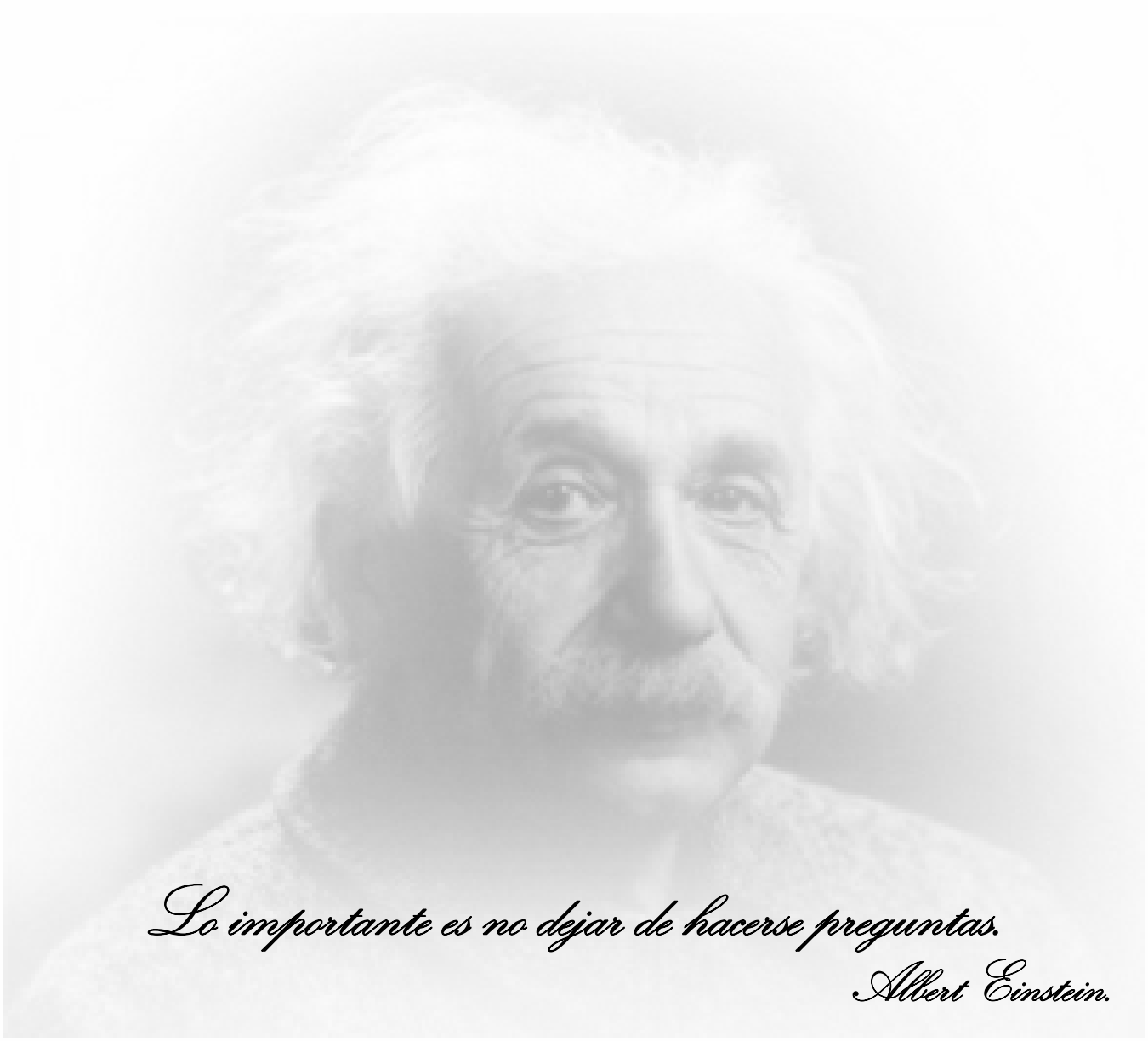
UNIVERSIDAD DE LAS CIENCIAS INFORMATICAS
FACULTAD 3

Arquitectura del sistema de Administración Financiera para SAREN.

TRABAJO DE DIPLOMA PARA OPTAR POR EL TÍTULO DE INGENIERO EN CIENCIAS INFORMÁTICAS.

Autor: Yunier Pérez Barroso.
Tutor: Ing. Yosvany Marquez Ruiz.

Ciudad de la Habana
Mayo 2008


Lo importante es no dejar de hacerse preguntas.
Albert Einstein.

DECLARACIÓN DE AUTORÍA

Declaro que soy el único autor de este trabajo y autorizo a la Facultad 3 de la Universidad de las Ciencias Informáticas a hacer uso del mismo en su beneficio.

Para que así conste firmo la presente a los ___ días del mes _____ del año _____.

_____ _____

Yunier Pérez Barroso Ing. Yosvany Márquez Ruiz

AGRADECIMIENTOS

A la Revolución y a la Universidad de las Ciencias Informáticas por hacerme partícipe de esta gran idea y permitirme aportar mi pequeño grano de arena a la misma.
A mi mamá y mi papá por su apoyo en todo momento, todo lo que soy se lo debo a ellos, a su ejemplo.
A mi hermano por su preocupación, por ayudarme a disminuir la carga cuando más me hace falta.
A tati por estar a mi lado siempre, por sopórtame tantas y tantas horas de cansancio.
A Maikel y Alejandro: sin ustedes hermanos la carga hubiera sido más pesada. De todo corazón gracias.
A Eddy por ayudarme tanto con tu conocimiento, con tus eternas críticas, pero sobre todo por ser mi camarada desde siempre.
Al Yosva, por ser mí amigo por encima de todo, por apoyarme en todas mis decisiones, por criticarme y porque trabajando juntos he aprendido muchas cosas.
A Yaumy,Henrik, Isma, Lulu por aconsejarme, ayudarme a crecer y permitirme contar con ustedes cuando necesitaba ayuda.
A Zolia y Gustavo, con ustedes empezó todo esto, gracias.
A todos los que de una forma u otra me han brindado su ayuda en los momentos difíciles: el piquete de la facultad 10, el proyecto Prisiones, a los buenos colegas de R&N, en fin a todos gracias también.

Con todo mi afecto, muchas gracias.

Yunier.

DEDICATORIA

A mis familiares.

RESUMEN

Las necesidades actuales que tiene toda organización para el logro de sus objetivos, demandan la construcción de grandes y complejos sistemas de software que requieren de la combinación de diferentes tecnologías y plataformas de hardware y software para alcanzar un funcionamiento acorde con dichas necesidades. Lo anterior, exige de los profesionales dedicados al desarrollo de software poner especial atención y cuidado al diseño de la Arquitectura sobre la cual estará soportado el funcionamiento de sus sistemas.

El presente trabajo, describe la Arquitectura del Sistema de Administración Financiera ofreciendo un marco idóneo para unificar los componentes que pertenecen a la solución bajo una misma línea; a través de los diferentes artefactos que resultan más representativos, acorde a la metodología RUP (Rational Unified Process), para el proceso de desarrollo. Además contiene un análisis de los resultados obtenidos a partir de los objetivos planteados y las restricciones arquitectónicas que se pueden encontrar y hacer que colapsen o provoquen un mal funcionamiento de la solución en un momento determinado.

ÍNDICE

Introducción

Las necesidades actuales que tiene toda organización para el logro de sus objetivos, demandan la construcción de grandes y complejos sistemas de software que requieren de la combinación de diferentes tecnologías y plataformas de hardware y software para alcanzar un funcionamiento acorde con dichas necesidades. Lo anterior, exige de los profesionales dedicados al desarrollo de software poner especial atención y cuidado al diseño de la Arquitectura sobre la cual estará soportado el funcionamiento de sus sistemas.

Un Sistema de Administración Financiera comprende la automatización de un conjunto de leyes, normas y procedimientos destinados a la obtención, asignación, uso, registro y evaluación de los recursos financieros, que tiene como propósito la eficiencia de la gestión de los mismos para la satisfacción de las necesidades colectivas.

La Administración Financiera del Sector Público en la República Bolivariana de Venezuela está integrada por un conjunto de subsistemas que deben operar en forma interrelacionada, atendiendo el mandato que en ese sentido establece la Ley Orgánica de la Administración Financiera del Sector Público (LOAFSP).

En el marco del Programa de Modernización de los Registros y Notarías de la República Bolivariana de Venezuela se decidió sustituir, en el Ministerio del Poder Popular para las Relaciones Interiores y de Justicia (MPPRIJ), el modelo financiero vigente para adoptar el que está reflejado en la LOAFSP; lo anterior incluye: establecer los niveles intermedios de administración de los fondos y una dirección descentralizada que fungen como unidades de ordenación de compromisos y pagos. Como parte de este proceso, se decidió la implementación de un software que controlará los procesos de Administración Financiera para el Servicio Autónomo de Registros y Notarias (SAREN), que es un ente descentralizado adscripto al MPPRIJ encargado de la dirección de los Registros y Notarias.

El sistema incluye el control y adecuado uso de los recursos financieros, así como el manejo eficiente y protección de los activos de SAREN cumpliendo y respetando lo establecido en la LOAFSP. Este es un elemento distintivo del nuevo modelo y fue el motivo fundamental que propició la no instauración de las aplicaciones que existían hasta el momento.

Por tanto, teniendo en cuenta que la institución SAREN no presenta en la actualidad una estructura financiera adecuada, los sistemas automatizados que están vigentes no se corresponden con las leyes, con la dinámica actual de la institución y que existen problemas de malversación, descontrol sobre todo de las finanzas y los recursos de SAREN, lo que ocasiona pérdidas de ingresos y desmoralización en la institución como ente del gobierno, resulta evidente que la construcción del software que resuelva toda esta realidad representa una salida necesaria a todos estos problemas.

Partiendo de esto, la situación problemática gira en torno a la construcción de esta solución de software y radica en *la inexistencia de una línea que defina la infraestructura que debe tener el Sistema de Administración Financiera en SAREN, que satisfaga las cualidades que se desean desarrollar en el mismo y centre el proceso de desarrollo en torno a ellas*. Esta situación lógicamente es el principal foco de atención de esta investigación de la cual se deriva el siguiente problema científico:

¿Qué elementos debe definir la Línea Base de la Arquitectura del Sistema de Administración Financiera en SAREN para que sea flexible, escalable, reutilizable y mantenible?

El objeto de estudio lo constituye el proceso de desarrollo del software de Administración Financiera que establece las fronteras del campo de acción el cual se centra en la arquitectura de software de este sistema.

Para dar solución a la problemática, el estudio realizado comprende en primer lugar la consulta de bibliografía relacionada con la Administración Financiera con el objetivo de obtener un mayor grado de familiarización con sus términos, lo anterior incluye la LOAFSP y la documentación concerniente a las características de SAREN principalmente la que tiene que ver con su estructura financiera. En segundo lugar se examinaron una serie de temas referentes a la arquitectura de software, dígase patrones, estilos, frameworks, lenguajes de descripción arquitectónica, herramientas así como buenas prácticas que están vigentes en la actualidad. Además se hizo un análisis de las características que debe tener un software para garantizar los objetivos que se plantean en la investigación, que comprende las bases para asegurar las funcionalidades y los temas referentes al desarrollo de las mismas, así como los factores afines con la integración entre los subsistemas que componen la solución. Todos estos puntos avalados en todo

momento por los resultados de la investigación y estudio en profundidad de la Arquitectura de Software y del rol de Arquitecto como máximo exponente de la misma.

