

Patrones Arquitectónicos de Aplicaciones Empresariales

Ingeniería del Software – III
Roberto García González

1 Introducción

Para estructurar aplicaciones empresariales sofisticadas, especialmente aquellas que operan a través de varias capas de procesamiento.

Arquitectura, las partes más importantes de una aplicación empresarial y cómo se conectan esas partes. Los patrones arquitectónicos representan decisiones importantes sobre esas partes, los patrones de diseño ayudan a materializar esa arquitectura.

El patrón arquitectónico dominante es el de capas, cómo se descompone una aplicación empresarial en capas y cómo estas capas trabajan juntas.

Las aplicaciones empresariales normalmente involucran persistencia de datos. Normalmente hay muchos datos y se acceden a ellos desde múltiples sitios al mismo tiempo, concurrentemente. Generalmente, deben integrarse con otras aplicaciones empresariales. Incluso si se dispone de una única tecnología para la integración de aplicaciones, existen diferencias en los procesos de negocios y modelos conceptuales que dificultan dicha integración.

1.1 Ejemplos de aplicación empresarial

- Aplicación B2C (de empresa a cliente) tienda en línea: la gente navega y compra. Existe un muy alto volumen de usuarios. La lógica de negocio es clara: capturar pedidos, cálculos simples de precios y envíos y la notificación de envíos. Una interfaz web genérica. Base de datos para el almacenamiento de pedidos y el inventario.

- Sistema para automatizar la tramitación de contratos de arrendamiento. Muchos menos usuarios. Lógica de negocio más compleja: cálculo de facturas mensuales para un contrato de arrendamiento, gestión de los adelantos y los pagos atrasados, validación de datos durante el proceso de solicitud de un contrato de arrendamiento, etc. Más complejidad en la interfaz de usuario, base de datos compleja e integración con paquetes externos de valoración de activos y fijación de precios.

- Aplicación de seguimiento de gastos simple para una empresa pequeña. Pocos usuarios y lógica simple. Desafíos: urgencia o posibilidades de crecimiento más adelante. Aunque esta aplicación puede ser pequeña, la mayoría de empresas tienen un montón de aplicaciones de este tipo, por lo que el efecto acumulativo puede ser una arquitectura compleja.

1.2 Patrones

Un patrón describe un problema que se produce frecuentemente y las pautas para solucionarlo. Se basan en la práctica y aunque la base es aplicar las mismas pautas, la solución nunca es exactamente la misma.

2 Arquitectura por Capas

Una de las técnicas más comunes que para descomponer un problema informático complejo. La capa superior utilizan los servicios definidos por la inferior, pero la capa inferior no es consciente de la capa superior. Además cada capa normalmente esconde los niveles más bajos de las capas por encima.

Ventajas:

- Sólo es necesario conocer la capa superior y es posible despreocuparse de las otras capas por debajo.
- Se pueden reemplazar capas por implementaciones alternativas de los mismos servicios básicos.
- Se minimizan las dependencias entre capas.
- Una capa puede ser reutilizada por múltiples servicios de niveles superiores.

Desventajas:

- Posibilidad de cambios en cascada (p.e. nuevo campo en la interfaz de usuario supone cambio en la base de datos y en todas las capas intermedias).
- Las capas extra pueden reducir el rendimiento.

La parte más complicada es decidir que capas y que funcionalidad poner en cada capa.

2.1 Evolución de las capas en las aplicaciones empresariales

El concepto de capas se popularizó en los 90 con la aparición de los sistemas cliente-servidor. Estos son sistemas de dos capas: la cliente con la interfaz de usuario y código de aplicación y la servidor, generalmente una base de datos.

El problema surgió con la lógica de dominio: reglas de negocio, validaciones, cálculos,... No encaja ni en la capa cliente ni en la servidor. La respuesta es la arquitectura en tres capas: una capa presentación para la interfaz de usuario, una capa de dominio con la lógica de dominio y una capa de datos. El éxito del modelo Web ha acabado de consolidar la arquitectura de tres capas.

2.2 La Arquitectura de 3 Capas

Esta arquitectura, la más común en la actualidad, está compuesta por las siguientes capas:

- **Presentación:** comprende la lógica para manejar la interacción entre el usuario y la aplicación. Posibles opciones son la línea de comandos, sistemas de menú basados en texto, el cliente rico o la interfaz de usuario HTML.
- **Fuentes de Datos:** tiene que ver con la comunicación con otros sistemas que llevan a cabo tareas en nombre de la aplicación. Monitores de transacciones, otras aplicaciones, sistemas de mensajería,... pero generalmente base de datos.
- **Lógica de Dominio:** la lógica de dominio o empresarial es la funcionalidad específica que la aplicación debe hacer para el dominio de trabajo. Los cálculos sobre la base de las entradas y los datos almacenados, la validación de los datos de entrada o determinar la fuente de datos implicada son ejemplos de funcionalidades en la capa de dominio.

Un aspecto difuso de esta arquitectura es lo que sucede cuando no hay una persona utilizando el software, por ejemplo un servicio web o un proceso por lotes (batch). En este caso se difumina la diferencia entre presentación y datos, ya que las dos se refieren a conexiones con el exterior de la aplicación. La distinción a hacer es entre una interfaz que la aplicación proporciona como un servicio a otros, y el uso que se haga de los servicios proporcionados por otros como fuentes de datos.

2.3 ¿Dónde ejecutar las capas?

La separación entre capas es útil incluso si se ejecutan en el mismo ordenador. De todas formas, para la mayoría de las aplicaciones la decisión es si ejecutar una parte del procesamiento en el cliente o en el servidor. El caso más simple es ejecutar todo en el servidor, por ejemplo utilizando una interfaz HTML, lo que facilita la actualización y mantenimiento. Los argumentos en favor de ejecutar parte en el cliente se justifican básicamente por conseguir una mayor capacidad de respuesta o el funcionamiento sin estar conectado (offline).

3 Organización de la Lógica de Dominio

El enfoque más sencillo para gestionar la lógica de dominio es el **Transaction Script**¹. Es esencialmente un procedimiento que recibe los parámetros de entrada de la presentación, los procesa, almacena información en la base de datos, invoca operaciones de otros sistemas y responde con más datos a la presentación.

La organización fundamental es un procedimiento único para cada acción del usuario, por ejemplo un script. De todas formas no tiene porque ser un procedimiento de código autocontenido, podría haber subrutinas compartidas entre diferentes **Transaction Scripts**.

Una tienda online tendría **Transaction Scripts** para añadir productos al carro de la compra, pasar por caja, mostrar el estado del pedido,...

Ventajas:

- Modelo procedural simple.
- Funciona bien con una capa de datos simple.
- Hace obvios los límites de la transacción: se abre al principio del procedimiento y se cierra al acabar.

Desventajas: aparecen muchas a medida que la complejidad de la lógica de dominio aumenta. A menudo habrá código duplicado, difícil de eliminar. Una lógica compleja requiere el uso de objetos para definir un **Domain Model**. Aunque esta es la mejor elección, cuanto más rico sea el modelo de dominio más complejo será el mapeo de éste a la base de datos relacional.

