Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

GENERACIÓN DE UNA ARQUITECTURA DE SISTEMAS INFORMÁTICOS BASADA EN SPRING.NET Y NHIBERNATE

Marvin José Narciso Arévalo
Asesorado por el Ing. Luis Estuardo Montenegro Peque

Guatemala, noviembre de 2011

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

GENERACIÓN DE UNA ARQUITECTURA DE SISTEMAS INFORMÁTICOS BASADA EN SPRING.NET Y NHIBERNATE

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

MARVIN JOSÉ NARCISO ARÉVALO
ASESORADO POR EL ING. LUIS ESTUARDO MONTENEGRO PEQUE

AL CONFERÍRSELE EL TÍTULO DE

INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, NOVIEMBRE DE 2011

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANO	Ing. Murphy Olympo Paiz Recinos
VOCAL I	Ing. Alfredo Enrique Beber Aceituno
VOCAL II	Ing. Pedro Antonio Aguilar Polanco
VOCAL III	Ing. Miguel Ángel Dávila Calderón
VOCAL IV	Br. Juan Carlos Molina Jiménez
VOCAL V	Br. Mario Maldonado Muralles
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

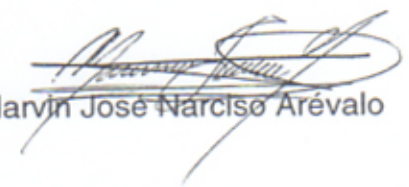
DECANO	Ing. Murphy Olympo Paiz Recinos
EXAMINADOR	Ing. César Rolando Batz
EXAMINADOR	Ing. Edgar Josue González
EXAMINADOR	Ing. Carlos Alfredo Azurdia
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

HONORABLE TRIBUNAL EXAMINADOR

En cumplimiento con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

GENERACIÓN DE UNA ARQUITECTURA DE SISTEMAS INFORMÁTICOS BASADA EN SPRING.NET Y NHIBERNATE

Tema que me fuera asignado por la Dirección de la Escuela de Ingeniería en Ciencias y Sistemas, con fecha 16 de febrero de 2011.



Marvin José Narciso Arévalo

Guatemala, 1 de noviembre de 2011.

Ing. Carlos Alfredo Azurdia Morales
Coordinador de Trabajos de Graduación
Escuela de Ciencias y Sistemas
Facultad de Ingenieria
Universidad de San Carlos de Guatemala

Estimado ingeniero:

Por este medio tengo a bien comunicarle, que he tenido la oportunidad de asesorar el trabajo de graduación titulado: **Generación de una Arquitectura de Sistemas Informáticos basada en Spring.Net y NHibernate**, elaborado por el estudiante **Marvin José Narciso Arévalo, con numero de carnet 1998-10925,** y a mi juicio cumple con los objetivos propuestos para su desarrollo.

Agradeciendo la atención prestada de antemano a la presente, me suscribo de usted.

Atentamente.

Ing. Luis Estuardo Montenegro Peque. MBA
Asesor
Luis Estuardo Montenegro Peque
Ingeniero en Ciencias y Sistemas -USAC-
Colegiado No. 9259

Universidad San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

Guatemala, 13 de Julio de 2011

Ingeniero
Marlon Antonio Pérez Turk
Director de la Escuela de Ingeniería
En Ciencias y Sistemas

Respetable Ingeniero Pérez:

Por este medio hago de su conocimiento que he revisado el trabajo de graduación del estudiante **MARVIN JOSÉ NARCISO ARÉVALO**, carné **1998-10925**, titulado: **"GENERACION DE UNA ARQUITECTURA DE SISTEMAS INFORMÁTICOS BASADA EN SPRING .NET Y NHIBERNATE"**, y a mi criterio el mismo cumple con los objetivos propuestos para su desarrollo, según el protocolo.

Al agradecer su atención a la presente, aprovecho la oportunidad para suscribirme,

Atentamente,

Ing. Carlos Alfredo Azurdia
Coordinador de Privados
y Revisión de Trabajos de Graduación

ESCUELA DE CIENCIAS Y SISTEMAS

UNIVERSIDAD DE SAN CARLOS
DE GUATEMALA

FACULTAD DE INGENIERÌA
ESCUELA DE CIENCIAS Y SISTEMAS
TEL: 24767644

El Director de la Escuela de Ingeniería en Ciencias y Sistemas de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer el dictamen del asesor con el visto bueno del revisor y del Licenciado en Letras, de trabajo de graduación titulado ***"GENERACIÓN DE UNA ARQUITECTURA DE SISTEMAS INFORMÁTICOS BASADA EN SPRING.NET Y NHIBERNATE",*** *presentado por el estudiante MARVIN JOSÉ NARCISO ARÉVALO, aprueba el presente trabajo y solicita la autorización del mismo.*

"ID Y ENSEÑAD A TODOS"

Ing. Marlon Antonio Pérez Turk
Director, Escuela de Ingeniería Ciencias y Sistemas

Guatemala, 11 de noviembre 2011

Universidad de San Carlos
de Guatemala

Facultad de Ingeniería
Decanato

Ref. DTG.479-2011

El Decano de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la aprobación por parte del Director de la Escuela de Ingeniería en Ciencias y Sistemas, al trabajo de graduación titulado: **GENERACIÓN DE UNA ARQUITECTURA DE SISTEMAS INFORMÁTICOS BASADA EN SPRING.NET Y NHIBERNATE,** presentado por el estudiante universitario, **Marvin José Narciso Arévalo,** autoriza la impresión del mismo.

IMPRÍMASE.

Ing. Murphy Olympo Paiz Recinos
DECANO

Guatemala, noviembre de 2011

/cc
c.c. archivo.

ACTO QUE DEDICO A:

Dios	Sobre todas las cosas.
Mis padres	José Efraín Narciso y María Concepción Arévalo Corzantes de Narciso, por todo el apoyo y ayuda que me han brindado siempre.
Mis hermanos	Brenda Virginia, Ángel Daniel y Fredy Mauricio, por todo su apoyo incondicional y comprensión.
Mis amigos	Con mucho cariño hacia todos, por tantos momentos de estudio y trabajo arduo.

AGRADECIMIENTOS A:

Ing. Luis Montenegro	Asesor del presente trabajo de graduación, por su valiosa asesoría, por compartir ideas de desarrollo, por sus conocimientos y todo el apoyo brindado para la culminación de mi carrera.
Ing. Carlos Azurdia	Revisor del presente trabajo de graduación, por su tiempo, ayuda y sugerencias para la elaboración de un buen trabajo.
Compañeros de trabajo	Por toda su ayuda, amistad, por compartir sus conocimientos y haberme apoyado a lo largo de la carrera.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES V
LISTA DE SÍMBOLOS VII
GLOSARIO IX
RESUMEN XV
OBJETIVOS XVII
INTRODUCCIÓN XIX

1. ANTECEDENTES DEL PROYECTO 1
   1.1. ¿Qué es una arquitectura? 1
   1.2. Situación actual de las arquitecturas 3
      1.2.1. Arquitectura evolutiva 3
      1.2.2. Arquitectura de sistemas 4
   1.3. Analogía del arquitecto de sistemas y el arquitecto de edificaciones. 5
   1.4. Análisis de la arquitectura de sistemas 9
   1.5. Creando la arquitectura de una empresa 10
   1.6. Puntos importantes en la arquitectura moderna 14
   1.7. Tipos de arquitecturas que se utilizan en la actualidad 20
   1.8. Modelos o vistas 21

2. PROTOCOLOS DE COMUNICACIÓN 23
   2.1. ¿Qué son los protocolos de comunicación? 23
      2.1.1. Reseña 24
      2.1.2. Nivel de protocolo 25
      2.1.3. Paquetes de información 26

2.1.4. Jerarquía de protocolos OSI 27
2.1.5. Interconexión e interoperatividad 29
2.1.6. Protocolos para redes e interconexión de redes 29
2.1.7. Protocolos de aplicaciones 30
2.1.8. Método de comunicaciones para *NetWare* 30
2.2. ¿Qué es http, https? 31
2.2.1. *Hypertext Transfer Protocol* 31
2.2.2. *Hypertext Transfer Protocol Secure* 32
2.3. SOAP 34
2.4. XML 36
2.4.1. Historia de XML 38
2.4.2. Estructura de XML 41
2.4.3. XML y los servicios *Web* 43
2.5. Mecanismos de transmisión y recepción 45
2.5.1. *Web Services* 45
2.5.2. *Remoting* 46
2.5.3. FTP (*File Transfer Protocol*) 47

3. IDENTIFICACIÓN DE TECNOLOGÍAS 49
3.1. Historia de Spring 49
3.2. ¿Qué es Spring.Net? 53
3.2.1. Conceptos previos 54
3.2.1.1. Inyección de dependencia 54
3.2.1.2. Programación orientada a aspectos 55
3.2.2. Ventajas 56
3.2.3. Desventajas 57
3.3. ¿Qué es NHibernate? 58

3.3.1. Características 58
3.3.2. Historia 59
3.4. Componentes de Spring.Net 61
3.5. Componentes de NHibernate 63
3.6. Manejo de la persistencia de objetos 65
3.6.1. ¿Qué se entiende por lógica de persistencia?.....65
3.6.2. ¿Dónde persisten los datos? 65
3.6.3. Estrategias de persistencia.....66
3.6.3.1. Estrategia típica.....67
3.6.3.2. Estrategia recomendada.....68

4. METODOLOGÍA DE GENERACIÓN DE LA ARQUITECTURA …..... 71
4.1. Descripción de la metodología a utilizar.....71
4.2. Descripción de la forma del manejo de la persistencia 73
4.3. Diagrama y explicación de dicha tecnología…… 77
4.4. Modelado del sistema 85

CONCLUSIONES ..... ……103
RECOMENDACIONES.....105
BIBLIOGRAFÍA 107

ÍNDICE DE ILUSTRACIONES

FIGURAS

1. Capas del modelo OSI 28
2. Código XML 37
3. Modelo de dominio 73
4. Modelo relacional 74
5. Archivo de configuración xml 75
6. Archivo hibernate.cfg.xml 76
7. Código de configuración NHibernate 77
8. Componentes Spring.Net 79
9. Componentes identificados en el escenario del ejemplo 80
10. Pantalla de ingreso a la aplicación 87
11. Pantalla de bienvenida al usuario 88
12. Pantalla del menú de expedientes 88
13. Pantalla de datos del expediente 90
14. Pantalla de ejemplo de datos 91
15. Pantalla de ejemplo de datos adjuntos 91
16. Menú expedientes 92
17. Pantalla de flujo de expedientes 93
18. Pantalla de datos de expedientes 94
19. Menú consultas 95
20. Pantalla de ingreso de datos de consultas 96
21. Pantalla que despliega la consulta de los expedientes 96
22. Menú de reportes especiales 97
23. Pantalla de ingresos de datos para reportes especiales 97

24. Pantallas para mensajes de diálogo 98
25. Menú de expedientes trasladados 98
26. Datos para el reporte de expedientes trasladados 99
27. Pantallas para mensajes de diálogo 100
28. Opción del menú para finalizar la sesión 100
29. Pantalla de finalización de sesión 101

LISTA DE SÍMBOLOS

Símbolo	Significado
</>	Fin de un tag xml
=	Igual
<>	Marcador de etiqueta en xml
#	Número
%	Porcentaje
/	*Slash*

GLOSARIO

AOP	Programación orientada a aspectos
API	Interfaz de programación de aplicaciones
Batch	En DOS, OS/2 y Microsoft Windows un archivo *batch* es un archivo de procesamiento por lotes. Se trata de archivos de texto sin formato, guardados con la extensión BAT que contienen un conjunto de comandos MS-DOS. Cuando se ejecuta este archivo, (mediante CMD) los comandos contenidos son ejecutados en grupo, de forma secuencial, permitiendo automatizar diversas tareas. Cualquier comando MS-DOS puede ser utilizado en un archivo batch.
CORBA	*Common Object Request Broker Architecture* (arquitectura común de intermediarios en peticiones a objetos).
DBA	Administrador de base de datos
DCOM	*Distributed Component Object Model* (Modelo de Objetos de Componentes Distribuidos)

DLL	Biblioteca de enlace dinámico
Framework	Estructura conceptual y tecnológica de soporte definida, normalmente con artefactos o módulos de *software* concretos, con base en la cual otro proyecto de *software* puede ser organizado y desarrollado. Típicamente, puede incluir soporte de programas, bibliotecas y un lenguaje interpretado entre otros programas para ayudar a desarrollar y unir los diferentes componentes de un proyecto.
Front-end	En diseño de *software* el *front-end* es la parte del *software* que interactúa con el o los usuarios y el *back-end* es la parte que procesa la entrada desde el *front-end.* La separación del sistema en *front-ends* y *back- ends* es un tipo de abstracción que ayuda a mantener las diferentes partes del sistema separadas. La idea general es que el *front-end* sea el responsable de recolectar los datos de entrada del usuario, que pueden ser de muchas y variadas formas, y procesarlas de una manera conforme a la especificación que el *back-end* pueda usar. La conexión del *front-end* y el *back-end* es un tipo de interfaz.
FTP	*File transfer protocol* (Protocolo de transferencia de archivos)

HTML	Lenguaje de marcado de hipertexto
HTTP	Protocolo de transferencia de hipertexto
IP	*Internet protocol* (Protocolo de Internet)
ISO	Organización internacional para la estandarización
Java script	Lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico.
LAN	Red de área local
LDAP	*Lightweight Directory Access Protocol* (Protocolo Ligero de Acceso a Directorios)
Middlewares	*Software* de conectividad que ofrece un conjunto de servicios que hacen posible el funcionamiento de aplicaciones distribuidas sobre plataformas heterogéneas. Funciona como una capa de abstracción de *software* distribuida, que se sitúa entre las capas de aplicaciones y las capas inferiores (Sistema Operativo y red).
MOM	*Message Oriented Middleware*
ODBC	*Open DataBase Connectivity*

ODBMS	*Object database management system*
ORM	*Object Relational Mapping*
OSI	Modelo de interconexión de sistemas abiertos
PC	Computadora personal
RPC	*Remote Procedure Calls*
SOAP	*Simple object access protocol*
SSL	*Secure socket layer*
TCP	*Transfer control protocol* (Protocolo de control de transmisión)
UML	Lenguaje unificado de modelado
Unix	Unix (registrado oficialmente como UNIX®) es un Sistema Operativo portable, multitarea y multiusuario; desarrollado, en principio, en 1969 por un grupo de empleados de los laboratorios Bell de AT&T.
W3C	*Word Wide Web Consortium*

Web	La *World Wide Web* es un sistema de distribución de información basado en hipertexto o hipermedios enlazados y accesibles a través de Internet. Con un navegador *Web*, un usuario visualiza sitios, compuestos de páginas *Web* que pueden contener texto, imágenes, videos u otros contenidos multimedia y navegar a través de ellas usando hiperenlaces.
Windows forms	Un formulario *Windows Form* es una representación de cualquier ventana mostrada en su aplicación. La clase *Form* se puede utilizar para crear ventanas estándar, de herramientas, sin bordes y flotantes. También, puede utilizar la clase *Form* para crear las ventanas modales como un cuadro de diálogo.
XML	*Extensible markup language*

RESUMEN

Cada día en el mundo de los sistemas de información surgen nuevos cambios y tendencias tanto para el desarrollo como para la funcionalidad de las aplicaciones, es por eso, que la arquitectura de dichos sistemas cobra un papel muy importante debido a que debe ser dinámica y adaptarse a los cambios de las nuevas generaciones, deben crearse cada vez más sofisticadas y robustas, capaces de adaptarse a la creciente diversidad de dichas aplicaciones.

Para todo tipo de sistemas comenzando por los monolíticos, hasta los sistemas más sofisticados de la actualidad, la arquitectura debe brindar a los actores implicados en el sistema plena confianza de que las aplicaciones, y los datos tendrán el trato adecuado y estarán seguros a la hora de almacenarse.

En la actualidad las arquitecturas están evolucionando para brindar todos estos elementos a las aplicaciones que soportan, y en las herramientas de Spring.Net y NHibernate se encuentran dos útiles tecnologías que permiten generar este tipo de arquitecturas modernas, ya que trabajan con los protocolos más utilizados y estándar para no depender de Sistemas Operativos de herramientas de desarrollo en específico.

En este trabajo se brinda un ejemplo práctico y sencillo para comprender mejor la forma en la que estas dos herramientas combinadas trabajan, para manejar de manera adecuada sistemas de información desarrollados en x lenguaje, así como, la forma en la que se maneja la persistencia de los datos utilizando el concepto de la programación orientada a aspectos.

OBJETIVOS

General

Investigar y analizar una arquitectura robusta para el manejo e implementación de sistemas informáticos, basada en las herramientas Spring.Net y NHibernate y hacer un análisis comparativo de las diferentes arquitecturas existentes y las ventajas y desventajas que cada una de ellas posee.

Específicos

1. Entender y conocer los aspectos que interactúan en las aplicaciones que corren sobre arquitecturas basadas en plataformas Spring.Net y NHibernate, conociendo suficientes argumentos para poder realizar una elección adecuada a la hora de diseñar una arquitectura basada en tecnologías emergentes.

2. Identificar y comprender los procesos necesarios para el diseño y desarrollo de arquitecturas utilizando plataformas nuevas.

3. Comprender qué es una arquitectura de sistemas informáticos, así como los estándares, protocolos y buenas prácticas involucradas en el desarrollo de dichas arquitecturas.

4. Tener una visión más amplia de las ventajas competitivas entre la utilización de una arquitectura y otra, esto basándose en la información de las ventajas y desventajas que se presentarán.

5. Conocer la evolución que ha sufrido Spring y NHibernate de tres años atrás, hasta la actualidad, así como, conocer el mejor momento de aplicación de dichas herramientas.

INTRODUCCIÓN

En los inicios de la informática, la programación se consideraba un arte y se desarrollaba como tal, debido a la dificultad que entrañaba para la mayoría de las personas, pero con el tiempo se han ido descubriendo y desarrollando formas y guías generales, con base en las cuales se puedan resolver los problemas. A éstas, se les ha denominado Arquitectura de *Software*, porque a semejanza de los planos de un edificio o construcción, indican la estructura, funcionamiento e interacción entre las partes del *software*.

En el libro *An introduction to Software Architecture*, David Garlan y Mary Shaw, definen que la Arquitectura es un nivel de diseño que hace foco en aspectos "más allá de los algoritmos y estructuras de datos de la computación; el diseño y especificación de la estructura global del sistema es un nuevo tipo de problema".

En el transcurso de este documento se observan los elementos más importantes a tomar en cuenta al momento de desarrollar una Arquitectura de sistemas entre los cuales destacan: los protocolos, *Web Services*, la programación orientada a aspectos, el manejo de la persistencia, etcétera.

Se repasará un poco sobre las diferentes arquitecturas y tecnologías para desarrollos que se cuentan en la actualidad y se analizarán sus ventajas y desventajas, para poder tener un parámetro de comparación con las arquitecturas desarrolladas utilizando las herramientas de estudio Spring.Net y NHibernate.

Spring.Net y NHibernate nacen de la necesidad de contar con una arquitectura robusta capaz de soportar numerosas y pesadas aplicaciones con un sinfín de transacciones, brindando un acceso confiable y seguro a distintas bases de datos, la idea fundamental de este tipo de realización de arquitecturas es que a la hora de cambiar, por ejemplo, de base de datos, no se tenga que realizar ningún cambio en el código fuente.

Básicamente sólo se tendrá que cambiar la configuración de la aplicación, esto se logra a través de la inyección de dependencia que maneja Spring.Net, es decir, se deben tener tantas clases como motores de base de datos se quiera que la aplicación soporte e inyectarlas en cualquier momento sin hacer ningún cambio en código fuente, únicamente cambiando la configuración de la aplicación.

Para finalizar se mostrará el modelo a seguir para realizar una Arquitectura con el pequeño ejemplo práctico de aplicación, los pasos que se deben seguir y el orden para lograr un producto final confiable y funcional que pueda ser utilizado en aplicaciones, sin importar la plataforma en la que estén desarrolladas o el Sistema Operativo sobre el cual corran.

