UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

# TESINA DE LICENCIATURA

**Título:** Manipulación de objetos persistentes en clientes de aplicaciones enterprise
**Autores:** Bottero, Sebastian Eduardo – García, Nicolás Javier
**Director:** Rossi, Gustavo
**Codirector:**
**Asesor profesional:** Butti, Matías Daniel
**Carrera:** Licenciatura en Sistemas (2003) – Licenciatura en informática (1990)

## Resumen

El desarrollo de este trabajo de grado tiene como objetivo lograr definir una forma general de atacar los inconvenientes que se suscitan al manipular objetos de dominio en el componente cliente de Aplicaciones Enterprise de gran tamaño, en una arquitectura Cliente-Servidor.

A lo largo del trabajo se lleva a cabo un análisis detallado del comportamiento de este tipo de aplicaciones, del cual se desprenden una serie de requerimientos no funcionales que son necesarios para su correcto funcionamiento. Al intentar satisfacer estos requerimientos, se generan un conjunto de problemas recurrentes, que de no estar bien resueltos por la plataforma de desarrollo y por las pautas definidas por la arquitectura, generan problemas de productividad y de calidad en el producto, y pueden hacer muy difícil el cumplimiento de estos requerimientos no funcionales.

Como aporte principal de este trabajo se plantea una solución a estos problemas, mediante la definición de un framework, el cual es fundamental en cualquier plataforma de desarrollo de este tipo de aplicaciones. Finalmente se enuncian diferentes maneras de proveer las configuraciones necesarias para la utilización del framework, y se especifica una de ellas mediante el uso de lenguajes específicos del domino, en este caso basado en XML.

## Palabras Claves

Aplicaciones Enterprise, Manejo de objetos persistentes en el componente Cliente, Arquitectura de Software, Plataforma de desarrollo, Arquitectura Cliente-Servidor, Sistemas de Información Enterprise, Objetos persistentes, Transferencia de Entidades, Concurrencia, Loqueo pesimista / optimista, Transacciones largas o de negocio, Requerimientos no Funcionales, Frameworks.

## Trabajos Realizados

- Estudio sobre Sistemas de Información Enterprise y procesos de negocios.
- Estudio de Arquitecturas de Software
- Análisis del comportamiento de Aplicaciones Enterprise
- Descripción de requerimientos no funcionales
- Propuestas de solución a problemas recurrentes
- Análisis de Plataformas de Desarrollo
- Frameworks. Análisis y Diseño
- Propuesta de lenguajes de configuración (Domain Specific Languages)

## Conclusiones

Se establecieron las características necesarias para hacer factible el desarrollo de Aplicaciones Enterprise de gran tamaño sobre una arquitectura Cliente Servidor, identificando y proveyendo una solución integral a los problemas recurrentes que se suscitan al manipular objetos de dominio en el componente cliente.
Este trabajo provee dichas soluciones mediante un framework, el cual es una parte fundamental de la arquitectura de este tipo de aplicaciones y de la plataforma de desarrollo utilizada, ayudando así a cumplir con el objetivo de facilitar y estandarizar el desarrollo.

## Trabajos Futuros

Uno de los principales trabajos a futuro es la implementación integral del Framework basada en el diseño propuesto.
Además es necesaria la definición de un lenguaje utilizado para la representación de la meta información requerida para configurar el comportamiento del framework. Este tipo de lenguaje puede estar basado en el concepto de “Lenguaje Específico del Dominio” o DSL.
Teniendo en cuenta la variedad de plataformas disponibles para la implementación de clientes ricos, un muy interesante trabajo a futuro sería definir una Arquitectura dirigida por modelos o MDA, con sus respectivos modelos y transformaciones, extendiendo de esta manera su uso.

**Fecha de la presentación: Agosto de 2011**

# Manipulación de objetos persistentes en clientes de aplicaciones enterprise

Sebastián Eduardo Bottero

Nicolás Javier García

Tesis de grado

*Facultad de Informática*
*Universidad Nacional de La Plata*
*Argentina*

Director: Dr. Gustavo Rossi

**CAPÍTULO 1 - INTRODUCCIÓN** **1**

DEFINICIÓN DEL PROBLEMA 1

*Sistemas Enterprise* *2*

*Arquitectura de Sistemas Enterprise* *2*

CONTRIBUCIONES 3

ESTRUCTURA DEL DOCUMENTO 4

**CAPÍTULO 2 - APLICACIONES ENTERPRISE DE GRAN TAMAÑO** **6**

DESCRIPCIÓN 6

SISTEMAS DE INFORMACIÓN ENTERPRISE 7

PROCESOS DE NEGOCIOS 8

*Definiciones* *8*

*Características de los procesos de negocios* *10*

REQUERIMIENTOS FUNCIONALES 10

REQUERIMIENTOS NO FUNCIONALES 12

**CAPÍTULO 3 - ARQUITECTURA DE SOFTWARE** **20**

EVOLUCIÓN DE LA INFRAESTRUCTURA DE PROCESAMIENTO DE INFORMACIÓN 20

UN POCO DE HISTORIA 20

¿QUÉ ES LA ARQUITECTURA DE SOFTWARE? 23

ESTILOS DE ARQUITECTURA 24

*Pipes and filters* *25*

*Publisher Subscriber (invocación implícita)* *25*

*Blackboard (pizarrón)* *26*

*Layered (en capas)* *28*

Arquitecturas que no necesitan layers 29

Dos layers 30

Tres layers 31

N layers 32

ARQUITECTURA CLIENTE-SERVIDOR 32

*Cliente* *33*

Tipos de Clientes 33

*Servidor* *35*

*Ventajas y Desventajas* *35*

Ventajas 35

Desventajas 37

**CAPÍTULO 4 - TRANSFERENCIA DEL MODELO DE ENTIDADES** **38**

Modelo de Dominio 38
Distribución del Modelo de Dominio 39
Escenarios para la distribución de Objetos 41
*Interfaces locales y remotas 41*
*Invocaciones interprocesos 43*
¿Dónde hay que distribuir? 44
*Aplicaciones con bases de datos 44*
*Arquitecturas Cliente Servidor 44*
*Servidores Web y de Aplicaciones 45*
*Límites explícitos de distribución 45*
Alternativas Conceptuales 45
*Objetos de transferencia - Data Transfer Object 45*
*Objetos de dominio desconectados - Detach Object 47*
*DTO's vs Detach Object 48*
*DTO dinámicos 49*
**CAPÍTULO 5 - ANÁLISIS DEL PROBLEMA 50**
Concurrencia 50
*Concurrencia en memoria 50*
*Manejo de Concurrencia 51*
*Problemas esenciales de la concurrencia 51*
*Aislamiento e inmutabilidad 53*
*Estrategia de loqueo Pesimista y Optimista 54*
*Transacciones 56*
*Patrones para control de concurrencia off line 57*
Transacciones largas 58
*Transacciones de sistema y de negocio 59*
Objetos de gran tamaño y carga por demanda 61
Deshacer operaciones - UNDO 62
*Beneficios del Undo 63*
*Undo en aplicaciones Web 63*
*El patrón Undo 64*
Otras características de los objetos persistentes en el cliente 65
**CAPÍTULO 6 - SOLUCIÓN 66**
Plataforma de desarrollo 66
Framework propuesto 67
*Objetivos 67*

*Características del framework* *69*
ARQUITECTURA DEL FRAMEWORK 70
DIVISIÓN EN CAPAS 70
*Capa de servicios CRUD* *71*
*Capa de DyTOs* *73*
Principales componentes 75
*Capa de State Transfer Objects* *76*
Principales componentes 77
*Integración entre capas* *80*
META-INFORMACIÓN 83
*Tipos de meta-información necesaria* *83*
*Implementación propuesta* *84*
TRANSACCIONES LARGAS 85
*Unit of Work* *85*
Tipos de comandos 86
DESHACER CAMBIOS (UNDO) 88
MANEJO DE CONCURRENCIA 88
*Lockeo optimista* *89*
REFERENCIAS LAZY Y CARGA POR DEMANDA 90
**CAPÍTULO 7 - CONCLUSIÓN Y TRABAJO FUTURO 92**
CONCLUSIONES 92
TRABAJO FUTURO 93
*Lenguaje específico para meta información (DSL)* *93*
Interpretaciones del DSL externo 94
*Arquitectura dirigida por modelos (MDA)* *94*
*Implementación integral del Framework* *95*
**ANEXO A - DSL PARA ESPECIFICAR LA CONFIGURACIÓN DE UNA ENTIDAD, RESPECTO A CÓMO DEBE SER TRASLADADA Y TRATADA EN EL CLIENTE 97**
DESCRIPCIÓN 97
*Esquema XML* *97*
*XML de Ejemplo* *99*
ELEMENTOS 100
**BIBLIOGRAFÍA 102**
RECURSOS WEB 104

# *Agradecimientos*

*Queremos empezar agradeciendo a nuestros padres, quienes nos apoyaron siempre y nos inculcaron los valores que guían nuestro transitar por la vida. Porque se esforzaron en todo momento para que nada nos falte, y porque les debemos lo que somos y lo que logramos, a ellos está dedicado este trabajo de grado.*

*A nuestras respectivas novias por su compañía y comprensión durante los años que le dedicamos a esta Tesis.*

*A nuestros abuelos, hermanos y amigos porque son una parte importante en nuestra vida. Sabemos que contamos con ellos siempre.*

*A Gustavo por dirigir este trabajo y porque la pasión que transmite al dictar sus materias nos sirvió de guía en los primeros años donde lo que prima es la incertidumbre.*

*Para terminar, queremos agradecer muy especialmente a Matías, quien nos ayudó inmensamente en este trabajo de grado. Sus consejos y motivación constante por hacer las cosas bien, fue el faro que permitió que nuestra Tesis llegara a buen puerto.*

*Muchas gracias a todos,*

*Nicolás y Sebastian*

# *Capítulo 1*

# Introducción

En la mayoría de los procesos de desarrollo de Aplicaciones Enterprise nos encontramos con requerimientos no funcionales, que pueden entorpecer el proceso de desarrollo y afectar la calidad de la aplicación. Satisfacerlos una vez comenzado el desarrollo puede volverse una tarea muy complicada o por lo menos tediosa, por lo que propondremos una serie de soluciones a éste y otros problemas recurrentes en el proceso de desarrollo de este tipo de aplicaciones.

## Definición del Problema

Muchas personas intentan definir el término "arquitectura", pero pocas se ponen de acuerdo al respecto. Aún así, existen dos elementos comunes a todas las definiciones; uno es la descomposición de alto nivel que divide al sistema en partes; el otro, las decisiones que son difíciles de cambiar.

Cada vez es más común concluir que no hay sólo un camino para definir una arquitectura, si no que existen múltiples arquitecturas dentro de un sistema, y la visión de qué es arquitecturalmente significativo puede cambiar a lo largo del tiempo.

Podemos ver a la arquitectura de una aplicación como la estructura del sistema en términos de elementos y relaciones entre dichos elementos. La definición de la arquitectura también abarca la especificación de la tecnología adecuada para cada parte del sistema; existiendo estilos arquitectónicos y patrones arquitecturales que permiten reusar soluciones exitosas.

Una vez que se define la arquitectura de aplicación, es necesario definir pautas que estandaricen el proceso de desarrollo sobre ella, en busca de mejorar la productividad, mejorar la calidad del producto y simplificar su futuro mantenimiento. En este punto es recomendable implementar una “Plataforma de desarrollo” guiada por la arquitectura definida, los requerimientos no funcionales y los requerimientos funcionales representativos. La implementación de la plataforma de desarrollo mencionada se logra, cuando sea conveniente, integrando frameworks[1] existentes con desarrollos propios.

Este trabajo está motivado por la necesidad de evaluar un problema recurrente en Aplicaciones Enterprise que, de no estar bien resuelto por la

[1] Utilizamos el término en inglés, debido a la aceptación del mismo en el lenguaje académico.

plataforma de desarrollo y por las pautas definidas por la arquitectura, genera problemas de productividad y de calidad en el producto, y puede hacer muy difícil el cumplimiento de los requerimientos no funcionales.

## Sistemas Enterprise

Los sistemas de información enterprise proveen una plataforma tecnológica que permite a una organización integrar y coordinar sus procesos de negocios, permitiendo que la información sea compartida en tiempo real por todos los niveles de la jerarquía de administración de la empresa.

Las operaciones de este tipo de sistemas trabajan sobre objetos que modelan conceptos del dominio cuyo volumen de información es elevado y por ende son naturalmente de gran tamaño. A su vez deben ser persistentes en el tiempo y accedidos de manera concurrente, posiblemente por cientos de usuarios. Es usual también que existan gran cantidad de interfaces de usuario para manejarlos.

## Arquitectura de Sistemas Enterprise

Hay una tendencia creciente a implementar los Sistemas Enterprise sobre arquitecturas del tipo Cliente-Servidor. Esta tendencia se debe en parte a la disponibilidad y distribución con la que deben contar las aplicaciones de este tipo de sistemas y se acentúa debido a la proliferación en los últimos años de los sistemas distribuidos y particularmente de los sistemas Web.

El problema recurrente en este tipo de aplicaciones que motivó este trabajo es la Manipulación de Objetos Persistentes en la componente “Cliente” de este tipo de arquitecturas.

Para atacar este problema, se deben considerar todos los aspectos y requerimientos no funcionales asociados que se ven potenciados en las aplicaciones enterprise. Entre ellos podemos enunciar:

- La información está representada mediante grafos de objetos con gran profundidad, y debe ser accedida por el usuario con un razonable tiempo de respuesta.
- El usuario tiene una excesiva interacción con el sistema (cliente) para llevar a cabo una operación. Los datos generados durante esa interacción deben ser sincronizados con el servidor.
- Asegurar la integridad de la entidad en accesos concurrentes desde varios clientes
- Evitar la duplicación de código de los objetos de dominio del

servidor, en los objetos cliente.

- o Simplificar el mantenimiento.
- o Posibilidad de deshacer operaciones realizadas antes de consolidarlas en el servidor (persistirlas).

Parte del trabajo es definir una manera de especificar el comportamiento que los objetos de dominios tendrán en el cliente. Esta especificación, debe además, permitir a la plataforma de desarrollo interpretarlas, y resolver cada uno de los problemas que se suscitan al manipular estos objetos en el cliente.

Si bien existen diversas propuestas y soluciones, muchos de estos inconvenientes aún no tienen una solución consensuada o no llegan a cubrir todas las necesidades; y cada vez que se comienza con la definición de la plataforma de desarrollo de una aplicación, con las características mencionadas, se vuelven a atacar desde cero. Las diferentes tecnologías y frameworks para el desarrollo de este tipo de aplicaciones potencian aún más, la variedad de soluciones posibles teniendo que repensar una y otra vez la solución.

Por lo expuesto anteriormente se ve justificada la realización de un análisis sobre los diversos tipos de sistemas Enterprise y de cómo las características de una arquitectura Cliente-Servidor afecta su implementación.

Sobre la base del mencionado análisis se presentará una plataforma de desarrollo, independiente del lenguaje de implementación, a través del cual el desarrollador podrá especificar ciertas propiedades sobre las clases de dominio, relativas a la manipulación de sus instancias en el cliente. Por ejemplo, marcar que una asociación a una colección debe viajar paginada.

Para finalizar y como parte del objetivo principal del trabajo, se diseñará un framework que formará parte de la plataforma y será responsable de interpretar la especificación realizada por el usuario y resolver cada uno de los problemas asociados a la manipulación de objetos de dominio en el cliente.

## Contribuciones

El desarrollo de este trabajo de grado tiene como objetivo global lograr definir una forma de atacar los inconvenientes que se suscitan con frecuencia en una arquitectura Cliente-Servidor, particularmente en sistemas enterprise, cuando se desea manipular objetos de dominio en la componente cliente. Para ello se llevará a cabo el análisis y diseño pertinente, que permitirá determinar los requerimientos no funcionales que se deben atacar.

Concretamente las contribuciones principales que pretende dejar este trabajo de grado son las siguientes:

- Determinar los requerimientos no funcionales recurrentes que se deben tener en cuenta en Aplicaciones Enterprise sobre una arquitectura Cliente-Servidor.
- Describir las características necesarias en una plataforma de desarrollo para mejorar la productividad cumpliendo con los requerimientos no funcionales recurrentes.
- Estandarizar las propiedades mínimas que deben ser especificadas en las clases de dominio.
- Diseñar un framework que se encargue de resolver, de manera transparente, gran parte de los problemas recurrentes de la manipulación de objetos de dominio en la componente cliente de aplicaciones Enterprise.

## Estructura del Documento

En el capítulo 2 daremos una breve introducción a las Aplicaciones Enterprise. Es importante conocer este tipo de aplicaciones dado que este trabajo de grado se basa en las dificultades que nos encontramos al desarrollarlas. Describiremos también en este capítulo los requerimientos funcionales y no funcionales que están presentes en este tipo de aplicaciones.

En el capítulo 3 definiremos lo que se entiende por una arquitectura Cliente-Servidor. Debido a que nos centramos en la implementación de Aplicaciones Enterprise sobre este tipo de arquitecturas y los problemas que esto trae asociado, este tema es primordial. Realizaremos un breve repaso de cómo fue evolucionando hasta llegar a la forma en que hoy en día la conocemos, para culminar presentando las ventajas y desventajas que tiene trabajar con una arquitectura de este estilo.

Debido a que nuestro trabajo se basa en aplicaciones distribuidas sobre una arquitectura Cliente-Servidor, en el capítulo 4 realizaremos un análisis de las alternativas tecnológicas y conceptuales que existen en relación a la distribución del modelo de entidades de una Aplicación Enterprise.

En el capítulo 5 explicaremos por qué es necesario manipular objetos persistentes en el componente cliente de una aplicación y pondremos en evidencia los problemas que esto trae aparejado, identificando los inconvenientes principales y explicando detalladamente cada uno de ellos. Este tema es fundamental ya que gran parte de este trabajo de grado se centra en intentar solucionar los inconvenientes recurrentes explicados en este capítulo.

En el capítulo 6 hablaremos sobre las plataformas de desarrollo y presentaremos el diseño de un framework que presenta soluciones a los problemas detallados en los capítulos previos.

Para terminar, en el capítulo séptimo enunciaremos las conclusiones que surgen de este trabajo, así como también los trabajos a realizar en el futuro y que quedan fuera de los límites de este trabajo de grado.

# Capítulo 2

## Aplicaciones Enterprise de gran tamaño

*En la mayor parte del ciclo de vida de desarrollo de Aplicaciones Enterprise se pierden de vista las características de la Empresa u Organización para la cual se desarrollan este tipo de aplicaciones. En este capítulo describiremos a qué llamamos un Sistema Enterprise, sus principales características y los requerimientos funcionales y no funcionales que se desprenden de su análisis.*

### Descripción

La palabra Enterprise[2] puede tener varias connotaciones. A menudo el término es utilizado únicamente para referirse a grandes, sin embargo, puede ser usado para significar casi cualquier cosa, en virtud de que se ha establecido como 'palabra de moda' dentro del lenguaje corporativo.

Las Aplicaciones Enterprise son un tipo de software destinado a proporcionar soporte a la lógica de negocio de grandes empresas, las cuales tienen como objetivo mejorar sus procesos, es decir, su eficiencia, su productividad, su manera de relacionar la información, sus recursos humanos, tecnológicos y de infraestructura.

Los servicios provistos por las Aplicaciones Enterprise son típicamente orientados a los negocios, como por ejemplo, compras y procesos de pagos "on line", catálogos de productos interactivos, sistemas de facturación automatizados, seguridad, administradores de contenido, administración de la relación con el cliente (CRM), planificación y administración de recursos (ERP), inteligencia de negocio, manufactura, integración entre aplicaciones, etc.

Para analizar de forma completa las características de este tipo de aplicaciones, analizaremos aplicaciones de gran tamaño o que se apliquen en organizaciones donde los procesos de negocio sean complejos y estratificados. Esto involucra organizaciones con un gran número de personas que desarrollan sus tareas a través del sistema, con gran dispersión geográfica y realizando tareas involucradas en procesos complejos.

Algunas de las características principales de las Aplicaciones Enterprise son performance, escalabilidad, robustez y en la mayoría de los casos

[2] Utilizamos el término en inglés debido a que la industria está más familiarizada con él que con su traducción al español

integración con otras Aplicaciones Enterprise, las cuales son en conjunto, administradas de manera centralizada. [Woo03]

Este tipo de software es usualmente instalado en servidores y pueden proveer servicios simultáneamente a un gran número de empresas, usualmente sobre redes de computadoras. Esta es la principal diferencia entre el software de aplicaciones mono usuarios, el cual se ejecuta en la maquina local, propia del cliente y atiende a un único usuario a la vez.

## Sistemas de Información Enterprise

Un Sistema de Información Enterprise es generalmente un conjunto de sistemas de computación que ofrece servicios de alta calidad, que maneja grandes volúmenes de datos y es capaz de soportar grandes organizaciones.

Estos sistemas proveen una plataforma tecnológica que permite a las organizaciones integrar y coordinar sus procesos de negocio, proporcionando un sistema único que es fundamental para la organización y aseguran que la información pueda ser compartida en todos los niveles funcionales y jerarquías administrativas. Esto último resulta de gran valor al eliminar los problemas originados por la fragmentación de la información que se genera al utilizar múltiples sistemas de información en la misma organización, mediante la creación de estándares en las estructuras de datos propia de la empresa.

***Figura 1 - Sistemas Enterprise***

Un Sistema de Información Enterprise puede encontrarse alocado en uno o más centros de datos en donde se encuentran ejecutándose la o las Aplicaciones Enterprise que lo conforman.

## Procesos de Negocios

Un proceso de negocio comienza con una necesidad del cliente y termina con dicha necesidad satisfecha. Las organizaciones orientadas a procesos intentan romper las barreras estructurales de los departamentos y tratan de evitar los silos funcionales.

Los procesos de negocio son diseñados para agregar valor al cliente y no deben agregar actividades innecesarias. El resultado de un proceso de negocios bien diseñado es el aumento de la eficacia (valor para el cliente) y una mayor eficiencia (menos costo para la empresa).

Los procesos de negocio consisten en subprocesos, decisiones y actividades. Un subproceso es parte de un proceso de mayor nivel que tiene su propia meta, propietario, entradas y salidas. Las actividades son partes de los procesos de negocio que no incluyen ninguna toma de decisión ni vale la pena descomponer (aunque ello sea posible). Por ejemplo, “Responde al teléfono”, “Realizar una factura”.

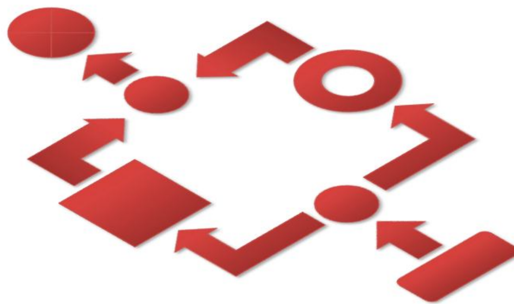

***Figura 2 - Procesos de negocios***

El análisis de los procesos de negocio incluye típicamente la asignación de los procesos y sub-procesos a nivel de actividad.

### Definiciones

<u>Davenport [Dav92]:</u>

> “Un conjunto de actividades estructuradas y medidas, destinadas a producir un producto específico para un cliente o mercado. Esto implica un fuerte énfasis en la forma en *cómo* se realiza el trabajo dentro de una organización, en contraste a enfatizar *qué* es lo se hace. Un proceso es entonces, una organización de las actividades laborales a través del tiempo y el espacio, con un principio y un fin, en donde se encuentran claramente definidas las entradas y salidas. Tomar un enfoque

orientado a procesos implica la adopción del punto de vista del cliente. Los procesos son las estructuras por la que una organización hace lo necesario para producir valor para sus clientes"

Esta definición contiene algunas características que un proceso debe poseer. Estas características se obtienen enfocándose en la lógica de negocio del proceso (cómo se realiza el trabajo), en lugar de tomar una perspectiva de producto (lo que se hace).

Tras la definición previa, podemos concluir que un proceso debe tener límites claramente definidos, de entrada y salida, que se compone de partes más pequeñas, las actividades, que están ordenados en el tiempo y el espacio, que debe haber un receptor de la obtención de resultados del proceso - un cliente - y que la transformación que tiene lugar en el proceso debe contener valor agregado al cliente.

Hammer & Champy's [Ham03]:

Puede considerarse como un subconjunto de la definición de Davenport. Ellos definen un proceso como "un conjunto de actividades que tenga uno o más tipos de entrada y crea un producto que tiene valor para el cliente."

Como podemos observar, Hammer y Champy tiene una percepción más orientada a la transformación, y hacen menos hincapié en el componente estructural – límites del proceso y el orden de las actividades en el tiempo y el espacio.

Rummler y Brache [Rum95]:

Utilizan una definición que está claramente centrada en la organización de los clientes externos, al afirmar que:

"Un proceso de negocio es una serie de pasos o medidas destinadas a producir un producto o servicio. (...) La mayoría de los procesos son funcionalmente transversales, que llenan el "espacio en blanco" entre las cajas del diagrama de la organización. Algunos procesos tienen como resultado un producto o servicio destinado a clientes externos de la organización. Llamamos e estos Procesos Primarios. Otros procesos de producción de productos son invisibles para el cliente externo, pero esenciales para la gestión eficaz de la empresa. Llamamos a estos Procesos de Apoyo."

En la definición anterior podemos distinguir dos tipos de procesos, primarios y procesos de apoyo, dependiendo de si un proceso está directamente involucrado en la creación de valor para el cliente, o relacionado con la organización interna de las

actividades. En este sentido, la definición de Rummler Brache sigue el modelo de la cadena de valor de Porter, que también se basa en una división de las actividades primarias y secundarias.

Johansson:

"Un conjunto de actividades vinculadas que tienen una entrada la cual se transforma para crear una salida. Idealmente, la transformación que se produce en el proceso debe añadir valor a la entrada y crear un producto que sea más útil y eficaz para el receptor"

Esta definición hace hincapié en la constitución de los vínculos entre las actividades y la transformación que se lleva a cabo dentro del proceso.

### Características de los procesos de negocios

Resumiendo las cuatro definiciones anteriores, podemos compilar la siguiente lista de características de un proceso de negocio.

- **Definición:** debe tener definidos claramente los límites, entrada y salida.
- **Orden:** debe consistir de actividades que están ordenadas de acuerdo a su posición en el tiempo y espacio.
- **Cliente:** debe existir un destinatario de los resultados del proceso, un cliente.
- **Valor agregado:** la transformación que tiene lugar dentro del proceso debe agregar valor al destinatario.
- **Integración:** un proceso no puede existir por sí solo, debe estar integrado en una estructura organizacional.
- **Funcionalidad Cross:** un proceso regularmente puede, pero no necesariamente debe, abarcar varias funciones.
- Frecuentemente, es considerado un prerrequisito contar con una persona que sea responsable de la eficiencia y mejora continua de los procesos.

## Requerimientos Funcionales

Si bien los requerimientos funcionales que las Aplicaciones Enterprise deben cumplimentar se derivan mayormente del dominio al cual esté orientado la empresa u organización [Wie06]. De la operatoria diaria surgen

ciertas necesidades funcionales generales que la aplicación debe contemplar.

