# Un Modelo Orientado a Objetos para el Desarrollo de Sistemas con Arquitecturas de Tres Capas

Trabajo de grado de la Licenciatura en Informática de

Pablo Hernán Murias

Director: Lic. Máximo Prieto.

Co-Director: Lic. Silvia Gordillo.

Facultad de Informática.

Universidad Nacional de La Plata

**Agradecimientos:**

A mis padres. Gracias por haberme apoyado y alentado siempre. Y gracias por hacer todo lo posible para que no me falte nada a lo largo de toda mi carrera.

A Laura, por estar al lado mío.

A mis directores de tesis, por haberme aconsejado y orientado durante el desarrollo de este trabajo.

A los profesores que me enseñaron cosas durante todo este camino.

A los amigos que me dio la facultad, con quienes compartí muy buenos momentos.

# Índice

# Capítulo 1

# Introducción

En este capítulo se describen la motivación y los objetivos del trabajo de grado, se enumeran brevemente los resultados obtenidos y, por último, se detalla la estructura del informe para ayudar al lector a tener una visión global del mismo.

## Motivación y objetivos propuestos

En la actualidad, la gran mayoría de los sistemas de información se construyen en el contexto del modelo cliente/servidor tradicional, es decir, por medio de arquitecturas de dos capas.

Otra tendencia que se observa en el desarrollo de sistemas de información es el uso del paradigma de orientación a objetos para modelar el funcionamiento interno del sistema.

En estos sistemas, la lógica de la aplicación se ejecuta en cada uno de los clientes que interactúan con un servidor de bases de datos para obtener y modificar dicha información. A grandes rasgos, en una arquitectura de dos capas, las "máquinas cliente" deben manejar la interacción con el usuario, ejecutar la lógica de la aplicación y tener la capacidad de poder interactuar con el servidor de bases de datos.

Por otro lado, gracias a la explosión del uso de Internet, se empezaron a necesitar sistemas que puedan ser ejecutados en equipos más chicos y de forma remota. En estos casos, una arquitectura de dos capas tiene dos limitaciones muy fuertes: por un lado, la lógica de la aplicación no puede ejecutarse en una maquina cliente con poca capacidad de procesamiento, y por otro lado, no se pueden intercambiar los datos entre los clientes y el servidor de bases de datos de la misma forma en la que se hace en una red local ya que los tiempos de transmisión son mucho más importantes.

Para superar estos inconvenientes, en los últimos tiempos se han desarrollado sistemas con una arquitectura diferente: **arquitectura de tres capas**. En este tipo de arquitectura **las máquinas clientes sólo manejan aspectos relacionados con la**

**interacción con el usuario mientras que la lógica de la aplicación se ejecuta en uno o más servidores dedicados a esta tarea.**

El principal cambio está dado por el hecho de que se dividen las funciones del cliente convencional de una arquitectura de dos capas: la lógica de la aplicación que antes se ejecutaba en el cliente ahora corre en una capa intermedia. Este nuevo componente, además de correr el núcleo de la aplicación, debe soportar múltiples clientes de manera concurrente.

Desarrollar un sistema con una arquitectura de tres capas implica que tendremos los siguientes desafíos:

- El servidor de la aplicación debe soportar múltiples clientes que acceden al sistema concurrentemente.
- Los clientes deben poder interactuar remotamente con el servidor de la aplicación.
- Los puntos de entrada al sistema deben ser identificados para poder medir y controlar el estado de ejecución de cada uno de los clientes.

Lo más difícil e interesante en la construcción de sistemas con arquitecturas de tres capas es construir el componente que soporta a los distintos clientes: el **"Middle Tier".**

Por lo tanto, el objetivo propuesto en este trabajo es **definir un modelo para el desarrollo de aplicaciones con arquitecturas de tres capas en el contexto del paradigma de orientación a objetos**. Dicho modelo ha de permitir particionar la funcionalidad del sistema y establecer de manera precisa los puntos de entrada al núcleo de la aplicación.

# Resultados

El principal resultado de este trabajo es la **definición de servicios de aplicación.** Los servicios de aplicación proveen toda la funcionalidad de una aplicación en particular, es decir, todo lo se pueda hacer con el sistema debe hacerse a través de un servicio de aplicación. De esta forma, los clientes pueden interactuar con el servidor conociendo que servicio desean ejecutar y que parámetros son necesarios para iniciar la ejecución.

Para la definición de los servicios de aplicación se utilizan como base el modelo de casos de uso presentado por *Jacobson* [Jacobson92], los casos de uso esenciales de *Constantine* [Constantine01] y el análisis esencial de sistemas de *McMenamin y Palmer* [McMenamin84].

Por otro lado, aunque un servicio de aplicación sólo pueda ser ejecutado en el contexto de una aplicación en particular, en este trabajo se desarrolló un **framework de clases** para el desarrollo de servidores para aplicaciones de tres capas basadas en servicios de aplicación. Esto implicó diseñar y programar un modelo de objetos que permita incorporar la definición de **servicios de aplicación, el envío y la recepción de mensajes remotos, soporte de acceso concurrente a los clientes de la aplicación, y manejo de la persistencia de objetos.**

El uso de servicios de aplicación y el framework desarrollado fueron probados con una aplicación ejemplo que permite hacer desarrollo de software en equipo contra un ambiente de programación Squeak [Ingalls97, Squeak] de manera remota.

# Relación de la tesis con trabajos similares

En la mayoría de los artículos y documentos consultados en este trabajo sobre arquitecturas de tres capas, los autores se limitan a definir la existencia de la nueva capa (el middle-tier), la funcionalidad esperada y aspectos técnicos tales como el manejo de la concurrencia, ventajas de pools de conexiones a las bases de datos y protocolos de comunicación [Ewald97, Bernstein97 , IBM, Orfalli99, Renzel97].

En este trabajo, además de analizar puntos de carácter tecnológico, se buscó darle otra mirada al problema: Nos pusimos en el lugar de los desarrolladores, quienes, además de tener en cuenta los problemas para resolver en un sistema con una arquitectura de tres capas (estos problemas son los mencionados en la mayoría de los trabajos encontrados), deben diseñar un modelo de objetos que resuelva la esencia del sistema, la lógica de la aplicación.

La definición de servicios de aplicación es un aporte al desarrollo de sistemas con arquitectura de tres capas. Los servicios de aplicación sirven como nexo entre la complejidad en el desarrollo de sistemas con arquitectura de tres capas y el modelo de objetos de la aplicación.

# Estructura de la tesis

En el capítulo 2 se describen el modelo cliente/servidor y las características de las arquitecturas de dos capas.

En el capítulo 3 se introduce el concepto de arquitectura de tres capas, sus ventajas con respecto a las arquitecturas de dos capas y se analiza en detalle cada una de las capas: interacción con los usuarios, lógica de la aplicación y almacenamiento persistente de los datos.

En el capítulo 4 se analizan los puntos que impiden desarrollar sistemas con arquitecturas de tres capas utilizando los mecanismos conocidos de distribución de objetos. Eso se usa como argumento para buscar una alternativa y es ahí donde se propone el uso de servicios de aplicación. También se analiza el modelo de casos de uso y se propone su utilización para la identificación de los servicios de aplicación.

En el capítulo 5 se describen los principios que guiaron el desarrollo concreto de este trabajo: el framework. También se presentan las decisiones tomadas para implementar la comunicación entre los clientes y el servidor, el modelo elegido para manejar la interacción con el usuario y el ambiente elegido para el desarrollo.

En el capítulo 6 se describen las clases principales que se desarrollaron en el framework. Se muestra cómo se le dio forma a las ideas propuestas en el capítulo 4 utilizando como marco los principios enumerados en el capítulo 5.

En el capítulo 7 se muestra cómo se implementó la persistencia de los objetos. Si bien este es un aspecto propio de la implementación del framework, se decidió que tuviese un capítulo aparte (en vez de describirlo en el capítulo 6) porque, como es conocido, el manejo de la persistencia no es un tema trivial.

En el capítulo 8 se analiza la aplicación ejemplo con la que se probó el concepto de servicios de aplicación y el framework desarrollado. Esta aplicación permite hacer desarrollo en equipo contra un ambiente Squeak remoto.

En el capítulo 9 se detallan las conclusiones y los puntos que pueden dar lugar a posibles trabajos futuros.

En resumen, este trabajo de grado **propone un modelo para la construcción de sistemas con arquitecturas de tres capas en el contexto del paradigma de orientación a objetos**. La tecnología de orientación a objetos tiene más de 25 años de antigüedad y mucho se ha escrito sobre ella. Aunque se asume que el lector conoce los

**conceptos principales del paradigma**, se enumeran brevemente estos conceptos en el **apéndice A**.

Por último, en el **apéndice B**, se describe la sintaxis de **Smalltalk**. El ambiente Squeak (una implementación de Smalltalk) fue elegido para el desarrollo del framework que se describe en los capítulos 6 y 7; es por eso que un conocimiento básico de la sintaxis de Smalltalk ayudará al lector a comprender algunos fragmentos de código que se encuentran en esos capítulos.

# Capítulo 2

# El Modelo Cliente/Servidor y las Arquitecturas de Dos Capas

En este capítulo se describe el modelo cliente/servidor y las características de las arquitecturas de dos capas. Esto le dará al lector un marco de la tecnología existente en el desarrollo de sistemas de información tradicionales.

## El modelo cliente/servidor

Resulta difícil dar una definición del modelo Cliente/Servidor que abarque todos sus aspectos y las tecnologías involucradas ya que hoy en día en la industria no encontramos una definición que tenga un consenso general.

Sin embargo, entre las **características esenciales** de las aplicaciones cliente/servidor encontramos las siguientes [Orfali99]:

- Servicios.
  El servidor es, principalmente, un proveedor de servicios; el cliente es un consumidor de los servicios provistos por el servidor.
- Recursos compartidos.
  Un servidor puede atender a muchos clientes al mismo tiempo.
- Protocolos asimétricos.
  Existe una relación de muchos a uno entre los clientes y el servidor.
- Transparencia de ubicación.
  El servidor puede estar en cualquier máquina en una red.
- Intercambio basado en mensajes.
  Los clientes y los servidores son sistemas débilmente acoplados que interactúan mediante algún mecanismo de pasaje de mensajes.
- Encapsulamiento de servicios.
  El cliente pide un servicio y es tarea del servidor decidir cómo se llevará a cabo. Si la interface entre los clientes y el servidor no cambia, el servidor puede ser modificado sin afectar a los clientes.
- Escalabilidad.

Los sistemas cliente/servidor pueden crecer horizontalmente (significa agregar nuevos clientes sin un gran impacto en la performance) o verticalmente (significa migrar el servidor a una máquina más poderosa o distribuir la carga del servidor entre más servidores).

Las características previamente mencionadas son ideales, pero no todas las aplicaciones "cliente/servidor" cumplen con todos los puntos enunciados. Por ejemplo, podemos tener una aplicación cliente/servidor sin transparencia de la ubicación del servidor.

Algunos de los ejemplos típicos de **aplicaciones cliente/servidor** son:

- Servidores de archivos.
  Son unas de las aplicaciones más simples usada para transmitir datos. Son usados para manejar archivos compartidos mediante una red.
- Servidores de bases de datos.
  Aquí el cliente realiza pedidos usando un lenguaje de consulta (generalmente SQL) para acceder a los datos. Este esquema es el usado en la mayoría de las aplicaciones departamentales.
- Servidores de transacciones.
  El cliente le transmite al servidor un conjunto de pedidos de servicios que deben ser ejecutados como una unidad de trabajo. Estos procedimientos remotos residen en el servidor, por lo tanto, la cantidad de mensajes intercambiados entre el cliente y el servidor es menor que con un servidor de bases de datos en donde se intercambian mensajes por cada sentencia SQL que debe ser ejecutada**.**
- Servidores de aplicaciones para Internet.
  En la forma más simple, un "web server" retorna documentos cuando el cliente se los pide por el nombre (usando HTTP como el protocolo de comunicación).

La estructura de una aplicación cliente/servidor puede variar según el tipo de aplicación que se tenga que desarrollar y la demanda de trabajo que tenga que soportar. Entonces, ¿Cuáles funciones corresponden al cliente y cuáles al servidor?, ¿En dónde corren los procesos cliente y servidor?

En una aplicación mono-usuario, el cliente y el servidor corren en la misma máquina, lo único que se requiere es que el sistema operativo sea capaz de ejecutar a ambos de manera concurrente. Otro caso es el de una aplicación departamental basada en una LAN, en donde tenemos varios clientes que se conectan a un único servidor. En esta configuración, el modelo es bastante simple, los clientes pueden conocer al servidor mediante algún archivo de inicio, la administración de la red no presenta grandes dificultades y se puede implementar seguridad al nivel de máquina. La situación cambia en una aplicación empresarial de gran porte, en donde existen varios servidores heterogéneos. Aquí se busca proporcionar a los clientes la ilusión de un único sistema, es decir, el cliente no debería conocer cuál servidor atendió cierto pedido y dónde se encuentra este. Los servidores se pueden dividir según la función provista, los recursos que administran, etc. Los servidores se pueden replicar ya sea debido a cuestiones de seguridad o por requerimientos de velocidad.

En el modelo cliente servidor existen dos alternativas en cuanto a la distribución de tareas entre el servidor y los clientes. Podemos tener **"servidores pesados y clientes livianos"** o **"servidores livianos y clientes pesados"**.

El caso de los servidores pesados se da generalmente en los ambientes en donde existe un mainframe que se encarga de manejar el almacenamiento de los datos y la lógica de la aplicación. El cliente suele ser una "terminal boba" (sin capacidad de procesamiento) que lo único que hace es mostrar una interface (generalmente textual) al usuario y pasar los comandos al servidor.

Por otro lado, tener clientes pesados es el ejemplo más común en aplicaciones Cliente/Servidor de mediano porte. El caso típico es tener muchos clientes que acceden a un servidor de bases de datos. Además de realizarse la interacción con el usuario (generalmente usando una interfaz gráfica), en la parte cliente se debe conocer cómo están almacenados los datos y se debe manejar la lógica de la aplicación. Los lenguajes y herramientas de programación tipo Visual Basic o Delphi entran en esta categoría.

**Componentes del Modelo Cliente/Servidor**

El modelo cliente/servidor está integrado por tres componentes principales: **el cliente, el servidor y el middleware.**

El cliente inicia la comunicación con el servidor mediante un pedido de servicio. Este pedido puede ejecutarse a partir de una interacción con un usuario o simplemente de manera automática. La manera en que inicia el pedido depende de la aplicación y del tipo de pedido realizado. Si el pedido se origina a partir de una interacción con el usuario, generalmente se utiliza una interface gráfica. Las aplicaciones-cliente requieren algún mecanismo de comunicación para poder comunicarse con el servidor. También es necesario un mecanismo de *multi-threading* para poder manejar conexiones con el servidor en background. De esta forma, se le permite al usuario interactuar con el sistema al mismo tiempo en que se está comunicando con el servidor.

El servidor corre la parte crítica de la aplicación, su misión es servir a múltiples clientes que tienen interés en los recursos que él administra. La mayor parte del tiempo, el servidor está esperando pedidos de algún cliente, estos pedidos se reciben en sesiones de comunicación en la forma de mensajes. Un servidor puede asignar una sesión de comunicación dedicada para cada cliente o puede tener un conjunto (pool) de sesiones disponibles que son asignadas dinámicamente. Los pedidos de los clientes deben ejecutarse concurrentemente para evitar que un solo cliente consuma todos los recursos del sistema. La ejecución concurrente de pedidos introduce problemas clásicos de concurrencia tales como integridad en el acceso a recursos compartidos, inanición de alguna tarea, y deadlock si se usan locks para acceder a los recursos compartidos. Además, debe proveer algún mecanismo de prioridades basado en el servicio pedido y en la identidad del cliente que realizó el pedido.

Generalmente, el servidor corre aplicaciones de **"misión crítica":** si deja de funcionar la empresa u organización no puede realizar sus actividades normalmente, por lo que la disponibilidad es esencial en estos sistemas.

El middleware corre en el cliente y en el servidor, es el software que se necesita para soportar interacciones entre clientes y servidores y permitir a los clientes acceder a los servicios soportados por el servidor. El middleware empieza a partir de que el cliente hace una llamada mediante un API, se usa para rutear la transmisión del pedido hasta el servidor y análogamente para la respuesta. El middleware no sólo incluye el software que permite que un cliente y un servidor se comuniquen, sino también el software para realizar comunicaciones entre distintos servidores (la interacción entre servidores también es cliente/servidor ya que un servidor es cliente de otro y viceversa).

# Arquitecturas de dos capas

La mayor parte del desarrollo de sistemas de información se realiza utilizando una arquitectura de dos capas (2-Tier). El sistema se divide en bases de datos compartidas y aplicaciones que acceden a estos datos. En la figura 2.1 podemos observar los componentes de una arquitectura de dos capas.

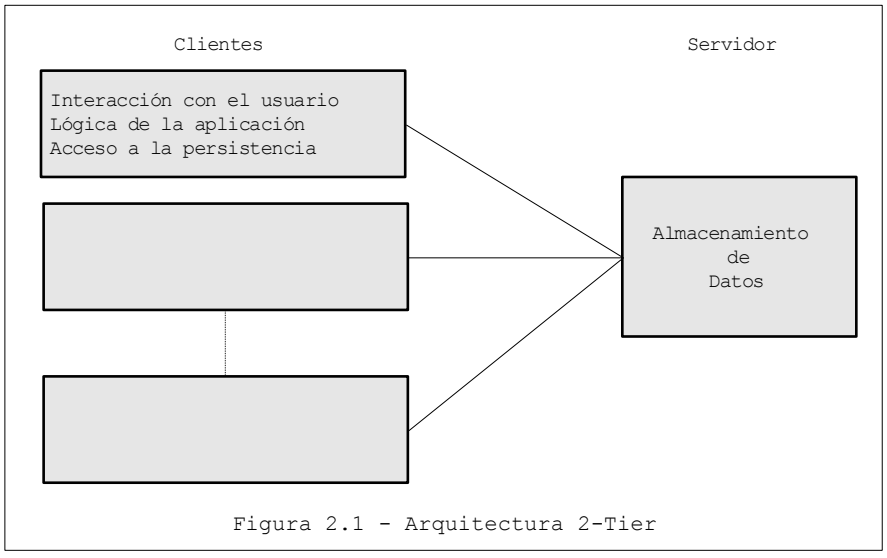



Figura 2.1 - Arquitectura 2-Tier

Las aplicaciones cliente manejan la presentación de la aplicación al usuario final y además procesan la mayor parte de la lógica de la aplicación. Cuando se necesita acceder a los datos le envían los pedidos a un servidor de bases de datos (DBMS) que los procesa y retorna los datos correspondientes.

Las bases de datos son contenidas en un servidor con suficiente espacio en disco y capacidad de procesamiento para soportar las demandas de los usuarios. Por otro lado,

la administración (estructura, políticas de seguridad, etc) es realizada por un grupo de personas dedicadas a esta función.

Las arquitecturas de dos capas tienen las siguientes ventajas:

- Los desarrolladores de aplicaciones tienen bastante experiencia con este tipo de arquitectura y por lo tanto el desarrollo se simplifica.
- Existen muchas herramientas en el mercado (lenguajes de cuarta generación, manejadores de bases de datos, etc) para desarrollar aplicaciones de este tipo.
- La mayoría de las organizaciones tienen datos que necesitan control y mantenimiento central.
- No es necesario implementar el manejo de concurrencia y el soporte de transacciones ya que estos puntos son resueltos por el DBMS.

y las siguientes desventajas:

- Las bases de datos **no proveen lenguajes de programación estandarizados que sean computacionalmente completos**. Esto implica que muchas veces se deban mover grandes volúmenes de datos al cliente para que este los procese y obtenga la información buscada.
- **La representación de los datos no está encapsulada**, dejando bastantes aspectos del control de integridad al programador de aplicaciones. Esto hace muy difícil cambiar la estructura de los datos sin tener que cambiar las aplicaciones que los acceden.
- Generalmente no se puede tener una representación adecuada del dominio de la aplicación por falta de construcciones de modelado.
- Los archivos planos y las bases de datos jerárquicas tienen serios límites en la estructura de los datos.
- La transmisión de datos sobre la red es alta ya que muchas sentencias SQL son enviadas y los datos seleccionados deben ser transmitidos para que puedan ser analizados por el cliente.
- Los clientes deben tener **capacidad de procesamiento** para correr la lógica de la aplicación y además interactuar con el usuario.

Las empresas que desarrollan bases de datos han intentado aliviar esta situación haciendo posible correr código en el servidor de bases de datos. El mecanismo utilizado es conocido como "Stored Procedures" (procedimientos almacenados).

Básicamente, un **stored procedure** es una **función que corre en un servidor de bases de datos relacionales**. Para escribir stored procedures, generalmente se usa una mezcla de SQL y algún lenguaje propietario de la base de datos en cuestión.

Usando stored procedures no es necesario que los datos crudos, resultado de una sentencia SQL, viajen hasta el cliente para que este los procese. Invocando al procedimiento, los datos se manipulan en la base de datos y se transmite el resultado al cliente (se economiza el uso de las redes y el procesamiento en los clientes).

Los stored procedures son una característica muy poderosa de las bases de datos modernas, pero tienen importantes puntos en contra: La **portabilidad** es muy **limitada** debido a que se escriben en un lenguaje propietario, esto no sólo afecta el

cambio a una base de datos distinta, sino también a diferentes versiones de la misma base de datos. Los stored procedures aumentan considerablemente el acoplamiento a la estructura de la base de datos ya que estos acceden directamente a las tablas, este acoplamiento reduce la flexibilidad. En consecuencia, algunos autores no recomiendan el uso de stored procedures [Ambler99].

