03] Alan Ganek. "Autonomic Computing: Implementing the Vision". AMS2003. Enero de 2003.

[Geist94] A. Geist, A. Geguelin, et al.. "PVM: Parallel Virtual Machine". MIT Press, Cambridge. Enero de 1994.

[Gelemter85] Gelernter, D.. "Generative communication in Linda". ACM Trans. Prog. Lang. syst. Vol 7 n1 pp 80-112. Enero de 1985.

[Gold83] Adele Goldberg and David Robson. "Smalltalk-80: The Language and Its Implementation". Addison-Wesley. ISBN: 0-201-11371-6. Enero de 1983.

[Goldfarb99] Charles F. Goldfarb, Paul Prescod. "Manual de XML". Prentice Hall. ISBN: 84-8322-105-5. Enero de 1999.

[Gosling96] James Gosling, Bil Joy and Guy Seele. "The Java language Specification". Addison-Wesley. Abril de 1996.

[Gotd04] gotdotnet.com. "Tutoriales del SDK de .Net Framework". http://es.gotdotnet.com/quickstart/default.aspx. Enero de 2004.

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

[Grace03] Paul Grace, Gordon Blair and Sam Samuel. "A Higher Level Abstraction for Mobile Computing Middleware". CADS- CAIS 2003. Enero de 2003.

[Graube89] OPSLA89, volume 24 of Sigpla Notices, pages 205-316. "Metaclasses Compatibiliry". ACM. Octubre de 1989.

[Haas05] Hugo Haas. "Web Services Activity Statement". http://www.w3.org/2002/ws/Activity. Enero de 2005.

[Han00] J. Han. "Temporal Logic Based Specifications of Component Interaction Protocols". Universidad de Extremadura. ISBN: 84-699-2538-5. Julio de 2000.

[Han99] Jun Han. "Semantic and Usage packaging for Software Components". Workshop on Object Interoperability (WOI99). Junio de 1999.

[He04] Hao He. "Web Services Architecture Usage Scenarios". http://www.w3.org/TR/ws-arch-scenarios/. Febrero de 2004.

[Henk94] Henk Barendregt, Eric Barendsen. "Introduction to Lambda Calculus". Aspen\aes Workshop on Implementation of Functional Languages. Octubre de 1994.

[Hey00] Dan Vander Hey. "One Customer, One View". Intelligent Enterprise. Marzo de 2000.

[Howe99] Denis Howe. "The Free Online Dictionary. ". www.foldoc.org. Marzo de 1999.

[Huggins02] James K. Huggins and Carles Wallace. "An Abstract State Machine Primer". www.cs.mtu.edu. Diciembre de 2002.

[Huggins99] Jim Huggins. "Abstract State Machines: Introduction". www.eecs.umich.edu/gasm/intro.html. Febrero de 1999.

[Idc01] IDC. "The Enterprise Application Integration Market Simmers with Robust Growth Expectations". IDC. Enero de 2001.

[ILCentral03] ILCentral. "An introduction to TCP/IP sockets and Winsock". https.//www.district86.k12.il.us/central/activities/computerclub/Tutorials/Winsock/Lesson1.htm. Octubre de 2003.

[Inmon01] William Inmon. "A Brief History of Integration". EAI Journal. Enero de 2001.

[Iona05] Iona Systems. "IONA Artix: ". http://www.iona.com/products/artix/welcome.htm. Abril de 2005.

[ISO92] ISO/IEC JTC1/SC212/WG17 CD 10746-1. "ISO RM-ODP Part1: Overview and guide use". International Standards organization. Enero de 1992.

[Ivar95] Ivar Jacobson. "Business process reengineering with object

technology". Addison-Wesley: ACM Press Books. ISBN: 0201422891. Julio de 1995.

[Ivknovic03] Igor Ivkovic, Michael Godfrey. "Enhacing Domain-Specific Software Architecture Recovery". 11th IEEE International Workshop on Program Comprehension (IWPC03). Marzo de 2003.

[Iway05] iWay Software. "IWay SOAPswitch". http://www.iwaysoftware.com/products/web_services.html. Marzo de 2005.

[Jacobs04] Ian Jacobs. "Architecture of the World Wide Web, Volume One". http://www.w3.org/TR/webarch/. Diciembre de 2004.

[Jnbrid04] JNBridge LLC. "JNBridgePro". http://www.jnbridge.com/index1.htm. Enero de 2004.

