FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

TÍTULO: Arquitectura en capas: análisis y estudio de caso del modelo arquitectónico N-Capas y sus variantes.

AUTOR: Vázquez Cendron, Agustín
DIRECTOR ACADÉMICO: Dr. Antonelli, Leandro
DIRECTOR PROFESIONAL: Lic. Tarrío, Diego Fernando
CARRERA: Licenciatura en Sistemas

Resumen

La arquitectura de software es una pieza central del desarrollo de sistemas de software modernos. Su objetivo consiste en desarrollar sistemas grandes de forma eficiente y estructurada. Define los componentes que llevan a cabo alguna tarea de computación, sus interfaces y la comunicación entre ellos, garantizando el cumplimiento de los atributos de calidad. Esta tesina explica brevemente la evolución de las arquitecturas de software, deteniéndose y analizando en detalle el modelo N-Capas, el cual se profundiza a través de un estudio de caso donde se describe cada componente y se revisan las customizaciones llevadas a cabo.

Palabras Clave

Arquitectura de Software. Arquitectura en Capas. Aplicaciones Enterprise. Sistemas Distribuidos. Patrones de Diseño. Principios arquitectónicos.

Conclusiones

La arquitectura es el cimiento del sistema, sobre el cual se garantizará el cumplimiento de los atributos de calidad, permitiendo construir un software robusto y escalable. Es clave a la hora de diseñar la arquitectura atenerse a las buenas prácticas y a los principios y patrones arquitectónicos. El modelo N-Capas, se presenta como una gran alternativa a la hora de realizar diseños de calidad. Facilita la comprensión y mantenimiento de un sistema complejo debido a que se puede hacer foco sólo en una capa en especial, en lugar de intentar entender todo el sistema.

Trabajos Realizados

Se realizó una breve reseña de la evolución de las arquitecturas de software, definiendo de manera sintética algunas de las variantes más conocidas.
Luego, se presentaron y describieron los patrones de diseño y principios arquitectónicos más importantes relacionados con el diseño de una arquitectura.
Finalmente, se desarrolló un estudio de caso del modelo N-Capas aplicado a un sistema, analizando los detalles y customizaciones que se llevaron a cabo y justificando las decisiones tomadas.

Trabajos Futuros

Sobre la base de la arquitectura planteada y descrita en el estudio de caso, se dan lugar a líneas de trabajo futuro. De esta manera, se presentan dos posibles extensiones a la Capa de Servicios, las cuales otorgarían mayor robustez al sistema. La primera hace referencia a la gestión de versiones de dicha capa, lo que permitiría un manejo más eficiente y ordenado de cara a los consumidores de los servicios. La segunda, propone la creación de una capa extra de servicios orientada a consumidores externos o sistemas satélites.

Fecha de la Presentación: Noviembre del 2018

Universidad Nacional de La Plata

Facultad de Informática

ARQUITECTURA EN CAPAS:

ANÁLISIS Y ESTUDIO DE CASO DEL MODELO ARQUITECTÓNICO "N-CAPAS" Y SUS VARIANTES

Tesina de grado de la carrera Licenciatura en Sistemas

Director Académico:

Dr. Leandro Antonelli

Director Profesional:

Lic. Diego Fernando Tarrío

Alumno:

Agustín Vázquez Cendron - 10927/0

Índice General

ÍNDICE GENERAL 1

ÍNDICE DE FIGURAS 6

ÍNDICE DE TABLAS 6

1. INTRODUCCIÓN 7

1.1 Motivación 7

1.2 Sistemas Distribuidos 8

1.3 Aplicaciones Enterprise 10

1.4 Objetivo 11

1.5 Estructura de la Tesina.......... 11

2. EVOLUCIÓN DE LA ARQUITECTURA EN CAPAS 12

2.1 Sistemas Monolíticos 12

2.2 Cliente/Servidor y su evolución a N-Capas 13

2.2.1 Arquitectura Web: un caso particular del Sistema Cliente/Servidor ..15

2.2.2 Arquitectura Web en 3 capas: Web Server dinámico 16

2.3. Sistemas Distribuidos con Objetos 16

2.4. Integración de Aplicaciones 17

2.4.1 Integración Punto a Punto 18

2.4.2 Integración por adaptadores..........18

2.4.3 Mediador de Mensajes 19

2.5 Arquitectura Orientada a Servicios 20

2.5.1 Procesos de Negocio como Consumidores de Servicios..........21

3. PATRONES DE ARQUITECTURA.......... **24**

3.1 Patrones de Arquitectura de Aplicaciones de tipo Enterprise..........24

3.2 Patrones estructurales de mapeo Objeto-Relacional..........24

3.2.1 Identity Field..........24

3.2.2 Foreign Key Mapping……………………………………………………………24
3.3 Patrones de modelado de la complejidad del Dominio…………………….....25
3.3.1 Transaction Script ……………………………………………………………..25
3.3.2 Domain Model……………………………………………………………………25
3.4 Patrones arquitecturales de acceso a datos………………………………….26
3.4.1 Table Data Gateway…………………………………………………………... 26
3.4.2 Active Record…………………………………………………………………..26
3.4.3 Data Mapper……………………………………………………………………26
3.5 Patrones de comportamiento objeto-relacional……………………………27
3.5.1 Identity Map ……………………………………………………………………27
3.5.2 Lazy Load………………………………………………………………………27
3.6 Patrones de base ……………………………………………………………....27
3.6.1 Layer Supertype……………………………………………………………...27
3.7 Patrones de concurrencia …………………………………………………….27
3.7.1 Optimistic Offline Lock ……………………………………………………....27
3.8 Patrones metadatos en mapeo objeto-relacional…………………………..28
3.8.1 Metadata Mapping …………………………………………………………….28
3.8.2 Query Object …………………………………………………………………...28
3.8.3 Repository……………………………………………………………………...28
3.9 Patrones de presentación Web……………………………………………....28
3.9.1 Model View Controller……………………………………………………....28
3.9.2 Page Controller…………………………………………………………….....29
3.9.3 Front Controller……………………………………………………………....29
3.9.4 Application Controller ……………………………………………………….29
3.10 Patrones de distribución……………………………………………………..29
3.10.1 Remote Facade ……………………………………………………………....29
3.10.2 Data Transfer Object ………………………………………………………...30
3.11 Patrones de manejo de sesión………………………………………………..30
3.11.1 Client Session State ………………………………………………………….30
3.11.2 Server Session State ………………………………………………………...31

4. PRINCIPIOS ARQUITECTÓNICOS 33

4.1 Separación de Intereses 33

4.2 Encapsulación 34

4.3 Inversión de Dependencia 35

4.4 Dependencias Explícitas 36

4.5 Responsabilidad Única 37

4.6 Don’t Repeat Yourself (DRY) 37

4.7 Ignorancia de Persistencia 38

4.8 Contextos limitados 38

5. ESTUDIO DE CASO: ARQUITECTURA N-CAPAS APLICADA AL SISTEMA DE LA EMPRESA 40

5.1 Introducción al Sistema de la Empresa 40

5.2 Aspectos generales de la arquitectura del sistema 41

5.2.1 Modularización de Aspectos Transversales 41

5.2.2 OOP (Object Oriented Programming) 41

5.2.3 Componentes estándar 41

5.2.4 Testeo unitario de componentes 42

5.2.5 Integración con otros sistemas 42

5.2.6 Maximización de alineamiento con Negocio 43

5.3 Características Técnicas 43

5.4 Diseño Arquitectónico 44

5.4.1 DAL (Data Access Layer) 45

5.4.2 BL (Business Layer) 45

5.4.3 SL (Services Layer) 45

5.4.4 PL (Presentation Layer) 45

5.4.5 Development Foundation 46

5.4.6 Business Foundation 46

5.4.7 Business Core Extension Services 46

5.4.8 ESB (Enterprise Service Bus) 47

5.4.9 BPM (Business Process Management) 47
5.4.10 Frontend..47
5.4.11 Customized Frontend 48
5.4.12 External Systems..49
5.4.13 Data Access Services..49
5.4.14 Look Services..50
5.4.15 IoC Services (Inversion Of Control Services) 50
5.4.16 Security Services..50
5.4.17 Internationalization Services 50
5.4.18 Visibility/Privacy Services 50
5.4.19 Data Transformation Services..51
5.4.20 Personalization Services..51
5.4.21 Electronic Folder Services 51
5.4.22 Reporting Services..51
5.4.23 Workflow Services...52
5.4.24 User Defined Fields (UDFs) 52
5.4.25 Business Rules...52
5.5 Tecnologías utilizadas en la implementación de la Arquitectura.............52
5.6 Esquema detallado de la Arquitectura.................................. 56
5.6.1 Business Logic (Lógica de Negocio).......................................57
5.6.2 DAOs 57
5.6.3 Models.. 57
5.6.4 Services... 58
5.6.5 DTOs.. 58
5.6.6 Web Services y WCF Services.. 58
5.6.7 Datasources.. 58
5.6.8 UI.. 58

6. CONCLUSIONES 59

Trabajos Futuros 61

Gestión de versiones de la Capa de Servicios 61

Capa de Servicios extra para los diferentes consumidores del Backend 62

REFERENCIAS..63

Índice de Figuras

Fig. 1.1 Componentes de un Sistema Distribuido……………………………………..9

Fig. 2.1 Evolución tecnológica de los Sistemas de Información ……………………...13

Fig. 2.2 Sistema Distribuido en 2 capas o Cliente/Servidor clásico…………... ……..14

Fig. 2.3 Sistema Distribuido en N capas.………………………………………………..15

Fig. 2.4 Arquitectura Web simple.…………………………………………………….. 15

Fig. 2.5 Arquitectura Web en 3 capas.…………………………………………………16

Fig. 2.6 Mecanismo de integración punto a punto.……………………………………..18

Fig. 2.7 Mecanismo de integración por adaptadores.…………………………………19

Fig. 2.8 Mecanismo de integración por mediador de mensajes.……………………...20

Fig. 4.1 Grafo de Dependencia Directa.………………………………………………....35

Fig. 4.2 Grafo de Dependencia Invertido ………………………………………………..36

Fig. 5.1 Esquema general de la Arquitectura del Sistema…..………………………...45

Fig. 5.2 Componentes de la Arquitectura del Sistema (1) ………………………46

Fig. 5.3 Componentes de la Arquitectura del Sistema (2) ....……………………49

Fig. 5.4 Diagrama detallado de la Arquitectura……………….. ...……………………49

Índice de Tablas

Tabla 5.1 Tecnologías utilizadas en la implementación………………………………..56

Capítulo I

Introducción

1.1 Motivación

La arquitectura de software es una pieza central del desarrollo de sistemas de software modernos. Su objetivo consiste en desarrollar sistemas grandes de forma eficiente, estructurada y con capacidad de reuso. Define, de manera abstracta, los componentes que llevan a cabo alguna tarea de computación, sus interfaces y la comunicación entre ellos, brindando el soporte necesario para la construcción de los casos de uso, y garantizando el cumplimiento de los atributos de calidad.

Citando a Clements [5]:

"La arquitectura es, a grandes rasgos, una vista del sistema que incluye los elementos principales del mismo, el comportamiento de esos elementos según se los percibe desde el resto del sistema y las formas en que los elementos interactúan y se coordinan para alcanzar la misión del sistema."

La arquitectura forma parte del proceso de diseño de software, el cual a su vez, forma parte del proceso de desarrollo de software. Esta etapa comprende requerimientos, diseño, implementación, prueba y mantenimiento. Cualquier decisión tomada en el establecimiento de la arquitectura -por ser una decisión temprana en el proyecto que incide en el desarrollo- es muy costosa de modificar posteriormente. Es por este impacto que debe ser cuidadosamente definida.

Los sistemas de software basados en la Web han tenido un gran auge en los últimos años, sustentado a su vez en la mejora de las tecnologías de Internet, de cómputo distribuido, de los lenguajes basados en objetos y de las arquitecturas y componentes de hardware. Debido a este gran éxito, el desarrollo de aplicaciones Web ha crecido de forma notable abarcando áreas como comercio electrónico, redes sociales, banca en línea, seguros, entretenimiento, etc. Por su parte, la mayoría de los grandes sistemas de información, como las Aplicaciones Enterprise, paulatinamente fueron migrando a ambientes Web como parte de su evolución. Sin embargo, este tipo de aplicaciones necesitan cumplir con requisitos de calidad, rendimiento, usabilidad, escalabilidad, mantenimiento, accesibilidad, etc.

De esta manera, el desarrollo de Aplicaciones Enterprise implica lidiar tanto con las dificultades derivadas de la implementación de la lógica de negocio en componentes de software como con aspectos de diversa índole y de alta complejidad. La combinación de dichos aspectos entre sí y con la lógica del negocio propiamente dicho, contribuye a aumentar la complejidad del software final,

-considerando el software en el sentido amplio de la palabra (código fuente, documentación, etc.).

Esta complejidad afecta tanto a la implementación inicial de las soluciones de negocio como al mantenimiento de las mismas en tiempos, costo y calidad. Estos factores son críticos en ambientes de negocio cambiantes, ya que no contar con software de calidad, en los tiempos en que éste se necesita para brindar un diferencial de negocio y con un costo superior al esperado, genera pérdidas que afectan en gran medida a las organizaciones.

A raíz de esto, surge la necesidad de contar un una arquitectura base que:

1) brinde los componentes de software necesarios para cubrir los distintos aspectos transversales a la aplicación (seguridad, logging, auditoría, etc.);

2) brinde los componentes de software de base para construir la solución de negocio propiamente dicha (acceso a datos, servicios, etc.);

3) brinde los lineamientos de construcción de las soluciones mediante estándares, de modo de brindarle al software final atributos de calidad compartidos entre todas las aplicaciones construidas sobre dicha arquitectura (mantenibilidad, escalabilidad, etc.);

4) permita la reutilización de los componentes de software, reduciendo de esta manera los tiempos de construcción y garantizando la calidad mediante componentes adecuadamente probados;

5) permita ser incorporada fácil y rápidamente al proceso de desarrollo, de modo de minimizar la inducción necesaria para utilizar la misma; brindando de esta manera la posibilidad de incorporar fácilmente recursos de desarrollo que hagan uso de la arquitectura,

6) brindar soporte para maximizar el alineamiento entre TI (Tecnología de la Información) y el negocio, de modo que los cambios en el último puedan reflejarse en las soluciones, en tiempos acordes a las necesidades del negocio.

1.2 Sistemas Distribuidos

Los sistemas distribuidos pueden concebirse como aquellos cuya funcionalidad se encuentra fraccionada en componentes que al trabajar sincronizada y coordinadamente otorgan la visión de un sistema único, siendo la distribución transparente para quien hace uso del sistema.

En términos computacionales, se dice que un sistema distribuido es aquel cuyos componentes, de hardware o de software, se alojan en nodos de una red comunicando y coordinando sus acciones a través del envío de mensajes.

El concepto de componente como pieza funcional autónoma y con interfaces bien definidas alcanza al área del desarrollo de software pero también puede

concebirse dentro de otras disciplinas como la idea de una pieza que, cuando se combina con los demás componentes, forma parte de un todo.

Otra definición que se puede formular es:

“Un sistema en el cual las componentes de hardware y software se ubican en una red de computadoras, y comunican y coordinan sus acciones sólo por envío de mensajes.” [19]

En este sentido, Tenenbaum [20] afirma:

“Un sistema distribuido es una colección de dos o más computadoras independientes que coordinan su procesamiento a través del intercambio sincrónico o asincrónico de mensajes.”

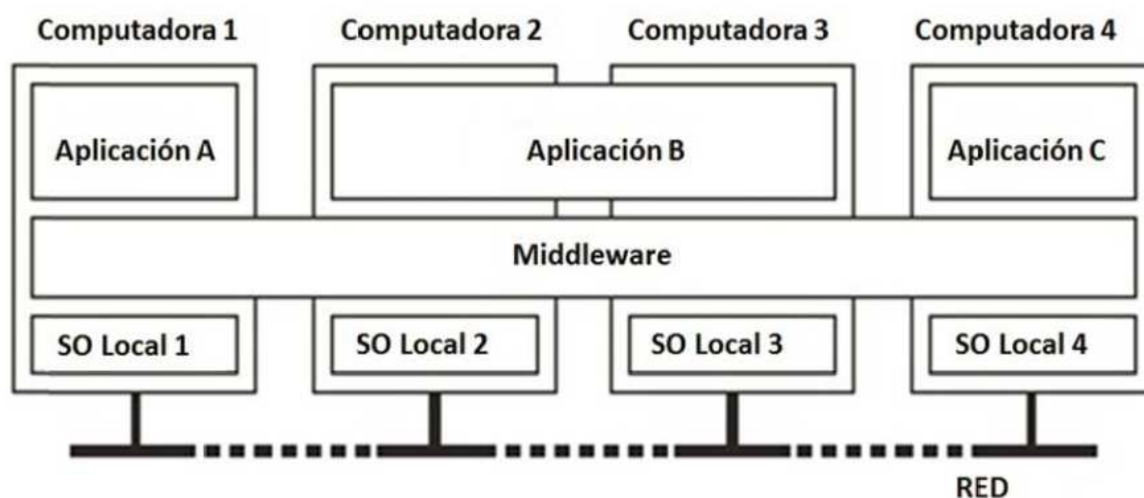

Fig. 1.1 Componentes de un Sistema Distribuido

En la figura 1.1 se pueden observar algunos de los componentes que se encuentran en un sistema distribuido, siendo la capa middleware la que oculta los detalles del entorno de ejecución local de cada computadora, permitiendo la visión de un sistema único.