1. ANTECEDENTES DEL PROYECTO

1.1. ¿Qué es una arquitectura?

Se debe entender por arquitectura en un proyecto informático a la disposición conjunta y ordenada de elementos *software* y *hardware* para cumplir una determinada función.

No es difícil comprender que si se mezclan arquitecturas distintas e inconsistentes sin ningún tipo de orden o planificación, el proyecto se puede convertir fácilmente en ingobernable, tanto o más cuanto mayor sea la envergadura del mismo. La mayoría de las organizaciones suelen favorecerse (de forma planificada o no) de algunas configuraciones concretas, la arquitectura de cada empresa debería describir estas configuraciones y el entorno que facilite crear nuevas funcionalidades que encajen en ella, incluyendo directivas, componentes de *software* reutilizables, herramientas, etcétera.

Para facilitar que las nuevas funcionalidades de que se dote a la arquitectura sean consistentes con el sistema actual y las posibles modificaciones futuras de este, se necesita conocer dicha arquitectura, pero es mucho más importante conocer la arquitectura operativa y organizacional de la empresa, una distinción importante es la que existe entre la arquitectura de una simple aplicación (micro arquitectura) y la que existe entre y a través de las distintas aplicaciones (macro arquitectura), no hace falta decir que esta última es la más compleja e importante.

¿Cuál es el papel de la arquitectura en una organización?

Es necesario imaginar que la organización consta de cuatro capas; la capa superior está formada por las actividades propias de la organización en sí misma, por debajo se encuentran las aplicaciones informáticas que soportan y facilitan esas actividades, luego de las aplicaciones se encuentra la arquitectura, que facilita que estas se desarrollen y ejecuten y en último lugar, yace la infraestructura como el *hardware* o las redes físicas.

Esta subdivisión en cuatro capas facilita determinar el papel que desempeña la arquitectura dentro de una organización, cada capa actúa como cliente de la capa inferior a ella y como servidor de la capa superior. Los arquitectos no deben malgastar su tiempo en temas relacionados con la infraestructura, tales como el Sistema Operativo, la mejor forma de separar la arquitectura de la infraestructura es tener en mente el esquema de cuatro capas antes mencionado: la infraestructura debe dar soporte a la arquitectura mezclar erróneamente conceptos de una y otra capa es un error muy común en muchas organizaciones, un error en el que no debe caer un arquitecto de sistemas es el de ser demasiado preceptivo.

Introducir demasiadas normas que creen una excesiva rigidez provocará problemas en el desarrollo de aplicaciones, un buen arquitecto de sistemas debe tener siempre en mente que su principal finalidad es permitir la creación de aplicaciones, facilitando la creatividad y la innovación de los creadores de las mismas.

La arquitectura de sistemas en los tiempos en los que solo existían los mainframes era muy sencilla: existía un lugar para cada cosa y cada cosa tenía su lugar adecuado, pero con el paso de los años y siempre en busca de una mayor flexibilidad se han ido introduciendo estructuras cada vez más y más complejas: arquitectura cliente/servidor, arquitectura a tres capas, *message brokers, data warehouses*, objetos distribuidos y arquitecturas *Webs.*

Un buen arquitecto debería empezar por recordar que su trabajo es hacer la vida más fácil a los desarrolladores y no al revés, existe otra idea que subyace tras todos los enfoques, es la especialización de dividir los problemas en sus partes constituyentes y resolverlas separadamente con equipos de especialistas centrados en un área única. La especialización deja dos puntos sin respuesta:

¿Cómo dividir los sistemas para que puedan ser definidos separadamente?
¿Cómo unirlos posteriormente para formar un todo homogéneo?

Estos son los principales retos de la moderna arquitectura de sistemas.

1.2. Situación actual de las arquitecturas

1.2.1. Arquitectura evolutiva

Si la mejora en los procesos y aplicaciones redunda en un mejor rendimiento de la empresa, y si para mejorar las aplicaciones se necesitan mejorar los sistemas, entonces la arquitectura de sistemas debería ser el vehículo de desarrollo para ambos.

En la práctica la arquitectura de los sistemas actuales constituyen, en muchos casos, grandes obstáculos para los dos, a principios de los años 90, la arquitectura de sistemas no iba más allá de una mera planificación hoy se define una arquitectura objetivo y se idea una estrategia y una planificación para completarlas dentro de determinados plazos de tiempo, donde la principal ventaja de este enfoque es que la hace comprensible a los ejecutivos, pues es similar a la forma en que tienen para dirigir sus negocios, uno de los inconvenientes es que no funciona, comienza definiendo una arquitectura objetivo y esto es un error.

El único objetivo que debe tener en mente el arquitecto de sistemas es el de la organización para la que trabaja, si no tarde o temprano entrará en conflicto con él, esta arquitectura del sistema debe ser lo suficientemente flexible como para acomodarse a los cambios de objetivos de la organización, es la clave principal para asegurar su longevidad.

1.2.2. Arquitectura de sistemas

Las organizaciones cada vez son más conscientes de la necesidad de contar con la mejor infraestructura tecnológica para operar sus negocios de una manera eficiente y competitiva, pero también, del alto costo que esto puede representar.

Ante esta situación ha surgido la necesidad de crear un rol conocido como Arquitecto de Sistemas, en el cual se ha delegado la responsabilidad de investigar, evaluar y seleccionar las mejores alternativas tecnológicas para atender las necesidades específicas del negocio a un costo razonable, es necesario dar un vistazo al origen de este rol, su analogía con los arquitectos de edificaciones, las competencias y el perfil requerido para desempeñarlo y la importancia que ha venido cobrando en las organizaciones el análisis de la arquitectura de sistemas.

1.3. Analogía del arquitecto de sistemas y el arquitecto de edificaciones

Por definición, el énfasis de la arquitectura está en las edificaciones y las estructuras habitables; sin embargo, la comunidad de ingeniería del *software* ha adoptado los términos de arquitectura y arquitecto y ha intentado extender sus definiciones al campo de los sistemas de información.

Es común para los ingenieros de *software* mencionar la arquitectura de las edificaciones como el modelo a seguir para la construcción de una infraestructura tecnológica. Inevitablemente, el enfoque en el análisis de la arquitectura permite comparar al ingeniero de *software* con una especie de arquitecto de un sistema de *software* permitiendo esta analogía, entender que la perspectiva del observador (el usuario) es importante y que la infraestructura que se está construyendo puede tener diferentes interpretaciones dependiendo de la motivación que exista para examinarla.

Si bien, la comparación entre la arquitectura tradicional y la arquitectura de sistemas tiene su fundamento, dado que en ambos casos los observadores (usuarios) están pendientes de la arquitectura de la edificación y se preocupan de que se cumplan las diferentes dimensiones (tamaño, mantenibilidad y solidez) a través del ciclo de vida del desarrollo del proyecto; en opinión de algunos autores esta analogía no es del todo precisa, pues mientras los arquitectos de edificaciones vienen de diferentes escuelas y reciben diferente entrenamiento al de los ingenieros especializados en ventilación, concreto, calor o aire acondicionado.

Los ingenieros de *software* usualmente están familiarizados con una o más tecnologías (seguridad, comunicaciones, almacenamiento, Sistemas Operativos) utilizadas para construir un sistema y su formación les permite ser promovidos a arquitectos de sistemas al estar en capacidad de asumir en diferentes proyectos, el lugar de expertos en diferentes campos de la tecnología.

Un ingeniero de *software* que acepte asumir el rol de arquitecto de sistemas debe poseer un talento y conocimientos especiales para llevar a cabo el diseño, construcción e implementación de una arquitectura tecnológica. Las responsabilidades del arquitecto de sistemas podrían sintetizarse en articular la visión arquitectónica; conceptuar y experimentar con diferentes alternativas tecnológicas; crear modelos, componentes y documentos de especificación de interfaces y validar la arquitectura contra los requerimientos y presunciones del impacto de la alternativa seleccionada sobre la estrategia tecnológica de la organización.

Para cumplir con las responsabilidades señaladas el arquitecto de sistemas debe desarrollar ciertas competencias que le permitan asumir su rol de una manera exitosa; estas competencias se pueden categorizar en los siguientes dominios:

- Tecnología: el arquitecto debe tener un sólido conocimiento de la razón de ser de la organización, de su infraestructura tecnológica y del proceso de desarrollo de sistemas de información, sin embargo, a pesar de pertenecer también al ámbito tecnológico las actividades del arquitecto de sistemas difieren de las que comúnmente realizan los desarrolladores.

  Más allá de tener claridad acerca de los requerimientos específicos del sistema, el arquitecto debe preocuparse por las implicaciones de la selección de una solución tecnológica en los objetivos de la organización, para lo cual debe tener la visión general del sistema y construir los modelos necesarios para representar el problema y su mejor solución, explorando diferentes alternativas, preparando documentos y explicando la arquitectura a los involucrados en el proyecto.

- Estrategia de negocios: el arquitecto debe poseer un alto conocimiento de la estrategia y la lógica detrás de los negocios de la organización, así como de los procesos operativos de las diferentes áreas del negocio, los ciclos de planeación y los procesos de toma de decisiones; en resumen, un buen entendimiento del contexto del negocio de la organización.

- Política organizacional: las arquitecturas comúnmente están dirigidas a atender las necesidades de varios y diversos usuarios, por lo que usualmente requieren de la participación de diferentes desarrolladores, es decir, que serán utilizadas a través de las diferentes áreas de la organización y por diferentes equipos de desarrollo que pueden ser internos o externos, aquí el arquitecto necesita entender las expectativas de la organización y de los usuarios con respecto a la arquitectura seleccionada, y debe asegurar que permanezcan comprometidos durante el desarrollo del proyecto.

- Consultoría: durante la construcción de una arquitectura participan diferentes proveedores, desarrolladores y usuarios, por lo cual el papel del arquitecto como consultor es fundamental al asistir a los involucrados del proyecto, que se convierten en sus clientes, en la resolución de dudas, en la preparación de presentaciones y en la coordinación de las actividades necesarias para desarrollar el proyecto con éxito.

- Liderazgo: en este sentido el arquitecto actúa como el líder que infunde al equipo una visión común del proyecto, y que lo motiva a trabajar comprometido para alcanzar los objetivos propuestos con la implementación de la arquitectura seleccionada.

1.4. Análisis de la arquitectura de sistemas

Las competencias y características propias del rol del arquitecto lo capacitan para llevar a cabo el análisis de la arquitectura de los sistemas de una organización, siendo esta importante por dos razones principales; en primer lugar porque la arquitectura de un sistema no puede confundirse con el sistema en sí mismo y en segundo lugar porque este análisis permite detectar tan pronto como sea posible si el sistema que se está tratando de construir es tan solo una ilusión o si en realidad se trata de un diseño factible, sin embargo, realizar el análisis de la arquitectura del sistema no garantiza que el proyecto sea exitoso, pues aunque el análisis de factibilidad resulte favorable es posible que la implementación del sistema no sea la más adecuada.

Si bien, los resultados positivos obtenidos durante el análisis son una voz de aliento para seguir adelante, todos los resultados deben ser confirmados a través de mecanismos de medición o análisis más detallados durante las diferentes etapas del ciclo de vida del desarrollo del sistema.

Otro aspecto importante que debe tener en cuenta el arquitecto de sistemas al efectuar el análisis de la arquitectura de los sistemas de una organización es evaluar siempre para cada proyecto la posibilidad de reutilizar la arquitectura ya existente, pues en caso de lograrse, es posible casi de manera inmediata aplicar los esquemas de seguridad y administración establecidos para la organización, así como reutilizar interfaces con datos comunes, como por ejemplo, los datos de los clientes o de los productos.

1.5. Creando la arquitectura de una empresa

Diseñar la arquitectura de sistemas de una gran organización es una tarea que puede resultarle intimidatoria a mucha gente, los requisitos que se necesitan son tres para empezar: un departamento de arquitectura, la redacción de un anteproyecto de diseño de la arquitectura y un documento que recoja los valores que se intentan impulsar con ella. Es fácil de olvidar, dado el alcance actual, que el uso de la informática en las organizaciones es un fenómeno relativamente reciente no es de extrañar, que las estructuras y procesos para obtener un valor real de esta inversión hayan retrasado la inversión en sí misma, el resultado de todo este ha sido un gigantesco enredo.

La tónica seguida generalizadamente hasta el momento por la mayoría de las empresas, ha sido buscar la robustez de los estándares, comprando toda su tecnología a un único fabricante o exigiendo el desarrollo de aplicaciones que funcionen sobre una rígida arquitectura.

Lo primero es crear la dirección de arquitectura (que puede estar formada por un equipo de personas o alguien a tiempo parcial, según el volumen de la organización) que coordine y se asegure que la arquitectura del sistema facilita a las aplicaciones existentes (y futuras) la consecución de los objetivos de la empresa. Las tareas del departamento de arquitectura implican una amplia visión de la actividad de la organización, estar al día de los más relevantes progresos tecnológicos y asegurarse de que los equipos que trabajan en el desarrollo de aplicaciones cumplen las directivas oportunas.

El jefe del departamento de arquitectura debe jugar entre el punto de vista del empresario y el del técnico, debiendo ser pragmático antes que idealista, evangelista antes que dictador y, por supuesto de la absoluta confianza del director de sistemas de la organización.

Para comprender donde no se debe meter el departamento de arquitectura, se debe volver al modelo de cuatro capas de la organización: los desarrolladores son sus clientes y los responsables de la infraestructura sus proveedores.

Los arquitectos de sistemas que pasan demasiado tiempo eligiendo Sistemas Operativos o rediseñando las líneas de negocio de su organización no invierten el tiempo necesario en su trabajo. De cualquier modo, la línea de división entre infraestructura y arquitectura puede ser muy difícil de ver; muchos de los elementos que en un momento dado forman parte de la arquitectura, posteriormente se consideran parte de la infraestructura, como ha ocurrido con las redes locales y ocurrirá con los sistemas de mensajería. En cualquier caso, es responsabilidad del jefe de arquitectura es promover este tipo de progresos porque como es lógico, con una débil infraestructura no es posible edificar una sólida arquitectura.

Una importante clave en el enfoque moderno de la informática es ver las aplicaciones como una colección de servicios, dichos servicios deberían ser, hasta donde sea posible, independientes unos de otros, de forma que se puedan sustituir, eliminar o añadir nuevas funcionalidades sin demasiado esfuerzo.

Separar de esta forma los servicios proporciona un valor agregado que es poder elegir la mejor tecnología para cada uno de ellos: los datos de los clientes en una base de datos relacional, los de los empleados en un directorio que cumpla las especificaciones LDAP y las descripciones de los productos en un servidor *Web*.

El problema en este paisaje es que la mayoría de los servicios con que cuenta una organización no están correctamente empaquetados y con sus interfaces bien definidas, sino que se encuentran sepultados bajo aplicaciones existentes en paquetes cerrados de sistemas propietarios, por esto es necesario un anteproyecto del sistema que se desea tener de forma que se pueda trazar un camino para llegar hasta él.

En este documento deben aparecer los servicios que se tienen y las aplicaciones que los usan, se podrían señalar también los puntos donde esos servicios ya existen y donde es preciso crearlos. Cualquier nuevo diseño debe responder a dos preguntas:

¿De qué servicios existentes se puede hacer uso?
¿A qué nuevo servicio se puede contribuir?

Esto supone no volver a pensar en aplicaciones aisladas para comenzar a pensar en términos de compartir servicios de aplicaciones.

El último punto es crear un documento con los valores de la arquitectura. Debe ser un documento breve, de menos de mil palabras, distribuido a todo el personal de sistemas de la empresa. Debe cubrir puntos como definir cuándo usar dos capas y cuándo tres, qué *middleware* usar y qué estándares (Sistemas Operativos, de mensajería, etcétera) no están abiertos a debate. Debe ser un documento deliberadamente minimalista, permitiendo libertad a la gente con creatividad para elegir cuál es la mejor solución para resolver el problema particular.

Para ello, hay tres ventajas fundamentales que este documento debe aportar.

- En primer lugar eliminar los debates estériles señalando claramente qué puntos no están abiertos a debate, de forma que el personal se centre en debatir los problemas reales.

- En segundo lugar ayudar a identificar al personal que cumple los mejores requisitos para trabajar con la arquitectura, de forma que se pueda utilizar este documento como criterio de selección de los nuevos técnicos.

- En tercer lugar, proporcionar a los técnicos la posibilidad de enfocar sus esfuerzos y su creatividad en campos en los que realmente serán apreciados.

1.6. Puntos importantes en la arquitectura moderna

Divulgar un conjunto de valores para la arquitectura requiere estar bien informado, el punto de vista de un jefe de arquitectura debe contemplar los siguientes puntos:

- Estándares: el alza de Internet y todas las tecnologías que van con ella está revolucionando el mundo de la informática, algunos de sus efectos son rápidamente visibles, pero otros no lo son tanto, el jefe de arquitectura debe estar bien informado de los estándares emergentes. En algunos casos se encuentra un serio problema a la hora de tomar partido por dos estándares que aparentemente, poseen la misma funcionalidad, por ejemplo, DCOM y CORBA, la decisión debe tomarse recopilando información claramente imparcial y, si no es posible o no es lo suficientemente aclaratoria, probándolos, lo peor que se puede hacer es no usar ninguno de los dos.

- La capa intermedia: el pegamento que une estas partes formando una aplicación completa es lo que se conoce como *middleware*. Es necesario imaginar dos programas (A y B) corriendo separadamente cada uno en una máquina. A llama a B con un problema, B trabaja en al resolución del mismo y cuando termina devuelve la respuesta a A.

- El *middleware* es el que permite esta funcionalidad, pero existen algunos problemas derivados de esta forma de actuación ¿qué debería de hacer el proceso A mientras que B trabaja en la resolución de su problema?, ¿esperar?, ¿cómo puede B comunicarle a A algo a menos que este se lo requiera?, ¿cómo puede B saber donde se encuentra A?, ¿qué hace A si B se cae?, ¿si B es reemplazado?, ¿si cientos de A requieren una respuesta de una única instancia de B? como se puede ver, los *middlewares* deben resolver complejas situaciones, pero sin añadir excesiva complejidad a la arquitectura.

  Existen dos tipos esenciales de *middlewares*: los basados en RPC (*Remote* Procedure *Calls*) y los basados en MOM (*Message Oriented Middleware*). *Middlewares* basados en MOM son inherentemente asíncronos y lo más difícil en ellos es definir correctamente la estructura de los mensajes. Los *middlewares* basados en RPC son más sencillos de usar, puesto que la sintaxis de las llamadas es prácticamente idéntica a la de una llamada en C, pero el rendimiento de estos sistemas suele ser más pobre.

  Existen algunos fabricantes que distribuyen productos capaces de funcionar en uno u otro modo. Las mejores herramientas hoy en día están basadas en CORBA, están orientadas a mensajes y tienen como inconveniente que son más difíciles de usar por los programadores. Las herramientas de Microsoft basadas en COM+ son más limitadas pero muy sencillas de usar y son recomendables si se antepone esta característica a la longevidad y calidad del producto.

- Estructura cliente servidor a dos o tres capas: la estructura cliente servidor a dos capas nació el día en que alguien conecto su PC a una máquina UNIX. A los usuarios les gusta la facilidad de uso de las PC y a los administradores la seguridad que les reporta un servidor UNIX, nadie lo llamó entonces arquitectura a dos capas porque no existía ninguna más.

  La novedad de contar con una interfaz gráfica en lugar de una pantalla verde en modo texto fue bien recibida, entonces los desarrolladores comenzaron a enriquecer sus productos con nuevas funcionalidades gracias a las mejores herramientas de desarrollo existentes para las PC, los clientes engordaron haciéndose más pesada y lenta la alternativa, meter parte de esta nueva funcionalidad en el *backend* no era demasiado atractiva, así que se buscó la solución introduciendo una tercera capa central situada entre el cliente y el servidor para sostener gran parte del peso de esta nueva funcionalidad.