La tercera opción para estructurar la lógica de dominio es el **Table Module**, diseñado para funcionar con **Record Set**, el cual se obtiene como resultado de realizar consultas a la base de datos. El **Table Module** está a medio camino entre **Transaction Script** y **Domain Model**. Organizar

¹ Ver detalles de los patrones (marcados en negrita) en: <http://cv.udl.es/access/content/group/51002-0910/0%20-%20Patrons/Patterns%20of%20Enterprise%20Application%20Architecture-1>

la lógica de dominio en base a las tablas permite una mayor estructura que con los procedimientos de **Transaction Script**, pero carece de herencia, estrategias y otros patrones de orientación a objetos. La mayor virtud de **Table Module** es que encaja perfectamente en arquitecturas como .NET.

3.1 La interfaz a la lógica de dominio

Las principales opciones son la interfaz basada en HTTP o la orientada a objetos. El caso más común de interfaz HTTP es el navegador web pero puede utilizarse esta interfaz para más que simplemente presentaciones web utilizando Servicios Web, la comunicación entre sistemas sobre el protocolo HTTP, normalmente utilizando XML para los mensajes intercambiados.

El hecho de que XML es un formato común con parsers² disponibles en muchas plataformas permite la comunicación entre aplicaciones basadas en plataformas diferentes, como lo hace el hecho de que HTTP es prácticamente universal. HTTP también facilita los temas de seguridad ya que actualmente suele ser un puerto abierto en los cortafuegos, mientras que razones de seguridad hacen difícil abrir otros puertos muy a menudo.

Sin embargo, una interfaz HTTP es poco explícita, no queda claro a primera vista que es lo que hay, mientras una interfaz orientada a objetos tiene objetos con métodos y parámetros marcados más explícitamente. Si no es necesario atravesar ningún cortafuegos, puede resultar más conveniente una interfaz orientada a objetos, que también facilita mecanismos tales como los de control de acceso o las transacciones.

4 Presentación Web

Uno de los mayores cambios en las aplicaciones empresariales en los últimos años es la aparición de interfaces de usuario basadas en navegadores web. Las principales ventajas: no hay software de cliente que instalar, una aproximación de interfaz de usuario común y el acceso universal.

El trabajo del servidor web es interpretar las URL de las solicitudes que recibe y pasar el control a una aplicación del servidor web. Principalmente, hay dos formas de estructurar una aplicación en un servidor web: como un script o como una página de servidor. El script es un programa, normalmente con funciones o métodos para procesar la llamada HTTP, ejemplos son los scripts CGI o los servlets de Java.

La salida es un cadena de texto HTML, la respuesta, que el script genera utilizando las operaciones de escritura comunes en muchos lenguajes de programación. Las páginas de servidor surgieron para hacer este proceso más fácil. La página se compone de porciones de HTML y de scripts mezclados, de forma que es más intuitivo como se construye el HTML de salida. Esta aproximación es la más apropiada cuando hay poco procesamiento de la respuesta. Ejemplos de páginas de servidor son PHP, ASP y JSP. Los scripts funcionan mejor para interpretar la solicitud y las páginas de servidor para formatear la respuesta.

La presentación web se ajusta muy bien al patrón **Model View Controler**³. La solicitud HTTP le llega a un controlador de entrada. Se remite a la lógica de negocio a un objeto apropiado del

² Herramientas para procesar documentos XML

³ Modelo-Vista-Controlador

modelo. El objeto del modelo se comunica con la fuente de datos y hace todo lo indicado por la solicitud así como recopilar información para la respuesta. Cuando se ha hecho todo esto, retorna al controlador de entrada. Este último decide que vista es la más apropiada y le pasa el control remitiéndole los datos necesarios a través de algún tipo de objeto de sesión HTTP compartido. Este proceso se ilustra en la Fig. 1.

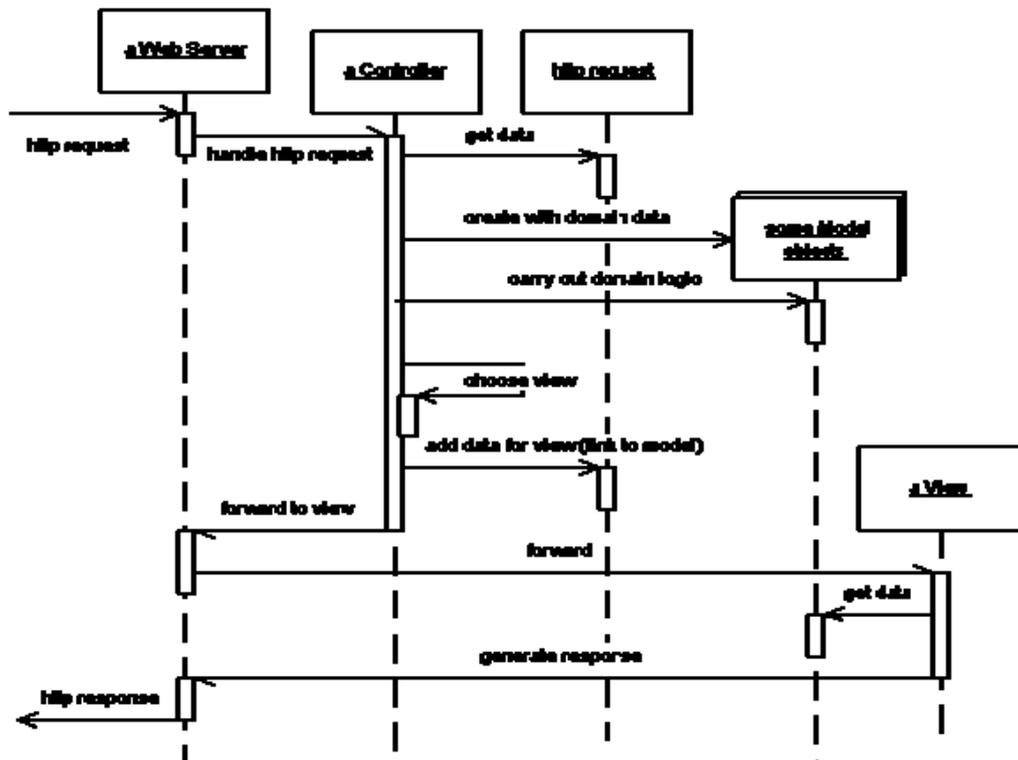


Fig. 1. Diagrama de secuencia del patrón Model View Controller en un servidor web

4.1 Patrones Vista

Respecto a la vista hay tres patrones: **Transform View**, **Template View** y **Two Step View**. Los patrones básicos para **Transform View** y **Template View** suponen una única etapa. **Two Step View** es una variación que puede aplicarse tanto a **Transform View** como a **Template View**.

Template View permite escribir la presentación en la estructura de la página e incrustar marcadores en la página para indicar dónde incluir el contenido dinámico. Es el patrón habitual para las páginas de servidor. Facilita el desarrollo pero provoca un código desordenado que se hace difícil de mantener.

Transform View utiliza algún tipo de lenguaje de transformación, normalmente XSLT que es muy efectivo para datos XML. El controlador de entrada selecciona la transformación XSLT apropiada y la aplica al XML generado a partir del modelo. Proporciona mucha flexibilidad, siendo fácil de generar vista para diferentes presentaciones o dispositivos, aunque representa un mayor grado de complejidad.