Entre las motivaciones para construir sistemas distribuidos, sin lugar a dudas está el hecho de compartir recursos: desde un elemento de hardware (impresoras, unidades de disco, memorias, etc.), hasta entidades de software (archivos, bases de datos, servicios, objetos, elementos multimedias, etc.). Pero también favorece el diseño de software modular donde cada componente se ejecuta en la plataforma más adecuada y brinda un servicio más eficiente debido a la especificidad de su construcción. Por ejemplo, concebir un sistema de software donde uno de los componentes es la gestión y manipulación de los objetos de información (datos), le da un papel importante a los Sistemas de Gestión de Bases de Datos - en inglés Database Management Systems o DBMS - que logran dar un servicio de acceso a los datos eficiente, correcto y consistente. [19]

1.3 Aplicaciones Enterprise

Las Aplicaciones Enterprise representan una importante herramienta de soporte para las organizaciones. Son las encargadas de relacionar la información, los recursos humanos, la tecnología, los procesos de negocio y los elementos de la infraestructura de la empresa, para cumplir con los objetivos de la organización.

Una Aplicación Enterprise modela el funcionamiento del proceso de la empresa. Su principal objetivo es la automatización de los procesos de negocio de la organización.

Por lo tanto, la mayor prioridad de estas aplicaciones es apegarse estrictamente a la lógica de negocio de la empresa, es decir, de acuerdo a cómo funcione la empresa deberán establecerse las reglas de cómo manejar sus datos: quién puede ver, quién modificar, cuándo se permite borrar algún registro o crear alguno nuevo, qué dependencias existen entre los distintos tipos de datos, etc.

Otra característica importante, es que generalmente las empresas de tamaño mediano y grande están organizadas por departamentos o áreas, y con frecuencia crear una plataforma gigantesca que resuelva las necesidades de todas las distintas áreas no es lo más óptimo, sino que resulta preferible desarrollar distintas aplicaciones o plataformas que compartirán cierta información y recursos entre sí. Esto implica que las diferentes aplicaciones que se desarrollen para la empresa deben convivir armónicamente y complementarse entre sí, formando lo que en informática se conoce como *ecosistema*.

El *ecosistema de software* de una empresa u organización —que tiene como propósito último el ayudarle a realizar sus actividades de forma más efectiva, así como lograr cosas que sin las herramientas informáticas adecuadas resultarían imposibles— muchas veces está a cargo de un equipo de desarrollo dentro de la propia empresa, pero cada vez es más común que se contraten proveedores especializados para desarrollar estas herramientas.

Cuando son varios los proveedores desarrollando aplicaciones para la misma empresa, el trabajo de integración es otro aspecto muy importante, para garantizar siempre la integridad de los datos y el apego a la ya mencionada lógica de negocio. También es importante manejar convenciones generales, que ayuden a los usuarios a que el conocimiento adquirido en cualquiera de las aplicaciones sea transferible, en la medida de lo posible, a otras aplicaciones del ecosistema del software empresarial.

A su vez, no podemos dejar afuera la importancia de crear un buen flujo de trabajo con el software o aplicaciones que no fueron desarrollados específicamente para la empresa, y que sin embargo también ayudan a satisfacer las necesidades de la misma, como pueden ser Word, Excel, Dropbox, Trello y G Suite, entre muchos otros.

El diseño de estas aplicaciones suele ser una tarea muy compleja. Los diseñadores tienen que prestar especial atención a varios aspectos que intervienen

en el producto a desarrollar, dado que no solo deberán atender los aspectos funcionales sino que tendrán que trabajar en cuestiones de eficiencia, escalabilidad, performance, seguridad, entre otras. Por otro lado, los arquitectos de software suelen enfrentar problemas que ocurren una y otra vez en distintos proyectos de software; como dividir a la aplicación en capas, manejar la concurrencia de acceso a recursos o vincular la aplicación con la estructura de datos. En estos casos se recurre a soluciones que resultaron exitosas en otros casos similares.

1.4 Objetivo

La presente Tesina tiene los siguientes objetivos:

- Realizar un estudio de caso del modelo de Arquitectura N-Capas aplicado en el Sistema de la empresa.
- Describir el esquema de la arquitectura, con una reseña sobre el significado de los componentes y capas presentados en dicho esquema.
- Analizar los detalles y customizaciones que se llevaron a cabo según las necesidades y las tecnologías utilizadas, justificando las decisiones tomadas.
- Comparar el esquema utilizado contra el esquema tradicional N-Capas.

1.5 Estructura de la Tesina

Lo que resta del documento se organiza de la siguiente manera:

- **Capítulo 2:** se describe brevemente la evolución la arquitectura en capas, definiendo de manera sintética algunas de las variantes más conocidas.
- **Capítulo 3:** en este capítulo se detallan algunos de los patrones de diseño de arquitectura más utilizados.
- **Capítulo 4:** se analizan los principales principios arquitectónicos que se deben respetar para lograr diseños de calidad.
- **Capítulo 5:** aquí se desarrolla el estudio de caso del modelo N-Capas aplicado al sistema de la empresa, analizando los detalles y customizaciones que se llevaron a cabo y justificando las decisiones tomadas.
- **Capítulo 6:** se presentan conclusiones sobre el trabajo expuesto, dando lugar a líneas de trabajo futuro.

Capítulo II

Evolución de la arquitectura en capas

Las aplicaciones o sistemas distribuidos concebidos como aquellos que funcionan como si se tratara de un sistema único, no hacen ninguna presuposición acerca de la variedad funcional de sus componentes ni de cómo estos componentes se comunican. De esta manera, las variantes que existen han dado origen a sistemas distribuidos de distinta índole y que van de la no-distribución (sistemas monolíticos) hacia la distribución completa (servicios orquestados por procesos de negocio).

A continuación se analizará la evolución de los sistemas y sus arquitecturas, que sin lugar a dudas ha ido de la mano, a su vez, de la evolución de las comunicaciones, plataformas y hardware.

2.1 Sistemas Monolíticos

Un sistema de información monolítico es aquel que se concibe como un único elemento funcional donde sus prestaciones se ofrecen a través de una sola pieza de código que resuelve tanto la presentación al usuario (interfaz), como el acceso a los datos (trata con operaciones de entrada/salida), y que a su vez resuelve la lógica algorítmica del problema que aborda.

Estos sistemas de información se construían en un único lenguaje de programación, sobre un único computador y con un único sistema operativo como plataforma. La comunicación con el usuario y también con el programador, era a través de terminales de caracteres que responden únicamente a comandos lanzados por consola. Su ubicación en el tiempo se remonta a los años ‘70.

Una versión mejorada de los sistemas de información así concebidos, pudo aportarse con las técnicas de programación estructurada, los lenguajes y metodologías de programación modular introducidas por Edsger Dijkstra. En este sentido, los sistemas de información comenzaron a ser concebidos como un conjunto de componentes o niveles de complejidad. Aunque permanecían ejecutándose en una misma computadora y, por lo tanto, no podemos considerarlos como distribuidos.

En la figura 2.1, la evolución marca que el primer paso fue reconocer un componente funcional que resuelve el acceso a los datos -con la aparición de los

DBMS (Data Base Management Systems)-, para luego identificar un componente de manejo de interfaz de usuario. [19] [21]

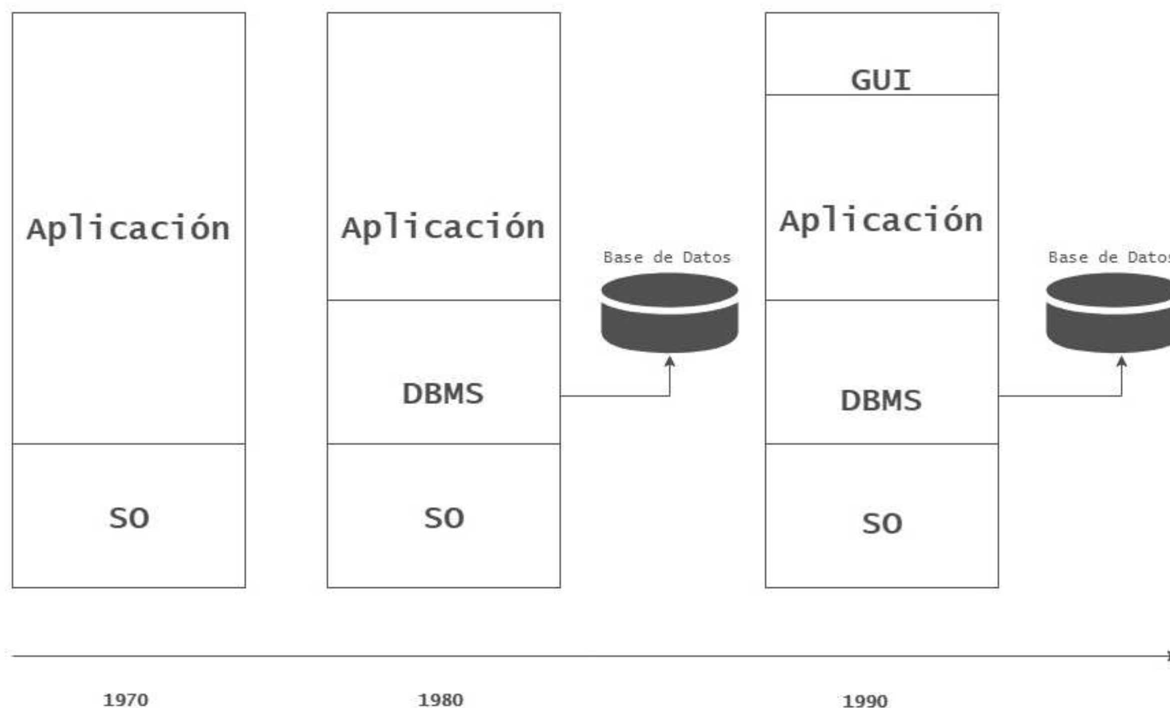

Fig. 2.1 Evolución tecnológica de los Sistemas de Información.

2.2 Cliente/Servidor y su evolución a N-Capas

La distribución funcional entre lógica de aplicación propiamente dicha y lógica de acceso a datos vislumbrada en la década del 80 condujo a concebir la idea de que estos dos componentes podrían residir en computadoras diferentes, aprovechando las consultas SQL que podían interpretar los DBMS, junto con la gran proliferación de PCs sobre LAN (Local Area Network), que se estaba dando en esos tiempos.

Este escenario dio origen a la primera versión de sistema distribuido, desde el aspecto funcional, que fue el denominado Cliente/Servidor o arquitectura de 2 capas.

La arquitectura Cliente/Servidor clásica, se sustenta en la idea de que existe un componente Cliente que resuelve la lógica de la interfaz con el usuario. El código de este componente se escribe generalmente en un lenguaje visual de cuarta generación (Delphi, VisualBasic, PowerBuilder). La segunda capa sería un componente Servidor que resuelve la lógica de acceso a los datos, al menos con la gestión que hace con los mismos un DBMS.

En esta arquitectura, la lógica de la aplicación no cuenta con un componente físico específico para su ejecución y despliegue, sino que es absorbida por alguno

de los dos componentes antes mencionados -Cliente o Servidor-, tal como se muestra en la figura 2.2.

Fig. 2.2 Sistema Distribuido en 2 capas o Cliente/Servidor clásico.

Este modelo de sistema distribuido posee las siguientes ventajas:

- Aprovechamiento del poder de las PCs.
- Permite que el procesamiento resida cerca de la fuente de datos, reduciendo el tráfico en la red.
- Facilita el uso de GUI (Graphic User Interface).
- Acepta el desafío de los sistemas abiertos.
- Favorece el diseño modular.

Por otra parte, posee ciertas desventajas:

- Un desbalance de tareas entre Cliente y Servidor puede provocar cuellos de botella.
- El desarrollo de aplicaciones es más complejo, creciendo también la complejidad de las herramientas de desarrollo y programación.

La desventaja fundamental que llevó a buscar evolucionar hacia un modelo superador lo constituyó sin dudas el desbalanceo de carga entre Cliente y Servidor, lo cual se vio acompañado por la instauración cada vez más firme de la Internet como la mejor solución a la no disponibilidad de recursos de cómputo y la globalización de los sistemas de información.

Por su parte, el modelo Cliente/Servidor clásico se transformó en poco escalable tanto a nivel vertical (aumento de cantidad de clientes y, por ende, de

requerimientos al servidor), como horizontal (un solo servidor accesible a través de una LAN ya no era suficiente; se requería acceso a través de WAN -Wide Area Network).

Surge así el modelo de 3 capas, donde básicamente se le otorga a la capa media (lógica de la aplicación) una plataforma de ejecución propia. Esta capa media puede ser una sola capa (arquitectura de 3 capas) o puede subdividirse en diferentes capas, cada una de las cuales es cliente de la anterior y servidor de la siguiente, obteniéndose así una arquitectura de N-Capas, como se muestra en la figura 2.3. [19]

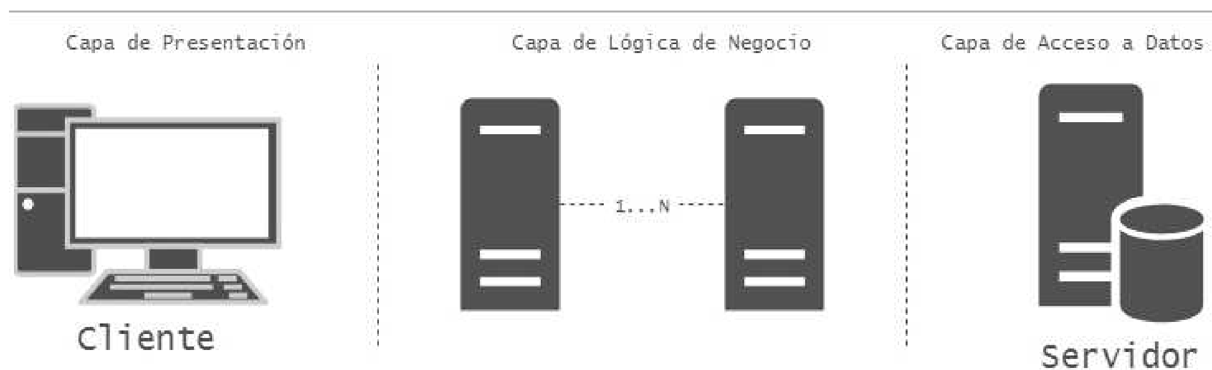

Fig. 2.3 Sistema Distribuido en N-Capas.

2.2.1 Arquitectura Web: un caso particular del Sistema Cliente/Servidor

La arquitectura Web es aquella que hace uso de Internet para construir sistemas de información. En este sentido, dichos sistemas se construyen haciendo uso de los protocolos y mecanismos de comunicación provistos por Internet. Esto es: acceder a recursos a través de una URL (Uniform Resource Locator), obtener dichos recursos con un protocolo de transferencia de archivos (FTP/HTTP) y mostrar los resultados al usuario a través de un Navegador Web.

La figura 2.4 muestra el esquema de la arquitectura Web simple, en el cual un Navegador Web (Web Browser) interpreta páginas HTML obtenidas por HTTP desde un Servidor Web (Web Server). [19]

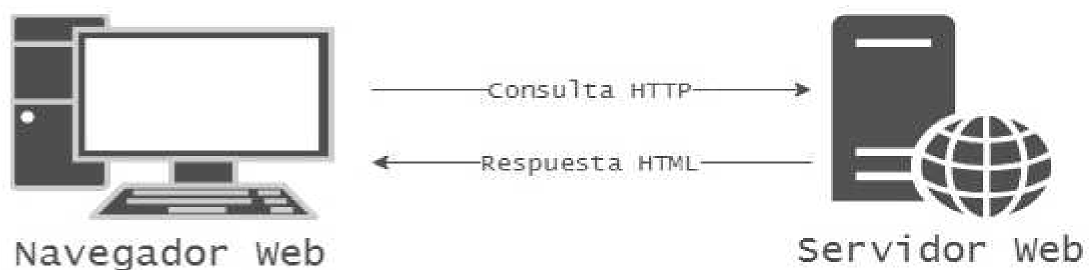

Fig. 2.4 Arquitectura Web simple.

2.2.2 Arquitectura Web en 3 capas: Web Server dinámico

El componente Web Server (Servidor Web) en sistemas distribuidos de tecnología Web, provee un tipo de servicio que se limita a recibir peticiones por HTTP y responder con un conjunto de registros (archivos) estructurados (o estáticos) como HTML.

Este escenario resulta insuficiente para construir sistemas de información debido a: 1) la ausencia de un componente que permita acceder a los datos y mantenerlos persistentes; y 2) ausencia de un componente que ejecute código para producir transformaciones sobre esos datos.

Surge así lo que se denomina Web Server dinámico, es decir, un servidor capaz de encadenar cómputo que se encuentra codificado dentro de las páginas HTML que administra. Esta arquitectura se ve reflejada en la figura 2.5. [19]

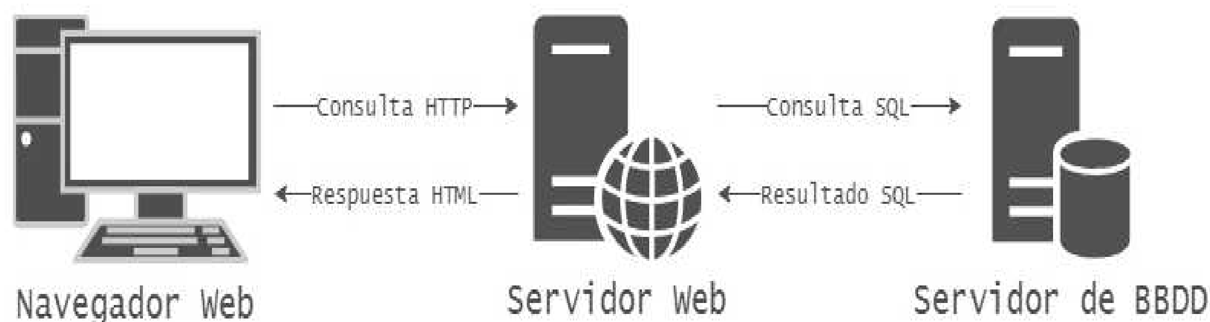

Fig. 2.5 Arquitectura Web en 3 capas.