# Resumen del capítulo

En este capítulo presentamos el **modelo cliente/servidor** y las características principales de las arquitecturas de dos capas. Esto nos sirve como base para que en el capítulo siguiente analicemos las **diferencias entre los dos tipos de arquitectura** y presentemos las **ventajas de las arquitecturas de tres capas** con respecto a las de dos capas.

# Capítulo 3

# Arquitecturas de Tres Capas

En este capítulo se enumeran las ventajas de una arquitectura de tres capas con respecto a una arquitectura de dos capas. Luego se analizan las tres capas: interacción con el usuario, dominio de la aplicación y persistencia, y se exploran alternativas en cuanto a su implementación.

## Introduciendo una nueva capa

En una arquitectura de tres capas (3-Tier) **se introduce una nueva capa en donde se ejecuta el núcleo de la aplicación**. Esta nueva capa surge de dividir la capa cliente de una arquitectura 2-Tier. Se le quitan responsabilidades al cliente. Mientras que en una arquitectura de dos capas la lógica de la aplicación corre en el cliente, ahora sólo se manejará la interacción con el usuario y la comunicación con el servidor de la aplicación.

En la siguiente figura se pueden observar gráficamente los cambios entre los dos tipos de arquitectura:

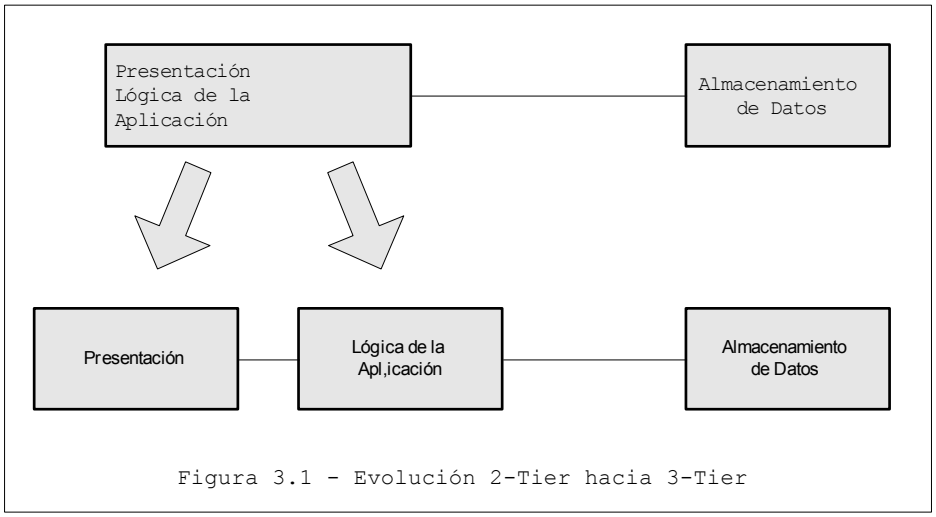



Figura 3.1 - Evolución 2-Tier hacia 3-Tier

La principal diferencia es que el núcleo de la aplicación se correrá en uno o más servidores a los que accederán los clientes. Esto hace necesario la existencia de un protocolo para la comunicación entre los clientes y el servidor. Además, se necesita que el código de la aplicación compartido por los distintos clientes sea "thread-safe", es decir, que no se generen inconsistencias por causa de la ejecución concurrente y el acceso a recursos compartidos. Más adelante, estos puntos serán analizados en detalle.

Al implementar una arquitectura 3-Tier se obtienen los siguientes beneficios:

- **Las aplicaciones pueden desarrollarse en base a un modelo del dominio de la aplicación**. No necesitan conocer ni la ubicación física ni la estructura interna de la información almacenada en la base de datos, sólo se hacen dependientes de un modelo lógico del dominio de la aplicación.

- **Disminuye el tráfico en las redes** ya que, a diferencia de una arquitectura de dos capas, no es necesario transmitir los datos a los distintos clientes para que estos los procesen.

- **Baja el procesamiento en el servidor de la base de datos** ya que se utilizan menos conexiones. En una arquitectura de dos capas se requiere una conexión por cliente mientras que en una arquitectura de tres capas usa un número mucho menor de conexiones por servidor de aplicación.

- **Se reducen los requerimientos de procesamiento en los clientes**. En una arquitectura de dos capas el cliente debe correr la totalidad de la aplicación. Ahora en el cliente sólo se manejarán aspectos de la interacción con el usuario y acceso al servidor de la aplicación. Este punto es muy importante ya que podemos usar "clientes livianos" ejecutando la aplicación en dispositivos menos poderosos que una computadora personal (por ejemplo, una computadora de mano).

- **Disminuye el costo de instalación y actualización de software en los clientes**. Las modificaciones que se hagan a la lógica de la aplicación se actualizan en el servidor de la aplicación y el cambio es visto inmediatamente por los clientes. Si las modificaciones afectan parte del software que corren los clientes, como sólo se manejan aspectos de interacción con el usuario, el tamaño de dicho software permite que las actualizaciones puedan ser descargadas desde la red.

A continuación se analizarán en detalle cada una de las capas en una arquitectura de tres capas, y se exploran alternativas en cuanto a la implementación.

# Interacción con el usuario

En esta capa se maneja la interacción entre el usuario y el sistema, generalmente se utiliza algún tipo de interface gráfica.

### Construcción de interfaces gráficas con tecnología de objetos

En el ámbito de la tecnología de objetos, existen diferentes frameworks para la construcción de interfaces gráficas. Los más conocidos son MVC (Model-View-Controller) o sus variantes MV (Model-View) y MVP (Model-View-Presenter) [Krasner88, Potel96].

La idea principal en estos frameworks es que el dominio de la aplicación no debe manejar aspectos de la interface con el usuario, y en la interface no se deben manejar aspectos del dominio de la aplicación. Gracias a esta **independencia entre los roles**, es posible crear software más limpio y flexible.

Los Views y los Controllers en MVC o los Presenters en MVP generan la interface gráfica en base al Model y manejan los comandos del usuario. El Model comúnmente es un objeto al nivel del dominio de la aplicación. El Model no debe tener ninguna conciencia de la existencia de una interface gráfica, es por esto que puede ser reutilizado en situaciones nuevas, que podrían tener requerimientos completamente diferentes a los previstos inicialmente para una interface gráfica en particular. Esto se logra mediante la aplicación del pattern observer [Gamma95].

En el **pattern observer** tenemos dos roles participantes: los observadores y el observado. Los observadores tienen referencias directas al observado y le pueden enviar mensajes en cualquier momento; mientras que el observado sólo tiene referencias indirectas a sus observadores (a priori no conoce si tiene cero, uno, o más observadores) y sólo les puede notificar de cambios en su estado. Éste es un buen ejemplo del uso del polimorfismo en un diseño de objetos, no importa la clase de los observadores, lo que importa es que respondan al protocolo para manejar notificaciones de cambios del observado.

En MVC, MV o MVP, el rol de observers lo juegan las Views mientras que el observado es el Model.

En una arquitectura de tres capas, resulta indispensable el hecho de que en la interface gráfica no se manejen cuestiones relacionadas con el dominio de la aplicación, debido a que los distintos componentes correrán remotamente entre ellos. Sin embargo, no se podrá usar el pattern observer en su especificación original ya que la cantidad de mensajes a transmitir entre el modelo y sus observadores es muy alta para un sistema distribuido (tenemos el problema de notificación de cambios distribuida).

**Alternativas para la construcción de interfaces gráficas en arquitecturas 3-tier**

La estructura de la capa de interacción con el usuario se puede dividir en dos partes: **presentación y control del diálogo** [Renzel97]. En la presentación sólo se manejan mecanismos para la interacción con el usuario (ventanas, menús, formularios) y en el control del diálogo se define la lógica para acceder a los objetos que componen el dominio de la aplicación y se mantiene el estado de la interacción entre el usuario y el sistema.

Estos dos componentes son la base para distribuir el manejo de la interacción con el usuario entre los clientes y el servidor.

Tenemos dos opciones: **presentación distribuida e interface con el usuario remota**. En el caso de una presentación distribuida, el cliente sólo corre la presentación mientras que el control del diálogo se maneja en el servidor. Por otro lado, en una interface con el usuario remota, la interface está basada en el cliente, este corre tanto la presentación como el control del diálogo.

**Presentación distribuida**

En este caso, los clientes sólo deben manejar aspectos puramente relacionados con la parte gráfica y aspectos de interacción básicos. Esto implica que los requerimientos de procesamiento en el cliente se mantienen relativamente bajos. Por otro lado, el servidor debe dirigir el control del diálogo, esto implica conocer el estado actual de la interacción.

Para manejar la interacción con el usuario se ejecuta el siguiente ciclo: la interface se genera en el servidor y luego es transmitida al cliente en donde es interpretada y mostrada al usuario. El usuario ejecuta alguna acción y el resultado de la interacción es transmitido al servidor en donde es procesado y, si es necesario, se genera una nueva interface o se modifica la actual.

Los mensajes que se intercambian entre el cliente y el servidor abarcan solamente cuestiones visuales y de interacción básica (por ejemplo, informar la selección de un elemento en una lista, notificar al servidor que se cambio el texto en un campo, etc). A causa de esto, crece considerablemente la cantidad de mensajes que intercambian el cliente y el servidor.

Tenemos dos alternativas para implementar la interacción con el usuario utilizando una presentación distribuida: **utilizar formularios y web-browsers o transmitir una especificación de la interface gráfica al cliente de la aplicación**.

En la primer alternativa, un browser (o navegador) corre en la máquina cliente y se comunica con el servidor (un "Web Server") usando algún protocolo de comunicación como HTTP.

Cuando el servidor recibe un pedido del cliente genera una nueva "página" (la página es la interface generada) y es transmitida de vuelta al cliente. El uso de protocolos estandarizados como HTML para describir las páginas y HTTP (sobre TCP/IP) para transmitirlas permite que cualquier navegador compatible pueda ser cliente de nuestra aplicación.

Esta opción es muy atractiva debido a que prácticamente no se debe instalar software en los clientes, ya que es muy común que la máquina cliente tenga instalado un "web browser", sin embargo, tiene dos puntos en contra muy importantes [Seacord98].

. Por un lado resulta muy complicado construir una interface compleja usando solamente páginas web, y además los principales browsers existentes en el mercado agregan extensiones propietarias al lenguaje HTML para incorporar contenido activo a las páginas. Esto implica que es posible que la página no pueda ser utilizada en distintos navegadores.

Otro problema en el uso de navegadores es la limitación para acceder a recursos locales en la máquina cliente, generalmente por cuestiones de seguridad.

La alternativa al uso de navegadores consiste en transmitir al cliente una interface gráfica clásica, como la que proveen la mayoría de los sistemas operativos. En esta opción tenemos diferentes niveles para describir la interface gráfica que se transmitirá al cliente.

Una opción de bajo nivel es que el servidor transmita al cliente una imagen con el estado de la pantalla y el cliente debe transmitir al servidor todos los eventos generados por el usuario (generalmente son eventos de teclado y mouse). El servidor procesa estos eventos y actualiza la imagen de la pantalla transmitiéndosela al cliente.

La principal contra de un sistema de este tipo es la alta cantidad de mensajes que transitan por las redes, debido a que cada cambio que se produzca en la interface a causa de la intervención del usuario deberá ser transmitido al servidor para que sea procesado, y si es necesario se deberá actualizar el estado de la interface en el cliente.

Una alternativa muy interesante en cuanto al desarrollo de presentaciones distribuidas está dado por tecnologías como Ultra Light Client [IBM99, Canoo] o Classic Blend [AppliedReasoning01].

En estas soluciones el servidor en lugar de transmitir una imagen con el estado de la interface gráfica, transmite una especificación de cómo debería verse la interface. Los clientes corren un software que interpreta esta especificación y construye la interface gráfica. Con respecto al manejo de eventos generados por el usuario, en la especificación de la interface se pueden elegir cuales eventos son enviados al servidor para que este los procese.

Este tipo de interface es mucho más económica en cuanto a la cantidad de mensajes transmitidos por la red entre los clientes y el servidor. Por ejemplo, supongamos que tenemos un campo en donde el usuario debe ingresar algún valor. En el primer caso, cada vez que el usuario ingrese una letra desde el teclado, se debe mandar un mensaje al servidor informándole del evento y el servidor debe procesar el evento y generar una nueva imagen con el estado de la interface gráfica. Mientras que en la segunda opción, sólo es necesario transmitir el evento de cambio de valor de un campo cuando el usuario termine de ingresar el nuevo valor.

**Interface con el usuario remota**

Aquí, el cliente corre la totalidad de la interface con el usuario. Los requerimientos de capacidad de procesamiento en el cliente son mayores que en el caso de una presentación distribuida y la información intercambiada entre el cliente y el servidor está asociada a la aplicación, es decir, se intercambian mensajes con semántica de la aplicación. A causa de esto, el software que se debe instalar en los clientes es más

complejo y de mayor tamaño, lo que es una contra en el momento de la instalación o actualizaciones de la aplicación. Por otro lado, si se lo compara con una interface generada en el servidor, disminuye el tráfico usado para la comunicación ya que el cliente posee más inteligencia y puede resolver algunos aspectos de la interacción con el usuario sin necesidad de delegarlo al servidor.

Como dijimos anteriormente, en una interface con el usuario remota, el control del diálogo se maneja en el cliente. Esto implica una mayor complejidad en cuanto al desarrollo de la aplicación, pero también más libertad para hacer optimizaciones apropiadas a la aplicación en particular.

# Capa intermedia: dominio de la aplicación

Esta capa es llamada "Middle-Tier" (capa del medio o capa intermedia) debido a que se ubica entre la capa de interacción con el usuario y la de almacenamiento de la información.

Aquí se encuentran los objetos que componen el dominio de la aplicación (por ejemplo, en una aplicación bancaria encontraríamos cuentas de clientes, operaciones financieras, productos financieros, etc).

Para desarrollar la aplicación usando una arquitectura de tres capas, debemos considerar que el sistema será usado por múltiples usuarios concurrentemente, que el software cliente accederá remotamente a nuestros objetos en el servidor y que la capa de presentación es remota con respecto al Middle-Tier y por lo tanto es costoso actualizarla.

Para construir el modelo de objetos del Middle-Tier en una arquitectura de tres capas, podemos clasificar los problemas a resolver en tres tipos:

- Tenemos que resolver la **complejidad inherente al dominio de la aplicación**, es decir, diseñar un modelo que soporte los requerimientos funcionales de la aplicación (la funcionalidad esperada del sistema). Estos problemas, no son propios de una arquitectura de tres capas, ocurren en el desarrollo de cualquier sistema de información (y por lo tanto no profundizaremos en estos puntos).

- En una arquitectura de tres capas, el núcleo de la aplicación debe soportar a múltiples clientes interactuando con él de manera **concurrente**. Esto implica que se tendrá que dar soporte de concurrencia y transacciones.

- Por último, además de las cuestiones anteriores, debemos resolver cómo se va a **dividir la funcionalidad del sistema**. Una arquitectura de tres capas requiere que especifiquemos de qué forma interactuarán los clientes con los objetos que forman el núcleo de la aplicación. Esto no sucede en una arquitectura de dos capas donde la presentación y la aplicación corren juntas. En una arquitectura de tres capas, los clientes acceden remotamente a la funcionalidad de la aplicación por lo que se deben definir puntos de acceso a la aplicación y cómo será la

interacción en esos puntos. Esto es necesario para evitar una situación caótica en la evolución del sistema.
En el capítulo siguiente se profundiza en este último tema, y se propone un modelo basado en servicios al nivel de la aplicación para desarrollar el Middle-Tier.

# Almacenamiento persistente de los datos

El almacenamiento persistente es manejado por uno o más DBMSs (gestores de bases de datos). La tecnología de bases de datos es bastante madura, y la mayoría de los productos existentes soportan manejo de transacciones, concurrencia, lenguajes de consulta, especificación de cuestiones de seguridad, resguardo de datos, recuperación de fallas, etc.

En esta sección hacemos un análisis de la capa de persistencia mirándola desde la capa intermedia. Lo que se busca es brindar persistencia a los objetos que componen el dominio de la aplicación.

En muchas situaciones los objetos deben sobrevivir al proceso que los crea y a los procesos en donde son usados. Estas situaciones incluyen a la mayoría de las "aplicaciones reales", es estos casos la persistencia es un requerimiento. En una arquitectura de tres capas, la capa de almacenamiento brinda soporte de persistencia a los objetos que forman el middle-tier.

Un objeto (de software) ocupa cierto espacio y existe en un determinado período de tiempo. Según Booch [Booch94], el espectro de persistencia abarca los siguientes casos:

- Resultados momentáneos en la evaluación de expresiones.
- Variables locales en activaciones de procedimientos.
- Variables globales y elementos de un "heap" cuya duración es mayor a su alcance.
- Datos que existen entre distintas ejecuciones de un programa.
- Datos que existen entre distintas versiones de un programa.
- Datos que van más allá de un programa.

Generalmente los lenguajes de programación manejan los tres primeros casos, y los restantes se resuelven usando algún tipo de base de datos.

El mismo autor define persistencia de la siguiente manera:

*"Persistencia es la propiedad que tiene un objeto mediante la cual su existencia trasciende el tiempo y/o espacio de su creador (el objeto continúa existiendo aunque su creador deje de existir y su localización puede ser distinta de la que tenía cuando fue creado)."*

Sin embargo, cuando nos referimos a bases de datos, abarcamos cuestiones que van más allá de persistencia pura, como son el soporte de transacciones, manejo de concurrencia, recuperación a fallas y políticas de seguridad.

Estos puntos hacen diferencia entre almacenamiento de datos puro (usar archivos planos) y bases de datos. Las dos opciones más comunes para implementar la persistencia de los objetos son:

- **Bases de Datos Relacionales**.
- **Bases de Datos Orientadas a Objetos**.

**Persistencia en bases de datos relacionales**

El modelo relacional y el modelo de objetos son esencialmente diferentes e integrarlos es complicado.

Las bases de datos relacionales están basadas en tablas bidimensionales en las cuales cada ítem aparece como una fila en una tabla. Las relaciones entre los datos se encuentran comparando los valores en las distintas tablas.

Los sistemas de bases de datos relacionales son buenos manejando grandes volúmenes de datos, permiten recuperar datos rápidamente pero proveen poco soporte para manipular estos datos (los lenguajes de consulta no son computacionalmente completos). Por otro lado, los lenguajes orientados a objetos permiten expresar relaciones complejas entre los objetos, además de la capacidad de manipularlos.

Tratar de expresar las relaciones del modelo de objetos en bases de datos relacionales no es una tarea simple. Esto se debe a la manera en que se accede a los datos en ambos modelos. En el modelo de objetos "navegamos" por su estructura a través de referencias directas (un objeto referencia directamente a distintos objetos) mientras que con una base de datos relacional debemos duplicar datos para poder hacer intersecciones entre tablas diferentes. En un ambiente de objetos todos los objetos tienen una identidad única, cuando se almacenan los objetos en las tablas de una base de datos relacional se debe hacer explícita esta identidad a través de un identificador único. Las referencias a otros objetos se hacen usando este identificador como clave foránea en la tabla en donde se almacene el objeto en cuestión.

Para poder almacenar satisfactoriamente objetos en una base de datos relacional se debe mapear la estructura de los objetos a la de las tablas. Los mapeos que hay que realizar son los siguientes [Ambler99]:

- Mapear clases a tablas.
  Se debe especificar en qué tabla se almacenarán las instancias de las clases persistentes. Se puede asociar una tabla por clase o asociar más de una clase a una tabla.
  Otro factor que influye es el manejo de la herencia. Tenemos tres alternativas:
  - Usar una tabla para cada clase concreta.
    Cada tabla tendrá los atributos asociados a la clase que representa más los atributos heredados. Esta opción es bastante directa pero surgen complicaciones cuando se modifica alguna superclase (si se agregan o se eliminan atributos) debido a que deben actualizarse las tablas asociadas a las subclases de la clase modificada.
  - Usar una tabla para cada jerarquía.
    Esta solución es buena, la principal contra que tiene es que una tabla tendrá atributos que serán usados en algunas filas y en otras no (deberán tener valor null), esto dependerá de qué clase sea el objeto almacenado en la fila.

    - Usar una tabla por clase.
      En este caso es en donde mejor se refleja la jerarquía de clases en la base de datos, sin embargo tiene varios puntos en contra. Por un lado, el espacio usado en la base de datos crece debido al alto número de tablas, pero el problema más grave es el costo (tiempo) que lleva leer y escribir datos (generalmente se deben leer varias tablas y hacer joins entre estas).
- Mapear atributos a columnas.
- Mapear relaciones entre los objetos.
  Las relaciones pueden ser uno a uno, uno a muchos y muchos a muchos.
  El caso de uno a uno, si una instancia de la clase A tiene una referencia a una instancia de la clase B, se puede resolver con una clave foránea en la tabla de A en donde el valor de la columna sea la clave de la instancia de B referenciada.
  El caso de muchos a muchos se debe resolver con una tabla intermedia que relacione las filas de A con las de B.
  En el caso de uno a muchos, una instancia de A referencia a una colección de instancias de B. En esta situación se puede usar una tabla intermedia (al igual que en el caso anterior) o, si sabemos que una instancia de B solo puede ser referenciada por una sola instancia de A (si fuese más de una no se podría usar esta técnica), entonces puede agregarse a la tabla asociada a B una columna cuyo valor será la clave de la instancia de A que referencia a la instancia de B.

La estructura resultante de hacer los mapeos mencionados dirigirá las operaciones básicas de persistencia de los objetos: recuperar objetos desde la base, insertar objetos nuevos, borrar objetos y actualizar objetos existentes.

**Persistencia en bases de datos orientadas a objetos**

La diferencia semántica entre el modelo relacional y el paradigma de orientación a objetos condujo al desarrollo de bases de datos orientadas a objetos (BDOO). Las BDOO soportan directamente el modelo de objetos y, por lo tanto, son más simples de usar que las bases de datos relacionales en un sistema basado en objetos. Hemos visto que si usamos una base de datos relacional debemos hacer transformaciones para trabajar con los dos modelos. Cada transformación no sólo incrementa los costos de desarrollo, sino que también afecta a los tiempos de ejecución cuando se utilice la aplicación.