[Jones99] Paul Jones. "VMware Virtual Platform for Linux: Beyond Dual-Booting". Internet.com. Mayo de 1999.

[Kaplan00] A. Kaplan and J. C. Wileden. "A Multi-Level Approach to Verifiable Correct Design and Construction of Multi-Language Software". Universidad de Extremadura. ISBN: 84-699-2538-5. Julio de 2000.

[Katiyar00] Dinesh Katiyar. "Notes on OLE and CORBA". ftp://ftp.theory.standord.edu/pub/katiyar/misc/ole_vs_corba.html. Marzo de 2000.

[Keller98] Rudolf K. Keller, Reinhard Schauer. "Design Components: Towards Software Composition at the Design Level". International Conference on Software Engineering (ICSE98). Abril de 1998.

[Kep03] Chamath Keppitiyagama, Norman C. Hutchinson. "Multiparty Communication Types for Distributed Applications". CADS-DAIS 2003. Enero de 2003.

[Keshav98] R. Keshav and R. Gamble. "Toward a Taxonomy of Architecture Integration Strategies". 3rd International Software Architecture Workshop. Septiembre de 1998.

[Keshav99] Reshma Madhusuda Keshav. "Architecture Integration Elements: Connector Models that form Middleware". University of Tulsa. Enero de 1999.

[Keuffel00] Warren Keuffel. "Best Practice Actually applied". Software Development. Julio de 2000.

[Kiczales91] G.Kiczales, J. desRivières y D.G.Bobrow. "The art of MetaObject Protocol". MIT Press, Cambridge, Massachusetts. Octubre de 1991.

[Kiczales92] G.Kiczales. "Towards a New Model of Abstraction in Software Engineering". Proceedings of the International Workshop on New Models for Software Architecture 92. Marzo de 1992.

[Kirtland00] Mary Kirtland. "The Programmable Web: Web Services Provides Building Blocks for the Microsoft .NET Framework". msdn. Septiembre de 2000.

[Kobryn00] Cris Kobryn. "UML Meets EJB and COM+". Software Development. Diciembre de 2000.

[Kon00] Fabio Kon, Roy H. Campbell. "Dependence Management in Component-Based Distributed System". IEEE Concurrency Vol 8 nº1. Enero de 2000.

[Kon98] Fabio Kon, Ashish Singhai, Roy H. Campbell, Dulcineia Carvalho. "2K: A Reflective, Component-Based Operating System for Rapidly Changin Environments". ECOOP98 Workshop On Reflective Object-Oriented Programming . Julio de 1998.

[Kon99] Fabio Kon, Roy H Campbell. "Supporting Automatic Configuration of Component-Based Distributed System". Usenix Conference on Object Oriented Technologies and System (COOTS 99). Mayo de 1999.

[Kons03] Alexander V. Konstantinou, Yechiam Yemini. "Programming Systems for Autonomy". AMS2003. Enero de 2003.

[Koster01] Rainer Koster, et al. "Infopipes for Composing Distributed Information Flows". Rainer Koster University of Kaiserslautern. Enero de 2001.

[Kozak00] Robert Kozak. "The Dish on Kylix. Cross-Platform Control. From Windows to Linux, and Back". DelphiZone.com. Mayo de 2000.

[Kramer96] Douglas Kramer. "The Java Platform. A White Paper". Sun Microsystems Java Soft. Mayo de 1996.

[Kursz03] Dawid Kurszyniec, Tomasz Wrzosek, VAidy Sunderam. "Heterogeneous Access to Service-based Distributed Computing: the RMIX Approach". Heterogeneous Computing Workshop 2003. Abril de 2003.

[Labra04] José Emilio Labra, et al.. "Intérpretes y Diseño de Lenguajes de Programación". Universidad de Oviedo. Abril de 2004.

[Lanc01] Lancaster University. "OpenCOM". http://www.comp.lancs.ac.uk/computing/research/mpg/reflection/opencom.php. Octubre de 2001.

[Larson96] Jim Larson . "An Introduction to Lambda Calculus and Scheme". http://www.jetcafe.org/~jim/lambda.html. Julio de 1996.

[Ledoux99] T.Ledoux. "OpenCorba: a Reflective Open Broker". En Proceedings of Reflection 99. LNCS, Springer-Verlag. Julio de 1999.

[Lee00] Craig Lee and James Stepanek. "On Future Global Grid Communication Performance". HCE. Junio de 2001.