2.3. Sistemas Distribuidos con Objetos

A medida que fue aumentando la complejidad de las soluciones distribuidas fue necesario incorporar conceptos, metodologías y buenas prácticas en la construcción de software y llevarlas al entorno distribuido.

El concepto de objeto en el sentido clásico de la Programación Orientada a Objetos y que tiene sus raíces principales en el lenguaje Smalltalk, se mantiene a nivel de diseño de aplicaciones pero cambia algunas características cuando se lo lleva a un entorno distribuido.

Un *objeto distribuido* es un objeto (componente funcional con estado interno, comportamiento, heredable y encapsulado), que por estar en un entorno distribuido también debe ser:

- **Transaccional:** los cambios de estado que produce se deben ejecutar completamente o no ejecutarse.
- **Seguro:** debe evitar vulnerabilidades que corrompan el estado del sistema.
- **Lockable:** debe fijar un cerramiento que garantice su ejecución correcta ante accesos concurrentes.
- **Persistente:** su estado interno debe conservarse más allá de su ciclo de vida.

Esta definición de objeto distribuido nos conduce a la necesidad de contar con una infraestructura que facilite la comunicación entre este tipo de componentes, logrando a su vez que alcancen esta comunicación a través de distintas plataformas de ejecución, sistemas operativos y lenguajes de programación.

Así surge el estándar *CORBA* (Common Object Request Broker Architecture), definido por la *OMG* (Object Management Group), y que surge de la iniciativa de un consorcio de empresas interesadas en establecer una arquitectura interoperable sobre la base del paradigma orientado a objetos. CORBA se constituye así en un ejemplo de sistema distribuido basado en objetos.

Entre los elementos distintivos del estándar -y que se puede decir que sentaron las bases para lo que se conoce actualmente como web services- encontramos:

- Un lenguaje de especificación de interfaz, IDL (Interface Definition Language), que define los límites de las componentes, o sea las interfaces contractuales con los potenciales clientes.
- Un *bus* de objetos u ORB (Object Request Broker), que permite comunicar objetos independientes de su locación. El cliente se despreocupa de los mecanismos de comunicación con los que se activan o almacenan los objetos servidores.

Provee un vasto conjunto de servicios de middleware distribuido y tiene claramente un mecanismo de comunicación mucho más complicado que los clásicos RPC (Remote Procedure Call) y MOM (Message-Oriented Middleware).

2.4. Integración de Aplicaciones

Más allá de los esfuerzos por construir estándares para mejorar la interoperabilidad y de la existencia de plataformas Web mucho más accesibles al programador y al usuario, el desarrollo de aplicaciones distribuidas a gran escala, seguía adoleciendo de problemas de integración.

La idea de construcción de aplicaciones integradas permitió que las organizaciones desarrollen software que resuelva cada parte de su negocio y se integre con aplicaciones que gestionen la parte administrativa de dicho negocio. Así

la idea de integración de aplicaciones comienza a cobrar un sentido muy relevante.

En este contexto surgen los sistemas ERP (Enterprise Resource Planning) que proveen variada funcionalidad integrada por un único repositorio de datos.

No se tardó demasiado en intentar agregar valor a los desarrollos de las organizaciones aportando sistemas de CRM (Customer Relationship Management) o sistemas de Data Warehouse, que requieren algún tipo de integración con los sistemas de índole operativa o propia del negocio.

Esta técnica se enfoca principalmente en la integración vía el modelo de datos y ha sufrido una evolución que incluye diversas variantes, las cuales se describen a continuación. [19]

2.4.1 Integración Punto a Punto

Se basa en integración uno a uno sustentada generalmente por un middleware asincrónico basado en colas de mensajes. Si bien es un esquema de alta disponibilidad -dada por el mecanismo de comunicación-, es rígido y difícil de adaptar a los cambios, además de resultar muy costoso de gestionar, monitorear y extender. Es una arquitectura accidental, completamente sincrónica, de grano grueso y poco escalable.

La Figura 2.6, presentada por M.Weske en (Bazán,2010), muestra el escenario de una integración con el mecanismo punto a punto.

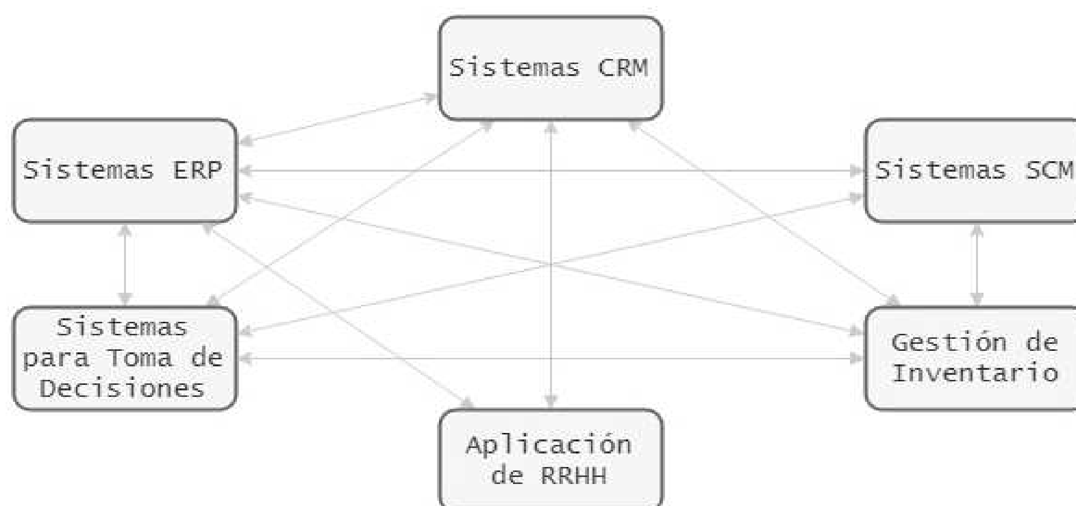

Fig. 2.6 Mecanismo de integración punto a punto.

2.4.2 Integración por adaptadores

La interacción se realiza por un *hub* donde se conecta cada elemento a integrar previa construcción del adaptador correspondiente para poder "enchufarse". Cada uno de los elementos está desconectado y no requiere utilizar el mismo

lenguaje, ya que existen adaptadores para establecer la comunicación. La complejidad se encuentra en la construcción del adaptador.

La figura 2.7 presenta una evolución de la figura 2.6 hacia un mecanismo de integración por adaptadores.

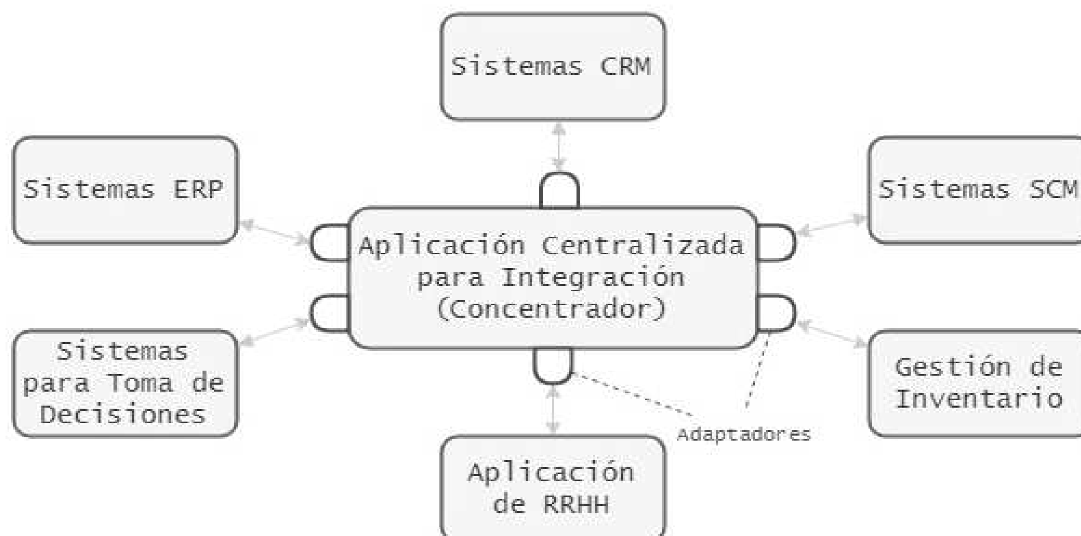

Fig. 2.7 Mecanismo de integración por adaptadores.

2.4.3 Mediador de Mensajes

Este mecanismo de integración representa una evolución del modelo anterior hacia una generalización del hub, permitiendo extraer la lógica de la integración fuera de las aplicaciones.

Se define declarativamente la forma de comunicación de las aplicaciones y se traslada al mediador la lógica necesaria para producir las transformaciones que generen salidas válidas para el otro extremo.

Utiliza colas para garantizar la distribución de los mensajes, que se manejan con un esquema de *publicador/suscriptor*. Esto asegura que el tráfico que se genera sea solamente el requerido.

La figura 2.8 muestra el próximo paso evolutivo desde la figura 2.6, donde se observa la integración a través de un mediador de mensajes.

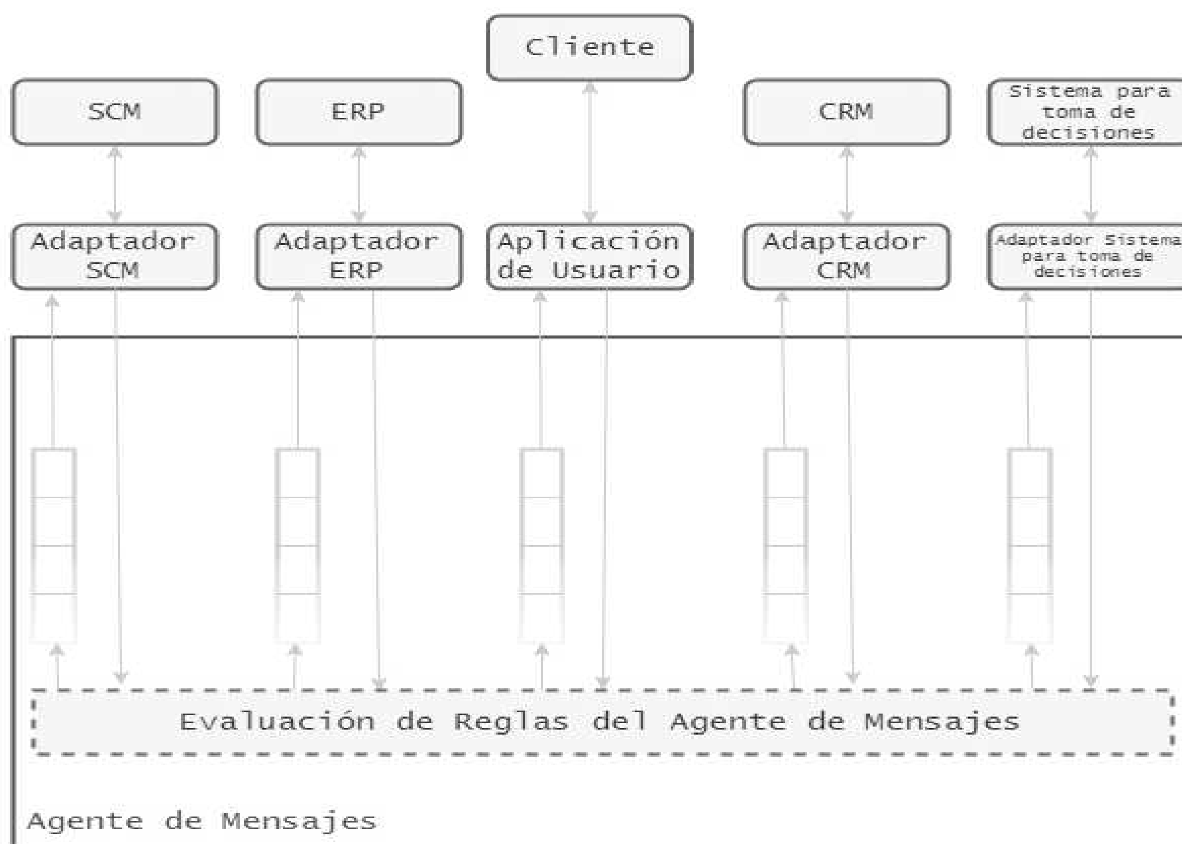

Fig. 2.8 Mecanismo de integración por mediador de mensajes.

2.5 Arquitectura Orientada a Servicios

Si bien la integración de aplicaciones como se ha planteado en 2.4 ha sido mejorada con diversos mecanismos como el uso de adaptadores y mediadores, sigue siendo una arquitectura accidental (se construye para cada caso), de mantenimiento costoso y lento, y difícil de gestionar, monitorizar y extender.

La Arquitectura Orientada a Servicios (SOA -Service Oriented Architecture-, por sus siglas en inglés) representa un cambio radical en la relación entre el mundo del negocio y el área de tecnología de la información. SOA constituye mucho más que un conjunto de productos aglutinados por una tecnología. Es un nuevo enfoque en la construcción de sistemas de IT que permite a las empresas aprovechar los activos existentes y abordar fácilmente los inevitables cambios en el negocio.

Si bien la industria del software ha venido enfocándose en una arquitectura orientada a servicios desde hace más de 20 años con la noción de reusabilidad y su aplicación a la construcción de software, lo cierto es que en los últimos años esto se ha fortalecido con la definición de estándares y la conformación de consorcios que participan en su definición.

Según el IEEE (Institute of Electrical and Electronics Engineers), una arquitectura de software es la organización fundamental de un sistema, reflejado por sus componentes, relaciones entre ellos y entorno, así como los principios que regirán su diseño y evolución.

Según OASIS (Organization for the Advancement of Structured Information Standards), se define como la estructura o estructuras de un sistema de información formado por entidades y sus propiedades externamente visibles, así como las relaciones entre ellas.

Adaptándose a la definición de OASIS, se define SOA como un paradigma capaz de organizar y utilizar las capacidades distribuidas, que pueden estar bajo el control de distintas organizaciones, y de proveer un medio uniforme para publicar, descubrir, interactuar y usar los mecanismos oportunos para lograr los efectos deseados. [19]

Podemos resumir los conceptos subyacentes fundamentales de este paradigma en los siguientes:

- **Proveedor:** entidad (organización o persona) que ofrece el uso de capacidades mediante servicios.
- **Necesidad:** carencia de una empresa para resolver la actividad de su negocio.
- **Consumidor:** entidad (organización o persona) que busca satisfacer una necesidad particular a través de las capacidades ofrecidas por servicios.
- **Capacidad:** tarea que el proveedor de un servicio puede proporcionar al consumidor.
- **Servicio:** mecanismo que permite el acceso a una o más capacidades alcanzables por medio de una interfaz preestablecida, y que se llevará a cabo de forma consistente con las normas establecidas para él.
- **Descripción del servicio:** información necesaria para hacer uso del servicio.
- **Interacción:** actividad necesaria para hacer uso de una capacidad con el objeto de obtener efectos deseados.

2.5.1 Procesos de Negocio como Consumidores de Servicios

En términos generales, cuando se habla de integración de aplicaciones nos referimos tanto a datos como a procesos.

Los datos pueden integrarse con cualquiera de los esquemas anteriormente planteados. Cualquiera de ellos implica el desencadenamiento de un flujo o cadena de mensajes que quedan incrustados dentro del esquema de integración mismo.

Así, los sistemas de gestión de workflow surgen como ciudadanos de primera clase a la hora de realizar una integración de aplicaciones eficiente y fácil de mantener y actualizar.

Gran parte del esfuerzo de implementar una arquitectura orientada a servicios orquestada por procesos de negocio se ha centrado sobre la integración de aplicaciones y su workflow. Sin embargo, existe una preocupación acerca de la incapacidad para acceder a datos de negocio y administrarlos de una manera ágil, tal como sucede con la lógica de negocio en las aplicaciones.

La integración de datos dirigida por procesos puede ayudar a enriquecer los servicios de negocios SOA y los procesos de negocio a través de una secuencia de servicios de datos combinados de manera reusable, que incorpora la intervención de tareas humanas, transformando la información en exacta, consistente y oportuna. [19]

Capítulo III

Patrones de Arquitectura

Un patrón codifica conocimiento específico acumulado por la experiencia en un dominio. Es una solución a un problema en un contexto.

Citando a Christopher Alexander [22]:

"Cada patrón describe un problema que ocurre una y otra vez en nuestro ambiente, y luego describe el núcleo de la solución a ese problema, de tal manera que puedes usar esa solución un millón de veces más, sin hacer jamás la misma cosa dos veces."

Alexander era arquitecto. Como ejemplo de patrones proponía: galería, paseo, patio compartido, columnata, estacionamiento.

En el mundo del software, por su parte, los patrones también existen, y podemos diferenciarlos o clasificarlos en distintos niveles:

- **Patrones de Arquitectura:** se centran en la estructura del sistema. En la definición de subsistemas, sus responsabilidades y reglas sobre las relaciones entre ellos. Proporcionan un conjunto predefinido de subsistemas, sus responsabilidades y los lineamientos para organizar sus relaciones.
- **Patrones de Diseño:** esquemas para refinar los subsistemas o componentes de sistema software o sus relaciones. Describen una estructura recurrente y común que resuelve un problema de diseño dentro de un contexto.
- **Patrones de Análisis:** usualmente específicos de aplicación o industria. Resuelven cuestiones como modelado del dominio, completitud, integración y equilibrio de objetivos múltiples, planeamiento para capacidades adicionales comunes, etc.
- **Patrones de Proceso o de Organización:** procesos de administración de proyectos; técnicas o estructuras de organización. Utilizados para la planificación de proyectos a gran escala, gestión, etc.
- **Patrones de Codificación o Idiomas:** patrones que ayudan a implementar aspectos particulares del diseño en un lenguaje de programación específico.

Serán los patrones arquitecturales, combinado con los estilos arquitectónicos y la experiencia en otros proyectos los que nos permitan reusar soluciones que facilitarán el armado de la arquitectura de nuestro sistema.