Una alternativa que hasta ahora no hemos mencionado consiste en usar bases de datos relacionales extendidas. Estos sistemas son bases de datos relacionales con funcionalidad agregada que maneja algunos aspectos del paradigma de objetos. Aunque se simplifiquen algunas cuestiones, las bases de datos relacionales extendidas siguen siendo bases de datos relacionales, por lo que todavía deben hacerse transformaciones y mapeos para darle persistencia a un modelo de objetos.

En una base de datos orientada a objetos, los objetos pueden ser persistidos sin necesidad de hacer transformaciones (no hay que escribir código que mapee los objetos a otras estructuras). Las bases de datos orientadas a objetos unen las nociones de almacenamiento en memoria volátil y almacenamiento persistente en un almacenamiento de un solo nivel [Loomis95].

Desde el punto de vista de un programador, lo ideal sería que la base de datos fuese "invisible", que estuviese fuertemente integrada al lenguaje. Cualquier estructura de objetos que pueda ser modelada en el lenguaje debería poder almacenarse directamente

en la base de datos. No se requiere que el desarrollador utilice un modelo para los objetos y otro para la base de datos.

Justamente, uno de los principales objetivos de las bases de datos orientadas a objetos es acercar la base de datos al ambiente de programación; esto se hace para evitar que el programador deba hacer explícito el acceso al mecanismo de persistencia. El objetivo buscado es que no existan diferencias entre trabajar con objetos que se encuentren en el medio de persistencia y con objetos en memoria.

Un modelo de objetos incluye relaciones entre estos. Hemos visto que se necesitan realizar ciertos mapeos y crear identificadores de objetos únicos para almacenar estas relaciones en una base de datos relacional. En una base de datos orientada a objetos estas relaciones son persistidas directamente. Esto no sólo disminuye la complejidad sino que también mejora el rendimiento en las consultas en donde se debe recorrer la estructura de estos objetos; la idea es que no se necesitan hacer joins entre tablas para recuperar los objetos (es conocido el alto costo que tienen las operaciones de join en una base de datos relacional).

Además de brindar persistencia a los objetos y de estar fuertemente integrada al lenguaje de programación, una base de datos orientada a objetos debe soportar las funciones clásicas de toda base de datos. Entre estas funciones encontramos el manejo del esquema de almacenamiento, control de concurrencia, soporte de transacciones, recuperación a fallas y control de acceso (seguridad).

El control de concurrencia y el soporte de transacciones permite que los objetos persistentes puedan ser compartidos por más de un usuario concurrentemente. Como en las bases de datos relacionales, se debe garantizar que la ejecución de transacciones sea serializable.

Si además de manejar estos puntos, una base de datos orientada a objetos provee la capacidad de ejecutar remotamente llamadas a métodos en el servidor y manejo de grandes volúmenes de objetos, puede ser llamada "Servidor de Objetos" [GemStone]. Un servidor de objetos soluciona muchos de los problemas que encontramos al desarrollar un sistema 3-tier usando objetos. Sin embargo, no se especifica un modelo de interacción entre los clientes y el servidor, y más importante no se definen los puntos de entrada al núcleo de la aplicación (en el capítulo siguiente se propone un modelo para superar estos problemas).

**¿Qué alternativa elegir?**

No existe una respuesta válida para la pregunta planteada. La elección depende de los requerimientos del sistema que estemos desarrollando, por ejemplo, en algunos casos no es necesario usar una base de datos, el almacenamiento puede basarse en el sistema de archivos mientras que en aplicaciones de tipo CAD una base de datos relacional no basta.

Además de los requerimientos del sistema, la política de la organización para la que se está desarrollando el sistema influirá en la elección. La mayoría de las organizaciones usan bases de datos relacionales para soportar sus sistemas actuales, es decir, confían el manejo de su información a estos sistemas de bases de datos por lo que en la mayoría de los casos prefieren utilizar esta tecnología que más allá de estar muy probada tiene fundamentos matemáticos que la hacen parecer más fuerte frente a los

otros tipos de bases de datos. Por otro lado, como hemos visto anteriormente, se requieren muchos más esfuerzos para usar bases de datos relacionales desde un sistema de objetos.

Las bases de datos orientadas a objetos están evolucionando día a día y existen algunos productos que han demostrado tener éxito en aplicaciones reales.

# Resumen del capítulo

En este capítulo analizamos las ventajas de una arquitectura de tres capas con respecto a una arquitectura de dos capas, y luego se presentaron con detalle los componentes de una arquitectura 3-tier. Fueron enumerados los diferentes modelos para implementar la interacción con el usuario (presentación distribuida e interface con el usuario remota) y se describieron con detalle los mecanismos para manejar la persistencia de los objetos.

Por otro lado, se presentaron los problemas para desarrollar el middle-tier, los cuales se analizan con más profundidad en el capítulo siguiente, donde también se propone un modelo para el desarrollo de esta capa, siempre en el contexto del paradigma de orientación a objetos.

# Capítulo 4

# Estructura del "Middle Tier"

En este capítulo se presentará un modelo conceptual para definir la interacción de los clientes con el modelo de la aplicación en una arquitectura de tres capas. Se propone el uso de servicios de aplicación y se estudia la forma de identificación de dichos servicios basándose en el modelo de casos de uso.

## Objetos distribuidos

La tecnología de objetos es muy interesante para el desarrollo de sistemas cliente/servidor flexibles debido a que los datos y la lógica de la aplicación están encapsulados en objetos. La distribución de objetos permite ubicar a los diferentes objetos en distintos lugares de un sistema distribuido.

Dos objetos son locales el uno para el otro si se encuentran en el mismo espacio de direcciones; un **objeto es remoto** con respecto a otro si se encuentran en un espacio de direcciones distinto (ya sea en máquinas diferentes o en la misma máquina).

Un objeto es una entidad que encapsula su estado y tiene comportamiento que puede ser invocado enviándole mensajes. Normalmente, los objetos "viven" en un solo programa (o sistema) y el mundo exterior no conoce su existencia y no tiene forma de acceder a ellos. Por otro lado, cuando hablamos de objetos distribuidos nos referimos a objetos que pueden encontrarse en cualquier máquina en la red y que pueden ser accedidos por clientes remotos mediante invocaciones de mensajes. Los clientes no necesitan saber en donde se encuentra el objeto, podría estar en la misma maquina o en una máquina en otro continente.

**El pattern proxy**

La mayoría de los mecanismos utilizados para distribuir objetos se basan en el pattern proxy [Gamma95, Buschmann96].

La idea es la siguiente. Supongamos que tenemos dos objetos, A y B, y que el objeto A necesita enviarle un mensaje al objeto B. En una situación normal, A tendría una referencia a B y le enviaría el mensaje directamente.

En la siguiente figura podemos observar este caso.

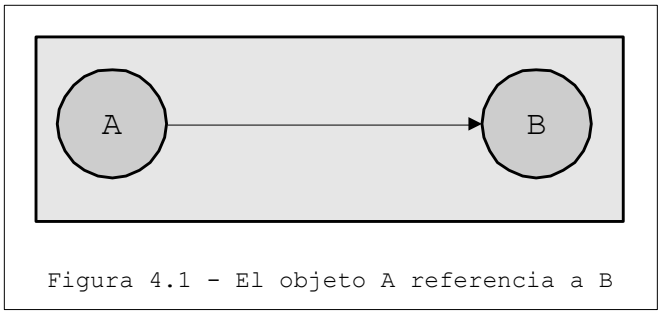



Figura 4.1 - El objeto A referencia a B

Ahora supongamos que A y B están distribuidos en dos máquinas diferentes. Esto implica que ahora A no puede tener una referencia directa a B.

Para solucionar el problema podríamos exigir que el objeto A conozca en qué lugar se encuentra el objeto B y que, para poder enviarle el mensaje, acceda a algún protocolo de bajo nivel para distribuir el mensaje por la red. Obviamente esta sería una mala solución, ya que estamos ensuciando la implementación de A para manejar el envío del mensaje. Y además, si en algún momento cambiase la ubicación de B, por ejemplo, traerlo al mismo espacio de direcciones en donde se encuentra A, la solución anterior dejaría de funcionar.

En el pattern proxy, lo que se propone es dejar al objeto A como estaba en el caso de una invocación local y se introduce un nuevo objeto que representa a B (un proxy) y que conoce cómo llegar hasta B. Entonces, cada vez que el proxy de B reciba un mensaje que era para B le reenviara el mensaje.

A continuación podemos ver gráficamente la solución utilizando un proxy.

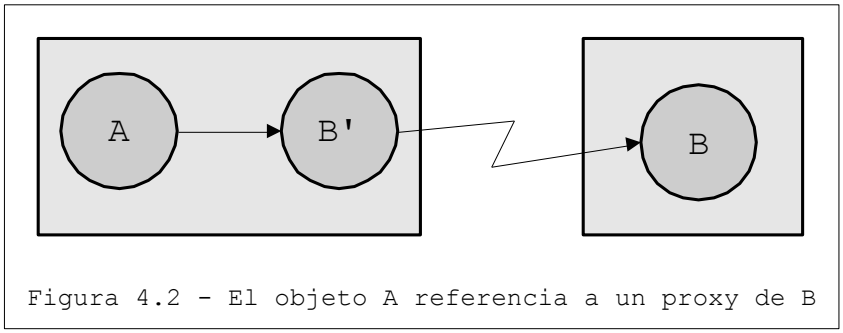



Figura 4.2 - El objeto A referencia a un proxy de B

Usando un proxy no tenemos que modificar a A, ya que para A sigue siendo una invocación local. Y en caso de que B cambie su ubicación, por ejemplo, traerlo al mismo espacio de direcciones que A, se puede reemplazar el proxy de B por B.

Como se dijo anteriormente, el uso del pattern proxy nos da la posibilidad de usar objetos distribuidos de una forma transparente. Sin embargo, en la mayoría de los casos, esta transparencia en cuanto a la ubicación real del objeto destino puede traer una serie de problemas con respecto a la performance general del sistema y su tolerabilidad a las fallas propias de la computación distribuida. A continuación se profundizará en estos puntos.

**Problemas para distribuir objetos**

Ahora que fue presentada la esencia de los objetos distribuidos, podríamos plantearnos utilizar esta tecnología para implementar una arquitectura de tres capas en donde se distribuyan los objetos que manejan la interacción con el usuario a los clientes de la aplicación y se mantengan en el servidor los objetos que manejan la lógica del sistema.

A simple vista, esta parece ser la solución ideal ya que podemos pasar de un sistema existente de dos capas a una arquitectura de tres capas de una forma bastante directa utilizando el pattern proxy y algún mecanismo de distribución.

Sin embargo, no estamos teniendo en cuenta características esenciales de la computación distribuida. Si ignoramos el hecho de que los objetos que forman el núcleo de la aplicación no se encuentran en el mismo espacio de direcciones que el código que maneja la interface con el usuario, tendremos un sistema que no podrá manejar las fallas y su performance podría llegar a ser inaceptable.

El problema surge de **asumir erróneamente que debería ser lo mismo enviar un mensaje a un objeto local que a un objeto remoto**. Y en este sentido, el uso indiscriminado del pattern proxy puede traer problemas de performance muy importantes.

Un modelo que ignore las diferencias fundamentales entre la interacción entre objetos distribuidos y objetos locales no podrá soportar requerimientos básicos de performance y robustez con respecto a fallas.

Los puntos más importantes que marcan dichas diferencias son los siguientes [Waldo94]:

- **Latencia**.
  La diferencia más notoria entre una invocación a un método sobre un objeto local respecto de un objeto remoto es la diferencia entre los tiempos de ejecución de ambas llamadas.
  El tiempo en que tarda en enviarse un mensaje remoto comparado con uno local es de varios ordenes de magnitud mayor. Y si se analizan los avances en los últimos años en cuanto a la velocidad de los microprocesadores comparándolos con las mejoras en tiempos de latencia en redes, la relación no mejorará (al contrario, la diferencia parece aumentar).

Ignorar la diferencia de performance entre invocaciones de métodos locales y remotos puede conducir a diseños cuyas implementaciones sufran de graves problemas con respecto a la performance, ya que se requerirán de muchas comunicaciones entre entidades que se encuentran en espacios de direcciones diferentes.

- **Acceso a memoria**.
  Este punto afecta a los lenguajes que permiten al programador obtener punteros a objetos relativos al espacio de direcciones. En estos lenguajes, las referencias a los objetos no son manejadas por el sistema. Los punteros en un espacio de direcciones sólo son válidos en ese espacio de direcciones, por lo que no se puede tratar de la misma forma un puntero a un objeto local y uno remoto.
  Aquí debemos aclarar que si utilizamos el pattern proxy, este punto no nos afecta ya que el proxy sabe cómo llegar al objeto real sin tener una referencia directa.

- **Falla parcial y concurrencia**.
  En el caso de la computación local tenemos dos casos con respecto a las fallas. O las fallas son totales y afectan a todas las entidades involucradas en la aplicación, o pueden ser detectadas por algún controlador de recursos locales (por ejemplo, el sistema operativo). En la computación distribuida, un componente puede fallar y los otros pueden continuar funcionando. Estas fallas no sólo son independientes, sino que no existe un agente común que pueda determinar cuál fue el componente que falló e informarle al resto de lo ocurrido. Esto implica que si no se toman medidas para manejar las posibles fallas inherentes a la invocación de un método en un objeto residente en un espacio de direcciones remoto (computación distribuida), el control puede no volver nunca, dejando al sistema en un estado de parálisis.
  Un caso extremo sería intentar que el programador asuma que todas las invocaciones de cualquier método podrían ser llamadas remotas, y entonces pedirle que controle posibles fallas (basándose en time-outs). Obviamente, una solución de este tipo ensuciaría por completo cualquier sistema.

Además de los puntos que recién mencionamos, el uso de objetos distribuidos de forma indiscriminada no soluciona todos los problemas que están relacionados con el desarrollo del sistema.

Asumamos por un momento que tenemos a nuestra disposición la tecnología para distribuir objetos de manera transparente y sin perdida de performance. Aunque esto suceda, todavía debemos resolver que procesamiento se hace en los clientes y que procesamiento se hace en el servidor, en qué estado se encuentran los distintos clientes (por ejemplo, para ejecutar procesos batch), y por último, todavía tenemos que solucionar el acceso concurrente a los recursos compartidos.

**En busca de una solución**

Como hemos analizado, hoy en día no podemos partir de un sistema tradicional con una arquitectura de dos capas y pasar directamente a uno con una arquitectura de

tres capas utilizando objetos distribuidos sin hacer alguna modificación en el flujo de mensajes existente entre los objetos que manejan la interacción con el usuario y los que manejan la lógica de la aplicación.

Todos estos problemas presentados hasta ahora sirven como motivación para buscar un modelo que sea factible para construir aplicaciones orientadas a objetos con arquitecturas de tres capas.

A continuación se introduce la idea principal de este trabajo con la que se espera resolver estos problemas.

# Servicios de aplicación

Como analizamos anteriormente, cuando desarrollamos una aplicación con una arquitectura de tres capas no podemos ignorar el hecho de que la interacción con el usuario se maneja en los distintos clientes que luego deben interactuar con los objetos remotos que forman el núcleo de la aplicación. Esto implica que debemos identificar en qué lugares del sistema se producen estas interacciones remotas.

Por otro lado, además del tema de la distribución, una arquitectura 3-tier debe soportar la existencia de múltiples clientes accediendo de manera concurrente. Esto implica que en el servidor de la aplicación necesitaremos controlar y medir de alguna manera que están haciendo los clientes de la aplicación.

Enumerando, necesitamos resolver y tener en cuenta los siguientes puntos:

- No podemos ignorar el tema de la distribución y los problemas asociados de latencia, falla parcial y concurrencia.
- Tenemos que identificar y definir de manera precisa de qué forma interactúan los distintos clientes con la aplicación, es decir, los puntos de entrada al sistema.
- En el servidor de la aplicación necesitamos observar (para poder controlar) que están haciendo los clientes en un momento dado.

Para llevar a cabo esto, en este trabajo proponemos lo siguiente:

> **Se definirán servicios de aplicación que brindarán la funcionalidad completa del sistema**.

**Los clientes piden la ejecución de servicios** y luego muestran los resultados de la ejecución (en los casos en que sea necesario).

Este conjunto de servicios se ubica entre los clientes y el modelo de objetos de la aplicación y deben proveer la funcionalidad completa del sistema, esto implica que un cliente accede a la aplicación sólo a través de servicios.

Los servicios de aplicación dependen de la aplicación en particular que se quiera desarrollar, por lo que sólo se pueden utilizar en el contexto de la aplicación para la que fueron diseñados. Por ejemplo, en una aplicación financiera un servicio de aplicación podría ser efectuar una nueva transacción bancaria.

El uso de servicios de aplicación ayuda a convivir con los problemas que se citaron anteriormente respecto a la computación distribuida.

Cuando el cliente solicita la ejecución de un servicio sabe que está interactuando con una entidad remota y se tiene la posibilidad de manejar los puntos anteriormente mencionados (latencia, fallas parciales, concurrencia).

Por otro lado, en el servidor de la aplicación se puede medir y controlar qué servicios esta ejecutando o ya ejecutó un cliente en particular.

Si particionamos la funcionalidad de la aplicación en servicios obtenemos los siguientes beneficios:

- Se divide la funcionalidad de la aplicación en componentes que pueden ser ejecutados.
  En consecuencia, queda más claro que hace la aplicación en respuesta al pedido de un cliente.

- Se reduce el número de comunicaciones entre los clientes y el servidor, sobre todo si se lo compara con una solución en donde se tratan de la misma manera los objetos locales y los objetos remotos.

- Podemos administrar de manera ordenada los problemas asociados a la computación distribuida que mencionamos anteriormente.

En la figura 4.3 se muestra la estructura de un sistema de tres capas utilizando servicios de aplicación:

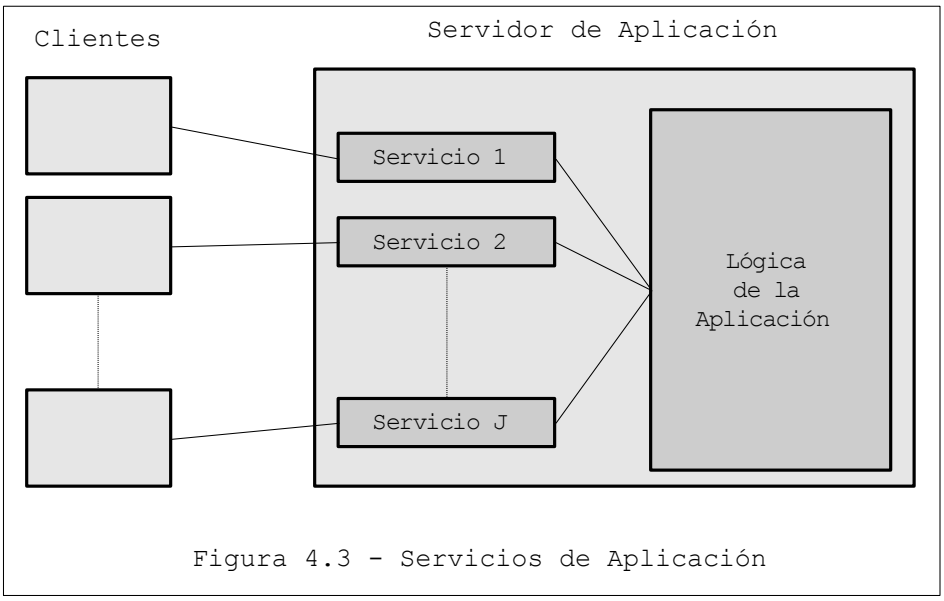



Figura 4.3 - Servicios de Aplicación

Ahora que fue presentada la idea de dividir la funcionalidad del sistema en servicios de aplicación, analizaremos las siguientes cuestiones:

¿Qué es realmente un servicio de aplicación y cómo identificamos los servicios que debería proveer el sistema?

Una vez que decidimos brindar la funcionalidad completa del sistema utilizando servicios de aplicación, esta pregunta se torna fundamental.

Para dar una respuesta estudiaremos a continuación el modelo de casos de uso propuesto por Jacobson [Jacobson92], la extensión realizada por Constantine y Lockwood en los casos de uso esenciales [Constantine01] y el trabajo de McMenamin y Palmer [McMenamin84].

¿Qué relaciones existen entre los distintos servicios de aplicación?

Aquí analizaremos dos tipos de relaciones entre los servicios de aplicación. Por un lado tenemos relaciones estructurales que entran en juego en la definición de un servicio de aplicación, y por otro lado debemos estudiar qué relaciones existen entre los distintos servicios cuando el sistema se encuentra en funcionamiento.

¿Cómo se inicia la ejecución de un servicio de aplicación?

A este punto lo podemos dividir en dos partes. Por un lado debemos resolver de qué forma se refiere a un servicio un cliente de la aplicación, es decir, cómo lo identifica. Y además esta la cuestión de, una vez identificado el servicio de aplicación, cómo se lo inicializa para que pueda ser iniciada su ejecución.

# El Modelo de casos de uso

Ahora estudiaremos el modelo de casos de uso, con el objetivo de encontrar una forma para definir los servicios de aplicación que deberá proveer un sistema.

El modelo de casos de uso, desarrollado por Jacobson [Jacobson92], describe al sistema, su medio ambiente y la relación entre ambos, es decir, lo describe desde un punto de vista externo. Se especifica la funcionalidad del sistema desde la perspectiva de un usuario del mismo y el desarrollo del sistema se basa en los requerimientos de los usuarios. Este punto es muy importante en la relación de los casos de uso con los servicios de aplicación.

El modelo se compone de casos de uso y de actores.