Las Aplicaciones Enterprise orientadas a empresas de envergadura, por lo general tienen como base algunas de las características que detallamos a continuación.

La información debe estar integrada

La información generada por cada área de la organización es un insumo para alguna otra y por ende se debe administrar de forma centralizada para facilitar su disponibilidad para quien la necesite.

Esto requiere en principio:

- La definición de los procesos en forma centralizada - centralización normativa
- La estandarización de la información de uso común - codificaciones
- Descentralización operativa

En resumen, las distintas áreas o sucursales operan en forma descentralizada siguiendo normas definidas de manera centralizada

Manejo de objetos del negocio de gran tamaño

El sistema debe soportar que sus operaciones puedan trabajar sobre objetos que, por modelar conceptos del dominio que disponen de mucha información, son naturalmente de gran tamaño.

Log de operaciones

Llevar registro de cada una de las operaciones que fueron ejecutadas por los usuarios finales, con el fin de realizar auditorías, controles y seguimientos.

Definición de perfiles y niveles de información

Se debe permitir la definición de perfiles, dominios de información y funcionalidad con los que se pueden trabajar en cada uno de los perfiles configurados.

Multimoneda

Las gestiones requieren la utilización de recursos financieros en distintas monedas en el mundo globalizado.

## Requerimientos No Funcionales

Como consecuencia de las características inherentes a las Aplicaciones Enterprise de gran tamaño, de los requerimientos funcionales y de los procesos que se deben cubrir en este tipo de aplicaciones, aparecen como parte fundamental del problema, requerimientos no funcionales que también deben ser considerados en la solución [Wie06]. A continuación se describen algunos de los más importantes:

Performance

La performance mide distintos tiempos, los cuales se describen a continuación:

- **Tiempo de respuesta:** tiempo que tarda el sistema en procesar un pedido externo (por ejemplo a partir de presionar un botón)
- **Tiempo de respuesta al usuario:** tiempo que tarda el sistema en responder al usuario (aunque posiblemente no haya terminado de procesar). En un proceso asíncrono, aún con un tiempo de respuesta alto, se podrá bajar el tiempo de respuesta al usuario si se da una respuesta inmediata indicando que su proceso fue encolado.
- **Tiempo de latencia:** Es el tiempo mínimo de cualquier respuesta.
- **Throughput:** Cuánto trabajo se puede hacer en un determinado tiempo. En Aplicaciones Enterprise suelen medirse transacciones por segundo.

La performance cumple un rol muy importante en cualquier aplicación. Este requerimiento, frecuentemente marca el éxito o fracaso del proyecto. La aplicación necesitará ejecutar la mayoría de las transacciones con un tiempo de respuesta adecuado, de manera de no degradar la calidad de servicio percibida por los usuarios. Son éstos los que determinan la aceptación del sistema.

El gran esfuerzo del equipo de desarrollo se puede ver deslucido por problemas de performance pues, generalmente, produce disconformidad en los usuarios del sistema.

El impacto de una mala performance en el sistema se puede ver cuantificado en tres puntos [Hai06]:

- Pérdida de productividad
- Pérdida de confianza y credibilidad del cliente
- Perdida en las ganancias

Si bien lo expresado respecto del tiempo de respuesta aplica a cualquier tipo de aplicación, alcanzar buena performance en Aplicaciones Enterprise es más complicado que en otro tipo de aplicaciones. Algunos factores que pueden afectar la performance son

- **A nivel físico:** el acceso a la base de datos, el enlace de red utilizado y la capacidad de procesamiento de los servidores involucrados.
- **A nivel diseño:** la arquitectura, las pautas de desarrollo definidas, la buena utilización de los frameworks, la optimización de consultas a BD, entre otros.

Tomar decisiones sobre la performance es una tarea difícil. Es importante tenerla en cuenta al tomar cualquier decisión de diseño, tanto a nivel de arquitectura como a nivel de desarrollo de la aplicación.

De todas maneras, aún habiéndola tenido en cuenta en las etapas tempranas de diseño e inclusive en la definición de la arquitectura, los problemas de performance pueden producirse, y el equipo debe estar preparado para poder afrontarla. No alcanzará sólo con analizar el diseño, sino que además son necesarias herramientas de medición y "profiling" para poder descubrir un posible cuello de botella, un algoritmo ineficiente, un método invocado varias veces de manera innecesaria o cualquier otra causante del problema.

Capacidad y Escalabilidad

Las Aplicaciones Enterprise deben ser capaces de crecer en escala para adecuar el servicio al crecimiento de la demanda. Es importante tener en mente en cada momento del desarrollo estos dos conceptos, y también saber cuáles son las cosas que *no* se deben hacer al construir aplicaciones enterprise [Sch06].

Si bien es frecuente realizar pruebas para verificar que se cumplan los tiempos de respuesta requeridos para la funcionalidad implementada, es importante realizar pruebas de carga para simular un ambiente donde se reproduzca la carga real. De esta manera, se podrá detectar el umbral de degradación del sistema, anticipando el problema con margen suficiente para su análisis y solución.

Usabilidad

Otro de los requerimientos muy importantes, por tener impacto directo en el usuario final, es la usabilidad. La usabilidad define las pautas que tienen como objetivo garantizar una interacción

cómoda, simple y que se ajuste a la operatoria diaria del usuario con el sistema.

Las mencionadas pautas, serán quienes guíen el diseño de las interfaces gráficas de la aplicación.

Para definir las pautas de usabilidad es importante conocer, además de la funcionalidad que se quiere proveer, la actividad diaria que está automatizando dicha funcionalidad para que sea natural su uso. Otro punto importante es determinar las distintas categorías de usuarios que pueden usar el sistema.

***Figura 3 - Usabilidad***

Frecuentemente ocurre que el sistema se releva con expertos del dominio que no serán los únicos usuarios del sistema. Existe una relación entre caso de uso y categoría de usuario. Esto no debe perderse de vista en la definición la interfaz.

Adaptabilidad al cambio de reglas de negocio

Esta necesidad puede producirse por la poca experiencia de los analistas y de los usuarios en el dominio que se está modelando, o por la variabilidad inherente de las reglas del negocio que se modela.

Este punto es importante en sistemas que evolucionan con el tiempo, debido a que la poca adaptabilidad puede derivar en cambios drásticos en la aplicación, lo cual trae aparejado un gran trabajo para poder realizar las adaptaciones.

Portabilidad a través de plataformas

Las Aplicaciones Enterprise deben tener la capacidad de operar bajo una variedad de plataformas y sistemas operativos.

Si bien no es objetivo de este capítulo hablar del marco de

arquitectura de este tipo de aplicaciones, cabe mencionar (para poder estudiar la portabilidad) que dispondremos de clientes y servidores de aplicación.

***Figura 4 - Multiplataforma***

Del lado del servidor de aplicación, la portabilidad podría surgir como una necesidad del equipo de desarrollo. Ellos serán quienes administran los servidores y por diferentes razones (económicas, disponibilidad, conocimiento, administración, entre otras) podrían sugerir cambios de plataformas o sistemas operativos en los equipos que alojan al servidor de la aplicación. Desarrollar, entonces, un producto que se pueda adaptar a estos cambios durante el desarrollo y la puesta en producción es un punto a tener en cuenta. [Mar10]

En aplicaciones de clientes pesados, la portabilidad del cliente también es importante dado que los usuarios pueden disponer de diferentes ambientes y sistemas operativos.

En el caso de aplicaciones Web, la portabilidad debe darse tanto en Web Server así como en los browsers. Estos últimos debido a la compatibilidad entre los distinto browsers disponibles en la actualidad, y las diversas herramientas y lenguajes disponibles para la implementación de aplicación ricas sobre internet - RIA.

Seguridad

Las Aplicaciones Enterprise deberán proveer una infraestructura de alta seguridad, acorde al nivel de exigencia de la industria u organización, que proteja al sistema contra las vulnerabilidades de las aplicaciones de gran escala [Vac09] [And01].

Este tipo de arquitectura debe contar con las siguientes consideraciones:

### *Control de acceso*

Podemos tomar el control de acceso como una colección de mecanismos que tienen influencia directa sobre el comportamiento, uso y contenido del sistema. El control de acceso administra específicamente que es lo que el usuario puede hacer, a que recursos tiene acceso y que operaciones puede realizar en el sistema [Tip03].

Asimismo la administración debe ser centralizada y almacenada de forma unificada.

***Figura 5 - Segmentación de usuarios***

### *Contraseña única (Single Sign-On)*

La implementación de este tipo de validación de usuarios permite a los mismos un acceso transparente a través de todos los módulos que posea el sistema [Tip03].

### *Encriptación*

Las Aplicaciones Enterprise deben hacer uso de técnicas de criptografía para asegurar la protección de la información sensible del sistema, como así también encriptaciones a nivel de transmisiones de datos en la red [Sta02].

Alta Disponibilidad

El sistema debe ser capaz de cumplir con la exigencia de disponibilidad que requiere la empresa u organización. Para las Aplicaciones Enterprise comúnmente es de 7 por 24 para las principales componentes de software del sistema transaccional, no incluyéndose subsistemas de consulta e históricos.

Mantenibilidad

La arquitectura sobre la que se desarrolla el sistema Enterprise debe separar claramente las capas de presentación y de

negocio, de esta manera, el producto ha de tener la habilidad de cambiar fácilmente la apariencia y percepción, permitiendo así la fácil adecuación a las distintas sucursales o áreas en las que se utiliza el sistema. Con una división modular y basada en componentes de cada una de las capas, se permite la posibilidad y facilidad de realizar adaptaciones, adecuaciones, modificaciones y extensiones ante los sucesivos cambios que suelen tener las empresas y organizaciones.

Hot deploy

Es necesario que se contemple en la arquitectura la posibilidad de realizar reinstalaciones de parte del sistema sin afectar al sistema completo, ante emergencias que no permitan sacar de línea el sistema.

Autonomía del usuario

Se debe garantizar la autonomía del usuario, permitiendo que defina circuitos que se adecuen a su ubicación geográfica, a sus gestiones y a sus dimensiones, sin requerir intervención del área de sistemas, y facilitando el tratamiento de información fuera de línea y con distinto nivel de agregación (por ejemplo Datawarehouses).

Descentralización de operaciones

Debe permitir la habilitación e inhabilitación de funciones en cada proceso desde la casa central a medida que las sucursales crecen en rendimiento y madurez.

Modos de Operación

- **Modo normal:** Es el modo en el que los usuarios del sistema podrán disponer de toda la funcionalidad. Dentro de este modo se cuenta con la posibilidad de trabajar en forma interactiva y en forma desatendida (batch) para el procesamiento masivo de datos.
- **Modo mantenimiento:** En este modo se encuentra el sistema cuando se realizan modificaciones de infraestructura, hardware, software de base o software de aplicación, brindando la posibilidad de deshabilitar temporalmente los componentes del sistema afectados en su funcionalidad por estas modificaciones.

## Uso de Red

Considerando que la interacción con sistemas externos y que la comunicación entre los distintos servidores de la arquitectura física (Servidores Web, Servidores de Aplicación y Bases de Datos) se hará a través de la red, es necesario privilegiar en el diseño y programación la optimización del uso de la red.

También es muy importante optimizar el uso de red desde el cliente al servidor. Existen estrategias de diseño y de arquitectura que permiten conseguir esta optimización, sin apartarse de un buen modelo de objetos. Abordaremos este tema en secciones posteriores.

## Facilidad de Instalación

Es necesario definir un proceso repetible para la instalación de la aplicación. Este proceso permitirá automatizar, con herramientas, la instalación del sistema en los distintos ambientes en etapas de desarrollo y producción. En el primer caso, se requieren múltiples ambientes, a saber:

- Ambiente de Desarrollo
- Ambiente de Aceptación de los usuarios
- Ambiente de Capacitadores
- Ambiente de Prueba de carga

En el segundo caso, se requieren:

- Ambiente de Preproducción
- Ambiente de Producción

El proceso no sólo deberá definir las actividades relacionadas con el deploy de la aplicación sino también pautas que deben ser cumplidas durante el desarrollo. Este proceso contempla la automatización de tareas sobre la base datos y tareas de configuración de datos de instalación. Para que la automatización sea posible, el equipo de desarrollo debe cumplir con estas pautas preestablecidas.

Para automatizar el proceso se debe disponer de herramientas que faciliten la tarea de deploy y registro de logs para su posterior verificación.

Adicionalmente, para atender las emergencias de la instalación, se debe desarrollar un proceso que permita realizar un hot deploy manteniendo la disponibilidad del sistema en producción.

## Integración con sistemas legacy

Una Aplicación Enterprise se construye de forma modular. Para darle visibilidad al sistema y servicio al cliente, los módulos se van instalando progresivamente.

Si el sistema reemplaza un sistema existente, surge como consecuencia de la decisión de la instalación progresiva, un proyecto que llamaremos convivencia. Este proyecto se ocupa de definir estrategias de migración de datos y de comunicación entre los sistemas legacy y el nuevo.

# *Capítulo 3*

# Arquitectura de Software

*La arquitectura de software es la base fundamental de toda infraestructura de procesamiento de información. En este capítulo discutiremos cómo evolucionaron las diferentes infraestructuras y arquitecturas, y nos detendremos en la más utilizada y extendida en los últimos tiempos, la arquitectura Cliente-Servidor.*

## Evolución de la infraestructura de procesamiento de información

Existen diversos puntos de vista sobre la manera en que debería efectuarse el procesamiento de datos, aunque muchos coinciden en que nos encontramos en medio de un proceso de evolución que se prolongará todavía por algunos años y que cambiará la forma en que obtenemos y utilizamos la información almacenada electrónicamente.

El principal motivo detrás de esta evolución es la necesidad que tienen las organizaciones (empresas, instituciones públicas o privadas), de realizar sus operaciones de manera más ágil y eficiente, lo cual se traduce en una mayor productividad del personal, en una reducción de los costos y gastos de operación, y al mismo tiempo generar de manera más rápida productos y servicios de mejor calidad.

La cantidad de variantes con las cuales son desarrolladas las aplicaciones que proporcionan dichos servicios y productos, se incrementó tal manera que se comenzó a reconocer la necesidad de una definición estandarizada de una aplicación base que sirva como modelo para todas las demás.

En este contexto, es necesario establecer una infraestructura de procesamiento de información que cuente con los elementos necesarios para proveer la información adecuada, exacta y oportuna en la toma de decisiones, y proporcionar así un mejor servicio a los clientes. [Hof99]

## Un poco de historia

La infraestructura sobre la cual se implementan los sistemas de información ha sufrido una evolución sostenida, así como los modelos y tecnologías que intervienen en el desarrollo de la misma.

Existen y existieron diversos modelos con las características necesarias para proveer esta infraestructura, independientemente del tamaño y

complejidad de las operaciones de las organizaciones y consecuentemente. A continuación repasaremos este proceso de evolución:

La era de la computadora central

Desde sus inicios, el modelo de administración de datos a través de computadoras se basaba en el uso de terminales remotas, que se conectaban de manera directa a una computadora central, la cual se encargaba de prestar servicios personalizados sólo a un grupo exclusivo de usuarios

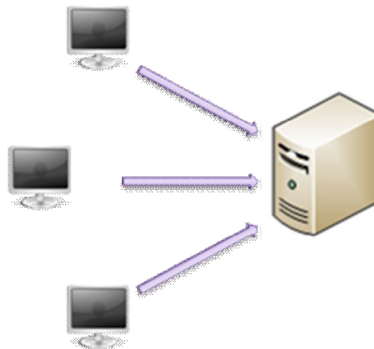

***Figura 6 - Computadora central***

La era de las computadoras dedicadas

Ésta es la era en la que cada servicio contaba con su propia computadora, lo que permitía que los usuarios de ese servicio se conectaran directamente. Esto es consecuencia de la aparición de computadoras más pequeñas, de fácil uso, más baratas y más poderosas que las computadoras convencionales utilizadas hasta ese momento.

La era de la conexión libre (computadoras de escritorio)

Hace más de 10 años que las computadoras de escritorio aparecieron de manera masiva. Esto permitió que una parte apreciable de la carga de trabajo de cómputo, tanto en el ámbito de cálculo como en el ámbito de la presentación, se lleven a cabo desde el escritorio del usuario.

En muchos de los casos, el usuario obtiene la información que necesita de alguna computadora de servicio. Estas computadoras de escritorio se conectan a las computadoras de servicio empleando software que permite la emulación de algún tipo de terminal. En otros casos se les transfiere la información haciendo uso de recursos magnéticos o por trascripción.

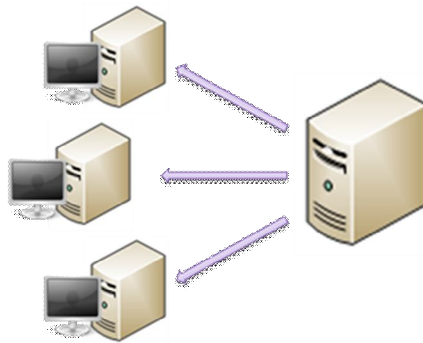

***Figura 7 - Computadora de escritorio***

La era del cómputo a través de redes

Esta era se basa en el concepto de redes de computadoras, en la que la información reside en una o varias computadoras, los usuarios de esta información hacen uso de computadoras para trabajar y todas ellas se encuentran conectadas entre sí. Esto brinda la posibilidad de que todos los usuarios puedan acceder a la información de todas las computadoras, y a su vez, que los diversos sistemas intercambien información.

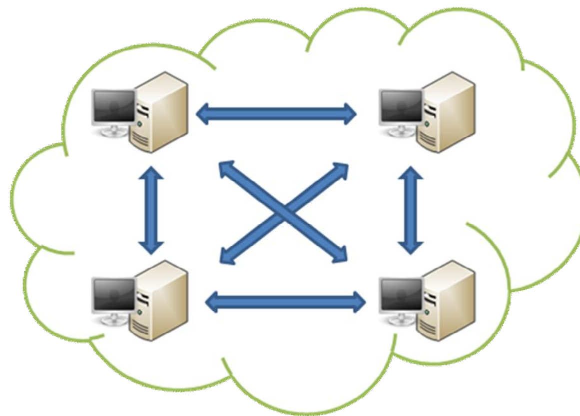

***Figura 8 - Red de computadoras***

La era del cliente servidor

En esta era la computadora de cada uno de los usuarios, llamada cliente, produce un requerimiento a cualquiera de las computadoras que proporcionan servicios, conocidas como servidores.

Los clientes y los servidores pueden estar conectados a una LAN o a una WAN, como la que se puede implementar en una empresa o a una red mundial como lo es la Internet.

Bajo este modelo cada usuario tiene la libertad de obtener la información que requiera en un momento dado proveniente de una o varias fuentes locales o distantes y de procesarla según le convenga, permitiendo además, el intercambio de información entre los distintos servidores.

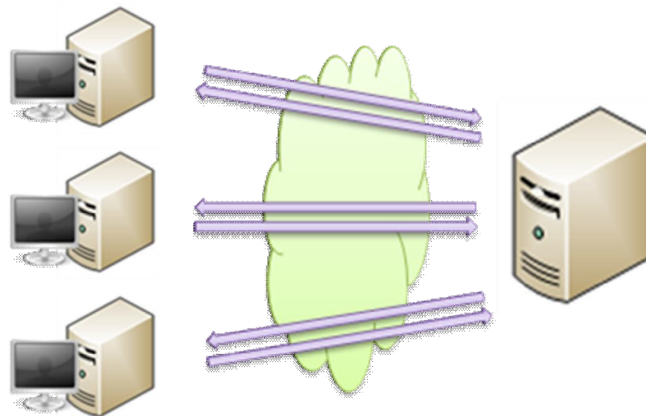

***Figura 9 - Cliente / Servidor***

## ¿Qué es la Arquitectura de Software?

En los comienzos de la informática la programación se desarrollaba casi como un arte, debido a la dificultad que significaba esto para la mayoría de las personas. Esta forma de ver la programación fue cambiando con el tiempo debido a la evolución de **plataformas de desarrollo**, en base a las cuales se puedan resolver los problemas. La parte fundamental de estas plataformas son las denominadas Arquitecturas de Software [Hoh03].

La definición del término **Arquitectura** varía de acuerdo al autor de la bibliografía consultada. Si bien existen muchas definiciones de lo más variadas, se puede observar que hay dos elementos que se repiten en todas ellas: Uno es la **descomposición de alto nivel** que divide al sistema en sus partes; el otro son las **decisiones que son difíciles de cambiar**.

Cada vez es más común concluir que no hay sólo un camino para definir la arquitectura de un sistema, si no que hay múltiples arquitecturas en un sistema, y la visión de qué es arquitecturalmente significativo puede cambiar a lo largo de la vida del sistema.

A los elementos mencionados se le pueden agregar aspectos que caracterizan una arquitectura de software. Por ejemplo, se puede decir que una arquitectura de software consiste en un conjunto de patrones y abstracciones coherentes que proporcionan el marco de referencia necesario para guiar la construcción del software para un sistema de

información [Fow02]. También contribuye a establecer los fundamentos de la **plataforma de desarrollo** a la que pertenece, para que analistas, diseñadores, programadores, etc. trabajen en una línea común que permita alcanzar los objetivos del sistema de información, cubriendo todas las necesidades.

Una arquitectura de software se selecciona y diseña con base en objetivos y restricciones. Los **objetivos** son aquellos prefijados para el sistema de información, pero no solamente los de tipo funcional, también otros objetivos como la estandarización en el desarrollo del sistema, el mantenimiento, la capacidad de ser auditada, flexibilidad e interacción con otros sistemas de información. Las **restricciones** son aquellas limitaciones derivadas de las tecnologías disponibles para implementar sistemas de información. Unas arquitecturas son más recomendables de implementar con ciertas tecnologías mientras que otras tecnologías no son aptas para determinadas arquitecturas. Por ejemplo, no es viable emplear una arquitectura de software de tres capas para implementar sistemas en tiempo real.

La arquitectura de software también definen, de manera abstracta, los componentes que llevan a cabo alguna tarea de computación, sus interfaces y la comunicación entre ellos. Toda arquitectura debe poder implementarse en una arquitectura física, que de manera simplificada consiste en determinar qué computadora tendrá asignada cada tarea.

Como conclusión podemos decir que el término se refiere al soporte necesario para la construcción de sistemas que garanticen el cumplimiento de los requerimientos planteados, asegurando calidad y aumentando al máximo la productividad.

## Estilos de Arquitectura

En función de las características de la aplicación, se elige inicialmente un estilo arquitectónico. A partir de este marco, se trabaja para definir la plataforma de desarrollo a utilizar:

Un estilo define:

- Glosario: tipos de elementos y tipos de conectores que participan. Ejemplos de elementos: cliente, servidor, base de datos, filtro, capas.
- Restricciones de combinación

Las arquitecturas complejas o compuestas resultan del agregado o la composición de estilos más elementales, los que podríamos catalogar dentro de patrones arquitecturales [Bus96]. A continuación describiremos tres de los estilos arquitectónicos más conocidos para luego comenzar con la presentación de la arquitectura que prevalece en las Aplicaciones Enterprise.

## Pipes and filters

Podemos utilizar este tipo de arquitecturas para dividir una tarea de procesamiento grande, en una secuencia de pequeños pasos de procesamientos independientes (Filters) que están conectados por canales (Pipes).

***Figura 10 - Pipes and Filters***

Cada filtro expone una interfaz muy simple, recibe los mensajes en el canal de entrada, procesa el mensaje, y publica los resultados en el canal de salida. Cada canal conecta un filtro con el próximo, enviando el mensaje de salida, de un filtro al siguiente.

Dado que todos los componentes utilizan la misma interfaz externa, es posible componer diferentes soluciones mediante la conexión de los componentes a pipes diferentes. Podemos añadir nuevos filtros, omitir o cambiar los ya existentes y todo ello sin tener que tocar los filtros.

En ocasiones a la conexión entre el canal y el filtro se la denomina puerto. En la forma básica, cada componente de filtro tiene un puerto de entrada y un puerto de salida.

## Publisher Subscriber (invocación implícita)

Es un modelo en el cual existen elementos de la arquitectura (subscribers) interesados en enterarse de ciertos eventos que pueden ocurrir en otro elemento de la arquitectura (publisher).

La solución tradicional sería que el publisher avise explícitamente a cada uno de los subscribers la ocurrencia de un evento, para que cada uno actúe en consecuencia. Para esto, el publisher será el responsable de conocer los eventos de interés. Este conocimiento explícito produce un gran acoplamiento entre el publisher hacia los subscribers y se da una responsabilidad incorrecta al publisher, pues no es él quien debe mantener la información de interés en los eventos.

Para evitar los problemas mencionados, se invierte el conocimiento. Los elementos interesados se "suscriben" a los eventos de interés del publisher. Cuando se produce un evento, el publisher simplemente avisa que ocurrió el evento y los subscribers se enteran a través del soporte del mecanismo de

observación.

Resumiendo, en este estilo participan dos tipos de elementos, los publishers y los subscribers. Estos elementos se relacionan a través de eventos.

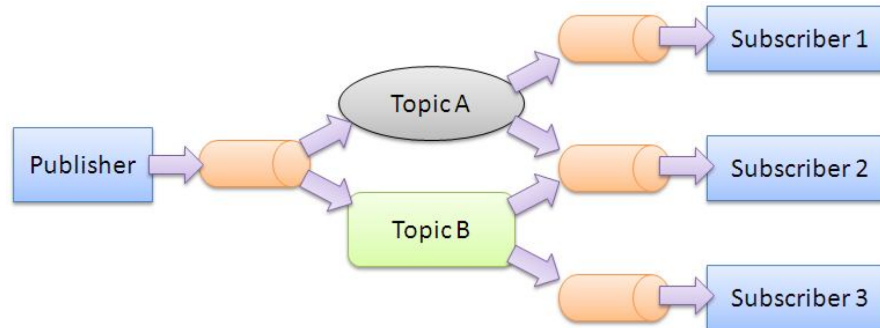


***Figura 11 - Publisher Subscriber***

Las principales ventajas de este estilo son:

- Simplicidad
- Desacoplamiento: El mecanismo evita tener que modificar el asunto ante la aparición de nuevos subscribers. Serán los subscribers los responsables de subscribirse a los eventos de interés.
- Puede mejorar eficiencia, eliminando la necesidad de polling por ocurrencia de evento

Entre sus desventajas encontramos:

- No se puede utilizar entre cualquier par de elementos de la arquitectura. Para poder implementar el mecanismo, es necesario que exista una relación entre el publisher y el subscriber que se mantenga en el tiempo.
- Difícil de comprender: Puede ser difícil conocer qué pasará en respuesta a una acción pues existen invocaciones que no se ven reflejadas directamente en el código.
- Pueden producirse fallas que dejen suscriptores sin recibir mensajes.

**Blackboard (pizarrón)**

El concepto de la arquitectura blackboard o pizarrón surgió en el campo de la inteligencia artificial hace más de una década. Su propósito es concentrar y compartir un problema entre múltiples agentes. El nombre de arquitectura blackboard o pizarrón evoca la metáfora en la cual un grupo de expertos

frente a un pizarrón colaboran en la resolución de un problema complejo.