La segunda decisión es si se utiliza un patrón basado en una única etapa o en dos. Los patrones de una sola etapa normalmente tienen un componente de vista para cada pantalla de la aplicación. La vista recibe datos basados en el dominio y los muestra como HTML.

Una patrón vista en dos fases, **Two Step View**, subdivide el proceso en dos pasos, el primero produce una pantalla lógica a partir de los datos del dominio y luego, el segundo, la presentación de la pantalla lógica como HTML. Hay una primera etapa para cada pantalla, pero sólo una segunda etapa para toda la aplicación, tal y como se muestra en la **Fig. 2**.

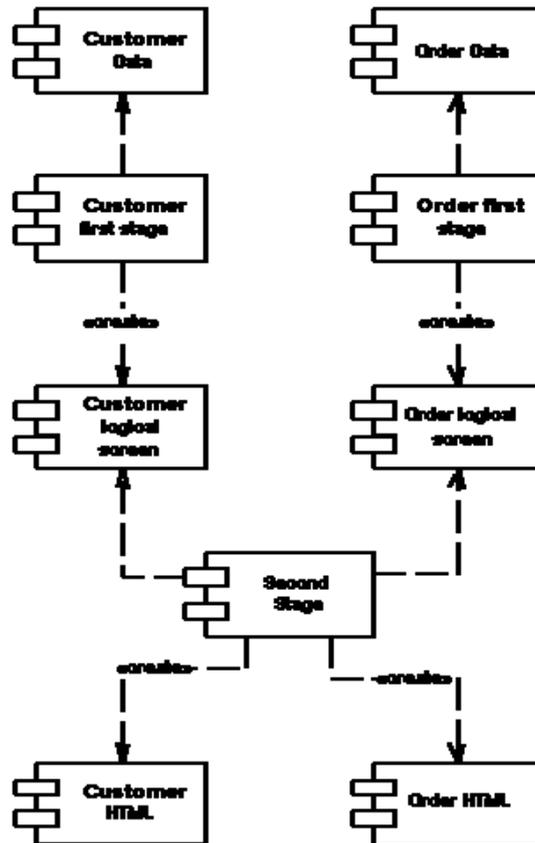


Fig. 2. Patrón vista en dos fases

La ventaja de **Two Step View** es que se pone la decisión de que HTML generar en un solo lugar, esto hace que los cambios globales al HTML sean más fáciles. Es apropiada para sitios donde diferentes pantallas comparten el mismo diseño básico.

4.2 Patrones Controlador de Entrada

Hay dos patrones para el controlador de entrada. El caso más común es tener un objeto controlador de entrada para cada página del sitio web, **Page Controller**. Normalmente habrá un **Page Controller** para cada acción: un botón o enlace.

Cada controlador de entrada tiene que hacer frente a dos responsabilidades: atender las solicitudes HTTP y decidir qué hacer con ellas. La aproximación del otro patrón controlador, **Front Controller**, es separar estas dos responsabilidades. Con el **Front Controller** hay un único objeto

que recibe todas las peticiones y crea un objeto separado para procesar la solicitud. Esto permite centralizar toda la gestión de las solicitudes HTTP en un único objeto.

5 Mapear a una Base de Datos Relacional

Cuando se utilizan objetos el problema es como interactuar con bases de datos relacionales, dos estructuras de datos bastante diferentes. Además, los objetos están en la memoria y es necesario coordinarlos con lo que está sucediendo en el disco, la base de datos.

Con una base de datos orientada a objetos no hay que preocuparse de mapeos. Esto mejora la productividad, ya que el mapeo supone alrededor de un tercio del esfuerzo de programación y mantenimiento. Sin embargo, la mayoría de los proyectos no utilizan bases de datos OO debido al riesgo de esta tecnología todavía no suficientemente madura en comparación a las BBDD relacionales.

5.1 Patrones Arquitectónicos

Los patrones arquitectónicos para el mapeo a base de datos relacional son principalmente **Active Record**, **Row Data Gateway**, **Table Data Gateway** y **Data Mapper**.

Row Data Gateway y **Table Data Gateway** se basan ambos en el patrón básico **Gateway**, una clase en memoria que mapea a una tabla en la base de datos y tiene un campo por columna de la base de datos. Los patrones basados en **Gateway** contienen todo el código de mapeo a la base de datos pero no código de la lógica de dominio.

La elección entre **Row Data Gateway** y **Table Data Gateway** depende de la lógica de dominio y también del soporte para el patrón básico **Record Set** que proporciona la plataforma utilizada para el desarrollo de la aplicación. Las plataformas de Microsoft, por ejemplo, utilizan **Record Set** ampliamente. Gran parte del funcionamiento de la interfaz de usuario se basa en manipular **Record Sets**. Por lo tanto, en este tipo de plataformas la opción natural es **Table Data Gateway**.

Por el contrario, **Row Data Gateway** es la mejor elección cuando los **Result Sets** son más difíciles de manipular porque no tienen tan buen soporte en la plataforma. En este caso lo que se obtiene es un objeto por cada registro en la fuente de datos.

Si la aplicación tiene un modelo de dominio que se parece mucho al modelo de la base de datos, entonces es mejor considerar **Active Record**, que combina el **Gateway** y el objeto de dominio en una sola clase que incluye tanto código de acceso a la BB.DD. como la lógica empresarial. Pese a todo, es recomendable mantener estas responsabilidades separadas.

La tercera opción es **Data Mapper**. Este es el patrón más complejo, pero también el más flexible. Se necesita una inversión de desarrollo pero permite que los objetos del dominio sean totalmente independientes de la base de datos. En cualquier caso, a medida que el mapeo se complica se hace recomendable no desarrollar el mapeo nosotros mismos y reutilizar herramientas existentes de mapeo Objeto-Relacional.

5.2 El Problema de la Concurrencia

Se trata del problema de cómo conseguir que los diversos objetos utilizados se carguen y salven en la base de datos. Por ejemplo, utilizando **Data Mapper**, los objetos del dominio no pueden hacer llamadas a la capa de mapeo ya que es totalmente transparente para ellos. Además, hay que seguir la pista de cada objeto modificado y asegurarse de escribir todos los cambios en la BB.DD.

Además, hay el problema de la concurrencia, asegurar que ningún otro proceso cambia cualquiera de los objetos que se han leído mientras se trabaja con ellos. Un patrón para resolver estos problemas es **Unit of Work**. Este patrón mantiene un registro de todos los objetos leídos desde la base de datos, junto con todos los modificados. También maneja cómo las actualizaciones se escriben en la base de datos.

Sin **Unit of Work**, la capa de dominio es que debe decidir cuándo leer y escribir en la base de datos. **Unit of Work** resulta de extraer el comportamiento de control del mapeo de datos a un objeto específico para esta funcionalidad.

5.3 Lectura de Datos

Es conveniente encapsular el SQL en las clases de mapeo objeto-relacional correspondientes. Los métodos de búsqueda encapsulan comandos SQL "SELECT" con una API, por ejemplo "buscar(id)" o "buscarCliente(idCliente)" encapsularían las consultas SQL correspondientes para recuperar un objeto con identificador id o un cliente con idCliente.