**Actores**

Para poder determinar cuales son los casos de uso que debe soportar el sistema, se debe identificar a los usuarios del sistema. Esto se hace a través de actores. Un actor define una categoría o tipo de usuario, y cuando el usuario hace algo con el sistema actúa como una ocurrencia de dicha categoría. Los actores definen roles en el uso del sistema, por lo que un mismo usuario podría representar a más de un actor.

Hay dos tipos de actores: primarios y secundarios. Los actores primarios utilizan alguna de las funciones esenciales del sistema, mientras que los actores secundarios realizan tareas de mantenimiento del sistema. Los actores secundarios existen para que los actores primarios puedan usar el sistema.

Un actor no sólo puede modelar un usuario humano, también pude modelar a otro sistema. Los actores representan cualquier entidad que intercambia información con el sistema, están afuera del sistema modelado y son lo único que esta fuera del sistema y tiene impacto en éste.

**Casos de uso**

Una vez que se ha definido lo que esta afuera del sistema (actores) se comienza a definir la funcionalidad interna. Esto se realiza especificando los casos de uso. Un caso de uso es una forma de usar el sistema. Citando a Jacobson [Jacobson92]:

> "*Cada caso de uso constituye un curso de eventos completo, iniciado por un actor, que especifica la interacción que tiene lugar entre el actor y el sistema. Un caso de uso es una secuencia de transacciones relacionadas realizadas por un actor y el sistema en un dialogo.*"

Cuando los actores usan el sistema, este ejecuta un caso de uso. La colección de casos de uso es la funcionalidad completa del sistema.

La identificación de casos de uso suele ser un proceso iterativo, en donde se realizan varios intentos y a medida que se conoce más sobre el sistema se modifican, se agregan y se refactorizan los casos de uso.

Para cada caso de uso se describe el curso de eventos más importante, el que da origen al caso de uso. Este curso de eventos es llamado curso básico. Las variantes del curso básico y los errores que pueden ocurrir en el se describen en cursos alternativos. Normalmente un caso de uso tiene un único curso básico y varios cursos alternativos.

Tanto los casos de usos como los actores son clases que pueden ser instanciadas. Cuando instanciamos un caso de uso se ejecuta una secuencia de acciones en el sistema. Estas acciones provocan algún resultado medíble por un actor en particular [Carroll95]. Aquí el termino clase e instancia se usa de manera diferente que en el paradigma de orientación a objetos, lo que se quiere hacer notar es que el mismo caso de uso (clase) puede ser usado (instancia) de diferentes formas, dependiendo del estado del sistema y del curso de eventos dentro del caso de uso. Una clase de caso de uso puede ser

modelada como una máquina de estados. Una instancia de un caso de uso recorre estos estados durante su vida. Cuál será el próximo estado depende del estado actual y del estímulo que recibirá. Un estimulo recibido de un actor causará que el caso de uso deje su estado actual y realice una o más acciones. La transacción es finalizada cuando el caso de uso ha entrado a un estado nuevamente y espera otro estímulo de un actor.

**Relaciones entre casos de uso**

Los sistemas grandes pueden contener cientos de casos de uso, por lo que para producir un modelo de casos de uso entendible, es necesario relacionar distintos casos de uso evitando redundancia en las descripciones de los mismos. Hay dos tipos de relaciones entre casos de uso: la relación usa y la relación extiende.

Si el número de casos de uso es alto, es común encontrar diferentes casos de uso con descripciones similares. Estas descripciones similares se usan para armar "casos de uso abstractos" (no tienen sentido por si mismos. sólo agrupan en un único lugar información común a otros casos de uso). Por ejemplo, supongamos que los casos de uso A, B y C comparten una parte de su descripción en común, entonces se crea el caso de uso abstracto D y se dice que A, B y C usan al caso de uso D (de esta forma, se puede sacar la parte en común a los casos de uso A, B y C).

La relación usa, sirve para evitar duplicar información común a más de un caso de uso. Por otro lado, la relación extiende se usa para agregar funcionalidad a casos de uso existentes. En lugar de re-escribir un caso de uso para manejar excepciones o caminos alternativos, se crea un nuevo caso de uso que extiende al primero y es en este nuevo caso de uso en donde se introducen los cambios. Se dice que el nuevo caso de uso extiende al original.

La relación extiende nos permite describir los primeros casos de uso (los más básicos) independientemente de cualquier funcionalidad que haya que extender, y cuando llegue el momento de agregar la nueva funcionalidad, no será necesario modificar el caso de uso original evitando la complejidad que esto implica.

**Pros y contras de la utilización de casos de uso**

Los usuarios interactúan con el sistema a través de los casos de uso. Si tomamos todos los casos de uso de un sistema, estos representan todo lo que un usuario podría realizar con el sistema.

Jacobson plantea que si pudiésemos identificar lo más temprano posible todas las formas en que un sistema será usado, y guiar el desarrollo para que el sistema pueda ser usado de esas formas, entonces tendríamos la seguridad de que estamos construyendo el sistema correcto.

Los casos de usos tienen dos roles importantes:

- Capturan los requerimientos funcionales de un sistema.
  Un modelo de casos de uso define el comportamiento esperado del sistema mediante un conjunto de casos de uso. Un modelo de casos de uso no reemplaza a un modelo de objetos del sistema. El modelo de casos de uso es una vista externa del sistema, y el modelo de objetos es una vista interna del mismo sistema.

- Estructuran cada modelo de objetos en una vista manejable.
  Los modelos de objetos para grandes sistemas son inevitablemente complejos; para manejar esta complejidad es necesario presentar el modelo de objetos en distintas vistas. Con el modelo de casos de uso se puede generar una vista por cada caso de uso, y en estas vistas sólo aparecen reflejados los objetos que participan el caso de uso. Un objeto dado podría participar en más de un caso de uso.

Por otro lado, debemos ser cuidadosos en el uso de casos de uso en el contexto del paradigma de orientación a objetos. Los casos de uso son funcionales por naturaleza, por lo que es tentador generar una solución (modelo de objetos) para cada caso de uso, para luego hacer una integración del sistema. Si ocurre esta situación, obtendremos un diseño en donde habrá objetos duplicados que se utilizan para manejar aspectos de un caso de uso en particular. Estos objetos, que en realidad representan lo mismo, tienen implementaciones diferentes según el caso de uso en donde se utilicen, y cuando se integre el sistema, se requerirá un rediseño de estos objetos para obtener un modelo apropiado.

Berard advierte sobre estos problemas y recomienda evitar el uso de está descomposición funcional del sistema como base para la creación de una arquitectura basada en objetos para dicho sistema [Berard].

# Casos de uso esenciales

Una extensión muy interesante al modelo de casos de uso es la que proponen Constantine y Lockwood [Constantine01] en donde se busca expresar la esencia de los casos de uso de un sistema.

Los casos de uso, según fueron propuestos por Jacobson [Jacobson92], describen las interacciones que ocurren entre los actores y el sistema en **términos concretos** (de ahora en adelante los llamaremos casos de uso concretos). Por ejemplo, a continuación se escribe la primer parte del caso de uso "Retornando un Item" para una maquina de reciclado [Jacobson92]:

*"El curso de eventos comienza cuando el cliente presiona el botón de inicio en el panel..."*

Los casos de uso concretos asumen alguna interface con el usuario en particular y la forma de interacción que maneja esa interface (en el ejemplo mostrado se asume que hay un botón que el cliente debe presionar para iniciar la tarea). El problema de esto es que **la descripción del caso de uso restringirá el diseño de la solución**.

Lo que propone Constantine es que, en lugar de modelar las interacciones entre los usuarios y una interface de usuario, la descripción del caso de uso se centre en los **objetivos de los usuarios**. En lugar de elaborar casos de uso detallados y cursos de interacción alternativos, se deben desarrollar descripciones simplificadas que capturen la esencia del caso de uso. La idea es generar descripciones de la esencia de un problema, estas descripciones deben ser abstractas, generalizadas y no dependientes de alguna tecnología en particular.

La definición de casos de uso esenciales, hecha por Constantine y Lockwood, se basa en el trabajo de McMennamin y Palmer [McMennamin84] acerca de la esencia de un sistema. Uno de los conceptos más interesantes definidos en este último trabajo es el de requerimiento verdadero.

Un requerimiento verdadero es una característica o capacidad que un sistema debería poseer para llevar a cabo su propósito, más allá de cómo esté implementado el sistema [McMenamin84]. El conjunto completo de requerimientos verdaderos de un sistema es llamado la esencia del sistema, o los requerimientos esenciales del sistema. Un requerimiento es falso si el sistema podría cumplir con su objetivo sin implementar el requerimiento.

Los falsos requerimientos aparecen principalmente por dos causas: dependencias tecnológicas y requerimientos arbitrarios. Las dependencias tecnológicas surgen cuando se incluyen características ligadas a alguna tecnología en particular en la especificación de los requerimientos, las cuestiones tecnológicas deberían plantearse cuando se solucione el problema, no en su descripción esencial (la descripción del caso de uso analizada anteriormente es un ejemplo de dependencia tecnológica). Los requerimientos arbitrarios aparecen porque en la especificación del requerimiento se incluye más de lo necesitado, o por la mala interpretación del lenguaje usado para generar la especificación de requerimientos (generalmente se usa el lenguaje común).

Un evento es algún cambio en el medio ambiente del sistema, y una respuesta es el conjunto de acciones realizadas por el sistema cada vez que ocurra algún evento. Usando las definiciones anteriores decimos que un sistema interactivo es un mecanismo de evento/respuesta.

En el contexto de servicios de aplicación, el cliente produce un evento al pedir la ejecución de un servicio y el sistema produce la respuesta ejecutando dicho servicio.

Para planear la esencia de un sistema, los desarrolladores de este necesitan saber cómo particionar el sistema en piezas que incorporen los requerimientos verdaderos del sistema. Para particionar los eventos correctamente, es necesario reconocer que es un evento. Esto implica que consideremos al sistema y a su medio ambiente, ya que los eventos ocurren en el medio ambiente del sistema y tanto los eventos externos como los temporales están fuera del control del sistema.

Una construcción aún mas detallada que los casos de uso concretos son los escenarios. Para describir un escenario se usa un estilo de escritura similar a contar una historia. Los escenarios tienden a ser muy concretos y detallados.

Por otro lado, los casos de uso esenciales suelen ser mas cortos y son descriptos con una narrativa más simple, en donde los detalles y las descripciones literales se reducen.

Si comparamos los escenarios, casos de uso concretos y casos de uso esenciales, los últimos forman el modelo más robusto, especialmente para manejar cambios tecnológicos. Esto se debe a que los casos de uso esenciales modelan una tarea en relación a la naturaleza esencial del problema, y no mezclan descripciones del problema con el diseño de soluciones para el problema.

# Identificación de los servicios de la aplicación

Como el lector debe imaginar, el modelo de casos de uso fue presentado para resolver el problema de la identificación y definición de los servicios de la aplicación.

El primer punto a resolver es la identificación de los servicios que debería soportar el sistema. Hay que tener en cuenta que el conjunto de servicios encontrados brindará la funcionalidad del sistema, es decir, un cliente solo podría interactuar con el sistema invocando uno o más servicios.

Los casos de uso nos dan una visión externa de la funcionalidad del sistema, y si usamos casos de uso esenciales no es necesario definir aspectos tales como la forma de la interface con el usuario o la manera en que se debe usar dicha interface.

Por lo tanto, **utilizaremos los casos de uso esenciales para especificar los servicios de la aplicación**. Asociando cada caso de uso esencial a un tipo de servicio de aplicación, obtenemos un conjunto de servicios que brindan la funcionalidad completa del sistema.

Usualmente, un modelo de casos de uso se utiliza para entender qué tiene que hacer el sistema y para ayudar a derivar un modelo de objetos que de soporte a los casos de uso encontrados. Ahora iremos un paso más adelante y modelaremos los servicios a partir de los casos de uso. Esto implica que los casos de uso formarán parte de nuestro modelo de objetos a través de los servicios de aplicación.

# Extensiones al modelo básico

Ahora que se han definido qué son y cómo identificamos los servicios de aplicación (basándonos en el modelo de casos de uso), presentaremos extensiones que si bien, no son esenciales, ayudan a mejorar el modelo básico.

### Relaciones entre servicios de aplicación

Existen dos clases de relaciones entre los servicios de aplicación. Por un lado tenemos relaciones en la definición de un servicio y por otro lado tenemos relaciones para la ejecución de los servicios.

Las relaciones de definición se desprenden de las asociaciones entre casos de uso (extiende y usa) que fueron presentadas anteriormente.

Con respecto a las relaciones de ejecución, encontramos por un lado restricciones de concurrencia y por otro lado tenemos relaciones de prerrequisitos.

Ambas relaciones sirven para controlar la activación de servicios.

Un servicio puede tener como prerrequisito uno o más servicios; esto implica que dicho servicio podrá empezar a ejecutarse sólo si se ejecutaron con éxito sus prerrequisitos. Para poder hacer esto, se debe mantener la historia de los servicios ejecutados y debemos poder especificar los prerrequisitos de un servicio. Con respecto a la ejecución con éxito de un prerrequisito, debemos poder configurar si el prerrequisito debe ser ejecutado por el mismo cliente o si alcanza con que haya sido ejecutado por cualquier cliente.

La otra relación involucrada en la ejecución de servicios es la restricción de concurrencia. Esto significa que un servicio dado podrá empezar a ejecutarse si no se están ejecutando otros servicios restrictivos.

Esta última relación es ideal para modelar servicios con mucha carga de procesamiento. Por ejemplo, podemos definir un gran servicio que ejecuta un procesamiento pesado de manera que tenga restricciones de concurrencia con la mayoría de los servicios existentes; de manera que una vez que comienza la ejecución de este gran servicio podemos asegurar que no se iniciaran el resto, dejándole la mayoría de los recursos.

### Definición de una política de seguridad

Podemos utilizar los servicios de aplicación para definir la seguridad de la aplicación basándonos en usuario/servicio. Esto implica que tenemos que poder definir cuales usuarios (o cuales grupos de usuarios) pueden ejecutar los diferentes servicios de la aplicación.

**Análisis y balance de carga del sistema**

Los servicios de aplicación pueden utilizarse como una medida de la carga de ejecución del sistema. Podemos medir cuantos servicios se están ejecutando en determinado momento, de qué tipo son los diferentes servicios y promediar los tiempos de ejecución.

Por otro lado, si logramos transparencia de la ubicación en cuanto al servidor o a los servidores disponibles, podemos realizar un balance de carga del sistema utilizando a los servicios como unidad de ejecución. Por ejemplo, podemos tener un cliente que inicia la ejecución de un servicio en su servidor asociado, pero la carga del servidor (cantidad de servicios que se están ejecutando) supera el límite definido, entonces el servidor podría delegar la ejecución del servicio a otro servidor asociado sin que el cliente tenga necesidad de conocer esta situación.

# Resumen del capítulo

En este capítulo se trataron los puntos más importantes para la construcción del middle-tier. Primero se demostró que no se puede resolver el problema utilizando solamente objetos distribuidos con la tecnología existente en la actualidad, y se planteo la necesidad de encontrar una solución.

Para resolver los problemas planteados se presentó el concepto de servicio de aplicación y su estrecha relación con los casos de uso esenciales. Como se comentó en el capítulo, el uso de servicios de aplicación nos sirve para evitar los problemas asociados con los objetos distribuidos y para definir concretamente de qué forma pueden interactuar los clientes de la aplicación con los objetos que componen el middle-tier y que se encuentran en el servidor.

En el capítulo siguiente se plantea la necesidad de desarrollar un framework que soporte el concepto de servicios de aplicación y su utilización en el contexto de diferentes sistemas.

# Capítulo 5

# Diseño del Framework: Marco

En este capítulo se describen los principios que guiaron el resultado concreto de este trabajo, y las decisiones más importantes que afectaron al modelo implementado. Dichas decisiones abarcan la comunicación entre los clientes y el servidor, la alternativa elegida para manejar la interacción con el usuario y el ambiente elegido para el desarrollo.

## Principios de diseño

A continuación se enumeran los principios que guiaron el proceso de análisis y diseño del framework resultante de este trabajo. Dichos puntos actuaron como marco durante todo este proceso, y siempre se trato de respetarlos.

- **Simplicidad**.
  Cuando hablamos de simplicidad en el desarrollo de software no debemos confundirnos y pensar que una solución simple es una solución fácil (se podría decir todo lo contrario). Diseñar y desarrollar sistemas informáticos es una tarea desafiante, por lo que obtener soluciones simples y elegantes suele ser muy difícil, y rara vez se llega a este tipo de solución la primera vez que uno se enfrenta con alguna clase de problemas (generalmente se necesitan varias iteraciones en el desarrollo de la solución para poder obtener un diseño simple).
  Seguramente, en el diseño del modelo de objetos que se realizó en este trabajo quedaron partes excesivamente complejas, cuestiones que se deben poder resolver de una forma más simple y elegante, sin embargo, la simplicidad en el diseño fue un punto que se tuvo en cuenta.

- **Indepencia de implementación entre los clientes y el servidor**.

La idea es que tanto los clientes como el servidor no tengan que estar implementados necesariamente en el mismo lenguaje. Esto permite que el servidor evolucione independientemente de los clientes, no se crean dependencias en cuanto al desarrollo de las distintas capas.
Además, podemos tener distintos tipos de clientes implementados de diferentes maneras.

- **Separación total del dominio de la aplicación respecto de la interacción con el usuario**.
  Esto es fundamental para poder tener clientes livianos, sin embargo, esto no es exclusivo de las arquitecturas de tres capas. Se considera una buena decisión de diseño separar la construcción de la interface con el usuario del modelo de la aplicación. Entre las soluciones más conocidas encontramos la arquitectura MVC (Model-View-Controller), o sus parientes MV (Model-View) y MVP (Model-View-Presenter).

- **Capacidad de reutilizar el sistema en diferentes contextos**.
  Sería más simple desarrollar y enfocarse en una sola aplicación de tres capas en particular e implementar un prototipo que sólo responda a dicha aplicación. Sin embargo, es mucho más interesante diseñar e implementar un modelo reusable en el contexto de diferentes aplicaciones de tres capas. La idea es diseñar un modelo de objetos que se pueda adaptar a distintas aplicaciones, que provea mecanismos para aliviar el desarrollo de la capa en donde corren los servicios de aplicación.
  Esto nos lleva al desarrollo de frameworks.
  Un **framework** es el diseño de un grupo de clases que colaboran para llevar a cabo un conjunto de responsabilidades, sólo que el diseño es abstracto (los componentes principales se modelan con clases abstractas) [Lajoie94].
  Los frameworks no son bibliotecas de clases, tienen algo más. El framework juega el rol del programa principal, coordina y secuencia la actividad de la aplicación. Esta inversión del control les da a los frameworks la potencia para ser estructuras extensibles [Johnson91].
  Básicamente existen dos formas de extender la funcionalidad de un framework. Por un lado se pueden crear nuevas subclases de las clases existentes y re-implementar métodos definidos por las clases principales del framework. La otra alternativa es instanciar componentes predefinidos y "engancharlos" en los mecanismos del framework. La composición de estos componentes dictará el comportamiento del framework.

Teniendo en cuenta estos principios, se diseño un modelo para el middle-tier que soporta la especificación de servicios de aplicación para utilizar la aplicación, es decir, interactuar con el modelo de objetos. Además, se cumple otro requisito indispensable en una arquitectura de tres capas, nuestro modelo soporta la ejecución concurrente por parte de múltiples clientes controlando los posibles conflictos en el acceso a recursos compartidos.

A continuación se repasan los principales puntos encontrados, las decisiones de diseño que fueron tomadas y cómo estas decisiones afectan al modelo resultante. Estos puntos abarcan el manejo de la interacción con el usuario, la elección de un protocolo para la comunicación entre los clientes y el servidor y, por último, el ambiente elegido para el desarrollo.

# Manejo de la interacción con el usuario

Como vimos en el Capítulo II, básicamente tenemos dos alternativas para manejar la interacción con el usuario: **presentación distribuida** e **interface con el usuario remota**. Como hemos detallado previamente, si las comparamos encontramos ventajas y desventajas en ambas. Por ejemplo, con una presentación distribuida se hace más fácil la actualización del software en los clientes, por otro lado, con el manejo remoto de la interface en el cliente se puede administrar de manera más inteligente el intercambio de mensajes entre el cliente y el servidor.

En el desarrollo del sistema optamos por el segundo caso, una interface con el usuario remota. Las razones principales son, primero, que podemos experimentar con mensajes entre el cliente y el servidor con semántica de la aplicación y, segundo, que en el caso de una presentación distribuida se tendría que haber desarrollado un toolkit gráfico, este es un punto complicado y no se encuentra entre los objetivos de este trabajo.

# Comunicación cliente-servidor

Una vez que fue elegido qué modelo de interacción con el usuario se iba a implementar, había que definir de qué manera se comunicarían los clientes con el servidor de la aplicación.

Para esto se estudiaron tres alternativas: algún mecanismo de distribución propietario (tipo RMI o RMP), el uso de mensajes XML y el estándar CORBA.

A continuación se describen las tres alternativas estudiadas (con sus ventajas y desventajas) y luego se detalla cuál fue la elegida y cuales fueron las razones.

**Mecanismo propietario de pasaje de mensajes remotos**

Los mecanismos que se estudiaron son RMI (Remote Method invocation - Invocación de Métodos Remotos) de Java [Flanagan99] y RMP (Remote Message Pasing - Pasaje de Mensajes Remotos) implementado en Squeak.

Ambos mecanismos permiten enviar mensajes a objetos remotos de manera similar que el envío de mensajes a objetos locales. El método utilizado consiste en tener un objeto local que representa al objeto remoto, y cuando este recibe algún mensaje se lo transmite al objeto remoto utilizando el sistema de distribución. Como mencionamos anteriormente, el objeto local que representa al objeto remoto es conocido como "proxy" [Gamma95, Buschmann96].