El pizarrón se utiliza como repositorio central para compartir la información. Esta información representa hechos, asunciones y deducciones hechas por los expertos en el transcurso de la búsqueda de solución de un problema. Cada experto aporta desde su conocimiento, una estrategia distinta para resolver el problema. Un moderador controla la tiza y escribe sobre la pizarra, determinando qué información brindada por los expertos, aporta a la solución del problema.

Una sesión de resolución comienza con la escritura de una especificación de problema por el moderador, junto con los hechos relevantes que puedan aportar a la solución del problema. Los expertos analizan el problema y hacen los aportes que puedan sobre la base de los conocimientos específicos de cada uno, requiriendo la atención del moderador para cada aporte. El moderador elige, entre todos los expertos, la contribución más prometedora y la escribe en la pizarra. Este proceso continúa de forma iterativa hasta que el problema es resuelto.

En términos más precisos, el pizarrón podría ser una base de datos que representa la memoria del sistema de resolución de problemas. Los expertos son subsistemas modulares denominados fuente de conocimientos, que representan los distintos puntos de vistas, estrategias y tipos de conocimiento de cómo resolver el problema. Este paradigma de resolución de problemas incluye:

- Sistemas basados en reglas
- Redes neuronales
- Sistemas de lógica difusa
- Algoritmos genéticos
- Sistemas legados y/o tradicionales

Un sistema de control representa la función del moderador, comprendido por detectores de eventos y coordinadores de agenda, que controla la interacción entre la pizarra, fuentes de conocimiento y fuentes externas como información de usuarios y otros sistemas de control.

Los detectores de eventos actualizan la información de la pizarra provista de las fuentes externas. El coordinador de la agenda elige el experto que puede escribir en la pizarra.

En el ambiente académico, los sistemas con modelos arquitectónicos de tipo blackboard son muy eficientes, pero únicos y artesanales, requiriendo grandes esfuerzos en investigación, diseño, desarrollo y mantenimiento.

Los sistemas con arquitecturas Blackboard poseen las siguientes características:

- Integración de fuentes de conocimientos dispares, manejados

de manera transparente por el sistema de control.

- Modularidad. Existe una independencia ente cada una de las fuentes de conocimientos facilitando el desarrollo y mantenimiento.
- Flexibilidad. Con arquitecturas Blackboard los sistemas se adaptan fácilmente a los requerimientos cambiantes.
- Rehúso. Obtenida de las siguientes maneras

La independencia de las fuentes de conocimiento hace posible la construcción de otros sistemas utilizando las mismas fuentes.

Sistemas legados pueden ser preservados e incorporados como parte del conocimiento base.

El Blackboard en sí mismo es una aplicación independiente que puede ser utilizada en otros dominios.

Extensibilidad. Nuevas fuentes de conocimiento pueden ser desarrolladas y agregadas sin impactar en el sistema existente.

### Layered (en capas)

Los elementos de este tipo de arquitecturas son layers. Si bien existen varias traducciones de la palabra, utilizaremos el término en inglés para evitar confusiones.

Cada layer tiene una responsabilidad bien definida, que requiere comunicación con las capas subsiguientes para llevarla a cabo. La organización de los layers se puede mostrar tanto horizontalmente como verticalmente [Cle01]. En el primer caso, una layer se puede comunicar con los layers de abajo. En el segundo caso, un layer se comunica con los layers inmediatamente a su derecha.

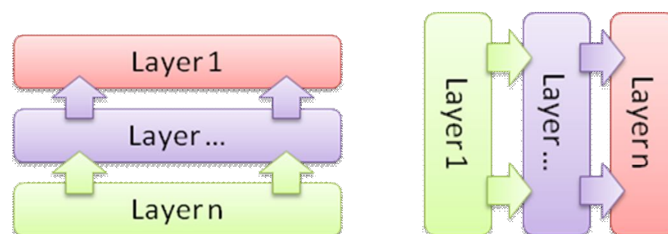


***Figura 12 - Layered***

Si bien en la teoría, un layer no debería poder interactuar directamente con otro layer si no está inmediatamente a su lado (arquitecturas layered opacas), en la práctica no siempre ocurre por cuestiones de performance o

simplicidad. El hecho de que no sea un modelo puro de layers atentará contra la mantenibilidad, pero es necesario hacer un análisis de costo-beneficio y definir pautas claras de cuándo sobrepasar capas.

Las principales ventajas de este estilo son:

- Desacoplamiento. Se puede desarrollar un layer aún sin disponer de los layers con los que debe colaborar. Alcanza tan solo con conocer el "contrato" de dichos layers. El contrato será el conjunto de servicios que ofrece un layer. Este desacoplamiento favorece la división de trabajo en el equipo de desarrollo y el testing aislado.
- Es posible cambiar implementaciones de layers (con la misma interface) sin modificar las restantes.
- Si se respeta el desacoplamiento y se mantiene el modelo lo más opaco que se pueda, un cambio en el contrato de dos layers no debería afectar a los demás. Por ejemplo, supongamos que se tienen los layers A, B y C. Si cambia el contrato entre B y C no debería tener que modificar A (asumiendo una arquitectura opaca donde A no consume servicios de C).
- Reuso. Un mismo layer puede ser utilizado o consumido por layers para distintos propósitos.
- La performance, si no se tiene en cuenta en las decisiones arquitectónicas, puede ser un problema en este tipo de arquitecturas. El estilo layered estimula la creación de múltiples layers para obtener todos los beneficios que trae consigo el desacoplamiento de las distintas partes del sistema; pero hay que evitar el sobre-diseño de layers ya que puede traer más problemas que soluciones. Además, como se mencionó anteriormente, en ciertos casos es inevitable que la mantenibilidad "ceda un poco" para dar paso a una mejora de la performance.

A continuación mostraremos la evolución de este estilo arquitectónico. Comenzaremos desde el momento en que no tenían sentido pensar en layers para llegar a arquitectura n-layered, pasando por arquitectura conocidas como de 2 y 3 layers.

### *Arquitecturas que no necesitan layers*

Durante el desarrollo de sistemas batch, sistemas que sólo interactuaban con archivos y no tenían interacción hombre máquina, no existía la necesidad de layers.

### *Dos layers*

A partir de los años 90, comenzó a surgir la idea de layers con los sistemas cliente-servidor. Estos sistemas se consideraban “sistemas de dos layers”: el layer de cliente y el layer de servidor.

En los primeros sistemas dos layers, los clientes no solo implementaban la interface de usuario (GUI), sino también la lógica de la aplicación. El servidor, generalmente, se encargaba solo de la persistencia de los datos. Por lo general era una base de datos. Los datos, podían ser consultados y modificados desde el cliente utilizando algún lenguaje de consultas y de ABM (SQL por ejemplo).

***Figura 13 - Dos Layers***

Para sistemas que solo tenían que implementar operaciones de consultas, altas, bajas y modificaciones, este esquema funcionaba bien y era muy productivo. Muchos de los ambientes de desarrollo brindaban herramientas WYSIWIG (What You See Is What You Get) que facilitaban la construcción de los clientes en este tipo de arquitecturas. Por ejemplo, proveían la posibilidad de enlazar fácilmente los componentes de la GUI con campos de la tabla. Una vez configurado el enlace, sincronizaba el dato que mostraba con el valor que tenía el campo en la BD.

El problema surgió cuando se quiso utilizar la misma estrategia (aquella de tener la lógica implementada en el cliente), aún cuando esta lógica se volvió sumamente complicada. Una herramienta que fue diseñada para desarrollar GUI e interactuar con datos, era ahora utilizada para escribir reglas, validaciones y cálculos complejos. Como consecuencia, la lógica de las interfaces de usuarios comenzó a ser cada vez más compleja, con código replicado, difícil de mantener y de testear. Además, cuando surgió la idea de poder presentar la misma aplicación en distintos clientes (por ejemplo, en cliente de escritorio y cliente web), la solución era reescribir toda la lógica en ambos clientes, solución que implicaba replicar código y complicar el mantenimiento.

Para evitar algunos de los mencionados problemas, se buscaron alternativas en las cuales la lógica se escriba del lado del servidor y los clientes solo tengan que consumirla para dibujar las interfaces de usuarios.

La primera idea que surgió, considerando que en ese momento el servidor era en general una base de datos, fueron los stored procedures. Este esquema permitía sistemas más modulares que el anterior, pues estaba pensado para escribir la lógica del sistema y no para dibujar GUI's. Además, se resuelve el problema de múltiples clientes que necesitan la misma lógica.

Si bien se resuelven algunos de los problemas que presentaba la primera idea de arquitecturas de dos layers, el lenguaje ofrecido por las diferentes bases de datos es poco natural para expresar las complicadas reglas de negocios, validaciones y cálculos. Las bases de datos fueron diseñadas para persistir datos y no para escribir algoritmos complicados que consuman estos datos.

### *Tres layers*

Al mismo tiempo que se fue haciendo popular el estilo cliente-servidor, el paradigma orientado a objetos empezaba a nacer. Considerado como un paradigma muy expresivo y natural para diseñar e implementar la lógica de dominio, fue la causa del surgimiento de un nuevo layer que permita separar la lógica, de los datos. Este layer es denominado **Modelo de Dominio** [Eva03]**.**

No es obligatorio el uso del paradigma orientado a objetos en este layer, pero es el más utilizado, principalmente en Aplicaciones Enterprise en las cuales abundan validaciones, cálculos y reglas complicadas.

A continuación se describen las responsabilidades de cada una de las capas:

- **Layer de presentación:** Interfaces gráficas responsables de mostrar información e interpretar la interacción del usuario para ejecutar las operaciones correspondientes que serán quienes interactúen con la lógica de dominio. Es responsable de hacer validaciones básicas de cliente.
- **Layer de Modelo de Dominio:** Lógica de dominio. Es responsable de realizar validaciones de datos ingresados por el usuario y de ejecutar la lógica de dominio correspondiente, que en Aplicaciones Enterprise incluirá frecuentemente cálculos complejos y reglas del dominio complicadas.
- Layer de acceso a datos: Comunicación con el mecanismo de persistencia que se utilice. En general una base de datos. Si fuera una base de datos relacional, este layer será el responsable de realizar el mapeo objeto-relacional para lograr mapear los objetos en tablas y los atributos en campos de esas tablas.

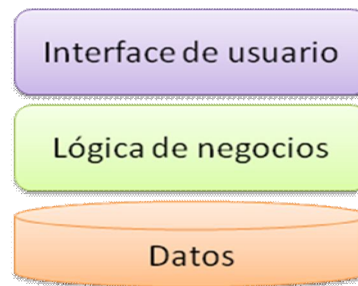


*Figura 14 - Tres Layers*

### *N layers*

A medida que los sistemas se tornaron más exigentes con sus arquitecturas, nuevos layers fueron surgiendo. Este surgimiento de nuevos layers permitió clarificar y distinguir la responsabilidad de cada elemento de una arquitectura en layers, además de reforzar el desacoplamiento necesario para favorecer el mantenimiento y extensión de las aplicaciones. Es así como surgen las arquitecturas n-layered.

Nuestro caso de estudio se centra en una arquitectura cliente-servidor la cual se puede catalogar actualmente dentro del tipo Layered (en capas).

## Arquitectura Cliente-Servidor

Cualquier entorno en donde un software solicite o reciba información de otro, podemos llamarla una arquitectura cliente-servidor. Esta arquitectura se basa en un cliente que realiza peticiones a otro programa, el servidor, el cual realiza el procesamiento necesario y genera una respuesta a la petición realizada. Aunque esta idea se puede aplicar a programas que se ejecutan sobre una sola computadora es más ventajosa en un sistema operativo multiusuario distribuido a través de una red [Cle02].

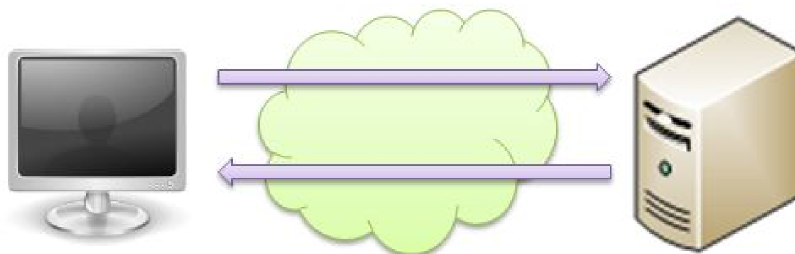

*Figura 15 - Arquitectura Cliente Servidor*

En esta arquitectura la capacidad de proceso está repartida entre los

clientes y los servidores, aunque son más importantes las ventajas de tipo organizativo debidas a la centralización de la gestión de la información y la separación de responsabilidades, lo que facilita y clarifica el diseño del sistema.

La separación entre cliente y servidor es una separación de tipo lógico, donde el servidor no se ejecuta necesariamente sobre una sola máquina ni es necesariamente un sólo programa. Los tipos específicos de servidores incluyen los servidores web, los servidores de archivo, los servidores del correo, etc. Mientras que sus propósitos varían de unos servicios a otros, la arquitectura básica seguirá siendo la misma [Har99].

Una disposición muy común son los sistemas multicapa en los que el servidor se descompone en diferentes programas que pueden ser ejecutados por diferentes computadoras aumentando así el grado de distribución del sistema.

La arquitectura cliente-servidor sustituye a la arquitectura monolítica en la que no hay distribución, tanto a nivel físico como a nivel lógico.

## Cliente

El componente cliente de una arquitectura cliente-servidor está representado actualmente por programas que requieren específicamente una conexión a otro programa, al que se denomina servidor y que suele ejecutarse en otra máquina.

Se utilizan para obtener datos o realizar procesamientos externos (por ejemplo páginas web, información bursátil o bases de datos), interactuar con otros usuarios a través de un gestor central (por ejemplo como lo hacen algunos servidores P2P), compartir información con otros usuarios (servidores de archivos y otras aplicaciones Groupware) o utilizar recursos de los que no se dispone en la máquina local (por ejemplo impresión)

Uno de los clientes más utilizados en la actualidad, sobre todo por su versatilidad, es el navegador web. Muchos servidores son capaces de ofrecer sus servicios a través de un navegador web en lugar de requerir la instalación de un programa específico.

### *Tipos de Clientes*

Existen varios tipos de clientes, dependiendo de capacidad de proceso que posea en comparación al servidor. Así podemos encontrar clientes pesados, clientes híbridos y clientes livianos

### Cliente pesado (rico)

Se denomina cliente pesado al programa "cliente" de una arquitectura cliente-servidor cuando la mayor capacidad de proceso está desplazada hacia la plataforma que ejecuta dicho programa. También se conoce como cliente rico (rich client). Un cliente pesado es la antítesis de un cliente liviano.

Un cliente pesado da el soporte necesario para proveer al usuario final una experiencia de alta calidad en determinado dominio, proveyendo interfaces de usuarios nativas de la plataforma en donde se ejecutan con una gran velocidad en el procesamiento local [Mca10].

Un cliente pesado tiene capacidad de realizar operaciones nativas tales como darg-and-drop, guardar recortes en portapapeles, navegación, personalización y almacenar y procesar datos; pero sigue necesitando las capacidades del servidor para una parte importante de sus funciones.

Un cliente de email suele ser un cliente pesado. Puede almacenar los mensajes de correo electrónico del usuario, trabajar con ellos y redactar nuevos mensajes, pero sigue necesitando una conexión al servidor para enviar y recibir los mensajes.

### Cliente híbrido

Un cliente híbrido no tiene almacenados los datos con los que trabaja, pero sí es capaz de procesar datos que le envía el servidor. Muchos programas de colaboración almacenan remotamente los datos para que todos los usuarios trabajen con la misma información, y utilizan clientes híbridos para acceder a esa información.

### Cliente liviano

Un cliente liviano es una computadora o un software en una arquitectura cliente-servidor que depende primariamente del servidor para las tareas de procesamiento, y se enfoca principalmente en transportar la entrada y la salida entre el usuario y el servidor. En contraste, un cliente pesado hace tanto procesamiento como sea posible y pasa solamente los datos para las comunicaciones y el almacenamiento al servidor.

Un cliente liviano no tiene capacidad de procesamiento y su única función es recoger los datos del usuario, dárselos al servidor, y mostrar su respuesta. Los primeros navegadores web eran clientes livianos, simplemente mostraban las páginas web que solicitaba el usuario.

Actualmente, el uso de lenguajes de scripting [Fla01], junto con las nueva especificación HTML5 dan a los navegadores una gran capacidad de procesamiento, por lo que pueden llegar a considerarse clientes Híbridos, y en muchos casos llegar a ser Clientes pesados [Hog11].

### Servidor

El componente servidor es el encargado de proveer servicios al componente cliente. Si bien a lo largo de este trabajo tomamos a la componente servidor como un unidad, esto no implica que dicha componente esté a su vez definida sobre alguna arquitectura, por ejemplo de N layers, la cual es la más utilizada en las Aplicaciones Enterprise.

Si bien éste es el significado original del término, es posible que un ordenador cumpla simultáneamente las funciones de cliente y de servidor.

Se conoce como servidor también a una computadora en la que se ejecuta un programa que realiza alguna tarea en beneficio de otras aplicaciones llamadas clientes, tanto si se trata de un mainframe, una computadora personal, una PDA o un sistema integrado; sin embargo, hay computadoras destinadas únicamente a proveer los servicios de estos programas: estos son los servidores por antonomasia.

### Ventajas y Desventajas

A continuación se enumeran las ventajas e inconvenientes que presentan las arquitecturas cliente-servidor.

#### *Ventajas*

<u>Centralización del control</u>

Con este tipo de arquitectura los accesos, recursos y la integridad de los datos son controlados por el servidor de forma que un programa cliente defectuoso o no autorizado no pueda dañar el sistema. Esta centralización también facilita, por ejemplo, la tarea de poner al día datos u otros recursos.

<u>Buena Escalabilidad</u>

Esto se debe a que se puede aumentar la capacidad de clientes y servidores por separado. Cualquier elemento puede ser aumentado (o mejorado) en cualquier momento, o se pueden

añadir nuevos nodos a la red (clientes y/o servidores). Sumado a esto, la arquitectura modular de los sistemas cliente-servidor nos permite el uso de ordenadores especializados (servidores de base de datos, servidores de archivos, etc.).

Red más fiable

La existencia de varias computadoras proporciona una red más fiable ya que un fallo en uno de los equipos no significa necesariamente que el sistema deje de funcionar.

Fácil mantenimiento (o encapsulación)

Dado que las funciones y responsabilidades están distribuidas entre varias computadoras independientes, es posible reemplazar, reparar, actualizar, o incluso trasladar un servidor, mientras que sus clientes no se verán afectados por ese cambio (o se afectarán mínimamente).

Soporte tecnológico

En la actualidad existe una amplia gama de tecnologías, suficientemente desarrolladas, diseñadas para el paradigma de cliente-servidor que aseguran la seguridad en las transacciones, la amigabilidad de la interfaz de usuario, y la facilidad de empleo (usabilidad).

Menor costo de operación

Permiten un mejor aprovechamiento de los sistemas existentes, protegiendo la inversión. Por ejemplo, la compartición de servidores (habitualmente caros) y dispositivos periféricos (como impresoras) entre máquinas clientes permite un mejor rendimiento del conjunto. Además proporcionan un mejor acceso a los datos. La interfaz de usuario ofrece una forma homogénea de ver el sistema, independientemente de los cambios o actualizaciones que se produzcan en él y de la ubicación de la información. Se puede agregar también, que a causa del movimiento de funciones desde un ordenador central hacia servidores o clientes locales origina el desplazamiento de los costes de ese proceso hacia máquinas más pequeñas y por tanto, más baratas.

Mayor número de usuarios

Las arquitecturas cliente-servidor eliminan la necesidad de mover grandes bloques de información por la red hacia las computadoras personales o estaciones de trabajo para su procesamiento. Los servidores controlan los datos, procesan

peticiones y después transfieren sólo los datos requeridos a la máquina cliente. Entonces, la máquina cliente presenta los datos al usuario mediante interfaces amigables. Todo esto reduce el tráfico de la red, lo que facilita que pueda soportar un mayor número de usuarios.

### *Desventajas*

Congestión del tráfico

Cuando una gran cantidad de clientes envían peticiones simultáneas al mismo servidor, puede ser que cause muchos problemas para éste (a mayor número de clientes, más problemas para el servidor). A veces, los problemas de congestión de la red pueden degradar el rendimiento del sistema por debajo de lo que se obtendría con una única máquina (arquitectura centralizada). También la interfaz gráfica de usuario puede a veces ralentizar el funcionamiento de la aplicación.

Cuellos de botella

Un servidor tiene que responder peticiones de muchos clientes a la vez. En ocasiones se puede dar que varios de esos clientes requieran el mismo recurso, por ejemplo el acceso a la base de datos. En estos casos se produce un cuello de botella ya que el flujo normal del sistema se ve interrumpido. A esto se suma un problema y es que cuando un servidor está caído, las peticiones de los clientes no pueden ser satisfechas.

Servidores potentes

El software y el hardware de un servidor son generalmente muy determinantes. Un hardware regular de una computadora personal puede no poder servir a cierta cantidad de clientes. Normalmente se necesita software y hardware específico, sobre todo en el lado del servidor, para satisfacer el trabajo. Por supuesto, esto aumentará el coste.

Complejidad tecnológica

Hay una alta complejidad tecnológica al tener que integrar una gran variedad de productos. Es más difícil también asegurar un elevado grado de seguridad en una red de clientes y servidores que en un sistema con un único ordenador centralizado.

# *Capítulo 4*

# Transferencia del modelo de Entidades

*Los objetos son abstracciones de entidades reales del mundo que nos rodea o de elementos internos del sistema. En las Aplicaciones Enterprise muchas de estas entidades son persistentes y en un gran número de ocasiones deben ser transferidas entre el cliente y el servidor. En este capítulo nos centramos en las soluciones conceptuales y tecnológicas que nos permitirán realizar esta tarea de la mejor manera.*

Basándonos en las necesidades que nos obligan a trabajar con objetos persistentes en el Cliente de una Aplicación Enterprise y teniendo en mente los problemas que esto conlleva, realizamos un análisis conceptual y tecnológico que tiene como objetivo mostrar y analizar las distintas alternativas que encontramos disponibles, cuáles son sus aportes, soluciones y falencias, y así poder justificar la necesidad de un nuevo enfoque que de una solución integral a los problemas recurrentes encontrados en el desarrollo de Aplicaciones Enterprise.

Si nos enfocamos en el manejo de objetos persistentes en la componente cliente de Aplicación Enterprise, nos encontramos con dos temas fundamentales los cuales necesitamos desarrollar, y ellos son:

- La manera en la cual la información contenida en los objetos persistentes, es transferida entre el cliente y el servidor.
- Las tecnologías que nos encontramos hoy día para la implementación de la componente cliente, sea éste cualquiera de los tres tipos descriptos.

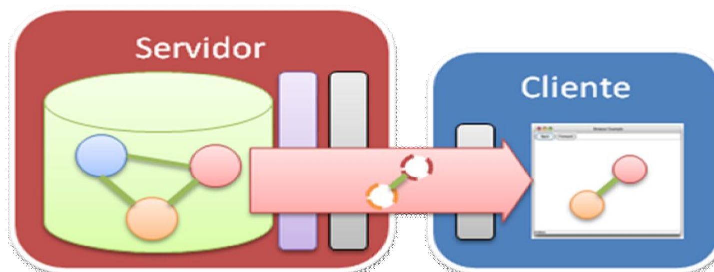


***Figura 16 - Objetos persistentes en la componente cliente***

## Modelo de Dominio

Trabajar con objetos nos ayuda a dar soluciones a las lógicas complejas de las Aplicaciones Enterprise. Un modelo de dominio crea una red de objetos

interconectados, donde cada objeto representa una entidad significativa, de gran tamaño, como puede ser toda una empresa, o tan chico como la línea de factura.

Contar con un modelo de dominio en una aplicación consiste en insertar una capa de objetos a nuestra arquitectura, que modele el área de negocio con la cual estamos trabajando. Desde esta visión podemos encontrar dentro de este domino, objetos que representan datos del negocio y objetos que representan reglas del negocio [Fow02].

En términos de tipos de elementos del modelo de dominio, podemos distinguir ciertos patrones sobre ellos [Eva03], como ser:

- Los objetos con identidad propia que representan algo en la continuidad del tiempo (Entidades),
- Los objetos que representan estados o algún otro valor (Value Object)
- Los objetos que modelan aquellos aspectos del sistema que son expresados como acciones u operaciones (Servicios).

Debido a que el comportamiento y la lógica de las empresas pueden sufrir cambios permanentemente, es importante poder modificar, construir y probar esta capa con facilidad. En consecuencia debemos minimizar el acoplamiento del modelo de dominio con el resto de las capas del sistema. Existen gran cantidad de patrones de arquitectura que nos permite mantener esta capa lo más desacoplada posible.

Al contar con un modelo de dominio, podemos utilizarlo en distintos escenarios. El caso más simple es una aplicación mono usuario, que lea de un archivo todo el grafo de objetos y lo cargue en memoria. Una aplicación de escritorio puede trabajar de esta manera, pero no es común debido a la gran cantidad de objetos que esta puede llegar a manejar.

El uso de las bases de datos orientadas a objetos nos permite abstraernos del uso de la memoria, dándonos la sensación de estar trabajando con todo el grafo de objetos en memoria, mientras que los objetos están en continuo movimiento entre la memoria y el disco. Sin una base de datos orientado a objetos, esto se debe realizar de manera manual o utilizando alguna de las herramientas disponibles para esto, como por ejemplo utilizar bases de datos relacionales e interactuar con ellas utilizando alguna herramienta de mapeo Objeto Relacional. [Kin04].

## Distribución del Modelo de Dominio

Si bien nuestro trabajo no se centra en los mecanismos de envío de información a través de redes, sí nos interesa dar una introducción a la distribución de objetos, sus características y sus problemas.

El poder de la distribución de objetos no está en el hecho de que un grupo de objetos se encuentran dispersos en toda la red, sino en que cualquier “agente” en el sistema puede interactuar directamente con un objeto que “vive” en un host remoto. En nuestro caso, el cliente de nuestra arquitectura puede interactuar con un objeto que vive en el servidor.

Hablaremos de manera resumida sobre las motivaciones de la distribución de objetos, lo que hace que los sistemas de objetos distribuidos sean tan útiles, y lo que constituye un "buen" sistema de objetos distribuidos.

A la hora de desarrollar aplicaciones distribuidas nos gustaría distribuir los datos y funciones de nuestro sistema, definiéndolos en base a la estructura y necesidades del mismo, en lugar de las características y restricciones que nos agrega la distribución en sí. Los sistemas de distribución de objetos tratan de resolver estos problemas y permitir a los desarrolladores tomar sus objetos y hacer que se "ejecuten" en un host remoto en lugar de la máquina local.

El objetivo de la mayoría de los sistemas de distribución de objetos es permitir que cualquier objeto resida en cualquier parte de la red, y que una aplicación pueda interactuar con estos objetos de la misma manera como lo hace con un objeto local. Algunas características adicionales que podemos mencionar son la capacidad de construir un objeto en un host y transmitirlo a otro, y la capacidad de que un agente en un host pueda crear un objeto nuevo en otro host.