Existen varias categorizaciones de patrones de arquitectura. Una de las más difundidas y utilizadas es la que brinda Martin Fowler en su libro Patterns of Enterprise Applications [23]. A continuación se listan las categorías que propone y los patrones más importantes de cada una.

3.1 Patrones de Arquitectura de Aplicaciones de tipo Enterprise

Patrones que resuelven distintos problemas asociados a las capas de una Aplicación de tipo Enterprise. Expresan un esquema de organización estructural esencial para la aplicación, que consta de subsistemas, sus responsabilidades e interrelaciones. En comparación con los patrones de diseño, los patrones arquitectónicos tienen un nivel de abstracción mayor. Podemos encontrar una buena fuente de consulta de estos patrones en el catálogo de Martin Fowler: http://www.martinfowler.com/eaaCatalog/

3.2 Patrones estructurales de mapeo Objeto-Relacional

El rol que posee la capa de acceso a datos de una arquitectura es de comunicar los objetos de dominio con la base de datos. Siendo que la mayoría de las bases de datos son relacionales, es necesario contar con patrones que nos faciliten esta traducción.

La separación que provee una capa de acceso a datos permite trabajar de manera aislada con la lógica de dominio dejando afuera el diálogo que tiene el dominio con la base de datos. Si bien todo desarrollador debe conocer cómo es esta interacción, esta separación permite que expertos en bases de datos puedan optimizar el acceso y forma de interpretar los datos (SQL generados) para lograr una mayor performance. De manera análoga, los diseñadores que modelen la capa de dominio podrán diseñar representado la realidad sin tener que preocuparse cómo y quién maneja de la persistencia.

3.2.1 Identity Field

Guarda un campo *ID* de la base de datos en un objeto para mantener la relación unívoca que existe entre un objetos y su representación de una fila en una tabla de una base de datos.

Las bases de datos relacionales, distinguen una fila de otra utilizando una identidad llamada *primary key*. Los objetos en memoria quedan identificados mediante un atributo de identidad (*Identity Field)* que se corresponde con la *primary key* que posee la tupla en la base de datos.

3.2.2 Foreign Key Mapping

Mantiene una relación entre objetos cuando existe una relación de *foreign key* en la base de datos.

Los objetos en memoria se referencian con otros directamente por sus referencias en memoria. Para persistir estas relaciones en la base de datos, es necesario guardar estas referencias. *Foreign Key Mapping* representa esta referencia a un objeto, que es a su vez, una *foreign key* en la base de datos.

3.3 Patrones de modelado de la complejidad del Dominio

Al momento de interactuar con la lógica de dominio nos encontramos con problemas de acceso, coordinación y responsabilidades de comportamiento.

Nos referimos a los elementos de software que atenderán los pedidos solicitados por algún cliente para la ejecución de la lógica de dominio, la coordinación de las actividades que se disparan con la solicitud y los responsables de ejecutar las reglas del negocio y lógica de dominio.

Los siguientes patrones presentan alternativas para simplificar lo antes expuesto.

3.3.1 Transaction Script

Es la forma más simple de ordenar la lógica de dominio. Organiza la lógica de negocio a través de procedimientos donde cada uno maneja un simple *request* (pedido) desde la capa de presentación.

La mayoría de la lógica que contienen las aplicaciones puede pensarse como una serie de transacciones. Una transacción puede ver cierta información organizada de una manera particular, otra puede hacer cambios sobre ésta. Cada interacción entre un cliente y un servidor contiene cierta lógica, en algunos casos puede ser simple como visualizar información y en otros puede incorporar una cierta cantidad de pasos de validaciones y cálculos.

Un *Transaction Script* organiza toda esa lógica como un solo procedimiento, conectándose directamente a la base de datos, o a través de un delgado *wrapper* de ésta. Cada transacción tiene su propio *Transaction Script*, aunque algunas tareas comunes pueden volcarse en subprocedimientos.

3.3.2 Domain Model

Un modelo de objetos del dominio que incorpora comportamiento y datos.

Cuando la lógica de negocio es muy compleja, y depende de muchos factores, los objetos son los indicados para modelarla. Un Modelo de Dominio crea una red de objetos interconectados, donde cada objeto representa algo significativo e individual.

3.4 Patrones arquitecturales de acceso a datos

3.4.1 Table Data Gateway

Un objeto que actúa como *Gateway* a la base de datos. Una instancia que maneja todas las filas en una tabla.

Mezclar código SQL en la lógica de la aplicación puede causar numerosos problemas. Los desarrolladores no suelen codificar SQL como lo harían los administradores de una base de datos. Por esta razón es que es necesario centralizar en pocos lugares la utilización de este código.

Table Data Gateway contiene todo el SQL para acceder a una sola tabla o vista. Los demás elementos de la aplicación invocan sus métodos para acceder a los datos.

3.4.2 Active Record

Un objeto que hace de *wrapper* a una fila de una tabla o vista de la base de datos encapsulando el acceso a los datos y agregando la lógica del dominio a sus datos.

Un objeto que posee tanto datos como comportamiento y muchos de sus datos son persistentes y necesitan ser almacenados en una base de datos.

Active Record pone la lógica de acceso a datos en el objeto de dominio, permitiendo que cada objeto de dominio sepa cómo cargarse y guardarse desde y hacia la base de datos.

3.4.3 Data Mapper

Una capa de *Mappers* mueve los datos entre objetos y una base de datos mientras mantiene la independencia entre ambas.

La estructura de los objetos en memoria y los datos persistidos en bases de datos relacionales es bastante diferente. Las partes de los objetos que representan colecciones y herencia no están presentes en las bases relacionales. La representación de la realidad del modelado de comportamiento del modelo de objetos no siempre encaja con el modelado de la base. Si se lleva el conocimiento de la estructura de la base de datos a los objetos, perderíamos independencia ya que un cambio en la estructura de persistencia impactaría directamente en el modelado de objetos.

Data Mapper es una capa de software que separa los objetos en memoria de su representación en la base de datos. Su responsabilidad es transferir los datos entre los dos y aislarlos a uno del otro. Con *Data Mapper* los objetos en memoria no necesitan saber que existe una base de datos, ni saber de código SQL, ni como son persistidos.

3.5 Patrones de comportamiento objeto-relacional

3.5.1 Identity Map

Se asegura que cada objeto sea cargado solo una vez, manteniendo cada objeto cargado en un mapa. Cuando se requiere un objeto, busca en en el mapa si ya lo tiene cargado.

La recuperación de objetos desde la base de datos a memoria tiene su trasfondo peligroso. Si no se procura mantener una identificación correcta de los objetos que se tienen en memoria se podría caer en el problema de hacer una carga de un mismo registro de la base de datos en dos objetos distintos en memoria. Asociado a esto, aparecen otros problemas no menores, como problemas de performance, ya que se pueden hacer lecturas innecesarias a la base de datos.

Un *Identity Map* mantiene un registro de todos los objetos que fueron cargados de la base de datos dentro de una transacción de negocio. Cuando se desea un objeto, se verifica que no esté en *Identity Map* para ver si ya fue cargado.

3.5.2 Lazy Load

Un objeto que no contiene los datos solicitados, pero sabe cómo obtenerlos.

Lazy Load actúa en el proceso de la búsqueda de objetos relacionados a otro objeto. Permite la recuperación de relaciones de objetos, a demanda, y de manera “perezosa” haciendo que la recuperación sea efectiva luego de la verificación de la solicitud de recuperación. De no ser necesaria la búsqueda de estos datos, se consigue un ahorro de tiempos.

3.6 Patrones de base

3.6.1 Layer Supertype

Una clase que es la base de todas las clases en su capa. Todo el comportamiento común a los objetos de dominio se detallan en una clase de dominio padre de todas, que generaliza el comportamiento.

3.7 Patrones de concurrencia

3.7.1 Optimistic Offline Lock

Previene conflictos de concurrencia en transacciones de negocio, detectando un conflicto y realizando un rollback de la transacción.

3.8 Patrones metadatos en mapeo objeto-relacional

3.8.1 Metadata Mapping

Mantiene información del mapeo objeto-relacional como *meta-información*. Para evitar duplicar la información de la correspondencia de los campos de las tablas de la base datos con los atributos de los objetos de dominio, se utiliza un *mapeador* de esta información para que pueda ser interpretada y mantenida de una manera más sencilla.

3.8.2 Query Object

Es un objeto que representa una consulta a la base de datos. Es decir, es una estructura de objetos, pero que se puede convertir en una consulta *SQL*. Se puede crear esta consulta haciendo referencia a las clases y sus campos, en lugar de tablas y columnas. De esa manera quienes escriben las consultas pueden hacerlo independientemente del esquema de la base de datos y los cambios al esquema pueden estar localizados en un solo lugar.

3.8.3 Repository

Es un mediador entre los objetos de dominios y la capa de mapeo de datos.

En sistemas con un modelo de dominio complejo se utilizan capas como las que proveen los *Data Mappers,* que aíslan a los objetos de dominio de los detalles de la base de datos. En este tipo de sistemas es beneficioso tener otra capa de acceso a datos por sobre la capa de mapeos en donde se concentre la construcción de consultas. Esto resulta casi necesario en dominios en donde existe un gran número de objetos de dominio o se utilizan muchas consultas.

Un repositorio actúa de mediador entre los objetos de dominio y la capa de mapeo, actuando como si el dominio fuera una gran colección de objetos que están en memoria.

3.9 Patrones de presentación Web

3.9.1 Model View Controller

Separa la interacción de la interfaz de usuario en tres roles distintos:

- **Model:** es la representación de la información con la cual el sistema opera, por lo tanto gestiona todos los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación. Envía a la “vista” (*View*) aquella parte de la información que en cada momento se le solicita para que sea mostrada (típicamente a un usuario). Las peticiones de acceso o manipulación de información llegan al *Model* a través del *Controller*.

- **Controller**: responde a eventos (usualmente acciones del usuario) e invoca peticiones al *Model* cuando se hace alguna solicitud sobre la información (por ejemplo, editar un documento). También puede enviar comandos a su *View* asociada si se solicita un cambio en la forma en que se presenta el *Model* (por ejemplo, desplazamiento o scroll por un documento o por los diferentes registros), por tanto se podría decir que el *Controller* hace de intermediario entre el *View* y el *Model* .
- **View**: presenta el *Model* (información y lógica de negocio) en un formato adecuado para interactuar (usualmente la interfaz de usuario -*UI*-), por tanto requiere de dicho *Model* la información que debe representar como salida.

3.9.2 Page Controller

Un objeto que maneja el *request* de una acción específica o una página en un sitio web.

Sus responsabilidades básicas son:

- Decodificar la URL y extraer los datos del formulario para resolver lo que la acción requiere.
- Crear e invocar a los objetos del *Modelo* para procesar la información. Toda la información relevante del *request HTTP* debe ser enviada al modelo para que éste no necesite conexión directa con el HTML.
- Determinar cuál *Vista* debe mostrar el resultado y enviar la información del *Modelo* a dicha *Vista*.

3.9.3 Front Controller

Es un controlador que maneja a todos los *requests* de un sitio web. En un sitio web complejo existe mucho comportamiento similar que posiblemente se necesite centralizar al momento de procesar un request. Estas cosas incluyen cuestiones como seguridad, internacionalización y vistas.

3.9.4 Application Controller

Centralización del manejo de la navegación y flujo de la aplicación. Tiene dos responsabilidades esenciales: decidir qué lógica de dominio ejecutar y decidir cuál vista mostrará el resultado.

3.10 Patrones de distribución

3.10.1 Remote Facade

Provee una fachada de grano grueso para interactuar con un conjunto de objetos de interfaz de grano fino para mejorar la eficiencia en el uso de red.

En un modelo orientado a objetos, en donde los objetos poseen una interfaz pequeña para un gran número de objetos, se logra control del comportamiento, un mayor entendimiento de la lógica y abstracción.

Una de las consecuencias de este modelo es la gran interacción que existe entre el modelo y por lo tanto numerosas invocaciones, que cuando son remotas provocan una gran utilización de red.

Es necesario entonces generar una interfaz que minimice el número de llamadas entre objetos para realizar una actividad.

3.10.2 Data Transfer Object

Un objeto que lleva información entre procesos con el objetivo de reducir el número de las invocaciones a métodos.

Cuando se trabaja en esquemas de interfaces remotas, cada llamada es cara. Como resultado de esto es necesario reducir el número de llamadas remotas, tratando de aumentar la cantidad de información que se acarrea en cada solicitud remota.

Una forma de lograr esto es utilizar numerosos parámetros, pero esto no siempre es válido en lenguajes de programación que retornan un solo valor.

La solución a esto es crear un objeto de transferencia (DTO, Data Transfer Object, por sus siglas en inglés) que pueda contener toda la información necesaria para la acción específica que disparó la llamada remota.

3.11 Patrones de manejo de sesión

3.11.1 Client Session State

Guarda el estado de la sesión del cliente.

Incluso los diseños más *server-oriented* necesitan al menos un poco de *Client Session State*, aunque sólo sea para mantener un identificador de sesión. Para algunas aplicaciones, se puede considerar poner todos los datos de la sesión en el cliente. En este caso, el cliente envía el conjunto completo de datos de sesión con cada solicitud, y el servidor devuelve el estado de la sesión completa con cada respuesta. Esto permite que el servidor sea completamente sin estado (Stateless Server).

Generalmente, se suele usar el *DTO* para manejar la transferencia de datos. El *DTO* puede ser serializado a través del *request* y así permitir la transferencia de datos complejos.

El cliente también necesita almacenar los datos. Si nuestro cliente es una aplicación de cliente enriquecido, puede hacerlo dentro de sus propias estructuras. Estos podrían ser los campos en la interfaz del cliente. Sin embargo, un conjunto de objetos no visuales a menudo es la mejor elección. Este podría ser el propio *DTO* o un modelo de dominio. De cualquier manera no suele ser un gran problema.

Por otro lado, si tenemos una interfaz *HTML*, entonces las cosas se

complican un poco más. Hay tres formas comunes de hacerlo: parámetros de *URL*, campos ocultos y *cookies*.

Los **parámetros de *URL*** son los más fáciles para trabajar con una pequeña cantidad de datos. Básicamente, significa que todas las *URL* en cualquier página de respuesta agregan el estado de la sesión como parámetros de la misma. La principal limitación para hacer esto es que el tamaño de una *URL* es acotado. Sin embargo, si sólo tiene un par de elementos de datos, esto funciona bien, por lo que es una opción popular para algo como un *ID* de sesión.

Algunas plataformas realizan una reescritura automática de *URL* para agregar un *ID* de sesión. Cambiar la *URL* puede ser un problema con los marcadores, por lo que es un argumento en contra de su uso para los sitios de consumidores.

Un **campo oculto** es un campo enviado al navegador que no se muestra en la página web. Se obtiene utilizando una etiqueta de la forma *<INPUT type="hidden">*. Para hacer que los campos ocultos funcionen, es necesario serializar el estado de la sesión en el campo oculto cuando se retorne la respuesta y volver a leerlo en cada solicitud. Luego, se debe elegir un formato para poner el los datos en el campo oculto. *XML* o *JSON* son una opción estándar obvia, pero también puede codificar los datos en algún esquema de codificación basado en texto. Hay que tener en cuenta que un campo oculto solo se oculta de la página mostrada, cualquiera puede mirar los datos mirando el código fuente de la página.

La última, y en ocasiones controvertida elección, son las ***cookies***. Las *cookies* se envían de ida y vuelta automáticamente. Al igual que un campo oculto, puede usarse si se serializa el estado de la sesión en la *cookie*. El límite en tamaño es lo que pueda contener esa *cookie*. Uno de los problemas de este enfoque es que a muchas personas no les gustan las *cookies*, y como resultado, las desactivan, haciendo que el sitio deje de funcionar. Sin embargo, cada vez más y más sitios dependen de las *cookies*, por lo que esto sucederá con menos frecuencia, y ciertamente no es un problema para un sistema puramente interno. Las *cookies* tampoco son más seguras que cualquier otra cosa, por lo que se puede suponer que se pueden realizar búsquedas de todo tipo.

Las *cookies* funcionan sólo dentro de un único nombre de dominio, por lo que si el sitio está separado en diferentes nombres de dominio, las *cookies* no viajarán entre ellas.

Algunas plataformas pueden detectar si las cookies están habilitadas, y si no, pueden usar la *reescritura de URL*.

3.11.2 Server Session State

Guarda el estado de sesión en el servidor, persistiéndolo de una manera serializada.

La forma más simple de este patrón se produce cuando un objeto de sesión se mantiene en la memoria de un servidor de aplicaciones. En este caso,

simplemente se puede tener algún tipo de mapa en la memoria que contenga estos objetos de sesión codificados por un *ID* de sesión. Luego, todo lo que el cliente necesita hacer es proporcionar el *ID* de sesión y el objeto de sesión podrá recuperarse del mapa para procesar la solicitud.

Una de las principales características de este patrón es su simplicidad. En la mayoría de los casos no es necesario realizar ningún esfuerzo de programación para que funcione, ya que todo el trabajo lo proporciona el servidor de aplicaciones. Si las herramientas que provee el servidor no alcanzan, el esfuerzo que se necesita sigue siendo bajo. Serializar un *BLOB* en una tabla de base de datos puede resultar en mucho menos esfuerzo que convertir los objetos del servidor a una forma tabular.

Por otro lado, el mayor problema de este patrón es la escalabilidad, que se resuelve con estrategias como *sticky sessions* (en base a la IP, por ejemplo, todas las requests de un usuario irían a parar al mismo nodo), o migración de datos de sesión entre nodos del *clúster* (si se detecta que una petición de una transacción en curso va a parar a otro nodo).

Capítulo IV

Principios arquitectónicos

“Estos son mis principios. Si no le gustan, tengo otros.”

Groucho Marx

Es recomendable diseñar soluciones de software con la idea de mantenibilidad en mente. Es decir, tener muy en claro a la hora de diseñar que el sistema debe ser mantenido a lo largo del tiempo, y que ese mantenimiento no debe ser costoso.