  Pese a sus evidentes ventajas, los sistemas en tres capas han tardado en generalizarse debido a que son mucho más caros y difíciles de desarrollar. La arquitectura cliente servidor a dos capas sigue siendo útil para la mayoría de los casos y ahora aparece la tecnología *Web* para acercar la arquitectura a tres capas a las masas.

- La *Web*: la tecnología *Web* ha cambiado sustancialmente la forma en que las aplicaciones son desarrolladas. Si nada lo para, en muy poco tiempo el estándar para crear interfaces gráficos será el HTML, pero la *Web* tiene muchas más cosas que ofrecer que meramente la apariencia externa, Internet pronto conectará a la totalidad de los ordenadores del planeta, esto cambiará por completo las reglas del desarrollo de aplicaciones.

  Las organizaciones que permanezcan aisladas no podrán beneficiarse de estas grandes ventajas: clientes (actuales y potenciales), empleados y proveedores estarán continuamente conectados, también se ha popularizado un gran conjunto de tecnologías actuales y pasadas (HTML, TCP/IP, Java, etcétera. Los arquitectos tienen ahora un mayor conjunto de herramientas con que jugar. Un caso típico, una herramienta cliente podría consistir simplemente en una amistosa interfaz HTML utilizable mediante un navegador favorito.

  Este cliente se conecta a través de un servidor *Web* con una capa intermedia formada por componentes escritos en diversos lenguajes (C++, Visual Basic y Java) y empaquetados con Actives o Java Beans. Todo ello, se sustenta sobre un servidor de transacciones (como Microsoft MTS o TP Tuxedo) y se conecta a una tercera capa de aplicaciones propietarias (mediante Microsoft MQ o IBM MQSeries) y con un servidor de bases de datos vía ODBC. Todo ello, permanece unido a través de un lenguaje basado en *scripts* como Java Scripts o Visual Basic Scripts.

Arquitecturas como esta popularizan por primera vez en la historia de la informática los beneficios de los lenguajes de componentes y la estructura a tres capas.

- Empaquetado: utilizar tecnología de componentes empaquetados puede ahorrar años de desarrollo. Ahorrar en costos de desarrollo y mantenimiento y beneficia además con el *robusted* que proporciona componentes que han sido testeados por cientos de usuarios antes que la organización.

Los problemas comienzan con la personalización. Habitualmente los paquetes de *software* no cumplen exactamente el 100% de los requisitos. Pueden adaptarse ellos a la organización o lo que parece más lógico la organización a ellos.

Como regla general, si se encuentra un paquete que cumple el 80% de los requerimientos, se debe comprar y personalizar y si los requerimientos que cumple están por debajo de este porcentaje, es mucho mejor que se desarrolle el sistema la organización. Las personalizaciones complejas pueden ser caras y, a menudo, inviables donde mejor se comportan los paquetes es en los procesos que son prácticamente iguales en multitud de empresas, como la facturación, o cuando lo que se desea es personalizar el contenido y no la funcionalidad, como en sistemas de datos de gestión documental.

A veces ocurre que se desea sustituir un paquete pero parte de su funcionalidad es indispensable para otros elementos: el paquete se ha convertido en parte de la arquitectura.

Para evitar estos indeseables efectos, se deben envolver los paquetes y realizar la integración contra este envoltorio, todo esto encarece el costo de desarrollo pero, a la larga, mejora la flexibilidad del sistema.

- Envolviendo sistemas propietarios: muchos sistemas propietarios proporcionan funcionalidades muy valiosas en determinadas misiones críticas. Son fruto de años y años de experiencia y su uso en una arquitectura bien diseñada puede ser muy beneficioso, la forma de conseguirlo es envolverlo con una interfaz que lo haga comportarse como uno de los paquetes de la organización.

  A pesar de usar tecnologías obsoletas, los sistemas propietarios suelen usar tres capas bien definidas: presentación lógica y almacenamiento de datos y la mejor forma de envolverla es acceder directamente a la lógica de la aplicación. Para ello la aplicación debe proporcionar un API de no ser posible, la segunda opción sería acceder directamente a los datos almacenados.

  Esto supone un perfecto conocimiento de la estructura de almacenamiento de la aplicación y, a menudo, tener que solventar problemas de sincronización y gestión de bloqueos.

  La última línea de ataque sería combatir el nivel de presentación, el envoltorio se comunica con la aplicación igual que lo haría un usuario tecleando datos y recogiendo respuestas de la salida por medio de una interfaz de terminal. Las tres técnicas son costosas y difíciles, pero, seguramente usar la aplicación será mucho mejor que desecharla o desarrollar otra similar.

1.7. Tipos de arquitecturas que se utilizan en la actualidad

Generalmente, no es necesario inventar una nueva arquitectura de *software* para cada sistema de información lo habitual es adoptar una arquitectura conocida en función de sus ventajas e inconvenientes para cada caso en concreto. Así, las arquitecturas universales son:

- Monolítica: donde el *software* se estructura en grupos funcionales muy acoplados.

- Cliente-servidor: donde el *software* reparte su carga de cómputo en dos partes independientes pero sin reparto claro de funciones.

- Arquitectura de tres niveles: especialización de la arquitectura cliente-servidor donde la carga se divide en tres partes (o capas) con un reparto claro de funciones: una capa para la presentación (interfaz de usuario), otra para el cálculo (donde se encuentra modelado el negocio) y otra para el almacenamiento (persistencia). Una capa solamente tiene relación con la siguiente.

Otras arquitecturas afines menos conocidas son:

- En pipeline
- Entre pares
- En pizarra
- Orientada a servicios
- Máquinas virtuales

1.8. Modelos o vistas

Toda arquitectura de *software* debe describir diversos aspectos del *software*. Generalmente, cada uno de estos aspectos se describe de una manera más comprensible si se utilizan distintos modelos o vistas. Es importante destacar que cada uno de ellos constituye una descripción parcial de una misma arquitectura y es deseable que exista cierto solapamiento entre ellos. Esto es así, porque todas las vistas deben ser coherentes entre sí, evidente dado que describen la misma cosa.

Cada paradigma de desarrollo exige diferente número y tipo de vistas o modelos para describir una arquitectura. No obstante, existen al menos tres vistas absolutamente fundamentales en cualquier arquitectura:

- La visión estática: describe qué componentes tiene la arquitectura
- La visión funcional: describe qué hace cada componente
- La visión dinámica: describe cómo se comportan los componentes a lo largo del tiempo y cómo interactúan entre sí.

Las vistas o modelos de una arquitectura pueden expresarse mediante uno o varios lenguajes, el más obvio es el lenguaje natural, pero existen otros lenguajes, tales como, los diagramas de estado, los diagramas de flujo de datos, etcétera, estos lenguajes son apropiados únicamente para un modelo o vista. Afortunadamente, existe cierto consenso en adoptar UML (*Unified Modeling Language*, lenguaje unificado de modelado) como lenguaje único para todos los modelos o vistas, sin embargo, un lenguaje generalista corre el peligro de no ser capaz de describir determinadas restricciones de un sistema de información (o expresarlas de manera incomprensible).

2. PROTOCOLOS DE COMUNICACIÓN

2.1. ¿Qué son los protocolos de comunicación?

Los protocolos son reglas de comunicación que permiten el flujo de información entre computadoras distintas que manejan lenguajes distintos, por ejemplo, dos computadoras conectadas en la misma red pero con protocolos diferentes no podrían comunicarse jamás, para ello, es necesario que ambas hablen el mismo idioma, por tal sentido, el protocolo TCP/IP fue creado para las comunicaciones en Internet, para que cualquier computador se conecte a Internet, es necesario que tenga instalado este protocolo de comunicación. Pueden estar implementados bien en *hardware* (tarjetas de red), *software* (*drivers*), o una combinación de ambos.

La base de Internet y razón principal de su éxito, son sus protocolos. Los protocolos son determinadas reglas a cumplir por los dispositivos que desean comunicarse, en una manera más coloquial de explicarlo es como un idioma, los dispositivos deben aprender la gramática, la sintaxis y todas las reglas del idioma para poder comunicarse con otro dispositivo que habla ese idioma de una manera óptima y satisfactoria. Dentro de cada nivel se utilizan distintas normas o protocolos, llegando incluso a depender, dentro de un nivel, la norma utilizada del servicio a prestar.

2.1.1. Reseña

Hace unos cuantos años parecía como si la mayor parte de los fabricantes de ordenadores y *software* fueran a seguir las especificaciones de la Organización Internacional para el estándar (*International Organization for Standarization*, ISO).

ISO define como los fabricantes pueden crear productos que funcionen con los productos de otros vendedores sin la necesidad de controladores especiales o equipamientos opcionales, su objetivo es la apertura y el único problema para implantar el modelo ISO/ISO fue que muchas compañías ya habían desarrollado métodos para interconectar su *hardware* y *software* con otros sistemas, aunque pidieron un soporte futuro para los estándares OSI, sus propios métodos estaban a menudo tan atrincherados, por lo que el acercamiento hacia OSI era lento o inexistente Novell y otras compañías de redes expandieron sus propios estándares para ofrecer soporte a otros sistemas y relegaron los sistemas abiertos a un segundo plano.

Sin embargo, los estándares OSI ofrecen un modo útil para comparar la interconexión de redes entre varios vendedores. En el modelo OSI, hay varios niveles de *hardware* y el *software* por lo que es necesario examinar lo que hace cada nivel de la jerarquía, para ver como los sistemas se comunican por LAN.

2.1.2. Nivel de protocolo

Los protocolos de comunicaciones definen las reglas para la transmisión y recepción de la información entre los nodos de la red, de modo que para que dos nodos se puedan comunicar entre sí, es necesario que ambos empleen la misma configuración de protocolos.

Entre los protocolos propios de una red de área local se pueden distinguir dos grupos principales. Por un lado están los protocolos de los niveles físicos y de enlace, niveles uno y dos del modelo OSI, que definen las funciones asociadas con el uso del medio de transmisión: envío de los datos a nivel de bits y trama, y el modo de acceso de los nodos al medio. Estos protocolos vienen unívocamente determinados por el tipo de red (Ethernet, Token Ring, etcétera).

El segundo grupo de protocolos se refiere a aquellos que realizan las funciones de los niveles de red y transporte, niveles tres y cuatro de OSI, es decir, los que se encargan básicamente del encaminamiento de la información y garantizar una comunicación extremo a extremo libre de errores. Estos protocolos transmiten la información a través de la red en pequeños segmentos llamados paquetes y si un ordenador quiere transmitir un fichero grande a otro, el fichero es dividido en paquetes en el origen y vueltos a ensamblar en el ordenador destino.

Cada protocolo define su propio formato de los paquetes en el que se especifica el origen, destino, longitud y tipo del paquete, así como la información redundante para el control de errores. Los protocolos de los niveles uno y dos dependen del tipo de red, mientras que para los niveles tres y cuatro hay diferentes alternativas, siendo TCP/IP la configuración más extendida.

Lo que la convierte en un estándar de hecho. Por su parte, los protocolos OSI representan una solución técnica muy potente y flexible, pero que actualmente está escasamente implantada en entornos de red de área local. La jerarquía de protocolo OSI.

2.1.3. Paquetes de información

La información es embalada en sobres de datos para la transferencia. Cada grupo, a menudo llamados paquetes, incluyen las siguientes informaciones:

- Datos a la carga: la información que se quiere transferir a través de la red, antes de ser añadida ninguna otra información. El término carga evoca a la pirotecnia, siendo la pirotecnia una analogía apropiada para describir como los datos son disparados de un lugar a otro de la red.

- Dirección: es destino del paquete y cada segmento de la red tiene una dirección, que solamente es importante en una red que consista en varias LAN conectadas, también hay una dirección de la estación y otra de la aplicación. La dirección de la aplicación se requiere para identificar a qué aplicación de cada estación pertenece el paquete de datos.

- Código de control: informa qué describe el tipo de paquete y el tamaño, los códigos de control también códigos de verificación de errores y otra información.

2.1.4. Jerarquía de protocolos OSI

Cada nivel de la jerarquía de protocolos OSI tiene una función específica y define un nivel de comunicación entre sistemas. Cuando se define un proceso de red, como la petición de un archivo por un servidor, se empieza en el punto desde donde el servidor hizo la petición, entonces, la petición va bajando a través de la jerarquía y es convertida en cada nivel para poder ser enviada por la red.

- Nivel físico: define las características físicas del sistema de cableado, abarca también los métodos de red disponibles, incluyendo *Token Ring*, *Ethernet y ArcNet*.

  Este nivel especifica lo siguiente:

  Conexiones eléctricas y físicas

  ¿Cómo se convierte en un flujo de bits la información que ha sido paquetizada?

  ¿Cómo consigue el acceso al cable la tarjeta de red?

- Nivel de enlace de datos: define las reglas para enviar y recibir información a través de la conexión física entre dos sistemas.

- Nivel de red: define protocolos para abrir y mantener un camino entre equipos de la red, también se ocupa del modo en que se mueven los paquetes.

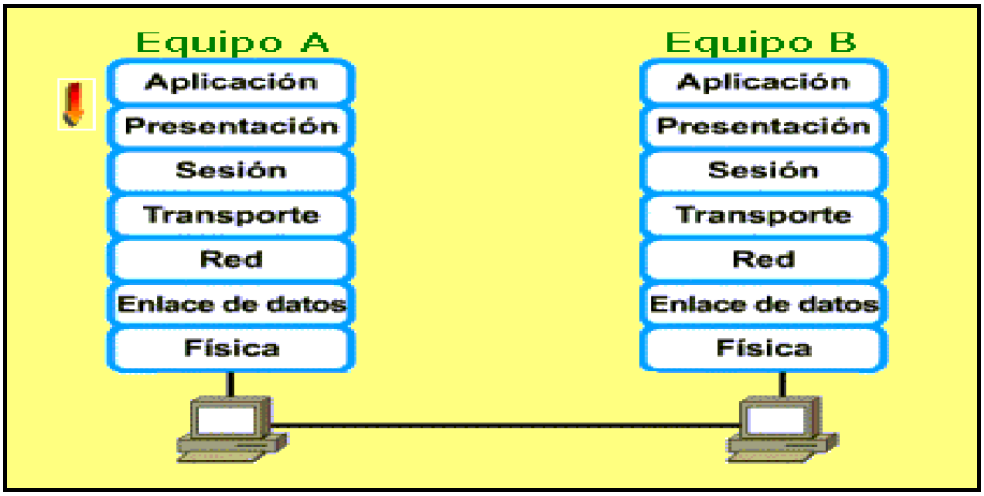
- Nivel de transporte: suministra el mayor nivel de control en el proceso que mueve actualmente datos de un equipo a otro.

- Nivel de sesión: coordina el intercambio de información entre equipos, se llama así, por la sesión de comunicación que establece y concluye.

- Nivel de presentación: en este los protocolos son parte del Sistema Operativo y de la aplicación que el usuario acciona en la red.

- Nivel de aplicación: en este el Sistema Operativo de red y sus aplicaciones se hacen disponibles a los usuarios. Los usuarios emiten órdenes para requerir los servicios de la red.

Figura 1. **Capas del modelo OSI**



Fuente: http://tecnicoensistemas2009115.blogspot.com/. 1 de febrero de 2011.

2.1.5. Interconexión e interoperatividad

Interconexión e interoperatividad son palabras que se refieren al arte de conseguir que equipos y aplicaciones de distintos vendedores trabajen conjuntamente en una red. La interoperatividad está en juego cuando es necesario repartir archivos entre ordenadores con Sistemas Operativos diferentes, o para controlar todos esos equipos distintos desde una consola central. Es más complicado que conectar simplemente varios equipos en una red. También, se debe hacer que los protocolos permitan comunicarse al equipo con cualquier otro a través del cable de la red.

El protocolo de comunicación nativo de NetWare es el SPX/IPX. Este protocolo se ha vuelto extremadamente importante en la interconexión de redes de NetWare y en la estrategia de Novell con sistemas de red. TCP/IP es más apropiado que el protocolo nativo de NetWare IPX para la interconexión de redes, así que se usa a menudo cuando se interconectan varias redes.

2.1.6. Protocolos para redes e interconexión de redes

El nivel de protocolo para redes e interconexión de redes incluye los niveles de red y de transporte; define la conexión de redes similares y en el encaminamiento (*routering*) entre redes similares o distintas. En este nivel se da la interconexión entre topologías distintas (interoperatividad) y es posible filtrar paquetes sobre una LAN en una interconexión de redes, de manera que no pasen a saltar a otra LAN cuando no es necesario.

2.1.7. Protocolos de aplicaciones

La interoperatividad se define en los niveles superiores de la jerarquía de protocolos. Se podría tener una aplicación de base de datos en la que parte del servidor trabaje en un servidor de red, y la parte de cliente lo hiciera en equipos DOS, OS/2, Macintosh y UNIX, otras aplicaciones interoperativas incluyen paquetes de correo electrónico las cuales permiten a los usuarios intercambiar archivos de correo en varios sistemas distintos (DOS, Macintosh, UNIX, etcétera) donde el *software* que se encarga de traducir de un sistema a otro cualquier diferencia que haya en la información de los paquetes de correo electrónico.

2.1.8. Método de comunicaciones para NETWARE

Esta sección trata el modo en que las estaciones tradicionales basadas en el DOS establecen comunicación con servidores NetWare por medio de SPX/IPX y también habla de soporte TCP/IP, Appel Talk y otros.

La interfaz (*shell*) de NETWARE

Para establecer una conexión entre una estación DOS y el servidor de archivos NetWare, primero se carga el *software* de peticiones del DOS (DOS *Requester*), este *software* carga automáticamente el nivel de protocolo SPX/IPX y mediante el soporte ODI permite incorporar protocolos o tarjetas de red adicionales, determinando si las órdenes ejecutadas son para el Sistema Operativo local o para el NerWare, si las órdenes son para NetWare, las dirige a través de la red y si son para el DOS, las órdenes se ejecutan en forma local.

El protocolo IPX está basado en el Sistema de red de Xerox (Xerox Network System, XNS).

El XNS, como la jerarquía de protocolo OSI, define niveles de comunicaciones desde el *hardware* hasta el nivel de aplicación. Novell utilizó el IPX de esta jerarquía (especialmente el protocolo entre redes) para crear el IPX. El IPX es un protocolo de encaminamiento, y los paquetes IPX contienen direcciones de red y de estación, toda esta información va en el paquete en forma de datos de cabecera.

2.2. ¿Qué es HTTP, HTTPS?

2.2.1. *Hypertext Transfer Protocol*

El protocolo de transferencia de hipertexto (HTTP, *HyperText Transfer Protocol*) es el usado en cada transacción de la *Web* (WWW) HTTP fue desarrollado por el consorcio W3C y la IETF, colaboración que culminó en 1999 con la publicación de una serie de RFC, siendo el más importante de ellos el RFC 2 616, que especifica la versión 1.1.

HTTP define la sintaxis y la semántica que utilizan los elementos *software* de la arquitectura *Web* (clientes, servidores, *proxies*) para comunicarse, es un protocolo orientado a transacciones y sigue el esquema petición-respuesta entre un cliente y un servidor, donde el cliente que efectúa la petición (un navegador o un *spider*) se le conoce como *user agent* (agente del usuario) y a la información transmitida se le llama recurso y se identifica mediante un URL.

Los recursos pueden ser archivos, el resultado de la ejecución de un programa, una consulta a una base de datos, la traducción automática de un documento, etcétera.

HTTP es un protocolo sin estado, es decir, que no guarda ninguna información sobre conexiones anteriores. El desarrollo de aplicaciones *Web* necesita frecuentemente mantener estado y para esto se usan las *cookies*, que es información que un servidor puede almacenar en el sistema cliente permitiendo a las aplicaciones *Web* instituir la noción de sesión y también rastrear usuarios ya que las *cookies* pueden guardarse en el cliente por tiempo indeterminado.