Otras tareas importantes fueron las pruebas de los diversos escenarios[1] arquitectónicos obtenidos de los análisis previos, para identificar ventajas y desventajas con el fin de asegurar las características que se consideran más importantes y de mayor prioridad. El resultado de este proceso fue la obtención de una serie de modelos[2] arquitectónicos que finalmente se sometieron a un proceso de evaluación para obtener el mejor candidato.

La investigación se sustenta sobre la hipótesis de que si se define una adecuada línea base para la Arquitectura de un software, partiendo de sus requisitos funcionales y centrada en sus requerimientos arquitectónicos (requisitos no funcionales); entonces se obtendrá un sistema que sea flexible, escalable, reutilizable y mantenible.

El objetivo general que se persigue es definir la línea que deben seguir los procesos de análisis, diseño e implementación, evaluando las mejores prácticas y estilos vigentes en la actualidad referentes a la Arquitectura de software que permita a los desarrolladores y demás involucrados tener una idea clara de lo que se está implementando.

Para dar cumplimiento al objetivo general se han definido los siguientes objetivos específicos:

- Describir la terminología y los conceptos generales que están relacionados con la arquitectura de este sistema.
- Definir las políticas de integración que incluye la Línea Base de la Arquitectura para el sistema.
- Definir componentes, funcionalidades que validen y apuntalen la arquitectura propuesta así como las relaciones que se establecen entre ellos.
- Describir los patrones y frameworks utilizados en el sistema así como su implantación en el mismo.

[1] El nombre se aplica a la descripción de situaciones futuras de alguna magnitud.

[2] Propuesta, normalmente de carácter teórico-práctico, que tiene una serie de características que se consideran dignas de emular. Generalmente, el modelo ilustra una situación deseable para ser analizada y puesta en práctica en un contexto educativo similar, o bien adaptarla a otras características del entorno.

- Describir los elementos que conforman las 4+1 vistas de la arquitectura del sistema.
- Validar la arquitectura obtenida.

Para el cumplimiento de los objetivos descritos anteriormente la investigación paso por las siguientes etapas:

1. Estudio del estado del arte de los procesos relacionados con la Administración Financiera.
2. Estudio de los temas relacionados con la arquitectura de software, que incluye patrones, frameworks, estilos arquitectónicos así como métodos de evaluación arquitectónica.
3. Estudio de los métodos y políticas de integración con otros sistemas.
4. Selección de los frameworks y patrones a utilizar resultado del paso 2.
5. Etapa de pruebas a los modelos arquitectónicos obtenidos.
6. Establecimiento de las pautas de programación y seguridad para un desarrollo limpio, ordenado, sin agujeros ni vulnerabilidades.

La realización de esta investigación puede servir de guía y modelo a otros sistemas similares pues comprende un análisis de los requerimientos que debe cumplir un software de este tipo e incorpora elementos que pueden resultar útiles y prácticos para dar solución a determinados problemas que se dan comúnmente durante el desarrollo.

La Arquitectura obtenida, a pesar de la estructura que presenta, tiene en cuenta posibles limitaciones y agravantes que puede presentar cualquier equipo de desarrollo. En este sentido se logra una combinación de sus elementos de manera tal que cumple los objetivos y asegura los puntos más importantes que se tienen en cuenta durante el desarrollo de todo software, como son la abstracción, la flexibilidad, la escalabilidad, la mantenibilidad y el soporte.

Este trabajo está dividido en 3 capítulos fundamentales.

Capitulo 1: Fundamentación Teórica.

Definición del marco teórico de la investigación que comprende el estudio del arte de la Arquitectura de Software y del rol de arquitecto.

Capitulo 2: Línea Base.

Abarca la descripción del documento de Línea Base de la Arquitectura del Sistema de Administración Financiera que incluye la solución a las interrogantes que puedan surgir al respecto y otros puntos que complementan el trabajo para ganar en claridad y comprensión.

Capitulo 3: Resultados de la Arquitectura Propuesta.

Comprende el análisis de los resultados obtenidos con la Arquitectura propuesta, es decir una valoración de la misma a partir de los objetivos planteados.

Capítulo 1

1. Fundamentación Teórica.

1.1 Introducción.

La Arquitectura de software es un tema sumamente delicado, teniendo en cuenta que define la línea que deben seguir la mayoría de los elementos que intervienen en el desarrollo del software que se esté implementando. El siguiente capítulo aborda una serie de puntos referentes a los conceptos y temas relacionados con la Arquitectura a partir de una profunda investigación y experiencias anteriores que comprende un análisis del estado del arte y otros trabajos o tendencias que se siguen actualmente en el mundo. También se abordan terminologías relacionadas con la Administración Financiera del Sector Público fundamentalmente vinculadas a la Recaudación, al Presupuesto, la Contabilidad etc.

1.2 Arquitectura de software

La Arquitectura de Software es un área de investigación y práctica dentro de la Ingeniería de Software. En particular, la arquitectura de sistemas grandes ha sido objeto de un interés creciente durante la pasada década.

Esta materia no nació en forma espontánea, a partir de 1990 el término Arquitectura de Software comenzó a ganar aceptación y fue elemento de especial atención por parte de la industria y la comunidad científica. Este campo fue creado por necesidad, la realidad marcaba que los sistemas de software estaban creciendo, sistemas de cientos de miles de líneas, o incluso de millones de líneas se estaban volviendo comunes.

Existen tres razones por las cuales la Arquitectura de Software es importante para sistemas de software grandes y complejos:

- Es un vehículo de comunicación entre los stakeholders *(interesados)*. La Arquitectura de Software es una representación abstracta del sistema, a la que la mayoría de los stakeholders, sino todos, pueden utilizar como base para crear entendimiento mutuo, formar consensos y comunicarse entre ellos.
- Es una expresión de las decisiones tempranas del diseño. La Arquitectura de Software de un sistema es el artefacto que permite en forma temprana establecer prioridades entre los diferentes aspectos a ser analizados y es el artefacto con más influencia en la calidad del sistema. Los aspectos de calidad tales como desempeño, seguridad, mantenimiento, costo del esfuerzo del desarrollo actual y costo del esfuerzo del desarrollo futuro, están todos presentes en la Arquitectura.
- Es una abstracción del sistema reusable y transferible. La Arquitectura de Software constituye un modelo relativamente pequeño e intelectualmente comprensible de cómo el sistema está estructurado, y como sus componentes trabajan juntos. Este modelo es transferible entre sistemas. En particular, puede ser utilizado en otros sistemas con requerimientos similares, y puede promover reutilización a gran escala y en una línea de productos de software.

Se ha comentado sobre el surgimiento e importancia de la Arquitectura de Software, ahora resta definirla:

La Arquitectura del Software es la organización fundamental de un sistema formada por sus componentes, las relaciones entre ellos y el contexto en el que se implantarán, y los principios que orientan su diseño y evolución.

[IEEE Std 1471-2000]

Hay algunas implicancias claves en esta definición:

- La Arquitectura es una abstracción de un sistema o sistemas. Esta representa sistemas en términos de componentes abstractos que tienen propiedades externamente visibles y relaciones.
- Como la Arquitectura es abstracta, esta elimina la información local, los detalles de componentes privados no son arquitectónicos.
- Los sistemas están compuestos por muchas estructuras (comúnmente llamadas *vistas*). Una vista por si sola puede representar nada más que una arquitectura trivial. Es más, una arquitectura debería ser descrita por un conjunto de vistas que soporten las necesidades de su análisis y comunicación.

1.2.1 Objetivos

- Permite comprender y mejorar la estructura de las aplicaciones complejas.
- Permite reutilizar dicha estructura (o partes de ella) para resolver problemas similares.
- Posibilita planificar la evolución de la aplicación, identificando las partes mutables e inmutables de la misma, así como los costes de los posibles cambios.
- Permite analizar la corrección de la aplicación y su grado de cumplimiento respecto a los requisitos iniciales.
- Permite el estudio de alguna propiedad específica del dominio.

1.2.2 Características

- La Arquitectura parte del diseño de software.
- Nivel del diseño de software donde se definen la estructura y propiedades globales del sistema.
- Incluye sus componentes, las propiedades observables de dichos componentes y las relaciones que se establecen entre ellos.
- Un aspecto crítico: Una arquitectura errónea puede llevar a problemas incontables.
- Representación de alto nivel de la estructura del sistema describiendo las partes que lo integran.
- Puede incluir los patrones que supervisan la composición de sus componentes y las restricciones al aplicar los patrones.
- Trata aspectos del diseño y desarrollo que no pueden tratarse adecuadamente dentro de los módulos que forman el sistema.

1.2.3 ¿De qué se Ocupa?

- Diseño preliminar o de alto nivel.
- Organización a alto nivel del sistema, incluyendo aspectos como la descripción y análisis de propiedades relativas a su estructura y control global, los protocolos de comunicación y sincronización utilizados, la distribución física del sistema y sus componentes, etc.

- Otros aspectos relacionados con el desarrollo del sistema, su evolución y adaptación al cambio: composición, reconfiguración, reutilización, escalabilidad, mantenibilidad, etc.

1.2.4 ¿De qué no se ocupa?

- Diseño detallado.
- Diseño de algoritmos.
- Diseño de estructuras de datos.

1.3 El rol de Arquitecto de software

El rol de arquitecto de software tiene una serie de responsabilidades dentro del trabajo del proyecto. El arquitecto es el encargado de seleccionar la arquitectura más adecuada para el sistema que responda a las necesidades del usuario, a los requisitos funcionales y no funcionales y además debe ser capaz de lograr los resultados esperados bajo las restricciones dadas.

El arquitecto debe tomar decisiones críticas y cruciales para el sistema que indican muchas veces la dirección que se va a tomar en el proyecto con respecto al mantenimiento y funcionamiento de la aplicación. Para tomar estas decisiones el arquitecto debe apoyarse en el equipo de trabajo y en las restricciones más importantes y cada uno de los cambios que se realicen dejarlos bien explicados y documentados para que las demás personas puedan entender lo que se hizo.

RUP plantea varias responsabilidades para un arquitecto de software:

- El arquitecto posee la responsabilidad técnica del sistema y debe ser capaz de desarrollar e implementar todas las funcionalidades de la aplicación económicamente.
- El arquitecto debe trabajar en conjunto con todos los implicados en el proyecto para obtener experiencia de cada uno.
- Debe ser flexible en caso de existir algún cambio en la aplicación que influya en la arquitectura.