Los principios descritos en este capítulo son una guía para desarrollar y tomar decisiones arquitectónicas que resulten en aplicaciones robustas y mantenibles. Usualmente, estos principios permiten construir aplicaciones a partir de componentes discretos que no están estrechamente relacionados o acoplados a otras partes de la misma, sino que se comunican a través de interfaces explícitas o sistemas de mensajería.

Cuando estos principios se aplican en conjunto es más probable que un desarrollador cree un sistema que sea fácil de mantener y ampliar en el tiempo.

4.1 Separación de Intereses

Un principio clave a la hora de diseñar y desarrollar una aplicación es el principio de *Separación de Intereses* (en inglés, Separation of Concerns).

Este principio afirma que el software debe separarse en función de los tipos de tareas que realiza. Es decir, el sistema debe dividirse en secciones distintas, tal que cada sección se enfoque en un interés delimitado. Un *interés,* es un conjunto de información que afecta al código de un programa. Puede ser algo tan general como los detalles del hardware para el que se va a optimizar el código, o tan concreto como el nombre de una clase que se pretende instanciar. Un programa o aplicación que utiliza una buena separación de intereses es un *programa modular* [24].

La modularidad, y por tanto la separación de intereses, se consigue a través de la encapsulación de información en una sección de código que tiene una interfaz bien definida.

El valor de la separación de intereses es simplificar el desarrollo y mantenimiento de los sistemas. En este sentido, cuando los intereses están bien separados, se pueden reutilizar, desarrollar y actualizar las distintas secciones individuales de forma independiente. La posibilidad de modificar una parte del código del programa sin tener que revisar y modificar las demás es de gran valor en el mantenimiento de software.

Por ejemplo, consideremos una aplicación que incluye lógica para identificar elementos notorios para mostrar al usuario, y que formatea dichos elementos en una manera particular para hacerlos aún más notorios. El comportamiento responsable de elegir qué elementos formatear debe estar separado del comportamiento responsable de -efectivamente- formatear los elementos, ya que se trata de “intereses separados” que sólo coinciden uno con el otro de manera casual.

Arquitectónicamente, las aplicaciones pueden construirse lógicamente siguiendo este principio separando el comportamiento de negocio central de la infraestructura y la lógica de la interfaz de usuario. Idealmente, las reglas y la lógica de negocio deberían residir en un proyecto separado, el cual no dependa de otros proyectos de la aplicación. Esto ayuda a garantizar que el modelo de negocio es fácil de testear y pueda evolucionar sin estar estrechamente vinculado a detalles de implementación de bajo nivel.

El principio de separación de intereses es un concepto clave detrás del uso de capas en las arquitecturas de aplicaciones.

4.2 Encapsulación

La encapsulación permite que las diferentes partes de una aplicación permanezcan aisladas unas de otras. De esta manera, los componentes y las distintas capas de la aplicación deberían poder ajustar su implementación interna sin romper a sus colaboradores, - si es que no se violaron los contratos externos.

El uso adecuado de la encapsulación ayuda a lograr un acoplamiento flexible y modularidad en los diseños de aplicaciones, ya que los objetos y paquetes se pueden reemplazar con implementaciones alternativas siempre que se mantenga la misma interfaz.

En las clases, la encapsulación se logra al limitar el acceso externo al estado interno de la clase. Si un actor externo desea manipular el estado del objeto, debe hacerlo a través de una función bien definida, en lugar de tener acceso directo al estado privado del objeto. Del mismo modo, los componentes de la aplicación y las aplicaciones mismas deberían exponer interfaces bien definidas para que las utilicen sus colaboradores, en lugar de permitir que su estado se modifique directamente. Esto libera el diseño interno de la aplicación para que evolucione con el tiempo sin preocuparse de que eso afecte a los colaboradores -siempre y cuando se mantengan los contratos públicos.

4.3 Inversión de Dependencia

La dirección de las dependencias dentro de una aplicación debe estar en la dirección de la abstracción, no en los detalles de implementación. La mayoría de las aplicaciones se escriben de manera tal que la dependencia del tiempo de compilación fluye en la dirección del tiempo de ejecución. Esto produce un *grafo de dependencia directa*. Es decir, si la clase A llama a una función de la clase B, que, a su vez, llama a una función de la clase C, entonces en tiempo de compilación A dependerá de B, que dependerá de C, como se muestra en la figura 4.1.

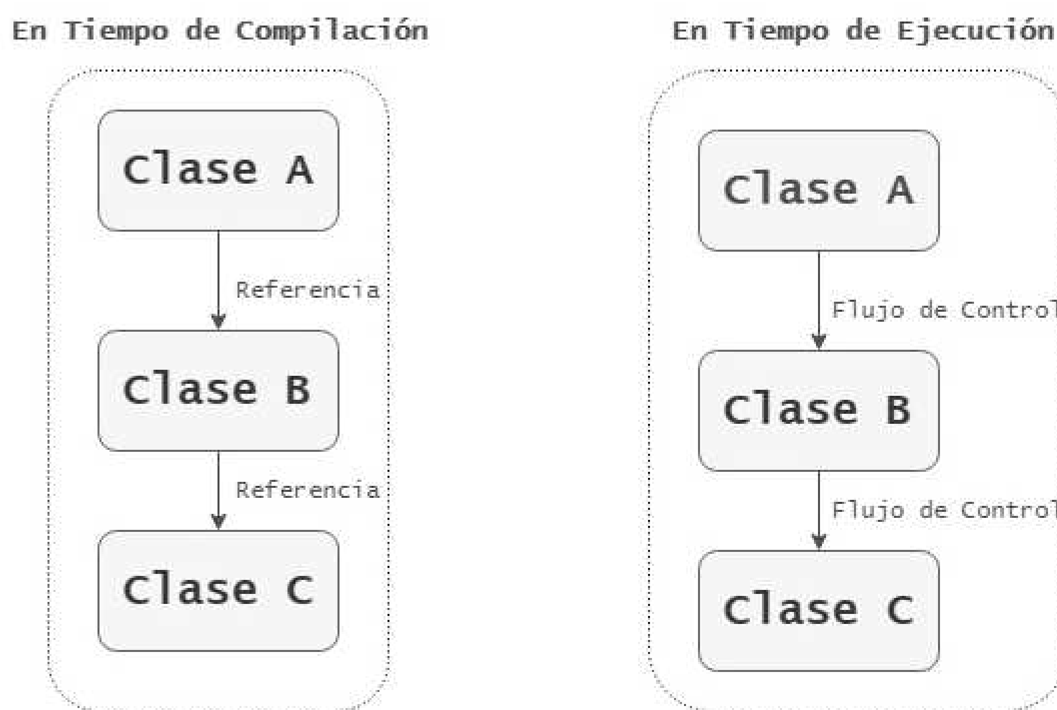

Fig. 4.1 Grafo de Dependencia Directa.

El uso del principio de inversión de dependencia permite a A invocar métodos sobre una abstracción que B implementa, haciendo posible que A llame a B en tiempo de ejecución, pero que B dependa de una interfaz controlada por A en tiempo de compilación (invirtiendo así la típica dependencia en tiempo de compilación). En tiempo de ejecución, por su parte, el flujo de ejecución del programa permanece sin cambios, pero la introducción de interfaces significa que las diferentes implementaciones de estas interfaces se pueden “enchufar” fácilmente (plugin). El grafo correspondiente se puede observar en la figura 4.2.

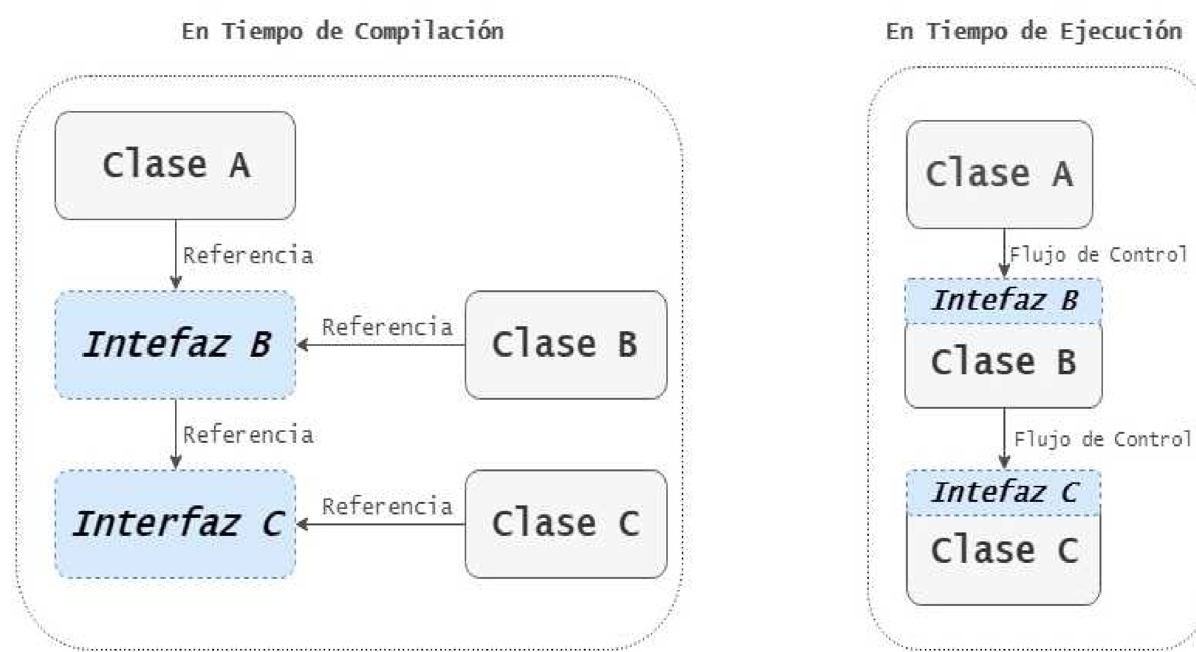

Fig. 4.2 Grafo de Dependencia Invertido.

La inversión de dependencia es una parte clave de la construcción de aplicaciones débilmente acopladas, ya que los detalles de implementación se pueden escribir para que dependan e implementen abstracciones de mayor nivel, y no al revés. Las aplicaciones resultantes son más fáciles de probar, modulares y mantenibles. Por su parte, la práctica de la *inyección de dependencia* es posible siguiendo el principio de inversión de dependencia.

Se podría resumir el principio de inversión de dependencia con la siguiente frase:

"Depender de *abstracciones*. No depender de *concreciones*."

4.4 Dependencias Explícitas

Los métodos y las clases deberían definir explícitamente cualquier objeto colaborador que necesiten para funcionar correctamente. Los constructores de clase proporcionan una oportunidad para que las clases identifiquen las cosas que necesitan para estar en un estado válido y para funcionar correctamente. Si se definen clases que se pueden construir y utilizar, pero que sólo funcionarán correctamente si existen ciertos componentes globales o de infraestructura, estas clases serán deshonestas con sus clientes. Así, el contrato del constructor le dice al cliente que sólo necesita las cosas especificadas (posiblemente nada si la clase usa un constructor predeterminado), pero luego en tiempo de ejecución resulta que el objeto realmente necesitaba algo más.

Al seguir el principio de dependencias explícitas, las clases y métodos serán honestos con sus clientes sobre lo que necesitan para funcionar. Esto hace que el código sea más autodocumentado y los contratos de código sean más fáciles de usar, ya que los usuarios confiarán en que siempre que proporcionen lo que se requiere en forma de método o parámetros de constructor, los objetos con los que trabajan se comportarán correctamente en tiempo de ejecución.

4.5 Responsabilidad Única

El principio de *responsabilidad única* se aplica fundamentalmente al diseño orientado a objetos, pero también puede considerarse como un principio arquitectónico similar a la s*eparación de intereses*. Establece que los objetos deben tener una única responsabilidad y sólo una razón para cambiar. Específicamente, la única situación en la que el objeto debería cambiar es si debe actualizarse la manera en que realiza su responsabilidad.

Seguir este principio ayuda a producir sistemas modulares y débilmente acoplados, ya que se pueden implementar muchos tipos de nuevos comportamientos como nuevas clases, en lugar de agregar una responsabilidad adicional a las clases existentes. Por su parte, agregar nuevas clases siempre es más seguro que cambiar las clases existentes, ya que ningún código depende de las nuevas clases.

En una arquitectura en capas, podemos aplicar el principio de responsabilidad única en un nivel alto a las capas en la aplicación. La responsabilidad de la *presentación* debe permanecer en el proyecto de la interfaz de usuario, mientras que la responsabilidad del *acceso a los datos* debe mantenerse dentro de un proyecto de infraestructura. La *lógica de negocio*, debe mantenerse en el proyecto central de la aplicación, donde se puede probar fácilmente y puede evolucionar independientemente de otras responsabilidades.

Cuando este principio se aplica a la arquitectura de una aplicación y se lleva a su punto final lógico, se obtienen microservicios. Un microservicio dado debería tener una responsabilidad única. Si se necesita extender el comportamiento del sistema, generalmente es mejor hacerlo agregando microservicios adicionales, en lugar de agregar responsabilidad a uno existente.

4.6 Don’t Repeat Yourself (DRY)

La aplicación debe evitar especificar el código relacionado con un concepto particular en múltiples lugares ya que es una fuente frecuente de errores.
En algún momento, un cambio en los requisitos requerirá cambiar este segmento de código, y la probabilidad de que al menos una instancia del código no se actualice generará un comportamiento incoherente del sistema.

En lugar de duplicar la lógica, es conveniente encapsularla en una construcción de programación (un módulo, por ejemplo), que sea la única

encargada de contener dicha lógica. Cualquier otra parte de la aplicación que requiera este comportamiento, usará el nuevo módulo.

No obstante, hay que tener cuidado de evitar combinar el comportamiento que es coincidentemente repetitivo. Por ejemplo, el hecho de que dos constantes diferentes tengan el mismo valor no significa que solo se tenga una constante, si conceptualmente se están refiriendo a cosas diferentes.

4.7 Ignorancia de Persistencia

La ignorancia de persistencia (en inglés Persistence Ignorance) se refiere a los objetos que deben ser persistidos, pero cuyo código no se ve afectado por la elección de la tecnología de persistencia, ya que no necesitan heredar de una clase base particular ni implementar una interfaz particular. La ignorancia de persistencia es valiosa porque permite persistir el mismo modelo de negocio de múltiples maneras, ofreciendo flexibilidad adicional a la aplicación. Las opciones de persistencia pueden cambiar con el tiempo, de una tecnología de base de datos a otra, o pueden requerirse formas adicionales de persistencia además de cualquier otra aplicación que haya comenzado.

Algunos ejemplos de violación de este principio incluyen:

- la “obligación” de heredar de una clase base,
- la “obligación” de implementar una interfaz determinada,
- clases responsables de persistirse ellas mismas (como lo indica el patrón Active Record),
- constructor default de clases obligatorio,
- propiedades que requieren una palabra clave para funcionar (como la palabra *virtual*, por ejemplo),
- atributos específicos de persistencia obligatorios.

El requisito de que las clases tengan cualquiera de las características o comportamientos anteriores agrega acoplamiento entre los objetos que se persistirán y la opción de tecnología de persistencia, lo que hace más difícil adoptar nuevas estrategias de acceso a datos en el futuro.

4.8 Contextos limitados

Los *contextos limitados* proporcionan una forma de abordar la complejidad en grandes aplicaciones u organizaciones dividiéndola en módulos conceptuales separados. Cada módulo conceptual representa un contexto separado de otros contextos (por lo tanto, limitado), y puede evolucionar independientemente. Cada contexto limitado idealmente debería ser libre de elegir sus propios nombres para conceptos dentro de él, y debería tener acceso exclusivo a su propio medio de persistencia.

Como mínimo, las aplicaciones Web individuales deben esforzarse por ser su propio contexto delimitado, con su propio espacio de persistencia para su modelo de negocio, en lugar de compartir una base de datos con otras aplicaciones. La comunicación entre contextos limitados ocurre a través de interfaces programáticas, en lugar de a través de una base de datos compartida, lo que permite que la lógica de negocio y los eventos tengan lugar en respuesta a los cambios.

Los contextos limitados se relacionan estrechamente con los *microservicios*, que también se implementan idealmente como sus propios contextos limitados individuales.

Capítulo V

Estudio de Caso: arquitectura N-Capas aplicada al Sistema de la Empresa

---

Teniendo en cuenta lo expuesto en secciones anteriores, en este capítulo se realizará un estudio de caso del modelo N-Capas aplicado en el Sistema de la empresa, analizando los detalles y customizaciones que se llevaron a cabo según las necesidades y las tecnologías utilizadas.

En este sentido, se describirán detalles del diseño arquitectónico de la aplicación, mostrando las ventajas de atenerse a los principios de los buenos diseños y a las buenas prácticas, justificando con la experiencia y literatura las decisiones tomadas.

5.1 Introducción al Sistema de la Empresa

La aplicación que se analizará es un *Sistema Integrado de Administración de Seguros*, que involucra tanto seguros corporativos como de líneas personales.

El sistema abarca desde el establecimiento del contacto ó prospecto que opera como potencial asegurado, pasando por el asesoramiento, la comercialización o venta del seguro previa cotización y/o licitación de costos, y hasta el cobro de las primas respectivas con la posterior liquidación a las Aseguradoras.

Además permite realizar una completa gestión y seguimiento de los siniestros que pudieran ocurrir.

Facilita el control de todos los circuitos asociados a la operatoria del seguro, contemplando la resolución de -entre otras- las siguientes funcionalidades:

- Cotización de Seguros Nuevos y Renovaciones
- Emisión de Pólizas y Endosos
- Facturación o recepción de Pólizas y Endosos emitidos por las Aseguradoras.
- Gestión de pagos por vías manuales y/o automáticas.
- Administración de Rendiciones o Facturaciones a Aseguradoras, Agentes Vendedores y Vinculados.
- El control de toda la documentación que el sistema genera.
- Análisis de Estadísticas.