[Lehrm03] Ole Lehrm Madsen. "Object-Oriented Language Interoperability". http://www.pervasive.dk/projects/langInter/langInter_summary.htm. Febrero de 2003.

[Lindh99] Tim Lindholm y Frank Yellin. "The JavaTM Virtual Machine Specification, Second Edition". Sun Microsystems. Enero de 1999.

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

[Maes87] Pattie Maes. "Issues In Computational Reflection". North-Holland. Bruselas, Bélgica. Agosto de 1987.

[Maloney00] Joh Maloney. "Morphic: The Self User Interface Framework". Sun Microsystems. Julio de 2000.

[Marshall03] J. Marshall. "HTTP Made Really Easy". http://www.jmarshall.com/easy/http. Octubre de 2003.

[Mateos00a] Francisco Domínguez Mateos, Merabti Madjid. "Scalable Heterogeneous Bridge for Networked Appliances". 2nd International Workshop on Networked Appliances (IWNA2000).Sponsored by IEEE . Diciembre de 2000.

[Mateos00b] Francisco Domínguez Mateos, Merabti Madjid. "Component Support Services for Heterogeneous Networked Appliances". 2nd International Workshop on Networked Appliances (IWNA2000).Sponsored by IEEE . Diciembre de 2000.

[Mateos00c] F. Dominguez, J. M. Cueva. "Interoperability Oviedo3/COM Objects through Automation". Universidad de Extremadura. ISBN: 84-699-2538-5. Julio de 2000.

[Mateos02] Francisco Domínguez Mateos, Rafael Capilla. "OVIBUS: A Scalable Platform for Combining Heterogeneus Components".

http://www.idt.mdh.se/CBprocesses. Octubre de 2002.

[Mateos98] Francisco Dominguez Mateos. "Justificación de la Creación de un Sistema de Componentes: FabesO3, basado en el Sistema Integral Orientado a Objetos: Oviedo3". Universidad de Oviedo. Septiembre de 1998.

[Matougui03] Selma Matougui, Antoine Beugnard. "Communication Abstractions: Components and Connectors". CADS-DAIS 2003. Enero de 2003.

[Maurer00] Peter M. Maurer. "What If they Gave a Revolution and Nobody Came?". http://www.csee.usf.edu/~maurer/whatif/whatif.htm. Marzo de 2000.

[Meyer00a] Bertrand Meyer. "Contracts for Components". Software Development. Julio de 2000.

[Meyer00b] Bertrand Meyer. "The Significance of doc-NET". Software Development. Noviembre de 2000.

[Meyer00c] Bertrand Meyer. "What to Compose". Software Development. Marzo de 2000.

[Meyer01] Bertrand Meyer. "The Power of a Unifying View". Software Development. Junio de 2001.

[Meyer02a] Bertrand Meyer. "Polyglot Programming". Software Development. Mayo de 2002.

[Meyer02b] Bertrand Meyer. "Achieving Interoperability". Software Development. Junio de 2002.

[Meyer02c] Bertrand Meyer. "The .NET Cost: Who Pays?". Software Development. Julio de 2002.

[Meyer99] Bertrand Meyer. "The Significance of Components". Sfotware Development. Noviembre de 1999.

[Midliardi01] Mauro Migliardi, Dawid Kurzyniec, Vaidy Sunderam. "Standards Based heterogeneous Metacomputing: The Desigh of HARNESS II". HCE. Junio de 2001.

[Mora02] Carlos de Mora Buendía, et al.. "Estructura y Tecnología de Computadores I". UNED. ISBN: 84-362-4642-X. Agosto de 2002.

[Morisawa02] Yoshitomi Morisawa, Katsoru Inoue, Koji Torri. "Architectural Style for Distributed Processing Systems and Practical Selection Method". Information & Software Technology . Enero de 2002.

[Mpi05] MPI Forum. "Message Passing Interface". http://www.mpi-forum.org. Enero de 2005.

[MSDN01] MSDN. "Overview of the .NET Framework". Microsoft corporation. Julio de 2001.

[Munindar98] Munindar P. Singh. "Agent Communication Languages: Rethinking the Principles". IEEE Computer. Enero de 1998.

[Need93] R. M. Needham. "Names in Distributed Systems, an Advanced Course". ACM Press/Addson-Wesley. pp315-26. Enero de 1993.

[Nenad00] Nenad Medvidovic, David S. Rosenblum, Rose F. Gamble. "Bridging Heterogeneous Software Interoperability Platforms". University of Tulsa. Enero de 2000.