Sabemos de la existencia de librerías (como por ejemplo los paquetes java.net y java.io en Java) que nos proveen un fácil acceso a la red y a los protocolos de red, y nos permiten además contar con una capa de abstracción para las operaciones específicas de nuestra aplicación. Parece que todo lo que tendríamos que hacer es extender estas librerías para permitir que los objetos invoquen métodos sobre la red, y así tener un sistema básico de objetos distribuidos. Dicho de esta manera parece simple, pero para darnos una idea de la complejidad de los sistemas de objetos distribuidos, pensemos solamente lo que implicaría pasar un parámetro por referencia.

En general, en los sistemas en los que se utilizan objetos distribuidos, se cuenta con:

- Una interfaz para crear instancias de la clase de los objetos en el host remoto
- Una interfaz entre la implementación del objeto y un administrador de objetos, a veces llamado skeleton
- Una interfaz para el cliente del objeto, a veces llamado objeto stub

El skeleton será utilizado por el servidor para crear nuevas instancias del objeto y enviar la invocación de los métodos a la implementación remota del

mismo. El objeto stub será utilizado por el cliente para invocaciones de métodos del objeto en el servidor.

Además de los objetos que intervienen en la distribución, existe un registro en donde cada instancia generada es registrada para que el cliente pueda obtener el *stub* correspondiente y hacer uso con el objeto distribuido.

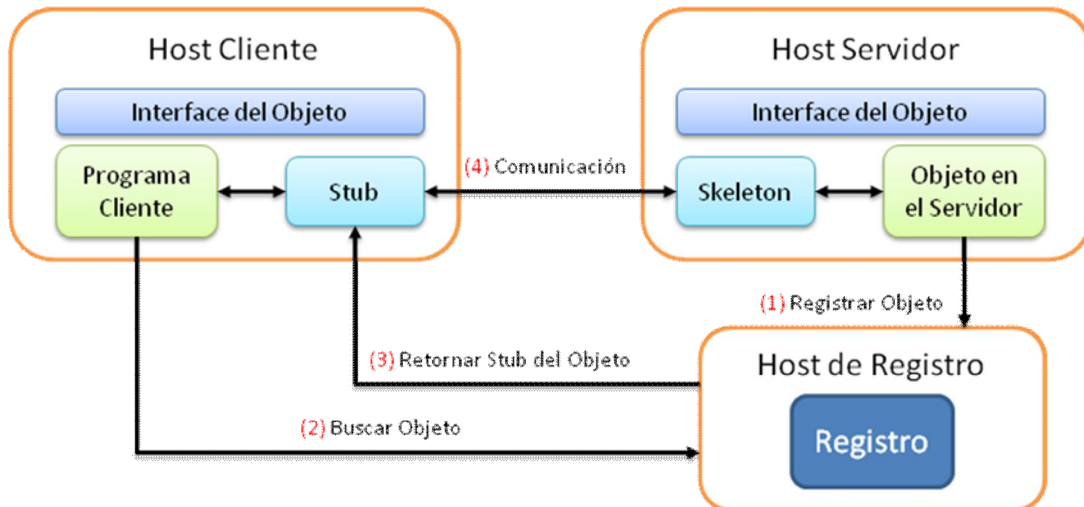


***Figura 17 - Objetos distribuidos***

## Escenarios para la distribución de Objetos

En algún momento, durante el diseño de cualquier arquitectura distribuida nos encontramos con el dilema de cómo distribuir nuestra aplicación. Existe la idea de que se pueden tomar un montón de objetos y distribuirlos a gusto en los distintos nodos de procesamiento. Además los middlewares existentes nos proveerán la transparencia necesaria para que los objetos se invoquen entre sí, sin tener en cuenta si se encuentran en el mismo proceso, en otro proceso o en otra máquina.

La transparencia en la distribución de objetos es valiosa, pero al mismo tiempo puede ocultar algunas cuestiones importantes como ser el rendimiento de nuestro sistema, además de volverlo difícil de construir e instalar.

### Interfaces locales y remotas

La principal razón por la que el modelo de distribución previamente mencionado no funciona, tiene relación directa con un hecho fundamental en las computadoras. Un llamado dentro de un mismo proceso es varios órdenes de magnitud más rápido que un llamado entre dos procesos distintos; sumándole a esto varios órdenes de magnitud más si la ejecución

de los procesos se realiza en distintas maquinas.

Para satisfacer un requerimiento del cliente, en la mayoría de los casos se deben realizar varias invocaciones a la interface remota del objeto distribuido, lo cual incrementa el tiempo de respuesta sobre los niveles aceptables.

Aunque muchas plataformas de objetos distribuidos reducen la complejidad de hacer una llamada remota, no pueden eliminar los pasos que se requieren para que la comunicación tenga lugar. Por ejemplo, debemos localizar los objetos remotos, y para esto debemos realizar una conexión al equipo remoto antes de que los datos se pueden serializar en un flujo de bytes, posiblemente deban ser cifrados y luego transmitidos a la computadora remota.

Cuando se considera el rendimiento de las redes, hay que analizar tanto la latencia, como el rendimiento. En términos simplificados, la latencia se describe el tiempo que transcurre antes de que el primer byte de datos llegue al destino. El rendimiento describe la cantidad de bytes de datos que se envían a través de la red dentro de un período de tiempo determinado (por ejemplo, un segundo). En las redes modernas la latencia puede ser un factor más importante que el rendimiento. Eso significa que puede tomar casi la misma cantidad de tiempo transmitir 10 bytes que transmitir 1.000 bytes de datos. Este efecto es especialmente pronunciado cuando se utiliza protocolos de conexión sin estado, como HTTP. En redes más rápidas a menudo se puede aumentar el rendimiento, pero la latencia es mucho más difícil de reducir.

Como resultado de esto, la interfaz para un objeto remoto debe ser diferente de la utilizada para un objeto local del mismo proceso. Al diseñar la interfaz de un objeto, son buenas prácticas ocultar gran parte de la información y proporcionar un conjunto de métodos de grano fino para acceder y manipular la información. Grano fino significa que cada método debe ser responsable de una sola pieza, bastante pequeña, y con una funcionalidad atómica. Este enfoque simplifica la programación, proporciona una mejor abstracción de las propiedades de los objetos y aumenta el potencial para su reutilización. Esto debe ser equilibrado contra el hecho de que el uso de métodos de grano fino implica más invocaciones para realizar una tarea de alto nivel.

Por lo general, las llamadas a funciones extras es aceptable cuando los métodos se invocan en el mismo proceso, sin embargo, la sobrecarga puede llegar a ser grave cuando estos métodos se invocan entre procesos a través de la red.

La mejor manera de evitar problemas de latencia, que son inherentes a las llamadas remotas, es hacer menos llamadas y pasar más datos con cada llamada. Una forma de lograr esto es declarar el método remoto con una larga lista de parámetros. Esto permite al cliente pasar más información al

componente remoto en una sola llamada. Hacer esto hace la programación propensa a errores. Por ejemplo, si un método remoto acepta 10 parámetros del mismo tipo, es fácil pasar argumentos en el orden equivocado. El compilador no será capaz de detectar este tipo de errores.

Además, una larga lista de parámetros no ayuda a devolver más información al cliente desde la llamada remota, porque la mayoría de lenguajes de programación limitan el tipo del valor devuelto a un solo parámetro. Coincidentemente, el valor de retorno es a menudo en donde se transmiten la mayoría de los datos.

Un ejemplo de lo anterior sería tener una entidad Dirección y una buena interfaz tendrá métodos separados para obtener la ciudad, la provincia, asignar la ciudad, asignar la provincia, etc. Una interfaz de grano fino sigue el principio general de la Orientación a Objetos de muchas piezas pequeñas que puedan ser combinadas y sobrescritas de varias maneras para extender el diseño en el futuro. Una interfaz de grano fino no funciona bien cuando es remota. Cuando las llamadas a métodos son lentas, uno quiere obtener o actualizar la ciudad, estado y código postal en una única llamada en lugar de tres. La interfaz resultante es entonces de grano grueso, diseñado no para la flexibilidad y extensibilidad, sino para reducir al mínimo las llamadas remotas.

Como primera conclusión podemos tomar que, cualquier objeto que pueda ser utilizado de forma remota debe tener una interfaz de grano grueso, mientras que todos los objetos que no se utilizan de manera remota deben tener una interfaz de grano fino.

### Invocaciones interprocesos

Algo que también debemos tomar en cuenta es que solo hay que pagar el precio de una llamada interproceso solo cuando ésta sea realmente necesaria. Por esta razón no se puede simplemente llevarse a un modelo distribuido un grupo de clases diseñadas para vivir en un único proceso. El diseño de la distribución es más que eso.

Si nos basamos en la estrategia de distribución en clases, se terminará con un sistema que realiza una gran cantidad de llamadas remotas y por lo tanto necesita interfaces de grano grueso, que van en contra del paradigma de Orientación a Objetos. Al final, incluso con interfaces de grano grueso en cada clase que pueda ser remota, igualmente se terminará con demasiadas llamadas remotas y un sistema que será difícil de modificar.

Por lo tanto, se puede decir que la primera ley del diseño de objetos distribuidos es: “No distribuir los objetos”.

## ¿Dónde hay que distribuir?

### Aplicaciones con bases de datos

Una primera separación se da desde la aparición de las bases de datos. La aplicaciones necesitan recuperara información de una base de datos para realizar el procesamiento necesario.

Las bases de datos generalmente se encuentran ejecutándose en un proceso distinto del de la aplicación. Si bien es posible eliminar la distribución y ejecutar toda la lógica en procesos almacenados de base de datos, a menudo eso no es práctico, debido a las limitaciones del lenguaje utilizado para programar los procesos almacenados, por lo que es necesario contar con procesos implementados en otro lenguaje de programación, y por ende distribuidos.

Quizá se pueden ejecutar en la misma máquina, pero una vez que haya procesos separados inmediatamente vamos a tener que pagar el costo de las llamadas remotas. Afortunadamente, SQL está diseñado como una interfaz remota, por lo que de alguna manera se minimiza este costo.

### Arquitecturas Cliente Servidor

Si bien la mayoría de las aplicaciones de escritorio mono usuario pueden ejecutarse íntegramente en un mismo proceso, esto se vuelve imposible en aplicaciones multiusuario, las cuales están basadas en una arquitectura cliente-servidor.

Este tipo de aplicaciones, dentro de las cuales están el tipo de aplicaciones tratadas en el presente trabajo, cuentan generalmente con un servidor, el cual se encarga de realizar la mayor parte del procesamiento, y con una serie de clientes, con más o menos comportamiento (clientes livianos o ricos) los cuales son los que proveen al usuario el acceso a la aplicación, y utiliza los servicios o procesos provistos por el servidor.

En una aplicación Cliente Servidor con clientes ricos, con las características previamente mencionadas en el Capitulo 3, es necesario proveer los mecanismos necesarios para que el cliente de la aplicación interactúe con el modelo de domino. Esta interacción será necesariamente distribuida, debido a las características inherentes de la arquitectura. Más adelante daremos distintas alternativas para poder cumplir con este requerimiento fundamental en este tipo de aplicaciones.

### Servidores Web y de Aplicaciones

Otra separación en procesos puede ocurrir en sistemas web, entre el servidor Web y el servidor de Aplicaciones. En igualdad de condiciones es mejor ejecutar el servidor web y el de aplicaciones en un único proceso.

Puede que haya que separarlos debido a las diferencias tecnológicas entre los servidores, como por ejemplo al utilizar paquetes de software diferentes, lo que a menudo se ejecutan en su propio proceso, por lo que de nuevo estamos distribuyendo.

### Límites explícitos de distribución

Si bien al diseñar un sistema tenemos que limitar los alcances de la distribución al máximo, y tenerla en cuenta en los lugares estrictamente necesarios, todavía podemos diseñar un sistema distribuido utilizando objetos de grano fino. La clave está en usarlos internamente y colocar objetos de grano grueso en los límites de distribución, cuya única función es proporcionar una interfaz remota para controlar los objetos de grano fino.

Los objetos de grano grueso en realidad no hacen nada más que actuar como una fachada para los objetos de grano fino y están allí sólo para fines de distribución.

Sólo los objetos que realmente necesitan un servicio remoto obtienen la fachada, perdiendo transparencia y haciendo explícito a los desarrolladores que están pagando el costo de la llamada remota.

## Alternativas Conceptuales

Como mencionamos previamente, no nos detendremos en detalle de cómo realizar la distribución de objetos. A partir de este momento asumimos que tenemos resuelto, mediante alguna de las tecnologías existentes, la distribución de las fachadas remotas y nos enfocamos ahora en buscar la mejor manera de trabajar con objetos persistentes en el cliente de Aplicaciones Enterprise.

### Objetos de transferencia - Data Transfer Object

De la mano de las fachadas remotas llegan los Data Transfer Objects (DTO). No sólo se necesitan métodos de grano grueso, también es necesario transferir objetos de grano grueso. Cuando se pide una dirección, se tiene que enviar esa información en un solo bloque. Por lo general, no se puede enviar el objeto de dominio en sí mismo, porque está atado en una red de

referencias locales de grano fino entre objetos. Entonces se toman todos los datos que el cliente necesite y se empaquetan en un objeto en particular para la transferencia.

Los DTO aparecen en ambos extremos, así que es importante que no se referencie nada que no se comparta por la red. Esto se reduce al hecho de que un DTO por lo general sólo hace referencia a otros DTOs y a objetos primitivos tales como enteros.

Un DTO es básicamente un objeto creado exclusivamente para transportar datos. Datos que pueden tener su origen en una o más entidades de información del dominio. Estos datos son incorporados a un objeto que puede ser pasado a través de la aplicación, localmente o, lo que es más importante, puede ser serializada y enviada a través de la red, de forma que los clientes puedan acceder a información del modelo desde un solo objeto y mediante una sola llamada.

Un DTO normalmente no provee lógica de negocios o validaciones de ningún tipo. Sólo provee acceso a las propiedades del objeto que lo implementa. Algunos autores remarcan que este objeto debe ser inmutable, dado que sus cambios no deben reflejarse en el sistema.

Beneficios

- Reducción del número de llamadas remotas - Al transmitir más datos en una sola llamada remota, la aplicación puede reducir el número de llamadas.
- Mejora del rendimiento - Las llamadas remotas pueden volver lenta una aplicación de manera drástica. La reducción del número de llamadas es una de las mejores maneras de mejorar el rendimiento.
- Grano grueso - Pasar más datos de ida y vuelta en una sola llamada oculta el comportamiento interno de una aplicación remota, detrás de una interfaz de grano grueso.
- Capacidad de prueba - Encapsular todos los parámetros en un objeto que se pueda serializar facilita y mejora las pruebas.

Contras

- Posible explosión de clases - Si se decide utilizar DTO’s con tipos, puede que tenga que crear uno (o dos, si se considera el valor de retorno) DTOs por cada método remoto. Incluso en una interfaz de grano grueso, esto podría conducir a un gran número de nuevas clases. Esta cantidad de clases puede ser difícil de codificar y gestionar. El uso de generación automática de código puede aliviar parte de este problema.
- Cálculo adicional - La acción de traducir de un formato de datos

en el servidor a un flujo de bytes que pueda ser transportado a través de la red y de nuevo en un formato de objeto dentro de la aplicación cliente, puede introducir una buena cantidad procesamiento extra. Normalmente, se agregan los datos de varias fuentes a un único DTO en el servidor, con lo que son necesarios cálculos adicionales en cada extremo para agregar y serializar la información.

- Esfuerzo adicional de codificación - Pasar parámetros a un método se puede hacer en una sola línea. El uso de DTO's requiere una instancia de un objeto nuevo e inicializar cada una de sus propiedades. Este código puede volverse tedioso de escribir.

## Objetos de dominio desconectados - Detach Object

Otro camino para la distribución, manteniendo la premisa de que es poco eficiente la distribución de grano fino, es decir, de cada uno de los objetos que conforman el modelo de domino; es tener un agente que migre objetos entre los procesos cliente y servidor. De esta manera contaremos con el modelo de domino en ambas componentes de la aplicación, reduciendo la necesidad de nuevos objetos destinados a este pasaje de información, y a la actualización de la misma, ya que el cliente estaría interactuando con el modelo de dominio de la misma manera que lo hace el servidor.

Este proceso se ve levemente simplificado si las tecnologías y plataformas utilizadas para desarrollar el cliente y el servidor de la aplicación son los mismos. De todas maneras, de esta idea se deprenden rápidamente algunos puntos en donde debemos poner el foco, como ser por ejemplo la profundidad de los datos enviados, el control del acceso concurrente de distintos clientes, etc.

Centrándonos en objetos del modelo de dominios persistentes, se suma la dificultad de tener que enviar al cliente objetos con los cuales normalmente trabajamos dentro de una sesión o transacción para que los cambios realizados sobre los mismos sean propagados al repositorio en donde son persistidos.

Justamente el nombre elegido para este tipo de objetos es Detach Object, debido a que el contexto utilizado para recuperarlos del repositorio ha finalizado, pero aún en estas circunstancias, pueden ser modificados. Las referencias a estos objetos siguen siendo válidas, es decir siguen siendo objetos persistentes, lo que permite poder ser nuevamente asociados en a un nuevo contexto, transacción o sesión para que todas las modificaciones realizadas fuera del mencionado contexto sean persistidas.

Esto nos permite mantener largas Transacciones de negocios, es decir,

largas unidades de trabajo desde el punto de vista del usuario.

**DTO's vs Detach Object**

La capa de presentación debe manipular la información que le provee la capa de negocio de manera que el usuario pueda interactuar con la misma (consultar, modificar, etc).

Para ello hay dos enfoques:

- Que la capa de presentación interactúe directamente con la capa de dominio, sin mediar ninguna abstracción.
- Que la capa de presentación le pida a un servicio de negocio que provea la información que debe manipular. El servicio de negocio retorna dichos objetos, que denominamos 'de transferencia' con información proveniente de objetos de dominio. Luego la capa de presentación modifica estos objetos de transferencia y llama a un servicio de negocio para que estos cambios sean persistidos.

Ambas opciones tienen las siguientes ventajas y desventajas

<u>Objetos de dominio</u>

- Facilidad de desarrollo - No hay que crear ninguna clase adicional.
- Transparencia / División en capas - Se está exponiendo el modelo de dominio a la capa de presentación, por lo cual cualquier cambio en ellos impacta en la presentación.
- Performance - Puede resultar poco eficiente trabajar con la granularidad del modelo de dominio al transportar los datos entre capas.

<u>Objetos de transferencia</u>

- Facilidad de desarrollo
    - Implementación estática - Requiere la generación de las clases correspondientes a estos objetos y el código para transferir la información entre ellos y los de dominio. El programador de la capa de presentación tiene una interfaz estática con la cual trabajar, delegando al compilador la tarea de chequear tipos y accesos. El chequeo estático también permite garantizar que las herramientas de refactorización de código de los entornos de desarrollos no rompan esta relación (ya que corrigen todas las ocurrencias o usos

de estos objetos).

- o Implementación dinámica - No hay que generar las clases pero se sigue necesitando el código de población. Se pierde el chequeo estático de tipos e interfaces realizados por el compilador.

- o Transparencia / División en capas - Se mantiene desacoplada la capa de presentación de la de negocio. La capa de negocio 'depende' de la capa de presentación. Dado que las copias entre los objetos de dominio y transferencia se realizan en la capa de negocio. Cualquier cambio de requerimientos que implique introducir nuevos campos en un formulario, requerirá la reescritura de los DTOs estáticos (si los hubiera) y del código de población.
- o Performance - Se puede optimizar las transferencias de datos entre capas.

## DTO dinámicos

Esta estrategia de envío de objetos del modelo de dominio al cliente es una variación de los DTOs.

Como ya mencionamos, uno de los inconvenientes principales de los DTOs es su descripción y su correspondiente código de población, y que el contenido de la interface está dirigido por el servidor en donde cualquier cambio de interface en el cliente que requiera datos adicionales requiere de cambios en los DTOs y sus populadores.

La generación de los DTOs se puede evitar usando un modelo dinámico, como por ejemplo utilizar un diccionario de propiedades. Esto todavía no resuelve el problema de la población, el cual se puede simplificar planteando un mecanismo de población utilizando un esquema declarativo para realizar esto de manera genérica.

A partir de este momento llamaremos DyTO (Dynamic Transfer Object) a esta solución basada en DTOs dinámicos.

# *Capítulo 5*

# Análisis del Problema

*Analizaremos las necesidades que deben ser satisfechas al trabajar con objetos persistentes en el cliente de Aplicaciones Enterprise, y analizaremos los problemas con los que nos encontramos al intentar satisfacerlos.*

La propuesta central es resolver la transferencia de las entidades entre las distintas capas de la arquitectura de una Aplicación Enterprise, dando soporte a las necesidades previamente descriptas. No está demás mencionar que esto resuelve un problema tecnológico, y no se origina de los requerimientos funcionales del sistema que se quiera desarrollar.

## Concurrencia

### Concurrencia en memoria

Siempre que exista un procesamiento realizado en un sistema, esto ocurre en un **contexto**, y normalmente en más de uno y de manera simultánea. En los tipos de sistemas que nos interesan, donde las interacciones entre el usuario y el sistema son prolongadas, es común el uso de **sesiones** como contextos de ejecución.

En teoría cada sesión debería tener una relación exclusiva con un **proceso** durante todo su tiempo de vida. Dado que los procesos se encuentran aislados apropiadamente entre ellos, esto nos ayudaría a reducir los conflictos de concurrencia. Una alternativa similar es iniciar un nuevo proceso por cada **requerimiento**, que fue, por ejemplo, la manera de trabajar de los primeros sistemas Web implementados utilizando CGI. Estas ideas son descartadas debido a que inicializar un proceso consume muchos recursos, pero a pesar de esto, es común que se trabaje de esta manera en sistemas donde los procesos deben manejar solo un requerimiento a la vez.

Lo discutido hasta el momento asume que estamos realizando el procesamiento en un único contexto de ejecución; pero desde que comenzamos a trabajar con **bases de datos,** aparece otro contexto de ejecución importante, las **transacciones**. Las transacciones ponen juntos todos los requerimientos que el cliente quiera tratar como si estuvieran en un único requerimiento. Esto puede ocurrir desde la aplicación hacia la base de datos (una *transacción de sistema*) o desde el usuario hacia una aplicación (una *transacción de negocios*). [Mar83]

## Manejo de Concurrencia

Para poder comprender en profundidad este tema debemos familiarizarnos con algunos conceptos generales de concurrencia, por lo tanto comenzaremos repasándolos. No pretendemos dar un tratamiento detallado sobre la concurrencia dentro del desarrollo de software; lo que haremos es dar una introducción a los temas de concurrencia en el desarrollo de Aplicaciones Enterprise.

Para profundizar este tema recomendamos leer algunos de los textos especializados en este tema enumerados en las referencias.

## Problemas esenciales de la concurrencia

Comenzaremos revisando los problemas esenciales de la concurrencia. Los llamamos esenciales porque son aquellos que los sistemas de control de concurrencia intentan prevenir. Estos problemas no son solamente de concurrencia, sino que además se suman aquellos que el mismo mecanismo de control crea en sus soluciones.

Perdidas de actualizaciones - “Lost Updates”:

> Es una idea simple de entender. La última actualización sobrescribe otras actualizaciones, lo que resulta en la pérdida de datos.
>
> Por ejemplo, dos editores copian electrónicamente el mismo documento. Cada editor edita la copia de forma independiente y luego guarda la copia modificada, con lo cual se sobrescribe el documento original. El último editor en guardar la copia modificada sobrescribe los cambios realizados por el primer editor.
>
> *Este problema podría evitarse si el segundo editor no pudiera realizar cambios hasta el primer editor haya terminado.*

Análisis inconsistente - ”Lecturas no repetidas”

> El análisis inconsistente se produce cuando se accede repetidas veces a la misma información y en cada lectura se obtienen diferentes resultados. Este problema es similar al que comentaremos a continuación (dependencia no confirmada), y se produce cuando se están modificando al mismo tiempo los datos que se están leyendo.
>
> Sin embargo, en el Análisis inconsistente, los datos leídos por la segunda operación ya terminaron de actualizarse. Además, del análisis inconsistente implica varias lecturas (dos o más) de la

misma información y en cada uno la información es diferente, de ahí el termino Lecturas no repetidas.

Por ejemplo, un editor lee el mismo documento dos veces, pero entre cada lectura, el escritor sobrescribe el documento. Cuando el editor lee el documento por segunda vez, este ha cambiado. La lectura original no es repetible.

*Este problema puede ser solucionado si el editor podría leer el documento solo después de que el escritor haya terminado.*

La dependencia no confirmada - “Lectura sucia”

Este problema se produce cuando una segunda operación lee una porción de información que está siendo actualizada concurrentemente por otra operación. La segunda operación está leyendo información que no se ha terminado de modificar aún y puede ser modificada nuevamente.

Por ejemplo, un editor está realizando cambios a un documento electrónico. Durante los cambios, un segundo editor toma una copia del documento que incluye todos los cambios realizados hasta el momento, y distribuye el documento a la audiencia. El primer editor entonces decide que los cambios realizados hasta el momento están mal entonces los elimina y guarda el documento. Los documentos distribuidos contienen ediciones que ya no existen, y que deberían ser tratados como si nunca hubieran existido.

*Este problema se puede eliminar si no se permite leer la información hasta que la primera operación finalice de modificarla.*

Lecturas fantasmas - “Phantom reads”

Esto se produce cuando se agregan o eliminan datos a la información que es leída por otra operación. La primera operación lee la información y la muestra, y dicha información contiene datos que pueden haber sido borrados, o no contiene datos agregados por otra operación.

Por ejemplo, un editor realiza cambios a un documento almacenado por un escritor, pero cuando los cambios son incorporados en la copia original del documento por el departamento de producción, estos encuentran que existe nuevo material en el documento agregado por el autor.

*Este problema podría ser eliminado si nadie podría agregar nuevo material al documento hasta que el editor y el departamento de producción terminen de trabajar con el*

*documento original.*

Cada uno de estos problemas provocan un fallo en la **corrección** (o **seguridad -** *safety*), y dan lugar a un comportamiento incorrecto que no se hubiera producido sin dos personas intentando trabajar con los mismos datos al mismo tiempo. Sin embargo, si la corrección fuera el único problema, estos problemas no serían tan graves. Después de todo, podemos arreglar las cosas de manera que sólo uno de los dos pueda trabajar con los datos a la vez.

Aunque esto ayuda con la corrección, reduce la **vivacidad** (*liveness*), es decir la cantidad de actividad concurrente que se puede realizar. A menudo es necesario sacrificar algo de corrección para obtener más vivacidad, y esto depende de la gravedad y la probabilidad de los fracasos y de la necesidad de trabajar con los mismos datos al mismo tiempo.

Para solucionar estos problemas utilizamos varios mecanismos de control, pero estas soluciones también introducen sus propios problemas, aunque no son tan graves como los problemas básicos.

### Aislamiento e inmutabilidad

Existen dos soluciones con respecto a los problemas de concurrencia mencionados, y estos son el **aislamiento** (isolation) y la **inmutabilidad** (immutability).