Es recomendable utilizar un patrón **Identity Map** para seguir la pista de lo que se ha leído para evitar hacerlo dos veces. Si se está utilizando **Unit of Work**, el lugar obvio para mantener el **Identity Map** es el objeto correspondiente al **Unit of Work**.

5.4 Patrones de Mapeo de Estructuras de Datos

Estos patrones describen cómo los datos en una base de datos relacional se mapean a datos en objetos. El problema surge de la diferente manera en que los objetos y las tablas manejan los enlaces. Los objetos manejan los enlaces almacenando referencias y las tablas mediante una clave apuntando a otra tabla.

El segundo tema es que los objetos pueden hacer fácilmente uso de colecciones para manejar múltiples referencias contenidas en un solo campo, mientras que la normalización fuerza que todos los campos de base de datos tengan un único valor. La solución es mantener la identidad relacional de cada objeto como un campo identidad, **Identity Field**, en el objeto, y utilizarla para mapear entre las referencias a objetos y las claves relacionales.

Cuando se leen objetos, cada vez que se encuentra una clave foránea, se mira en el **Identity Map** para establecer las referencias entre objetos. Si la clave no está en el **Identity Map**, entonces se va a la base de datos para cargarlo. Al guardar un objeto, se salva en la fila con la clave apropiada. Cualquier referencia entre objetos se sustituye por **Identity Field** del objeto referenciado.

Si un objeto tiene una colección, hay que lanzar otra consulta para encontrar todas las filas que enlazan con el identificador del objeto que tiene la colección. Salvar dicho objeto implica guardar cada objeto en la colección y asegurarse de que tiene una clave foránea al objeto de la colección.

5.5 Herencia

Las jerarquías de clases en OO vienen definidas por la relación de herencia. En las bases de datos relacionales no hay una forma estándar para representar la herencia, por lo que hay que realizar un mapeo al almacenarla en la base de datos. Existen tres opciones: una tabla para todas las clases de la jerarquía (**Single Table Inheritance**, ver Fig. 3), una tabla para cada clase concreta (**Concrete Table Inheritance**, ver Fig. 4) o una tabla por clase en la jerarquía (**Class Table Inheritance**, ver Fig. 5).

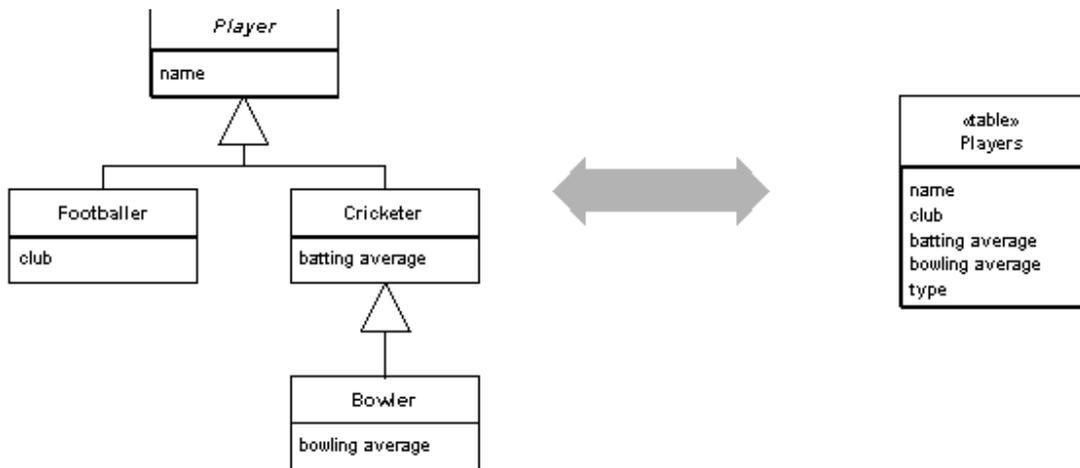


Fig. 3. Single Table Inheritance, uso de una tabla para almacenar todas las clases de una jerarquía

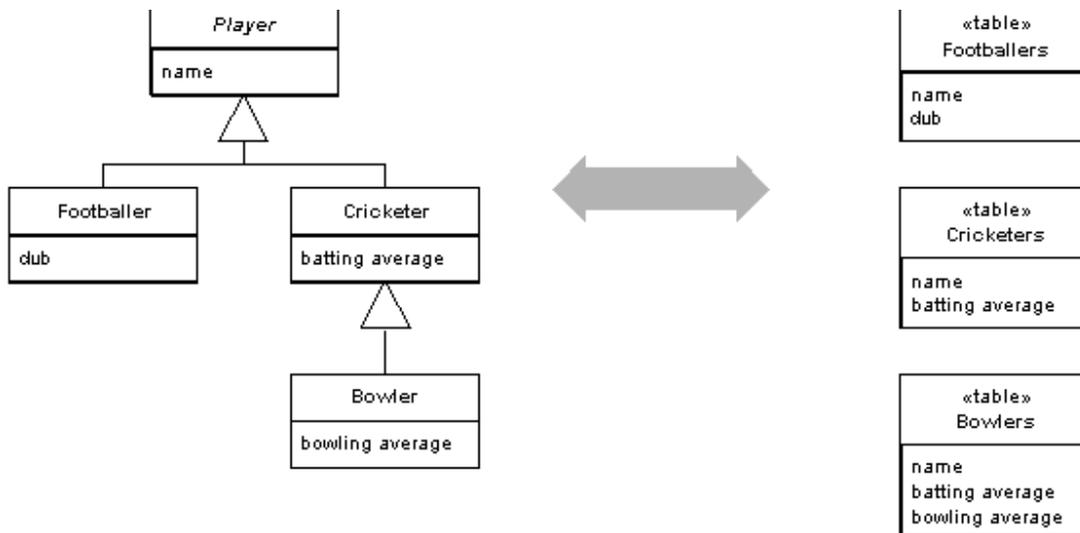


Fig. 4. Concrete Table Inheritance: uso de una tabla para cada clase concreta de la jerarquía

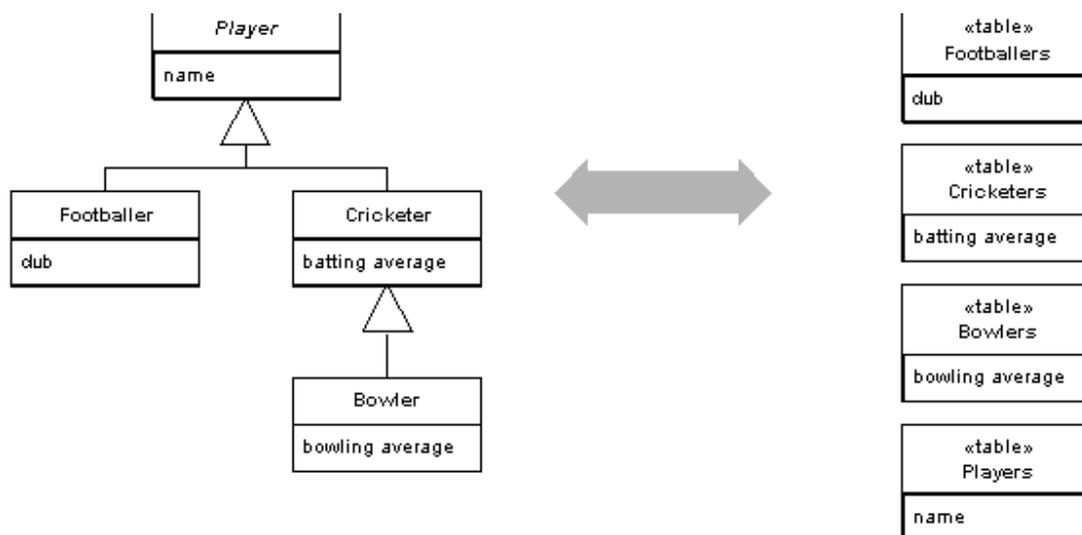


Fig. 5. Class Table Inheritance: uso de una tabla para cada clase de la jerarquía

El compromiso para elegir entre las tres opciones es entre la duplicación de la estructura de datos y la velocidad de acceso. **Class Table Inheritance** es la relación más simple entre clases y tablas, pero necesita múltiples JOINS para cargar un solo objeto, lo que reduce el rendimiento.