[Nierstrasz90] Franz Achermann, Oscar Nierstrasz. "A tour of Piccola". Software Composition Group. Enero de 1990.

[Nierstrasz91] Oscar Nierstrasz, Dennis Tsichritzis, Vicki de Mey, Marc Stadelmann. "Objects + Scripts = Applications". Espirit 1991. Enero de 1991.

[Nordl01] Johan Nordlander. "O-Haskell". http://www.cs.chalmers.se/~nordland/ohaskell/. Enero de 2001.

[Nutt03] Gary Nutt. "Distributed Virtual Machines". To be published. Enero de 2003.

[OMG04] OMG. "CORBA™/IIOP™ Specification". http://www.omg.org/technology/documents/formal/corba_iiop.htm. Marzo de 2004.

[OMG95] OMG. "Common Object Request Broker Architecture (CORBA) v2.4.2". OMG. Julio de 1995.

[Omni03] http://omniorb.sourceforge.net/. "omniORB2". http://omniorb.sourceforge.net/. Diciembre de 2003.

[OOO00] Objects Oriented Concepts, Inc. "ORBacus For C++ and Java". Objects Oriented Concepts, Inc. Enero de 2000.

[Ortin01] Francisco Ortín Soler. "Sistema Computacional de Programación Flexible Diseñado sobre una Máquina Abstracta Reflectiva no Restrictiva". Tesis Doctoral. Universidad de Oviedo. Diciembre de 2001.

[Ortin05] Francisco Ortín, José Redondo, Luis Vinuesa y Juan Manuel Cueva.

"Adding Structural Reflection to the SSCLI". 3rd International Conference on .Net Technologies.. Junio de 2005.

[Payton00] J. Payton, R.Gamble, et al. "The Opportunity for Formal Models of Integration". 2nd Int Conference on Information Reuse and Integration. Julio de 2000.

[Perez00] SAHARA: Arquitectura de Seguridad Integral para Sistemas de Agentes Móviles. "Jesús Arturo Pérez Díaz". Universidad de Oviedo. Enero de 2000.

[Pressman98] Roger S. Pressman. "Ingeniería del Software". McGraw Hill. ISBN: 84-481-1186-9. Enero de 1998.

[Prowel00] Bruce Prowel Douglass. "Components, States and Interfaces, Oh My!". Software Development. Abril de 2000.

[PUD00] Ivar Jacobson, Grady Booch, James Rumbaugh. "El Proceso Unificado de Desarrollo de Software". Addison Wesley. ISBN: 84-7829-036-2. Enero de 2000.

[Purtillo94] J. Purtillo. "The Polylith Software Bus". ACM Transaction Programming Languagues and Systems. Enero de 1994.

[Putman00] J. Putman and D. Hybertson. "Interaction Framework for Interoperability and Behavioral Analyses". Universidad de Extremadura. ISBN: 84-699-2538-5. Julio de 2000.

[Raman98] L. G. Raman. "OSI Systems and Network Management". Communications Magazine. Marzo de 1998.

[Rein00] R. van Rein. "On Practical Verification of Processes". Universidad de Extremadura. ISBN: 84-699-2538-5. Julio de 2000.

[Ren01] Frances Ren. "The Marketplace of Enterprise Application Integration (EAI)". http://www.public.asu.edu/~mbfr2047/eai.html . Enero de 2001.

[Reussner00] R. H. Reussner. "An Enhaced Model for Component Interfaces to Support Automatic and Dynamic Adaptation". Universidad de Extremadura. ISBN: 84-699-2538-5. Julio de 2000.

[Rinat01] Ran Rinat, Scott F. Smith. "Cells: Dynamic Trusted Components". The Johns Hopkins University. Marzo de 2001.

[Ruiz00] A. Ruiz, R. Corchuelo, O. Martin, A. Durñan and M. Toro. "Addressing Interoperability in Multi-Organizational Web-Based Systems". Universidad de Extremadura. ISBN: 84-699-2538-5.

Julio de 2000.

[Schawan04] Karsten Schwan, Greg Eisenhauer, Mustaq Ahamad, et. al.. "IQ-Echo: Interoperability and Quality of Service Across Heterogeneous Hardware/Software Platforms". Enero de 2004.

[Schon99] Bastiaan Schonhage, Anton Eliens. "Four ways to Architect your Distributed Objects". NOSA 99, Proceedings of the Second Nordic Workshop on Software Architecture, Edited by Jan Bosch. Marzo de 1999.