Los problemas de concurrencia ocurren cuando más de un agente activo, como un proceso o un hilo, tienen acceso a la misma porción de datos. Una manera de solucionar esto es el aislamiento, es decir, particionar los datos de manera que cualquier porción de estos pueda ser accedida solamente por un agente activo. Los procesos trabajan así con la memoria de los sistemas operativos. El sistema operativo aloca memoria exclusiva para un único proceso, y solamente ese procesos puede leer y escribir los datos ligados a él.

El aislamiento es una técnica vital que reduce las chances de error, ya que los programas al entran en una zona de aislamiento donde no deben preocuparse por los problemas de concurrencia. Un buen diseño concurrente es aquel que encuentra la manera de crear estas zonas y hacer lo posible para que la programación se produzca en una de ellas.

Solamente tendremos problemas de concurrencia si los datos que se comparten pueden ser modificados. Por lo tanto una manera de eliminar los conflictos de concurrencia es reconocer los **datos inmutables**. Obviamente no podemos hacer que todos los datos sean inmutables, ya que el objetivo principal de muchos sistemas es la modificación de datos. Pero detectar los

datos inmutables, o por menos aquellos que son inmutables la mayor parte del tiempo, nos permite despreocuparnos de los problemas de concurrencia sobre ellos.

Una manera de proveer el aislamiento es contar con soporte para distintas políticas de concurrencia, dependiendo del aislamiento que los datos puedan tener en cada operación.

Por ejemplo, teniendo un objeto "padre" con una colección de "hijos", podríamos requerir alguna de las siguientes políticas:

- Al modificar el padre, se bloquea el padre y cada uno de sus hijos.
- Al modificar un hijo y se bloquea solo ese hijo.
- Al modificar un hijo y se bloquea el hijo modificado y el padre.
- Al modificar un hijo y se loquean determinados hijos según algún criterio, como por ejemplo hijos del mismo tipo.

### Estrategia de loqueo Pesimista y Optimista

En muchas ocasiones, nos es imposible aislar los datos que necesitamos cambiar. Es aquí donde entra en juego las políticas de loqueo de datos.

En lo que respecta al control de concurrencia tenemos dos maneras de utilizarlo: optimista y pesimista. Por ejemplo, en el caso de los editores,

- Con una política optimista, ambos editores podrán obtener una copia electrónica para editar su documento; el primero en terminar podrá guardar sus cambios sin problemas. El control de concurrencia dará la alerta cuando el segundo editor intente guardar sus cambios, y existan conflictos con los cambios realizados por el primero, rechazando el pedido del segundo editor.
- Con una política pesimista, el control de concurrencia controla que solamente el primer editor que solicite la copia podrá sea el único que la podrá editar, y nadie más podrá, hasta que los cambios sean gradados.

Si bien vamos a analizar cada una de estas estrategias, se puede decir que ambas tienen un patrón de uso en común en donde distinguimos dos etapas:

- La etapa de **alocación,** donde se obtienen los recursos y locks correspondientes, y estos recursos son modificados. Esta etapa puede abarcar múltiples conexiones a la base de datos, realizándose sólo operaciones de consulta de datos. Durante esta etapa suele construirse un grafo de objetos que representa a los objetos obtenidos. También suele llevarse un

registro de los cambios realizados a estos objetos.

- La etapa de commit, donde se vuelcan los cambios realizados en los objetos a la base de datos. Esto suele ejecutarse en el marco de una sola transacción de base de datos (o múltiples si estamos usando varias bases de datos, pero controladas por un manejador de transacciones distribuido).

Una buena manera de pensar este problema es que el loqueo optimista "detecta" los problemas y el pesimista los "previene".

<u>Loqueo optimista</u>

El loqueo optimista explota la baja probabilidad de concurrencia en la modificación de objetos, impidiendo que se formalice una modificación mediante el commit de la transacción asociada si se verifica que el usuario estuvo trabajando con un objeto sucio (modificado por otro usuario). Más formalmente, se verifica que la versión sobre la que se realizaron las modificaciones que se quieren guardar sea la última. Esto se puede implementar mediante la fecha y hora de la modificación o números de versión. En el caso en que se detecte una diferencia de versión, la actualización no se realizará y se informará el problema al usuario.

Cada objeto de dominio posee como atributo el número de versión o la fecha y hora de la modificación. Sin embargo, existen objetos complejos que contienen un conjunto de objetos (nuestro ejemplo previo de padre e hijos). Si consideramos al padre como una unidad, no se puede permitir que dos usuarios distintos actualicen distintos hijos en forma concurrente, por lo cual como consecuencia de una modificación en un hijo, no sólo se debe actualizar el número de versión o la fecha y hora de la modificación del mismo sino también el del padre. [Gru89]

<u>Loqueo pesimista</u>

El loqueo pesimista impide cualquier modificación al objeto loqueado. Esto se implementa "marcando" al objeto u obteniendo un token de modificación. Esta estrategia se usa en casos en que exista una alta probabilidad de que dos usuarios intenten realizar modificaciones concurrentes o si el costo (evaluado en tiempo de trabajo) de que el usuario realice una modificación que no se concrete es muy alto para permitirlo, independientemente de lo remota que sea la probabilidad de que ocurra.

El problema más grave asociado al bloqueo pesimista se da cuando un usuario loquea un objeto (en nuestro ejemplo el

padre) y luego olvida des-loquearlo, permaneciendo loqueado hasta que se invalide la sesión de usuario, por ejemplo luego de un periodo de inactividad.

Ambas aproximaciones tienen sus pros y sus contras. El problema con el loqueo pesimista es que reduce la concurrencia. El loqueo optimista permite mayor progreso, dado que el loqueo se produce solo en el momento de guardar los cambios. El problema es ¿qué pasa si entramos en conflicto?

En nuestro ejemplo de modificación concurrente de un documento, previo a todos los cambios que se intenten guardar después de que el primer editor los guarda, deben chequear la versión del archivo y luego averiguar cómo fusionar los documentos eliminando los conflictos. En muchos casos esta operación no cuenta con demasiadas dificultades, pero al trabajar con datos de negocios, usualmente esto es bastante más complicado, y a menudo se debe tirar todos los cambios realizados y comenzar nuevamente.

La esencia de la elección de la estrategia de loqueo es la frecuencia y la severidad de los conflictos. Si los conflictos ocurren circunstancialmente y si las consecuencias no son gran cosa, deberíamos seleccionar el loqueo optimista, debido a que nos proporciona mejor concurrencia y usualmente es más simple de implementar. Por otro lado, si los resultados de un conflicto son muy costosos par el usuario, deberíamos utilizarla técnica de loqueo pesimista.

Ninguno de estas opciones está libre de problemas, es más, utilizándolas podemos introducir problemas que causan otros tantos como los básicos de la concurrencia que intentamos solucionar por primera vez.

### Transacciones

La primera herramienta para manejar la concurrencia en Aplicaciones Enterprise es la transacción. Una transacción es una secuencia de trabajo delimitada, con puntos de comienzo y de fin definidos, en donde todos los recursos que participan se mantienen en un estado consistente tanto al comienzo como al finalizar la misma.

Las transacciones ayudan a eliminar la mayoría de los aspectos complicados de la concurrencia en Aplicaciones Enterprise. Mientras se haga toda la manipulación de datos dentro de una transacción, nada malo podrá ocurrir. Lamentablemente esto no significa que podamos ignorar completamente el problema de la concurrencia.

### Patrones para control de concurrencia off line

Una frase común acerca de la concurrencia es que es una decisión puramente técnica, que se pueden decidir cando los requerimientos están completos. Nosotros estamos completamente en desacuerdo. La elección del control pesimista u optimista afecta por completo la interacción del usuario con el sistema. Un diseño inteligente del patrón de Loqueo Offline Pesimista, necesita una gran cantidad de entradas del usuario del sistema hacia el dominio. De manera similar un conocimiento del dominio es necesario al elegir un buen Loqueo De Grano Grueso.

Lidiar con la concurrencia es una de las tareas más difíciles de la programación. Es muy difícil testar código concurrente. Los defectos originados por la concurrencia son difíciles de reproducir y muy difíciles de seguir. Los patrones que hemos descripto nos han servido hasta el momento, pero es un territorio particularmente difícil.

En la medida de que sea posible, debemos dejar que algún sistema de transacciones, ya sea a nivel web aplicación o a nivel base de dato, se encargue de los problemas de concurrencia.

Si las transacciones se extienden a lo largo de la aplicación, tanto entre distintas maquinas como entre las distintas capas que la conformen, debemos enfrentarnos al difícil mundo del manejo de la concurrencia. Desafortunadamente, la falta de correspondencia entre las transacciones de sistemas y de negocios nos obliga a esto. Los siguientes patrones son algunas técnicas útiles en el tratamiento de control de concurrencia para las transacciones que se extienden a través del sistema.

No está de más mencionar que estas técnicas deben utilizarse solo si son necesarias. Si nuestro sistema permite ejecutar toda una transacción de negocio dentro de una transacción de sistema, la cual se ejecuta en una única solicitud, y podemos renunciar a la escalabilidad en la utilización de transacciones largas, no debemos dudar en utilizar esta solución, lo cual nos evitará una gran cantidad de problemas. Las siguientes técnicas son la que debemos utilizar cuando el sistema o los requerimientos no nos lo permiten.

Debido a la naturaleza de la concurrencia, tenemos que subrayar una vez más que los patrones son un punto de partida y no un destino, es decir, son muy útiles pero no tienen la pretensión de encontrar una cura para todos los males de concurrencia. Mantendremos el nombre de los patrones en inglés debido a que se los identifica mejor de ésta manera.

Optimistic Offline Lock

Nuestra primera opción para el manejo de problemas de concurrencia off-line es el patrón de Loqueo Optimista Offline, que básicamente utiliza el control de concurrencia optimista a través de las transacciones de negocios. Ésta es nuestra

primera opción dado que es el método más fácil de programar y nos da un mayor rendimiento al mejorar la vivacidad (liveness).

La limitación de este patrón está en que sólo se descubre que una transacción de negocios va a fallar al momento de finalizar la misma, y en algunos casos esto trae un dolor de cabeza a los usuarios.

Pessimistic Offline Lock

La alternativa es el Loqueo Pesimista Off-Line, con la que se encuentra lo antes posible si existen problemas de concurrencia, pero es difícil de programar y reduce la vivacidad.

Con cada una de estas alternativas se puede eliminar parte considerable de la complejidad que trae manejar el loqueo en cada objeto de nuestro modelo.

Coarse-Grained Lock

Un Loqueo de Grano Grueso nos permite manejar la concurrencia de todo un grupo de objetos.

Implicit Lock

Otra manera de simplificar la vida de los desarrolladores de la aplicación es utilizar un Lockeo Implícito, el cual nos ahora manejar el loqueo de manera directa. No solamente ahorra trabajo, sino que elimina los defectos difíciles de encontrar, generado cuando la gente olvida realizar este manejo.

## Transacciones largas

Las Aplicaciones Enterprise cuentan con operaciones que tienen características de transacciones largas, es decir, que abarcan más de un ciclo requerimiento - respuesta al servidor de aplicaciones.

Se considera que una transacción larga comienza desde el momento que se consultan los datos para su modificación hasta que se decide el guardado de los mismos.

En sistemas basados en la arquitectura cliente/servidor, la estrategia clásica implementada para soportar este tipo de transacciones, consiste en asignar una conexión de base de datos a cada cliente, delegando a las transacciones de la base de datos el loqueo y el cumplimiento de las **características ACID** que debe garantizar cada transacción.

El principal problema de esta estrategia es básicamente de escalabilidad. Los motores de bases de datos pueden alocar una cantidad finita de conexiones (cada conexión consume recursos) y soportar mayor cantidad de

usuarios depende directamente de la capacidad del servidor de bases de datos. Tengamos en cuenta que el patrón de uso de una aplicación no requiere un uso continuo de las conexiones, sino que está mejor descrito como por ráfagas, en las que se obtienen los datos de una pantalla o se persisten los cambios realizados por el usuario. De esta manera, se desprende que el servidor aloca una gran cantidad de conexiones que probablemente estén inactivas la mayor cantidad del tiempo.

Una evolución de este esquema es compartir las conexiones a la base de datos. Se inicializan una cantidad razonable de conexiones en función de la cantidad de transacciones concurrentes que se estime necesario soportar. Estas conexiones se comparten entre todos los usuarios del sistema y cada vez que un usuario necesite utilizar la base de datos, obtiene la conexión, ejecuta sus sentencias sobre la base de datos y la libera. Con este esquema es posible soportar (al menos desde el motor de bases de datos) una cantidad mucho mayor de usuarios.

Llamamos una **transacción larga o transacción de negocio** una serie de operaciones que pueden abarcar varios ciclos de pedido-respuesta entre el cliente y el servidor, en los cuales se utilizan distintas conexiones a la base de datos. Esto impide el uso del mecanismo de loqueo pesimista provisto por la base de datos y nuestro problema ahora es seguir teniendo las características ACID de las transacciones normales pero sin el soporte (o solo de forma parcial) de la base de datos.

### Transacciones de sistema y de negocio

En una transacción de base de datos, si alguna sentencia en la transacción resulta en una violación a alguna restricción de integridad, la base de datos debe deshacer los efectos de todas las sentencias ejecutadas previamente dentro de la transacción y notificar al cliente que la transacción ha fallado. Si todas las sentencias de la transacción son completadas satisfactoriamente, todo debe ser visible para el resto de los usuarios, y todo esto al mismo tiempo.

Sin embargo, una **transacción de sistema** o de base de datos, no tiene ningún significado para el usuario del sistema. Por ejemplo, para un usuario de un sistema bancario on-line, una transacción consiste en acceder al sistema, seleccionar una cuenta, crear algunos pagos de facturas, y finalmente hacer clic en el botón OK para pagar las cuentas.

Esto es lo que llamamos una **transacción de negocios**, que como es de esperarse, cuenta con las propiedades ACID, al igual que una transacción de sistema. Es decir, si el usuario cancela la operación antes de pagar las cuentas, los cambios realizados en las pantallas anteriores deberán ser cancelados, o en otro caso, la creación de los pagos no debería ser visible en el sistema que maneja el balance hasta que el usuario presione el botón

OK.

La respuesta obvia para dar soporte a las propiedades ACID en una transacción de negocios es ejecutarla íntegramente en una única transacción de sistema. Desafortunadamente las mayorías de las transacciones de negocios necesitan de varias solicitudes para completarse, lo que resulta en una transacción larga.

La mayoría de los sistemas de manejo de transacciones no trabajan de la mejor manera con transacciones largas. Esto no quiere decir que nunca se deba utilizar transacciones largas, sin embargo, si se utiliza una base de datos y no se cuentan con demasiados requerimientos de concurrencia, entonces es recomendable utilizar las transacciones de sistemas prolongadas, ya que nos soluciona varios problemas difíciles de resolver.

Todo esto teniendo en cuenta las necesidades de escalabilidad del sistema, dado que las transacciones largas convierten a la base de datos en un cuello de botella. Además la reestructuración necesaria para que los sistemas pasen de manjar transacciones largas a cortas es demasiado compleja y difícil de entender. Por esta razón muchas de las Aplicaciones Enterprise no pueden arriesgarse a utilizar transacciones prolongadas.

En estos casos tenemos que partir nuestras transacciones de negocios en una serie de transacciones cortas, esto significa que delegaremos al sistema el soporte a las propiedades ACID de las transacciones de negocios entre las transacciones de sistemas. A este problema lo llamamos concurrencia off line.

Aún así, las transacciones de sistemas siguen teniendo gran importancia. Cada vez que la transacción de negocio interactúe con un recurso transaccional, tales como las bases de datos, ésta interacción se realizará dentro de una transacción de sistema con el fin de mantener la integridad de ese recurso.

Sin embargo, como usted leerá a continuación no es suficiente hilvanar una serie de transacciones de sistema para dar soporte adecuadamente a una transacción de negocios. La aplicación Enterprise deberá agregar una interacción entre ellas.

La atomicidad y la durabilidad son las propiedades ACID más simples de soportar en las transacciones de negocios. Ambas son soportadas mediante la ejecución de la fase de confirmación de la transacción de negocios, cuando el usuario pulsa finalizar dentro de una transacción de sistema.

Antes de que la sesión intente bajar todos sus cambios al conjunto de registros, este primero abre una transacción de sistema. El sistema de transacciones garantiza que los cambios se van a realizar como una unidad y que se hará de manera permanente.

La única parte que puede complicarse, es mantener un conjunto de cambios precisos durante la vida de la transacción de negocios. Si la aplicación utiliza

un modelo de dominio (Domain Model) y unidad de trabajo (Unit Of Work) puede rastrear con precisión los cambios realizados en la transacción.

De las propiedades ACID, la más complicada de hacer cumplir en las transacciones de negocios, es el aislamiento. Las fallas en el aislamiento provocan en fallas en la consistencia. La consistencia indica que una transacción de negocio no debe dejar el conjunto de datos en un estado inválido. La responsabilidad de la aplicación, dentro de una transacción simple con respecto a la consistencia, es hacer cumplir todas las reglas de negocios. A lo largo de múltiples transacciones la responsabilidad de la aplicación es que una sesión no se mezcle con otras sesiones dejado el conjunto de datos inconsistentes, debido a la pérdida del trabajo realizado por el usuario.

Además de los problemas evidentes de pérdidas de actualizaciones (Lost Updates), existen problemas más sutiles con las lecturas inconsistentes (inconsistent reads). Cuando los datos son leídos en varias transacciones de sistemas, no está garantizado que estos sean consistentes. Diferentes lecturas pueden incluso introducir datos en la memoria que es suficientemente inconsistente como para causar un fallo en la aplicación.

Las transacciones de negocios están estrechamente relacionadas con las sesiones. Desde el punto de vista del usuario cada sesión es una secuencia de transacciones de negocios (aunque ellas solo sean de lectura de datos), por lo que suele asumirse que todas las transacciones de negocios se ejecutan en un única sesión del cliente.

Mientras que es posible diseñar unos sistemas que tenga múltiples sesiones para una transacción de negocio, esta es una buena manera de entrar en confusiones, por lo tanto no lo aremos.

## Objetos de gran tamaño y carga por demanda

A veces un objeto de dominio puede ser difícil de serializar debido a que tiene una estructura de relaciones complicada. Por complicada nos referimos a, o bien muchos colaboradores con distintos niveles de profundidad que hacen que el grafo no sea sencillo, o bien colecciones de una elevada cantidad de elementos.

En estos casos la carga de un objeto en el cliente podría significar cargar un gran número de objetos relacionados. Esto degradaría la performance si el cliente sólo necesita unos pocos objetos ya que podría no querer el modelo entero sino sólo un conjunto simplificado del mismo.

Cuando se cargan datos desde la base de datos a memoria es conveniente diseñar las cosas de modo que, cuando se cargue el objeto requerido también se carguen los objetos relacionados. Esto hace la carga del objeto más sencilla para el desarrollador, quien de otro modo tendría que cargar

todos los objetos que necesita explícitamente.

Es necesario entonces proveer un mecanismo que permita interrumpir por el momento la carga del objeto, dejando marcas en la estructura del objeto de modo que si el dato se necesita se pueda cargar en el momento de usarlo.

A la técnica que implica no cargar un componente hasta que es usado se la conoce como carga por demanda o “Lazy Loading”. Esta técnica evita inicializar todo el grafo de dependencias de un objeto distribuyendo la creación de las dependencias a medida que se necesitan. Es decir, se debe soportar la carga por demanda para evitar levantar el grafo completo de objetos y que los objetos de adentro del grafo se vayan cargando a medida que se necesiten.

A la hora de decidir cuándo utilizar carga por demanda se debe analizar cuánto se quiere traer de la base de datos cuando se carga un objeto, así como también cuántos llamados a la base de datos se van a requerir. Desde el punto de vista del tiempo de respuesta, no tiene sentido usar carga por demanda en propiedades que, en el repositorio de datos pueden ser recuperados por en un único procesamiento junto con el objeto, porque la mayoría de las veces incrementa el costo traer datos extras en una llamada, aún si los datos son grandes. Por lo que sólo vale la pena considerar el uso de este mecanismo si la propiedad requiere un llamado extra a la base de datos para accederla.

En lo que a performance se refiere es cuestión de decidir cuándo se quiere hacer el llamado para traer los datos restantes. En la mayoría de los casos es una buena idea traerse todo lo que se va a necesitar en una llamada para tenerlo a mano, principalmente si el llamado corresponde a una interacción simple con la interfaz de usuario. El momento apropiado para utilizar carga por demanda es cuando está involucrado un llamado extra y los datos que se estarían trayendo no son utilizados junto con el objeto principal.

Dado que el uso de la carga por demanda de objetos agrega un poco de complejidad al programa, muchos recomiendan no usarla a menos que se esté completamente seguro que se va a necesitar.

## Deshacer operaciones - UNDO

Durante una transacción larga, y en forma previa a la confirmación de la operación, debe permitirse volver atrás con algunos cambios realizados.

La posibilidad de hacer undo de operaciones es una característica esencial de toda aplicación con clientes ricos. Si bien su implementación puede ser un poco complicada, los beneficios que le trae al usuario de la aplicación son los suficientes como para tenerla en cuenta.

## Beneficios del Undo

Alivio de las preocupaciones

La habilidad de hacer undo es un alivio para muchos usuarios. Es tranquilizador saber que uno siempre puede deshacer algo si uno se equivoca, y cuando esa opción no está disponible, puede ser incómodo.

Una forma de reducir la complejidad

La posibilidad de hacer undo crea una sensación de simplicidad aun en las aplicaciones más complejas. La habilidad de hacer undo genera una actitud de confianza por el hecho de saber que siempre se puede deshacer si uno se equivoca. Una vez que se tiene esa actitud el usuario percibe las cosas más simples de lo que realmente son.

Un método para construir la confianza

Se genera confianza en el poder del undo para hacer su magia permitiéndonos jugar y experimentar con una aplicación. Experimentando y jugando nosotros aprendemos y con el conocimiento viene la confianza en nuestras habilidades.

## Undo en aplicaciones Web

La diferencia fundamental entre una aplicación y un sitio web es que las aplicaciones, en general, pueden ser vistas como entidades que operan realizando transformaciones sobre datos, por lo tanto a lo largo del ciclo de vida de una aplicación existen diferentes estados de los datos, algunos inducidos por el usuario, por ejemplo al escribir un documento, o realizado por el sistema, como por ejemplo sincronización, aplicación de lógica de negocios, etc. Por otra parte un sitio web es un conjunto de información, estructurada según un amplio conjunto de paradigmas y patrones (categorización, etiquetado, jerarquización) que sufre muy pocas transformaciones, o en algunos casos ninguna, a lo largo de su ciclo de vida. En este sentido los diferentes estados en que se puede encontrar un sitio web, es decir, las páginas que se muestran, dependen de la relación entre los diferentes elementos que lo componen, y sus modificaciones responden mayormente a la voluntad del usuario del sitio.

Por lo tanto, en el caso de la aplicación los datos cambian en el tiempo, en el caso de los sitios web los datos no cambian, son estáticos. Entonces es muy difícil considerar que los mismos paradigmas pueden aplicar en ambos contextos funcionales.

Los elementos de navegación de un browser han sido diseñados para funcionar muy bien en un contexto en que el usuario recorre una estructura de documentos, ya sea con caminos predeterminados o no. En el caso de las aplicaciones estos elementos deberían ser reemplazados por la posibilidad de hacer undo/redo (deshacer/rehacer) lo que actuaría en el sentido de "navegación" entre estados de los datos que la aplicación está procesando (deshacer un cambio en un formulario, etc.).

Muchos dirán que esto es algo trivial, sin embargo basta mirar un poco el mundo de las "aplicaciones web" y casi ninguna hace uso de estas herramientas.

### El patrón Undo

A continuación una breve, e introductoria, explicación del patrón Undo.

¿Qué es?

> Permite al usuario la posibilidad de deshacer una acción realizada con anterioridad

¿Cuándo usarlo?

> En cualquier instancia donde es factible perder trabajo realizado (por ejemplo al borrar o modificar datos). Cuanto más costosa es la recuperación de datos más importante es la posibilidad de deshacer.

¿Por qué?

> El software que permite deshacer es software en el que puedes confiar.
>
> Todos cometemos errores, por ejemplo en "Aceptar" en un diálogo de cerrar, cuando en realidad no era lo que queríamos, perdiendo el trabajo realizado. En este sentido el poder de la costumbre es muy fuerte, llevándonos a hacer click en aquellas opciones más comúnmente usadas, incluso en aquellos casos en que deberíamos seleccionar otra opción. En este contexto es mucho más útil proveer la posibilidad de deshacer que la de diálogos de confirmación. Nunca usar una advertencia cuando puedes proveer la opción de deshacer.

¿Cómo funciona?

> Decide cuales opciones deben ser posibles de deshacer y genera una pila de acciones a deshacer.
>
> Después de una acción que puede ser deshecha provee un link

o botón para deshacer la acción.

## Otras características de los objetos persistentes en el cliente

Los objetos manipulados por la el cliente pueden relacionarse entre ellos de la misma manera que se relacionan los objetos de dominio, incluyendo también la relación de herencia.

Una colección de objetos mostrada en una grilla o listado puede ser requerida y transmitidas al cliente con un orden y un filtro inicial.

Objetos con grandes volúmenes de datos suelen tener grandes colecciones de objetos, las cuales deberán tener un tratado especial para mantener la performance en nuestro cliente. En general se necesitará algún mecanismo de “paginado” para este tipo de colecciones, el cual debe ser eficiente al momento de recuperar estos datos, evitando tener que realizar estas operaciones en memoria En este caso se deberá mantener además el orden y el filtro seleccionado.

Los objetos manipulados por el cliente deben soportar también algún mecanismo que permita definir atributos calculados, útiles para la visualización de la información.

# *Capítulo 6*

# Solución

*En este capítulo definiremos el concepto de Plataforma de Desarrollo la cual tiene como objetivo dar solución a los problemas planteados anteriormente. Además describiremos en profundidad el framework que será parte fundamental de nuestra plataforma.*

## Plataforma de desarrollo

La idea de una plataforma para el desarrollo, es definir el **conjunto de frameworks y servicios que colectivamente provean una forma sencilla y coherente de integrar las distintas funcionalidades a desarrollar y definir los lineamientos y las pautas de desarrollo**.

Estos frameworks y servicios representan la base para los diferentes desarrollos, proveyendo una funcionalidad genérica para utilizarla como base del desarrollo de los requerimientos funcionales específicos, ayudando así a unificar la manera en que éstos son desarrollados.

Dado el contexto en el cual se encuentre el presente trabajo, nos centraremos en una plataforma de desarrollo orientada a las Aplicaciones Enterprise de gran tamaño.

Una de las características de las Aplicaciones Enterprise es su gran envergadura en relación a la cantidad de funcionalidad a implementar, y por ende también en el tamaño del equipo encargado del diseño y desarrollo de la misma. Al contar con una plataforma compuesta de frameworks y herramientas, disponibles en la comunidad y eventualmente propios, que dan solución a los distintos problemas que se presentan en el proceso de desarrollo, permiten una estandarización del mismo.