Concrete Table Inheritance evita las JOINS, permitiendo recuperar un objeto desde una única tabla. Pero **Concrete Table Inheritance** es frágil ante los cambios. Finalmente, el inconveniente más grande de **Single Table Inheritance** es el gasto de espacio, ya que en cada fila debe haber columnas para cada posible subtipo, aunque hace la modificación más fácil y evita las JOINS.

5.6 Mapeo de Metadatos

El enfoque más sofisticado de mapeo es utilizando **Metadata Mapping**, que se basa en escribir los detalles del mapeo en un archivo de metadatos, datos sobre los datos. Los detalles del mapeo son cómo las columnas de la base de datos se corresponden con los campos en los objetos. Los metadatos permiten tener que escribir código repetitivo utilizando técnicas de generación de código a partir de los metadatos o mediante programación reflexiva (el código puede modificarse a si mismo).

Una línea de metadatos puede decir algo así como:

```
<field name="customer" targetClass="Customer" dbColumn="custID" targetTable="customers"
lowerBound="1" upperBound="1" setter="loadCustomer"/>
```

A partir de esta línea de metadatos se puede generar el código de lectura, el de escritura, la generación automática de JOINS, el SQL, forzar la multiplicidad de la relación,... Como puede verse, se trata de una solución flexible y potente. De hecho, muchas herramientas comerciales de mapeo objeto-relacional suelen emplear metadatos.

6 Concurrencia

Siempre que tengamos múltiples procesos o hilos manipulando los mismos datos, existe el riesgo de problemas de concurrencia. Las transacciones proporcionan un marco que ayuda a evitar muchos de los aspectos más complicados de concurrencia. Siempre y cuando se haga toda la manipulación de datos dentro de una transacción.

Desafortunadamente, hay muchas interacciones con un sistema que no se pueden mantener dentro de una sola transacción de base de datos. En estos casos hay que aplicar concurrencia offline, es decir control de la concurrencia para datos manipulados durante múltiples transacciones de base de datos. Además, hay la concurrencia del servidor de aplicaciones: dar soporte a múltiples hilos de ejecución en un servidor de aplicaciones.

6.1 Problemas de Concurrencia

Los principales problemas que los sistemas de control de la concurrencia intentar evitar son:

- Actualizaciones perdidas: el usuario A edita un archivo para hacer algunos cambios. Mientras, el usuario B realiza un pequeño cambio en el mismo archivo de manera que termina antes que A. Desafortunadamente, cuando A leyó el archivo no incluía los cambios de B, así que cuando A acaba y guarda sus cambios, sobrescribe la versión de B de forma que su cambio se pierde.
- Lecturas inconsistentes: ocurren cuando se leen dos datos que eran correctos por separado, pero no eran correctos en el mismo momento. El usuario A quiere saber cuántos archivos hay en una carpeta que contiene dos subcarpetas. A mira en la primera subcarpeta y ve 7 archivos. Antes de mirar en la otra subcarpeta, el usuario B añade 2 ficheros en la primera subcarpeta y 3 a los 5 que había en la segunda. A mira ahora en la segunda subcarpeta y cuenta 8 archivos. Para él el total de archivos de la carpeta es 15. Lamentablemente 15 no ha sido en ningún momento la respuesta correcta. El recuento correcto era 12 antes de los cambios del usuario B y 17 después. Cualquiera de las dos respuestas hubiese sido correcta, incluso si no era el valor en ese momento.

Para evitar estos problemas se utilizan mecanismos de control de la concurrencia.

6.2 Contextos de Ejecución

Desde el punto de vista de la interacción de la aplicación con el exterior, los contextos de ejecución más importantes son la solicitud (request) y la sesión (session). La solicitud corresponde a una llamada desde el exterior que se procesa en la aplicación, que opcionalmente envía de vuelta una respuesta. Durante una solicitud el procesamiento se produce en gran medida en los servidores y se supone que el cliente espera la respuesta.

Una sesión es una interacción entre un cliente y un servidor a lo largo de un periodo de tiempo. Una sesión puede constar de una sola solicitud, pero comúnmente consta de una serie de peticiones, una serie que el usuario considera como una secuencia lógica coherente. Normalmente comenzará con la autenticación del usuario (log in) y terminará cuando el usuario sale de su cuenta (log out).

Otro contexto de ejecución, éste ligado al procesamiento y no a la interacción, es el proceso. Un proceso es un contexto de ejecución denominado pesado ya que proporciona aislamiento para los datos que se procesan internamente. Por el contrario, los hilos (threads) son contextos de

ejecución más ligeros que los procesos, de hecho un proceso puede dar soporte a múltiples hilos. Los hilos permiten soportar múltiples flujos de ejecución desde un único proceso, pero los hilos no proporcionan aislamiento para los datos que procesan y comparten el mismo espacio de memoria, lo que puede provocar problemas de concurrencia.

Finalmente, otro concepto relacionado con los contextos de ejecución, cuando se trata con bases de datos, son las transacciones, que permiten agrupar varias solicitudes del cliente y tratarlas como si todo el procesamiento se hiciese en una única solicitud, evitando así muchos problemas de concurrencia.

6.3 Aislamiento

Los problemas de concurrencia tienen su origen en que más de un agente activo, proceso o hilo, tiene acceso al mismo dato. El aislamiento es una forma de evitar estos problemas: repartir el acceso a los datos de manera que cualquier dato sólo pueda ser accedido por un agente activo. Así es como los procesos trabajan en la memoria del sistema operativo. De forma similar existen los bloqueos de archivos que evitan que más de un usuario abra simultáneamente un mismo archivo para modificarlo.

6.4 Control de la Concurrencia Optimista y Pesimista

Cuando tenemos datos que pueden ser modificados pero no es posible aislarlos, ya que son compartidos, es necesario aplicar medidas de control de la concurrencia. Existen dos opciones: las medidas optimistas y las pesimistas.

Partamos del escenario en que dos usuarios quieren editar el mismo archivo al mismo tiempo. Con un bloqueo optimista, ambos pueden hacer una copia del archivo y editarlo libremente. El control de concurrencia entra en juego cuando el segundo usuario intenta guardar sus cambios. Se detecta el conflicto entre ambos cambios, se le avisa al segundo usuario que no es posible guardar sus cambios y es entonces responsabilidad del segundo usuario ver cómo puede evitar perder sus cambios. Con una aproximación pesimista, quien primero bloquea el archivo para poder modificarlo impide que nadie más pueda editarlo.