- Generación de Reportes a medida.
- Manejo de múltiples Empresas y Productores.

5.2 Aspectos generales de la arquitectura del sistema

Debido a que las aplicaciones en las cuales se desarrollan las lógicas de negocio requieren ciertas funciones de base comunes a todas, se plantea el diseño de la arquitectura como capas bien diferenciadas -tomando como referencia el modelo N-Capas-, de modo de reutilizar dichas capas de base en todas las aplicaciones que necesiten de ellas.

Estas capas de base no tienen ninguna dependencia de la capa de negocio, lo cual brinda la cualidad de reutilización buscada.

A su vez, este planteo en capas permite escalar la solución y reemplazar implementaciones de acuerdo a las necesidades tecnológicas y de negocio que se vayan presentando.

5.2.1 Modularización de Aspectos Transversales

Los aspectos transversales de la aplicación (aquellos que no tienen que ver con el negocio, pero que impactan a todo lo ancho del mismo) son modularizados, de modo que estén bien diferenciados los componentes de software que implementan cada uno de estos aspectos.

A su vez, a nivel de arquitectura se brinda soporte para la inyección de estos componentes en la misma, de modo que sean utilizados en la solución final sí -y sólo sí- son necesarios. Esto reduce la cantidad de componentes que serán puestos en producción, desplegando solamente aquellos que sean necesarios, simplificando la administración de configuración y el propio despliegue.

5.2.2 OOP (Object Oriented Programming)

Al desarrollar la arquitectura se toma como paradigma de programación la Programación Orientada a Objetos (POO). Este paradigma permite analizar, diseñar y desarrollar sistemas con un menor *gap* entre la realidad y el modelo computacional, pudiéndose implementar de esta manera sistemas más cercanos a las necesidades del cliente y con menores tiempos de desarrollo.

Por otro lado, la POO brinda herramientas que permiten que el código resultante tenga mayores niveles de flexibilidad, extensión y calidad.

Luego, la definición de objetos está contenida en los lenguajes de programación de la actualidad (.NET, Java, etc.), de modo que se facilita la integración entre sistemas construidos con dichos lenguajes, en particular a través de servicios.

5.2.3 Componentes estándar

En los lenguajes de programación modernos existe gran variedad de frameworks que brindan las funcionalidades de base que poseen gran parte de las aplicaciones Enterprise (logging, mapping objeto-relacional, inyección de dependencias, configuraciones dinámicas, etc.), siendo algunos de estos frameworks estándares de mercado.

El diseño de la arquitectura del sistema se basa en la utilización de frameworks considerados estándar (o de gran utilización en el mercado, lo cual constituye un estándar "de hecho"). Esto permite, por un lado, contar con una amplia documentación, ejemplos de uso, etc., y, por otro, aprovechar el hecho de que al ser productos de amplia utilización en el mercado, los desarrolladores, diseñadores, y arquitectos en general tienen experiencia en los mismos, lo cual redunda en un beneficio en tiempos al momento de utilizar dichos componentes, ya que se requiere una menor -o ninguna- capacitación previa.

5.2.4 Testeo unitario de componentes

Como se refirió previamente, los distintos componentes de una aplicación van sumando a la complejidad total del sistema, lo cual da un marco para la existencia de defectos en el software final.

En el caso de la arquitectura, por encontrarse en la base de la solución de negocio, la existencia de defectos en la misma provoca en la mayoría de los casos una "explosión" de defectos en la aplicación final, por un efecto multiplicativo al pasar de capa en capa, donde cada una contribuye a propagar y expandir el defecto del componente del nivel anterior.

A raíz de esto, para garantizar que la arquitectura no tenga impactos negativos en la calidad final del producto, o que éstos sean producto de la implementación del negocio propiamente dicho, se plantea la utilización del testeo unitario de componentes. De esta manera, cada componente se prueba en forma completa antes de liberarlo para su uso. Luego, también se efectúan testeos de integración para verificar el correcto funcionamiento de los componentes cuando deben trabajar en conjunto.

Como ventaja adicional, el mismo test sirve de documentación del uso del componente ya que indica cuál es el comportamiento esperado ante cierta configuración del mismo.

5.2.5 Integración con otros sistemas

Dado que las soluciones desarrolladas sobre la arquitectura en el marco de una organización en general deberán integrarse con otras aplicaciones de la misma o incluso de otra organización, éste es un aspecto que forma parte esencial de la filosofía seguida en el diseño y desarrollo de la arquitectura.

Por lo tanto, la arquitectura diseñada sigue a su vez los lineamientos y

características de un Arquitectura Orientada a Servicios (SOA), donde las aplicaciones desarrolladas sobre la misma se basan en la utilización (y reutilización) de servicios, tanto como punto de acceso a la aplicación, como así también medio de acceso de ésta hacia otros sistemas o aplicaciones de la organización o incluso de terceros.

Siendo una arquitectura SOA, los servicios serán brindados/accedidos mediante estándares, los cuales garantizan la integración con sistemas heterogéneos (tecnológicamente hablando).

5.2.6 Maximización de alineamiento con el Negocio

La arquitectura se construye con la intención de que las aplicaciones implementadas sobre la misma se encuentren alineadas con el negocio, tanto en lo que respecta a funcionalidad como a procesos.

Por lo tanto, se considera que otro de los pilares de las aplicaciones construidas sobre la arquitectura será la implementación de herramientas de BPM (Business Process Management), las cuales permitirán la definición y ejecución de procesos de negocio dentro de la aplicación, para de esta manera aumentar el grado de alineamiento entre ésta y el negocio.

Dicho alineamiento responde a la necesidad de la organización de que los procesos de negocio cumplan con los objetivos definidos para los mismos, de modo que se pueda avanzar en las estrategias definidas.

5.3 Características Técnicas

La arquitectura se implementa totalmente sobre el *Framework .NET* de Microsoft, utilizando la versión 4.6 del mismo para aprovechar la totalidad de las funcionalidades provistas en las distintas capas de la arquitectura, lo cual favorece el desarrollo al contar con herramientas de base, sin necesidad de desarrollar las mismas.

Tal como se mencionó previamente, el enfoque utilizado en la arquitectura es la incorporación de componentes de terceros, en particular los que revisten cierto aspecto de estandarización, ya sea por ser un estándar formal de mercado o por ser un componente de amplia utilización en el mismo.

De esta manera, a nivel de arquitectura se utilizan los siguientes frameworks/componentes dentro de la implementación base de la misma, pudiendo en algunos casos hacer un reemplazo por otro producto que respete las mismas interfaces y el mismo comportamiento:

- **Spring.NET,** para la implementación de Inversión de Control (IoC) e inyección de dependencias.
- **NHibernate,** para la persistencia de objetos del dominio. Este producto se basa en Hibernate (Java) el cual es un estándar de mercado, siendo NHibernate la solución para la implementación de ORM (Object-Relational

Mapping, Mappeo Objeto-Relacional en español) sobre .NET que suma más adeptos.

- **ADO.NET,** para el acceso a datos en aquellos casos donde NHibernate no performe o donde la utilización del mismo no brinde beneficios sobre ADO.NET.
- **Spring.NET, WCF,** para la publicación de los servicios consumidos por el ESB (Enterprise Service Bus, se describirá más adelante).
- **ASP.NET,** para la implementación de WebLook (se describirá más adelante), ya que dicho componente del *Framework .NET* permite la generación de páginas que pueden ser renderizadas en los exploradores de mayor uso en el mercado. A nivel de arquitectura estarían soportadas también las plataformas Silverlight y WPF, pero la implementación del Frontend Web se desarrolla en ASP.NET. En el caso de que algún cliente requiera sus propia versión del Frontend, puede optar por estas otras plataformas en función de sus necesidades de soporte a browsers y usabilidad.
- **Ajax,** para la implementación de RIA (Rich Internet Application, o Aplicación de Internet Enriquecida) a través de comunicaciones asincrónicas con el servidor, de modo de aumentar la usabilidad del sistema disminuyendo la cantidad necesaria de *postbacks* o recargas de página.

Luego, para la integración con otras aplicaciones/sistemas, dependerá del caso la inclusión de algún otro Framework y/o el desarrollo de las correspondientes interfaces que lo encapsulen.

5.4 Diseño Arquitectónico

A continuación, en la figura 5.1, se puede observar el esquema general de la arquitectura, en el cual se encuentran las distintas capas que se conectan e interactúan entre sí.

Luego, se presentará una breve reseña al respecto del significado de cada componente, las cuales, en líneas generales, siguen el modelo arquitectónico N-Capas descrito en capítulos anteriores.

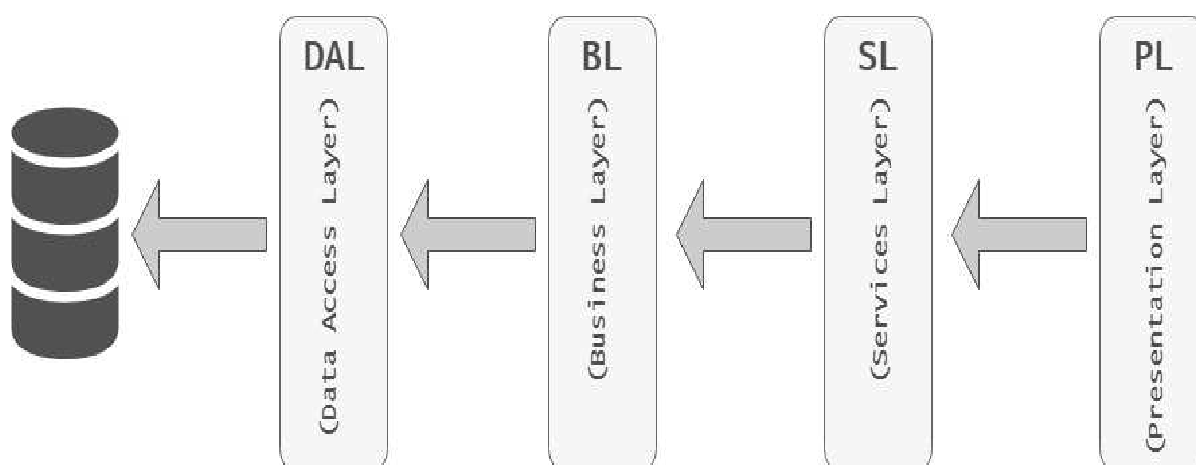

Fig. 5.1 Esquema general de la Arquitectura del Sistema (1).

5.4.1 DAL (Data Access Layer)

Esta capa se encarga de encapsular el acceso a la base de datos, proveyendo servicios CRUD (Create / Read / Update / Delete) al resto de la arquitectura.

De esta forma, el resto de la aplicación se ve liberada de lidiar con los aspectos del acceso a base de datos.

5.4.2 BL (Business Layer)

Esta capa se encarga de encapsular la lógica de negocio *core* del sistema. Es decir, en esta capa se encuentran implementados los componentes que contienen la lógica de negocio en forma independiente a cómo están instrumentados los procesos de negocio.

5.4.3 SL (Services Layer)

Esta capa se encarga de publicar los servicios de negocio, ocultando los componentes de la BL de otros sistemas y de la capa de presentación. Las interacciones con la aplicación deben llevarse a cabo a través de esta capa. Es la capa responsable de orquestar cada caso de uso, delegando al modelo de dominio -la BL-, la responsabilidad de la lógica de negocio en sí. Aspectos como la seguridad, auditoría, transacciones se ubican aquí.

Los servicios convierten los *DTOs* de entrada en objetos de negocio y viceversa. Nunca deben retornar objetos de negocio.

Es el punto de entrada al sistema, siguiendo el esquema SOA, para los diferentes consumidores (que pueden ser otras aplicaciones/sistemas o la PL del propio sistema).

5.4.4 PL (Presentation Layer)

Esta capa se encarga de los aspectos de presentación al usuario de la aplicación, consumiendo el negocio a través de la capa de servicios.

Por otra parte, una visión más detallada del esquema arquitectónico del sistema se puede apreciar en las figuras 5.2 y 5.3. Aquí se observan los componentes que conforman el sistema en su conjunto e interactúan para lograr el alineamiento del negocio y sus diferentes procesos, como así también la interacción que proveen hacia y desde otros sistemas.

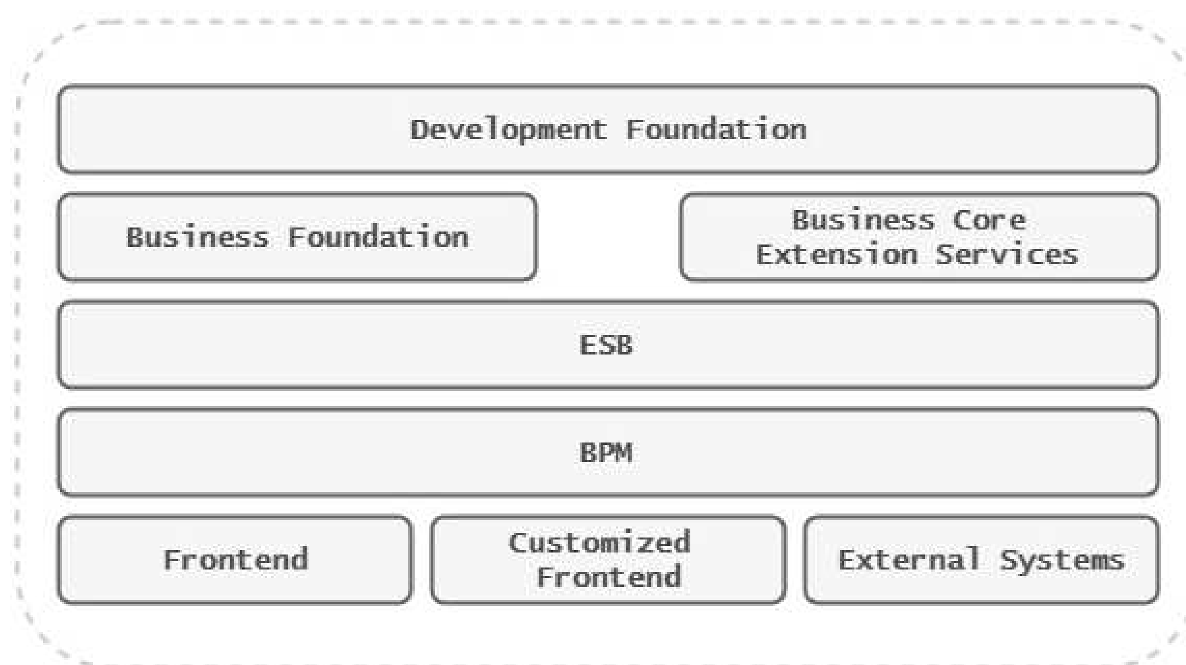

Fig. 5.2 Componentes de la Arquitectura del Sistema (1).

5.4.5 Development Foundation

Esta capa provee de componentes de servicios de bajo nivel, los cuales brindan acceso a elementos de la arquitectura independientes del negocio y compartidos por la familia de productos de la empresa.

Estos componentes podrán ser utilizados tanto por aplicaciones hechas por la empresa como por aplicaciones construidas en el cliente (o tercerizadas por él).

5.4.6 Business Foundation

Esta capa provee de los servicios de negocio consumibles en la aplicación. Dichos servicios condensan la lógica del negocio de la aplicación, la cual se encuentra construida sobre el *Development Foundation*, que le brinda el software de base.

5.4.7 Business Core Extension Services

Esta capa provee los servicios de extensión a la capa de negocios, mediante la cual el cliente puede extender la funcionalidad de negocio provista por la *Business Foundation*. De esta forma, el cliente puede lograr un mayor alineamiento del core del negocio implementado como base a la visión del negocio que posee.

5.4.8 ESB (Enterprise Service Bus)

Este componente provee funciones de enrutamiento, escalamiento e integración de servicios, asegurando un único punto de acceso a los servicios de la aplicación para aquellos que deseen consumirlos.

Este componente de la arquitectura estaría provisto por el producto implementado en el cliente (o que el cliente considere implementar) y no estaría dentro de lo que se considera *Development Foundation*.

Mediante esta herramienta se efectuaría la orquestación de los servicios de negocio, los cuales responderán a los procesos de negocio diseñados y desplegados en la herramienta de *BPM*.

5.4.9 BPM (Business Process Management)

Este componente provee soporte para la definición y despliegue de procesos de negocio, brindando el conjunto de KPIs (Key Performance Indicators) que permitan evaluar los resultados de aplicar dichos procesos de negocio.

Dentro de la *Development Foundation* se proveerá de una implementación de Workflow que brindará una aproximación a BPM como parte de la arquitectura.

En caso de que las necesidades de definición de procesos de negocio del cliente requieran de herramientas más sofisticadas (por no cumplir dicho componente de la arquitectura las necesidades del cliente), este último podrá implementar otra herramienta de BPM la cual consuma los servicios de negocio brindados a través del ESB.

Luego, en caso de que se necesite una interacción en sentido contrario, se deberá plantear el desarrollo correspondiente con las consideraciones del caso.

5.4.10 Frontend

Esta capa representa la interfaz de usuario a través de la cual el mismo accede a las funcionalidades del sistema.

Dicha implementación del frontend tendrá las características visuales y de interacción propias de las aplicaciones implementadas por la empresa en la arquitectura y plataforma de referencia (Win, Web, etc.).