[Serrano05] Alejandro Serrano. "Cw, la evolución de C#". Todo Programación nº9. Studio Press. Enero de 2005.

[Siebel05] Siebel Systems, Inc. "Universal Application Network". http://www.siebel.com/universal-application-network/solutions-description.shtm. Abril de 2005.

[Singhai99] Ashish Singhai, Aamod Sane, Roy H. Campbell. "Quarterware for Middleware". International Conference on Distributed Computer Systems. Julio de 1999.

[Skon02] Aaron Skonnard. "Publishing and Discovering Web Services with DISCO and UDDI". MSDN Magazine. Febrero de 2002.

[Smith00] Roger Smith. "Less Glue Code, Better Interfaces". Software Development. Noviembre de 2000.

[Smith82] B.C.Smith. "Reflection and Semantics in a procedural language. PhD Thesis". Technical Report 272 Massachusetts Institute of Technology. Laboratory for Computer Science. Cambridge, Massachusetts. Octubre de 1982.

[Smith95] Randall B. Smith and David Ungar. "Programming as an Experience: The Inspiration for Self". Springer-Verlag. Lecture Notes In Computer Science; Vol. 952. Proceedings of the 9th European Conference on Object-Oriented Programming. ISBN: 3-540-60160-0. Julio de 1995.

[Snell00] James Snell. "Exposing Application Services With SOAP". http://www.xml.com/pub/2000/7/12/soap/mssoaptutorial.html. Julio de 2000.

[Sonic05] Sonic Software. "Sonic ESB". http://www.sonicsoftware.com/index.ssp. Abril de 2005.

[Sour98] J.L.Contreras y J.-L.Sourrouille. "Adaptive Active Object". En Object-Oriented Technology. ECOOP 98 Workshop Reader. S.Demeyer y J.Bosch, eds. Pág 369-371. Lecture Notes in

Computer Science 1543. Springer-Verlag. Julio de 1998.

[Squeak05] Alan Kay, et al.. "Squeak". www.squeaklan.org. Enero de 2005.

[STEE61] Steel T.B.. " A first version of UNCOL". Western Joint Computer Conference pp 371-378. Enero de 1961.

[Steel60] T. B. Steel Jr. "UNCOL: Universal Computer Oriented Language". Datamation. Febrero de 1960.

[Strob01] Walter Stroebel. "Soap2Corba and Corba2Soap". http://soap2corba.sourceforge.net. Mayo de 2001.

[Sun05] Sun Microsystems. "The Sun Grid Engine". http://www.sun.com/software/gridware. Enero de 2005.

[Sun95] Sun Microsystems. "The Java Virtual Machine Specification. Release 1.0 Beta Draft". Sun Microsystems Computer Corporation. Agosto de 1995.

[Sun97] Javasoft. "Java Remote Method Invocation Specification (RMI)". Sun Microsystems. Diciembre de 1997.

[Sun98] Sun Microsystems. "InfoBus 1.2 Specification". http://java.sun.com. Enero de 1998.

[Szyperski00a] Clemens Szyperski. "Components and Architecture". Software Development. Octubre de 2000.

[Szyperski00b] Clemens Szyperski. "Point, Counterpoint". Software Development. Febrero de 2000.

[Szyperski00c] Clemens Szyperski. "Components and Contracts". Software Development. Mayo de 2000.

[Szyperski01a] Clemens Szyperski. "Components and Web Services". Software Developer Magazine. Agosto de 2001.

[Szyperski01b] Clemens Szyperski. "Components and Continuity". Software Development. Abril de 2001.

[Szyperski02a] Clemens Szyperski. "Universe of Composition". Software Development. Agosto de 2002.

[Szyperski02b] Clemens Szyperski. "Services Rendered". Software Development. Enero de 2002.

[Szyperski02c] Clemens Szyperski. "Back to the Universe". Software Development.

Septiembre de 2002.

[Szyperski98] Clemens Szyperski. "Component Software. Beyond Object-Oriented Programming". Addison Wesley. ISBN: 0-201-17888-5. Enero de 1998.

[Szyperski99a] Clemens Szyperski. "Components and Objects Together". Software Development. Mayo de 1999.

[Szyperski99b] Clemens Szyperski. "Greetings from DLL Held". Software Development. Octubre de 1999.

[Tane02] Tanenbaum Andrew, Van Steen Maarten. "Distributed Systems Principles and Paradigms". Prentice Hall. ISBN: 0-13-088893-1. Enero de 2002.