El bloqueo optimista trata de detectar el conflicto, mientras que el pesimista bloquea los recursos para prevenirlo. Si los conflictos son suficientemente raros, o las consecuencias no son críticas, se recomienda utilizar el optimista ya que permite un mayor grado de concurrencia. Es decir, es posible que más usuarios puedan trabajar en paralelo sin tener que esperar a que los recursos se vayan liberando, consiguiéndose por lo tanto un mayor rendimiento.

6.5 Prevención de Lecturas Inconsistentes

Se pueden evitar las lecturas inconsistentes mediante bloqueos pesimistas. Para leer un dato se necesita un bloqueo de lectura (compartido) y para escribirlo se necesita un bloqueo de escritura (exclusivo). Muchos usuarios pueden tener el bloqueo de lectura a la vez ya que es compartido, pero si alguien tiene el de lectura nadie puede escribir. Por otro lado, una vez que alguien tiene el bloqueo de escritura, entonces nadie puede leer ni escribir.

6.6 Punto Muerto

Un problema de las técnicas pesimistas es llegar a un punto muerto que impida seguir el procesamiento. Si dos usuarios están editando diferentes archivos, por lo que están bloqueados, y para completar sus tareas necesitan ambos editar el archivo que está siendo bloqueado por el otro usuario, entonces han llegado a un punto muerto desde el que no pueden seguir trabajando.

Varias técnicas intentan evitar los puntos muertos. Una es tener un software que detecte los puntos muertos y seleccione a una víctima que deberá entonces liberar todos los recursos que tiene bloqueados. Una aproximación similar es dar una caducidad a cada bloqueo, aunque lamentablemente entra en juego incluso cuando no se ha llegado a ningún punto muerto todavía.

Otra forma de evitar puntos muertos es que los usuarios adquieran todos sus bloqueos a la vez al comienzo de su trabajo, y una vez que los tienen impedir que puedan bloquear más recursos. Los bloqueos se consiguen siguiendo un orden común para todos los usuarios, por ejemplo para bloquear archivos se sigue un orden alfabético según sus nombres.

6.7 Transacciones

La principal herramienta para el control de la concurrencia en aplicaciones empresariales es la transacción. Una transacción es una secuencia de acciones delimitada de manera clara por un punto inicial y uno final. Todos los recursos implicados están en un estado consistente tanto cuando la transacción comienza y como cuando termina. Además, cada transacción debe completarse totalmente, todas las acciones realizadas, o ninguna acción debe tener efecto. Es decir, o todo o nada.

Las propiedades de una transacción son:

- **Atomicidad:** cada paso en la secuencia de acciones de la transacción debe completarse satisfactoriamente o todo lo realizado hasta el momento debe deshacerse.
- **Consistencia:** los recursos del sistema deben estar en un estado consistente tanto al principio como al final de la transacción.
- **Aislamiento:** el resultado de una transacción no debe ser visible para el resto de transacciones hasta que no se haya completado con éxito.
- **Persistencia:** el resultado de una transacción exitosa debe ser permanente.

Los recursos gobernados por las transacciones suelen ser bases de datos, el caso más común, aunque también pueden ser colas de mensajes, impresoras, etc.

Para conseguir el mayor rendimiento, las transacciones deben ser tan cortas como sea posible. Se recomienda que las transacciones no abarquen varias solicitudes. Normalmente se inicia la transacción al comienzo de la solicitud y se finaliza al final de la solicitud.

6.8 Transacciones de Negocio y de Sistema

Las transacciones de sistema están soportadas por bases de datos y monitores de transacciones. Se componen de un grupo de comandos SQL delimitados por instrucciones para abrir y cerrar la transacción.

Para el usuario de un sistema de banca en línea una transacción se compone de los pasos: seleccionar una cuenta, establecer algunos pagos de facturas y finalmente hacer click en el botón "Aceptar" para realizar los cargos en las cuentas. Esto es lo que llamamos una transacción de negocio.

La manera obvia para hacer que una transacción de negocio tenga las mismas propiedades que una de sistema (atomicidad, consistencia, aislamiento y persistencia) es ejecutarla toda dentro de una transacción de sistema. El problema es que las transacciones de negocio suelen abarcar varias

solicitudes. Así que, si se quiere tener una aplicación con capacidad de respuesta, es necesario romper la transacción de negocio en una serie de transacciones de sistema cortas. Si no se hiciera así, como puede pasar tiempo entre solicitud y solicitud, la aplicación podría llegar a quedar bloqueada debido a la existencia de muchos recursos bloqueados por diferentes transacciones de negocio.

Atomicidad y persistencia son las propiedades más fáciles de conseguir en las transacciones de negocio. Ambas se basan en aplicar los cambios de la transacción, cuando el usuario pulsa "Aceptar", dentro de una transacción de sistema. El problema en este caso es mantener un registro detallado de los cambios realizados durante la transacción de negocio. Si la aplicación utiliza un **Domain Model** se puede utilizar **Unit of Work** para monitorizar los cambios.

El problema es conseguir el aislamiento entre transacciones de negocio. Los fallos en el aislamiento conducen a problemas de consistencia. Dentro de una transacción la responsabilidad de la aplicación para mantener la consistencia es obligar el cumplimiento de todas las reglas de negocio. Cuando entran en juego múltiples transacciones, la responsabilidad es asegurar que una sesión no sobrescriba los cambios realizados por otras sesiones.

6.9 Patrones para el Control de la Concurrency Offline

Estos patrones son algunas técnicas que pueden resultar útiles para tratar con la concurrencia que abarca más que una transacción de sistema. Son sólo necesarias si no se pueden enmarcar todas las transacciones de negocio dentro de una sola solicitud, por lo tanto tampoco dentro de una transacción de sistema.

Optimistic Offline Lock es un patrón que usa control de concurrencia optimista en las transacciones de negocio. Es fácil de programar y proporciona una buena capacidad de respuesta. Su limitación es que sólo se descubre que una transacción de negocio va a fallar cuando se intentan acometer la transacción así que todo el trabajo realizado puede perderse. La alternativa es **Pessimistic Offline Lock**, ya que no se llega a la situación anterior aunque es más difícil de programar y reduce la capacidad de respuesta de la aplicación.

Con cualquiera de las aproximaciones previas, se puede reducir enormemente la complejidad de desarrollo si no se intenta gestionar los bloqueos a nivel de cada objeto individual. Es mejor operar a un nivel de detalle menor y gestionar la concurrencia para grupos de objetos.

6.10 Concurrency a Nivel de Servidor de Aplicaciones

Hasta ahora se ha hablado de concurrencia de múltiples sesiones operando sobre una fuente de datos compartida. Otra forma de concurrencia es a nivel del servidor de aplicaciones. Cómo maneja dicho servidor múltiples solicitudes simultáneas. La mayor diferencia es que la concurrencia en el servidor de aplicaciones no involucra transacciones.

La forma más sencilla de manejar este caso es usar un proceso por sesión, cada sesión se ejecuta en su propio proceso. La gran ventaja es que el estado de cada proceso está totalmente aislado de los otros procesos. El problema es que se utiliza muchos recursos de servidor, cada proceso supone un gran consumo de memoria.