- El arquitecto obtiene una arquitectura sólida después de haber pasado por varias iteraciones del producto, ya en la fase de elaboración se debe tener la arquitectura base y estable, porque este es precisamente el hito de esta fase de RUP.
- En caso de tener un sistema complejo, RUP aconseja que exista un equipo de arquitectos para dividir las tareas porque la arquitectura en este caso puede llegar a ser demasiado compleja.
- El arquitecto debe tener conocimiento y experiencia en el desarrollo de sistemas, porque él es quien va a explicarle la arquitectura a cada uno de los miembros del equipo de trabajo.
- El arquitecto debe tener presente además que la arquitectura está dirigida por los casos de uso.
- Para el arquitecto la línea base de la arquitectura es el documento más importante en su trabajo.
- El arquitecto es responsable de seleccionar los estándares de codificación para el desarrollo del sistema.

1.4 Componentes, conectores y relaciones

Se entiende por *componentes* los bloques de construcción que conforman las partes de un sistema de software. A nivel de lenguajes de programación, pueden ser representados como módulos, clases, objetos o un conjunto de funciones relacionadas (Buschman et al., 1996). La noción de componente puede llegar a ser muy amplia: el término puede ser utilizado para especificar un conjunto de componentes.

Se distinguen tres tipos de componentes (Perry y Wolf, 1992), denominados también *elementos,* que son:

_ *Elementos de Datos:* contienen la información que será transformada.
_ *Elementos de Proceso*: transforman los elementos de datos.
_ *Elementos de Conexión:* llamados también *conectores,* que bien pueden ser elementos de datos o de proceso, y mantienen unidas las diferentes piezas de la Arquitectura.

Una *relación* es la conexión entre los componentes (Buschman et al., 1996). Puede definirse también como una abstracción de la forma en que los componentes interactúan en el sistema a través de los elementos de conexión. Es importante distinguir que una relación se concreta mediante conectores.

Según Bass et al. (1998), en virtud de que está conformado por componentes y relaciones entre ellos, todo sistema, por muy simple que sea, tiene asociada una Arquitectura. Sin embargo, no es necesariamente cierto que esta Arquitectura sea conocida por todos los involucrados en el desarrollo del mismo. Esto hace evidente la diferencia entre la Arquitectura del sistema y su descripción. Esta particularidad propone la importancia de la representación de una arquitectura.

Kazman et al. (2001) presentan la Arquitectura de software como el resultado de decisiones tempranas de diseño, necesarias antes de la construcción del sistema. Según Bass et al. (1998), uno de los aspectos importantes de una arquitectura de software es que, por ser un artefacto de diseño, direcciona atributos de calidad asociados al sistema. Kazman et al. (2001) proponen que las arquitecturas facilitan o inhiben estos atributos. Es por ello que se propone el estudio de los atributos de calidad asociados a la Arquitectura de un sistema de software, y cuál es su impacto sobre el mismo.

1.5 Calidad Arquitectónica.

Barbacci et al. (1995) establecen que el desarrollo de formas sistemáticas para relacionar atributos de calidad de un sistema a su arquitectura provee una base para la toma de decisiones objetivas sobre acuerdos de diseño y permite a los ingenieros realizar predicciones razonablemente exactas sobre los atributos del sistema que son libres de prejuicios y asunciones no triviales. El objetivo de fondo es lograr la habilidad de evaluar cuantitativamente y llegar a acuerdos entre múltiples atributos de calidad para alcanzar un mejor sistema de forma global.

1.5.1 Atributos de Calidad.

Según Barbacci et al. (1995) la *calidad de software* se define como el grado en el cual el software posee una combinación deseada de atributos. Tales atributos son requerimientos adicionales del sistema (Kazman et al., 2001), que hacen referencia a características que éste debe satisfacer, diferentes a los requerimientos funcionales.

Estas características o atributos se conocen con el nombre de *atributos de calidad*, los cuales se definen como las propiedades de un servicio que presta el sistema a sus usuarios (Barbacci et al. 1995).

A grandes rasgos, Bass et al. (1998) establece una clasificación de los atributos de calidad en dos categorías:

- *Observables vía ejecución*: aquellos atributos que se determinan del comportamiento del sistema en tiempo de ejecución. La descripción de algunos de estos atributos se presenta en la tabla 1.
- *No observables vía ejecución*: aquellos atributos que se establecen durante el desarrollo del sistema. La descripción de algunos de estos atributos se presenta en la tabla 2.

Atributo de Calidad	**Descripción**
Disponibilidad *(Availability)*	Es la medida de disponibilidad del sistema para el uso (Barbacci et al.,1995).
Confidencialidad *(Confidentiality)*	Es la ausencia de acceso no autorizado a la información (Barbacci et al.,1995).
Funcionalidad *(Functionality)*	Habilidad del sistema para realizar el trabajo para el cual fue concebido (Kazman et al., 2001).
Desempeño *(Performance)*	Es el grado en el cual un sistema o componente cumple con sus funciones designadas, dentro de ciertas restricciones dadas, como velocidad, exactitud o uso de memoria. (IEEE 610.12). Según Smith (1993), el desempeño de un sistema se refiere a aspectos temporales del comportamiento del mismo. Se refiere a capacidad de respuesta, ya sea el tiempo requerido para responder a aspectos específicos o el número de eventos procesados en un intervalo de tiempo. Según Bass et al. (1998), se refiere además a la cantidad de comunicación e interacción existente entre los componentes del sistema.

Confiabilidad *(Reliability)*	Es la medida de la habilidad de un sistema a mantenerse operativo a lo largo del tiempo (Barbacci et al., 1995).
Seguridad externa *(Safety)*	Ausencia de consecuencias catastróficas en el ambiente. Es la medida de ausencia de errores que generan pérdidas de información (Barbacci et al.,1995).
Seguridad Interna *(Security)*	Es la medida de la habilidad del sistema para resistir a intentos de uso no autorizados y negación del servicio, mientras se sirve a usuarios legítimos (Kazman et al., 2001).

Tabla 1. Descripción de atributos de calidad observables vía ejecución.

Es importante destacar que tener conocimiento de los atributos observables, no necesariamente implica que se satisfacen los atributos no observables vía ejecución. Por ejemplo, un sistema que satisface todos los requerimientos observables puede o no, ser costoso de desarrollar, así como también puede o no ser imposible de modificar. De igual manera, un sistema altamente modificable puede o no, arrojar resultados correctos.

Atributo de Calidad	Descripción
Configurabilidad *(Configurability)*	Posibilidad que se otorga a un usuario experto a realizar ciertos cambios al sistema (Bosch et al., 1999).
Integrabilidad *(Integrability)*	Es la medida en que trabajan correctamente componentes del sistema que fueron desarrollados separadamente al ser integrados. (Bass et al. 1998)
Integridad *(Integrity)*	Es la ausencia de alteraciones inapropiadas de la información (Barbacci et al., 1995).
Interoperabilidad *(Interoperability)*	Es la medida de la habilidad de que un grupo de partes del sistema trabajen con otro sistema. Es un tipo especial de *integrabilidad* (Bass et al. 1998)
Modificabilidad *(Modifiability)*	Es la habilidad de realizar cambios futuros al sistema. (Bosch et al. 1999).
Mantenibilidad	Es la capacidad de someter a un sistema a reparaciones y evolución (Barbacci et

(Maintainability)	al., 1995). Capacidad de modificar el sistema de manera rápida y a bajo costo (Bosch et al. 1999).
Portabilidad *(Portability)*	Es la habilidad del sistema para ser ejecutado en diferentes ambientes de computación. Estos ambientes pueden ser hardware, software o una combinación de los dos (Kazman et al., 2001).
Reusabilidad *(Reusability)*	Es la capacidad de diseñar un sistema de forma tal que su estructura o parte de sus componentes puedan ser reutilizados en futuras aplicaciones (Bass et al. 1998).
Escalabilidad *(Scalability)*	Es el grado con el que se pueden ampliar el diseño arquitectónico, de datos o procedimental (Pressman, 2002).
Capacidad de Prueba *(Testability)*	Es la medida de la facilidad con la que el software, al ser sometido a una serie de pruebas, puede demostrar sus fallas. Es la probabilidad de que, asumiendo que tiene al menos una falla, el software fallará en su próxima ejecución de prueba (Bass et al. 1998).

Tabla 2. Descripción de atributos de calidad no observables vía ejecución.

Bosch (2000) establece que los requerimientos de calidad se ven altamente influenciados por la Arquitectura del sistema. Al respecto, Bass et al. (1998) afirman que la calidad del sistema debe ser considerada en todas las fases del diseño, pero los atributos de calidad se manifiestan de maneras distintas a lo largo de estas fases. De esta forma, establecen que la arquitectura determina ciertos atributos de calidad del sistema, pero existen otros atributos que no dependen directamente de la misma.

Independientemente de esto, es importante tener en cuenta que no puede lograrse la satisfacción de ciertos atributos de calidad de manera aislada. Encontrar un atributo de calidad puede tener efectos positivos o negativos sobre otros atributos que, de alguna manera, también se desean alcanzar (Bass et al., 1998).

En su mayoría, los atributos de calidad se pueden organizar y descomponer de maneras diferentes, en lo que se conoce como *modelos de calidad*. Los modelos de calidad de software facilitan el entendimiento del proceso de la Ingeniería de Software (Pressman, 2002).

1.6 Estilos arquitectónicos y patrones.

De manera concreta, al diseñar una Arquitectura de software se deben crear y representar componentes que interactúen entre ellos y tengan asignadas tareas específicas, además de organizarlos de forma tal que se logren los requerimientos establecidos. Se puede partir con patrones de soluciones ya probados, con la intención de no comenzar de cero las propuestas y utilizar modelos que han funcionado. Estas soluciones probadas se conocen como estilos arquitectónicos, patrones arquitectónicos y patrones de diseño, que van de lo general a lo particular.

El estilo afecta a toda la Arquitectura de software y puede combinarse en la propuesta de solución. Por otra parte, un patrón arquitectónico se enfoca en dar solución a un problema en específico, de un atributo de calidad, y abarca solo parte de la Arquitectura. Un patrón de diseño ayuda a diseñar la estructura interna de un componente específico, es decir, su detalle. Aunque estos estilos y patrones se pueden adoptar, también pueden adaptarse con objeto de lograr alguna funcionalidad concreta esperada.

Bosch (2000) establecen que la imposición de ciertos estilos arquitectónicos mejora o disminuye las posibilidades de satisfacción de ciertos atributos de calidad del sistema. Con esto afirman que cada estilo propicia atributos de calidad, y la decisión de implementar alguno de los existentes depende de los requerimientos de calidad del sistema. De manera similar, plantean el uso de los patrones arquitectónicos y los patrones de diseño para mejorar la calidad del sistema. Al respecto, Buschmann et al. (1996) afirman que un criterio importante del éxito de los patrones – tanto arquitectónicos como de diseño - es la forma en que estos alcanzan de manera satisfactoria los objetivos de la Ingeniería de Software. Los patrones soportan el desarrollo, mantenimiento y evolución de sistemas complejos y de gran escala.