La principal contra de este tipo de mecanismos propietarios para implementar objetos distribuidos se da en el hecho de que tanto el objeto que envía el mensaje como el que lo recibe, deben estar ejecutándose en el mismo tipo de ambiente. Por ejemplo, es imposible usar tanto RMI como RMP entre Squeak y Java.

Esto implica que, si se hubiera elegido una de estas alternativas, tanto los clientes como el servidor deberían programarse en el mismo lenguaje, y entonces no podríamos cumplir con el objetivo de tener la mayor independencia posible entre los clientes y el servidor.

**CORBA**

CORBA es un estándar público desarrollado por el OMG (Object Management Group - Grupo de Administración de Objetos). El estándar especifica de manera uniforme, cómo deben enviar y recibir mensajes los objetos distribuidos, la estructura de los mensajes intercambiados, cómo llega un mensaje al objeto destinatario y cómo se retornan los resultados al objeto que generó el llamado. La sigla CORBA significa Common Object Request Broker Architecture y el ORB es el núcleo del estándar.

Un ORB permite que objetos que se encuentran distribuidos intercambien mensajes de una manera transparente. El ORB oculta la ubicación de los objetos (el cliente no necesita saber en donde reside el objeto receptor del mensaje), la implementación (el cliente no debe conocer cómo está implementado el receptor, esto abarca al lenguaje utilizado, el ambiente de ejecución y el hardware), el estado de ejecución del receptor y también oculta los mecanismos de comunicación usados para hacer que el mensaje llegue desde el objeto cliente al objeto destino [Vinoski96].

A diferencia de los mecanismos propietarios nombrados anteriormente, CORBA provee interoperabilidad entre distintos lenguajes, es decir, no se requiere que los objetos distribuidos estén implementados usando todos el mismo lenguaje (podríamos tener objetos Java, C++ y Smalltalk interactuando en el mismo sistema distribuido). Para que esto sea posible, se deben declarar las interfaces que soportan los distintos objetos (una interface es

un conjunto de métodos que un objeto implementa, para cada método en la interface se deben declarar el tipo retornado y los tipos de los parámetros que el método recibe). Las interfaces se escriben utilizando un lenguaje puramente declarativo que se conoce como IDL (Interface Definition Languaje - Lenguaje de Definición de Interfaces).

CORBA fue diseñado para que puedan agregarse de forma transparente nuevas características que se conocen como servicios CORBA [Montlick99].

Algunos de los servicios que están especificados en el estándar son:

- Servicio de nombres.
  Se asocian nombres con objetos para que estos puedan ser encontrados más fácilmente. Los nombres pertenecen a un contexto de nombres y se garantiza que en un contexto no hay nombres repetidos.

- Servicio de eventos.
  Mediante está característica, los objetos distribuidos se pueden comunicar automáticamente cuando sucede algún evento de interés. Se soportan los modelos "push" y "pull", esto es, los objetos consumidores de los eventos pueden chequear cada cierto tiempo si ocurrió algún evento o pueden ser notificados automáticamente.

- Servicio de ciclo de vida.
  Este servicio define de qué forma los objetos son creados, eliminados y movidos o copiados en el sistema.

- Servicio de control de concurrencia.
  Se permite que múltiples clientes coordinen el acceso a recursos compartidos. El uso concurrente de un recurso se regula mediante locks.

- Servicio de consultas.
  Este servicio posibilita la selección de objetos que satisfagan algún criterio arbitrario.

Está lista muestra sólo una parte de los servicios definidos en CORBA, sin embargo, es conveniente aclarar que la mayoría de las implementaciones comerciales del estándar no soportan todos los servicios que éste propone.

**Mensajes XML**

XML (Extensible Markup Languaje - Lenguaje de Marcado Extensible) es un lenguaje diseñado para hacer que la información se autodescríba. La idea es usar marcas (conocidas como tags) que dan estructura a la información. De manera inversa a la mayoría

de los formatos utilizados en informática, un documento XML también tiene sentido para un humano debido a que solo se usa texto común [Bosak99].

XML es un formato basado puramente en texto. Un documento XML se compone de elementos XML, cada uno de los cuales consiste en una marca de comienzo (por ejemplo, *<persona>*), una marca de fin *(</persona>*) y la información entre las dos marcas (el contenido). Una diferencia fundamental con respecto a otros formatos que almacenan la información (texto) entre marcas, como HTML, es que en XML no hay un límite a las posibles marcas que se pueden usar en un documento.

A continuación se describe información de una persona usando XML:

```
<persona>
       <apellido> Perez </apellido>
       <nombre> Juan </nombre>
       <fechaNacimiento>
              <dia> 5 </dia>
              <mes> 10 </mes>
              <año> 1987 </año>
       </fechaNacimiento>
       <domicilio>
              <ciudad> La Plata </ciudad>
              <calle> 7 </calle>
              <numero> 1400 </numero>
       </domicilio>
</persona>
```

En lugar de describir el orden y la forma en que la información podría ser mostrada, las marcas indican el significado de los datos. Cualquier receptor de este documento puede decodificarlo.

La tecnología alrededor de XML está basada en estándares abiertos, esto implica que toda la comunidad informática trabaja para asegurar la interoperabilidad de los documentos XML, evitando la dependencia de un producto de un solo vendedor.

Los estándares relacionados son los siguientes:

- XML. Es la definición del lenguaje XML.

- XML Namespaces. Se describe la sintaxis para espacios de nombres (un espacio de nombres es una colección de nombres únicos en algún contexto) y el soporte que deberían proveer los parsers aptos para namespaces.

- Document Object Model (DOM). Se trata de proveer un estándar para acceso a información estructurada mediante herramientas de scripting.

- Extensible Stylesheet Languaje (XSL). Es un lenguaje para transformar documentos XML en otra representación orientada a cómo se mostrará la información (por ejemplo, HTML).

- XML Linking Languaje (XLL) y XML Pointer Languaje (XPointer). Se trata de hacer hypertextos usando XML, los links pueden ser multidireccionales y pueden estar basados a un nivel de los objetos que forman un documento en lugar de a nivel de página.

**Alternativa elegida**

La alternativa elegida para la comunicación entre los clientes y el servidor fue utilizar mensajes XML.

La primera opción que se descartó es usar un mecanismo de pasaje de mensajes propietario, ya que esto implicaba que los clientes y el servidor debían estar implementados necesariamente usando el mismo lenguaje de programación.

La alternativa CORBA es la más completa de las tres, pero también la más compleja. A favor de CORBA podemos decir que es un estándar que va más allá de una solución propietaria (el OMG está compuesto por más de 800 empresas de primer nivel) y por lo tanto soporta interoperabilidad entre distintos lenguajes. La principal contra de CORBA es el grado de complejidad que hubiese tenido la solución, tendríamos que escribir interfaces en IDL y todos los clientes deberían tener un ORB activo para comunicarse con el servidor.

Las razones principales que llevaron a elegir mensajes XML como mecanismo de comunicación fueron su simplicidad con respecto a CORBA (además, los clientes quedan más livianos ya que no necesitan tener un ORB activo), y por otro lado, XML también es un estándar.

Además, se puede tener total independencia en el desarrollo de los clientes con respecto al middle tier.

La principal contra de esta alternativa es el tamaño de los mensajes XML con respecto a un formato binario y el costo de interpretar los mensajes (parsear los mensajes) tanto en los clientes como en el servidor. En un formato binario se puede reducir el tamaño de los mensajes y, por lo tanto, disminuir el tiempo en las transmisiones de los mensajes. A favor del XML podemos decir que la tasa de compresión de un documento XML suele ser buena (debido a que es un formato textual y muchas cadenas de texto se repiten en el documento), por lo que si el mensaje supera cierto límite, podría ser compactado antes de la transmisión y descompactado por el receptor, aunque esto implicaría mas tiempo y recursos.

# Ambiente elegido para el desarrollo

El ambiente elegido para el desarrollo de este trabajo fue Squeak [Squeak, Ingalls97]. **Squeak** es una implementación "**open source**" del lenguaje Smalltalk [Goldberg83]. Por ser un **Smalltalk**, Squeak es un lenguaje orientado a objetos puro. Esto implica que sólo tenemos objetos (no hay tipos primitivos) y que la única forma de interactuar con los objetos es mediante el envío de mensajes.

Uno de los axiomas que tuvieron en cuenta los creadores de Smalltalk fue elegir un acotado número de principios y aplicarlos uniformemente en el lenguaje.

Resumiendo, las características principales de Smalltalk son:

- Es un lenguaje orientado a objetos puro. Esto implica que todo es un objeto (no tiene tipos primitivos y las clases también son objetos), y que la única forma de interactuar con los objetos es enviándole mensajes.

- Además de ser un lenguaje, es un ambiente de objetos. Esta es una diferencia muy importante con otros lenguajes orientados a objetos. En la mayoría de los lenguajes, el programador escribe el código que luego será compilado y recién en runtime se crearán los objetos que componen el sistema desarrollado. En Smalltalk la situación es diferente. Al estar basado en una imagen virtual, el programador tiene a su disposición objetos con los que puede interactuar mientras desarrolla el sistema.

- Tiene clases y provee herencia simple.

- Polimorfismo y binding dinámico. Esta es una propiedad fundamental del paradigma de orientación a objetos.

- No tiene manejo de tipos estático. Esto significa que el programador no tiene que especificar el tipo de retorno en un método ni los tipos de los argumentos que recibe un método.

- Es un lenguaje altamente reflexivo.

- Tiene manejo de memoria automática (garbage collection).

El proyecto Squeak es un desarrollo open source (todo el código esta disponible para ser estudiado y modificado) muy activo y dinámico. En la actualidad, Squeak corre en la mayoría de los sistemas operativos permitiendo que nuestro código pueda portarse sin modificar una sola línea de código. Además se han desarrollado aplicaciones en áreas muy diversas. Un ambiente Squeak estándar tiene, entre otras características, dos frameworks para desarrollar interfaces gráficas (MVC y Morphic), un motor de gráficos 3D, un servidor de páginas web, facilidades para el procesamiento de imágenes, síntesis de sonidos y soporte a numerosos protocolos de Internet (HTTP, FTP, SMTP, etc).

La sintaxis de Squeak es como la de cualquier otro Smalltalk. En el apéndice B se describe esta sintaxis. Esto nos servirá para leer fragmentos de código en los próximos capítulos.

## Resumen del capítulo

Este capítulo sirve como marco para comprender las ideas generales del framework desarrollado en este trabajo. Se presentaron los principios de diseño que guiaron el desarrollo del framework, la alternativa elegida para manejar la interacción con el usuario y el protocolo que se utiliza para la comunicación entre los clientes y el servidor de la aplicación.

En el capítulo siguiente se describen en detalle las clases principales del framework desarrollado.

# Capítulo 6

# Diseño del Framework: Núcleo

En este capítulo se describe el resultado concreto de este trabajo: un framework para el desarrollo de sistemas con arquitecturas de tres capas. Aquí se muestra como se le dio forma a las ideas propuestas en el capítulo 4 utilizando como marco los principios enumerados en el capítulo 5. A diferencia de los capítulos anteriores, en este capítulo se detallan aspectos del código (software) escrito en este trabajo.

## Estructura general

El framework desarrollado se puede dividir en cinco subsistemas:

- **Objetos de aplicación y atributos**.
  Este subsistema permite acceder a las propiedades de los objetos que forman el modelo de la aplicación a desarrollar desde los clientes de la aplicación. Las clases principales son ApplicationObject y ObjectAttribute.

- **Soporte a mensajes remotos entre los clientes y el servidor**.
  Para esto se modelaron los diferentes tipos de mensajes que los clientes le pueden enviar al servidor y las respuestas del servidor a los clientes. Como dijimos en el capítulo anterior, el protocolo utilizado se basa en XML.

- **Manejo de múltiples contextos de ejecución**.
  En este subsistema se modelaron clases que representan contextos de ejecución para los clientes que dialogan con el sistema. En este subsistema se programaron puntos críticos como el manejo de sockets para las comunicaciones TCP/IP entre los clientes y el servidor, y también hay una utilización importante del modelo de

procesos (o threads) de Squeak para dar soporte concurrente a los distintos contextos.

- **Implementación de los servicios de aplicación**.
  Aquí se modelaron los conceptos definidos en el capítulo 4, en particular, la idea de servicio de aplicación se reflejó en la clase ApplicationService.

- **Soporte de persistencia a los objetos**.
  En este subsistema se desarrolló un modelo abstracto para manejar la persistencia de los objetos.

En este capítulo analizaremos los primeros cuatro subsistemas, y en el siguiente nos dedicaremos por completo a la persistencia de los objetos. A continuación veremos con detalle qué es un ApplicationObject, el manejo de atributos, la jerarquía de mensajes remotos y por último analizaremos el subsistema en donde se da soporte a múltiples contextos de ejecución.

# Objetos de aplicación

Una de las clases principales que forma parte del corazón del framework es la clase ApplicationObject. Un ApplicationObject tiene dos características esenciales: un identificador único (llamado objectId) y atributos que le dan estructura.

El objectId es un identificador único que representa al applicationObject en el sistema. Cuando se crea una nueva instancia de un ApplicationObject, automáticamente se le asocia un objectId.

No pueden existir dos applicationObjects diferentes con el mismo objectId. Por lo tanto, debemos implementar un mecanismo que garantice la unicidad de los objectIds. Para esto tenemos dos enfoques, por un lado se podría tener un contador global que es incrementado cada vez que se usa como objectId en algún ApplicationObject. La contra de esta alternativa es que la generación de objectIds se transforma en un cuello de botella en el caso de que diferentes threads (o procesos) necesiten crear instancias de ApplicationObject o de alguna de sus subclases. Además, el contador debe almacenarse en la persistencia, ya que no queremos perderlo y repetir objectIds que ya fueron usados. El segundo enfoque es utilizar un mecanismo más liviano que baje las probabilidades de crear dos objectIds iguales.

La alternativa elegida fue la segunda y el método usado para crear el objectId se descompone en tres pasos:

1) La primer parte del objectId es la identificación de la clase. Dicha identificación es una cadena de 6 caracteres que se define en el método de clase `#classId`. Esto implica que cuando agregamos una nueva subclase de ApplicationObject debemos redefinir el método `#classId`.
2) La segunda parte del objectId se basa en el momento de creación del objeto (el año, el día en el año y los segundos en el día).
3) Por último, se generan dos números al azar (de cuatro cifras) y se concatenan a la cadena de caracteres generada.

Con este método, la probabilidad de generar dos objectIds iguales es muy baja, ya que el classId impide que objetos de diferentes clases puedan tener el mismo objectId, la marca de tiempo reduce el conflicto a un segundo en una fecha en particular y los números random reducen la probabilidad dentro de dicho segundo.

Los métodos como classId se conocen como "hook methods" [Gamma95]. Los hook methods se definen en clases abstractas, pero no son implementados. La idea es que un framework que utilice hook methods, asumirá que están implementados en las subclases de las clases abstractas, y es responsabilidad de los programadores que utilizan el framework re-implementar los hook methods que el framework necesita para funcionar correctamente.

Los objectIds tienen dos usos principales. Por un lado, como vimos en el capítulo II, se pueden usar para manejar la persistencia de los objetos: el objectId es el identificador único que tienen los objetos persistentes cuando se aplanan y se almacenan en una base de datos.

El otro uso de los objectIds consiste en utilizarlos como referencias a objetos remotos en sistemas distribuidos.

En este trabajo se utilizaron para tener referencias remotas desde los clientes hacia el núcleo de la aplicación. Por ejemplo, el cliente conoce el objectId de un objeto al que le quiere modificar cierto atributo y envía un mensaje para realizar el cambio, el servidor de la aplicación obtiene el ApplicationObject a partir del objectId y ejecuta la tarea.

La segunda característica principal de ApplicationObject es la capacidad de definir atributos (instancias de la clase ObjectAttribute). Los atributos se definen a nivel de clase, ya que las instancias de una misma clase deben tener los mismos atributos. La definición de atributos se hace en el método de clase `#createAttributes`, y las subclases de ApplicationObject deben redefinir dicho método para agregar los atributos deseados (`#createAttributes` es otro ejemplo de un "hook method").

Aunque los atributos se crean directamente, el manejo interno de los atributos se deriva a una instancia de ApplicationObjectDescriptor. Los descriptors son objetos auxiliares que manejan los atributos de las instancias de ApplicationObject y sus subclases. Además de obtener o modificar los valores de los atributos, los ApplicationObjectDescriptor manejan los errores que pueden producirse al operar con atributos (por ejemplo, tratar de obtener el valor de un atributo que no existe).

Uno de los principales usos de los atributos en los ApplicationObject es la posibilidad de generar strings XML que representen la estructura y el estado de un ApplicationObject. Este string XML se puede enviar a un sistema remoto para crear un "espejo" del ApplicationObject y acceder a su estado.

El protocolo principal de ApplicationObject es el siguiente:
Métodos de clase:

- `#classId`.
  Retorna un string de 6 caracteres que identifica a la clase. Las subclases de ApplicationObject deben redefinirlo.
- `#createAttributes`.
  Crea los atributos de las instancias de la clase. Las subclases de ApplicationObject deben redefinirlo invocando a la implementación en la superclase.
- `#atObjectId:`.
  Dado un objectId, retorna la instancia asociada.

Métodos de instancia:

- `#objectId`.
  Devuelve el objectId del ApplicationObject.
- `#getValueForAttributedNamed:`.
  Dado el nombre de un atributo, devuelve el valor de dicho atributo en el objeto receptor del mensaje.
- `#setValue:forAttributeNamed:`.
  Asigna el valor de un atributo.

En la siguiente figura podemos observar un diagrama de clases centrado en ApplicationObject y cómo esta clase se relaciona con los atributos (tema que detallaremos a continuación):

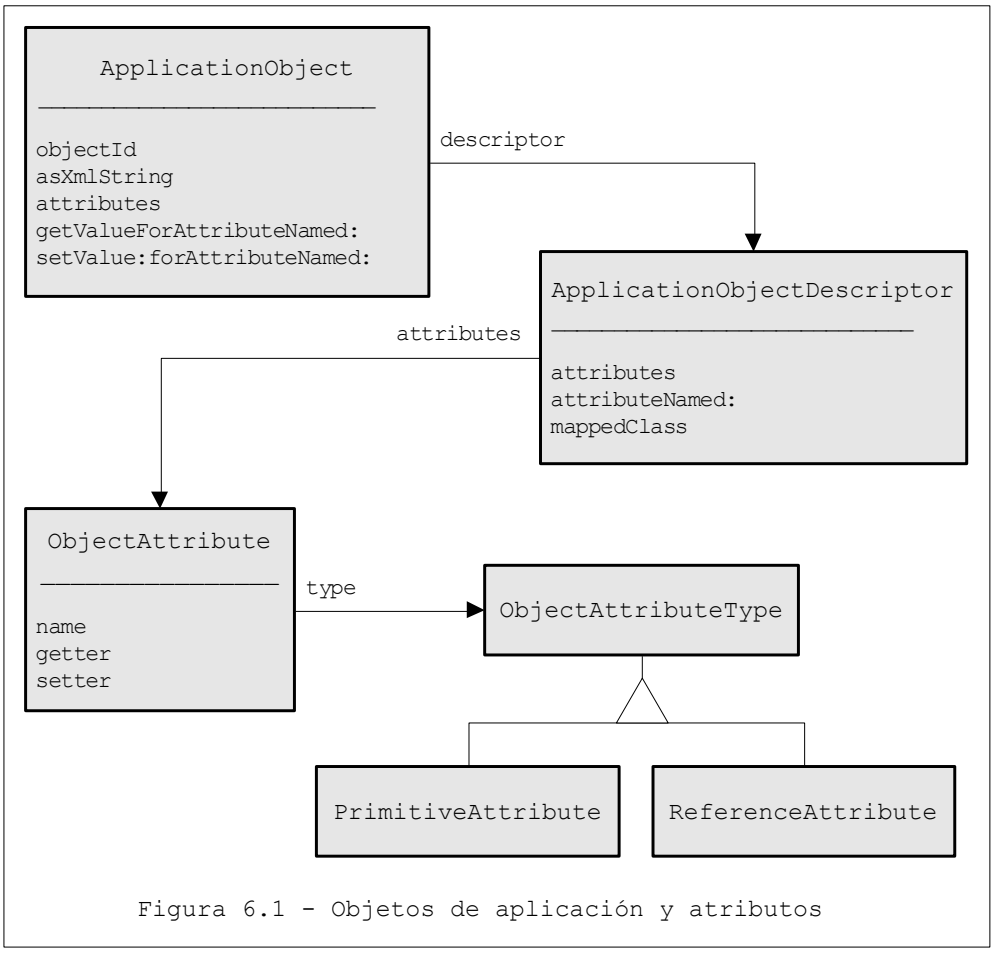



Figura 6.1 - Objetos de aplicación y atributos

# Atributos

En el paradigma de Orientación a Objetos se dice que los objetos agrupan comportamiento y estado. El estado de un objeto esta reflejado en sus variables de instancia. Dada una clase, se pueden definir métodos tanto para acceder como para modificar el valor (otro objeto) de una variable de instancia.

Los ObjectAttribute representan estos conceptos con un nivel de indirección extra. Los atributos generan una estructura pública para las instancias de las clases que los definan. Por ejemplo, podemos tener una clase llamada Empleado con atributos como nombre, fecha de ingreso, jefe, etc.

El uso de atributos permite lo siguiente:

- Se pueden mapear a atributos algunas de las variables de instancia (no necesariamente a todas).
- Se pueden mapear a atributos, propiedades que no estén ligadas a ninguna variable de instancia y cuyo resultado requiera alguna computación extra.
- Se pueden nombrar atributos, y acceder o modificar su valor a través del nombre.

Para soportar esta funcionalidad, un ObjectAttribute define un nombre, un "getter" (es el nombre del método usado para obtener el valor del atributo), un "setter" (es el nombre del método usado para asignar el valor del atributo; hay que aclarar que no es necesario definir un setter, ya que podemos tener atributos de sólo lectura).