Para ser más eficiente se puede utilizar un fondo común de procesos (process pool). Se trata de un conjunto de procesos fijo que se van turnando para ir atendiendo las solicitudes que van llegando al servidor de aplicaciones. De esta manera cada proceso se ocupa de una solicitud cada vez, pero puede manejar múltiples solicitudes de sesiones diferentes en secuencia. El principal problema de

esta aproximación es que hay que asegurar que los recursos utilizados para atender una solicitud se liberen al final de la solicitud.

Se puede mejorar aún más el rendimiento si los procesos ejecutan múltiples hilos. Con el enfoque un hilo por solicitud, cada solicitud es tratada por un hilo dentro de un proceso. El problema es que no hay aislamiento entre los hilos. Por lo tanto, lo común en muchos servidores de aplicaciones es un enfoque de solicitud por proceso, normalmente con un fondo de procesos que se ajusta según las necesidades de rendimiento y los recursos hardware del servidor.

7 Estrategias de Distribución

El diseño dibuja objetos independientes para clientes, pedidos, productos, etc. Cada uno puede convertirse en un componente separado que puede ejecutarse en nodos de proceso diferentes. Se tiene así una aplicación de objetos distribuidos que se comunican de manera remota.

7.1 Interfaces Locales y Remotas

El anterior escenario, objetos distribuidos, es el grado máximo de distribución de una aplicación pero no ha tenido éxito por problemas de eficiencia. Una llamada entre dos procesos separados corriendo en máquinas diferentes es órdenes de magnitud más lenta que una llamada dentro de un proceso. Como resultado de esta gran diferencia, se hace necesaria un tipo de interfaz diferente para los objetos destinados a ser utilizados remotamente y para los utilizados localmente.

La interfaz local es la mejor para los métodos que permiten pequeños cambios. Por ejemplo, para una clase Dirección, en una interfaz local habría métodos separados para obtener la ciudad, establecer la ciudad, obtener la calle, etc.

Por el contrario, para llamadas remotas no son apropiados los métodos tan detallados, ya que es preferible minimizar el número de llamadas remotas que son mucho más lentas. Son preferibles los métodos que realizan más funcionalidad, por ejemplo obtener o actualizar la ciudad, calle y número en una única llamada en lugar de con tres llamadas. El problema de este tipo de interfaces es la poca flexibilidad que ofrecen este tipo de métodos, muy sensibles a cualquier cambio.

Debido a la gran diferencia de rendimiento, la principal recomendación respecto al diseño de objetos distribuidos, es evitar siempre que sea posible tener que distribuirlos.

Entonces, ¿cómo utilizar eficazmente múltiples procesadores? En la mayoría de los casos la recomendación es recurrir al clustering, replicar el servidor de aplicaciones en múltiples nodos. Es decir, poner todas las clases en un único proceso y luego ejecutar múltiples copias de ese proceso sobre los diversos nodos. Así cada proceso utiliza las llamadas locales a hacer el trabajo y así lo hace más rápido, pero existen múltiples copias del proceso para atender las solicitudes que van llegando.

7.2 Escenarios de Distribución

Lo primero es intentar minimizar las llamadas remotas, pero no siempre es posible eliminarlas completamente. Una separación obvia que causa llamadas remotas es la existente entre cliente y servidor. Otra separación típica es entre el servidor de aplicaciones y la base de datos, ya que no suele ser práctico ejecutar ambos en el mismo servidor por problemas de rendimiento. Esto último no es un problema porque SQL está diseñado como una interfaz remota.

Puede ocurrir también una separación entre el servidor web y el de aplicaciones. Es mejor ejecutarlos en el mismo proceso, pero puede ser necesario separarlos por problemas de incompatibilidad. Al menos, un buen paquete de software de este tipo debería proporcionar una interfaz apropiada para las llamadas remotas. Si existen otras razones para distribuir la aplicación, la recomendación general es dividirla en componentes con interfaces remotas que permitan minimizar el número de llamadas remotas.

7.3 Recomendaciones de Desarrollo de Objetos Distribuidos

La recomendación es utilizar interfaces con métodos detallados internamente, de manera local y objetos con interfaces con métodos más adecuados para llamadas remotas, que ofrezcan más funcionalidad por cada llamada, en los puntos en los que haya distribución. La función principal de estos últimos objetos será proporcionar una interfaz remota para los objetos internos. De esta forma actuarán como fachada remota, **Remote Facade**, mientras que se dispondrá de métodos más detallados, mucho más flexibles, para la comunicación a nivel interno.

Otro patrón recomendado es **Data Transfer Object**. Cuando las llamadas remotas envían objetos, normalmente no es recomendable enviar el objeto de dominio ya que este está vinculado a una interfaz local de gran detalle. Los objetos **Data Transfer Object** permiten representar los objetos a enviar de una manera independiente.

8 Puesta en Práctica

En esta sección se intentan dar pautas sobre que patrones utilizar al desarrollar una aplicación empresarial.

8.1 Capa Dominio

Lo primero es decidir que aproximación seguir con la capa de lógica de dominio. Las tres opciones principales son: **Transaction Script**, **Table Module** y **Domain Model**.

La más simple es **Transaction Script**, que se ajusta al modelo procedural. La lógica de cada transacción de sistema se encapsula en un script. Su desarrollo no es complicado pero su gran defecto es que no es capaz de manejar lógica de negocio compleja, siendo particularmente susceptible a la duplicación de código. Es apropiado sólo para aplicaciones simples, como una tienda online con un catálogo sencillo y carro de la compra.

En el otro extremo de está **Domain Model**. Ideal para lógicas de dominio complejas, una vez que uno se acostumbra a trabajar con **Domain Model** incluso casos más simples se pueden abordar con facilidad. La gran dificultad es la conexión del modelo a una base de datos relacional.

Table Module representa el término medio entre las dos opciones anteriores. No puede manejar lógica de dominio demasiado compleja pero encaja muy bien con el modelo relacional de bases de datos. Para entornos como .NET que disponen de **Record Set**, esta alternativa se ajusta muy bien.

8.2 Capa de Datos

Una vez que se ha seleccionado la aproximación más apropiada para la capa de dominio, es el momento de ver como conectarla con las fuentes de datos. La elección anterior delimita las opciones así que se presentaran para cada caso:

- **Transaction Script**: separar la base de datos utilizando los patrones **Row Data Gateway** y **Table Data Gateway**. Si la plataforma proporciona soporte para **Record Set**, la opción

clara es **Table Data Gateway**. En cualquier caso, no es necesario preocuparse de las cuestiones estructurales del mapeo ya que las estructuras de datos en memoria de estas opciones se encargan del mapeo a la base de datos. Se recomienda **Unit of Work**, pero normalmente es fácil seguir la pista de lo que se ha cambiado en el script. Normalmente no es necesario preocuparse de temas de concurrencia porque el script a menudo corresponde casi exactamente con una transacción de sistema. Una excepción es cuando una solicitud recupera datos para editarlos y la siguiente solicitud se encarga de salvar los cambios. En este caso **Optimistic Offline Lock** es casi siempre la mejor opción.