1.6.1 Estilos Arquitectónicos.

En la caracterización de David Garlan, Andrew Kompanek, Ralph Melton y Robert Monroe [GKM+96], también de Carnegie Mellon, se define el estilo como una entidad consistente en cuatro elementos:

1. Un vocabulario de elementos de diseño: componentes y conectores tales como tuberías, filtros, clientes, servidores, parsers, bases de datos, etcétera.
2. Reglas de diseño o restricciones que determinan las composiciones permitidas de esos elementos.
3. Una interpretación semántica que proporciona significados precisos a las composiciones.
4. Análisis susceptibles de practicarse sobre los sistemas construidos en un estilo, por ejemplo análisis de disponibilidad para estilos basados en procesamiento en tiempo real, o detección de abrazos mortales para modelos cliente-servidor.

Los estilos son entidades que ocurren en un nivel sumamente abstracto, puramente arquitectónico, que no coincide ni con la fase de análisis propuesta por la temprana metodología de modelado orientada a objetos (aunque sí un poco con la de diseño), ni con lo que más tarde se definirían como paradigmas de arquitectura, ni con los patrones arquitectónicos. A continuación se analizan estas tres discordancias una por una:

El análisis de la metodología de objetos, tal como se enuncia en [RBP+91] está muy cerca del requerimiento y la percepción de un usuario o cliente técnicamente agnóstico, un protagonista que en el terreno de los estilos no juega ningún papel. En arquitectura de software, los estilos surgen de la experiencia que el Arquitecto posee; de ningún modo vienen impuestos de manera explícita en lo que el cliente le pide.

Los paradigmas como la Arquitectura Orientada a Objetos (p.ej. CORBA), a componentes (COM, JavaBeans, EJB, CORBA Component Model) o a servicios, tal como se los define en [WF04], se relacionan con tecnologías particulares de implementación, un elemento de juicio que en el campo de los estilos se trata a un nivel más genérico y distante, si es que se llega a tratar alguna vez. Dependiendo de la clasificación que se trate, estos paradigmas tipifican más bien como sub-estilos de estilos más englobantes (peer-to-peer, distribuidos, etc) o encarnan la forma de implementación de otros estilos cualesquiera.

Los patrones arquitectónicos, por su parte, se han materializado con referencia a lenguajes y paradigmas también específicos de desarrollo, mientras que ningún estilo presupone o establece preceptivas al respecto. Si hay algún código en las inmediaciones de un estilo, será código del lenguaje de descripción arquitectónica o del lenguaje de modelado; de ninguna manera será código de lenguaje de programación. Lo mismo en cuanto a las representaciones visuales: los estilos se describen mediante simples cajas y líneas, mientras que los patrones suelen representarse en UML [Lar03].

¿Cuántos y cuáles son los estilos, entonces?

La tabla 3 resume los principales estilos arquitectónicos, los atributos de calidad que propician y los atributos que se ven afectados negativamente (atributos en conflicto), de acuerdo a Bass et al. (1998).

Estilo	Descripción	Atributos asociados	Atributos en conflicto
Datos Centralizados	Sistemas en los cuales cierto número de clientes accede y actualiza datos compartidos de un repositorio de manera frecuente.	Integrabilidad Escalabilidad Modificabilidad	Desempeño
Flujo de Datos	El sistema es visto como una serie de transformaciones sobre piezas sucesivas de datos de entrada. El dato ingresa en el sistema, y fluye entre los componentes, de uno en uno, hasta que se le asigne un destino final (salida o repositorio).	Reusabilidad Modificabilidad Mantenibilidad	Desempeño
Máquinas Virtuales	Simulan alguna funcionalidad que no es nativa al hardware o software sobre el que está implementado.	Portabilidad	Desempeño
Llamada y Retorno	El sistema se constituye de un programa principal que tiene el control del sistema y varios subprogramas que se comunican con éste mediante el uso de llamadas.	Modificabilidad Escalabilidad Desempeño	Mantenibilidad Desempeño

Componentes Independientes	Consiste en un número de procesos u objetos independientes que se comunican a través de mensajes.	Modificabilidad Escalabilidad	Desempeño Integrabilidad

Tabla 3. Estilos Arquitectónicos y Atributos de Calidad

1.6.2 Patrones

Buschmann et al. (1996) define *patrón* como una regla que consta de tres partes, la cual expresa una relación entre un contexto, un problema y una solución. En líneas generales, un patrón sigue el siguiente esquema:

- *Contexto.* Es una situación de diseño en la que aparece un problema de diseño
- *Problema*. Es un conjunto de fuerzas que aparecen repetidamente en el contexto
- *Solución*. Es una configuración que equilibra estas fuerzas. Esta abarca:

    - Estructura con componentes y relaciones.
    - Comportamiento a tiempo de ejecución: aspectos dinámicos de la solución, como la colaboración entre componentes, la comunicación entre ellos, etc.

Ventajas:

- Son soluciones simples y técnicas.
- Muy prácticos (deslumbran) y útiles.

Desventajas:

- Son soluciones concretas a problemas concretos.
- Libro de recetas, lo que hace difícil averiguar el patrón adecuado ante un problema concreto. (LePus)
- No dejan huella: En una implementación es difícil saber qué patrón se utilizó. (Ingeniería inversa).
- Facilitan la reutilización del diseño pero no tanto la de la implementación.

- No están formalizados (al menos no de forma simple).

Patrones de Arquitectura.

Partiendo de esta definición, propone los *patrones arquitectónicos* como descripción de un problema particular y recurrente de diseño, que aparece en contextos de diseño específico, y presenta un esquema genérico demostrado con éxito para su solución. El esquema de solución se especifica mediante la descripción de los componentes que la constituyen, sus responsabilidades y desarrollos, así como también la forma como estos colaboran entre sí.

Así mismo, Buschmann et al. (1996) plantean que los patrones arquitectónicos expresan el esquema de organización estructural fundamental para sistemas de software. Provee un conjunto de subsistemas predefinidos, especifica sus responsabilidades e incluye reglas y pautas para la organización de las relaciones entre ellos. Propone que son plantillas para Arquitecturas de software concretas, que especifican las propiedades estructurales de una aplicación - con amplitud de todo el sistema - y tienen un impacto en la Arquitectura de subsistemas. La selección de un patrón arquitectónico es, por lo tanto, una decisión fundamental de diseño en el desarrollo de un sistema de software.

La colección de patrones arquitectónicos debe ser estudiada en términos de factores de calidad e intereses, en anticipación a su uso. Esto quiere decir que un patrón puede ser analizado previamente, con la intención de seleccionar el que mejor se adapte a los requerimientos de calidad que debe cumplir el sistema. De manera similar, Barbacci et al. (1997) proponen que debe estudiarse la composición de los patrones, dado que ésta puede dificultar aspectos como el análisis, o poner en conflicto otros atributos de calidad. La tabla 4 presenta algunos patrones arquitectónicos, además de los atributos que propician y los atributos en conflicto, de acuerdo a Buschmann et al. (1996).

Patrón Arquitectónico	Descripción	Atributos asociados	Atributos en conflicto
Layers	Consiste en estructurar aplicaciones que pueden ser descompuestas en grupos de	Reusabilidad Portabilidad	Desempeño Mantenibilidad

	subtareas, las cuales se clasifican de acuerdo a un nivel particular de abstracción.	Facilidad de Prueba	
Pipes and Filters	Provee una estructura para los sistemas que procesan un flujo de datos. Cada paso de procesamiento está encapsulado en un componente filtro (*filter*). El dato pasa a través de conexiones (*pipes*), entre filtros adyacentes.	Reusabilidad Mantenibilidad	Desempeño
Blackboard	Aplica para problemas cuya solución utiliza estrategias no determinísticas. Varios subsistemas ensamblan su conocimiento para construir una posible solución parcial ó aproximada.	Modificabilidad Mantenibilidad Reusabilidad Integridad	Desempeño Facilidad de Prueba
Broker	Puede ser usado para estructurar sistemas de software distribuido con componentes desacoplados que interactúan por invocaciones a servicios remotos. Un componente *broker* es responsable de coordinar la comunicación, como el reenvío de solicitudes, así como también la transmisión de resultados y excepciones.	Modificabilidad Portabilidad Reusabilidad Escalabilidad Interoperabilidad	Desempeño
Model-View-Controler	Divide una aplicación interactiva en tres componentes. El modelo *(model)* contiene la información central y los datos. Las vistas *(view)* despliegan información al usuario. Los controladores *(controlers)* capturan la entrada del usuario. Las vistas y los controladores constituyen la interfaz del usuario.	Funcionalidad Mantenibilidad	Desempeño Portabilidad
Presentation-Abstraction-	Define una estructura para sistemas de software interactivos de agentes de	Modificabilidad Escalabilidad	Desempeño Mantenibilidad

Control	cooperación organizados de forma jerárquica. Cada agente es responsable de un aspecto específico de la funcionalidad de la aplicación y consiste de tres componentes: presentación, abstracción y control.	Integrabilidad	
Microkernel	Aplica para sistemas de software que deben estar en capacidad de adaptar los requerimientos de cambio del sistema. Separa un núcleo funcional mínimo del resto de la funcionalidad y de partes específicas pertenecientes al cliente.	Portabilidad Escalabilidad Confiablidad Disponibilidad	Desempeño
Reflection	Provee un mecanismo para sistemas cuya estructura y comportamiento cambia dinámicamente. Soporta la modificación de aspectos fundamentales como estructuras tipo y mecanismos de llamadas a funciones.	Modificabilidad	Desempeño

Tabla 4 Patrones arquitectónicos y atributos de calidad.

Con la intención de hacer una comparación clara entre estilo arquitectónico y patrón arquitectónico, la tabla 5 presenta las diferencias entre estos conceptos, construida a partir del planteamiento de Buschmann et al. (1996).

Estilo Arquitectónico	Patrón Arquitectónico
Sólo describe el esqueleto estructural y general *para aplicaciones*	Existen en varios rangos de escala, comenzando con patrones que definen la estructura básica de *una* aplicación
Son independientes del contexto al que puedan ser aplicados	Partiendo de la definición de *patrón*, requieren de la especificación de un contexto del problema

Cada estilo es independiente de los otros	Depende de patrones más pequeños que contiene, patrones con los que interactúa, o de patrones que lo contengan
Expresan técnicas de diseño desde una perspectiva que es independiente de la situación actual de diseño	Expresa un problema recurrente de diseño muy específico, y presenta una solución para él, desde el punto de vista del contexto en el que se presenta
Son una categorización de sistemas	Son soluciones generales a problemas comunes

Tabla 5. Diferencias entre estilo arquitectónico y patrón arquitectónico.

Patrones de Diseño:

Un *patrón de diseño* provee un esquema para refinar los subsistemas o componentes de un sistema de software, o las relaciones entre ellos. Describe la estructura comúnmente recurrente de los componentes en comunicación, que resuelve un problema general de diseño en un contexto particular (Buschman et al., 1996).

Son menores en escala que los patrones arquitectónicos, y tienden a ser independientes de los lenguajes y paradigmas de programación. Su aplicación no tiene efectos en la estructura fundamental del sistema, pero sí sobre la de un subsistema (Buschman et al., 1996), debido a que especifica a un mayor nivel de detalle, sin llegar a la implementación, el comportamiento de los componentes del subsistema. La tabla 6 presenta algunos patrones de diseño, junto a los atributos de calidad que propician y los atributos que entran en conflicto con la aplicación del patrón, según Buschmann et al. (1996).