Este es un claro ejemplo de lo que en programación se conoce como reflexión. No es muy común tener el nombre de un método y ejecutarlo dinámicamente en algún objeto que no conocemos a priori, y por lo tanto, no son muchos los lenguajes o ambientes de programación que soportan este tipo de funcionalidad (por ejemplo, en la primeras versiones del lenguaje Java no se podría haber ejecutado código de este tipo).

Esta es una funcionalidad muy potente, ya que nos permite interactuar remotamente con los objetos que se encuentran en el núcleo de la aplicación de una forma bastante simple. Por ejemplo, para obtener cierta información de un objeto del dominio de la aplicación solo necesitamos conocer el objectId del objeto y el nombre del atributo buscado. Solamente con estos dos datos el framework se encargará de encontrar el objeto y luego ejecutará reflectivamente el método asociado para obtener el valor de dicho atributo (este método es el getter del atributo). Luego, retornará el valor al cliente remoto.

Además, todo ObjectAttribute tiene un tipo que define los valores que puede tomar el atributo. El tipo es representado por una instancia de alguna subclase de ObjectAttributeType. Esta clase tiene dos subclases, por un lado tenemos PrimitiveAttribute que representa tipos primitivos (números, valores booleanos, strings, fechas) y por otro lado tenemos la clase ReferenceAttribute que representa a cualquier subclase de ApplicationObject.

Como dijimos anteriormente, todas las propiedades de un atributo (nombre, tipo, getter, setter y condiciones de filtro) se definen en los métodos `#createAttributes` de ApplicationObject y sus subclases.

# Aplanamiento de objetos

Una vez que se tomó la decisión de utilizar mensajes XML para comunicar a los clientes con el servidor, surgió la necesidad de encontrar representaciones adecuadas para transmitir los objetos (o parte de estos) que formaban el dominio de la aplicación.

Obviamente, todo indicaba que debíamos usar XML para representar a los objetos y poder embeber estas porciones de XML en los mensajes remotos.

Se implementaron dos funciones básicas, por un lado, un objeto puede generar un documento XML que lo representa y, por otro lado, dado un documento XML que representa a un objeto se puede reconstruir el objeto.

El protocolo consiste de dos mensajes y se implemento en la clase Object (es la raíz de la jerarquía de herencia), esto implica que cualquier objeto tiene esta funcionalidad. Los mensajes son:

- `#asXmlString`.
  Retorna un string (sequencia de caracteres) con formato XML que representa al receptor del mensaje.
- `#fromXmlString:`.
  Recibe como argumento un string con formato XML y retorna el objeto representado por el documento XML.

Tanto en la generación del documento, como en la instanciación de un objeto a partir del documento XML, se hicieron dos grupos en cuanto al tipo de objeto. Por un lado, tenemos los llamados "objetos primitivos". Estos objetos representan cosas básicas y tienen equivalentes en la mayoría de los lenguajes de programación. Los ejemplos de este tipo de objetos son números, valores booleanos, fechas, cadenas de caracteres, etc.

A continuación se muestra el documento XML que genera la ejecución de "`3 asXmlString`" (obtener la representación XML del numero 3) y la ejecución de "`'Pepe' asXmlString`" (obtener la representación XMl del string 'Pepe'):

```
<PrimitiveObject>
       <Type> Number </Type>
       <Value> 3 </Value>
</PrimitiveObject>


<PrimitiveObject>
       <Type> String </Type>
```

```
        <Value> Pepe </Value>
</PrimitiveObject>
```

La situación cambia completamente si tratamos con objetos "no primitivos". Para esto, los mensajes `#asXmlString` y `#fromXmlString:` fueron redefinidos en la clase ApplicationObject. Los documentos XML generados por los ApplicationObjects se pueden dividir en tres partes. Por un lado se tiene el tipo del objeto (esto es similar al caso de los objetos primitivos que como vimos tienen un tag llamado <Type>), también se guarda la identificación del objeto (obviamente, utilizando el objectId del mismo) y por último el estado de los atributos del objeto.

Por ejemplo, supongamos que tenemos un objeto de la clase CajaDeAhorro (que representa una caja de ahorro en un banco) que tiene dos atributos (para simplificar), el número de la cuenta y el saldo. El documento XML generado tendría la siguiente forma:

```
<ApplicationObject>
        <Type> CajaDeAhorro </Type>
        <ObjectId> CAJAHO200262785332171553 </ObjectId>
        <Attributes>
                <Attribute>
                        <Name> numero </Name>
                        <Value>
                                <PrimitiveObject>
                                        <Type> String </Type>
                                        <Value> 123-55787/4 </Value>
                                </PrimitiveObject>
                        </Value>
                </Attribute>
                <Attribute>
                        <Name> saldo </Name>
                        <Value>
                                <PrimitiveObject>
                                        <Type> Number </Type>
                                        <Value> 270.80 </Value>
                                </PrimitiveObject>
                        </Value>
                </Attribute>
        </Attributes>
</ApplicationObject>
```

Una vez más, para poder obtener el valor de los atributos se utilizo reflexión, a partir del valor del atributo se genero el XML correspondiente de manera recursiva.

En este ejemplo, ambos atributos eran primitivos, pero en una situación en donde el valor de un atributo es otro ApplicationObject, el documento XML generado solo queda reflejado el objectId del objeto referenciado.

Por último, cuando se definen los atributos (en el método de clase `#createAttributes`) podemos elegir si el atributo debe ser usado para los mensajes XML (si el atributo se declara público entonces es utilizado en la generación del XML) ya que es posible tener atributos que no participen en los mensajes remotos.

# Mensajes remotos

Para desarrollar el subsistema de mensajería en el servidor, se tenían que resolver dos puntos. Por un lado debíamos poder recibir mensajes desde los distintos clientes, interpretarlos y ejecutar las acciones necesarias, y por otro lado, una vez que se obtenía la respuesta, esta debía ser transmitida al cliente.

Para resolver estos puntos, se diseño una jerarquía de clases que modela los distintos tipos de mensajes remotos. La raíz de la jerarquía es la clase RemoteMessage (subclase de Object), en esta clase se define el manejo de un identificador de mensaje.

Luego, la jerarquía se divide en dos grupos. Por un lado están modelados los mensajes que pueden enviar los clientes al servidor y por otro lado los que envía el servidor a los clientes.

A continuación podemos observar los dos primeros niveles de la jerarquía (todas las clases de la figura son abstractas).

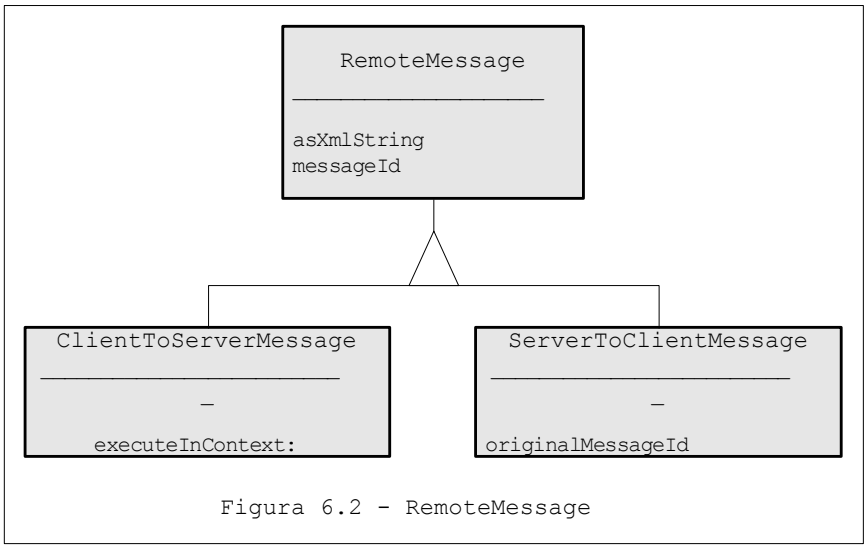



Figura 6.2 - RemoteMessage

Como podemos observar en la figura, en la clase abstracta RemoteMessage se define el método `#asXmlString` (esto implica que los mensajes saben aplanarse a un string xml) y `#messageId`, que devuelve un número que identifica al mensaje.

En la clase ClientToServerMessage se define el método abstracto `#executeInContext:`, este método recibe como argumento un contexto de ejecución (más adelante hablaremos de los contextos de ejecución) y lo que hace es ejecutar el mensaje en el contexto recibido. Esto implica que los mensajes de tipo ClientToServerMessage saben ejecutarse en un contexto de ejecución. La jerarquía de clases completa que modelan los mensajes que puede recibir el servidor de un cliente es la siguiente (la letra A al final del nombre de la clase significa que es una clase abstracta):

```
ClientToServerMessage (A)
       ExecutionContextMessage (A)
              LoginMessage
              LogoffMessage
       ActivateAppServiceMessage
       ObjectMessage (A)
              ObjectAttributeMessage (A)
                     GetValueFromAttributeMessage
                     SetValueToAttributeMessage
              StartAppServiceMessage
```

Enumerando las clases, tenemos:

- ExecutionContextMessage, LoginMessage y LogoffMessage. Estos mensajes están asociados con la creación y la destrucción de los contextos de ejecución. Cuando se ejecuta un mensaje de logon (entrada al sistema), se valida el usuario y la password, y en caso de ser correctas se asocia dicho usuario al contexto de ejecución. Por otro lado, cuando se ejecuta un mensaje de logoff (salida del sistema), se libera el contexto de ejecución asociado junto con todos los recursos involucrados (sockets, conexiones a la base de datos, y threads utilizados).

- ActivateAppServiceMessage. Este es un mensaje un tanto particular, ya que la única información que transmite es el nombre de un servicio que el cliente desea activar (hay que aclarar que para ejecutar un servicio de aplicación, este debe ser activado primero).

- ObjectMessage. En esta clase se modelan mensajes dirigidos a objetos en particular. Esto es diferente de los tipos de mensajes estudiados hasta ahora, ya que en los de login y logoff se estaba diciendo que el cliente quería entrar o salir del sistema respectivamente, y en el de activación de un servicio el cliente pedía la activación basándose en el nombre del servicio. Para poder identificar al objeto sobre el que se ejecuta el mensaje, este contiene el objectId de dicho objeto.

- ObjectAttributeMessage, GetValueFromAttributeMessage y SetValueToAttributeMessage. Estos grupos de mensajes operan sobre algún atributo

del objeto asociado. En la clase ObjectAttributeMessage se define el nombre del atributo asociado, y en las subclases se manejan las dos operaciones básicas asociadas con un atributo, es decir, obtener el valor actual del atributo y asignarle un valor nuevo.

- StartAppServiceMessage. Para explicar la idea definida en esta clase debemos entender cómo se activa y configura un servicio de aplicación. Como veremos más adelante, la clase que modela los servicios de aplicación es subclase de ApplicationObject. Esto implica que un servicio de aplicación tiene un objectId que lo identifica y que puede tener una serie de atributos necesarios para su ejecución.
  Entonces, para ejecutar un servicio se ejecutan los siguientes pasos: primero el servicio es activado por nombre (usando un ActivateAppServiceMessage), a partir de ahí una nueva instancia es creada con un objectId que la identifica (esto es transmitido al cliente), ahora el cliente puede configurar el servicio con los atributos necesarios para la ejecución (usando mensajes del tipo SetValueToAttributeMessage) y por último se arranca la ejecución del servicio con un StartAppServiceMessage.

Ahora nos queda por ver de qué forma se manejan los mensajes que el servidor les envía a los clientes. Este es un punto mucho más simple que el anterior, ya que se encontraron dos tipos de mensajes que fueron suficientes para manejar la interacción entre las dos partes. La jerarquía de clases resultante es la siguiente:

```
ServerToClientMessage (A)
       SimpleAcknowledgeMessage
       ResultMessage
```

La clase SimpleAcknowdledgeMessage modela mensajes de respuesta simples que lo único que informan es que el mensaje enviado por el cliente se recibió y fue ejecutado con éxito. En cambio, las instancias de ResultMessage retornan un resultado al cliente.

Por ejemplo, una vez que se ejecuta el logon con éxito, al cliente se le devuelve un mensaje de reconocimiento simple (SimpleAcknowledgeMessage), mientras que cuando se activa un servicio al cliente se le retorna una referencia (via objectId) al servicio activado.

En los ResultMessage se utiliza la funcionalidad que tienen los objetos en el sistema de generar un documento XML que los represente (este documento es embebido en el mensaje enviado al cliente).

Los ResultMessage también se usan para transmitir situaciones de error al cliente, en estos casos, el resultado es un error (una instancia de la clase ApplicationError).

# Soporte a múltiples clientes

Una requisito fundamental del middle-tier en una arquitectura de tres capas es la capacidad de atender de manera concurrente a múltiples clientes remotos. Es decir, se debe poder aislar la sesión de cada cliente que interactúa con el servidor, de las sesiones del resto de los clientes.

En el modelo desarrollado, la sesión de un cliente contra el servidor es llamada "contexto de ejecución", y es definida por la clase ExecutionContext.

Cuando un cliente se conecta al sistema, un nuevo contexto de ejecución es creado y a partir de este momento, toda la interacción entre el cliente y el servidor pasará por dicho contexto.

La estructura de una instancia de ExecutionContext se compone de un canal de comunicación con el cliente (una instancia de la clase CommunicationChannel) que maneja los aspectos de bajo nivel del envió y la recepción de mensajes remotos utilizando sockets TCP/IP, una conexión a la base de datos y un conjunto de procesos (o threads) llamado workerProcesses que ejecutan los comandos enviados por el cliente.

Durante su existencia, los contextos de ejecución ejecutan el siguiente ciclo: "duermen" hasta que su canal de comunicación haya recibido algún mensaje por parte del cliente, y cuando esto sucede, reciben el mensaje y lo ejecutan.

A continuación podemos observar la implementación de este ciclo:

```
ExecutionContext>>startMainLoop
     "Ejecuta el loop principal."

1          [ true ] whileTrue: [
2               communicationChannel dataAvailable ifTrue: [
3                    self receiveAndExecuteMessage ].
4               communicationChannel waitForDataUntil:
5                    (Socket deadlineSecs: 30) ].
```

En la línea 1, "`[ true ] whileTrue: [ ... ]`" indica que es un ciclo infinito, ya que la condición del ciclo es siempre verdadera.

En el código podemos observar que cuando el canal de comunicación tiene datos disponibles (un mensaje completo o parte de un mensaje) el contexto de ejecución se envía el mensaje `#receiveAndExecuteMessage` a si mismo.

La implementación de `#receiveAndExecuteMessage` es la siguiente:

```
ExecutionContext>>receiveAndExecuteMessage
     "Recibe un mensage, lo procesa y lo ejecuta."

 1         | receivedMessage |
 2         receivedMessage  :=  ClientToServerMessage  fromXmlString:
 3              communicationChannel receiveStringOrNil.
 4         receivedMessage isNil ifTrue: [
 5              self send: (ApplicationError parsingMessageError).
```

```
 6              ^self ].
 7          receivedMessage isLogoffMessage ifTrue: [
 8              [ self finalize ] fork.
 9              ^self ].
10          self execute: receivedMessage.
```

Aquí podemos observar (en las líneas 2 y 3) que el contexto de ejecución obtiene el string recibido (un documento XML) desde su canal de comunicación y con este string intenta instanciar un ClientToServerMessage. Si el string recibido es incorrecto (líneas 4, 5 y 6), la instanciación fallará y entonces de le devuelve al cliente un error (en este caso es un error de parseo). Si el mensaje se pudo interpretar entonces pueden suceder dos cosas: si es un mensaje de logoff está indicando que el cliente quiere terminar la sesión de ejecución, por lo que el contexto de ejecución es finalizado (líneas 7, 8 y 9); si el mensaje es de otro tipo, entonces es ejecutado (línea 10).

Ahora veamos la implementación del método `#execute:` en Execution Context. Como vimos en el código anterior, este método recibe como argumento una instancia de alguna de las subclases concretas de ClientToServerMessage.

La implementación es la siguiente:

```
ExecutionContext>>execute: aRemoteMessage

 1          | workBlock executionProcess |
 2          workBlock := [
 3              | p |
 4              p := Processor activeProcess.
 5              self addWorkerProcess: p.
 6              DatabaseConnectionManager default
 7                  addDatabaseConnection: databaseConnection
 8                  forProcess: p.
 9              aRemoteMessage executeInContext: self.
10              DatabaseConnectionManager default
11                  removeDatabaseConnectionForProcess: p.
12              self removeWorkerProcess: p. ].
13          executionProcess := workBlock newProcess.
14          executionProcess priority:
15              Processor userSchedulingPriority.
16          executionProcess resume.
```

En las líneas 2 a la 12 se define código que se ejecutará por un nuevo proceso (o thread) y en las líneas 13 a la 16 se activa este nuevo proceso. Esto se hace así para permitir volver rápidamente al ciclo que ejecuta el contexto de ejecución, y de esta forma poder manejar casi instantáneamente nuevos mensajes enviados por el cliente.

Ahora analicemos el bloque de código que ejecuta el nuevo proceso. En la línea 4 se obtiene una referencia al nuevo proceso. En Smalltalk, y por lo tanto en Squeak, todo es un objeto (esta simple oración indica una de las características más poderosas del lenguaje), inclusive los procesos que ejecutan el código (son instancias de la clase Process), entonces, podemos enviarle mensajes al objeto que representa el hilo de ejecución. En la línea 5, registramos dicho proceso en la colección de workerProcesses del contexto de ejecución, en las líneas 6, 7 y 8 asociamos la conexión a la base de datos del contexto de ejecución con el proceso.

La línea 9 es la más importante del bloque de código, en esa línea se invoca la ejecución del mensaje recibido sobre el contexto de ejecución. Y por último, en las líneas 10 a 12 se libera la conexión a la base de datos y se saca al proceso de la lista de workerProcesses.

Volviendo a la línea 9, este es un claro ejemplo del uso del polimorfismo. No importa de qué clase sea el mensaje remoto recibido como argumento (en el código es llamado aRemoteMessage), sino que pueda entender el método `#executeInContext:` con el contexto de ejecución como argumento.

Aquí también se ve una de las principales ventajas del uso del polimorfismo. Supongamos que dentro de cierto tiempo necesitáramos incorporar en el sistema un nuevo tipo de mensajes remotos que los clientes puedan enviarle al servidor de la aplicación. En ese caso, el código anterior, que maneja la ejecución de los mensajes, no debería ser modificado en absoluto. Lo único que se necesitaría escribir es una implementación de `#executeInContext:` en la clase del mensaje remoto nuevo incorporado al sistema.

Con respecto a la comunicación con el cliente asociado, el executionContext tiene asociado un canal de comunicación (instancia de la clase CommunicationChannel). Los communicationChannel son una abstracción de un socket TCP/IP con funcionalidad configurada dinámicamente. Esto se logra con una colección de capas de comunicación (instancias de la clase CommunicationLayer). Antes de enviar un mensaje por el socket, o inmediatamente después de recibir un mensaje desde el socket, el mensaje es procesado por las distintas capas de manera ordenada.

Por ejemplo, supongamos que un cliente necesita reducir lo máximo posible el tamaño de los mensajes intercambiados y además requiere que sean firmados digitalmente. Para esto, el canal de comunicación debería tener dos capas de comunicación, una que comprima y descomprima los mensajes (comprime al enviar y descomprime al recibir) y otra que firme digitalmente los mensajes (cuando son enviados) y que valide la firma de los mensajes recibidos.

Este es un ejemplo del pattern llamado "layers" (capas) [Buschmann96].

En la figura 6.3 podemos observar un diagrama de composición de instancias centrado en ExecutionContext.

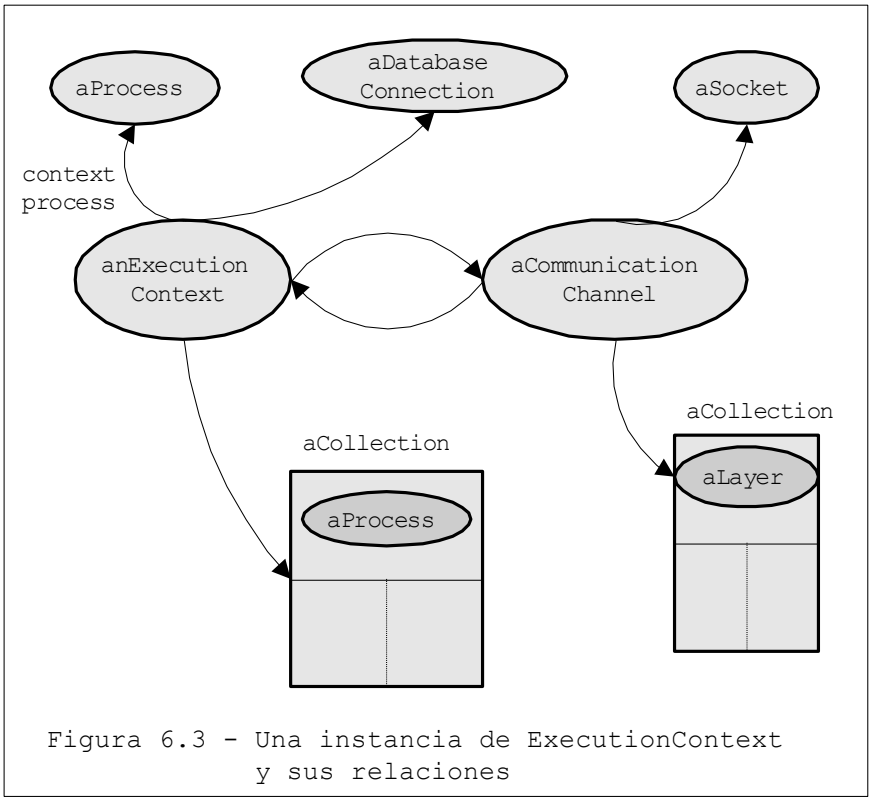



Figura 6.3 - Una instancia de ExecutionContext y sus relaciones

# Servicios de aplicación

El concepto de servicio de aplicación fue modelado con la clase ApplicationService. Para dar soporte a los servicios de aplicación se creó una subclase de ExecutionContext llamada ServiceExecutionContext. En esta clase se tiene la funcionalidad de obtener una referencia a un servicio de aplicación mediante su nombre, activar un servicio y luego poder iniciar la ejecución del mismo. En la jerarquía de mensajes remotos que se describió anteriormente (la raíz de esta jerarquía es la clase RemoteMessage), existen dos clases de mensajes remotos que deben ejecutarse en un ServiceExecutionContext, es decir, sólo se usan para tareas relacionadas con un servicio. Estas clases son ActivateAppServiceMessage y StartAppServiceMessage.