- **Table Module**: es la mejor opción para las plataformas con soporte para **Record Set**. En este caso la opción de mapeo a la base de datos es **Table Data Gateway**. En muchos casos la implementación de **Record Set** tiene algún mecanismo de control de la concurrencia incorporado, lo que de hecho lo convierte en una **Unit of Work**.
- **Domain Model**: el gran inconveniente de esta opción es que la conexión con la base de datos es complicada. Si el **Domain Model** es bastante simple, un par de docenas clases similares a la estructura de la base de datos, entonces un **Active Record** tiene sentido. Si se quiere desligar las cosas un poco se puede utilizar **Table Data Gateway** o **Row Data Gateway**. A medida que las cosas se complican, será necesario considerar **Data Mapper**, que mantiene **Domain Model** independiente de todas las otras capas. **Data Mapper** es también la alternativa más complicada por lo que la recomendación es utilizar alguna herramienta existente⁴ para realizar el mapeo. Finalmente, también se recomienda utilizar **Unit of Work**, que permite focalizar y simplificar todo el esfuerzo de control de la concurrencia.

8.3 Capa Presentación

La primera decisión es si proporcionar una interfaz de cliente rico o una de navegador HTML. Un cliente rico proporciona una interfaz de usuario más agradable, pero conlleva más problemas a la hora del despliegue a los clientes y el mantenimiento. Por lo tanto, lo mejor es utilizar una interfaz de navegador HTML si la funcionalidad que esta tecnología proporciona es suficiente, y un cliente rico si no. En cualquier caso, los recientes avances en la Web 2.0 a la hora de proporcionar interfaces HTML con mucho mayor nivel de interactividad, hace cada vez más atractivas las interfaces HTML.

Para la interfaz HTML la recomendación es utilizar un patrón **Model View Controller**. El modelo será el ya definido en la capa correspondiente, habrá entonces que definir el controlador y la vista, los otros dos componentes de **Model View Controller**.

Las herramientas de desarrollo pueden forzar estas elecciones. Por ejemplo, con Visual Studio, lo más fácil es utilizar **Page Controller** y **Template View**. Con Java existen muchos paquetes de desarrollo donde elegir. Una opción popular es Struts, que supone **Front Controller** y **Template View**.

Si existe libertad de elección, la recomendación es **Page Controller** para sitios orientados a documentos, particularmente si hay una mezcla de páginas estáticas y dinámicas. Interfaces más complejas hacen más recomendable **Front Controller**.

⁴ http://en.wikipedia.org/wiki/List_of_object-relational_mapping_software

Respecto a la vista, la elección es entre **Template View** y **Transform View**. Realmente depende de si se prefiere utilizar páginas de servidor o XSLT durante el desarrollo. Si el objetivo es un sitio con múltiples estilos de vista, la recomendación es **Two Step View**.

Finalmente, si es posible intenta ejecutarlo todo en un proceso, sin llamadas remotas. Si no es posible, encapsula la capa de dominio con un **Remote Facade** y utiliza **Data Transfer Object** para la comunicación con el servidor web.

8.4 Recomendaciones para Tecnologías Específicas

Java y .NET son las plataformas más utilizadas para el desarrollo de aplicaciones empresariales. A continuación se proporcionan recomendaciones para ambas tecnologías.

8.4.1 Java y J2EE

No es necesario recurrir a Enterprise Java Beans (EJBs) para desarrollar una aplicación J2EE. Se puede recurrir a los denominados POJOs (Plain Old Java Objects), simples objetos Java, y JDBC para el acceso a la base de datos.

Si la lógica de dominio es bastante simple y solo necesita **Transaction Scripts**, se pueden utilizar Session Beans como **Transaction Scripts** y Entity Beans como **Row Data Gateways**. La aproximación alternativa a EJBs, basada en POJOs, utiliza objetos Java para **Transaction Scripts** sobre **Row Data Gateway** o **Table Data Gateway**. También se puede utilizar los Row Sets de JDBC 2.0 como **Record Sets** y **Table Data Gateway**.

Si se tiene **Domain Model**, entonces lo recomendable es utilizar Entity Beans como los componentes del modelo de dominio. Si el dominio es bastante simple y coincide bastante con la estructura de la base de datos, entonces los Entity Beans pueden actuar de **Active Records**. Es una buena práctica encapsular los Entity Beans con Session Beans que actúen como **Remote Facades**.

La otra alternativa es no utilizar EJBs y recurrir a POJOs. En este último caso es más fácil ejecutar toda la aplicación en el servidor web y evitar cualquier llamada remota entre presentación y dominio. En el caso de los EJBs puede ser necesario disponer de un servidor EJBs además del servidor web.

8.4.2 .NET

El patrón dominante en .NET es **Table Module**, para el que esta plataforma proporciona abundantes herramientas que aprovechan **Record Set**. No es recomendable por lo tanto recurrir a **Transaction Script** excepto en los casos más sencillos. También es posible **Domain Model**, pero la plataforma no ofrece herramientas tan útiles como para **Table Module**. Por lo tanto, lo recomendable es recurrir a **Domain Model** sólo cuando la complejidad del modelo sea realmente alta.

Normalmente no es necesario separar el servidor web de la lógica de dominio en .NET por lo que raramente será necesario **Remote Facade**.

8.4.3 Servicios Web

Los servicios web se enfocan más a la integración de aplicaciones que al desarrollo de simples aplicaciones. No es recomendable romper una aplicación en servicios web a menos que sea estrictamente necesario ya que implica muchas llamadas remotas. Lo mejor es desarrollar la

aplicación y exponer las partes relevantes como servicios web, que realizarán la función de **Remote Facades**.

9 Referencias

La lista completa de patrones está disponible en el Campus Virtual:

<http://cv.udl.es/access/content/group/51002-0809/0%20-%20Patrons/Patterns%20of%20Enterprise%20Application%20Architecture-1>

Esta documentación se basa en:

Martin Fowler y David Rice:

Patterns of Enterprise Application Architecture.

Addison-Wesley, 2003. ISBN 0321127420

Referencias adicionales:

Harold, E. R.; Jeans, W. S. (2001). **XML in a Nutshell: A Desktop Quick Reference**. O. Reilly.

McLaughlin, B. (2001). **Java and XML (2nd edition)**. O. Reilly.

Cauldwell, P.; Charla, R.; Chopra, V. (2002). **Servicios Web y XML**. Anaya Multimedia.

McGovern, J.; Tyagi, S.; Stevens, M. E.; Mathew, S. (2003). **Java Web Services Architecture**.

Morgan Kaufmann.

Conallen, J. (1999) **Building Web Applications with UML**. Addison Wesley. ISBN: 0-201-61577-0

<http://safari.awprofessional.com/0201615770>

McLaughlin, Brett (2002) **Building Java Enterprise Applications Volume I: Architecture**. O'Reilly.

ISBN : 0-596-00123-1

<http://safari.awprofessional.com/0596001231>

Lam, H.; Thai, T.L. (2002) **.NET Framework Essentials, 2nd Edition**. Addison Wesley. ISBN: 0-596-00302-1

<http://safari.awprofessional.com/0596003021>

Lane, D.; Williams, H.E. (2004) **Web Database Application with PHP and MySQL, 2nd Edition**.

O'Reilly. ISBN: 0-596-00543-1

<http://safari.awprofessional.com/0596005431>