Patrón de Diseño	Descripción	Atributos asociados	Atributos en conflicto
Whole-Part	Ayuda a constituir una colección de objetos	Reusabilidad	Desempeño

	que juntos conforman una unidad semántica.	Modificabilidad	
Master-Slave	Un componente maestro *(master)* distribuye el trabajo a los componentes esclavos *(slaves)*. El componente maestro calcula un resultado final a partir de los resultados arrojados por los componentes esclavos.	Escalabilidad Desempeño	Portabilidad
Proxy	Los clientes asociados a un componente se comunican con un representante de éste, en lugar del componente en sí mismo.	Desempeño Reusabilidad	Desempeño
Command Procesor	Separa las solicitudes de un servicio de su ejecución. Maneja las solicitudes como objetos separados, programa sus ejecuciones y provee servicios adicionales como el almacenamiento de los objetos solicitados, para permitir que el usuario pueda deshacer alguna solicitud.	Funcionalidad Modificabilidad Facilidad de Prueba	Desempeño
View Handler	Ayuda a manejar todas las vistas que provee un sistema de software. Permite a los clientes abrir, manipular y eliminar vistas. También coordina dependencias entre vistas y organiza su actualización.	Escalabilidad Modificabilidad	Desempeño
Forwarder-Receiver	Provee una comunicación transparente entre procesos de un sistema de software con un modelo de interacción punto a punto *(peer to peer).*	Mantenibilidad Modificabilidad Desempeño	Configurabilidad
Client-Dispatcher-Server	Introduce una capa intermedia entre clientes y servidores, es el componente despachador *(dispatcher)*. Provee una ubicación transparente por medio de un nombre de servicio, y esconde los detalles del	Configurabilidad Portabilidad Escalabilidad Disponibilidad	Desempeño Modificabilidad

	establecimiento de una conexión de comunicación entre clientes y servidores.		
Publisher-Subscriber	Ayuda a mantener sincronizados los componentes en cooperación. Para ello, habilita una vía de propagación de cambios: un editor (*publisher*) notifica a los suscriptores (*suscribers*) sobre los cambios en su estado.	Escalabilidad	Desempeño

Tabla 6. Patrones de diseño y atributos de calidad.

Puntualizar que la mayoría de estos patrones de diseño tiene su base en el catálogo de patrones contenido en el libro “Design Patterns: Elements of Reusable Object-Oriented Software”, también conocido como el LIBRO GOF (Gang-Of-Four Book) [G95]. El cual es considerado uno de los libros más vendidos del mundo de la programación orientación a objetos y recomendado para todos aquellos que se quieren dedicar en serio a la misma. Con la publicación de este libro, los patrones de software adquirieron una gran relevancia.

La figura (ver figura 1) presenta la relación de abstracción existente entre los conceptos de estilo arquitectónico, patrón arquitectónico y patrón de diseño. En ella se representa el planteamiento de Buschmann et al. (1996), que propone el desarrollo de arquitecturas de software como un sistema de patrones, y distintos niveles de abstracción.

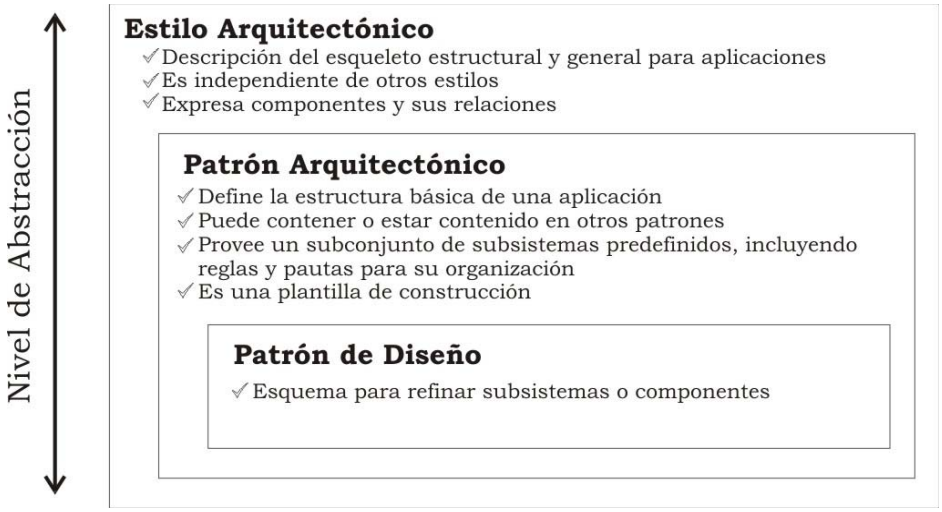


Figura 1. Relación de abstracción entre estilos y patrones

Los estilos y patrones ayudan al Arquitecto a definir la composición y el comportamiento del sistema de software, y una combinación adecuada de ellos permite alcanzar los requerimientos de calidad.

Ahora bien, la organización del sistema de software debe estar disponible para todos los involucrados en el desarrollo del sistema, ya que establece un mecanismo de comunicación entre los mismos. Tal objetivo se logra mediante la representación de la Arquitectura en formas distintas, obteniendo así una descripción de la misma de forma tal que puede ser entendida y analizada por todos los involucrados, con miras a obtener un sistema de calidad. Estas descripciones pueden establecerse a través de las *vistas arquitectónicas*, las *notaciones* como UML y los *lenguajes de descripción arquitectónica* (Bengtsson, 1999).

1.7 Vistas Arquitectónicas

De acuerdo al nivel de responsabilidad dentro del desarrollo de un sistema y la relación que se establezca con el mismo, son muchas las partes involucradas e interesadas en la Arquitectura de software (Kruchten, 1999), a saber:

- El *analista del sistema*, quien la utiliza para organizar y expresar claramente los requerimientos y entender las restricciones de tecnología y los riesgos.
- *Usuarios finales y clientes*, que necesitan conocer el sistema que están adquiriendo.
- El *gerente de proyecto*, que la utiliza para organizar el equipo y planificar el desarrollo.
- Los *diseñadores*, que lo utilizan para entender los principios subyacentes y localizar los límites de su propio diseño.
- Otras *organizaciones desarrolladoras* (si el sistema es abierto), que la utilizan para entender cómo interactuar con el sistema.
- Las *compañías subcontratadas*, que la utilizan para entender los límites de su sección de desarrollo.
- Los A*rquitectos*, quienes velan por la evolución del sistema y la reutilización de componentes.

Todas estas personas deben comunicarse de manera efectiva para discutir y razonar acerca de la Arquitectura, y así alcanzar las metas del desarrollo. En virtud de esto, Kruchten (1999) plantea que debe tenerse una representación del sistema que todos puedan comprender.

Una única representación de la Arquitectura del sistema resultaría demasiado compleja y poco útil para todos los involucrados, pues contendría mucha información irrelevante para la mayoría de estos. Es por ello que se plantea la necesidad de representaciones que contengan únicamente elementos que resultan de importancia para cierto grupo de involucrados.

Buschmann et al. (1996) establece que una *vista arquitectónica* representa un aspecto parcial de una arquitectura de software, que muestra propiedades específicas del sistema. Bass et al. (1998), haciendo uso indistinto de los términos estructura y vista, proponen que las *estructuras arquitectónicas* pueden definirse agrupando los componentes y conectores de acuerdo a la funcionalidad del sistema, sincronización y comunicación de procesos, distribución física, propiedades estáticas, propiedades dinámicas y propiedades de ejecución, entre otras.

Por su parte, Kruchten (1999) define una *vista arquitectónica* como una descripción simplificada o abstracción de un sistema desde una perspectiva específica, que cubre intereses particulares y omite

entidades no relevantes a esta perspectiva. Para la definición de una vista, Kruchten propone la identificación de ciertos elementos, que se mencionan a continuación:

- Punto de vista: involucrados e intereses de los mismos.
- Elementos que serán capturados y representados en la vista y las relaciones entre estos.
- Principios para organizar la vista.
- Forma en que se relacionan los elementos de una vista con otras vistas.
- Proceso a ser utilizado para la creación de la vista.

Kazman et al. (2001), Bass et al. (1998), Hofmeister et al. (2000) y Kruchten (1999), proponen, en función de las características del sistema o del proceso de desarrollo del mismo, distintas vistas arquitectónicas. Es importante resaltar que las vistas propuestas no son independientes entre sí, puesto que son perspectivas distintas de un mismo sistema (Kruchten, 1999). Por tal motivo, las vistas arquitectónicas deben estar coordinadas, de manera tal que al realizar cambios, estos se vean correctamente reflejados en las vistas afectadas, garantizando consistencia entre las mismas.

1.8 Notaciones

1.8.1 Unified Modeling Language (UML)

UML ha conseguido un rol importante en el proceso de desarrollo de software en la actualidad (Booch et al., 1999). La unificación del método de diseño y las notaciones, ha ampliado, entre otras cosas, el mercado de herramientas CASE que soportan el proceso de diseño de Arquitecturas de software. UML ofrece soporte para clases, clases abstractas, relaciones, comportamiento por interacción, empaquetamiento, entre otros. Estos elementos se pueden representar mediante nueve tipos de diagramas, que son: de clases, de objetos, de casos de uso, de secuencia, de colaboración, de estados, de actividades, de componentes y de despliegue.

Bengtsson (1999) presenta las características generales de UML, y las razones por las que resulta interesante su aplicación para efectos de la representación de una Arquitectura de software. Establece

que en UML existe soporte para algunos de los conceptos asociados a las arquitecturas de software, como los componentes, los paquetes, las librerías y la colaboración. UML permite la descripción de componentes en la Arquitectura de software en dos niveles; se puede especificar sólo el nombre del componente o especificar las clases o interfaces que implementan estos.

De igual forma, UML provee una notación para la descripción de la proyección de los componentes de software en el hardware. Esto corresponde a la vista física del modelo 4+1 (Kruchten, 1999). La proyección de los componentes de software permite a los Ingenieros de Software hacer mejores estimaciones cuando se intenta medir la calidad del sistema implementado, lo cual contribuye a la búsqueda de la mejor solución para un sistema específico. Esta notación puede ser extendida con mayor nivel de detalle y los componentes pueden ser conectados entre sí con el uso de las bondades del lenguaje UML.

Los patrones y frameworks están también soportados por UML, mediante el uso combinado de paquetes, componentes y colaboraciones, entre otros. Booch et al. (1999) proponen de forma detallada todos los aspectos que hacen de UML un lenguaje conveniente para la representación de las arquitecturas de software.

Sin embargo pese a que este lenguaje, por todas las facilidades que se han mencionado, puede ser utilizado para modelar arquitecturas, este no abarca la totalidad de los procesos y conceptos relacionados con esta disciplina por lo que resulta insuficiente y poco práctico aplicarlos en algunas de ellas. En este sentido, para satisfacer completamente el modelado de arquitecturas de software existen Lenguajes de Descripción de Arquitecturas (ADL).