2.2.2. *Hypertext Transfer Protocol Secure*

(En español: Protocolo seguro de transferencia de hipertexto), más conocido por sus siglas HTTPS, es un protocolo de red basado en el protocolo HTTP, destinado a la transferencia segura de datos de hipertexto, es decir, es la versión segura de HTTP.

El sistema HTTPS utiliza un cifrado basado en las *Secure Socket Layers* (SSL) para crear un canal cifrado (cuyo nivel depende del servidor remoto y del navegador utilizado por el cliente) más apropiado para el tráfico de información sensible que el protocolo HTTP. De este modo se consigue que la información sensible (usuario y claves de paso normalmente) no puede ser usada por un atacante que haya conseguido interceptar la transferencia de datos de la conexión, ya que lo único que obtendrá será un flujo de datos cifrados que le resultará imposible de descifrar. Cabe mencionar que el uso del protocolo HTTPS no impide que se pueda utilizar HTTP.

Es aquí, cuando el navegador advertirá sobre la carga de elementos no seguros (HTTP), estando conectados a un entorno seguro (HTTPS).

Los protocolos HTTPS son utilizados por navegadores como: Safari, Internet Explorer, Mozilla Firefox, Opera y Google Chrome, entre otros, siendo utilizados principalmente por entidades bancarias, tiendas en línea y cualquier tipo de servicio que requiera el envío de datos personales o contraseñas, el puerto estándar para este protocolo es el 443 y para conocer si una página *Web* que se está visitando, utiliza el protocolo HTTPS y es, por tanto, segura en cuanto a la trasmisión de los datos que se están transcribiendo, se debe observar si en la barra de direcciones del navegador, aparece https al inicio, en lugar de http, o sino hacer un *scan*.

No obstante, por una vulnerabilidad en Internet Explorer, no siempre es posible identificar un sitio seguro y tampoco es posible estar seguros de que el sitio que se está visitando sea realmente aquel al que se quería ir al escribir la dirección, según los siguientes enlaces:

Demostración en: http://lcamtuf.coredump.cx/ietrap2/ (Inglés) La fuente: http://secunia.com/advisories/25564/ (Inglés).

Algunos navegadores utilizan un ícono (generalmente un candado) en la parte derecha de la barra de direcciones para indicar la existencia de un protocolo de comunicaciones seguro e incluso cambian el color del fondo de la barra de direcciones por amarillo (Firefox) o verde (Internet Explorer) para identificar páginas *Web* seguras.

2.3. SOAP

Son las siglas de *Simple Object Access Protocol*, este protocolo es un derivado de otro creado por David Winer, XML-RPC en 1998. Con este protocolo se podían realizar RPC o *remote procedure calls*, es decir, se podía bien en cliente o servidor realizar peticiones mediante HTTP a un servidor *Web*.

Los mensajes debían tener un formato determinado empleando XML para encapsular los parámetros de la petición. Con el paso del tiempo el proyecto interesó a importantes multinacionales entre las que se encuentran IBM y Microsoft y de este interés por XML-RPC se desarrollo SOAP.

En el núcleo de los servicios *Web* se encuentra el protocolo simple de acceso a datos SOAP, que proporciona un mecanismo estándar para empaquetar mensajes. SOAP ha recibido gran atención debido a que facilita una comunicación del estilo RPC entre un cliente y un servidor remoto. Pero existen multitud de protocolos creados para facilitar la comunicación entre aplicaciones, incluyendo RPC de Sum, DCE de Microsoft, RMI de Java y ORPC de CORBA.

¿Por qué se presta tanta atención a SOAP? una de las razones principales es que SOAP ha recibido un increíble apoyo por parte de la industria. SOAP es el primer protocolo de su tipo que ha sido aceptado prácticamente por todas las grandes compañías de *software* del mundo, hay compañías que en raras ocasiones cooperan entre sí y están ofreciendo su apoyo a este protocolo. Algunas de las mayores compañías que soportan SOAP son Microsoft, IBM, SUN, Microsystems, SAP y Ariba.

Algunas de las ventajas de SOAP son:

- No está asociado con ningún lenguaje: los desarrolladores involucrados en nuevos proyectos pueden elegir desarrollar con el último y mejor lenguaje de programación que exista pero los desarrolladores responsables de mantener antiguas aflicciones heredadas podrían no poder hacer esta elección sobre el lenguaje de programación que utilizan.

  SOAP no específica una API, por lo que la implementación de la API se deja al lenguaje de programación, como en Java, y la plataforma como Microsoft .Net.

- No se encuentra fuertemente asociado a ningún protocolo de transporte: la especificación de SOAP no describe como se deberían asociar los mensajes de SOAP con HTTP. Un mensaje de SOAP no es más que un documento XML, por lo que puede transportarse utilizando cualquier protocolo capaz de transmitir texto.

- No está atado a ninguna infraestructura de objeto distribuido: la mayoría de los sistemas de objetos distribuidos se pueden extender y ya lo están, algunos de ellos para que admitan SOAP.

- Aprovecha los estándares existentes en la industria: los principales contribuyentes a la especificación SOAP evitaron, intencionadamente, reinventar las cosas optando por extender los estándares existentes para que coincidieran con sus necesidades.

Por ejemplo, SOAP aprovecha XML para la codificación de los mensajes, en lugar de utilizar su propio sistema de tipo que ya están definidas en la especificación esquema de XML. Y como ya se ha mencionado SOAP no define un medio de trasporte de los mensajes; los mensajes de SOAP se pueden asociar a los protocolos de transporte existentes como HTTP y SMTP.

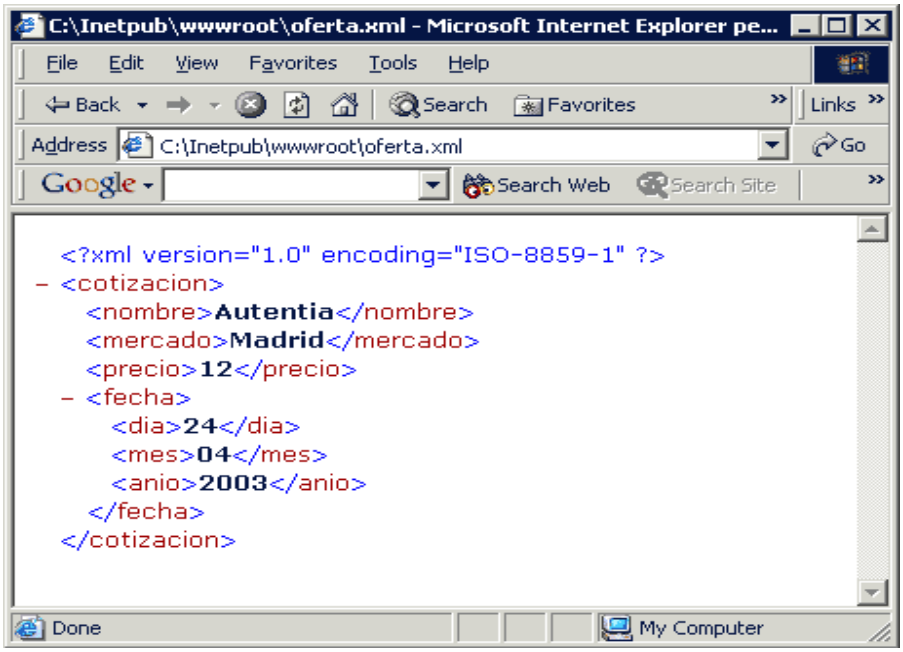
- Permite la interoperabilidad entre múltiples entornos: SOAP se desarrolló sobre los estándares existentes de la industria, por lo que las aplicaciones que se ejecuten en plataformas con dichos estándares pueden comunicarse mediante mensaje SOAP con aplicaciones que se ejecuten en otras plataformas. Por ejemplo, una aplicación de escritorio que se ejecute en una PC puede comunicarse con una aplicación del *back-end* ejecutándose en una computadora capaz de enviar y recibir XML sobre HTTP.

2.4. XML

(Extensible Markup Language) XML, es el estándar de *Extensible Markup Language*. XML no es más que un conjunto de reglas para definir etiquetas semánticas que organizan un documento en diferentes partes. XML es un metalenguaje que define la sintaxis utilizada para definir otros lenguajes de etiquetas estructurados. En primer lugar para entenderlo bien hay que olvidarse un poco, solo un poco de HTML. En teoría HTML es un subconjunto de XML especializado en presentación de documentos para la *Web*, mientras que XML es un subconjunto de SGML especializado en la gestión de información para la *Web*.

En la práctica XML contiene a HTML aunque no en su totalidad. La definición de HTML contenido totalmente dentro de XML y por lo tanto que cumple con la especificación SGML es XHTML (*Extensible, Hypertext Markup Language)*. Desde su creación, XML ha despertado encontradas pasiones, y como para cualquier tema en Internet, hay gente que desde el principio se deja iluminar por sus expectativas, mientras otras muchas lo han ignorado.

Figura 2**.** **Código XML**



Fuente: Internet Explorer 6.

2.4.1. Historia de XML

XML fue creado al amparo del *Word Wide Web* Consortium (W3C) organismo que vela por el desarrollo de WWW partiendo de las amplias especificaciones de SGML. Su desarrollo se comenzó en 1996 y la primera versión salió a la luz el 10 de febrero de 1998.

La primera definición que apareció fue: sistema para definir validar y compartir formatos de documentos en la *Web*. Durante el año 1998 XML tuvo un crecimiento exponencial, y con ello se refiere a sus apariciones en medios de comunicación, menciones en páginas *Web*, soporte *software*, etcétera.

Respecto a sus objetivos son:

- XML debe ser directamente utilizable sobre Internet
- XML debe soportar una amplia variedad de aplicaciones
- XML debe ser compatible con SGML
- Debe ser fácil la escritura de programas que procesen documentos XML
- El número de características opcionales en XML debe ser absolutamente mínimo, idealmente cero.
- Los documentos XML deben ser legibles por humanos y razonablemente claros.
- El diseño de XML debe ser preparado rápidamente
- El diseño de XML debe ser formal y conciso
- Los documentos XML deben ser fácilmente creables
- La concisión en las marcas XML es de mínima importancia

- Esta especificación, junto con los estándares asociados (Unicode e ISO/IEC 10646 para caracteres, Internet RFC 1766 para identificación de lenguajes, ISO 639 para códigos de nombres de lenguajes, e ISO 3166 para códigos de nombres de países), proporciona toda la información necesaria para entender la Versión 1.0 de XML y construir programas de computador que los procesen.

Principales características:

- Es una arquitectura más abierta y extensible. No se necesitan versiones para que puedan funcionar en futuros navegadores. Los identificadores pueden crearse de manera simple y ser adaptados en el acto en Internet/Intranet por medio de un validador de documentos (*parser*).

- Mayor consistencia, homogeneidad y amplitud de los identificadores descriptivos del documento con XML (los RDF *Resource Description FrameWork*), en comparación a los atributos de la etiqueta del HTML.

- Integración de los datos de las fuentes más dispares. Se podrá hacer el intercambio de documentos entre las aplicaciones tanto en el propio PC como en una red local o extensa.

- Datos compuestos de múltiples aplicaciones. La extensibilidad y flexibilidad de este lenguaje permitirá agrupar una variedad amplia de aplicaciones, desde páginas Web hasta bases de datos.

- Gestión y manipulación de los datos desde el propio cliente Web

- Los motores de búsqueda devolverán respuestas más adecuadas y precisas, ya que la codificación del contenido Web en XML consigue que la estructura de la información resulte más accesible.

- Se desarrollarán de manera extensible las búsquedas personalizables y subjetivas para robots y agentes inteligentes. También conllevará que los clientes Web puedan ser más autónomos para desarrollar tareas que actualmente se ejecutan en el servidor.

- Se permitirá un comportamiento más estable y actualizable de las aplicaciones *Web*, incluyendo enlaces bidireccionales y almacenados de forma externa (El famoso epígrafe 404 *file not found* desaparecerá).

- El concepto de hipertexto se desarrollará ampliamente (permitirá denominación independiente de la ubicación, enlaces bidireccionales, enlaces que pueden especificarse y gestionarse desde fuera del documento, hiperenlaces múltiples, enlaces agrupados, atributos para los enlaces, etcétera. Creado a través del Lenguaje de enlaces extensible (XLL).

- Exportabilidad a otros formatos de publicación (papel, *Web*, CD-rom, etcétera). El documento maestro de la edición electrónica podría ser un documento XML que se integraría en el formato deseado de manera directa.

2.4.2. Estructura del XML

El metalenguaje XML consta de cuatro especificaciones (el propio XML sienta las bases sintácticas y el alcance de su implementación):

- DTD (*Document Type Definition*) definición del tipo de documento. Es, en general, un archivo/s que encierra una definición formal de un tipo de documento y, a la vez, especifica la estructura lógica de cada documento. Define tanto los elementos de una página como sus atributos. El DTD del XML es opcional. En tareas sencillas no es necesario construir una DTD, entonces se trataría de un documento bien formado y si lleva DTD será un documento validado.

- XSL (eXtensible Stylesheet *Language*): define o implementa el lenguaje de estilo de los documentos escritos para XML. Desde el verano de 1997 varias empresas informáticas como Arbortext, Microsoft e Inso vienen trabajando en una propuesta de XSL (antes llamado "xml-style") que presentaron a W3C. Permite modificar el aspecto de un documento. Se pueden lograr múltiples columnas, texto girado, orden de visualización de los datos de una tabla, múltiples tipos de letra con amplia variedad en los tamaños.

  Este estándar está basado en el lenguaje de semántica y especificación de estilo de documento (DSSSL, *Document Style Semantics and Specification Language*, ISO/IEC 10179) y, por otro lado, se considera más potente que las hojas de estilo en cascada (CSS, *Cascading Style Sheets*), usado en un principio con el lenguaje DHTML.

Se espera que el CSS sea usado para visualizar simples estructuras de documentos XML (actualmente se ha conseguido mayor integración en XML con el protocolo CSS2 (*Cascading Style Sheets, level* 2) ofreciendo nuevas formas de composición y una más rápida visualización) y, por otra parte, XSL pueda ser utilizado donde se requiera más potencia de diseño como documentos XML que encierran datos estructurados (tablas, organigramas, etcétera).

- XLL (eXtensible *Linking Language*): define el modo de enlace entre diferentes enlaces. Se considera que es un subconjunto de HyTime (Hipermedia/*Timed-based structuring Language* o Lenguaje de estructuración hipermedia/basado en el tiempo, ISO 10744) y sigue algunas especificaciones del TEI (*Text Encoding Initiative* o Iniciativa de codificación de texto). Desde marzo de 1998 el W3C trabajo en los enlaces y direccionamientos del XML.

  Provisionalmente se le renombró como Xlink y a partir de junio se le denomina XLL. Este lenguaje de enlaces extensible tiene dos importantes componentes: Xlink y el Xpointer. Va más allá de los enlaces simples que solo soporta el HTML. Se podrá implementar con enlaces extendidos. Jon Bosak establece los siguientes mecanismos hipertextuales que soportará esta especificación:

  - Denominación independiente de la ubicación

  - Enlaces que pueden ser también bidireccionales

- Enlaces que pueden especificarse y gestionarse desde fuera del documento a los que se apliquen. (Esto permitirá crear en un entorno intranet/extranet un banco de datos de enlaces en los que se puede gestionar y actualizar automáticamente.

- Hiperenlaces múltiples, (anillos, múltiples ventanas, etcétera)

- Enlaces agrupados, (múltiples orígenes)

- Transclusión, (el documento destino al que apunta el enlace aparece como parte integrante del documento origen del enlace)

- Se pueden aplicar atributos a los enlaces, (tipos de enlaces)

- XUA (XML User Agent), estandarización de navegadores XML. Todavía está en proceso de creación de borradores de trabajo. Se aplicará a los navegadores para que compartan todas las especificaciones XML.

2.4.3. XML y los servicios *Web*

Finalmente, ya que se conoce algo más sobre XML queda responder ¿Por qué XML es utilizado en los servicios *Web*?:

- Es un estándar abierto, es decir, que es reconocido mundialmente ya que muchas compañías tecnológicas integran en su *software* compatibilidad con dicho lenguaje.

Esto quiere decir, que la gran mayoría de *software* de escritorio de Sistema Operativo y aplicaciones móviles permiten la compatibilidad con XML esto lo hace muy potente a la hora que permite la comunicación entre distintas plataformas de *software* y *hardware* (y si bien este es el sentido final de los servicios *Web*).

- Simplicidad de sintaxis, esto quiere decir, que es muy fácil de escribir código en XML y la representación de los datos es casi entendible por cualquier ser humano. Esto lo hace muy flexible a la hora de querer representar datos de cualquier especie, bastará con contar con cualquier editor de texto y aprender unas cuantas instrucciones básicas y ya se estará en condiciones de escribir código XML, el cual será soportado o entendido por cualquier aplicación que pueda leer documentos XML. El hecho de que XML sea tan fácil de codificar y de entender lo hace el lenguaje ideal para utilizarlo en los servicios *Web*.

- Independencia del protocolo de transporte, el hecho de que XML es un lenguaje de marcado de texto, no necesita de ningún protocolo de trasporte especial, solo necesita de un protocolo que pueda trasferir texto o documentos simples. Esto trae a la memoria que en el mercado existen muchos protocolos con estas características, entre los más conocidos están HTTP y SMTP por nombrar algunos.

Volviendo al tema de los servicios *Web* una de las características de estos es la independencia del protocolo de transporte.

2.5. Mecanismos de transmisión y recepción

2.5.1. *Web Services*

Son componentes de *software* autónomos y modulares que encapsulan la lógica de funciones, brindando así un servicio. Están disponibles a través de Internet y pueden ser accedidos programáticamente a través de protocolos de *Web* estándares. Los *Web Services* satisfacen una tarea específica o un conjunto de las tareas interactuando con las aplicaciones y pudiendo trabajar con muchos otros servicios de la *Web* de una manera interoperable para realizar su parte dentro de un flujo complejo de trabajo o una transacción del negocio.

Están conectados en forma débil a las aplicaciones, lo cual les da independencia de ellas ya que no están formando un solo ejecutable. Un servicio debe ser una aplicación completamente autónoma e independiente. A pesar de esto, no es una isla, porque expone una interfaz de llamado, basada en mensajes, capaz de ser accedida a través de la red.

Generalmente, los servicios incluyen tanto lógica de negocios como manejo de estado (datos), relevantes a la solución del problema para el cual fueron diseñados. La manipulación del estado es gobernada por las reglas de negocio.

2.5.2. *Remoting*

Permite crear fácilmente aplicaciones ampliamente distribuidas, tanto si los componentes de las aplicaciones están todos en un equipo como si están repartidos por el mundo. Se pueden crear aplicaciones de cliente que utilicen objetos en otros procesos del mismo equipo o en cualquier otro equipo disponible en la red. También, se puede utilizar *Remoting* para comunicarse con otros dominios de aplicación en el mismo proceso. *Remoting* permite un enfoque abstracto en la comunicación entre procesos que separa el objeto utilizado de forma remota de un dominio de aplicación de cliente o servidor y de un mecanismo específico de comunicación. Como resultado, se trata de un sistema flexible y fácilmente personalizable.

Se puede reemplazar un protocolo de comunicación con otro o un formato de serialización con otro, sin tener que recompilar el cliente ni el servidor. Además, el sistema de interacción remota no presupone ningún modelo de aplicación en particular. Se puede comunicar desde una aplicación *Web*, una aplicación de consola, un servicio de Windows, desde casi cualquier aplicación que se desee utilizar. Los servidores de interacción remota también pueden ser cualquier tipo de dominio de aplicación. Cualquier aplicación puede albergar objetos de interacción remota y proporcionar sus servicios a cualquier cliente en su equipo o red.

2.5.3. FTP (*File Transfer Protocol*)

Existe en la actualidad, dentro de lo que es Internet, un servicio que permite trabajar con archivos (copiar, modificar, borrar) desde una PC hacia un servidor remoto. En dichos servidores remotos se alojan grandes cantidades de *shareware* y *freeware*, que están a disposición del público para que haga un *download* a su computadora.