En el caso de la implementación Web del Frontend, la misma presenta las siguientes características:

- Utilización de ASP.NET para la implementación de las distintas páginas de la aplicación, aprovechando las características de renderización del Framework, las cuales ocultan a la aplicación las consideraciones propias del explorador siendo utilizado para navegar la aplicación. Este es un aspecto clave en la utilización de ASP.NET en lugar de, por ejemplo, Silverlight.
- Utilización de controles de DevExpress® y Ext.NET (Sencha) para el maquetado y el manejo de código Javascript.
- Utilización de AJAX (tecnología) para brindar RIA mediante un mayor dinamismo en el uso de la aplicación, evitando los Postback que actúan en detrimento de la usabilidad del sistema. Al efecto, se utiliza la AjaxControlToolkit que brinda esta funcionalidad en ASP.NET.
- Generación dinámica de interfaces de usuario, la cual permite generar páginas en forma dinámica en función de archivos de configuración (u otros objetos construidos al efecto siguiendo una interfaz determinada), pudiendo incluirse en dicha generación componentes de tipo WebControls y UserControls creados por la empresa o por el cliente. De esta forma, la arquitectura brinda flexibilidad al cliente al momento de generar las páginas que consumirán los usuarios. A su vez, la posibilidad de de utilizar UserControls y WebControls, le da la potestad al cliente para incorporar y reutilizar comportamientos determinados en sus interfaces de usuario.
- Aplicación de estilos a las páginas de la aplicación, de modo de mantener una apariencia uniforme a lo largo de la misma, manteniendo una coherencia en toda la aplicación que favorece el aspecto de usabilidad de la misma. A su vez, permite la customización por parte del cliente de los estilos aplicados de modo que se aproxime lo más posible a la estructura visual de las aplicaciones corporativas del mismo, en caso de que cuente con un estándar de interfaces en cuanto a lo que presentación se refiere.

5.4.11 Customized Frontend

Esta capa representa la interfaz de usuario implementada por el cliente, a través de la cual el mismo accede a las funcionalidades del sistema.

Esta implementación puede consistir en customizar/ajustar el frontend provisto por *Frontend*, o en implementar un frontend directamente por el cliente sin utilizar las características brindadas por la arquitectura para la plataforma de interés (Win, Web, etc.).

Aun en el caso de que el frontend lo implemente directamente el cliente sin usar las clases base de interfaz de usuario de la arquitectura, el cliente aún podrá hacer uso de otros servicios brindados por esta última de acuerdo a sus necesidades (por ejemplo: servicios de seguridad). De esta forma, el cliente lidiará

solamente con el aspecto visual y de interacción de la aplicación, sin tener que preocuparse del resto de los aspectos involucrados en una aplicación Enterprise.

5.4.12 External Systems

Representación de los sistemas externos que interactúan con la aplicación, en uno u otro sentido. Dicha interacción se efectúa a través de los servicios de transformación e interfaces provistos en el *Development Foundation*, donde la implementación en sí dependerá de las posibilidades tecnológicas de interacción entre ambos sistemas y otros requerimientos no funcionales asociados a dicha interacción (tiempos de respuesta, disponibilidad, ventanas de tiempo, etc.).

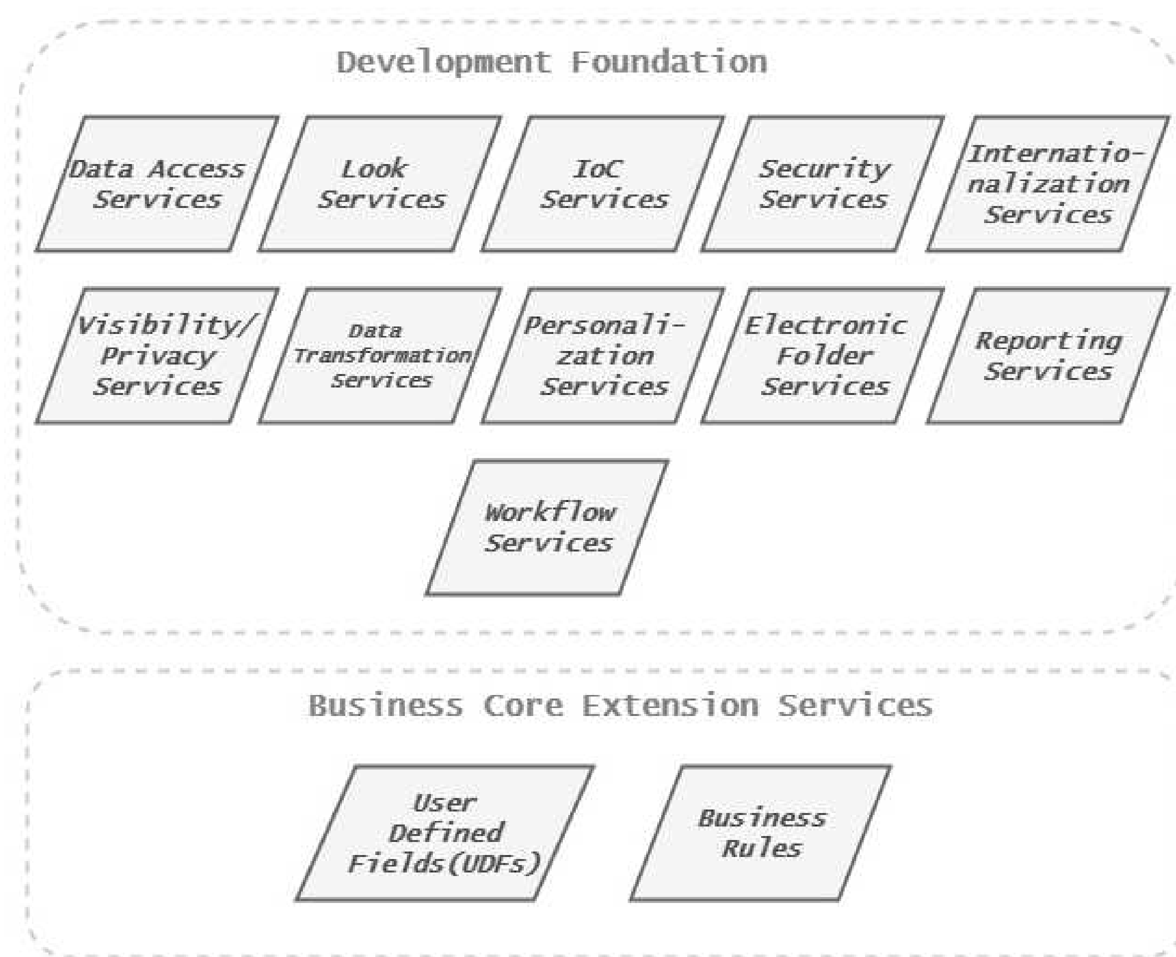

Fig. 5.3 Componentes de la Arquitectura del Sistema (2).

5.4.13 Data Access Services

Estos componentes de arquitectura encapsulan el acceso a la base de datos, y serán consumidos desde las distintas capas/componentes que requieran efectuar operaciones CRUD sobre la base de datos.

Dado que una de las premisas de la arquitectura es desarrollar bajo POO, en esta capa de acceso a datos surge la necesidad de utilizar herramientas de ORM (Object-Relational Mapping) de modo de establecer las correspondencias entre las tablas físicas y los objetos de negocio (Business Entities). En este sentido, para la implementación del acceso a datos se utiliza NHibernate como producto de ORM.

5.4.14 Look Services

Estos componentes brindan servicios de presentación para Win y Web (XLook y WebLook) a ser consumidos en la capa de presentación. Estos servicios están relacionados con la obtención de definiciones al respecto de entidades, relaciones, etc.

5.4.15 IoC Services (Inversion of Control Services)

Estos componentes brindan servicios de *Inyección de Dependencia* e *Inversión de Control*. Mediante estos servicios se dota a la aplicación de la capacidad de intercambiar implementaciones en distintos aspectos de la misma (acceso a datos, lógicas de negocio, etc.) brindándole al cliente flexibilidad en la implementación.

A su vez, desde el punto de vista del diseño, permite desarrollar las distintas partes de la aplicación sin atarse a una implementación en particular, lo cual es sin duda una buena práctica de desarrollo de software.

5.4.16 Security Services

Estos componentes brindan servicios de seguridad a lo largo de toda la arquitectura. Dentro de dichos servicios se encuentra la implementación de autenticación y autorización, lo cual es la base para la definición del acceso a los sistemas y para la definición de los perfiles de permisos de los usuarios que hagan uso de dichos sistemas.

5.4.17 Internationalization Services

Estos componentes brindan servicios de internacionalización a las capas de presentación de datos, ya que se encarga de traducir los términos encontrados en las interfaces de usuario al lenguaje/dialecto correspondiente.

De esta forma, la misma aplicación puede ser implementada en distintas regiones o países sin necesidad de implementar customizaciones a nivel de código o formularios al efecto.

5.4.18 Visibility/Privacy Services

Estos componentes brindan servicios de soporte para la visibilidad y confidencialidad de datos.

Por visibilidad se entiende restringir la información presentada al usuario en función de su pertenencia a un determinado grupo, empresa, unidad de negocios u otros, de modo que no pueda acceder información que no corresponda a su grupo de pertenencia.

Por confidencialidad se entiende restringir el acceso a la información en función del nivel de acceso que posee el usuario, de modo que pueda consultar la información generada por él mismo, o la información que esté disponible dentro de su nivel de acceso.

Ambos conceptos están relacionados con el punto 5.4.16 (Security Services).

5.4.19 Data Transformation Services

Estos componentes brindan servicios de transformación e importación de datos. En algunos casos estos servicios de transformación estarán dados a nivel de ESB, en relación con la transformación de datos propia de la integración de servicios cuando dos sistemas heterogéneos tienen que intercomunicarse. En el caso de que esta funcionalidad no sea brindada o implementada en el ESB, entran en escena los servicios brindados por este componente.

Es decir, su objetivo principal es dar soporte a las interfaces entre sistemas o a las interfaces de migración de datos.

5.4.20 Personalization Services

Estos componentes brindan servicios de soporte a la personalización de la aplicación, de modo de establecer vínculos entre el usuario y la forma de usar el sistema. A través de estos servicios el usuario puede configurar la aplicación de modo que cubra mejor sus necesidades de usabilidad, mejorando la manera en que lleva a cabo su trabajo e incrementando la satisfacción del mismo con la aplicación.

5.4.21 Electronic Folder Services

Estos componentes brindan servicios de *carpeta electrónica* a las distintas entidades administradas en la aplicación. Mediante estos servicios la documentación generada en el proceso de negocio queda vinculada a la entidad que la genera y/o que es alimentada a través de dicha documentación.

Dentro de estos servicios, se encuentran las implementaciones de integración entre la *carpeta electrónica* y el sistema de Content Management que pueda necesitar implementar el cliente.

5.4.22 Reporting Services

Estos componentes brindan servicios de reporting en sí mismos o de integración con otros sistemas de Reporting. Con reporting nos estamos refiriendo tanto a la generación de reportes (planos, master-detail, customizados) como a la

utilización de herramientas de BI (Business Intelligence), tales como cubos y dashboards.

Estos servicios podrán ser consumidos desde el frontend o desde procesos (interacción desatendida), dependiendo de las características del reporte (o cubo/dashboard/otro) generado y del soporte que brinde a los distintos modos de ejecución.

En los reportes generados se aprovecharán las características de exportación brindadas por el o los componentes utilizados para tal fin.

5.4.23 Workflow Services

Estos componentes brindan servicios de soporte a la definición y ejecución de workflows, en forma integrada con la aplicación. De esta manera, la aplicación podrá generar eventos que hagan evolucionar los workflows en ejecución en distintos sentidos.

Los workflows podrán ser de larga o corta ejecución (microflujos), dependiendo del nivel de interacción humana o sincronismo que requieran. A nivel de arquitectura, en la implementación de workflow que se provea, estará el soporte para una u otra modalidad de workflow.

5.4.24 User Defined Fields (UDFs)

Estos componentes brindan servicios de definición de atributos de información en el cliente. Mediante los mismos, el cliente puede agregar información a contener en las distintas entidades de la aplicación a través de archivos de configuración, los cuales generarán a su vez los elementos de despliegue necesarios para que dicha información adicional esté disponible en el sistema, pudiendo consumirse y actualizarse la misma.

Esto permite que el cliente pueda cubrir sus necesidades de registración de información adicional, sin necesidad de incurrir en customizaciones a nivel de código, lo cual brinda un mayor grado de flexibilidad y evita costos adicionales de desarrollo.

5.4.25 Business Rules

Estos componentes brindan servicios de definición y aplicación de reglas de validación definidas por el cliente. De esta manera, el cliente puede agregarle al sistema las validaciones/reglas que considere necesarias para restringir la operación del sistema desde el punto de vista del negocio, sin que ello implique incurrir en customizaciones mediante desarrollo.

Por otra parte, agiliza la implementación de reglas de negocio al interactuar el analista de negocio directamente con el responsable del área de IT de desplegar dichas reglas en la aplicación. La interacción se produce por completo dentro del ambiente de la organización del cliente con las ventajas que ello conlleva.

Adicionalmente a la aplicación de reglas de validación, los servicios de *Business Rules* dan soporte a las herramientas de BPM para establecer los siguientes pasos a seguir en un proceso de negocio, en función del resultado de la aplicación de una regla definida. Es decir, se puede integrar el resultado de una regla a los procesos de negocio para determinar en forma dinámica cómo se lleva a cabo la ejecución del mismo, en aquellos casos donde se presenten alternativas o se deban contemplar condiciones para detener la ejecución del mismo.

5.5 Tecnologías utilizadas en la implementación de la Arquitectura

A continuación, en la tabla 5.1 se presenta un breve resumen de las tecnologías, frameworks y paletas de componentes utilizadas en la implementación de los distintos componentes de la arquitectura.

Se puede observar que la mayoría corresponden a tecnologías que son estándares del mercado, lo que -como se ha mencionado anteriormente-, es una buena práctica y otorga grandes beneficios a la hora de construir software.

Componente	Tecnologías/Frameworks	Comentarios
Data Access Services	NHibernate (ORM) / ADO.Net	NHibernate se utiliza para la persistencia de las entidades de negocio. ADO.Net se utiliza para la obtención de datos en grillas de navegación y búsqueda en selectores, si bien éstos también pueden conectarse a servicios de negocio.
Look Services	C# / XML / ADO.Net	Código propietario basado en lectura de configuraciones en archivos XML o tablas de configuración
IoC Services	Spring .NET / Service Locator (patrón de diseño) / Fluent Configuration	Componentes estándares del mercado a través de los cuales se realizan las técnicas de Inversión de Control e Inyección de Dependencias.

Security Services	C# / Membership Provider	Código propietario implementando los esquemas de seguridad tradicionales, de modo de mantener compatibilidad con aplicaciones legacy de la empresa en cuanto a la definición de usuarios, perfiles, etc. Las implementaciones de autenticación se basan en Membership Provider, de modo de utilizar los mecanismos nativos del framework .NET.
Internationalization Services	En Desarrollo/Investigación	Utilización de mecanismos nativos de Internacionalización/Localización del framework (System.Globalization), aplicados a mensajes generados en Backend y a los componentes de Frontend.
Visibility/Privacy Services	C# / Service Locator / Otros	Implementación propietaria, basada en los servicios de IoC de modo de obtener dinámicamente los Filtros de Visibilidad/Privacidad a aplicar en la obtención de los datos.
Personalization Services	En Desarrollo. C# / Service Locator / Otros	Desarrollo propietario enfocado a la customización de distintos componentes de la arquitectura/aplicaciones, la cual se basa en diversas tecnologías según el caso.

Electronic Folder Services	C# / Service Locator / Otros	Desarrollo propietario, basado en el concepto de *Carpeta Electrónica*, manteniendo compatibilidad con versiones legacy del sistema de modo de poder consultar/generar información en la CE de las entidades desde módulos versión 5 y módulos versión 3/4 (Delphi)
Reporting Services	En Desarrollo/Investigación	Actualmente se encuentran implementados reportes utilizando Report Viewer (Visual Studio 2008). Se encuentra en estado de investigación/desarrollo la utilización de otras paletas de componentes de Reportes (por ejemplo, DevExpress ®) y la utilización de productos/frameworks para la ejecución programada de reportes
Workflow Services	En Desarrollo/Investigación	Se encuentra en desarrollo la implementación de la API interna de integración con motores de Workflow de los productos. En Investigación/desarrollo la conexión entre la API interna de integración con algunos motores específicos (Bizztalk ®) y con Workflow Foundation (.NET 3.5/4)
User Defined Fields (UDFs)	C# / NHibernate / Fluent Configuration / ADO.Net	Desarrollo propietario, basado en la configuración Fluent del mapeo de BEs en NHibernate y la aplicación de ADO.Net para obtener la información de campos UDFs en grillas de navegación

Business Rules	En Desarrollo/Investigación	Uno de los posibles enfoques a seguir es la reutilización de los componentes de evaluación de reglas/validaciones integrada a Microsoft Workflow Foundation (componente del framework)

Tabla 5.1 Tecnologías utilizadas en la Implementación

5.6 Esquema detallado de la Arquitectura

En los párrafos anteriores de este capítulo se ha descrito la arquitectura y se han definido sus componentes junto a las tecnologías utilizadas para su realización.

El siguiente diagrama (figura 5.4) muestra un esquema más detallado de la arquitectura y permite identificar los diferentes componentes y sus interrelaciones. También se puede observar que el esquema tradicional N-Capas ha sido adaptado y customizado de acuerdo a las necesidades del sistema, siempre ateniéndose a los principios y buenas prácticas mencionadas en esta tesina.