1.8.2 Lenguajes de Descripción de Arquitectura (ADLs)

La literatura referida a los ADL es ya de magnitud respetable. Para desarrollar esta sección se han tomado en consideración los estudios previos de Kogut y Clements [KC94, KC95, Vestal [Ves93], Luckham y Vera [LV95], Shaw, DeLine y otros [SDK+95], Shaw y Garlan [SG94, SG95] y Medvidovic [Med96], así como la documentación de cada uno de los lenguajes. Algunos de los ADLs que se revisarán en este estudio son ACME, Darwin, Jacal y UniCon.

ACME.

El proyecto Acme comenzó a principios de 1995 en la Escuela de Ciencias de la Computación de la Universidad Carnegie Mellon. Hoy este proyecto se organiza en dos grandes grupos, que son el lenguaje Acme propiamente dicho y el Acme Tool Developer's Library (AcmeLib). De Acme se deriva, en gran parte, el ulterior estándar emergente ADML. Fundamental en el desarrollo de Acme ha sido el trabajado de destacados arquitectos y sistematizadores del campo, entre los cuales el más conocido es sin duda David Garlan, uno de los teóricos de arquitectura de software más activos en la década de 1990.

Objetivo principal: La motivación fundamental de Acme es el intercambio entre arquitecturas e integración de ADLs. Garlan considera que Acme es un lenguaje de descripción arquitectónica de segunda generación; podría decirse que es de segundo orden: un metalenguaje, una lengua franca para el entendimiento de dos o más ADLs, incluido Acme mismo. Con el tiempo, sin embargo, la dimensión metalingüística de Acme fue perdiendo prioridad y los desarrollos actuales profundizan su capacidad intrínseca como ADL puro.

Darwin.

Darwin es un lenguaje de descripción arquitectónica desarrollado por Jeff Magee y Jeff Kramer [MEDK95, MK96]. Darwin describe un tipo de componente mediante una interfaz consistente en una colección de servicios que son provistos (declarados por ese componente) o requeridos (o sea, que se espera ocurran en el entorno). Las configuraciones se desarrollan instanciando las declaraciones de componentes y estableciendo vínculos entre ambas clases de servicios.

Darwin soporta la descripción de arquitecturas que se reconfiguran dinámicamente a través de dos construcciones: instanciación tardía (lazy) y construcciones dinámicas explícitas. Utilizando instanciación lazy, se describe una configuración y se instancian componentes sólo en la medida en que los servicios que ellos provean sean utilizados por otros componentes. La estructura dinámica explícita, en cambio, se realiza mediante constructos de configuración imperativos. De este modo, la declaración de configuración deviene un programa que se ejecuta en tiempo de ejecución, antes que una declaración estática de la estructura.

Cada servicio de Darwin se modeliza como un nombre de canal, y cada declaración de binding es un proceso que trasmite el nombre del canal al componente que requiere el servicio. En una implementación generada en Darwin, se presupone que cada componente primitivo está implementado en algún lenguaje de programación, y que para cada tipo de servicio se necesita un ligamento (glue) que depende de cada plataforma. El algoritmo de elaboración actúa, esencialmente, como un servidor de nombre que proporciona la ubicación de los servicios provistos a cualquier componente que se ejecute.

Objetivo principal: Como su nombre lo indica, Darwin está orientado más que nada al diseño de arquitecturas dinámicas y cambiantes.

Jacal.

Es un lenguaje de descripción de arquitecturas de software de propósito general creado en la Universidad de Buenos Aires, por un grupo de investigación del Departamento de Computación de la Facultad de Ciencias Exactas y Naturales.

Objetivo principal – El objetivo principal de Jacal es lo que actualmente se denomina “animación” de arquitecturas. Esto es, poder visualizar una simulación de cómo se comportaría en la práctica un sistema basado en la arquitectura que se ha representado.

Más allá de este objetivo principal, el diseño de Jacal contempla otras características deseables en un ADL, como por ejemplo contar con una representación gráfica que permita a simple vista transmitir la arquitectura del sistema, sin necesidad de recurrir a información adicional. Para este fin, se cuenta con un conjunto predefinido (extensible) de conectores, cada uno con una representación distinta.

UniCon.

UniCon (Universal Connector Support) es un ADL desarrollado por Mary Shaw y otros [SDK+95]. Proporciona una herramienta de diseño para construir configuraciones ejecutables basadas en tipos de componentes, implementaciones y “conexiones expertas” que soportan tipos particulares de conectores. UniCon se asemeja a Darwin en la medida en que proporciona herramientas para desarrollar

configuraciones ejecutables de caja negra y posee un número fijo de tipos de interacción, pero el modelo de conectores de ambos ADLs es distinto.

Oficialmente se define como un ADL cuyo foco apunta a soportar la variedad de partes y estilos que se encuentra en la vida real y en la construcción de sistemas a partir de sus descripciones arquitectónicas. UniCon es el ADL propio del proyecto Vitruvius, cuyo objetivo es elucidar un nivel de abstracción de modo tal que se pueda capturar, organizar y tornar disponible la experiencia colectiva exitosa de los arquitectos de software.

Objetivo principal: El propósito de UniCon es generar código ejecutable a partir de una descripción, a partir de componentes primitivos adecuados. UniCon se destaca por su capacidad de manejo de métodos de análisis de tiempo real a través de RMA (Rate Monotonic Analysis).

A pesar de las potencialidades de los ADLs para representar arquitecturas de una manera más completa. El presente trabajo opta por UML puesto que es el lenguaje establecido por el proyecto general, abarca la mayoría de la representación arquitectónica que el sistema necesita y además porque UML es el lenguaje del que mayor conocimiento existe por parte del equipo de arquitectura.

1.9 Modelos y arquitecturas de referencia

Los modelos particularizan un estilo imponiendo una serie de restricciones sobre el mismo y realizando una descomposición y definición estándar de componentes.

- OSI (*Open System Interconection*) que particulariza el estilo de organización en capas, con 7 niveles.
- RM-ODP para el diseño de sistema distribuidos y abiertos (ISO/ITU-T, 1995). Hace transparente la heterogeneidad de plataformas, sistemas operativos, lenguajes. Se basa en el estilo de componentes independientes.
- CCM, COM, JavaBeans - Sistemas basados en el intercambio de eventos.

1.9.1 Arquitectura JEE

En la Arquitectura JEE se contemplan cuatro capas, en función del tipo de servicio y contenedores:

- Capa de **cliente**, también conocida como capa de presentación o de aplicación. Nos encontramos con componentes Java (applets o aplicaciones) y no-Java (HTML, JavaScript, etc.).
- Capa **Web**. Intermediario entre el cliente y otras capas. Sus componentes principales son los servlets y las JSP. Aunque componentes de capa cliente (applets o aplicaciones) pueden acceder directamente a la capa EJB, lo normal es que Los servlets/JSPs pueden llamar a los EJB.
- Capa **Enterprise JavaBeans**. Permite a múltiples aplicaciones tener acceso de forma concurrente a datos y lógica de negocio. Los EJB se encuentran en un servidor EJB, que no es más que un **servidor de objetos distribuidos**. Un EJB puede conectarse a cualquier capa, aunque su misión esencial es conectarse con los sistemas de información empresarial
  (un gestor de base de datos, ERP, etc.)
- Capa de **sistemas de información empresarial**.

La visión de la Arquitectura es un **esquema lógico**, no físico. Cuando hablamos de capas nos referimos sobre todo a **servicios** diferentes (que pueden estar físicamente dentro de la misma máquina e incluso compartir servidor de aplicaciones y JVM).

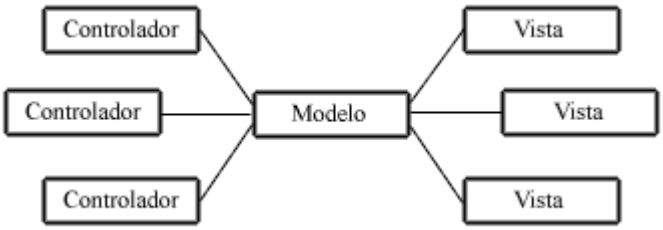
1.9.2 Arquitectura MVC

La Arquitectura MVC (*Model/View/Controller*) fue introducida como parte de la versión Smalltalk-80[3]. Fue diseñada para reducir el esfuerzo de programación necesario en la implementación de sistemas múltiples y sincronizados de los mismos datos. Sus características principales son que el Modelo, las Vistas y los Controladores se tratan como entidades separadas; esto hace que cualquier cambio producido en el Modelo se refleje automáticamente en cada una de las Vistas.

[3] Del lenguaje de programación Smalltalk.

Este modelo de Arquitectura se puede emplear en sistemas de representación gráfica de datos, o en sistemas CAD[4], en donde se presentan partes del diseño con diferente escala de aumento, en ventanas separadas.

En la siguiente figura, vemos la Arquitectura MVC en su forma más general. Hay un Modelo, múltiples Controladores que manipulan ese Modelo, y hay varias Vistas de los datos del Modelo, que cambian cuando cambia el estado de ese Modelo.



Este modelo de arquitectura presenta varias ventajas:

- Hay una clara separación entre los componentes de un programa; lo cual nos permite implementarlos por separado
- Hay un API[5] muy bien definido; cualquiera que use el API, podrá reemplazar el Modelo, la Vista o el Controlador, sin aparente dificultad.
- La conexión entre el Modelo y sus Vistas es dinámica; se produce en tiempo de ejecución, no en tiempo de compilación.

Al incorporar el modelo de arquitectura MVC a un diseño, las piezas de un programa se pueden construir por separado y luego unirlas en tiempo de ejecución. Si uno de los Componentes, posteriormente, se observa que funciona mal, puede reemplazarse sin que las otras piezas se vean afectadas. Este escenario contrasta con la aproximación monolítica típica de muchos programas Java. Todos tienen un

[4] Del inglés **C**omputer **A**ided **D**esign. CAD es todo sistema informático destinado a asistir al diseñador en su tarea específica.

[5] Interfaz de programación de aplicaciones (Applications Programming Interface): una serie de funciones que están disponibles para realizar programas para un cierto entorno.

Frame que contiene todos los elementos, un controlador de eventos, un montón de cálculos y la presentación del resultado. Ante esta perspectiva, hacer un cambio aquí no es nada trivial.

1.10 Frameworks/ Marcos de trabajo

La conferencia de la Universidad de Málaga "Arquitectura, marcos de trabajo y patrones", Canal (2005) establece una serie de características y clasificaciones para los frameworks:

- Definen una arquitectura adaptada a las particularidades de un determinado dominio de aplicación, definiendo de forma abstracta una serie de componentes y sus interfaces y estableciendo las reglas y mecanismos de interacción entre ellos.
- Suele incluirse la implementación de algunos de los componentes e incluso varias implementaciones alternativas.
- El usuario

    - Selecciona, instancia, extiende y reutiliza los componentes del marco.
    - Completa la Arquitectura con nuevos componentes específicos dentro de las relaciones estructurales del marco.

- Básicamente se presentan como un diseño reutilizable de todo o parte de un sistema, representado por un conjunto de componentes abstractos y la forma en que los componentes interactúan.
- Una alternativa es verlos como un esqueleto de una aplicación que debe ser adaptado por el programador según sus necesidades concretas.
- Un marco de trabajo define el patrón arquitectónico que relaciona los componentes de un sistema.
- Presentan dos niveles:

    - Especificación de la Arquitectura marco.
    - Implementación del marco de trabajo (normalmente un lenguaje orientado a objetos).