Generalmente, estos servidores permiten el acceso a cualquier usuario (servidores llamados *anonymous*) pero también existen los servidores que tienen acceso restringido por medio de *passwords.* Estas transferencias de archivos se hacen por medio de un *software* conocido como FTP (del inglés, *File Transfer Protocol*). Existen hoy en día muchos programas de este tipo, con diferentes prestaciones.

3. IDENTIFICACIÓN DE TECNOLOGÍAS

3.1. Historia de Spring

El *Spring Framework* (también conocido simplemente como Spring) es un *framework* de código abierto de desarrollo de aplicaciones para la plataforma Java. La primera versión fue escrita por Rod Jonhson, quien lo lanzó primero con la publicación de su libro Expert One-on-One Java EE Design *and Developmen*t (Wrox Press, octubre 2002). También hay una versión para la plataforma .NET, Spring.Net. El *framework* fue lanzado inicialmente bajo Apache 2.0 *License* en junio de 2003. El primer gran lanzamiento fue la versión 1.0, que apareció en marzo de 2004 y fue seguida por otros en septiembre de 2004 y marzo de 2005.

A pesar de que *Spring Framework* no obliga a usar un modelo de programación en particular, se ha popularizado en la comunidad de programadores en Java al considerársele una alternativa y sustituta del modelo de Enterprise JavaBean. Por su diseño el *framework* ofrece mucha libertad a los desarrolladores en Java y soluciones muy bien documentadas y fáciles de usar para las prácticas comunes en la industria.

Mientras que las características fundamentales de este *framework* pueden emplearse en cualquier aplicación hecha en Java, existen muchas extensiones y mejoras para construir aplicaciones basadas en *Web* por encima de la plataforma empresarial de Java (*Java Enterprise Platform*). A partir de 2009 las actualizaciones del producto (en su forma binaria) estarán disponibles únicamente para la última versión publicada del *Framework*.

Para acceder a las actualizaciones en forma binaria para versiones anteriores habrá que pagar una subscripción. Sin embargo, estas actualizaciones estarán disponibles libremente (y gratuitamente) en forma de código fuente en los repositorios públicos del proyecto.

Los primeros componentes de lo que se ha convertido en *Spring Framework* fueron escritos por Rod Johnson en el año 2000, mientras trabajaba como consultor independiente para sus clientes en la industria financiera en Londres. Mientras escribía el libro Expert One-on-one J2EE *Design And Development* (*Programmer to programmer*), Rod amplió su código para sintetizar su visión acerca de cómo las aplicaciones que trabajan con varias partes de la plataforma J2EE podían llegar a ser más simples y más consistentes que aquellas que los desarrolladores y compañías estaban usando en aquel entonces.

En el año 2001 los modelos dominantes de programación para aplicaciones basadas en *Web* eran ofrecidas por el API Java Servlet y los Enterprise JavaBeans, ambas especificaciones creadas por Sun Microsystems en colaboración con otros distribuidores y partes interesadas que disfrutaban de gran popularidad en la comunidad Java. Las aplicaciones que no eran basadas en *Web*, como las aplicaciones basadas en cliente o aplicaciones en batch, podían ser escritas con base en herramientas y proyectos de código abierto o comercial que proveerán las características requeridas para aquellos desarrollos.

Rod Johnson es reconocido por crear un *framework* que está basado en las mejores prácticas aceptadas, y ello las hizo disponibles para todo tipo de aplicaciones, no solo aquellas basadas en *Web*. Estas ideas también estaban plasmadas en su libro y, tras la publicación, sus lectores le solicitaron que el código que acompañaba al libro fuera liberado bajo una licencia *open source*.

Se formó un pequeño equipo de desarrolladores que esperaban trabajar en extender el *framework* y un proyecto fue creado en Sourceforge en febrero de 2003. Después de trabajar en su desarrollo durante más de un año lanzaron una primera versión (1.0) en marzo de 2004. Después de este lanzamiento Spring ganó mucha popularidad en la comunidad Java, debido en parte al uso de Javadoc y de una documentación de referencia por encima del promedio de un proyecto de código abierto.

Sin embargo, *Spring Framework* también fue duramente criticado en 2004 y sigue siendo el tema de acalorados debates. Al tiempo en que se daba su primer gran lanzamiento muchos desarrolladores y líderes de opinión vieron a Spring como un gran paso con respecto al modelo de programación tradicional; esto era especialmente cierto con respecto a Enterprise JavaBeans. Una de las metas de diseño de *Spring Framework* es su facilidad de integración con los estándares J2EE y herramientas comerciales existentes. Esto quita en parte la necesidad de definir sus características en un documento de especificación elaborado por un comité oficial y que podría ser criticado.

Spring Framework hizo que aquellas técnicas que resultaban desconocidas para la mayoría de programadores se volvieran populares en un período muy corto de tiempo.

El ejemplo más notable es la inversión de control. En el año 2004, Spring disfrutó de unas altísimas tasas de adopción y al ofrecer su propio *framework* de programación orientada a aspectos (*aspect-oriented programming,* AOP) consiguió hacer más popular su paradigma de programación en la comunidad Java.

En 2005 Spring superó las tasas de adopción del año anterior como resultado de nuevos lanzamientos y más características, fueron añadidas. El foro de la comunidad formada alrededor de *Spring Framework* (*The Spring Forum)* que arrancó a finales de 2004 también ayudó a incrementar la popularidad del *framework* y desde entonces ha crecido hasta llegar a ser la más importante fuente de información y ayuda para sus usuarios.

En el mismo año los desarrolladores del proyecto abrieron su propia compañía para ofrecer soporte comercial y establecieron una alianza con BEA. En diciembre de 2005 la primera conferencia de Spring fue realizada en Miami y reunió a 300 desarrolladores en el transcurso de tres días, seguida por una conferencia en Amberes en junio de 2006, donde se concentraron más de 400 personas.

3.2. ¿Qué es Spring.Net?

Spring.Net es un *framework Web* para .Net basado en el *framework* Spring de Java. Iniciando una serie de artículos sobre herramientas de apoyo en la plataforma de programación .Net, se presentará el *framework* de aplicaciones Spring.Net. Iniciado como una variante del proyecto *Spring Framework* para Java Enterprise Edition, ha intentado llevar a otra plataforma las técnicas de programación ya arraigadas en los últimos años entre los desarrolladores de aplicaciones escritas en Java.

Spring.Net es un *framework* de código abierto, (liberado bajo la Apache *License* 2.0) para el desarrollo de aplicaciones que facilita la construcción de aplicaciones empresariales en .NET.

Proveyendo componentes basados en patrones de diseño ya probados y que pueden ser integrados en todas las capas de la arquitectura de una aplicación, Spring.Net ayuda a incrementar la productividad y mejora la calidad y el desempeño de aplicaciones. Spring.Net surge como una variante del proyecto *Spring Framework,* orientado para aplicaciones Java/J2EE.

Spring Framework hizo que aquellas técnicas que resultaban desconocidas para la mayoría de programadores se volvieran populares en un período muy corto de tiempo, el ejemplo más notable es la inyección de dependencia. Spring disfruta de altísimas tasas de adopción y al ofrecer su propio *framework* de programación orientada a aspectos (*aspect-oriented programming,* AOP) ha conseguido hacer más popular su paradigma de programación en la comunidad Java.

3.2.1. Conceptos previos

3.2.1.1. Inyección de dependencia *(Dependency Injection)*

Dependency Injection es una de las implementaciones del concepto inversión de control (*Inversion of Control*). Desde la perspectiva de la inversión de control se reconoce que los objetos involucrados en la lógica de negocio de una aplicación dependen de otros objetos de negocio, objetos de acceso a datos y recursos compartidos. Todos ellos, se ejecutan en un ambiente denominado Contenedor Ligero (*Lightweight Container*), el cual es responsable de establecer las dependencias entre los objetos.

Según Rod Jonson en su libro Expert one-on-one J2EE development without EJB, el patrón *Dependency Injection* consigue plasmar los planteamientos de la inversión de control en el desarrollo de aplicaciones. Para ello, establece que todos los objetos de negocio deben proveer métodos públicos que permitan que el Contenedor determine las dependencias entre los objetos y los vincule a través de constructores y propiedades, los cuales reciben como argumentos los recursos que necesita el objeto de negocio. El primer método es denominado *Constructor injection* y el segundo *Setter injection*.

Aunque fue iniciado y popularizado por la comunidad Java, este patrón no está restringido a un solo lenguaje de programación. De hecho, se han realizado implementaciones posteriores del mismo en otras plataformas de programación, como .NET.

Este patrón tiene las siguientes ventajas:

- La búsqueda de recursos es removida del código de la aplicación
- No hay dependencia con respecto a una interfaz de programación de aplicaciones provista por un contenedor específico.
- No requiere interfaces especiales

A pesar de lo expuesto anteriormente, hay situaciones en donde la incorporación de un contenedor que haga uso de *Dependency Injection* puede resultar problemático y no puede resolver totalmente el problema. Esto ocurre cuando los objetos del negocio se vuelven dependientes del contenedor y se requiere una lógica condicional para configurar el contenedor, o los objetos del negocio necesitan acceder a los servicios del contenedor en tiempo de ejecución.

3.2.1.2. Programación orientada a aspectos

La programación orientada a aspectos (POA) es un paradigma de programación relativamente reciente cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de conceptos. Gracias a la POA se pueden capturar los diferentes conceptos que componen una aplicación en entidades bien definidas, de manera apropiada en cada uno de los casos y eliminando las dependencias inherentes entre cada uno de los módulos. De esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reusables.

Por aspectos se entienden dichos problemas que afectan a la aplicación de manera horizontal y que este paradigma persigue el poder tenerlos de manera aislada de forma adecuada y comprensible, dando la posibilidad de construir el sistema componiéndolo junto con el resto de los componentes.

Su consecución implicaría las siguientes ventajas:

- Un código menos enmarañado, más natural y más reducido
- Mayor facilidad para razonar sobre los conceptos, ya que están separados y las dependencias entre ellos son mínimas.
- Un código más fácil de depurar y más fácil de mantener
- Se consigue que un conjunto grande de modificaciones en la definición de una materia tenga un impacto mínimo en las otras.
- Se tiene un código más reusable y que se puede acoplar y desacoplar cuando sea necesario.

3.2.2. Ventajas

- Es un *framework* de desarrollo de aplicaciones, no se orienta únicamente a una parte de una aplicación.
- Aunque influye sobre todas las capas de una aplicación, no es un *framework* invasivo y puede ser fácilmente reemplazado
- Es una extensión de la plataforma .NET, a la cual añade funcionalidades

- Reduce drásticamente la cantidad de código escrito

- Incorpora técnicas ya utilizadas en otras plataformas de desarrollo de aplicaciones empresariales.

- Permite la integración con otros *frameworks open source*, como NHibernate, IBatis.NET.

3.2.3. Desventajas

- Escasa o nula integración con IDES de desarrollo, como Visual Studio

- A diferencia de su contraparte, Spring para Java, no ha logrado una aceptación importante de la industria del *software*.

- Nuevamente, en comparación con Spring, no cuenta con una comunidad numerosa y activa y, por tanto, sus características están detrás de las alcanzadas por Spring.

- Su curva de aprendizaje es muy escarpada, debido a los nuevos conceptos de programación que implementa.

3.3. ¿Qué es NHibernate?

Hibernate es una herramienta de Mapeo objeto-relacional para la plataforma Java (y disponible también para .Net con el nombre de NHibernate) que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) que permiten establecer estas relaciones. NHibernate es la conversión de Hibernate de lenguaje Java a C# para su integración en la plataforma .NET. Al igual que muchas otras herramientas libres para esta plataforma, NHibernate también funciona en Mono.

Al usar NHibernate para el acceso a datos el desarrollador se asegura de que su aplicación es agnóstica en cuanto al motor de base de datos a utilizar en producción, pues NHibernate soporta los más habituales en el mercado: MySQL, PostgreSQL, Oracle, MS SQL Server, etcétera. Solo se necesita cambiar una línea en el fichero de configuración para que se pueda utilizar una base de datos distinta.

NHibernate es *software* libre, distribuido bajo los términos de la LGPL (Licencia Pública General Menor de GNU).

3.3.1. Características

Como todas las herramientas de su tipo, Hibernate busca solucionar el problema de la diferencia entre los dos modelos de datos coexistentes en una aplicación: el usado en la memoria de la computadora (orientación a objetos) y el usado en las bases de datos (modelo relacional).

Para lograr esto permite al desarrollador detallar cómo es su modelo de datos, qué relaciones existen y qué forma tienen. Con esta información Hibernate le permite a la aplicación manipular los datos de la base operando sobre objetos, con todas las características de la POO.

Hibernate convertirá los datos entre los tipos utilizados por Java y los definidos por SQL. Hibernate genera las sentencias SQL y libera al desarrollador del manejo manual de los datos que resultan de la ejecución de dichas sentencias, manteniendo la portabilidad entre todos los motores de bases de datos con un ligero incremento en el tiempo de ejecución. Hibernate está diseñado para ser flexible en cuanto al esquema de tablas utilizado, para poder adaptarse a su uso sobre una base de datos ya existente. También tiene la funcionalidad de crear la base de datos a partir de la información disponible.

Hibernate ofrece también un lenguaje de consulta de datos llamado HQL (*Hibernate Query Language*), al mismo tiempo que una API para construir las consultas programáticamente (conocida como criteria). Hibernate para Java puede ser utilizado en aplicaciones Java independientes o en aplicaciones Java EE, mediante el componente *Hibernate Annotations* que implementa el estándar JPA, que es parte de esta plataforma.

3.3.2. Historia

Hibernate fue una iniciativa de un grupo de desarrolladores dispersos alrededor del mundo conducidos por Gavin King. Tiempo después, JBoss Inc. (empresa comprada por Red Hat) contrató a los principales desarrolladores de Hibernate y trabajó con ellos en brindar soporte al proyecto.

La rama actual de desarrollo de Hibernate es la 3.x, la cual incorpora nuevas características, como una nueva arquitectura Interceptor/Callback, filtros definidos por el usuario, y opcionalmente el uso de anotaciones para definir la correspondencia en lugar (o conjuntamente con) los archivos XML. Hibernate 3 también guarda cercanía con la especificación EJB 3.0 (aunque apareciera antes de la publicación de dicha especificación por Java Community Process) y actúa como la espina dorsal de la implementación de EJB 3.0 en JBoss.

NHibernate es un *framework* de O/RM (*Object/Relational Mapping*), un *port* de Hibernate de Java, que tiene como función principal mapear los objetos desde una aplicación .Net a una base de datos Relacional. Primero, es necesario hablar de como NH (NHibernate) realiza su funcionamiento básico.

- Bases de datos soportadas

  Están soportadas en una lista bastante amplia entre las que se encuentran las bases de datos más utilizadas a nivel de mercado, tales como Oracle, SQL Server, MySQL, etcétera.

- Relaciones

  Es factible modelar relaciones de 1:1 o 1:N, asociando en las clases propiedades que contienen una lista de objetos (filas) de otro tipo de entidad (tabla) definida. Es importante definir correctamente, como NHibernate debe cargar la entidad cuando tiene relaciones de 1:N, dado que esto puede significar problemas graves de performance. Lo ideal es configurarlo para que solo se carguen las listas cuando realmente se necesitan. Verificar el atributo Lazy.

- Performance

  Dependiendo del modelo claro está, pero en general se ha conseguido muy buena performance en los accesos a base de datos, probando especialmente con Oracle, SQL Server 2000/2005, MySQL, Sybase y Postgress. En caso de necesitar hacer alguna consulta muy complicada a la base de datos, NHibernate permite generarla a través de código SQL directo, evitando tiempos adicionales innecesarios.

3.4. Componentes de Spring.Net

La estructura arquitectónica de Spring.Net difiere de su predecesor, Spring para Java, y se concentra en ofrecer funcionalidades agrupadas en seis librerías:

- Spring.Core: es la parte fundamental del *framework*, y provee la funcionalidad de la inyección de dependencia. Muchas de las librerías de Spring.Net dependen de este módulo y extienden sus funcionalidades.

- Spring.Aop: provee soporte de programación orientada a aspectos y a objetos de negocio. Esta librería complementa el contenedor de inyección de dependencia de la librería Spring.Core para proveer de una base sólida para la construcción de aplicaciones empresariales y aplicar servicios a objetos de negocio.

- Spring.Web: extiende a ASP.NET añadiendo una variedad de características como la inyección de dependencia para páginas ASPX, enlace de datos bidireccional, páginas principales para ASP.NET 1.1 y un mejorado soporte de localización (generación de contenido en base a la ubicación geográfica del usuario).

- Spring.Services: deja exponer cualquier objeto normal (que no herede de una clase de servicio base) como si se tratase de un servicio empresarial (COM+) u objeto remoto. Los servicios *Web* de .NET consiguen una flexibilidad adicional al momento de configurarse con soporte para inyección de dependencia y sobreescritura de metadata de sus atributos. También se provee integración con Windows Service.

- Spring.Data: provee una capa de abstracción de acceso a datos que puede ser usada con una variedad de proveedores de acceso a datos, desde ADO.NET a otros proveedores de mapeo de objetos a partir de bases de datos relacionales. También contiene una capa de abstracción que elimina la necesidad de escribir código y declarar el manejo de transacciones para ADO.NET.

- Spring.ORM: provee capas de integración con librerías populares de mapeo relacional de objetos (Object-Relational Mapping, ORM), como NHibernate e IBatis for .NET. Esto provee funcionalidades como el soporte para la gestión declarativa de transacciones.

3.5. Componentes de NHibernate

La persistencia permite al programador almacenar, transferir y recuperar el estado de los objetos. Para esto existen varias técnicas:

- Serialización

  En ciencias de la computación, la serialización (o *marshalling* en inglés) consiste en un proceso de codificación de un objeto (programación orientada a objetos) en un medio de almacenamiento (como puede ser un archivo, o un *buffer* de memoria) con el fin de transmitirlo a través de una conexión en red como una serie de *bytes* o en un formato humanamente más legible como XML.

  La serie de *bytes* o el formato pueden ser usados para crear un nuevo objeto que es idéntico en todo al original, incluido su estado interno (por tanto, el nuevo objeto es un clon del original). La serialización es un mecanismo ampliamente usado para transportar objetos a través de una red, para hacer persistente un objeto en un archivo o base de datos, o para distribuir objetos idénticos a varias aplicaciones o localizaciones.

- Motor de persistencia

  En la actualidad existen distintos motores de persistencia, estos facilitan el mapeo objeto-relacional de atributos entre una base de datos relacional tradicional y suplen la funcionalidad de una base de datos orientada a objetos.

Estos motores buscan solucionar el problema de la diferencia entre los dos modelos usados hoy en día para organizar y manipular datos: el usado en la memoria de la computadora (orientación a objetos) y el usado en las bases de datos (modelo relacional).

Para lograrlo, estos motores permiten al desarrollador detallar, ¿cómo es su modelo de datos?, ¿qué relaciones existen? y ¿qué forma tienen?

- Base de datos orientada a objetos

En una base de datos orientada a objetos, la información se representa mediante objetos como los presentes en la programación orientada a objetos. Cuando se integran las características de una base de datos con las de un lenguaje de programación orientado a objetos, el resultado es un sistema gestor de base de datos orientada a objetos (ODBMS, *object database management system*).

Un ODBMS hace que los objetos de la base de datos aparezcan como objetos de un lenguaje de programación en uno o más lenguajes de programación a los que dé soporte. Un ODBMS extiende los lenguajes con datos persistentes de forma transparente, control de concurrencia, recuperación de datos, consultas asociativas y otras capacidades.

Las bases de datos orientadas a objetos se diseñan para trabajar bien en conjunción con lenguajes de programación orientados a objetos como Java, C#, Visual Basic.NET y C++. Los ODBMS usan exactamente el mismo modelo que estos lenguajes de programación.