Una vez estandarizado el proceso, el objetivo de los equipos es diseñar e implementar los requerimientos funcionales de la aplicación, delegando en la plataforma la solución a los problemas descriptos en el capítulo anterior. Esto último es parte fundamental de objetivo de esta plataforma de desarrollo.

Basándonos en esto, nos vemos en la situación de definir de qué manera trabajará nuestra plataforma para permitir manipular objetos persistentes en el cliente de una Aplicación Enterprise. En esta definición debemos tener en cuenta cuáles son los datos que necesitamos, de qué manera los obtenemos y cómo se comportarán.

## Framework propuesto

Basándonos en el análisis realizado hasta el momento y teniendo en cuenta el alcance de este trabajo, en lo que resta del presente capítulo presentaremos el diseño de un framework que dará soporte al manejo de objetos persistentes en la componente cliente de Aplicaciones Enterprise.

### Objetivos

Su objetivo principal es resolver la transferencia de información entre las distintas capas de la arquitectura, manteniendo desacopladas la capa de presentación del resto de las capas, y lidiando con los problemas inherentes al manejo de objetos persistentes en el cliente.

Podemos decir entonces que este framework es de naturaleza estructural, es decir, resuelve un problema tecnológico, y no se origina de los requerimientos funcionales del sistema.

En el contexto de éste objetivo general, encontramos una serie de requerimientos básicos los cuales son necesarios satisfacer y están detallados a continuación:

- Brindar una solución que permita la ***transferencia de información*** entre la capa de presentación y la del modelo de dominio, manteniendo las mismas desacopladas según el criterio de diseño.
- Resolver las operaciones que tienen características de ***transacciones largas o de negocios***, es decir, que abarcan más de un ciclo pedido-respuesta al servidor de aplicaciones.
- Brindar la posibilidad de ***volver atrás cambios*** realizados durante una transacción larga, en forma previa a la confirmación de la operación [Raz09] [Gam02] [Dat01]
- Mantener la ***relación de los objetos manipulados por la capa de presentación,*** de la misma manera que se relacionan los objetos de dominio, incluyendo también la relación de herencia.
- Proveer algún mecanismo que permita definir ***atributos calculados***, útiles para la visualización de la información.
- Los servicios básicos de manipulación de datos, ***operaciones CRUD***, cuentan con un comportamiento común para todos los objetos del sistema. Por esto es necesario brindar el mecanismo necesario para agregarles validaciones u acciones adicionales [Mar86].

- Soportar un ***comportamiento transaccional*** y asegurar la integridad de los datos de acuerdo a las siguientes situaciones de concurrencia:
- Puede existir ***concurrencia*** en la modificación de dos o más objetos. Si en el período entre que se consulta y se guardan los datos, otro usuario modificó el mismo la operación debiera fallar. [Ram02] [Cor09][ Goe06]
- Permitir la manipulación de un objeto individual que modele una entidad funcional, junto con todos sus colaboradores, o una colección de objetos independientes (listas huérfanas).
- Proporcionar mecanismos de ***paginación*** en las propiedades del tipo colección, a efectos de no recargar al cliente con grandes cantidades de información que no es utilizada en la transacción, o en algún momento determinado. [Ant09]
- Proporcionar distintos controles de ***concurrencia***. Un ejemplo de esto es controlar solo los objetos que forman parte de la entidad funcional y no afectar a los objetos relacionados con él. Por ejemplo, si se tiene un objeto Padre que contiene una colección de Hijos, si se modifica uno de los hijos sólo se controla que no haya accesos concurrentes al hijo, pero no así al objeto padre.

  Otro escenario posible sería marcar como modificado el objeto principal (padre) cuando tiene como atributo una lista o un mapa, y a estas colecciones se le agregan o eliminan elementos, ya que la modificación del conjunto de elementos sería considerada como una actualización del padre. Sin embargo, si se modifica el estado de uno de los elementos de la lista o mapa, el padre no se considera actualizado y por lo tanto no se controla concurrencia.

Además de dar soporte a los requerimientos básicos mencionados, debemos tener presente que éste framework debe simplificar el esfuerzo del desarrollo de los requerimientos funcionales, y del posterior mantenimiento de los mismos, evitando duplicar funcionalidad tanto en la definición de la información manipulada como de los procesos de transferencia.

Otro punto fundamental es la eficiencia de los servicios, la cual no debe verse sustancialmente afectada por el componente, por debe entonces priorizarse una funcionalidad acotada y eficiente a los casos comunes, a una completa pero con mucha penalidad en este aspecto.

### Características del framework

El framework deberá satisfacer una serie de requerimientos, destinados a brindar una mayor flexibilidad en su uso. La mayoría están relacionados con el manejo de las colecciones. [Mak10]

<u>Paginado de colecciones de objetos</u>

Una colección mostrada en una grilla o listado, puede llevarse al cliente con un orden, un filtro inicial y paginada. En caso de ser paginada, cuando se trae la segunda página se debe respetar el orden y el filtro original [Pag09]

<u>Orden y filtro en colecciones paginadas</u>

Al requerimiento anterior se le agrega la posibilidad de que en dicha colección pueda especificarse un orden y un filtro inicial. Es decir, permitir que se especifique la forma en que va a venir ordenada y dar la posibilidad de indicar un filtro que se aplique a los elementos de la colección.

Tanto el orden como el filtrado se debe realizar sin afectar el modelo de dominio, es decir, los cambios van a percibirse en el objeto que llega al cliente pero el objeto de dominio va a permanecer intacto, con su orden por defecto y sin verse afectado por ningún filtro.

Se debe tener especial cuidado con las colecciones paginadas a la hora de pensar la solución a este requerimiento, ya que se debe asegurar que en el momento de traer la segunda página se respete el orden y el filtro original.

Hay una restricción respecto al uso del orden en las colecciones, y es que para que el usuario pueda cambiar el orden de la colección que se trajo (filtrada o no) se tiene que haber llevado todas las páginas que entran en el filtro. Esta restricción es necesaria para evitar conflictos entre distintas páginas que fueron recuperadas con distintos criterios de orden.

<u>Límite para colecciones</u>

Existen sistemas que poseen un modelo de dominio donde algunos objetos contienen colecciones cuyo tamaño es muy grande. Cuando esto sucede, es muy probable que a pesar de paginar dichas colecciones se torne muy difícil manejar tanta cantidad de datos en el cliente (mayormente por limitaciones en el hardware del cliente).

Es por esto que vemos la necesidad de pedirle a nuestro framework que permita limitar el número de elementos que una colección en particular pueda llevar al cliente. Es decir, que sea posible especificar sobre una colección un límite de tamaño cuando dicha colección no se quiera completa, ya sea porque es demasiado grande o por problemas de performance.

Chequeo de integridad

Los cambios que se realicen sobre el objeto que fue transferido al cliente van a ser persistidos posteriormente en el objeto de dominio alojado en el servidor. Toda aplicación tiene reglas que deben ser satisfechas para mantener la integridad de sus objetos de dominio. Por esto, es de gran utilidad que el framework tenga una manera de verificar la integridad de un objeto antes de persistirlo. De este modo, cuando se hace una modificación no se producirán cambios si los objetos no superan dicho chequeo.

## Arquitectura del Framework

En esta sección nos adentramos en cómo está conformada la arquitectura del framework, dando un primer vistazo al modo de funcionamiento y a su modo de uso. [Joh92]

Para entender la arquitectura del framework es necesario definir los siguientes conceptos, la mayoría de los cuales están basados en patrones de diseños [Gam02]:

StateTransferObject , STOList y STOMap

Son el contenedor de los datos transferidos desde el servidor al cliente. Son estructuras genéricas que contienen la información requerida por el cliente.

DyTO , DyTOList y DyTOMap

Es el objeto que envuelve los datos transferidos (STO) y mantiene los cambios realizados en el cliente en la UnitOfWork. [Sun02]

UnitOfWork

Registra los cambios realizados en el cliente en un log de comandos que concretan los cambios en los objetos de dominio en el servidor [Fow02]

DyTODynamicProxy, DyTODynamicProxyList y DyTODynamicProxyMap

Son proxies dinámicos que encapsulan a los DyTOs y le permite al cliente manipularlos como si fueran DTOs específicos. [Bus07]

## División en capas

El framework está organizado en capas de niveles de abstracción. Cada capa delega la comunicación en la capa inmediatamente inferior. Este

diseño permite basar cada capa en las abstracciones provistas por su inmediata inferior, permitiendo atacar un problema cada vez. [Erl07]

Las capas del framework son las siguientes:

- Capa de Servicios CRUD
- Capa de DyTOs
- Capa de StateTransferObject

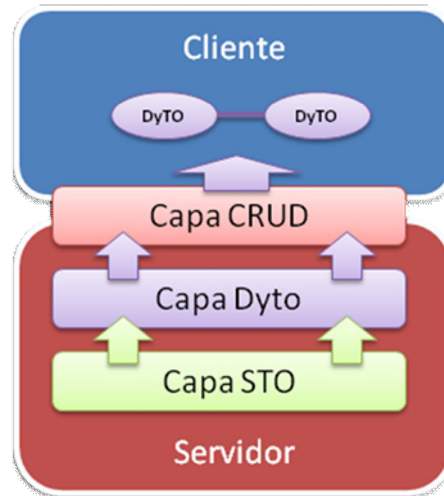


***Figura 18 - Arquitectura en capas***

**Capa de servicios CRUD**

Es la capa con la cual interactúa el usuario del framework. Su función es proveer al desarrollador la interface de programación necesaria para poder realizar las operaciones de Creación, Recuperación, Modificación y Eliminación (CRUD – Cerate Retrieve Update Delete) sobre los objetos del modelo de dominio.

Su principal característica es la abstracción que provee al cliente del framework, de la implementación de los objetos de transferencia dinámicos (DyTOs), utilizados para “simular” a los objetos del modelo de dominio .en el cliente de la aplicación a desarrollar.

Los objetos que maneja esta capa pueden verse como Data Transfer Objects (DTOs), pero con la salvedad de que el desarrollador no debe implementar cada objeto de transferencia específico, ni su mecanismo de extracción; sino que simplemente se limita a describir lo que desea obtener, utilizando la manera de representar la meta-información.

Toda información manipulada en el cliente se almacena en un objeto DyTO, encapsulado en un proxy dinámico que cumple con la descripción provista.

Un objeto DyTO puede contener además de sus propiedades y de las referencias a sus colaboradores, listas o mapas de otros objetos.

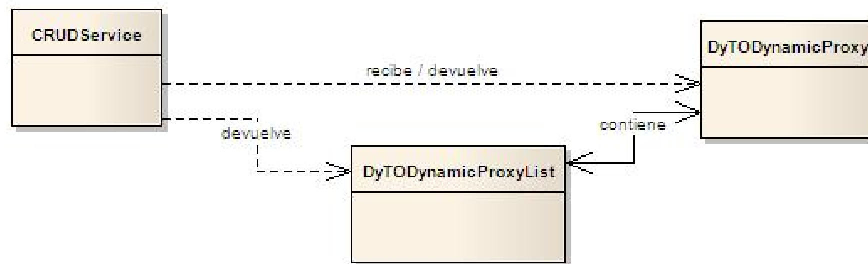


***Figura 19 - Capa de servicios CRUD***

Mediante este mecanismo el cliente interactúa con estos objetos de la misma manera que lo haría con los objetos del modelo de dominio, permitiendo un acceso fuertemente tipado al mismo, abstrayéndolo de los detalles de implementación del framework.

Dentro de la meta-información utilizada para la descripción de los objetos a manipular en el cliente de nuestra aplicación, podremos encontrar la definición de los siguientes tipos de propiedades:

- **Propiedades simples** - Normalmente son los tipos primitivos, strings, fechas, etc.
- **Propiedades DyTO** - Representan la extracción de objetos de dominio apuntados por una propiedad del DyTO (un colaborador). Esta extracción puede ser el DyTO del objeto de dominio asociado o una referencia al mismo, la cual nos permitirá obtenerlo de manera lazy.
- **Propiedades calculadas**, como por ejemplo el total de una factura.
- **Listas o Mapas de DyTOs**.
- **Jerarquías de clases**

Esta capa permite realizar las operaciones CRUD sobre los DyTOs. Todas las modificaciones realizadas se almacenan en el DyTO y luego serán aplicadas al objeto de dominio que corresponda, en las correspondientes capas del framework.

El punto de acceso a los servicios que permiten realizar estas operaciones se encuentra definido en una interfaz, que permite construir las descripciones basándose en la meta-información provista, para luego poder generar los objetos necesarios en el cliente de la aplicación basándose en los objetos de dominio que se describan en la misma.

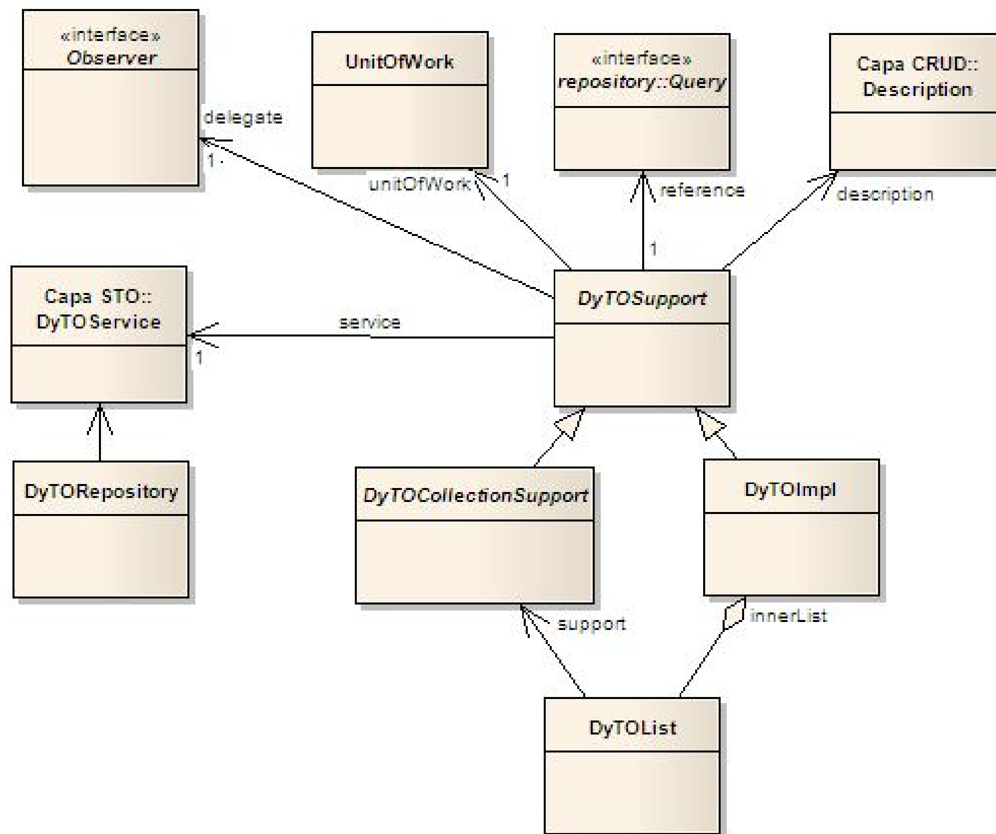


*Figura 20 - Diagrama de clases – Capa CRUD*

## Capa de DyTOs

Esta capa es la responsable del manejo de los objetos de transferencia de datos dinámicos (DyTOs). En ella se realizarán las operaciones necesarias para la creación, manipulación e interpretación de las distintas piezas que conforman un DyTO, tales como el log de cambios realizados (Unit Of Work), la carga de colaboradores o colecciones por demanda, atributos calculados, etc.

Es decir, se encarga de la manipulación y conversión de los objetos de la capa inferior en los objetos DyTOs que serán enviados al cliente mediante la capa de servicios y viceversa.

Esta capa delega en la capa subyacente tanto la obtención de los objetos del modelo de dominio como la aplicación de los cambios realizados en la capa superior.

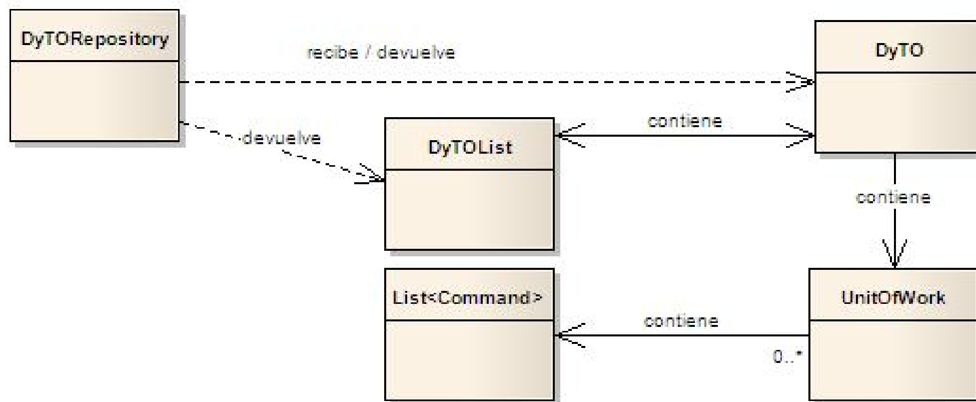


***Figura 21 - Capa de DyTOs***

Un DyTO cuenta con las siguientes características y funcionalidades:

- Contiene un registro de todos los cambios realizados sobre el DyTO (patrón UnitOfWork). Con cambios, nos referimos a cualquier modificación de sus atributos. Gracias a este registro, podemos saber en todo momento si el DyTO sufrió modificaciones en el cliente. Éste registro es el que viaja del cliente al servidor para realizar las modificaciones sobre el objeto de dominio, evitando así enviar todo el objeto.
- Soporta la carga por demanda de referencias lazy, es decir, podemos definir que colaboradores o colecciones del DyTOs serán cargadas solamente si son necesarias en el cliente, y no en otro caso.
- Permite la definición de atributos calculados, los cuales residen exclusivamente en el cliente. Con esto permitimos contar con cálculos necesarios en el cliente, sin tener que realizar requerimientos extras al servidor, salvo el de la carga del DyTO en el cliente.
- También permite deshacer los cambios realizados en él, así como obtener la identificación del objeto que está representando.
- Además, todo DyTO implementa el patrón de diseño Observer, lo cual le permite observar a cada uno de los DyTOs que colaboran con él. A su vez puede, ser observado por otro DyTO y otros objetos que deseen enterarse de los cambios que lo afecten.

### *Principales componentes*

Dentro de esta capa encontramos los componentes necesarios para proveer las características y funcionalidades listadas anteriormente:

<u>DyTOImpl</u>

Es la implementación de los objetos de transferencias dinámicos.

<u>DyTOList</u>

Representa a una lista de DyTOs y es, en sí mismo, un DyTO, lo que indica que se controlan las modificaciones que se realizan sobre la lista, tal como el agregado y la eliminación de elementos.

<u>DyTOMap</u>

Representa un mapa de DyTOs y es, en sí mismo, un DyTO, lo que indica que se controlan las modificaciones que se realizan sobre el mapa, tal como el agregado y la eliminación de elementos.

<u>DyTOSupport</u>

Implementa la funcionalidad común a cualquier tipo de DyTO, por ello DyTOImpl, DyTOList y DyTOMap delegan parte de sus responsabilidades en ella.

<u>DyTOCollectionSupport</u>

Implementa comportamiento común para las colecciones de dytos, DyTOList y DyTOMap delegan parte de su funcionalidad en ella.

<u>UnitOfWork</u>

Registra actualizaciones que se han realizado sobre el DyTO. Estas actualizaciones se registran como comandos, que serán enviados al servidor para realizar estas modificaciones en las entidades que corresponda.

<u>DyTORepository:</u>

Reside en el cliente de la aplicación y le solicita a DyTOService la información retornada en los StateTransfer Objects. Éstos son convertidos a DyTOs, listas de DyTOs (DyTOList) o mapas de DyTOs (DyTOMap).

Subject Map y Observable Delegate:

Es el mecanismo que permite que una modificación en un DyTO que no provino de la UI, sino de un procesamiento interno, cause el refresco automático del componente gráfico que le está mostrando el DyTO al usuario. Si se elimina esta funcionalidad del DyTO es responsabilidad del proceso que modificó el DyTO refrescar la UI de manera explícita.

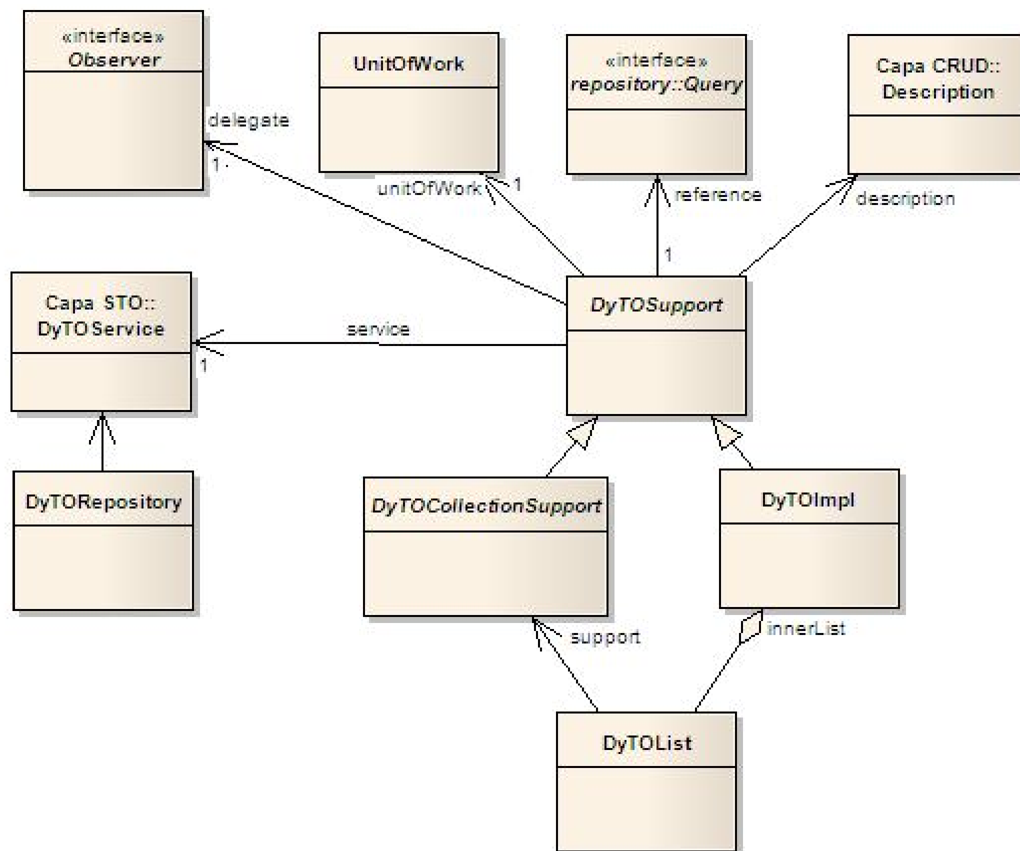


***Figura 22 - Diagrama de clases - DyTO***

**Capa de State Transfer Objects**

Esta es la capa de más bajo nivel. Su responsabilidad es trasladar información del servidor al cliente y cambios desde el cliente al servidor. Es la que interactúa con la capa de persistencia o repositorio de objetos, tanto para obtener los datos como para almacenar las actualizaciones en el almacenamiento persistente.

A partir de las descripciones generadas en la capa de servicios CRUDs y del resultado devuelto por el repositorio, esta capa se encarga de crear los objetos de transferencia entre el servidor y el cliente, cuyo estado puede ser un subconjunto del estado del objeto de dominio que representa. Este objeto de transferencia lo denominaremos StateTransferObject (STO).

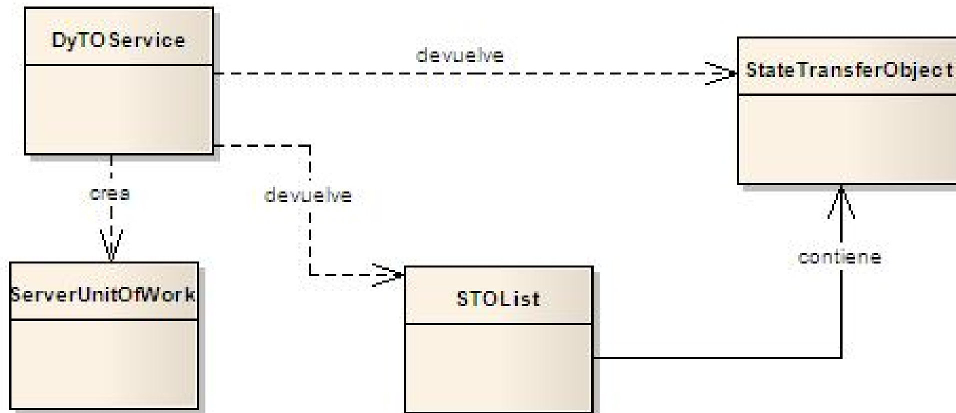


***Figura 23 - Capa de State Transfer Object***

Esta capa además, permite modificar un objeto de dominio mediante el procesamiento del registro de modificaciones recibido desde la capa superior (UOW). Cada cambio que figura en el registro se materializa en el objeto de dominio para luego ser persistido utilizando el repositorio.

### *Principales componentes*

Se identifican los siguientes participantes:

Query, Reference

Se usan estas clases para referenciar a objetos en el repositorio configurado para el servicio.

StateTransfer Object

Es el objeto de transferencia entre el servidor y el cliente. Contiene principalmente un mapa que asocia el nombre de la propiedad al valor de la misma. Estos valores pueden ser datos simples, otro StateTransfer Object, una STOList o un STOMap (estos dos últimos están representados por la relación de agregación).

Sin embargo, si la propiedad apunta a otro StateTransfer

Object, el valor de la misma es una referencia que “apunta” al objeto de dominio cuyo estado es contenido en el STO.

Con el objetivo de no replicar las instancias de los STO, se mantiene un Identity Map que asocia referencias a instancias únicas de State Transfer Objects.

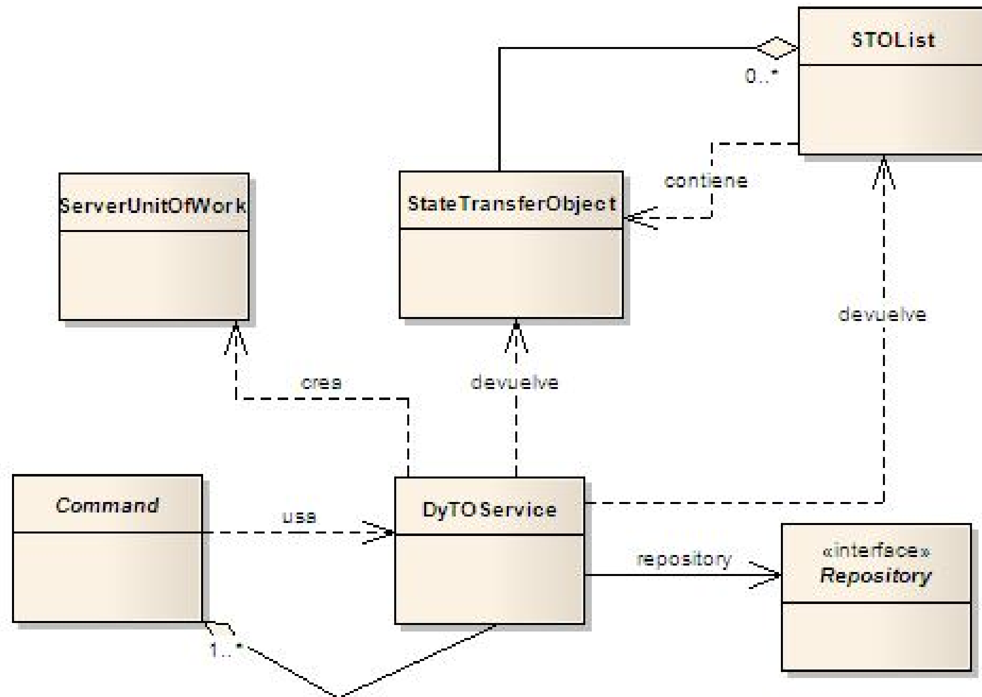


***Figura 24 - Diagrama de clases - State Transfer Object***

<u>STOList y STOMap</u>

Son colecciones paginadas de StateTransfer Objects.