- Al desarrollo de aplicaciones a partir de un marco de trabajo se le denomina extensión del marco:

- Puntos de entrada (hot spots): Componentes o procedimientos cuya implementación final depende de la aplicación concreta.
- Definidos por el diseñador del marco para que sean la forma natural de la extensión del mismo.

Dependiendo de la extensión tenemos:

Marcos de trabajo de caja blanca

Que se extienden mediante herencia, concretamente mediante la implementación de las clases y métodos abstractos definidos como puntos de entrada. Se tiene acceso al código del marco y se permite reutilizar la funcionalidad de sus clases mediante herencia y redefinición de métodos.

Marcos de trabajo de caja de cristal

Admiten la inspección de su estructura e implementación, pero no su modificación ni extensión, excepto en los puntos de entrada.

Marcos de trabajo de caja gris

Los puntos de entrada no son simplemente métodos abstractos, de los que se declara meramente su signatura, sino que se definen por medio de un lenguaje de especificación de alto nivel los requisitos que deben cumplirse a la hora de implementarse.

Marcos de trabajo de caja negra

Su instanciación se realiza por medio de composición y delegación, en lugar de utilizar la herencia. Los puntos de entrada se definen por medio de interfaces que deben implementar los componentes que extiendan el marco.

Dependiendo de su aplicabilidad tenemos:

Marcos de trabajo horizontal

Válidos para todos los dominios de aplicación concretados en un aspecto del sistema.
Infraestructuras de comunicación, interfaces de usuario, entornos visuales, etc.

Marcos de trabajo distribuido (Middelware Application Frameworks) o Plataforma de Componentes Distribuidos para integrar componentes distribuidos.

Aíslan las dificultades conceptuales y técnicas del desarrollo de aplicaciones distribuidas basadas en componentes (CORBA, Active X/OLE/COM, JavaBeans, ACE, Hector, Aglets).

Marcos de trabajo vertical

Desarrollados específicamente para un dominio de aplicación.
Telecomunicaciones, fabricación, servicios telemáticos y multimedia, etc.

Ventajas de la utilización de marcos de trabajo:

- Son composicionales.
- Son el grado más alto de reutilización dentro del desarrollo de software.
- El diseño arquitectónico se reutiliza.
- Reducción de costes/Mejora de la calidad.
- Están intrínsecamente unidos a los componentes ya que además de proporcionar funcionalidad de los componentes permiten la composición entre ellos de forma consistente.

1.10.1 Spring.net

Es un framework/marco de trabajo que ayuda a construir aplicaciones empresariales sobre la plataforma .NET, basado en el framework Spring de Java del cual hereda los principales conceptos, tales como Inyección de Dependencia y Programación Orientada a Aspectos (AOP por sus siglas en ingles).

La mayoría de las aplicaciones empresariales están compuestas por una variedad de capas físicas y por un conjunto de funcionalidades. Independientemente de la Arquitectura empleada y atendiendo a la gran variedad de objetos internos y externos, que se manejan en cada nivel, la utilización de spring.net resulta una buena decisión.

Ventajas

Precisamente, al utilizar Spring.net se aseguran una serie de aspectos muy importantes en las aplicaciones, aspectos que comprende el framework y no requieren casi ningún código por parte del equipo de desarrollo.

- Implementación de Patrones de Diseño: La utilización de patrones en una arquitectura resulta una decisión muy acertada y eficiente puesto que constituyen buenas prácticas y soluciones a problemas muy comunes en el desarrollo de cualquier aplicación. Spring.net implementa una gran variedad de estos patrones de diseño de una forma bastante sencilla y práctica. Por ejemplo tenemos *Factory, Abstract Factory, Builder, Decorator, and Service Locator* por solo mencionar algunos.
- Inversión del Control (IoC)[6]: Delega la responsabilidad de instanciar los objetos en un archivo de configuración XML, esto resulta muy útil puesto que evita cualquier dependencia que pueda existir entre los objetos, más si son de capas o aplicaciones externas. El nombre viene precisamente de que los objetos generalmente son instanciados antes de ser utilizados en el momento en que se carga el XML.
- Basado en Programación Orientada a Aspectos (AOP): Ayuda a modificar dinámicamente el modelo estático para incluir el código requerido para cumplir los requerimientos secundarios sin tener que modificar el modelo estático original. Mejor aún, normalmente se puede tener este código adicional en una única localización en vez de tenerlo repartido por el modelo existente, como se haría si estuviéramos trabajando Orientado a Objetos (OO).

[6] Cuando hablamos acerca de Inversión de control: *"La pregunta es, ¿Qué aspecto del control se invierte?"*. Después de platicar sobre el término Inversión de control se sugiere renombrar el nombre del patrón, o al menos darle un nombre más explicito, y se comenzó a usar el término *inyección de dependencia*.

Desventajas

- La configuración de Spring está inflada. Configurar mediante XML es muy laborioso e introduce errores al no tener una herramienta para hacerlo.
- Perdida de las ventajas del tipado fuerte al configurar con XML.
- Spring no es ligero. Es decir tiende a aumenta el tiempo de respuesta de la aplicación lo que repercute en el rendimiento de la misma. Esto se debe a que en el momento de la llamada el framework interpreta todas las instancias que tiene configuradas en el fichero.
- 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.

1.10.2 NHibernate

Utilizar un framework/ marco de trabajo de Object Relational Mapping (ORM) para resolver la lógica de persistencia es una técnica madura que ha demostrado ser extremadamente superior a las técnicas tradicionales basadas en el uso de APIs como ADO.NET.

Esta sección pretende argumentar esta idea a partir el uso de NHibernate, un framework de ORM open-source para .NET basado en el excelente framework Hibernate surgido en la comunidad Java en el año 2002.

Ventajas.

Una estrategia elegante y compatible con las buenas prácticas de diseño pregonadas en estos últimos 10 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, en este caso NHibernate ofrece entre otras las siguientes ventajas:

- Transparente: Los objetos del dominio desconocen todo lo concerniente a 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 puede cargar jerarquías de objetos en forma polimórfica.
- Soporte de los 3 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: Permite cargar objetos que solo contengan la clave del objeto completo.
- Soporte de caching: En el contexto de una transacción, puedo disminuir la cantidad de veces que voy 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, etc. simplemente cambiando la configuración correspondiente.

1.11 Evaluando una Arquitectura de Software

Un vez generada y documentada la Arquitectura de software, ésta debe evaluarse para verificar que cumpla con todos los requerimientos; específicamente con los atributos de calidad establecidos. Dicha evaluación puede realizarse mediante técnicas cualitativas, como cuestionarios o escenarios, o a través de técnicas cuantitativas, como simulaciones o modelos matemáticos. En la literatura existen diferentes métodos de evaluación para verificar desde múltiples atributos de calidad hasta algunos en específico.

Ejemplos de estos métodos de evaluación son ATAM, ABAS, SAAM, SNA, ALMA, RMA, teoría de colas, teoría de confiabilidad, entre otras.

Cuanto más temprano se encuentre un problema en un proyecto de software, mejor. El costo de arreglar un error durante las fases de requerimientos o diseño, es mucho menor al costo de arreglar ese mismo error en la fase de verificación. Dado que la Arquitectura es un producto temprano de la fase de diseño, esta tiene un profundo efecto en el sistema y en el proyecto.

Una mala Arquitectura puede llevar a un proyecto al fracaso. Todos los requerimientos de calidad pueden quedar insatisfechos.

La Arquitectura también determina la estructura del proyecto: configuración, agenda y presupuesto, alcance, entre otros aspectos. Es mejor cambiar la Arquitectura antes que otros artefactos, que están basados en ella, se estabilicen.

Realizar una evaluación de la arquitectura es la manera más económica de evitar desastres.

1.11.1 ¿Cuándo una Arquitectura puede ser evaluada?

Generalmente, la evaluación de la Arquitectura ocurre después que esta ha sido especificada, pero antes que empiece la implementación. En un proceso iterativo y/o incremental, la evaluación se puede realizar al final de cada ciclo. Sin embargo, uno de los atractivos de la evaluación de arquitecturas es que se puede efectuar en cualquier etapa de la vida de una arquitectura. En particular, existen dos variaciones útiles: temprana y tardía.

1.11.2 Evaluación temprana

La evaluación no tiene porque esperar a que la Arquitectura este totalmente especificada. Esta puede ser utilizada en cualquier etapa del proceso de creación de la Arquitectura, para examinar las decisiones arquitectónicas ya tomadas y decidir entre las opciones que están pendientes.

Por supuesto, la completitud y fidelidad de la evaluación es directamente proporcional a la completitud y fidelidad de la descripción de la Arquitectura.

1.11.3 Evaluación tardía

Esta variación toma lugar no solo cuando la Arquitectura está terminada, también cuando la implementación esta completa. Este caso ocurre cuando la organización hereda un sistema legado. La técnica para evaluar un sistema legado es la misma que para evaluar un sistema recién nacido. Una evaluación es útil para entender el sistema legado, y saber si este cumple con los requerimientos de calidad y comportamiento.

En general, una evaluación debe realizarse cuando hay suficiente de la Arquitectura como para justificarlo. Una buena regla sería: *realizar una evaluación cuando el equipo de desarrollo empieza a tomar decisiones que dependen de la Arquitectura y el costo de deshacerlas sobrepasa al costo de realizar una evaluación*.

1.11.4 ¿Qué resultado produce la evaluación de una Arquitectura?

En términos concretos, la evaluación de la Arquitectura produce un informe, la forma y contenido del mismo varía según el método utilizado. En particular, produce repuestas a dos tipos de preguntas:

- ¿Es esta Arquitectura adecuada para el sistema para la cual fue diseñada?
- ¿Cuál de dos o más Arquitecturas propuestas es la más adecuada para el sistema?

Decimos que una Arquitectura es adecuada cuando cumple dos criterios:

- El sistema resultante de ella cumple con los objetivos de calidad. No todas las propiedades de calidad del sistema son resultado directo de la Arquitectura, pero muchas lo son.
- El sistema puede ser construido con los recursos con que se cuenta: el plantel, el presupuesto, el sistema legado (si hay), entre otros. Esto es, la Arquitectura es construible.

Un resultado que también produce la evaluación de una Arquitectura es la captura y priorización de las metas que la arquitectura debe cumplir para poder ser considerada adecuada.

La evaluación de una Arquitectura no produce resultados cuantitativos. No es de interés, por ejemplo, evaluar el performance en cantidad de transacciones por segundo a esta altura, dado que el sistema no está construido aún. Lo que interesa, en un espíritu de mitigación de riesgos, es aprender cómo un atributo de calidad es afectado por una decisión de diseño arquitectónico, para que de esta forma se pueda estudiar con cuidado dicha decisión.