Los ODBMS son una buena elección para aquellos sistemas que necesitan un buen rendimiento en la manipulación de tipos de dato complejos. Los ODBMS proporcionan los costes de desarrollo más bajos y el mejor rendimiento cuando se usan objetos gracias a que almacenan objetos en disco y tienen una integración transparente con el programa escrito en un lenguaje de programación orientado a objetos, al almacenar exactamente el modelo de objeto usado a nivel aplicativo, lo que reduce los costes de desarrollo y mantenimiento.

3.6. Manejo de la persistencia de objetos

3.6.1. ¿Qué se entiende por lógica de persistencia?

La lógica de persistencia abarca todo el código que una aplicación requiere para poder grabar y recuperar la información inherente a su dominio. Por ejemplo, una aplicación de *eBanking* va a persistir y recuperar información de cuentas, clientes, transacciones, etcétera.

3.6.2. ¿Dónde persisten los datos?

Por lo general, la mayoría de estas aplicaciones persisten su información en una base de datos relacionales (RDBMS). Si bien existen otros medios alternativos, como ser una base de datos orientada a objetos (OODBMS), ninguno de estos se compara con la madurez y la popularidad de las bases de datos relacionales.

Utilizar un medio relacional ofrece una serie de ventajas muy importantes, por ejemplo:

- Tecnología madura
- Muy eficiente en la grabación y recuperación de grandes volúmenes de datos.
- Soportan transacciones (la mayoría)
- Aseguran la integridad de los datos (niveles de aislamiento, *locking*, etcétera).
- Excelente manejo de la seguridad
- Protocolo de consulta estándar (SQL)
- Buen soporte (muchos DBA's)
- Oferta variada

3.6.3. Estrategias de persistencia

En una aplicación .NET, al igual que sucede con otras tecnologías, se puede implementar la lógica de persistencia aplicando distintas estrategias. Al principio, determinar cuál es la estrategia más adecuada es una decisión de arquitectura más que importante.

3.6.3.1. Estrategia típica

Para el desarrollador.NET, la estrategia más típica es utilizar directamente ADO.NET bajo esta estrategia, la mayor parte del código escrito se centra en recuperar un *snapshot* de la base de datos en un DataSet, modificar eventualmente el DataSet en memoria, para posteriormente, a través del DataAdapter correspondiente, aplicar los cambios contra la base de datos subyacente. Si bien los DataSets funcionan correctamente, desde el punto de vista de la orientación a objetos evidencian las siguientes desventajas:

- Los DataSets representan información tabular, no representan objetos del dominio.

- Los DataSets representan relaciones entre tablas, no representan los distintos tipos de asociaciones que surgen entre los objetos del dominio.

- Los DataSets son extremadamente sensibles a los cambios que puedan surgir en el esquema de la base de datos.

- El tipo de código generado para manipular los DataSets tiende a ser repetitivo y relativamente difícil de mantener.

3.6.3.2. Estrategia recomendada

Una estrategia más elegante y compatible con las buenas prácticas de diseño pregonadas en estos últimos diez años, es la de diseñar un modelo de objetos del dominio que represente el 100% de la información que maneja la aplicación y utilizar un *framework* de *Object Relational Mapping* (ORM) que resuelva en forma transparente la persistencia de estos objetos contra una base de datos relacional. Utilizar un *framework* ORM ofrece entre otras las siguientes ventajas:

- Persistencia transparente: los objetos del dominio no saben nada acerca de la base de datos donde son persistidos, el *framework* lo resuelve en forma automática utilizando archivos de *mapping* expresados en XML.

- Soporte de polimorfismo: se pueden cargar jerarquías de objetos en forma polimórfica.

- Soporte de los tres niveles de mapeo de herencia: se puede mapear toda una jerarquía de clases a una sola tabla, crear una tabla por cada clase concreta o crear una tabla por cada escalón de la jerarquía.

- Soporte completo de asociaciones: los frameworks de ORM soportan el mapeo de todos los tipos de relaciones que pueden existir en un modelo de objetos del dominio (asociaciones 1..1, 1...N, N..M, unidireccionales y bidireccionales).

- Soporte de carga de objetos Proxy: se pueden cargar objetos que solo contengan la clave del objeto completo.

- Soporte de *caching*: en el contexto de una transacción, se puede disminuir la cantidad de veces que se va contra la base de datos cacheando en memoria los objetos que son accedidos varias veces.

- Soporte de múltiples dialectos SQL: se puede independizar completamente del tipo de base de datos utilizada. La aplicación puede persistir sus datos en SQL Server, en Oracle, en MySQL, etcétera, simplemente cambiando la configuración correspondiente.

Los dialectos soportados actualmente por NHibernate son:

- RDBMSDialect
- Microsoft SQL Server
- 2000NHibernate.Dialect.MsSql2000Dialect
- Microsoft SQL Server 7NHibernate.Dialect.MsSql7Dialect
- DB2NHibernate.Dialect.DB2Dialect
- PostgreSQLNHibernate.Dialect.PostgreSQLDialect
- MySQLNHibernate.Dialect.MySQLDialect
- Oracle (any version)NHibernate.Dialect.OracleDialect
- Oracle 9/10gNHibernate.Dialect.Oracle9Dialect
- SybaseNHibernate.Dialect.SybaseDialect
- FirebirdNHibernate.Dialect.FirebirdDialect

Para los desarrolladores ADO.NET el valor de la propiedad hibernate.connection.connection_string les debe resultar familiar. Se trata justamente de la cadena de conexión en formato ADO.NET que NHibernate utilizará para conectarse con la base de datos. Utilizar la API de NHibernate para persistir y recuperar los objetos.

Para poder utilizar la API de NHibernate desde la propia aplicación lo único que se debe hacer, es agregar una referencia a la librería NHibernate.dll ubicada en la carpeta de <NHibernate-Home>\bin más la referencia a la biblioteca de clases que contienen los objetos de dominio y sus *mappings*.

Una sesión de NHibernate funciona como una fachada que encapsula el acceso a las funcionalidades más importantes que ofrece el *framework*. A través de la sesión, NHibernate se puede manejar el contexto transaccional de la propia lógica.

Se entiende por persistencia (en programación) como la acción de preservar la información de un objeto de forma permanente (guardar), pero a su vez también, se refiere a poder recuperar la información del mismo (leer) para que pueda ser nuevamente utilizada.

En el caso de persistencia de objetos la información que persiste en la mayoría de los casos son los valores que contienen los atributos en ese momento, no necesariamente la funcionalidad que proveen sus métodos.

Desde la óptica de la persistencia, se podrían clasificar los objetos en:

- Transitorios: cuyo tiempo de vida depende directamente del ámbito del proceso que los instanció.

- Persistentes: cuyo estado es almacenado en un medio secundario para su posterior reconstrucción y utilización, por lo que su tiempo de vida es independiente del proceso que los instanció.

4. METODOLOGÍA DE GENERACIÓN DE LA ARQUITECTURA

4.1. Descripción de la metodología utilizar

Como se había mencionado con anterioridad se cuentan con varias alternativas para la realización y la generación de la arquitectura que se usará, cabe mencionar que en este capítulo únicamente quedará plasmado como funcionará dicha arquitectura como tal, luego del estudio y análisis de lo que es Spring.Net y el NHibernate como el ORM para el manejo de la comunicación con la base de datos y así mismo, para la persistencia se podrán dar cuenta que son dos herramientas muy valiosas y que aportan mucho para el manejo de la arquitectura de un sistema.

Para este propósito se dejará plasmada la arquitectura utilizando el *framework* Spring.Net y NHibernate para poder presentar un nuevo esquema de trabajo. Como ya se ha visto en capítulos anteriores de Spring.Net, se pueden usar los módulos que dicho *framework* contiene para el manejo de las diferentes transacciones u operaciones que se realicen.

Algunas recomendaciones para dicho *framework son*:

- Spring.Core: es el módulo fundamental y proporciona los servicios básicos en los que se asientan el resto, así como los patrones básicos (contenedor con inversion control) en los que se basa.

- Spring.Aop: módulo para la programación orientada a los aspectos

- Spring.Web: extiende ASP.NET con distintas funcionalidades, como soporte extendido de localización.

- Spring.Services: permite acceder a cualquier objeto normal como un servicio COM+ u objeto remoto.

- Spring.Data y Spring.ORM: abstracción de datos que se pueden usar con varios proveedores de datos, desde ADO.NET a varios ORM.

Con base en los diferentes módulos que maneja el Spring.Net así será el uso que se le dará a cada uno, sacando el mayor provecho de ellos, también se aprovecharán las bondades que proporciona el NHiernate como ORM que se utilizará para el manejo de mapeo de objetos relacionales hacia la base de datos.

La idea de la generación de esta arquitectura es dejarla con modularidad ya que con esto se estará logrando una mayor flexibilidad para futuras modificaciones, para ello y para el manejo de las transacciones se utilizará un *Web Service,* el cual permitirá recibir llamadas que se le harán desde el *Front-End* de cualquier aplicación, dicho *Web Service* realizará la distribución de cada una de las transacciones para que un componente DLL pueda ejecutar la lógica del negocio de cada una de las transacciones, y tener dentro de cada uno de los DLL, el manejo de la comunicación hacia la base de datos para lo cual se utilizará un ORM en este caso NHibernate.

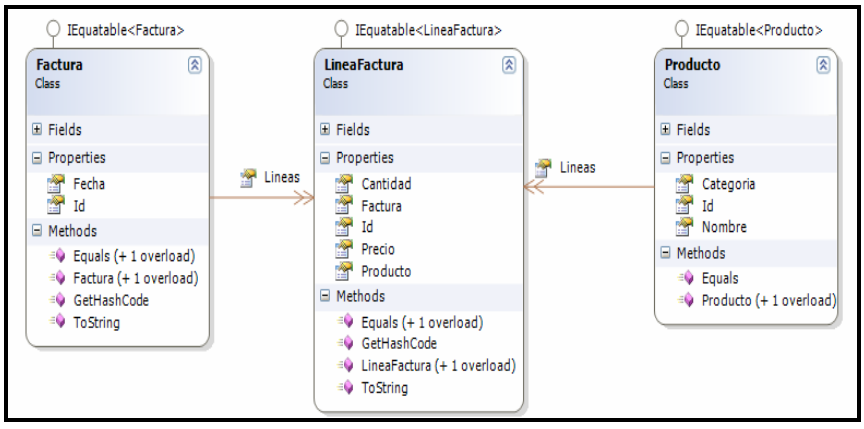
4.2. Descripción de la forma de manejo de la persistencia

Para el manejo de la persistencia de datos de esta arquitectura se utilizará el NHibernate, siendo un *framework* de ORM (*Object/Relational Mapping*), un *port* de Hibernate de Java, que tiene como función principal mapear los objetos desde una aplicación .Net a una base de datos relacional.

Primero, se debe hablar de como NH (NHibernate) realiza su funcionamiento básico y como se implementara en esta arquitectura.

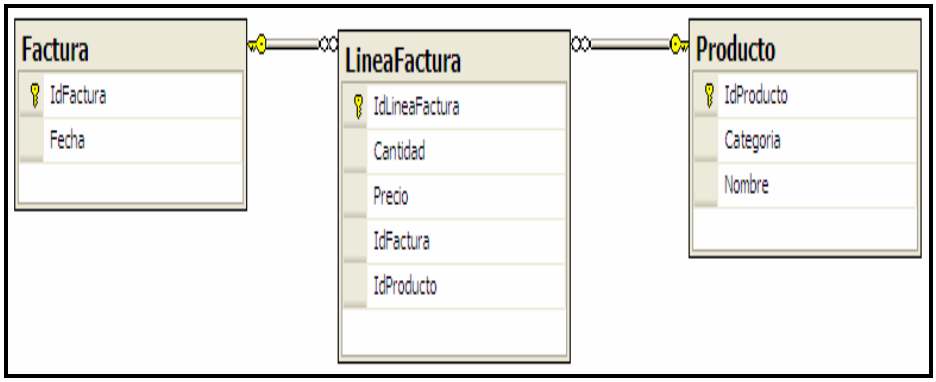
Para iniciar se utilizará el modelo de dominio para luego poder pasar al modelo relacional, la ventaja que da este *framework* es que se puede hacer en ambas vías, es decir, primero el modelo relacional y luego el modelo del domino o viceversa. Para este caso se muestra un modelo de dominio sencillo, el cual permite ver como son los componentes de cada una de las entidades.

Figura 3. **Modelo de dominio**



Fuente: elaboración propia.

Figura 4. **Modelo relacional**



Fuente: elaboración propia.

Una vez realizado cada uno de los modelos para manejar los archivos de mapeo de la información, con el objetivo de poder conocer la correspondencia que existe entre los objetos y las tablas por medio de dichas configuraciones, se puede lograr de forma programática o bien haciendo uso de archivos XML, dichos archivos contendrán la información necesaria para poder saber en qué tablas se tienen o tiene que guardar cada objeto o en qué tabla se tienen que realizar búsquedas.

Como se puede observar en la figura de abajo se muestra un archivo de configuración XML, dentro de este pueden encontrar atributos, *namespace* de la clase y todas las propiedades que servirán a lo largo de la persistencia dentro de la base de datos, como se darán cuenta en dicho archivo se cuenta con toda la información de las tablas que se estarán manejando en el modelo que se plasmará para el proyecto al cual se aplicará.

Figura 5. **Archivo de configuración xml**

```
<?xml version="1.0" encoding="utf-8" ?>
  <hibernate-mapping xmlns="urn:nhibernate-mapping-2.2" assembly="NH01"
namespace="Libreria.Proyecto.Entidades">
    <class name="Factura">
      <id name="Id" column ="IdFactura" type="int" unsaved-value="0">
        <generator class="identity"/>
      </id>
      <property name="Fecha" type="DateTime" not-null="true"/>
      <bag name="Lineas" cascade="all-delete-orphan">
        <key column="IdFactura"/>
        <one-to-many class="LineaFactura"/>
      </bag>
    </class>
  </hibernate-mapping>
```

Fuente: elaboración propia.

Una sesión es un marco de trabajo en el cual NHibernate, establece una conversación entre la aplicación y el motor de base de datos relacional. Para construir una sesión, representada por ISession alguien proveerá la sesión, para esto se necesita de: ISessionFactory. El ISessionFactory se encarga de crear sesiones en la aplicación. En un momento, la aplicación puede tener una o más sesiones abiertas.

Cada sesión representa un primer nivel de caché, los objetos que son traídos desde la base o son guardados desde la aplicación y se encuentran en la cache de primer nivel. Si se solicita un objeto a la base, primero se busca en la caché, si se encuentra ahí el objeto, se lo devuelve a la aplicación sin solicitarlo al motor relacional. Si no se encuentra en la caché, se realiza la consulta a la base.

Por lo general, debe tener un ISessionFactory por aplicación, sería necesario tener más de uno en el caso de que se esté trabajando con más de una base de datos a la vez, esto cuando se manejan diferentes *pool* de conexiones, que en algún momento pueden llegar a ser de gran utilidad. Para configurar el ISessionFactory, es decir, para decirle con qué motor de base de datos se va a trabajar, la cadena de conexión (*connection string*), el *driver* que se utilizará entre otras cosas, como toda configuración en NH, se puede realizar de manera programática (por código) o mediante archivos de configuración XML, y dentro de esta última se puede hacer mediante el App.config o mediante el hibernate.cfg.xml. En este caso se utilizará la última:

Figura 6. **Archivo hibernate.cfg.xml**

```
<?xml version="1.0" encoding="utf-8" ?>
        <hibernate-configuration  xmlns="urn:nhibernate-configuration-2.2" >
          <session-factory name="NH01">
                <property
name="connection.provider">NHibernate.Connection.DriverConnectionProvider</property>
                <property
name="connection.driver_class">NHibernate.Driver.SqlClientDriver</property>
                <property
name="dialect">NHibernate.Dialect.MsSql2005Dialect</property>
                <property name="connection.connection_string">Data
Source=localhost\SQLEXPRESS;Initial Catalog=NH01;Integrated Security=True</property>
                <property name="show_sql">true</property>

                <mapping assembly="Dario.NH01" />

          </session-factory>
        </hibernate-configuration>
```

Fuente: elaboración propia.

El código necesario para configurar NHibernate al comienzo de la aplicación es:

Figura 7. **Código de configuración NHibernate**

```
Configuration cfg = new Configuration();
cfg.Configure("hibernate.cfg.xml");
ISessionFactory sesiones = cfg.BuildSessionFactory();
ISession sesion = sesiones.OpenSession();
```

Fuente: elaboración propia.

Primeramente, se debe crear un objeto del tipo *Configuration*, configurarlo mediante *Configure* ("hibernate.cfg.xml") y que se encargue de crear el ISessionFactory, por medio del método BuildSessionFactory(). Una vez creado el objeto sesiones del tipo ISessionFactory, ya se uede comenzar a crear objetos ISession.

En este caso se trabajará con una sola sesión en la aplicación. Cabe mencionar que como buena práctica se debe realizar un singleton del objeto ISessionFactory debido a que es un proceso costoso para la aplicación, en otras palabras, la ejecución de BuildSessionFactory se debe hacer una vez durante la ejecución, y mantener la conexión a lo largo del proceso que se está ejecutando, sería una mala práctica el realizar operaciones de abrir y cerrar la Base de datos durante todo el proceso de lógica del negocio, por tal motivo es que se realiza una sola vez el instanciamiento de la misma.

4.3. Diagrama y explicación de dicha tecnología

Basándose en la tecnología que se utilizará es necesario dar una breve explicación del funcionamiento de la misma y de cómo será aplicada a nuestro modelo del sistema.

En esta parte debe dejar claro que el modelo del sistema que se aplicará, será uno elegido al azar ya que con estas explicaciones se podría aplicar a cualquier modelo que se requiera.

La explicación del diagrama de la arquitectura del Spring.Net se puede identificar de la siguiente manera:

- *Framework:* porque define la forma de desarrollar aplicaciones J2EE, dando soporte y simplificando complejidad propia del *software* corporativo.

- Inversión de control *(*IoC*):* promueve el bajo acoplamiento a partir de la inyección de dependencias (DI) entre los objetos (relaciones).

- Orientación a aspectos (AOP): presenta una estructura simplificada para el desarrollo y utilización de aspectos (módulos *multiple object crosscutting*).

- Se puede usar en cualquier entorno de trabajo

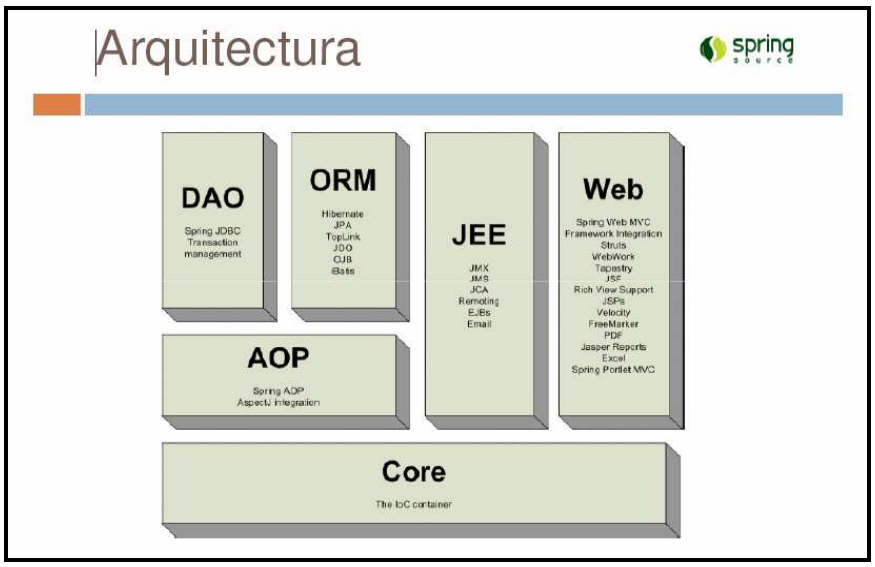
- Adoptar la filosofía POJO permite a Spring

- Beneficiarse de las características específicas del entorno sin sacrificar la portabilidad. Aplicación NO asociada al *Framework.*

- Aunque se centra en el nivel de arquitectura de negocio (manejo de EJB´s) es válido también para el resto de niveles de aplicación.