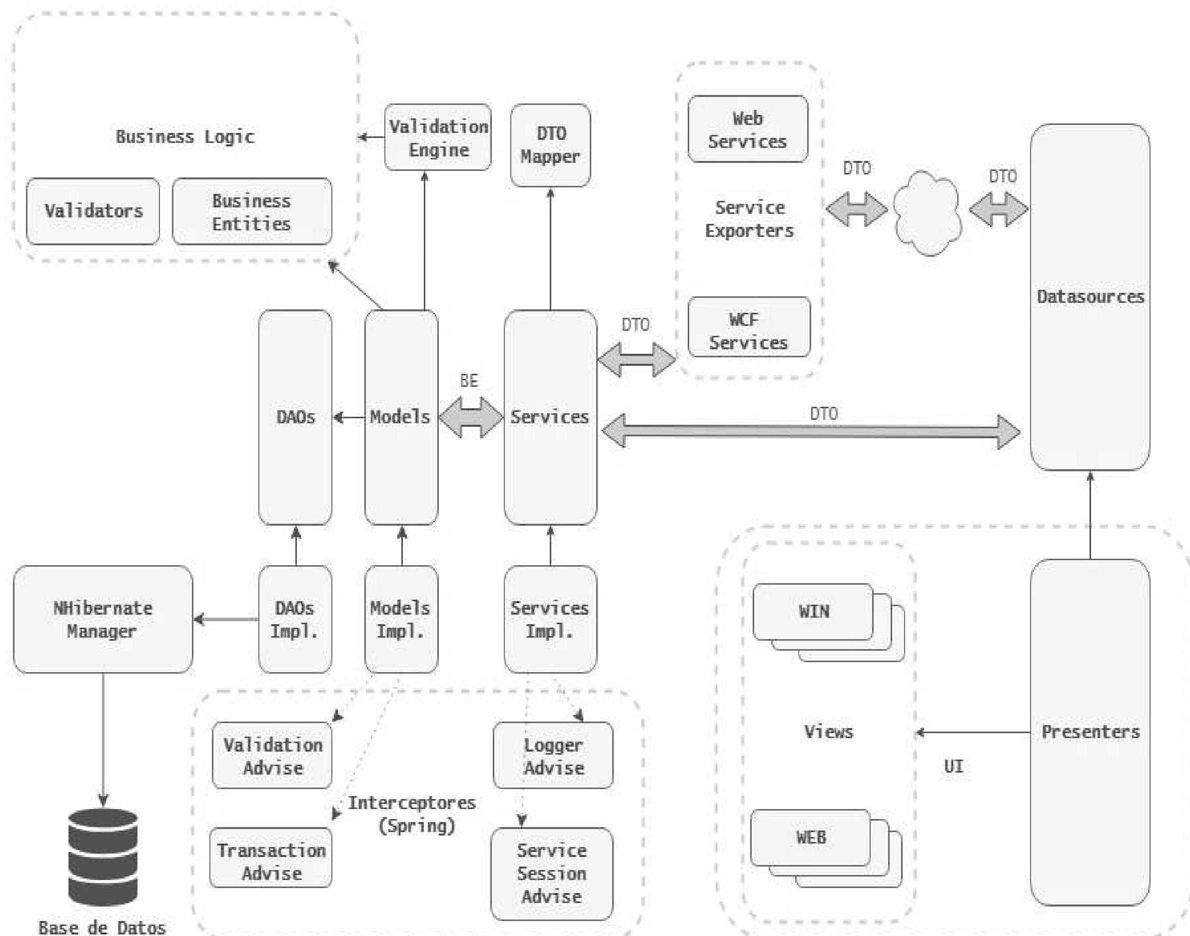

Fig. 5.4 Diagrama detallado de la Arquitectura.

A continuación se describirán sólo algunos de los componentes presentados en el diagrama, ya que muchos de ellos han sido mencionados y explicados en secciones anteriores.

5.6.1 Business Logic (Lógica de Negocio)

En esta capa se encuentra toda la lógica del negocio. Se definen todas las entidades del negocio (*BE*, *Business Entities*) que se utilizarán, junto con sus diferentes métodos y propiedades. También, en caso de que sean necesarios, se definen validadores para éstas entidades, que serán implementados en otra capa de la arquitectura.

Se puede ver a esta capa como una red de objetos que interactúan entre sí para proveer la funcionalidad de negocio requerida.

5.6.2 DAOs (Data Access Objects)

Esta capa se encarga de abstraer el acceso a los datos, a través de objetos que encapsulan la comunicación con el medio persistente, siguiendo los lineamientos del patrón de diseño *DAO (Data Access Object)*. Esto trae grandes beneficios en cuanto a la implementación, ya que no importa de qué forma o en qué lugar se encuentren los mismos, mientras cumplan con esta las interfaces definidas en esta capa.

Responde por otro lado, al patrón de diseño *Repository*, que permite guardar y obtener datos de una entidad, sin importar cómo y dónde se guardan.

En ésta capa, se definen las interfaces propias de cada *BE* -además de una interfaz genérica *IDAO*, que no posee mensajes propios sino que se utiliza para estrategias de *tipado*.

A su vez, las interfaces generadas anteriormente se implementan conjuntamente con *NHibernate*, el *ORM* (Object-Relational Mapping, mapeo objeto-relacional) mencionado en secciones previas, el cual nos permite realizar un mapeo entre las diferentes *BEs* con su correspondiente tabla de base de datos, persistir y obtener las entidades, realizar consultas, tests de persistencias, etc.

5.6.3 Models

Esta capa permite separar los servicios que serán expuestos de la capa de acceso a datos. Actúa como un proveedor de servicios interno, a través del cual se accede a la lógica de negocio y a la capa de persistencia.

Permite la Inyección de Dependencias e Inversión de Control, y es un punto central y estratégico para aplicar lógica de *AOP (Aspect Oriented Programming)* y *CCC (Cross-Cutting Concerns),* así como reglas de negocio *customs.*

Aquí también, podrán ser implementadas las validaciones creadas anteriormente.

A su vez, es el que se encarga de comunicarse con la capa de *DAOs* para obtener y persistir las *Business Entities*, y para cualquier otra lógica que requiera de acceso a datos.

5.6.4 Services

Es la capa que provee un punto de acceso a la lógica de negocio a través de los diferentes servicios, exponiendo “hacia afuera” las funcionalidades disponibles del *Backend*. Es la responsable de orquestar cada *Caso de Uso*.

Esta capa será la encargada de transformar un *DTO* en una *BE*, y *vice versa*, a través de un *DTOMapper,* que “mapea” propiedades de un DTO con propiedades de una BE.

5.6.5 DTOs

El *DTO* es el encargado de transferir los datos obtenidos de los servicios de *Backend* hasta los *Web Services o WCF (Windows Communication Foundation)*, o hasta los *Datasources*. Un *DTO* debe ser lo más plano posible, ya que sólo lleva y trae la información necesaria para el funcionamiento

5.6.6 Web Services y WCF Services

Un Servicio Web permite publicar una interfaz. WCF permite publicar los contratos de los DTO (propiedades y métodos) y los servicios correspondientes al mismo.

5.6.7 Datasources

Un *Datasource* permite acceder a los datos desde la Interfaz de usuario, creándose a partir de los *DTO*. También posee información de contexto del *DTO* (usuario logueado, datos de conexión, etc.).

5.6.8 UI (User Interface, o Interfaz de Usuario)

En esta capa se encuentra todo lo concerniente a la presentación de la información al usuario final. Se encuentra la lógica que consume los servicios a través de los *Datasources.* Puede estar desarrollada tanto en tecnologías Web como Mobile, como cualquier otra que sea capaz de consumir los servicios y presentar la información.

Capítulo VI

Conclusiones

El diseño de la arquitectura es una de las etapas más importantes y críticas dentro del desarrollo de Aplicaciones Enterprise, las cuales se caracterizan por ser un eje fundamental en la estructura de las Organizaciones y poseer un alto grado de complejidad.

La arquitectura es el cimiento del sistema, sobre el cual se garantizará el cumplimiento de los atributos de calidad, permitiendo construir un software robusto, escalable y que satisfaga los requerimientos establecidos, combinando diferentes tecnologías y plataformas de hardware y software para alcanzar un funcionamiento acorde con dichas necesidades.

Por esta razón, es clave a la hora de diseñar la arquitectura atenerse a las buenas prácticas y a los principios y patrones arquitectónicos mencionados en este documento, ya que de esa manera se consigue aplicar -y aprovechar- la experiencia adquirida de otros proyectos para lograr soluciones de calidad.

En este sentido, en este documento se ha presentado un breve resumen de la evolución de los modelos arquitectónicos más conocidos -comenzando por los sistemas monolíticos tradicionales-, para luego centrarnos en el modelo N-Capas, el cual se presenta como una gran alternativa a la hora de realizar diseños de calidad y con gran respaldo de la comunidad de arquitectos e investigadores.

La división en capas facilita la comprensión de un sistema complejo debido a que se puede hacer foco sólo en una capa en especial, en lugar de intentar entender todo el sistema, ya que cada una tiene un rol determinado dentro del mismo y sus componentes tienen que estar alineados a tal fin. A su vez, cada capa oculta los detalles de implementación al resto, favoreciendo la interacción entre sí y simplificando el mantenimiento y la escalabilidad del código.

Respetar el rol de cada capa es algo fundamental dentro de una arquitectura de este estilo, ya que permite que el código sea fácil de comprender, mantener y testear.

En esta tesina, a su vez, hemos definido y explicado los patrones arquitectónicos más utilizados por los arquitectos de software para diseñar soluciones que cumplan con la funcionalidad requerida por el sistema y que además sean de calidad.

Por otra parte, se han detallado algunos de los principios y buenas prácticas que nos sirven de guía para alcanzar esa calidad deseada y permiten construir un software fácil de mantener y extender.

A modo de conclusión de lo expuesto en esta tesina -y teniendo en cuenta la experiencia adquirida y las lecciones aprendidas durante los años de trabajo sobre la arquitectura del sistema-, podemos definir algunas buenas prácticas y pautas que se deben seguir para lograr un correcto uso y aprovechamiento de las ventajas que provee el diseño de una arquitectura basada en el modelo arquitectónico N-Capas:

- La capa de *Business Logic* no debe quedar anémica. Es decir, cada clase que representa una Entidad de Negocio (*BE*) debe tener un comportamiento específico y un sentido propio dentro del sistema. Por su parte, ninguna *BE* debe tener la responsabilidad de interactuar con los *DAOs.* Ni siquiera debería tener conocimiento alguno del medio de persistencia. Aquí entran en juego los principios de Single Responsability y Persistence Ignorance.
- En relación al punto anterior, debe ser idealmente la capa de *BL* donde se encuentre el 100% de la lógica de negocio del sistema. Esta capa debe ser completamente independiente de los detalles de persistencia y del resto de los componentes de la arquitectura. Sólo se debe encargar de proveer la funcionalidad requerida por el negocio.
- En los *DAOs* no debe haber -en la medida de lo posible- ningún tipo de código relacionado a un motor de base de datos específico. Toda comunicación con la fuente de datos se debe realizar apoyándonos en las herramientas que nos provee el framework *ORM*.
- Cada caso de uso debe poder alcanzarse a través de la capa de servicios. Si una funcionalidad no la puedo lograr a través de los servicios expuestos, y requiero de código que está implementado en la capa de presentación -por ejemplo-, significa que hay deficiencias en el diseño de los servicios.
- En este sentido, la capa de presentación se debe encargar de mostrar los datos al usuario y de tener el código relacionado únicamente con aspectos visuales. Es una falla grave que la capa de presentación posea lógica de negocio y hace que la aplicación sea difícil de mantener, escalar y testear.
- Los servicios convierten los *DTOs* de entrada en *BEs* y *vice versa*. Nunca deben retornar entidades de negocio.
- Evitar el sobrediseño. No aplicar patrones complejos o grandes diseños si la lógica que se debe agregar no lo amerita. Esto produce un código difícil de entender y mantener, repercutiendo el tiempo que demanda el proceso de desarrollo.
- No “ensuciar” la lógica de negocio *core* de la aplicación con funcionalidades propias de un cliente particular. Para ello se deben utilizar las técnicas de Inyección de Dependencias provistas por la arquitectura.

Trabajos Futuros

Gestión de versiones de la Capa de Servicios

Cuando existen muchos consumidores de una determinada API de Servicios es necesario poder administrar las distintas versiones para evitar incompatibilidades ante un cambio. Más aún, cuando el productor de la API de servicios es a la vez un consumidor y maneja los servicios según sus propias necesidades, es común que se dé una evolución descontrolada de las interfaces. Cuidar las interfaces es necesario para evitar cambios sorpresivos para los consumidores externos, trabajo extra, desvíos de tiempos, etc.

Si la API cambia, es posible que los consumidores se mantengan en una versión anterior para hacer el cambio de forma progresiva y controlada. Por otra parte, la lógica de negocio puede evolucionar, mientras que la API puede mantenerse y, por lo tanto, deben llevar versionados separados. Si la lógica evoluciona y la API no lo hace, los consumidores no se enterarán. A su vez, ante los cambios de versión, es necesario conocer rápidamente qué fue lo que se modificó: servicios nuevos, servicios eliminados, servicios que cambiaron su nombre, cambios de parámetros, etc., en conjunto con la documentación pertinente del cambio.

Hoy en día, un producto que se vende como una aplicación de servicios, debe tener personas encargadas de la gestión de la API, junto con las herramientas necesarias para hacerlo de la mejor forma.

Existen distintos frameworks que facilitan esta tarea, pero en general se trata de herramientas de bajo nivel o apuntadas a que el mismo sector de desarrollo gestione la API de Servicios. Sería deseable que sectores no técnicos puedan llevar a cabo esta tarea, ya que no sólo deben documentar la funcionalidad que brinda la API sino también interactuar con clientes externos.

Los objetivos de este nuevo trabajo serían:

- Investigar sobre las distintas formas de versionar una API de servicios, la importancia de su buena administración y documentación para, de esta forma, minimizar el impacto de los cambios de interfaces.
- Analizar las formas de manejar distintos niveles de Servicios: internos (solo consumibles por una aplicación propia del productor de los servicios) y externos o publicables (utilizables para cualquier consumidor) y la funcionalidad de cada nivel.
- Investigar los distintos componentes que existen actualmente para llevar a cabo esta tarea y utilizarlos para desarrollar una herramienta que apunte a facilitar el trabajo de sectores no técnicos.

- La herramienta desarrollada debe promover los siguientes objetivos:
  - documentación de servicios,
  - documentación de versiones de la API de servicios,
  - comparación automática de distintas versiones.

Capa extra de Servicios para los diferentes consumidores del Backend

La idea de exponer un capa extra de servicios para sistemas satélites que utilizan los servicios *core* del Backend es una idea derivada de lo expuesto en 6.1.1.

Fundamentalmente, lo que se busca con esta nueva capa -que estaría “por encima” de la capa de servicios original-, es que los consumidores externos que sólo están interesados por una parte de la lógica de negocio -y cuya demanda de servicios es más acotada-, vean una versión resumida y específica de los servicios que ofrecen dichas funcionalidades del negocio.

De esta manera, se lograría una “doble encapsulación” de la lógica de dominio, ya que la capa de servicios *core* quedaría encapsulada por esta nueva capa para todos aquellos clientes que sean externos o “satélites” (es decir, aquellos que no son el propio Frontend del sistema), permitiendo una evolución y mantenimiento más transparente de la misma.

Cualquier modificación que se realice sobre los servicios que son consumidos por otros sistemas sería absolutamente transparente siempre que se respeten los contratos de la capa superior.

Referencias

---

1. J. Conallen. Modeling web application architectures with uml. Communications of the ACM, 42(10):63–70, 1999.
2. Ahmed E. Hassan and Richard C. Holt. Architecture recovery of web applications. Software Engineering, International Conference on, 0:349–366, 2002.
3. Grady Booch. The architecture of web applications. DeveloperWorks: IBM developer solutions, 2001.
4. Grady Booch. Object-Oriented Analysis and Design with Applications (Second Edition). Addison-Wesley. 1994.
5. Paul Clements. “A Survey of Architecture Description Languages”. Proceedings of the International Workshop on Software Specification and Design, Alemania, 1996.
6. Buschmann, Meunier, Rohnert, Sommerlad, Stal. Pattern - Oriented Software Architecture, A System of Patterns. John Wiley & Sons. 1996.
7. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995.
8. Ralph Johnson, Brian Foote. Designing Reusable Classes. Department of Computer Science, University of Illinois, Urbana-Champaign. 1991.
9. Mike Potel. MVP: Model-View-Presenter: The Taligent Programming Model for C++ and Java. Taligent, Inc. 1996.
10. Klaus Renzel, Wolfgang Keller. Client/Server Architectures for Business Information Systems - A Pattern Language. 1997.
11. Pressman, R. S., & Troya, J. M. (1988). Ingeniería del software.
12. Don Roberts, Ralph Johnson. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. University of Illinois. 1996..
13. Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener. Designing Object-Oriented Software. Prentice Hall. 1990.
14. Architecting Modern Web Applications with ASP.NET Core and Azure. Steve Smith, Cesar de la Torre, Sr. Program Manager, .NET product team, Microsoft. Redmond, Washington 98052-6399. 2017
15. SOA for dummies - by Judith Hurwitz,Robin Bloor,Carol Baroudi, and Marcia Kaufman. Wiley Publishing 2007. ISBN-13: 978-0-470-05435-2 ISBN-10: 0-470-05435-2. Parte I.
16. Business Process Management. Concepts,Languages, Architectures – by Mathias Weske. ISBN 978-3-540-73521-2 Springer Berlin Heidelberg New York. 2007. Parte 1 – Capítulo 2
17. Teorías de la Cátedra Desarrollo de Software en Sistemas Distribuidos, Facultad de Informática de la UNLP. Año 2013

18. Teorías de la Cátedra Taller de Tecnologías de Producción de Software. Año 2014
19. Patricia Bazán et al. Aplicaciones, servicios y procesos distribuidos: una visión para la construcción del software; 1a ed . - La Plata : Universidad Nacional de La Plata, 2017. Libro digital, PDF: http://hdl.handle.net/10915/62354
20. Tanenbaum, A et al. Distributed Systems: Principles and Paradigms, 2e. Prentice Hall. ISBN-13: 978-1530281756. 2007
21. Weske, M. (2008) Business Process Management: Concepts, Languages, Architectures. Springer. (3, 67). ISBN 978-3-540-73521-2.
22. Christopher Alexander. A pattern language. Oxford University Press, 1977.
23. Fowler, Martin Patterns of Enterprise Application Architecture, Addison-Wesley 2002.
24. Laplante, Phillip (2007). What Every Engineer Should Know About Software Engineering. CRC Press. ISBN 0849372283.