Volviendo a la clase ApplicationService, el ciclo de vida de un servicio de aplicación se divide en dos etapas. Primero debe ser activado por un cliente en el contexto de un

ServiceExecutionContext. Una vez que el servicio fue activado, el cliente puede configurar parámetros necesarios para la ejecución del servicio, y en la segunda etapa, una vez que fueron configurados adecuadamente todos los parámetros requeridos para la ejecución, el cliente de la aplicación puede pedir el inicio de la ejecución del servicio.

Para implementar este manejo de parámetros se utilizó lo desarrollado con los ApplicationObjects, por lo que la clase ApplicationService es una subclase de ApplicationObject, y por lo tanto hereda todo el manejo de atributos que fue documentado previamente. De esta forma, los parámetros del servicio de aplicación son modelados como atributos.

ApplicationService es una clase abstracta en donde se declaran tres métodos que deberán ser reimplementados en las subclases. Dichos métodos son los siguientes:

- `#validateRequiredAttributes`.
  En este método se valida que los argumentos necesarios para ejecutar el servicio de aplicación estén definidos con un valor apropiado.
- `#mustReturnResultToClient`.
  Aquí se debe devolver un valor booleano que indica si el resultado de la ejecución del servicio de aplicación debe ser retornado al cliente. Por ejemplo, si tenemos un servicio que construye una lista de elementos que pidió el cliente, entonces este método tiene que devolver true (verdadero) ya que justamente el cliente está esperando el resultado (la listas de items). Por otro lado, si tenemos un servicio que debe eliminar un elemento, entonces no es necesario que se retorne un valor al cliente, es suficiente indicar si el servicio se pudo ejecutar con éxito o no.
- `#startNow`.
  En este método se inicia la ejecución del servicio de aplicación. Obviamente que lo que aquí se haga dependerá de la aplicación en particular y del modelo de objetos del sistema.

En el capítulo VII se podrá observar un ejemplo de la manera en que fueron implementados estos métodos en un servicio de aplicación en particular.

# Resumen del capítulo

A diferencia de los capítulos anteriores, en este capítulo se analizaron los aspectos concretos de este trabajo de grado: el framework desarrollado.

Hemos visto que el framework se divide en cinco subsistemas. En este capítulo analizamos cuatro de estos subsistemas (objetos de aplicación y atributos, soporte a mensajes remotos entre los clientes y el servidor, manejo de múltiples contextos de ejecución e implementación de los servicios de aplicación) y en el capítulo siguiente se analiza el subsistema restante, el soporte de persistencia a los objetos.

# Capítulo 7

# Manejo de la Persistencia

En este capítulo se describe el manejo de la persistencia y su relación con los conceptos analizados en el capítulo anterior.

Es importante aclarar que el manejo de la persistencia es totalmente independiente del medio físico en donde se almacenen realmente los datos persistentes. Esto se logró mediante la clase DatabaseConnection.

## Objetos persistentes

La clase principal para el manejo de la persistencia es PersistentObject (objeto persistente), subclase de ApplicationObject. La idea es que para poder almacenar un objeto en el medio de persistencia, la clase del objeto debe ser subclase de PersistentObject.

El protocolo principal de la clase PersistentObject es el siguiente:

Métodos de clase:

- `#select:`.
  Dada una condición filtro, retorna todas las instancias en las que el filtro sea verdadero.
- `#selectAll`.
  Retorna todas las instancias desde el medio de persistencia.

Métodos de instancia:

- `#savePersistent`.
  Almacena al receptor en el medio de persistencia. Si es la primera vez (si el objeto no fue persistido anteriormente) lo agrega, en otro caso, actualiza el registro del objeto en el almacenamiento.
- `#removePersistent`.

Elimina al receptor del medio de persistencia.

- `#isSaved`.
  Retorna un booleano que indica si el receptor ya se encuentra en el medio de persistencia o no.
- `#databaseConnection`.
  Retorna la conexión a la base de datos asociada (luego se dará más detalle sobre este punto).

Del protocolo anterior, los mensajes más importantes son `#savePersistent` y `#removePersistent`. En la implementación de estos mensajes, se delega el trabajo a la databaseConnection asociada.

Por ejemplo, en el caso de `#savePersistent`:

```
PersistentObject>>savePersistent

1          ^self isSaved
2               ifTrue: [ self databaseConnection insert: self ]
3               ifFalse: [ self databaseConnection update: self ]
```

En este caso, dependiendo de si el objeto ya existe en el medio de persistencia o no, se le pide a la conexión asociada que lo inserte (línea 2) o que lo actualice (línea 3), y en ambos casos, el objeto receptor es pasado como argumento a los mensajes activados (mediante la pseudo-variable self).

El manejo de conexiones a la base de datos es un punto delicado ya que se deben aislar los accesos múltiples originados a partir de los diferentes clientes usando el sistema en un momento dado. Una vez más, para soportar este requerimiento se utilizaron los contextos de ejecución. Como vimos anteriormente, los contextos de ejecución referencian una conexión con la base de datos, esta conexión se asocia con el workerProcess del contexto (recordemos que workerProcess es una variable de instancia de los contextos de ejecución que referencia al thread encargado de la ejecución en el contexto).

Cuando a un persistentObject se le pide su conexión con la base de datos, este le delega el pedido a la única instancia de la clase DatabaseConnectionManager, dicha instancia maneja las conexiones y su relación con los contextos de ejecución.

A continuación podemos observar el código utilizado por DatabaseConnectionManager para obtener la conexión con la base de datos:

```
DatabaseConnectionManager>>databaseConnection

1          ^connectionsTable
2               at: (Processor activeProcess)
3               ifAbsent: [ defaultConnection ].
```

Como podemos observar, la variable de instancia connectionsTable asocia procesos (o threads) con conexiones a la base de datos, en la línea 2 se accede al proceso actual

(mediante el mensaje `#activeProcess` a Processor) y si el proceso no tiene asociada una conexión, usa la definida por defecto (línea 3).

En la figura 6.1 tenemos un diagrama en donde podemos ver las relaciones entre las clases principales que manejan la persistencia:

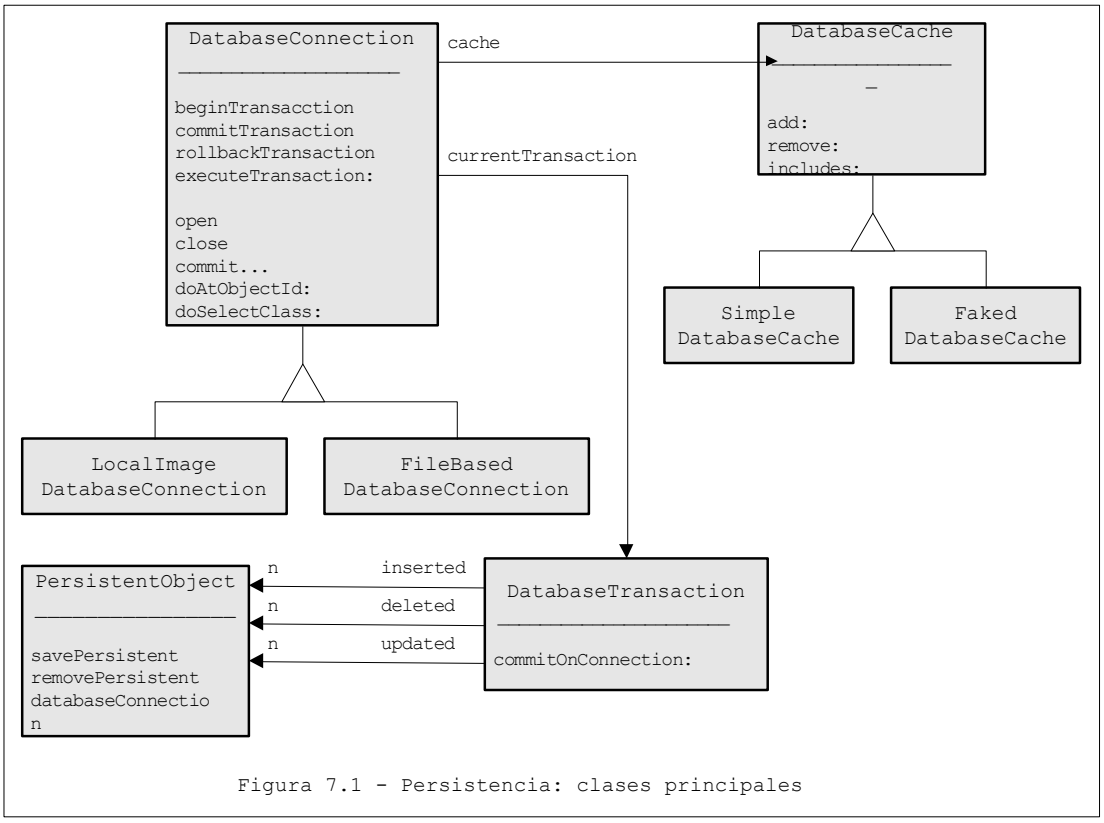



Figura 7.1 - Persistencia: clases principales

**Crítica a la clase PersistentObject**

El hecho de tener que subclasificar la clase PersistentObject para conseguir que las instancias de una clase sean persistentes no es algo bueno. Lo mismo sucede con la clase ApplicationObject para el manejo de atributos.

En este trabajo se utilizo esta solución porque es la manera más rápida de obtener la funcionalidad buscada (ya sea persistencia o soporte de atributos), pero se supo desde un comienzo que esta no es la solución ideal.

El hecho de tener que subclasificar estas clases, hace que nuestro framework sea de "**caja blanca**" y lo mejor es tener un framework de "**caja negra**" [Johnson91].

En un framework de caja blanca, los usuarios del framework deben subclasificar las clases principales del mismo, redefiniendo métodos utilizados por el framework. Esto

implica que los usuarios del framework deben conocer la implementación para poder utilizar el framework de forma adecuada.

Una alternativa para customizar un framework es proveer un conjunto de componentes que provean el comportamiento específico para una aplicación. La idea es que el framework tenga una librería de componentes, y que pueda ser configurado mediante la instanciación de dichos componentes.

Los frameworks de caja negra son una evolución de los frameworks de caja blanca [Roberts96], ya que no es necesario conocer la implementación interna del framweork para poder usar el framework de manera apropiada.

Volviendo a la persistencia de los objetos, lo ideal es no imponer ninguna jerarquía para que las instancias de una clase sean persistentes, es decir, que cualquier objeto independientemente de cuales sean sus superclases pueda ser almacenado en el medio de persistencia.

Además de las ventajas recién nombradas, esto permitiría que el subsistema de persistencia pueda utilizarse con aplicaciones existentes sin necesidad de hacer grandes cambios en dichas aplicaciones.

# Conexiones con la base de datos

La clase DatabaseConnection es una clase abstracta que define el protocolo básico para persistir y recuperar objetos de un medio persistente. Esta abstracción permitió independizar el manejo de la persistencia en el framework con el modo en que se almacenen físicamente los datos a persistir. Por ejemplo, podemos tener una jerarquía como la siguiente:

```
DatabaseConnection (A)
       LocalImageDatabaseConnection
       FileBasedDatabaseConnection
       RelationalDatabaseConnection (A)
              MySQLDatabaseConnection
              DB2DatabaseConnection
```

Una limitación del diseño actual es que no podríamos tener más de un tipo de conexión en el mismo servidor y realizar transacciones que involucren a más de una de estas clases de conexiones.

Todo el manejo de persistencia en DatabaseConnection se basa en que las subclases redefinan de manera apropiada los siguientes métodos de instancia:

- `#open`.

Abre la conexión. Esto implica validar el usuario, la password y el resto de la configuración que la conexión tenga.

- `#close`.
  Cierra la conexión. Esto se utiliza para liberar recursos.
- `#commitDatabaseTransaction:`.
  Este método recibe como argumento una instancia de la clase DatabaseTransacction y efectúa los cambios indicados en la transacción sobre el medio de persistencia.
- `#doAtObjectId:`.
  Este método recibe como argumento un objectId y retorna el PersistentObject asociado o nil si no existe un objeto con el objectId recibido en el medio de persistencia.
- `#doSelectClass:`.
  Dada una clase, este método debe retornar todas las instancias de dicha clase que se encuentren almacenadas en la base de datos.

De esta forma, si queremos crear una nueva clase subclase de DatabaseConnection (por ejemplo, OracleDatabaseConnection) redefiniendo estos métodos con la implementación adecuada, el mecanismo de persistencia está listo y no es necesario realizar ningún cambio.

# Cache de objetos en memoria

Una característica interesante de DatabaseConnection es que ya esta implementado el manejo de un cache de objetos. El cache de objetos se mantiene vivo mientras dura una transacción y se usa para optimizar el acceso a objetos que ya fueron buscados (la idea es no tener que leer desde la base de datos el mismo objeto dos veces mientras dura una transacción).

La jerarquía de clases que modela el cache es la siguiente:

```
DatabaseCache (A)
       SimpleDatabaseCache
       FakedDatabaseCache
```

DatabaseCache es una clase abstracta que define el protocolo del cache, la subclase SimpleDatabaseCache provee una implementación básica de dicho protocolo en donde se utiliza un diccionario que asocia objectIds con los objetos del cache. La clase FakedDatabaseCache (la traducción podría ser: cache simulada) es una implementación del

pattern "Null Object" [Wolf96]. En esta clase se redefinen todos los métodos de DatabaseCache de manera que no haga nada. Esto es bueno para evitar condiciones de bifurcación en DatabaseConnection del tipo: "si el cache esta activo hacer [....] si no esta activo hacer [....]".

# Transacciones

En el contexto del manejo de bases de datos, una transacción es una colección de acciones contra la base de datos con las siguientes propiedades:

- **Atomicidad**. La transacción se ejecuta toda completa o no se ejecuta nada.
- **Consistencia**. La transacción debe preservar la consistencia interna de la base de datos.
- **Aislación**. La transacción debe ejecutarse como si estuviese sola, independientemente del resto de las transacciones que se pudieran estar ejecutando contra la base de datos.
- **Durabilidad**. Los resultados de la transacción no deben perderse en caso de fallas.

Estas propiedades son conocidas como "propiedades ACID" (ACID - Atomicity, Consistency, Isolation, Durability). Gracias a estas propiedades, las transacciones se usan como la unidad fundamental de recuperación, consistencia y concurrencia en un sistema cliente/servidor [Orfali99].

En nuestro modelo, una transacción representa una unidad de trabajo en donde se pudo haber insertado objetos a la base de datos o el borrado o la modificación de objetos existentes en la base de datos.

Las instancias de la clase DatabaseTransacction representan este concepto, por lo que contienen tres colecciones con objetos insertados, eliminados y modificados.

El manejo de las transacciones se realiza desde DatabaseConnection.

Internamente se lleva un contador de las llamadas al método que inicia una nueva transacción (`#beginTransaction`) para poder anidar dichas llamadas.

El protocolo para el manejo de transacciones es el siguiente:

- `#beginTransaction`.
  Incrementa el contador de transacciones y si dicho contador estaba en cero, crea una nueva transacción (instancia de la clase DatabaseTransacction) y la marca como la transacción actual en la conexión.
- `#commitTransaction`.
  Decrementa el contador de transacciones y si dicho contador llego a cero y el flag de rollback es falso, refleja en la base de datos los cambios representados por la transacción.
- `#rollbackTransacction`.
  Setea en verdadero el flag que indica que se debe volver atrás la transacción actual y decrementa el contador de transacciones. Si el contador llego a cero descarta la transacción.
- `#executeTransaction:`.
  Este método recibe como argumento un bloque de código a ejecutarse de manera transaccional. Lo que se hace es llamar a `#beginTransaction`, ejecutar el bloque de código y si el resultado es un error llamar a `#rollbackTransaction`, mientras que si todo salió bien se invoca el método `#commitTransaction`.

El método más importante en la clase DatabaseConnection es `#commitDatabaseTransaction:` que recibe una instancia de DatabaseTransaction como argumento. Este método es uno de los que debíamos redefinir si queremos generar un nuevo tipo de conexión ya que en la clase abstracta DatabaseConnection se define pero no se implementa. Es en este método en donde debemos garantizar que los cambios se hacen todos o no se hace ninguno sobre la base de datos. Y también aquí se debe chequear si hubo interferencia con otra transacción.

# Resumen del capítulo

En este capítulo se describió cómo fue implementado el manejo de la persistencia de los objetos. El subsistema de persistencia soporta transacciones y un cache de objetos, aunque todo esto depende de la clase abstracta DatabaseConnection. En esta clase se especifican los métodos que deberían redefinir las subclases para dar soporte persistente en diferentes medios (bases de datos relacionales, acceso directo a archivos, etc).

Por otro lado, se remarco que en el estado actual el framework es de caja blanca, y se sugirió como un posible trabajo futuro extender lo desarrollado para obtener un framework de caja negra.

Con el subsistema de persistencia se termina la descripción de los cinco subsistemas que componen al framework desarrollado.

En el capítulo siguiente analizaremos de qué manera fue implementada una aplicación ejemplo que se uso para testear el framework desarrollado y, principalmente, para verificar la utilización del concepto de servicio de aplicación.

# Capítulo 8

# Una Aplicación Ejemplo

En este breve capítulo se describe la aplicación ejemplo que fue desarrollada para testear la construcción de un sistema con arquitectura de tres capas utilizando servicios de aplicación.

El sistema ejemplo elegido consiste en tener herramientas que permitan a varios usuarios hacer desarrollo remoto sobre un ambiente Squeak.

A continuación explicaremos cómo se desarrolla software normalmente en Squeak y los conceptos modelados para permitir el desarrollo por parte de un equipo de personas, luego se describirán los casos de uso esenciales identificados y los servicios de aplicación correspondientes y por último se comenta cómo fue realizado el cliente de la aplicación.

## Programación en equipo utilizando Squeak

Para el desarrollo de software, Squeak provee un IDE (Integrated Development Environment) basado en el concepto de imagen. Una imagen Squeak es un archivo que contiene los objetos que componen al ambiente Squeak. Entre otras cosas, además de los objetos comunes se encuentran los procesos que se deben ejecutar y el código compilado de todos los métodos (los bytecodes). Este archivo esta codificado usando un formato binario independiente de la plataforma. Cuando la máquina virtual es ejecutada, lee el archivo imagen para inicializar el ambiente.

El concepto de imagen es muy útil para un desarrollador ya que permite guardar el estado del sistema, cerrar el ambiente y mas adelante en el tiempo, al leer la imagen nuevamente, volver al estado en que fue guardado.

Una de las características del ambiente Squeak estándar, es que no esta pensado para un equipo de desarrolladores. No tiene herramientas que permitan el desarrollo en paralelo por más de un usuario.

Para permitir el desarrollo en una misma imagen Squeak por parte de varios usuarios, fueron modelados (como clases) dos conceptos importantes: Developer (desarrollador) y ClassDefinition (definición de clase).

Una instancia de la clase Developer representa a un programador en un equipo. Esto implica que a partir de ahora, para que un usuario pueda modificar clases en el ambiente Squeak, debe tener asociado un objeto Developer que lo represente. Cuando el usuario se conecta al sistema se le pide el nombre de usuario y una password, e inmediatamente se busca que exista una instancia de la clase Developer asociada con el usuario.

Por otro lado, con la clase ClassDefinition se modelan aspectos relacionados con el desarrollo en equipo que no se encuentran en las clases estándar de Squeak. Para cada clase en la imagen que este disponible para ser vista o modificada remotamente por el equipo de desarrollo, se deberá tener una instancia de la clase ClassDefinition asociada. Las instancias de ClassDefinition, además de conocer a qué clase en el ambiente Squeak representan, tienen un dueño (instancia de la clase Developer) y un indicador que determina si hay una edición abierta o no. A continuación, en la figura 8.1, se muestra un diagrama con las relaciones entre las instancias de estas clases.

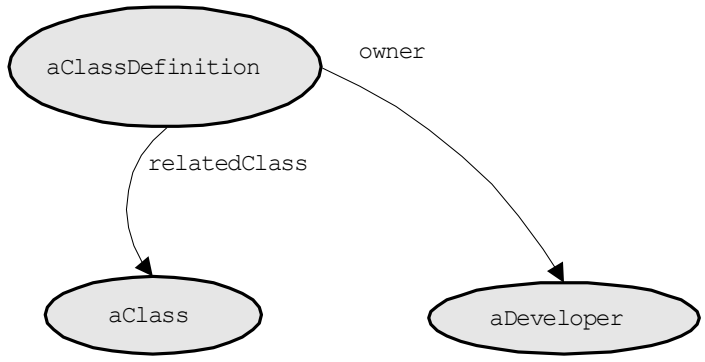



Figura 8.1 - Relaciones entre instancias de ClassDefinition y Developer

Para que un usuario (o Developer) pueda modificar una clase, se siguen los siguientes pasos. Primero se debe verificar que la clase no tenga una edición abierta (esto se hace mediante el flag llamado isOpen en las instancias de ClassDefinition), si la ClassDefinition está cerrada, entonces el Developer se pone como dueño y abre una nueva edición. A partir de este momento, solo el dueño de la clase puede realizar cambios sobre esta. Cuando terminó de hacer los cambios, el dueño cierra la edición y entonces se permite que algún otro usuario (o Developer) se convierta en el dueño de la ClassDefinition y realice modificaciones en la clase asociada.