<u>DyTOService y DyTOServiceImpl</u>

Son la interface e implementación del servicio de DyTOs. Utiliza la capa de acceso al almacenamiento persistente y devuelve StateTransfer Object, STOList y STOMap en base a los objetos del modelo de dominio.

Procesa las actualizaciones recibidas como listas de comandos.

La interfaz de este servicio está planteada en función de las necesidades específicas de la transferencia de objetos y cambios en los mismos, con el menor overhead posible. El protocolo resultante, por más que se exprese de manera similar al de los repositorios, utiliza objetos diseñados para restringir al

mínimo el estado transferido.

La complejidad resultante de usar objetos especializados para cada tipo de operación, hace que esta capa no sea muy conducente a ser usada por desarrolladores de aplicaciones. No se recomienda el uso del servicio directo en una aplicación, pero sí puede usarse como mecanismo básico de transferencia de estado usado por un framework de más alto nivel, como por ejemplo la capa de DyTOs.

## Integración entre capas

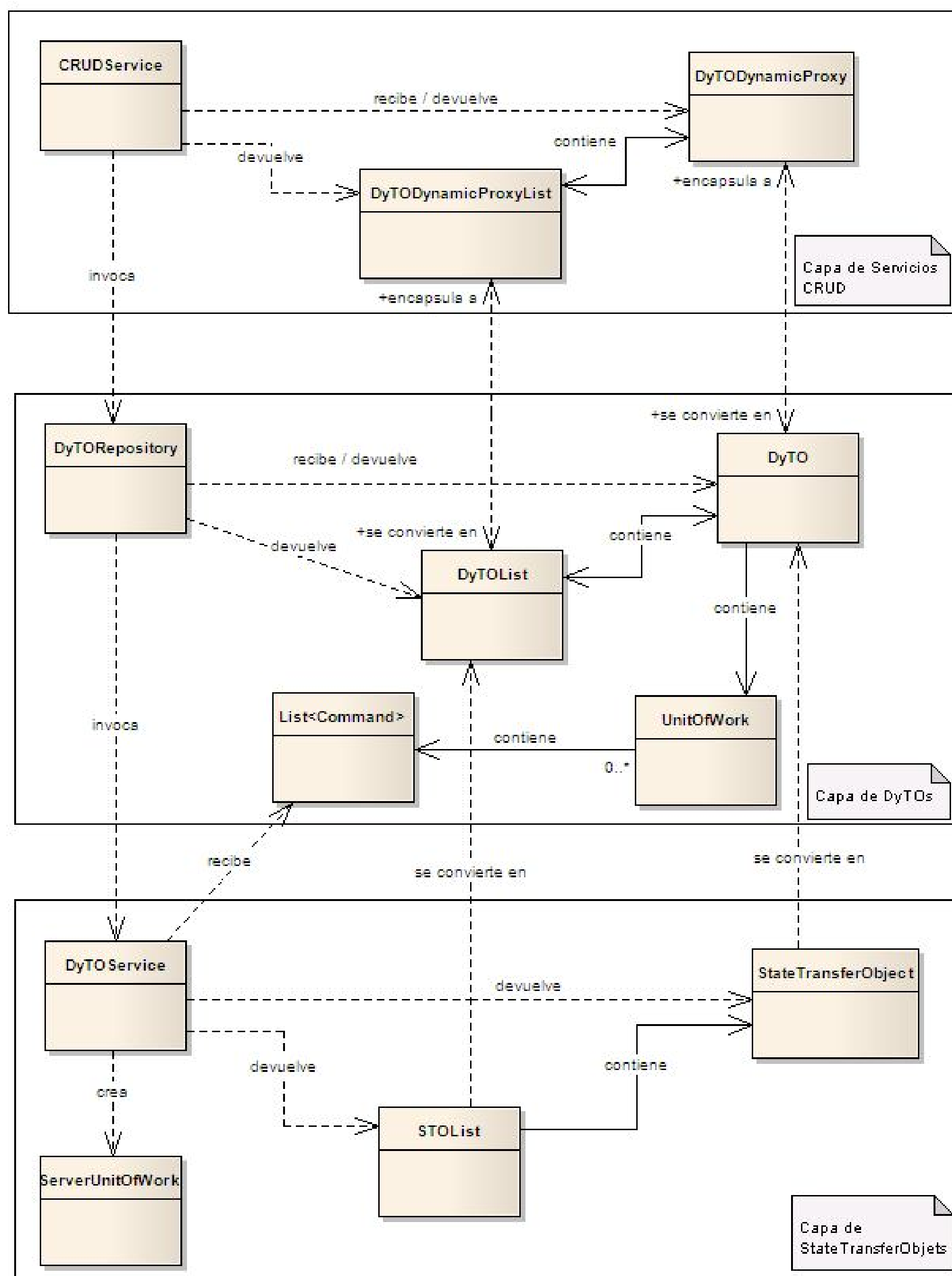


*Figura 25 - Interacción entre capas*

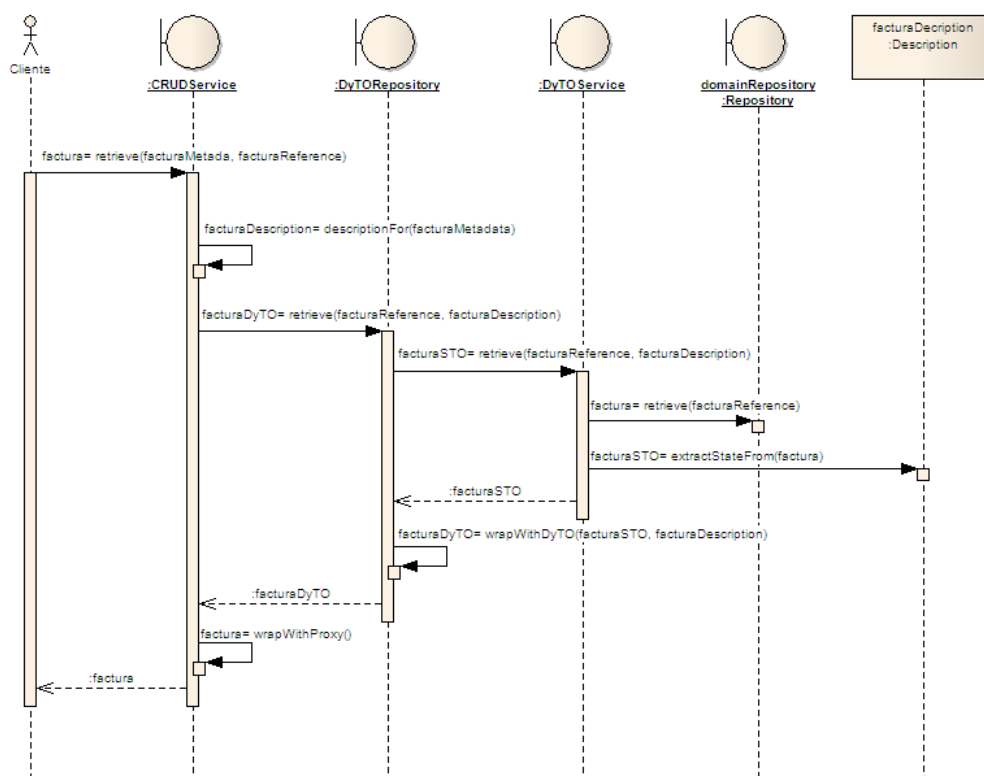


*Figura 26 - Diagrama de secuencia – Operación Retrieve*

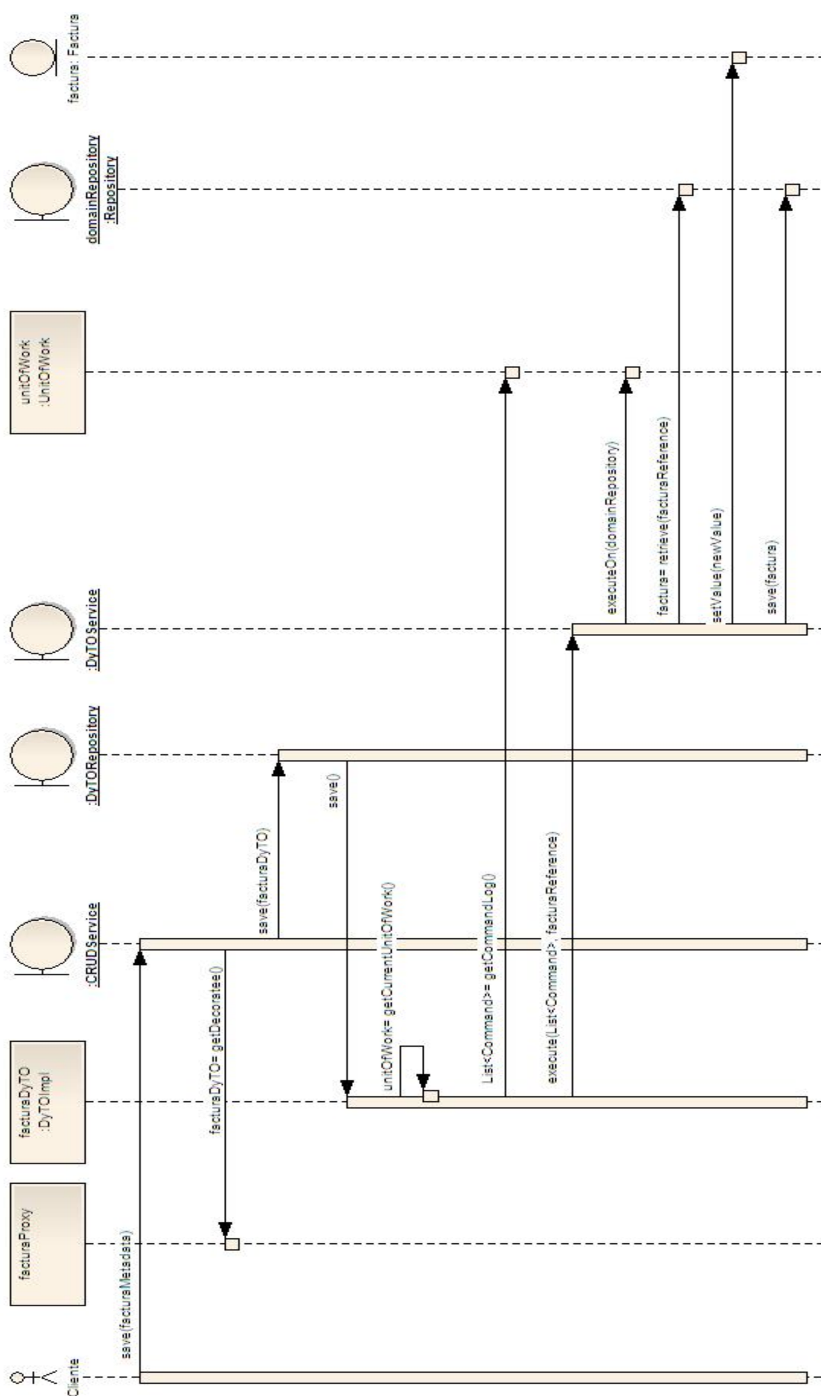


***Figura 27 - Diagrama de secuencia – Operación Retrieve***

## Meta-Información

En incisos anteriores se mencionó la necesidad de contar con una meta-información para poder identificar cómo y de qué manera hacer el mapeo de los objetos del modelo de dominio a los DyTOs que se envían al cliente. Definiremos aquí qué meta-información es necesaria para cumplir con los requerimientos del framework.

Anteriormente explicamos que generalmente no es necesario llevar al cliente todo el grafo tal cual está en los objetos de dominio del servidor. Alcanza entonces con que al cliente viaje sólo un sub-grafo del grafo.

Por esto, lo primero que se debe definir es una manera de especificar qué propiedades del objeto de dominio vamos a querer llevar al cliente.

Para una correcta división de las capas de arquitectura, es importante que no se indique el objeto de dominio del cual se obtienen los datos, sino que esto se realice en el momento que se recupera la información.

Un DyTO debe tomar sus datos a partir de un único objeto de dominio principal, pudiendo combinar (o aplanar) información de todos los objetos relacionados con él. Pero no es posible mezclar en un DyTO información de múltiples objetos de dominios independientes (sin ninguna relación entre ellos).

### Tipos de meta-información necesaria

Una vez definidas cuáles propiedades son importantes en el cliente, vamos a necesitar un mecanismo para dar información adicional sobre cierto tipo de propiedades. Es decir, además de especificar que al cliente debe viajar el atributo X que es una colección, también necesitaremos decirle al framework si ese atributo de tipo colección va a cargarse bajo demanda, utilizando paginado, etc.

A continuación se describe el conjunto de meta-información necesaria de acuerdo al tipo de la propiedad que viaja al cliente.

<u>Propiedades simples</u>

Las propiedades simples no tienen meta-información.

<u>Propiedades calculadas</u>

Si un DyTO necesita contener propiedades calculadas, se deberá proveer el mecanismo para dar soporte a esto, como por ejemplo algún objeto extra encargado de realizar el cálculo, y también definir la información necesaria para indicarle esto al framework.

Propiedades del tipo DyTO

Para los objetos colaboradores, otros DyTOS, es necesario especificar si se llevará al cliente siempre o solo bajo demanda.

Propiedades del tipo colección

Para una propiedad con una colección de DyTOs, se va a requerir especificar la siguiente información:

- Indicar si la colección se retorna al cliente de manera paginada o no.
- Como consecuencia de lo anterior, en caso de ser paginada, deberemos indicar también el tamaño que va a tener cada página.
- Indicar si la colección se enviará al cliente bajo demanda.
- En este tipo de propiedades es necesario también que se especifique el tipo de los objetos de la colección para que de esta manera el framework sepa cómo construir los elementos de dicha colección.
- En caso de tratarse de una colección, se debe poder especificar si dicha colección se trae al cliente como una lista, un conjunto o un mapa.
- Cuando se trata de una colección del tipo Mapa, necesitaremos además indicar si las claves del mapa son otros DyTOs o son simplemente tipos primitivos, efectos de diferenciar cómo construir dichas claves.

Más meta-información:

Cuando la propiedad del DyTO tiene distinto nombre que la propiedad del objeto de dominio que representa, se debe indicar como navegar el grafo de colaboradores asociado al objeto de dominio para obtener dicha propiedad. A menos que se especifique lo contrario, se asume que la propiedad del DyTO se obtiene de la propiedad con el mismo nombre en el objeto de dominio.

### Implementación propuesta

Existen en la actualidad gran cantidad de herramientas y tecnologías con la cual podríamos representar esta información.

Para el objetivo de este trabajo utilizaremos una combinación de interfaces, las cuales indican qué propiedades deben viajar al cliente; y mediante el uso

de anotaciones agregar la información extra necesaria para, por ejemplo, indicar el tamaño de las páginas en caso de tratarse de colecciones paginadas o indicar si la propiedad se enviar al cliente bajo demanda.

Para poder manipular los datos en el sistema desde el cliente, sería necesario entonces construir una interface que contenga:

- Métodos para acceder y modificar cada uno de los atributos sobre los cuales se quiere trabajar.
- Meta información mediante el uso de anotaciones que especifique la forma de obtención o comportamiento de esa información y la declaración de datos calculados, si los hubiere.

El framework proveería una interface del tipo marca, pada poder identificar que se trata de una interface con información perteneciente al framework, y no cualquier otra perteneciente a la aplicación que se esté desarrollando. Esta interface la llamamos DyTOModel, la cual debe extender todas las interfaces con este tipo de información.

## Transacciones largas

Uno de los requerimientos mencionados que debería cumplir este framework es ser capaz de resolver las operaciones que tienen características de transacciones largas o de negocios, es decir, que abarcan más de un ciclo pedido-respuesta al servidor de aplicaciones.

Veremos entonces cómo el conjunto de las siguientes funcionalidades del framework dan una solución integral a este requerimiento.

### Unit of Work

Se puede decir que un DTO dinámico está siempre involucrado en una transacción larga o de negocio. Para resolver este problema se decidió utilizar los principios del patrón Unit of Work [Fow02].

Un DyTO posee entonces una UnitOfWork la cual actúa como log de la transacción. En un grafo de objetos cada uno tendrá su propia UOW en la cual se irán registrando todas las operaciones que se van realizando sobre el DyTO.

La forma en que se registran las operaciones es a través de comandos (patrón Command [Gam02]) existiendo un tipo de comando distinto para cada tipo de operación. Si bien cada DyTO registra las operaciones realizadas sobre si mismo guardando su propio log de comandos, es necesario conocer el orden preciso en el cual cada DyTO fue modificado. Para esto se mantiene un índice único de comandos el cual permite conocer

el orden en el cual se realizaron cada una de las acciones.

Al finalizar la transacción, se consolidan las UOW de todos los DyTOs pertenecientes al grafo armando un único log, el cual contiene todos los comandos ordenados según el índice mencionado. Una vez hecho esto, se envían todos los cambios al servidor en forma de un único log de comandos. Luego se limpia la UOW para iniciar una nueva transacción.

### *Tipos de comandos*

Como explicamos más arriba, todas las modificaciones realizadas a un DyTO (o a sus DyTOs colaboradores), se materializan como comandos que se agrupan en la UOW del DyTO. [Kir04]

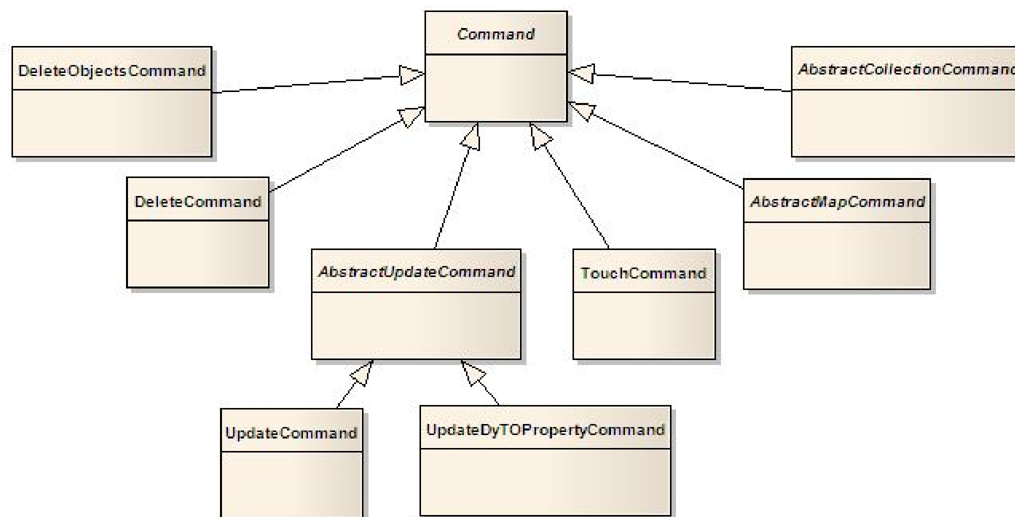


***Figura 28 - Comandos básicos***

De acuerdo al tipo de actualización que se realice, se genera un tipo de comando diferente. El framework dará soporte a los siguientes tipos de actualizaciones:

- Actualización de propiedad simple: ***UpdateCommand***
- Reemplazo de una propiedad DyTO: ***UpdateDytoPropertyCommand***
- Eliminar un objeto (hacerlo transient): ***DeleteCommand***
- Eliminar un conjunto de objetos: ***DeleteObjectsCommand***
- Marcar un objeto como modificado: ***TouchCommand***

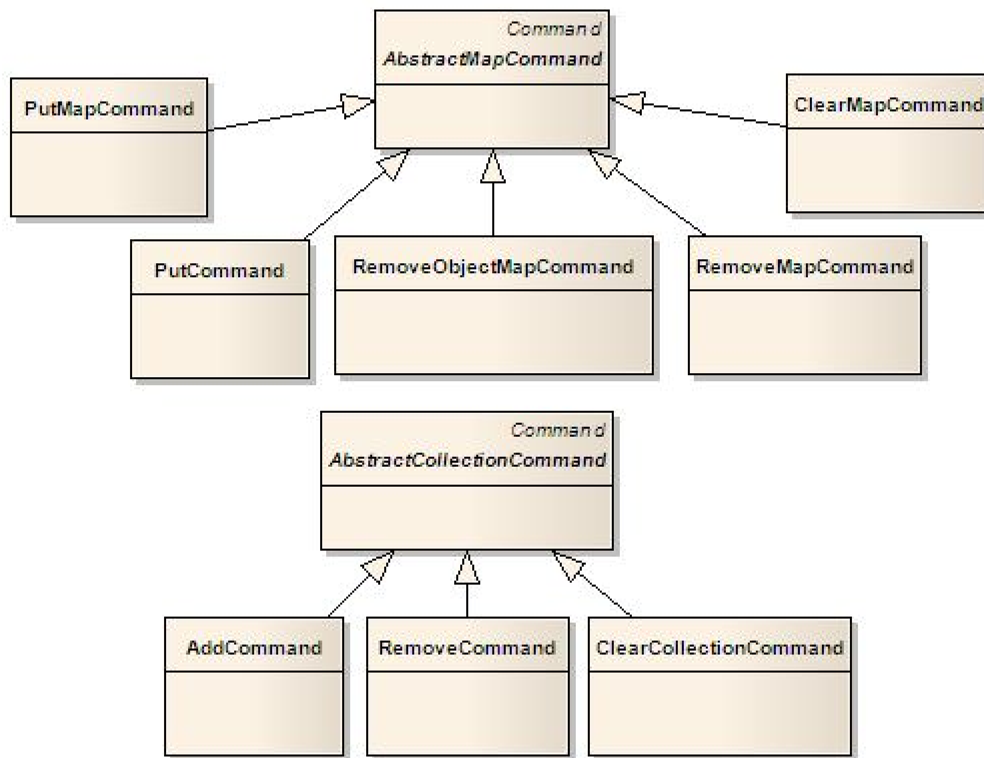


***Figura 29 - Comandos sobre colecciones***

Y los siguientes comandos para el manejo de colecciones:

- Agregar un objeto que es un Dyto a un map: ***PutMapCommand***
- Agregar un objeto que no es un Dyto sino un dato simple (ejemplo: string) a un map: ***PutCommand***
- Remueve el objeto que se encuentra bajo la clave "key" dentro del map apuntado por la referencia: ***RemoveObjectMapCommand***
- Remueve el objeto que se encuentra bajo la clave "key" dentro del map apuntado por la referencia, accediendo para ello al cache de la UnitOfWork: ***RemoveMapCommand***
- Remover todos los elementos de un map: ***ClearMapCommand***
- Agregar un elemento DyTO a una colección: ***AddCommand***
- Remover un elemento de una colección: ***RemoveCommand***
- Remover todos los elementos de una colección: ***ClearCollectionCommand***

## Deshacer cambios (UNDO)

El hecho de manejar una UOW con comandos que reflejan las operaciones, abre la posibilidad de brindar una forma de deshacer los cambios que se van realizando en el cliente. Esta necesidad surge para aquellos casos en que la transacción de negocio involucra varias pantallas, permitiendo volver atrás los cambios cuando se decide 'Cancelar' dicha pantalla.

El mecanismo elegido para brindar la posibilidad de UNDO es la definición de *checkpoints,* los cuales guardan un registro de las modificaciones realizadas sobre un DyTO permitiendo deshacer las mismas. Para esto, todos los cambios que se realicen en un DyTO y en sus colaboradores generan comandos con acciones inversas y se guardan en el checkpoint actual del DyTO.

Si se quieren cancelar los cambios hechos en una ventana, el botón de cancelar debe llamar a la operación undo sobre el DyTO, provocando que el checkpoint ejecute todos los comandos para deshacer los cambios.

El hecho de definir checkpoints y guardar en él los comandos que permitan deshacer la operación, nos da la posibilidad de definir varios checkpoints aceptando o deshaciendo los cambios realizados después de haberlo definido. Es decir, esto nos permitiría definir un checkpoint por cada ventana anidada en la que se requiera cancelar los cambios.

El mecanismo entonces es el siguiente, cada vez que se quiera abrir una nueva ventana se debería establecer un checkpoint. Cuando se aceptan los cambios de la ventana abierta, el checkpoint se ignora. Si se cancela, se vuelve atrás el estado de los DyTOs modificados, quedando los mismos tal como estaban antes de definir el checkpoint.

## Manejo de concurrencia

Un tema muy importante que se debe resolver es cómo se maneja la concurrencia.  En este inciso se detalla la forma en la cual nuestro framework tratar este tema.

Durante la modificación de un objeto en el cliente se realiza una marca en el mismo para indicar que ha sido modificado, de esta manera se asegura que cuando se quiere grabar, si hay un conflicto de concurrencia la operación falle. Esto se realiza por cada uno de los objetos y no afecta a los objetos relacionados con él (por ejemplo: si se tiene un objeto padre que contiene una colección de hijos y se modifica uno de los hijos, sólo ese hijo se marca como modificado, pero no así el objeto padre). El único caso que marca como modificado el objeto principal (padre) es cuando tiene como atributo una colección, y a estas colecciones se le agregan o eliminan elementos, ya que la modificación del conjunto de elementos se considera como una

actualización del padre. Sin embargo, si se modifica el estado de uno de los elementos de la lista o mapa, el padre no se considera actualizado.

Por lo tanto, si para un caso de uso se necesita mantener la integridad de la información de una entidad de negocio compuesta por más de un objeto, es responsabilidad del caso de uso la implementación de esta restricción. Por ejemplo: si se modifica un ítem de una factura se considera que la factura fue modificada y no se quiere que otro usuario pueda modificar en forma concurrente los datos de cabecera o cualquier otro ítem de la factura.

### Lockeo optimista

Todo DyTO mantiene una referencia al objeto de dominio fuente. Esta referencia está compuesta básicamente por el tipo del objeto (su clase), el identificador y la versión. Al momento de ejecutar las actualizaciones en el servidor, se verifica la versión del objeto con respecto a la referencia. Si alguna de las versiones de los objetos modificados en el grafo no concuerda con su referencia, se aborta la transacción.

El manejo de versionado de los objetos de dominio que realizan los frameworks de persistencia, consiste en incrementar la versión del objeto si el estado del objeto cambió. Quedando fuera de este manejo el caso en donde se modifican el estado de los objetos contenidos en una colección.