[Trados96] Alain Trados y Eric Uber. "Using Borland Delphi and C++ Together". A tecnical Paper for Developer. Borland Online. Marzo de 1996.

[Troelsen01] Andrew Troelsen. "C# and the .NET Platform". Apress. ISBN: 1-893115-59-3. Enero de 2001.

[Turing36] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". Proceedings of The London Mathematical Society, Series 2,42. Enero de 1936.

[UBMC] UMBC AgentWeb. Software agents mailing list. "Mobile Agent Definitions". http://www.cs.umbc.edu/agentslist/archive/current. Abril de 2000.

[UC01] Instituto de Física de Cantabria. "GRID Recursos distribuidos para el procesado de grandes volúmenes de información". CSIC-UC. Abril de 2001.

[UML99] Grady Booch, James Rumbaugh, Ivar Jacobson. "El Lenguaje Unificado de Modelado". Addison Wesley. ISBN: 84-7829-028-1. Enero de 1999.

[Ungar00] David Ungar. "How to Program in Self 4.1". Sun Microsystems. Septiembre de 2000.

[Ungar91] David Ungar and Randall B. Smith. "Self: The Power of Simplicity". LISP and Symbolic Computacion: And International Journal. Marzo de 1991.

[Valle99] Antonio Vallecillo Moreno. "Un Modelo de Componentes para el Desarrollo de Aplicaciones Distribuidas". Tesis Doctoral.

Universidad de Málaga. Marzo de 1999.

[Ven98] Bill Venners. "Inside the Java Virtual Machine". McGraw-Hill. ISBN: 0-07-135093-4. Enero de 1998.

[W3C00a] John J. Barton, Gaurav Misra. "SOUP: Simple Object Update Protocol". W3C.org. Agosto de 2000.

[W3C00b] W3C. "Simple Object Access Protocol (SOAP) 1.1". W3C.org. Mayo de 2000.

[W3C01a] w3c. "XML Schema Part 0: Primer". http://www.w3.org/TR/xmlschema-0. Mayo de 2001.

[W3C01b] W3C. "Web Services Description Language (WSDL) 1.1". http://www.w3.org/TR/wsdl. Marzo de 2001.

[W3C05] W3C. "Naming and Addressing: URIs, URLs, ...". http://www.w3.org/Addressing/. Febrero de 2005.

[W3C99] W3C. "Namespaces in XML". http://www.w3.org/TR/REC-xml-names/. Noviembre de 1999.

[Wang00] Nanbo Wang, Doublas C. Schmidt, and Carlos ORayan. "Overview of the CORBA Component Model". Overview of the CORBA Component Model. Enero de 2000.

[Wiki04] wikipedia en español. "Computación distribuida". http://es.wiipedia.org/wiki/Computación_distribuida. Junio de 2004.

[Will90] Tony Williams. "On Inheritance: What It Means and How To Use it". Microsoft Corporation. Febrero de 1990.

[Williams88] Tony Williams. "Dealing with the Unkown". Microsoft Corporation. Diciembre de 1988.

[Winer01] Dave Winer, Jake Savin, UserLand Software. "A Busy Developer Guide to SOAP 1.1". http://www.soapware.org/bdg. Febrero de 2001.

[Wirth84] Niklaus Wirth. "Algoritmos + Estructuras de Datos = Programas". Ediciones del Castillo. Madrid.. ISBN: 84-219-0172-9. Enero de 1984.

[Wolczko02] Mario Wolczko. "Classes and Metaclasses in Smalltalk". University of Manchester. Enero de 2002.

[Wsif03] The Apache Software Foundation. "WSIF Apache Web Services

Invocation Framework". http://ws.apache.org/wsif. Enero de 2003.

[Ximi05] Ximian. "Mono Project". http://www.mono-project.com/about/index.html. Febrero de 2005.

[Yee01] Andre Yee. "Demystifying Business Process Integration". EaiQ. Enero de 2001.

[Yuri91] Yuri Gurevich. "Evolving Algebras: An Attempt to Discover Semantics". The University of MIchigan. Febrero de 1991.

[Zeiger01] BErnad P. Zeigler, et al.. "DEVS Modeling and Simulation: A new layer of middleware". AMS2001. Enero de 2001.

[Zhou03] Jiehan Zhou, Eila Niemela. "Middard, Middleware, Middardware and Middleware technology". CADS-DAIS 2003. Enero de 2003.