# Casos de uso y servicios de aplicación

Los casos de uso encontrados para este sistema representan las acciones básicas que los desarrolladores pueden hacer remotamente contra el ambiente Squeak. Para cada uno los casos de uso, se creo una subclase de ApplicationService que lo representa.

La jerarquía de clases que modelan los servicios de aplicación se muestra a continuación (la letra A significa que la clase es abstracta):

```
ApplicationService (A)
       GetClassNamesService
       ClassDefinitionService (A)
              GetClassDefinitionService
              ChangeClassOwnerService
              OpenClassEditionService
              CloseClassEditionService
              MethodService (A)
                     NewMethodService
                     ExistentMethodService (A)
                            GetMethodSourceService
                            SetMethodSourceService
                            RemoveMethodService
```

Si tomamos como ejemplo el servicio de aplicación que nos permite cambiar el dueño de una clase, podemos ver una implementación de los métodos abstractos que toda subclase de ApplicationService debe implementar. Como vimos en el capítulo V, estos métodos son: `#validateRequiredAttributes`, `#mustReturnResultToClient` y `#startNow`. A continuación podemos ver el código de estos métodos en la clase ChangeClassOwnerService:

```
ChangeClassOwnerService>>validateRequiredAttributes
     "Retorna un booleano indicando si el servicio tiene
            seteado todos los atributos necesarios para su
                   ejecucion."

1         classDef:=     ClassDefinition     atClassNamed:     self
className.
2         developer := Developer atName: self developerName.
3         ^classDefinition notNil and: [ developer notNil ]
```

Como se puede observar en el código, los atributos requeridos del servicio anterior son tener una ClassDefinition asociada y un Developer válido.

```
ChangeClassOwnerService>>mustReturnResultToClient
     "Retorna un booleano indicando si el servicio debe
            retornar un valor al cliente o no."

1          ^false
```

Este método es muy simple, lo único que se hace es devolver false (falso), indicando que el servicio no retorna ningún valor en particular al cliente (a este sólo le basta saber si el servicio se pudo ejecutar o no).

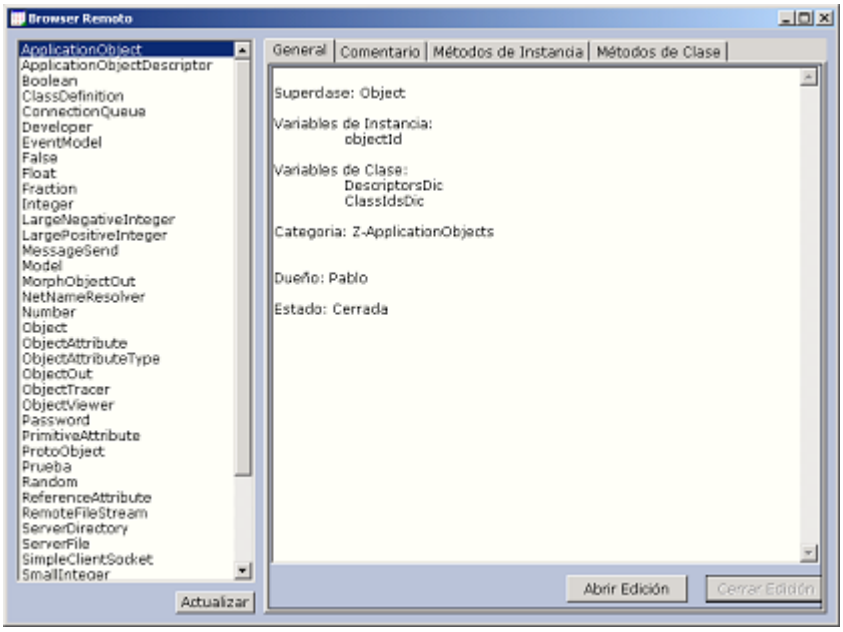
```
ChangeClassOwnerService>>startNow
     "Ejecuta el servicio."

1          classDefinition isOpen ifTrue: [
2               ^ApplicationError
3                    errorCode: '1000'
4                    description: 'La clase está abierta.' ].
5          classDefinition ownerDeveloper: developer.
6          ^classDefinition savePersistent
```

Aquí por fin se ejecuta el servicio de aplicación. Primero se chequea si la clase esta abierta, en ese caso se devuelve un error indicando que el servicio no pudo ser ejecutado. Si la clase no está abierta, se asigna el nuevo dueño (owner) y se almacena el cambio de manera persistente.

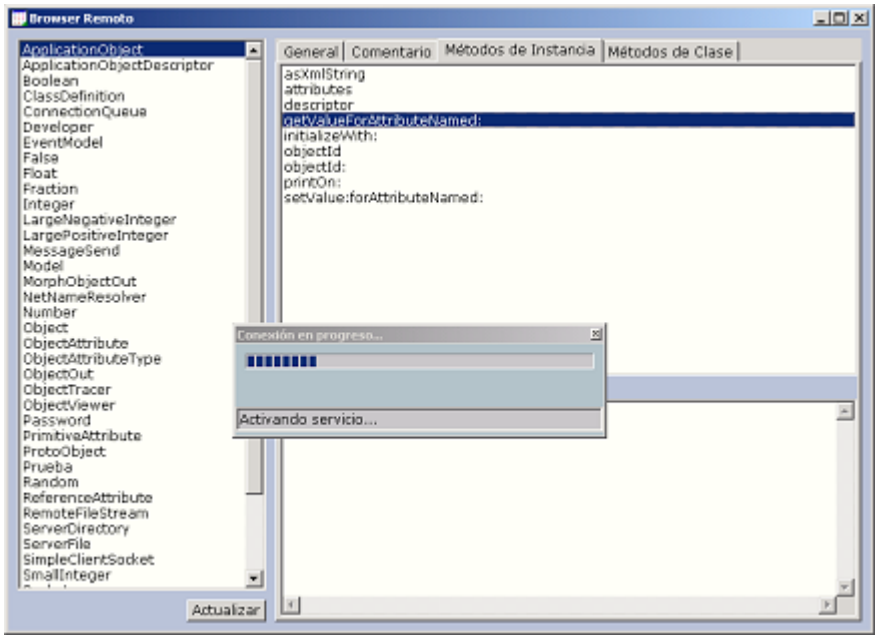
# Desarrollo del cliente de la aplicación

El cliente de la aplicación se desarrollo utilizando el ambiente Dolphin Smalltalk [Object Arts]. Es una aplicación que sólo se puede ejecutar bajo el sistema operativo Microsoft Windows y que utiliza un API para el manejo de sockets y otra para el manejo de documentos XML, propias de dicho sistema operativo.

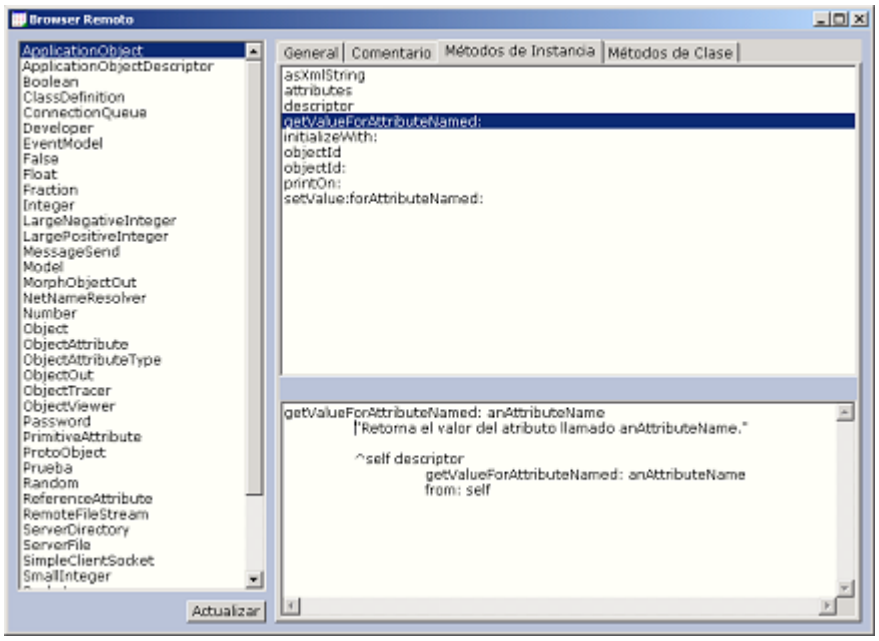
A continuación podemos observar una serie de pantallas de la aplicación cliente. En la figura 7.2 el usuario ha seleccionado una clase (en este caso ApplicationObject) y está mirando la solapa general en donde se indica cuál es la superclase, las variables de instancia, las variables de clase, la categoría, quien es el dueño (un Developer) y el estado de la edición actual (cerrada).




En la siguiente figura el usuario está parado en la solapa en donde se muestran los métodos de instancia de la clase, y al seleccionar uno de estos métodos (el método `#getValueForAttributeNamed:`) el usuario ha activado el servicio que recupera el código fuente de un método y se lo muestra.



Por último, en la figura 7.4, se puede observar que el servicio fue ejecutado y el usuario está viendo el código fuente del método seleccionado.




Cabe aclarar que, además de poder ver las definiciones de las clases y los códigos fuente de los métodos, el cliente de la aplicación permite modificar o borrar métodos existentes y crear métodos nuevos.

# Capítulo 9

# Conclusiones y Trabajo Futuro

En este capítulo se describen las conclusiones obtenidas como resultado de esta tesis, y los posibles trabajos futuros que se podrían realizar.

## Conclusiones

Los sistemas con arquitecturas de tres capas presentan muchas ventajas comparados con los sistemas cliente/servidor tradicionales de dos capas. Sin embargo, desarrollar un sistema con una arquitectura 3-tier es más complejo y costoso.

En la mayor parte de la bibliografía consultada se ataca el problema desde un punto de vista puramente tecnológico, es decir, se tocan aspectos referentes a la comunicación entre los clientes y el servidor, el modelo de interacción con el usuario a implementar y el soporte de concurrencia necesario en el servidor de la aplicación, también llamado middle-tier.
Obviamente, todos esos problemas existen y es necesario resolverlos. Sin embargo, el tema puede abordarse desde otro punto de vista.

En el contexto del paradigma de orientación a objetos, en este trabajo se analizó de qué forma impacta el hecho de tener una arquitectura de tres capas en el modelo de objetos resultante, y de qué forma se puede reducir dicho impacto.

El modelo de casos de uso presentado por Jacobson [Jacobson92], el trabajo sobre análisis esencial de sistemas de McMennamin y Palmer [McMenamin84] y los casos de uso esenciales de Constantine y Lockwood [Constantine01] fueron fundamentales para orientar el desarrollo de este trabajo hacia los servicios de aplicación.
Basándonos en estos estudios, fue tomando forma el concepto de servicio de aplicación.

La definición de los servicios de aplicación representa un avance en un área poco explotada en el resto de los trabajos similares ya que, como dijimos anteriormente, tener objetos que representan a los servicios de aplicación en el modelo de objetos del sistema,

sirve para acotar y manejar el hecho de que el sistema se ejecute en un servidor de aplicaciones en el contexto de una arquitectura de tres capas.

Los servicios de aplicación nos ayudan a definir de manera precisa cómo van a interactuar los clientes con los objetos que componen el middle-tier, acotando el impacto en el modelo de objetos del que hablamos anteriormente.

Por otro lado, el framework desarrollado en Squeak, permitió llevar al software el concepto de servicio de aplicación y se pudo verificar la utilidad de los mismos con el ejemplo que fue presentado en el capítulo 8 (desarrollo de software remoto en Squeak).

Para desarrollar este framework se diseñaron y programaron los aspectos de concurrencia, comunicaciones a través de sockets TCP/IP, el subsistema de mensajería XML y el soporte de persistencia de objetos.

Todo esto, además de ser una experiencia de desarrollo muy interesante, permite tener una herramienta para atacar los problemas del tipo tecnológico que nombramos al principio de esta sección, y no tener que resolverlos desde cero cada vez que tengamos que desarrollar una aplicación con arquitectura de tres capas.

Sin embargo, tal como fue planteado en capítulos anteriores a medida que se presentaron los distintos temas, y como veremos en la siguiente sección, todavía existen muchas mejoras posibles para hacerle a nuestro framework.

Resumiendo, el uso de servicios de aplicación con el framework desarrollado en este trabajo proveen un soporte muy importante para el desarrollo de sistemas con arquitectura de tres capas.

Los beneficios son dobles: por un lado, el concepto de servicio de aplicación es muy útil para estructurar la implementación del sistema y, por otro lado, el hecho de tener un framework permite que se acelere el desarrollo de nuevas aplicaciones y que las mejoras hechas al mismo impacten en todas las aplicaciones que lo puedan llegar a utilizar.

# Trabajo futuro

A continuación analizaremos los posibles puntos a mejorar en el futuro. El orden de presentación de estos puntos es de temas conceptuales generales a puntos particulares y de implementación.

La tarea más importante a realizar es probar el concepto de servicio de aplicación en diferentes tipos de aplicaciones.

Como vimos en el capítulo anterior, en este trabajo los servicios de aplicación se utilizaron para implementar un sistema de desarrollo remoto de software en Squeak. Sería muy interesante probar los servicios de aplicación en otro tipo de sistemas.

Los beneficios de esta tarea serían muy importantes ya que no solo se utilizaría el concepto de servicio de aplicación en otro contexto, sino que también, con la experiencia ganada, se puede revisar el concepto y hacer las modificaciones apropiadas.

Con respecto a las decisiones que dieron marco al framework desarrollado, hay ciertos puntos que podrían modificarse para investigar alternativas interesantes que no fueron exploradas en este trabajo.

Los más importantes son:

- Probar otro mecanismo en cuanto al manejo de la interacción con los usuarios.
  Por ejemplo, se podría tener una presentación distribuida basada en páginas HTML. En este marco no tiene sentido la jerarquía de clases que modelan los mensajes remotos desarrollada en el framework, pero tendríamos que resolver la configuración y la activación de los servicios en el servidor de la aplicación.
  La principal ventaja de una solución de este tipo es que no es necesario instalar ningún software en las máquinas cliente (sólo se necesita un web-browser, y hoy en día es muy común que los mismos sistemas operativos tengan algún browser instalado). Por otro lado, el punto en contra más importante es que la interacción con el usuario se ve limitada al uso de formularios HTML (aunque hoy en día tenemos varias alternativas para mejorar la calidad de esta interacción).

- Probar una alternativa al uso de mensajes XML.
  Aquí hay dos puntos a tener en cuenta, por un lado tenemos el tema de la performance y tamaño de los mensajes XML y por otro lado tenemos la distribución de objetos.
  Con respecto al primer punto, si bien los mensajes XML son muy descriptivos y relativamente simples de generar, utilizando por ejemplo un formato binario, el tamaño de los mensajes puede ser reducido, y también se podrían bajar los tiempos utilizados para parsear e interpretar los mensajes recibidos tanto en los clientes como en el servidor.
  Con respecto a la distribución de objetos, en el contexto de CORBA o algún mecanismo de pasaje de mensajes remotos como RMI, existe todo un abanico de posibilidades para investigar. Por ejemplo, analizar cómo impactaría el uso de CORBA con los servicios de aplicación.

En cuanto a la implementación del framework, los puntos más interesantes para desarrollar son:

- Pasar de un framework de caja blanca a uno de caja negra.
  Como explicamos en el capítulo 7, sería necesario rediseñar el framework en cuanto a las clases ApplicationObject y PersistentObject.
  Esto no sólo permitiría que nuestro framework pueda utilizarse con aplicaciones ya existentes sin tener que modificar las jerarquías de clases en dichas aplicaciones, sino que también ayudaría a construir un framework más robusto y más completo.

- Desarrollar una herramienta para definir los servicios de aplicación.

Actualmente un servicio de aplicación se modela con una clase que debe implementar cierto protocolo requerido por el resto del sistema. Este protocolo abarca mensajes que describen las características del servicio (por ejemplo, si el servicio debe devolver una respuesta al cliente) y mensajes que sirven para ejecutar el servicio.
Lo ideal sería que la especificación de un servicio de aplicación sea un objeto que sepa instanciar nuevos servicios. La idea de este punto es tener una herramienta que pueda utilizarse para crear las especificaciones.
De esta forma, se reduciría la cantidad de clases utilizadas para definir los servicios.
Además, sería posible agregar nuevos servicios o cambiar la definición de servicios existentes en una aplicación con el sistema ejecutándose. Obviamente, esta es una característica muy conveniente.

- Un punto que quedo pendiente es implementar las relaciones entre los servicios de aplicación. Como fue planteado en el capítulo 4, los servicios de aplicación pueden tener restricciones de concurrencia y de prerrequisitos que deben ser analizadas en el momento en que se intenta iniciar la ejecución del servicio. Estos puntos no fueron desarrollados en el framework.

- Por último, sería muy interesante avanzar un paso más en el desarrollo del subsistema de persistencia de objetos. Por ejemplo, para poder utilizar un motor comercial de bases de datos relacionales.

Como podemos ver, hay muchas oportunidades para investigar y mejorar los temas planteados en este trabajo.

# Referencias


[Ambler99] Scott W. Ambler. Mapping Objects to Relational Databases. 1999. (http://www.AmbySoft.com/mappingObjects.pdf)

[AppliedReasoning01] AppliedReasoning. Classic Blend: A Next Generation Java Technology For Building Real-time Internet Applications. 2001. (http://www.appliedreasoning.com)

[Berard] Edward V. Berard. Be Careful With Use Cases. (http://www.toa.com/pub/use_cases.htm)

[Bernstein97] Philip A. Bernstein, Eric Newcomer. Principles of Transaction Processing. Morgan Kaufmann Publishers Inc. 1997.

[Booch94] Grady Booch. Object-Oriented Analysis and Design with Applications (Second Edition). Addison-Wesley. 1994.

[Bosak99] Jon Bosak, Tim Bray. XML and the Second-Generation Web. Scientific American. May 1999 issue.

[Buschmann96] Buschmann, Meunier, Rohnert, Sommerlad, Stal. Pattern - Oriented Software Architecture, A System of Patterns. John Wiley & Sons. 1996.

[Canoo] Caoo Engineering. Ultra Light Client (ULC) Technology White Paper. (http://www.canoo.com/ulc/)

[Carroll95] John M. Carroll (Editor). Scenario-Based Design: Envisioning Work and Technology in System Development. John Wiley & Sons. 1995.

[Constantine01] Constantine, Lockwood. Structure and Style in Use Cases for User Interface Design.

[Ewald97] Tim Ewald. The Power of Processors. Component Strategies Magazine. June, 1997.

[Flanagan99] Flanagan, Farley, Crawford, Magnusson Java Enterprise in a Nutshell. O'Reilly & Associates, Inc. 1999.

[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software.
Addison-Wesley. 1995.

[GemStone] GemStone Systems, Inc. White Paper: Partitioning. (http://www.gemstone.com/products/s/papers_partition.html)

[Goldberg83] Adele Goldberg, Davud Robson. Smalltalk-80, The Language and its Implementation. Addison-Wesley. 1983.

[IBM] IBM Application Framework for e-business: Web Application Programming Model.

[IBM99] IBM. Ultra Light Client Guide and Reference. 1999.

[Ingalls97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, Alan Kay. Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. OOPSLA 1997, Atlanta.

[Jacobson92] Jacobson, Christerson, Jonsson, Overgaad. Object-Oriented Software Engineering. Addison-Wesley. 1992.

[Johnson91] Ralph Johnson, Brian Foote. Designing Reusable Classes. Department of Computer Science, University of Illinois, Urbana-Champaign. 1991.

[Krasner88] Glenn E. Krasner, Stephen T. Pope. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. ParcPlace Systems, Inc. 1988.

[Lajoie94] Richard Lajoie, Rudolf Keller. Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts and Motifs in Concert. Centre de recherche informatique de Montreal, Canada. 1994.

[Loomis95] Mary E. S. Loomis. Object Databases The Essentials. Addison-Wesley. 1995.

[McMenamin84] McMennamin, Palmer. Essential System Analysis. Prentice Hall. 1984.

[Montlick99] Terry Montlick. The Distributed Smalltalk Survival Guide. Cambridge University Press. 1999.

[Object Arts] Object Arts home page: http://www.object-arts.com/

[Orfali99] Robert Orfali, Dan Harkey, Jeri Edwards. Client/Server Survival Guide. Third Edition. John Wiley & Sons. 1999.

[Potel96] Mike Potel. MVP: Model-View-Presenter: The Taligent Programming Model for C++ and Java. Taligent, Inc. 1996.

[Renzel97] Klaus Renzel, Wolfgang Keller. Client/Server Architectures for Business Information Systems - A Pattern Language. 1997.

[Roberts96] Don Roberts, Ralph Johnson. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. University of Illinois. 1996.

[Seacord98] Robert Seacord, Scott Hissam. Browsers for Distributed Systems: Universal Paradigm or Siren's Song? 1998.

[Squeak] Squeak home page. http://www.squeak.org.

[Ungar87] David Ungar, Randall B. Smith. Self: The Power of Simplicity. OOPSLA '87 Conference Proceedings, pp. 227-241, Orlando, FL, October, 1987.

[Vinoski96] Steve Vinoski. CORBA, Integrating Diverse Applications Within Distributed Heterogeneous Environments. IONA Technologies Inc. 1996.

[Waldo94] Jim Waldo, Geoff Wyant, Ann Wollrath, Sam Kendall. A Note on Distributed Computing. Sun Microsystems Laboratories Inc. 1994.

[Wirfs-Brock90] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener. Designing Object-Oriented Software. Prentice Hall. 1990.

[Woolf96] Bobby Woolf. The Null Object Pattern. Pattern Languages of Program Design. 1996.