- Reduce la complejidad de la programación de aplicaciones a básicamente implementación de interfaces.

Figura 8. **Componentes Spring.Net**



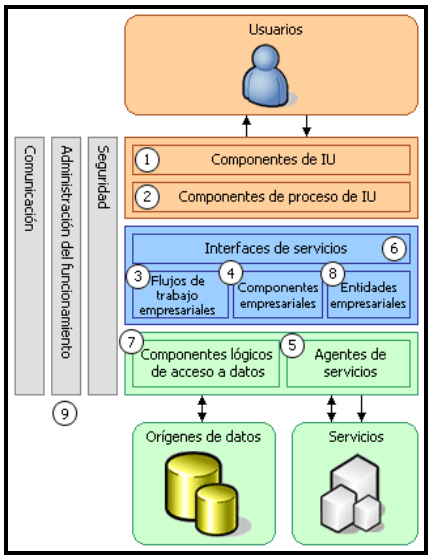
Fuente: www.springframework.net. 1 de marzo de 2011.

El análisis de la mayoría de las soluciones empresariales basadas en modelos de componentes por capas, muestran que existen varios tipos de componentes habituales. En la figura se muestra una ilustración completa en la que se indican estos tipos de componentes.

Nota: el término componente hace referencia a una de las partes de la solución total, como los componentes de *software* compilado (por ejemplo, los ensamblados de Microsoft .NET, ORM, etcétera) y otros elementos de *software*, como las páginas *Web* y los programas de Microsoft®, BizTalk®, *Server Orchestration.*

Aunque la lista que se muestra en la figura no es completa, representa los tipos de componentes de *software* más comunes encontrados en la mayoría de las soluciones distribuidas. A lo largo de este capítulo se describirá cada uno de ellos.

Figura 9. **Componentes identificados en el escenario del ejemplo**



Fuente: www.springframework.net. 1 de marzo de 2011.

Los tipos de componentes identificados en el escenario de diseño de ejemplo son:

- Componentes de interfaz de usuario (IU): la mayor parte de las soluciones necesitan ofrecer al usuario un modo de interactuar con la aplicación. En el ejemplo de aplicación comercial, un sitio *Web* permite al cliente ver productos y realizar pedidos, y una aplicación basada en el entorno operativo Microsoft Windows® permite a los representantes de ventas escribir los datos de los pedidos de los clientes que han telefoneado a la empresa. Las interfaces de usuario se implementan utilizando formularios de *Windows Forms*, páginas Microsoft ASP.NET, controles u otro tipo de tecnología que permita procesar y dar formato a los datos de los usuarios, así como, adquirir y validar los datos entrantes procedentes de estos.

- Componentes de proceso de usuario: en un gran número de casos, la ínteractuación del usuario con el sistema se realiza, de acuerdo con un proceso predecible. Por ejemplo, en la aplicación comercial, se puede implementar un procedimiento que permita ver los datos del producto. De este modo, el usuario puede seleccionar una categoría de una lista de productos disponibles y, a continuación, elegir uno de los productos de la categoría seleccionada para ver los detalles correspondientes.

  Del mismo modo, cuando el usuario realiza una compra, la ínteractuación sigue un proceso predecible de recolección de datos por parte del usuario, por el cual este en primer lugar proporciona los detalles de los productos que desea adquirir, a continuación los detalles de pago y por último, la información para el envío. Para facilitar la sincronización y organización de las interactuaciones con el usuario, resulta útil usar componentes de proceso de usuario individuales.

De este modo, el flujo del proceso y la lógica de administración de estado no se incluyen en el código de los elementos de la interfaz de usuario, por lo que varias interfaces podrán utilizar el mismo motor de ínteractuación básica.

- Flujos de trabajo empresariales: una vez que el proceso de usuario ha recopilado los datos necesarios, estos se pueden utilizar para realizar un proceso empresarial. Por ejemplo, tras enviar los detalles del producto, el pago y el envío a la aplicación comercial, puede comenzar el proceso de cobro del pago y preparación del envío. Gran parte de los procesos empresariales conllevan la realización de varios pasos, los cuales se deben organizar y llevar a cabo en un orden determinado.

  Por ejemplo, el sistema empresarial necesita calcular el valor total del pedido, validar la información de la tarjeta de crédito, procesar el pago de la misma y preparar el envío del producto. El tiempo que este proceso puede tardar en completarse es indeterminado, por lo que sería preciso administrar las tareas necesarias, así como los datos requeridos para llevarlas a cabo. Los flujos de trabajo empresariales definen y coordinan los procesos empresariales de varios pasos de ejecución larga y se pueden implementar utilizando herramientas de administración de procesos empresariales, como BizTalk *Server Orchestration*.

- Componentes empresariales: independientemente de si el proceso empresarial consta de un único paso o de un flujo de trabajo organizado, la aplicación requerirá probablemente el uso de componentes que implementen reglas empresariales y realicen tareas empresariales.

Por ejemplo, en la aplicación comercial, deberá implementar una funcionalidad que calcule el precio total del pedido y agregue el costo adicional correspondiente por el envío del mismo. Los componentes empresariales implementan la lógica empresarial de la aplicación.

- Agentes de servicios: cuando un componente empresarial requiere el uso de la funcionalidad proporcionada por un servicio externo, tal vez sea necesario hacer uso de un código para administrar la semántica de la comunicación con dicho servicio.

  Por ejemplo, los componentes empresariales de la aplicación comercial descrita anteriormente, podrían utilizar un agente de servicios para administrar la comunicación con el servicio de autorización de tarjetas de crédito y utilizar un segundo agente de servicios, para controlar las conversaciones con el servicio de mensajería. Los agentes de servicios permiten aislar las idiosincrasias de las llamadas a varios servicios desde la aplicación y pueden proporcionar servicios adicionales, como la asignación básica del formato de los datos que expone el servicio al formato que requiere la aplicación.

- Interfaces de servicios: para exponer lógica empresarial como un servicio, es necesario crear interfaces de servicios que admitan los contratos de comunicación (comunicación basada en mensajes, formatos, protocolos, seguridad y excepciones, entre otros) que requieren los clientes. Por ejemplo, el servicio de autorización de tarjetas de crédito debe exponer una interfaz de servicios que describa la funcionalidad que ofrece el servicio, así como la semántica de comunicación requerida para llamar al mismo. Las interfaces de servicios también se denominan fachadas empresariales.

- Componentes lógicos de acceso a datos: la mayoría de las aplicaciones y servicios necesitan obtener acceso a un almacén de datos en un momento determinado del proceso empresarial. Por ejemplo, la aplicación empresarial necesita recuperar los datos de los productos de una base de datos para mostrar al usuario los detalles de los mismos, así como insertar dicha información en la base de datos cuando un usuario realiza un pedido.

  Por tanto, es razonable abstraer la lógica necesaria para obtener acceso a los datos en una capa independiente de componentes lógicos de acceso a datos, ya que de este modo se centraliza la funcionalidad de acceso a datos y se facilita la configuración y el mantenimiento de la misma, para este punto es donde se utilizará el ORM, esto con el objetivo de manejar la comunicación con la base de datos propia.

  Cabe mencionar que dicho ORM es el NHirbenate el cual se puede configurar y manejar con diferentes bases de datos, esto con base en su archivo de configuración, la ventaja de dicho archivo es que es parametrizable ya que es un XML, el cual permitirá ser cambiado y luego no necesariamente realizar cambios a nivel de toda la aplicación.

- Componentes de entidad empresarial: la mayoría de aplicaciones requieren el paso de datos entre distintos componentes. Por ejemplo, en la aplicación comercial es necesario pasar una lista de productos de los componentes lógicos de acceso a datos a los componentes de la interfaz de usuario para que este pueda visualizar dicha lista.

Los datos se utilizan para representar entidades empresariales del mundo real, como productos o pedidos. Las entidades empresariales que se utilizan de forma interna en la aplicación suelen ser estructuras de datos, como conjuntos de datos, Data Reader o secuencias de lenguaje de marcado extensible (XML), aunque también se pueden implementar utilizando clases orientadas a objetos personalizadas que representan entidades del mundo real, necesarias para la aplicación, como productos o pedidos.

- Componentes de seguridad, administración operativa y comunicación: la aplicación probablemente utilice también componentes para realizar la administración de excepciones, autorizar a los usuarios a que realicen tareas determinadas y comunicarse con otros servicios y aplicaciones.
  Básicamente el modelo de la arquitectura a desarrollar se aplicará al modelo utilizando Spring.Net y NHibernate.

4.4. Modelado del sistema

Básicamente, el sistema de ejemplo que se modelará es el de realización de Control de expedientes, en el que se puede identificar que esta arquitectura es un modelo n-capas, el cual ya se había mencionado con anterioridad, una breve descripción del sistema es el siguiente:

- Se procederá a tener un *Web Service* el cual brindará todo el manejo de transacciones de la lógica del negocio, que se manejará dentro de la aplicación, para ello se contará con el uso del Spring.Net y para el manejo de la base de datos el ORM (NHibernate).

- Se debe contar con componentes que jueguen un papel muy importante en el desarrollo del proyecto, cada componente es una DLL que proporcionará toda la lógica del negocio dependiendo la opción que se realice para cada una de las transacciones ya que el *Web Service* es el encargado de orquestar hacia que DLL se redirecciona la petición solicitada en cierto momento.

- Se manejarán archivos de configuración, utilizando los modelos de dominio y los modelos de entidad relación para el manejo del mapeo de los XML, que se utilizarán y que contendrán toda la información de las tablas que se utilizarán a lo largo del sistema.

- Se contará con una Base de datos de cualquier tipo, dependiendo las necesidades del proyecto, así como, el análisis de costos de la empresa, es decir, en este punto la empresa es la que tiene la decisión de qué Base de datos es la más conveniente para el manejo de su información.

- Se utilizará el *Web forms* para el manejo de todo el *front-end* del sistema que se realizará en su momento, cabe mencionar que aquí se deja claro que no quedará un sistema desarrollado únicamente, se está haciendo énfasis y dando lineamientos que se pueden utilizar para la generación de una arquitectura, utilizando el Spring.Net y el NHibernate.

Como prototipo para el manejo y aplicación del sistema se puede utilizar el siguiente sistema que puede servir como ejemplo.

El proceso incluye desde el manejo de expedientes iniciando desde su ingreso o registro, pasando por todo el proceso de asignación, reasignación, seguimiento, resolución, rechazo o cierre del expediente, registrando el historial de las fases en las cuales ha estado o está un expediente y sus documentos.

Para ingresar al Sistema se deberá escribir el nombre de:

- Usuario y
- Contraseña asignada

Figura 10. **Pantalla de ingreso a la aplicación**

Fuente: elaboración propia.

Se despliega una pantalla de bienvenida con el nombre del usuario que ingresó.

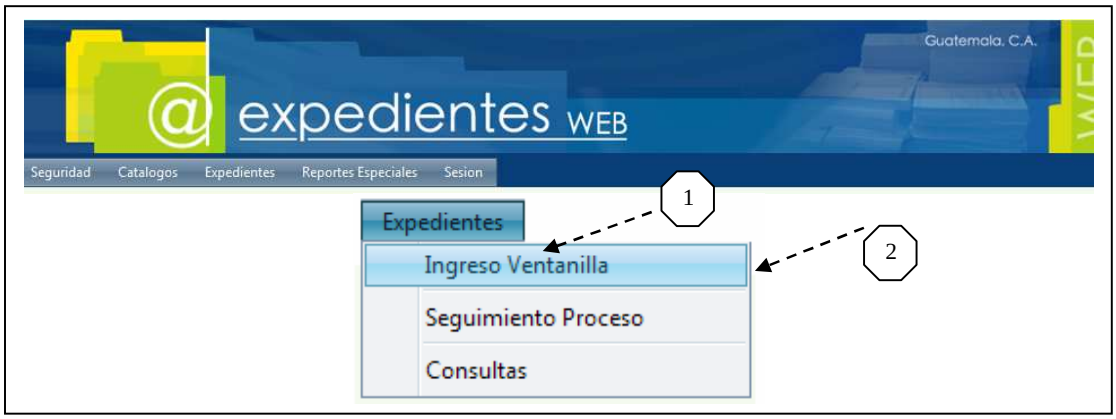
Figura 11. **Pantalla de bienvenida al usuario**

Fuente: elaboración propia.

Se debe seleccionar una opción en el menú:

Expedientes e Ingreso Ventanilla

Figura 12. **Pantalla del menú de expedientes**



Fuente: elaboración propia.

Al dar clic en Ingreso Ventanilla, se despliega la siguiente pantalla:

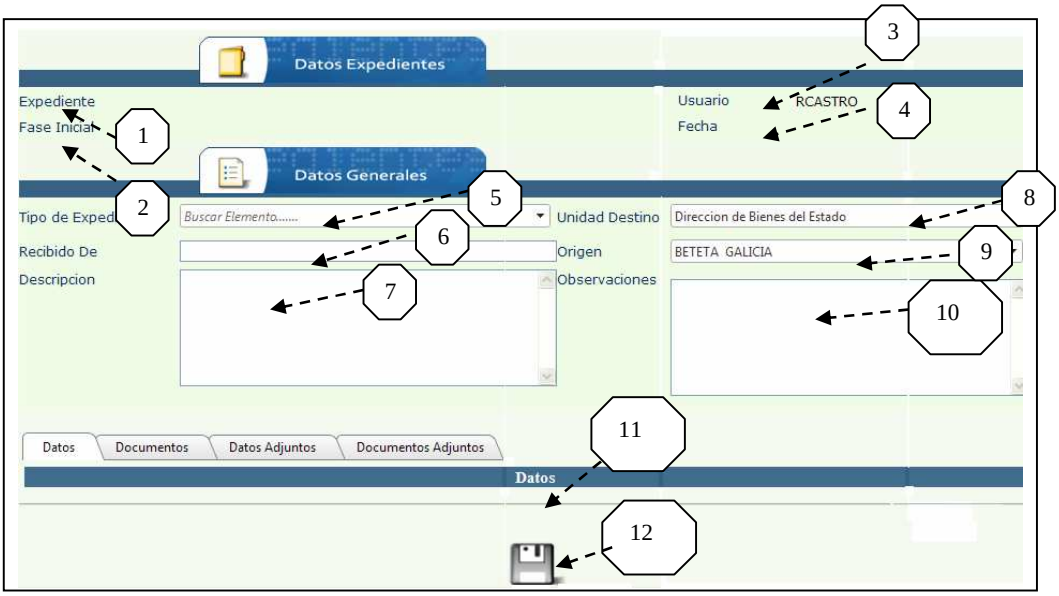
Datos del expediente:

- Expediente
- Fase inicial
- Usuario y
- Fecha

Datos generales:

- Tipo de expediente: se despliega una lista de opciones
- Recibido de: nombre de quien se recibe el expediente
- Descripción: breve detalle del expediente recibido
- Unidad de destino: unidad, dirección o ministerio de finanzas
- Origen: procedencia de expedientes (interno o externo)
- Observaciones: algún detalle importante
- Plantillas de datos, documentos y adjuntos
- Grabación de datos de expedientes

Figura 13. **Pantalla de datos del expediente**



Fuente: elaboración propia.

Pestañas de datos:

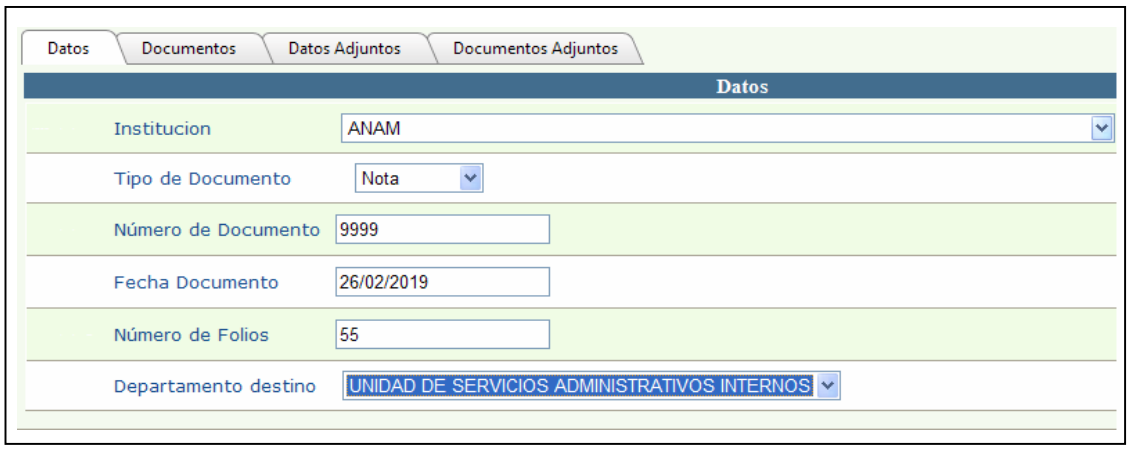
- Datos: que varían dependiendo del tipo de expediente, ejemplo.
  - Institución (Seleccionar de una lista que despliega)
  - Tipo de documento (Seleccionar de una lista)
  - Número de documento
  - Fecha de documento
  - Departamento destino, etcétera
- Documentos
- Datos adjuntos
  - Tipo de dato (Seleccionar de una lista)
  - Valor
  - Valor adicional

Se Graba la descripción, con opción a eliminar

- Documentos adjuntos

Ejemplo de datos:

Figura 14. **Pantalla de ejemplo de datos**



Fuente: elaboración propia.

Ejemplo de datos adjuntos:

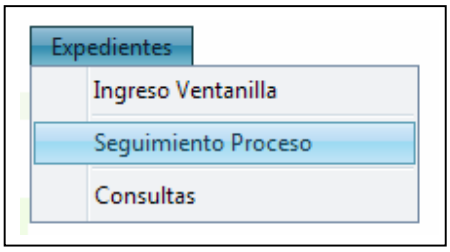
Figura 15. **Pantalla de ejemplo de datos adjuntos**

Fuente: elaboración propia.

Se debe seleccionar una opción en el menú:

- Expediente
- Seguimiento proceso

Figura 16. **Menú expedientes**



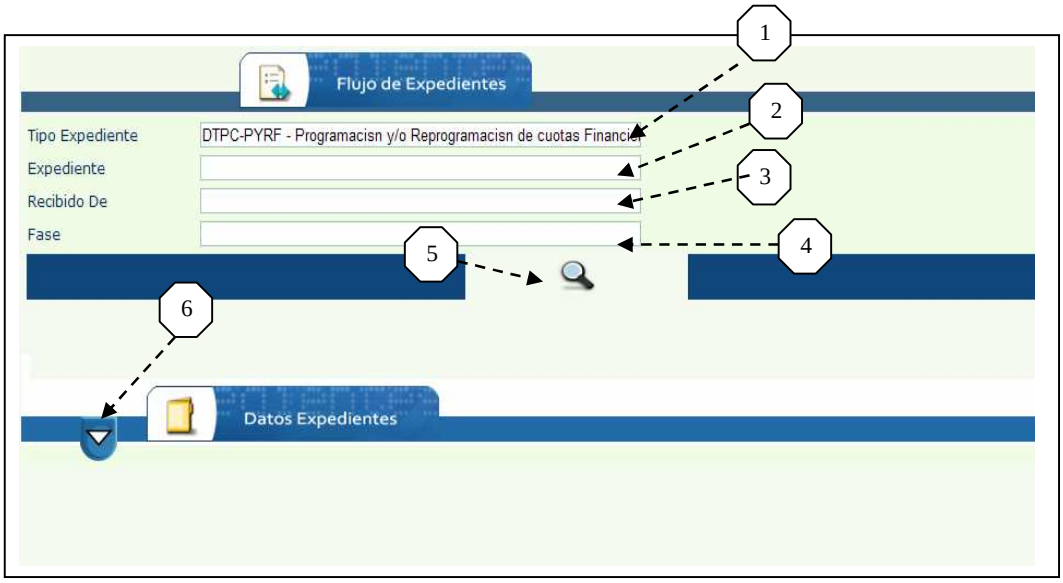
Fuente: elaboración propia.