Para solucionar esta limitación, se debe proveer la manera de marcar un objeto como modificado, incrementando por ejemplo un número de versión alternativo. Este atributo debe estar mapeado por el framework de persistencia elegido en los casos en que se requiera implementar la actualización recursiva.

Se implementa entonces en todos los métodos que cambien el estado de un DyTO un mecanismo para que los objetos de dominio sean informados cuándo deben cambiar su versión.

En el cliente, cada vez que se realiza una modificación sobre el DyTO se agrega un encargado de realizar esta operación en el objeto de dominio.

Como explicábamos más arriba, si para un caso de uso se necesita mantener la integridad de la información de una entidad de negocio compuesta por más de un objeto, es responsabilidad del caso de uso la implementación de esta restricción.

Esto no se realiza en la definición del dyto, sino que es un comportamiento de los objetos de dominio, los cuales deben dar soporte a este tipo de versionado de acuerdo a la política de concurrencia definida para la entidad de negocio. El dyto entonces sólo se encarga de notificar cuál es el objeto que se modificó, y los objetos de dominio se deben encargar de propagar estos cambios a todos los objetos relacionados según la política de

concurrencia, para evitar que haya conflictos con las modificaciones de otros usuarios.

## Referencias Lazy y carga por demanda

Una ventaja importante que tiene este framework con respecto a implementaciones del patrón DTO es la carga por demanda de objetos enviados al cliente. Es posible especificar qué objetos serán cargados completamente y cuáles serán sólo referenciados. Esta información se encuentra especificada mediante la meta-información.

Los DyTOs comienzan su vida conteniendo referencias en las propiedades que apuntan a otros DyTOs. Estas referencias son reemplazadas por DyTOs construidos bajo demanda a partir del STO. Si el STO no ha sido cargado, se dispara la carga haciendo un pedido al servidor para armar el STO correspondiente.

Luego se encapsular al STO en un DyTO, se reemplaza la referencia por el DyTO en el mapa de estado y se agrega el DyTO al Identity Map [Fow02] correspondiente, presente para garantizar la unicidad de los objetos en el grafo del DyTO. Todos los DyTOs comparten la referencia al mismo Identity Map.

La posibilidad de usar “Referencias Lazy” (a objetos que se cargarán y enviarán al cliente bajo demanda) no es exclusiva de las propiedades de tipo DyTO. Las propiedades que apunten a colecciones tendrán la oportunidad de usar también la carga por demanda, permitiendo de esta manera implementar el paginado en las colecciones de objetos. Las propiedades de los DyTOs que devuelven colecciones tienen inicialmente un STOList. El framework construye una implementación de lista (DyTOList), basada en un decorador de listas que permite decorar los elementos contenidos en la medida que estos son requeridos.

Una STOList tiene sólo StateTransfer Objects. Una DyTOList encapsula (en la medida que le son pedidos), los StateTransfer Objects de la STOList subyacente.

Cuando se requiere un elemento que no se encuentra en las páginas que actualmente maneja el STOList, entonces el framework dispara el pedido de la siguiente página y la misma pasa a formar parte del STOList. Es importante tener en cuenta que por más que se requiera un sólo objeto de la página siguiente, igualmente se traerán las referencias a todos los objetos que formen parte de esa página. Esto evita realizar  sucesivos pedidos al servidor y se basa en el hecho de que la probabilidad de necesitar el resto de los elementos de esa página es muy alta.

En todo momento se sabe si la lista que está en el cliente está completa o restan traer elementos. Esto es posible gracias a que cuando se trae una

página, en el servidor se trata de traer el elemento siguiente perteneciente a la otra página. Si no se tiene éxito, entonces se sabe que la colección no tiene más elementos y no es necesario seguir trayendo páginas.

Todo este mecanismo es transparente tanto para el desarrollador como para el usuario del DyTO que contiene la lista.

# *Capítulo 7*

# Conclusión y trabajo futuro

*El trabajo estuvo centrado principalmente en investigar y proveer una solución integral al manejo de objetos persistentes en la componente cliente de Aplicaciones Enterprise. Hasta aquí hemos desarrollado todo el contenido de la investigación sobre el cual, ahora, se debe reflexionar y establecer nuevas metas. A continuación se presentan las conclusiones que se desprenden de la labor efectuada y, además, se plantearán las tareas relacionadas con la continuidad del trabajo en el tema.*

## Conclusiones

Como una primera conclusión podemos mencionar que el principal aporte de este trabajo de grado es, identificar y proveer una solución integral a los problemas recurrentes que se suscitan al manipular objetos de dominio en la componente cliente de Aplicaciones Enterprise de gran tamaño, en una arquitectura Cliente-Servidor. Esta solución es provista mediante la definición de un framework, el cual es fundamental en cualquier plataforma de desarrollo de este tipo de aplicaciones.

En los primeros capítulos se detallaron los conceptos básicos necesarios para comprender el contexto en el cual nuestro trabajo tiene valor. Si bien los problemas identificados los podemos encontrar en muchos tipos de aplicaciones, dejamos en claro que aparecen mayormente en el desarrollo de aplicaciones de gran tamaño, que intervienen en un Sistema Enterprise.

Se analizó además la evolución de las infraestructuras utilizadas para dar soporte al procesamiento de la información, pasando por diferentes tipos de arquitecturas, para finalmente detallar la arquitectura Cliente-Servidor; la cual cuenta con una serie de características que la convierten en la mejor opción para el tipo de aplicaciones tratadas en este trabajo, teniendo en cuenta además que es la más utilizada y extendida en los últimos tiempos.

En este punto nos encontramos con la necesidad de analizar los requerimientos funcionales y no funcionales que están presentes en este tipo de aplicaciones. Debido a esto se enumeraron algunos de los requerimientos funcionales comunes entre las Aplicaciones Enterprise, más allá de los surgidos de cada dominio particular; y se describieron en detalle los problemas no funcionales que deben tenerse en cuenta.

Basándonos en este análisis, encontramos que al intentar satisfacer los requerimientos no funcionales, surgen una serie de inconvenientes entre los que podemos mencionar:

- Los problemas de concurrencia
- El soporte a las transacciones largas
- El manejo de objetos de gran tamaño y carga por demanda
- La posibilidad de deshacer operaciones

Cada uno de estos temas fue tratado en profundidad, detallando su origen y planteando distintas alternativas de solución, lo que permitió proveer las bases para definir una arquitectura que de soporte a todos estos temas utilizando soluciones "exitosas", es decir, soluciones ampliamente probadas en el desarrollo de aplicaciones.

Llegamos al punto en donde contamos con las herramientas necesarias para diseñar un framework capaz de cumplir con el objetivo del trabajo y proveer los lineamientos básicos para definir una plataforma de desarrollo orientada al desarrollo de Aplicaciones Enterprise sobre arquitecturas Cliente-Servidor.

## Trabajo futuro

La realización de este trabajo de grado mostró un conjunto de soluciones a determinados problemas y definió para tal fin un framework que es parte fundamental dentro de una plataforma de desarrollo, la cual es la encargada de definir los lineamientos y estándares que ayudan a la simplificación del desarrollo.

Sin embargo, finalizado el trabajo, se detectan ciertos disparadores que motivan la continuidad de la labor en el tema. El trabajo futuro entonces, incluye las siguientes actividades y desarrollos:

### Lenguaje específico para meta información (DSL)

Uno de los trabajos a futuros que se desprenden rápidamente, es definir un lenguaje específico para la representación de la meta información, necesaria para configurar el comportamiento del framework. Este tipo de lenguajes lo podemos catalogar dentro de un “Lenguaje Específico del Dominio” (DSL – Domain Specific Language), ya que tiene como foco la implementación del código de configuración. [Fow10] [Fow06]

Este lenguaje podrá estar definido en términos del lenguaje de propósito general utilizado para la implementación del framework, como pueden ser Java o C#; o utilizando algún lenguaje de marcas como por ejemplo XML.

Estas opciones están catalogadas como “DSLs Externos” [Fowler infoQ], ya que además del lenguaje es necesario contar con las librerías necesarias para procesar dicho lenguaje.

En el **Anexo A** daremos una especificación posible de este tipo de DSL

utilizando XML, la cual cumple con todas la características mencionadas en los capítulos previos, necesarias para expresar la meta-información utilizada por el framework.

Otra alternativa en relación a la definición de un lenguaje para representar la meta-información, es definir un DSL interno, el cual por su definición nos ahorra tener que implementar alguna librería para procesarlo, debido a que es parte del lenguaje utilizado para definirlo, es decir, utiliza la misma sintaxis e interpretación. [Fowler infoQ]

### *Interpretaciones del DSL externo*

Como mencionamos, en caso de que el lenguaje definido para la representación de la meta-información necesite de librerías para poder procesarlo (DSL externo), otro trabajo a futuro es justamente la implementación de estas librerías para generar e interpretar la meta-información, de manera que pueda ser utilizada por clientes ricos implementados en varios lenguajes y/o plataformas a utilizar.

De este modo podríamos, utilizando el mismo lenguaje, hacer uso de la misma meta-información y utilizarla en cliente implementados, por ejemplo en Java, JavaScript, Flex o en cualquier otra plataforma que permita la implementación de este tipo de clientes ricos.

### Arquitectura dirigida por modelos (MDA)

Una Arquitectura dirigida por modelos (Model Driven Architecture) no es más que un framework para el desarrollo de software, definido por la OMG. La clave de una MDA es la importancia que toman los modelos en el desarrollo de software. El proceso de desarrollo está centrado fundamentalmente en el modelado del sistema.

El ciclo de vida de una MDA es similar a una ciclo de vida tradicional .Los artefactos en un MDA son modelos formales, como por ejemplo modelos que pueden ser interpretados por computadoras. Los siguientes modelos son el corazón de la MDA:

- Modelo independiente de la plataforma (PIM), es un modelo con un alto nivel de abstracción que es independiente de la tecnología de implementación.
- Modelo específico de la plataforma (PSM), es un modelo a medida para especificar el sistema en términos de construcciones que están disponibles en una tecnología de implementación específica. Un modelo PIM es transformado en uno o más modelos específicos.

- o Código, es una descripción (interpretación) de un sistema en términos del código fuente. Cada PSM es transformado en código.

Tradicionalmente la transformación de modelo en modelo, o de un modelo a código fuente es tradicionalmente realizada a mano. En contraste a esto, las transformaciones en una MDA son ejecutadas por herramientas. No es novedad que actualmente existen herramientas que permiten la transformación de un PSM a código (cualquier herramienta de modelado UML permite derivar a código en una variedad de lenguajes). Lo nuevo en MDA es la transformación automatizada entre modelos independientes de plataformas (PIM) a modelos más específicos.

Teniendo en cuenta la variedad de plataformas disponibles para la implementación de clientes ricos, un muy interesante trabajo a futuro sería definir los modelos y transformaciones necesarias para definir una MDA sobre la solución provista.

A continuación se muestra un esquema de alto nivel el cual muestra los modelos necesarios que intervendrían en esta arquitectura:

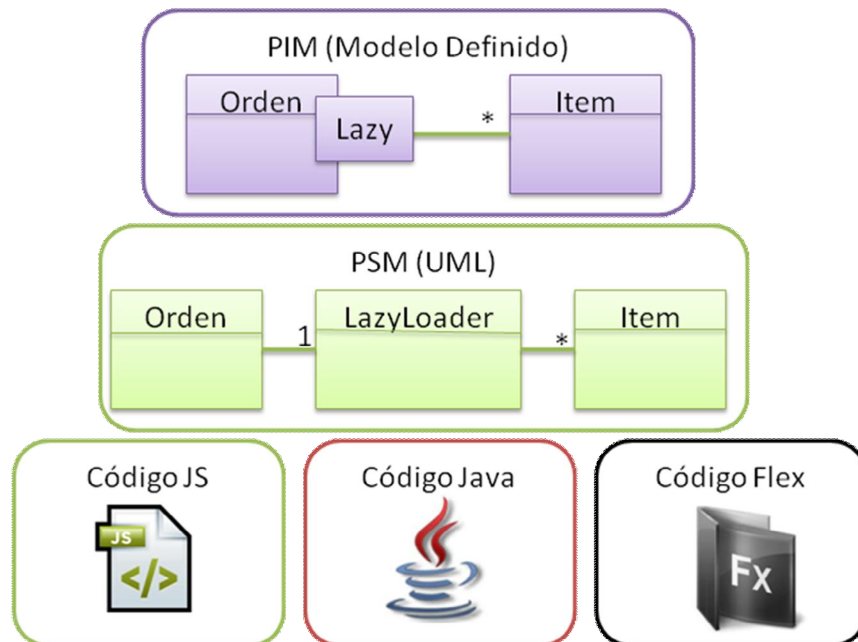


***Figura 30 - MDA: Diagrama de alto nivel***

**Implementación integral del Framework**

Teniendo en cuenta las consideraciones arriba descriptas, y como uno de los trabajos a futuro principales, se encuentra la necesidad de proveer una implementación completa del framework diseñado en este trabajo.

Para tal fin es menester seguir las recomendaciones y definiciones dadas en el capítulo anterior. Prestando especial atención a la división de las capas del framework y a cada uno de los requerimientos no funcionales solucionado en dicho diseño.

También se deberá acompañar esta implementación con el lenguaje utilizado para la definición, manipulación e interpretación de la meta-información a utilizar.

# Anexo A

## DSL para especificar la configuración de una entidad, respecto a cómo debe ser trasladada y tratada en el cliente

*Si bien existen muchas formas de especificar la información necesaria para la configuración del framework, en este anexo proveemos una definición básica de un posible lenguaje para especificar esta meta-información. El metalenguaje utilizado es XML.*

### Descripción

Una alternativa para especificar la meta-información es utilizar XML.

Al utilizar XML contamos con la ventaja de poder definir la meta-información en tiempo de configuración o ejecución, en lugar de tener que realizarlo en tiempo de diseño o implementación, como sería el caso por ejemplo del uso de interfaces u anotaciones Java.

Esto además nos permite centrarnos solamente en el objetivo que tiene la meta-información: definir las propiedades de una entidad relativas a cómo debe ser trasladada y tratada en el cliente.

Un lenguaje específico del domino (DSL – Domain Specific Language) evita forzar la escritura de la configuración utilizando un lenguaje de programación de propósito general. Por este motivo, XML puede ser tomado como un DSL debido a que está enfocado en la definición de la meta-información del framework y es independiente del lenguaje de propósito general utilizado para la implementación del framework.

Estrictamente hablando, este es un lenguaje específico del dominio "externo", dado que junto con la definición del nuevo lenguaje (XSD) deberemos dar soporte a la interpretación/compilación del mismo, el cual escapa al alcance de este trabajo.

### Esquema XML

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">
  <xs:element name="dytos">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="dyto" />
```

```
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="dyto">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="property" />
      </xs:sequence>
      <xs:attribute name="id" use="required" type="xs:NCName" />
      <xs:attribute name="dyto-type" type="xs:NCName" />
    </xs:complexType>
  </xs:element>
  <xs:element name="property">
    <xs:complexType>
      <xs:choice minOccurs="0">
        <xs:element ref="list" />
        <xs:element ref="map" />
        <xs:element ref="set" />
      </xs:choice>
      <xs:attribute name="name" use="required" type="xs:NCName" />
      <xs:attribute name="dyto-ref" type="xs:NCName" />
      <xs:attribute name="type" type="xs:NCName" />
      <xs:attribute name="lazy" type="xs:boolean" />
      <xs:attribute name="path" />
    </xs:complexType>
  </xs:element>
  <xs:element name="list">
    <xs:complexType>
      <xs:attribute name="pageSize" use="required" type="xs:integer"
/>
      <xs:attribute name="type" type="xs:NCName" />
      <xs:attribute name="calculated" type="xs:boolean" />
      <xs:attribute name="order" type="xs:NCName" />
    </xs:complexType>
  </xs:element>
  <xs:element name="set">
    <xs:complexType>
      <xs:attribute name="pageSize" use="required" type="xs:integer"
/>
      <xs:attribute name="type" type="xs:NCName" />
      <xs:attribute name="calculated" type="xs:boolean" />
      <xs:attribute name="order" type="xs:NCName" />
    </xs:complexType>
  </xs:element>
  <xs:element name="map">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="keys" />
        <xs:element ref="values" />
      </xs:sequence>
      <xs:attribute name="pageSize" use="required" type="xs:integer"
/>
      <xs:attribute name="calculated" type="xs:boolean" />
    </xs:complexType>
  </xs:element>
```

```
  <xs:element name="keys">
    <xs:complexType>
      <xs:attribute name="type" type="xs:NCName" />
      <xs:attribute name="dyto-ref" type="xs:NCName" />
    </xs:complexType>
  </xs:element>
  <xs:element name="values">
    <xs:complexType>
      <xs:attribute name="type" type="xs:NCName" />
      <xs:attribute name="dyto-ref" type="xs:NCName" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```

## XML de Ejemplo

```
<?xml version="1.0" encoding="UTF-8"?>

<dytos>

  <dyto id="persona" dyto-type="org.modelo.PeronaDyTO">

    <!-- Propiedades simples -->
    <property name="dni" />
    <property name="nombre" path="nombrePersona" />

    <!-- Propiedades simples en lenguajes tipados -->
    <property name="edad" type="java.lang.Integer" />
    <property name="calle" path="direccion.path"
type="java.lang.String" />

    <!-- Propiedades DyTO -->
    <property name="padre" dyto-ref="persona" />
    <property name="hermano" path="hermanos[0]" dyto-ref="persona"
/>

    <!-- Colecciones -->
    <property name="nombresDeMascotas">
      <list pageSize="10" type="java.lang.String" order="desc"
        calculated="true" />
    </property>

    <property name="direcciones" lazy="true">
      <set pageSize="10" dyto-ref="direccion" order="asc" />
    </property>

    <property name="hijos" lazy="true">
      <map pageSize="10">
        <keys type="java.lang.String" />
        <values dyto-ref="persona" />
      </map>
    </property>

  </dyto>
```

```
  <dyto id="direccion" dyto-type="org.model.DireccionDyTO">
    <property name="calle" type="java.lang.String" />
    <property name="nro" type="java.lang.Integer" />
  </dyto>
</dytos>
```

## Elementos

### dyto

Meta-Información utilizada para definir qué propiedades llevar al cliente y de qué manera serán recuperadas.

- ***Id (requerido):*** Propiedad utilizada para identificar la meta-información a utilizar en el momento de interactuar con el framework
- ***dyto-type (opcional):*** En el caso de utilizar un lenguaje tipado, indica el tipo del objeto en el cliente. Por ejemplo, si utilizamos Java como lenguaje, esto debería ser una interface.

### property

Meta-Información utilizada para especificar qué propiedad del objeto de dominio compondrá el DyTO en cuestión.

- ***name (requerido):*** Nombre de la propiedad del DyTO que contendrá la propiedad correspondiente del objeto de dominio, especificada en el atributo **path**.
- ***path (opcional):*** Path utilizado para navegar el grafo asociado al objeto de domino y recuperar el valor de la propiedad en cuestión. Por defecto tiene el mismo valor que el atributo **name**.
- ***type (opcional):*** Indica el tipo primitivo de la propiedad.
- ***dyto-ref (opcional):*** Indica el id del Dyto referenciado.
- ***Lazy (opcional):*** indica si la propiedad es recuperada por demanda. Por defecto el valor es falso. Este atributo es válido en caso de propiedades DyTOS y colecciones. En el caso de que la propiedad sea una colección, esto indica que se traerá paginada.
- ***calculated (optional):*** Indica que la propiedad no existe en el dominio, y es calculada para ser enviada al cliente. Este atributo es válido en caso de propiedades DyTO.

### list & set

Meta-Información utilizada para especificar qué propiedad es una lista o un conjunto.

- ***pageSize (opcional):*** Tamaño de las páginas en caso de que la propiedad sea lazy.
- ***order (opcional):*** Indica el orden con el cual es retornada la colección. Por defecto las colecciones son retornadas sin orden.
- ***calculated (optional):*** Indica que la colección no existe en el dominio, y es calculada para ser enviada al cliente.
- ***type (opcional):*** Indica el tipo primitivo de la propiedad.
- ***dyto-ref (opcional):*** Indica el id del Dyto referenciado.

### map

Meta-Información utilizada para especificar qué propiedad es un map.

- ***pageSize (opcional):*** Tamaño de las páginas en caso de que la propiedad sea lazy.
- ***calculated (optional):*** Indica que la colección no existe en el dominio, y es calculada para ser enviada al cliente.

### keys

Especifica si las claves del mapa son de tipos primitivos o DyTOs

- ***type (opcional):*** Indica el tipo primitivo de la propiedad.
- ***dyto-ref (opcional):*** Indica el id de la definición del Dyto referenciado.

### values

Especifica si los valores del mapa son de tipos primitivos o DyTOs

- ***type (opcional):*** Indica el tipo primitivo de la propiedad.
- ***dyto-ref (opcional):*** Indica el id de la definición del Dyto referenciado.

# *Bibliografía*

**[And01]** Anderson, Ross J., y Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems.* 1°. Wiley, 2001.

**[Bus07]**
Buschmann, Frank, Henney, Kevlin, y Schmidt, Douglas C. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing.* 1°. John Wiley and Sons, 2007.

**[Bus96]** Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, y Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns.* 1°. Wiley, 1996.

**[Cle01]** Clements, Paul, Rick Kazman, y Mark Klein. *Evaluating Software Architectures: Methods and Case Studies.* 1°. Addison-Wesley Professional, 2001.

**[Cle02]** Clements, Paul, y otros. *Documenting Software Architectures: Views and Beyond.* 1°. Addison-Wesley Professional, 2002.

**[Cor09]** Coronel, Carlos, Morris Steven, y Rob Peter. *Database Systems: Design, Implementation, and Management (with Bind-In Printed Access Card).* 9°. Course Technology, 2009.

**[Dav92]** Davenport, Thomas H. *Process Innovation: Reengineering Work Through Information Technology.* 1°. Harvard Business Press , 1992.

**[Erl07]** Erl, Thomas. *Service-oriented architecture: concepts, technology, and design.* 1°. Prentice Hall Professional Technical Reference, 2007.

**[Eva03]** Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* 1°. Addison-Wesley Professional, 2003.

**[Fla01]** Flanagan, David. *JavaScript: The Definitive Guide.* 4°. O'Reilly Media, 2001.

**[Fow02]** Fowler, Martin. *Patterns of Enterprise Application Architecture.* 1°. Addison-Wesley Professional, 2002.

**[Fow10]** Fowler, *Domain-Specific Languages.* 1°. Addison-Wesley Professional, 2010

**[Gam02]**
Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* 2°. Pearson Education, 2002.

**[Goe06]** Goetz, Brian. Java Concurrency in Practice. 1°. Addison-Wesley Professional, 2006.

**[Gru89]** Robert Edward Gruber, Massachusetts Institute of Technology. *Laboratory for Computer Science. Optimistic concurrency control for nested distributed transactions*. 1°. Laboratory for Computer Science, Massachusetts Institute of Technology, 1989.

**[Hai06]** Haines, Steven. *Pro Java EE 5 Performance Management and Optimization.* 2°. Apress, 2006.

**[Ham03]** Hammer, Michael, y James Champy. *Reengineering the Corporation: A Manifesto for Business Revolution (Collins Business Essentials).* 1°. Harper Paperbacks; Rev Upd edition, 2003.

**[Har99]** Harke, Dan. *Client/Server Survival Guide, 3rd Edition.* 3°. Wiley, 1999.

**[Hof99]** Hofmeister, Christine, y Robert Nord. *Applied Software Architecture.* 1°. Addison-Wesley Professional, 1999.

**[Hog11]** Hogan, Brian P. *HTML5 and CSS3: Develop with Tomorrow's Standards Today (Pragmatic Programmers).* 1°. Pragmatic Bookshelf, 2011.

**[Hoh03]** Hohmann, Luke. *Beyond Software Architecture: Creating and Sustaining Winning Solutions.* 1°. Addison-Wesley Professional, 2003.

**[Joh92]** Johnson, R.E. «Documenting Frameworks Using Patterns.» *OOPSLA '92 Proceedings.* 1992.

**[Kin04]** King, Gavin, y Christian Bauer. *Hibernate in Action: Practical Object/Relational Mapping.* 1°. Manning Publications, 2004.

**[Kir04]** Michael Kircher, Prashant Jain. Pattern-*oriented software architecture: Patterns for resource management*. 1° . John Wiley and Sons, 2004.

**[Mar10]** Mak, Gary. *Hibernate Recipes: A Problem-Solution Approach (Expert's Voice in Open Source)*. 1°. Apress, 2010.

**[Mar83]**
Martin, James. Managing the data-base environment. 1°. Prentice-Hall, 1983.

**[Mar86]** Martin, James. *Managing the Data Base Environment.* 1°. Prentice Hall, 1986.

**[Mca10]** McAffer, Jeff, Jean Michel Lemieux, y Chris Aniszczyk. *Eclipse Rich Client Platform.* 2°. Addison-Wesley Professional, 2010.

**[Ram02]** Ramakrishnan, Raghu, y Johannes Gehrke. *Database Management Systems.* 3°. McGraw-Hill Science/Engineering/Math, 2002.

**[Rum95]** Rummler, Geary A., y Alan P. Brache. *Improving Performance: How to Manage the White Space in the Organization Chart (Jossey-Bass Management).* 2°. Jossey-Bass, 1995.

**[Sch06]** Schlossnagle, Theo. *Scalable Internet Architectures.* 1°. Sams, 2006.

**[Sta02]** Stallings, William. *Cryptography and Network Security: Principles and Practice.* 3°. Prentice Hal, 2002.

**[Tip03]** Tipton, Harold F., y Micki Krause. *Information Security Management Handbook.* 5°. Auerbach Publications, 2003.

**[Vac09]** Vacca, John R. *Computer and Information Security Handbook (The Morgan Kaufmann Series in Computer Security).* 1°. Editado por John R. Vacca. 2009.

**[Wie06]** Wiegers, Karl. *More About Software Requirements: Thorny Issues and Practical Advice.* 1°. Microsoft Press, 2006.

**[Woo03]** Woolf, Bobby, y Gregor Hohpe. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* 1°. Addison-Wesley Professional, 2003.

## Recursos Web

**[Ant09]** Anthony, Patricio. JBossCommunity. Hibernate. *Pagination.* 2009. http://community.jboss.org/wiki/Pagination

**[Dat01]** Data & Object Factory, LLC. *Design Patterns.* 2001. http://www.dofactory.com/Patterns/Patterns.aspx

**[Fow06]** Fowler, Martin. *Introduction to Domain Specific Languages*. 2006 http://www.infoq.com/presentations/domain-specific-languages

**[Sun02]** Sun Microsystems. *Core J2EE Patterns - Transfer Object.* 2002. http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html

**[Raz09]** Razan, Paul. The code project. *Multilevel Undo and Redo Implementation in C#.* 2009. http://www.codeproject.com/KB/architecture/UndoRedoPart1.aspx
http://www.codeproject.com/KB/architecture/UndoRedoPart2.aspx
http://www.codeproject.com/KB/architecture/UndoRedoPart1.aspx