Flujo de expediente:

- Tipo de Expediente
- Expediente
- Recibido De
- Fase
- Ícono para búsqueda y
- Datos Expedientes

Aparecerá la siguiente pantalla:

Figura 17. **Pantalla de flujo de expedientes**



Fuente: elaboración propia.

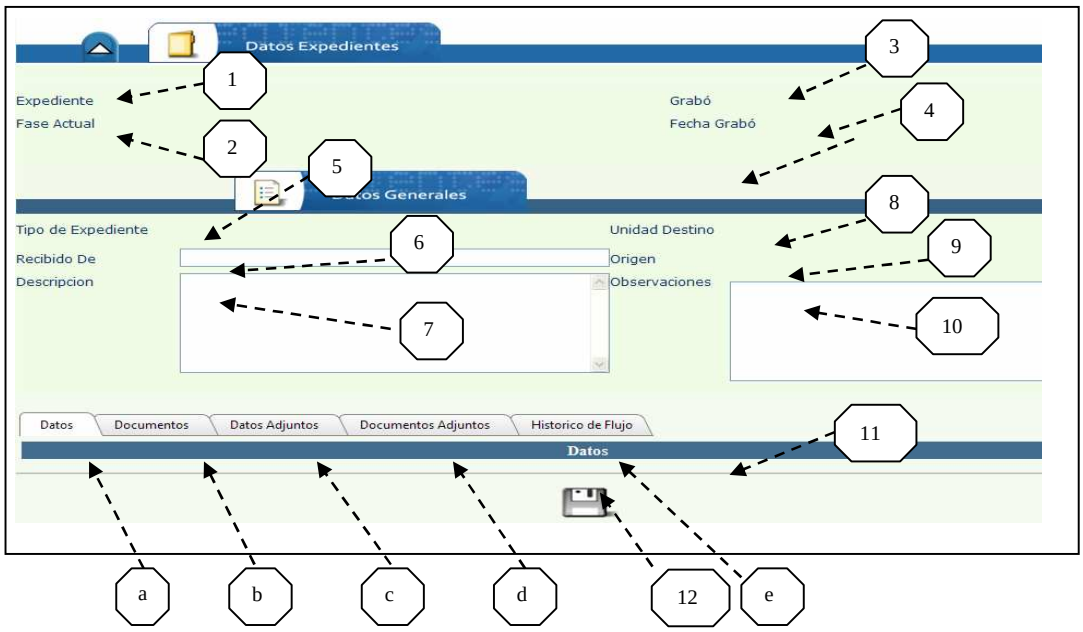
Datos expedientes:

- Expediente
- Fase actual
- Grabó
- Fecha grabó
- Tipo de expediente
- Recibo de
- Descripción
- Unidad de destino
- Origen
- Observaciones
- Datos

- Datos
- Documentos
- Datos adjuntos
  - Tipo de dato
  - Valor
  - Valor adicional
- Documentos adjuntos
- Histórico de flujo

- Grabar

Figura 18. **Pantalla de datos de expedientes**

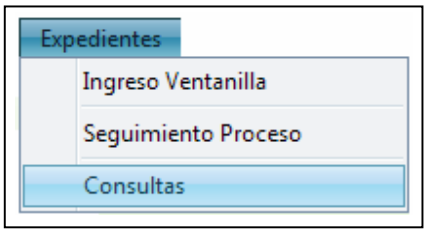


Fuente: elaboración propia.

Se debe seleccionar una opción en el menú:

Expedientes, Consultas

Figura 19. **Menú consultas**

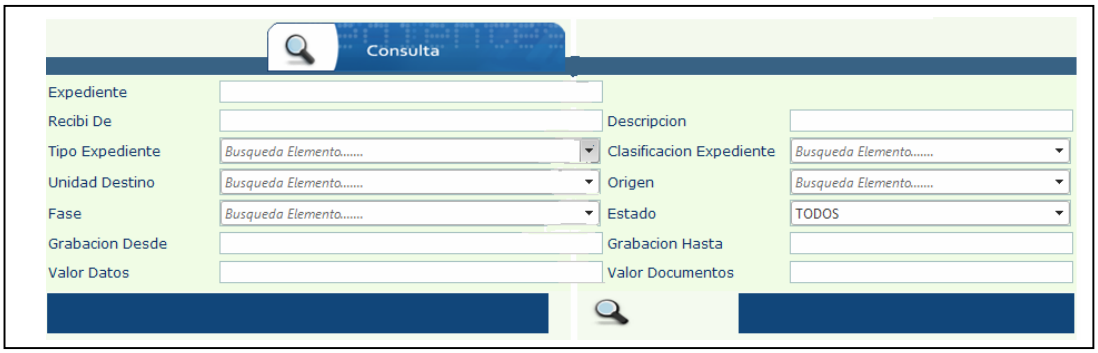


Fuente: elaboración propia.

Consulta de expedientes:

- Expediente
- Recibí de
- Tipo de Expediente
- Unidad de destino
- Fase
- Grabación desde
- Valor datos
- Descripción
- Clasificación Expediente
- Origen
- Estado
- Grabación hasta
- Valor Documentos
- Consulta expediente

Figura 20. **Pantalla de ingreso de datos de consultas**

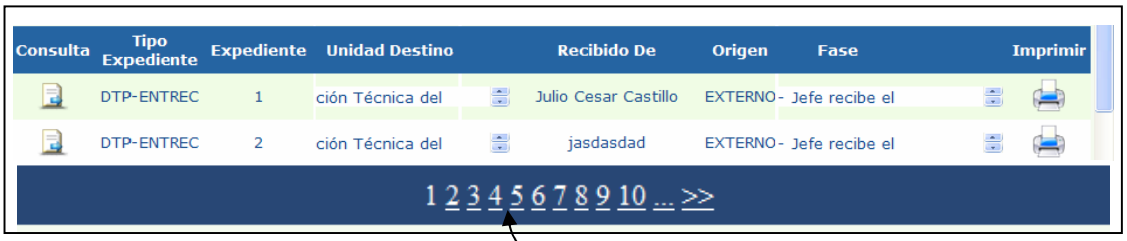


Fuente: elaboración propia.

Consulta expediente:

Consulta, Tipo expediente, Expediente, Unidad destino, Recibido de, Origen, Fase e Imprimir

Figura 21. **Pantalla que despliega la consulta de los expedientes**



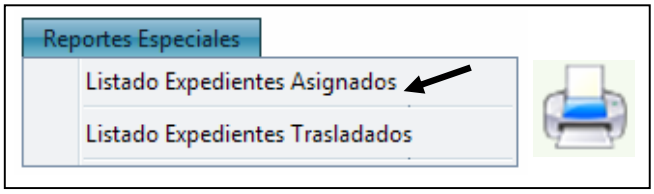
Fuente: elaboración propia.

Puede desplegarse más de una página...

Se debe seleccionar una opción en el menú:

Reportes especiales, listado de expedientes asignados:

Figura 22. **Menú de reportes especiales**



Fuente: elaboración propia.

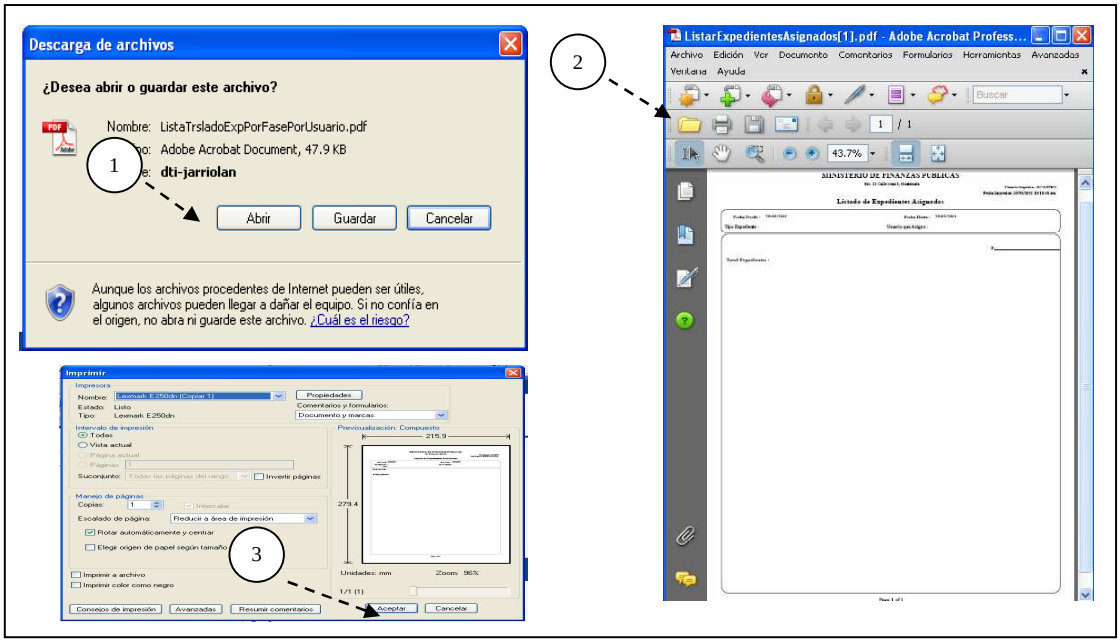
- Fecha desde dd/mm/aaaa
- Fecha hasta dd/mm/aaaa
- Tipo de expediente: buscar elemento de una lista que se despliega
- Usuario que asigna: buscar de una lista que se despliega
- Tipo de descripción: buscar de una lista que se despliega
- Imprimir: imprimir o guardar el listado

Figura 23. **Pantalla de ingresos de datos para reportes especiales**

Fuente: elaboración propia.

Al hacer clic en el ícono de la impresora se desplegará el mensaje de Abrir, Guardar o Cancelar, si se seleccionó abrir aparecerá el formulario y el detalle para imprimir: Aceptar.

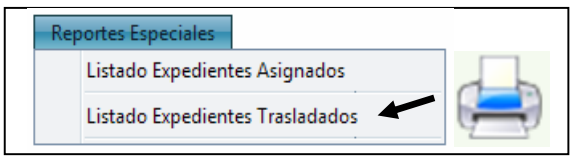
Figura 24. **Pantallas para mensajes de diálogo**



Fuente: elaboración propia.

Reportes especiales, listado de expedientes trasladados:

Figura 25. **Menú de expedientes trasladados**



Fuente: elaboración propia.

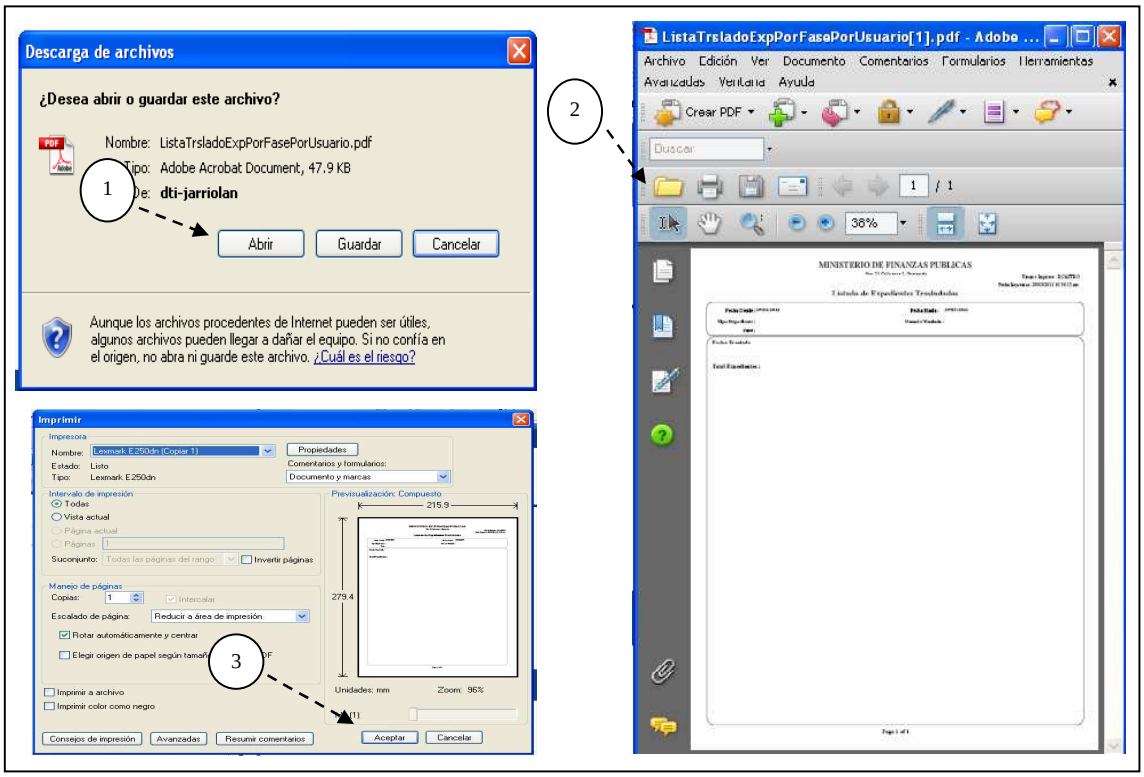
- Fecha desde: dd/mm/aaaa
- Fecha hasta: dd/mm/aaaa
- Tipo de expediente: de una lista que se despliega
- Usuario traslada: de una lista que se despliega
- Fase: de una lista que se despliega
- Tipo de descripción: puede haber una lista
- Para imprimir o guardar el listado

Figura 26. **Datos para el reporte de expedientes trasladados**

Fuente: elaboración propia.

Al hacer clic en el ícono de la impresora se desplegará el mensaje de Abrir, Guardar o Cancelar, si se seleccionó abrir aparecerá el formulario y el detalle para imprimir: Aceptar.

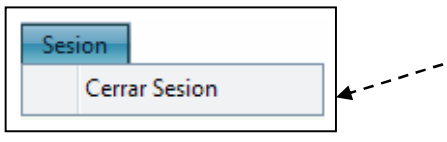
Figura 27. **Pantallas para mensajes de diálogo**



Fuente: elaboración propia.

Para cerrar la sesión hacer clic en Cerrar Sesión.

Figura 28. **Opción del menú para finalizar la sesión**



Fuente: elaboración propia.

Puede reiniciar la sesión ingresando nuevamente su Usuario y Contraseña asignada.

Figura 29. **Pantalla de finalización de sesión**

Fuente: elaboración propia.

CONCLUSIONES

1. Para contar con sistemas de información confiable y disponible al momento de requerirla, es necesario disponer de una arquitectura estable y robusta que cumpla las necesidades que requieren las aplicaciones.

2. Basados en estudios realizados en este documento de las distintas tecnologías con las que se cuenta en la actualidad, y teniendo en mente que el avance de las mismas es muy acelerado, se tomó la decisión de realizar una arquitectura moderna y que se ajuste a sistemas de información, tanto económica como tecnológicamente.

3. Después de haber realizado un análisis de la situación actual del manejo de información, se llega a la conclusión de que la generación de una arquitectura basada en Spring.Net, se realiza de una forma más simple y permite independencia con la base de datos, al realizar cambios sencillos de configuración.

4. Se realiza el manejo de sistemas de información de una forma clara y breve, de forma que se puede tener el conocimiento de los datos para su mejor utilidad.

5. Es importante conocer nuevas tendencias en cuanto a paradigmas para el desarrollo de aplicaciones nuevas y en este sentido se estudiará la programación orientada a aspectos, relativamente recientes, cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de incumbencias.

RECOMENDACIONES

1. Contar con el equipo (servidores, infraestructura de red, etcétera) necesario y actualizado para soportar el manejo de la arquitectura generada y así cumplir con la fusión para la cual fue diseñada, en otras palabras no se puede esperar mucho de la arquitectura si no cuenta con el *hardware* que la soporte adecuadamente.

2. Utilizar bases de datos adecuadas para el manejo de datos y tratar de definir una de ellas, de esta forma se minimizarán los cambios que se tengan que realizar en las clases que se pegan a la capa de datos y en las configuraciones de los archivos necesarios.

3. Adquirir equipos con amplia capacidad de expansión para soportar cambios futuros, como por ejemplo, ampliación de memoria, espacio en disco, etcétera. Esto debido a que actualmente la tecnología avanza aceleradamente.

4. Realizar una buena definición de todos los aspectos que soportará la arquitectura, desde el inicio con el fin de trabajar sobre una misma línea y llegar a un objetivo trazado sin realizar tantos cambios en el camino y explotar al 100% las bondades de dicha arquitectura.

5. La arquitectura generada debe ser utilizada como una herramienta capaz de soportar las aplicaciones desarrolladas, en todo su esplendor es decir, trabajando al 100% y desplegando toda su funcionalidad.

BIBLIOGRAFÍA

1. CAPRA, Luis. *Framework de persistencia de objetos* [en línea]. Disponible en Web: <http://markmail.org/download.xqy?id=ih65btdkitykd5v&number=1>. [Consulta: en abril 2011].

2. JBOSS, Community. *Hinernate* [en línea]. Disponible en Web: <https://www.hibernate.org/>. [Consulta: en marzo 2011].

3. MILES, Russ; POLLACK, Mark; EICHINGER, Erich. *The definitive guide to Spring for .NET*. USA: Apress, 2009. 600 p. ISBN: 1430224096.

4. Nhibernate. *Ventajas y desventajas* [en línea]. Disponible en Web: <http://blog.pucp.edu.pe/item/15355>. [Consulta: en mayo 2011].

5. OLIVA, Gabriel. *Inyección de dependencias con Spring framework* [en línea]. Disponible en Web: <http://www.slideshare.net/gabriel.oliva/MSDN-Webcast-Inyeccin-de-dependencias-con-Spring-Framework>. [Consulta: en enero 2011].

6. QUINTANA, Darío. *Inyección de dependencia* [en línea]. Disponible en Web: <http://blog.darioquintana.com.ar/2006/12/28/inyeccin-de-dependencia-con-springnet/>. [Consulta: en marzo 2011].

7. QUINTANA, Darío. *Tutorial NHibernate primeros pasos*. [en línea]. Disponible en Web: <http://darioquintana.com.ar/articles/tutorial-de-nhibernate-primeros-pasos>. [Consulta: en marzo 2011].

8. SCOTT, Millett. *NHibernate with ASP.NET problem design solution*. USA: Kindle, 2010. 423 p.

9. SPRING.NET. *Application framework* [en línea]. Disponible en Web: <http://www.springframework.net/>. [Consulta: en febrero 2011].

10. SPRING.NET. *Training* [en línea]. Disponible en Web: <http://www.springframework.net/documentation.html#Books>. [Consulta: en febrero 2011].

11. TARINGA. *NHibernate* [en línea]. Disponible en Web: <http://www.taringa.net/posts/offtopic/131352/nHibernate.html>. [Consulta: en abril 2011].

12. TENEMBAUM, Andrew. *Redes de computadoras*. México: Prentice Hall Hispanoamérica, 1998. 814 p. ISBN: 9688809586.

13. TUXPUC. NHibernate [en línea]. Disponible en Web: <http://tuxpuc.pucp.edu.pe/articulo/nhibernate-ventajas-y-desventajas>. [Consulta: en marzo 2011].

14. VERSIÓNCERO. *Spring.Net* [en línea]. Disponible en Web: <http://www.versioncero.com/noticia/409/springnet>. [Consulta: en diciembre 2010].

15. VISCUSO, German. *Bases de objetos* [en línea]. Disponible en Web: <https://www.db4o.com/espanol/db4o%20Whitepaper%20-%20Bases%20de%20Objetos.pdf>. [Consulta: en marzo 2011].

16. WIKIPEDIA. *Hibernate* [en línea]. Disponible en Web: <http://es.wikipedia.org/wiki/Hibernate>. [Consulta: en abril 2011].

17. YOUTUBE. *Introducción a NHibernate* [en línea]. Disponible en Web: <http://www.youtube.com/watch?v=PKHIO-tvYiM&hl=es>. [Consulta: en marzo 2011].