Una evaluación Arquitectónica dice si una arquitectura es adecuada respecto a un conjunto de metas, y problemática con respecto a otro conjunto de metas. En ocasiones, las metas pueden ser contradictorias entre ellas, o algunas ser más importantes que otras. El director de proyecto es quien deberá tomar la decisión si la Arquitectura evalúa bien o mal en las distintas áreas. La evaluación ayuda a encontrar debilidades, no dirá “si” o “no”, “bien” o “mal”, “6 en 10”, dirá donde están los riesgos.

1.12 La Administración Financiera

La Administración Financiera es la técnica que tiene por objeto la obtención, control y el adecuado uso de recursos financieros que requiere una empresa, así como el manejo eficiente y protección de los activos de la empresa.

Es la disciplina que se encarga del estudio de la teoría y de su aplicación en el tiempo y en el espacio, sobre la obtención de recursos, su asignación, distribución y minimización del riesgo en las organizaciones a efectos de lograr los objetivos que satisfagan a la coalición imperante.

1.12.1 Objetivos.

- Planear el crecimiento de la empresa, tanto táctica como estratégica.
- Captar los recursos necesarios para que la empresa opere en forma eficiente. Asignar recursos de acuerdo con los planes y necesidades de la empresa.

- Optimizar los recursos financieros.
- Minimizar la incertidumbre de la inversión. Maximización de las utilidades Maximización del Patrimonio Neto Maximización del Valor Actual Neto de la Empresa Maximización de la Creación de Valor.

1.12.2 Funciones

La mayor parte de las decisiones empresariales se miden en términos financieros. La importancia de la función administrativa financiera depende del tamaño de la empresa, en compañías pequeñas, la función financiera la desempeña el departamento de contabilidad. Al crecer una empresa es necesario un departamento separado ligado al presidente de la compañía (o director general) por medio de un vicepresidente de finanzas, conocido como gerente financiero. El tesorero y el contralor se reportan al vicepresidente de finanzas. El tesorero coordina las actividades financieras, tales como: planeación financiera y percepción de fondos, administración del efectivo, desembolsos de capital, manejo de créditos y administración de la cartera de inversiones. El contador se ocupa de actividades contables, administración fiscal, procesamiento de datos así como la contabilidad financiera y de costos. La función administrativa financiera está muy ligada con la economía y la contaduría.

1.13 Propuesta de Solución: Arquitectura Software Administración Financiera

Para darle solución a la situación problemática planteada, se define la Línea Base de la Arquitectura para establecer la infraestructura que soporta los procesos lógicos que se llevan a cabo durante el desarrollo del Sistema de Administración Financiera de SAREN en la República Bolivariana de Venezuela. Específicamente esta propuesta abarca diez módulos principales que responden al negocio y dos independientes que introduce la arquitectura para el desarrollo de componentes, servicios y la integración.

La arquitectura obtenida está dirigida a alcanzar un sistema flexible, escalable, mantenible. Para lograr estos objetivos se apoya en una serie de recursos. A continuación se relacionan algunos de ellos:

- Implementación del Patrón Fachada.
- Implementación del Patrón Proxy.

- Implementación del Patrón MVC.
- Implementación del Patrón Reflection.
- Utilización de un estilo y patrón multicapa.
- Utilización del framework Spring.net.
- Utilización del framework NHibernate.

La incorporación de todos estos elementos es muy particular de la solución, con su utilización se busca respetar las prioridades fundamentales del cliente y la Arquitectura. En este sentido se tienen en cuenta los requisitos funcionales, los no funcionales y las disponibilidades del proyecto. Los aportes más significativos son:

- Utilizar spring.net de una forma parcial, centrado fundamentalmente en su filosofía de inyección de dependencias para lograr una mayor abstracción entre las capas.
- Agregar funcionalidades y métodos a NHibernate, relacionados fundamentalmente con el trabajo con procedimientos almacenados, que resulten prácticos para el desarrollo gracias a que es open-source.
- Hacer uso de las interfaces como recurso para mantener la menor dependencia posible y facilitar posteriores mantenimientos al software.

1.14 Conclusiones

Una vez descritos los procesos que se analizan dentro del marco de este trabajo, se pude concluir que la arquitectura de software es muy importante para garantizar un flujo de trabajo adecuado y preciso. En efecto, durante su definición se tienen en cuenta una serie de aspectos que contemplan el aseguramiento de los requisitos funcionales y no funcionales del software; en este caso vinculado a los procesos relacionados con la Administración Financiera del sector público en SAREN.

El diseño de una Arquitectura de software debe considerarse una parte fundamental, crítica e imprescindible en el desarrollo de un sistema de software, ya que es precisamente en esta fase en donde recae toda la creatividad, experiencia y creación de la propuesta de solución que más se adecue a las necesidades del cliente y le permita lograr sus objetivos.

Se trata de un concepto que nació hace ya varios años, no obstante, emerge recientemente como concepto formal, como un proceso de Ingeniería. En general, la mayoría no tiene un método bien definido para desarrollar la industria de software y aunque no es una tarea sencilla el adoptar la creación de una Arquitectura de software, se requiere romper paradigmas en la forma de trabajo de las personas. Los profesionales de la industria de software y específicamente, quienes están dedicados al diseño de sistemas, deben capacitarse ampliamente en el campo de la Arquitectura de software para cumplir con esta importante etapa del ciclo de vida de un sistema.

Capítulo 2

2. Línea Base.

2.1 Introducción.

Este capítulo tiene como propósito describir los puntos más significativos para la Arquitectura del software de Administración Financiera. Es decir, ofrecer un marco idóneo para unificar los componentes que pertenecen a la solución bajo una misma línea a través de los diferentes artefactos que resultan más representativos, acorde a la metodología RUP (Rational Unified Process), para el proceso de desarrollo. Lo anterior incluye 4 de las 5 vistas arquitectónicas donde se recogen casos de usos, paquetes, subsistemas, clases, componentes y demás elementos que avalen y apoyen la solución.

2.1 Alcance

La Arquitectura del software de Administración Financiera incluye patrones arquitectónicos, patrones de diseño y el uso de frameworks/marcos de trabajo, que proporcionan una serie de funcionalidades que representan significativos pasos de avance en el desarrollo del sistema. Asimismo se propone un desarrollo basado en componentes, conectores y subsistemas para su desarrollo y mantenimiento.

La Línea Base abarca además los temas referentes a la integración entre los módulos que componen la solución así como mecanismos, estructuras y políticas que son necesarios para la correcta ejecución de la misma. La Arquitectura se presenta desde dos enfoques: uno **Horizontal** que comprende la comunicación entre los módulos y las prioridades para su desarrollo atendiendo a las funcionalidades que se quieran ir obteniendo en cada iteración y otra **Vertical** que se centra en las especificidades que tiene la Arquitectura para cada módulo de manera individual.

Independientemente de que el software en general abarca 10 módulos que responden al negocio, este documento se limita solamente a una primera iteración que incluye 4 de ellos. No obstante en las restantes se tiene que ir refinando y actualizando el documento hasta obtener una versión final.

2.2 Concepciones Generales.

La solución de software está dirigida a la automatización de los procesos administrativos y financieros según la distribución adoptada actualmente a tal efecto en SAREN (Servicio Autónomo de Registros y Notarias) en la República Bolivariana de Venezuela (ver figura 2).

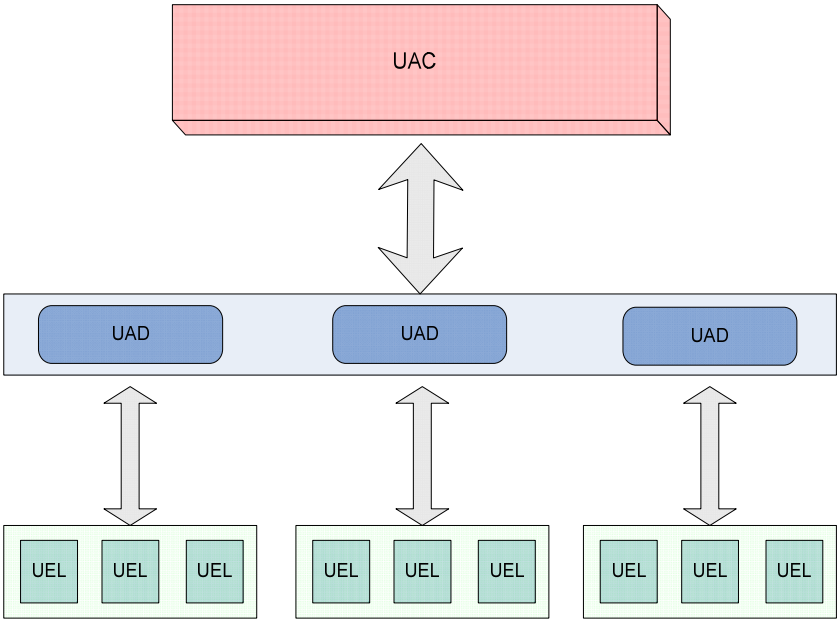


Figura 2. Estructura Financiera de SAREN.

Algunos conceptos y términos que se han utilizado derivan una serie de palabras que resultan claves para comprender el proceso y las diferentes situaciones que se puedan encontrar en este documento.

2.2.1 Definiciones, Acrónimos y Abreviaturas

UE: Unidad Ejecutora.
UAC: Unidad Administradora Central.
UAD: Unidad Administradora Desconcentrada.
UEL: Unidad Ejecutora Local.
SAREN: Servicio Autónomo de Registros y Notarías.
LOAFSP: Ley Orgánica de la Administración Financiera del Sector Público.
MPPRIJ: Ministerio del Poder Popular para las Relaciones Interiores y de Justica.

2.3 Descripción Arquitectónica

El Sistema de Administración Financiera consta de 10+2 módulos, lo anterior significa que son diez módulos que responden a los requisitos funcionales (Negocio) y dos que funcionan independientes ("MóduloComún" y "ArquitecturaBase"). El primero se utiliza para la construcción y desarrollo de los servicios y componentes que apuntalan la Arquitectura. El segundo es donde se implementan todos los componentes de negocio que involucren la totalidad de la solución y los mecanismos de integración del sistema según la Arquitectura propuesta.

El Sistema está concebido para desempeñarse como una aplicación de escritorio y desarrollado sobre la plataforma .Net. Cada uno de los subsistemas y funcionalidades que lo componen estarán disponible en la aplicación dependiendo de los requerimientos y competencia que tenga la UE donde se esté implantando. Es decir, en las UEL, UAD o UAC, la aplicación brindará y garantizará la configuración e infraestructura necesaria para establecer y definir los elementos de la solución que son del dominio de la misma (requisitos funcionales y no funcionales).

El sistema forma parte de la solución general para automatizar los Registros y Notarias en la República Bolivariana de Venezuela. Por tanto, toma como punto de partida la Arquitectura Base General de dicha solución y de ella hereda algunos elementos tales como: la topología de red (ver Anexo 1) y su infraestructura, la Arquitectura de Datos (ver Anexo 2), los mecanismos de seguridad (basada en roles), el lenguaje de programación (C Sharp), algunos puntos del modelo de despliegue (ver Anexo 3) y la utilización de un framework que sustenta el desarrollo basado en capas.

Sin embargo debido a la magnitud del Sistema de Administración Financiera y a partir de experiencias pasadas, la Línea Base para este software rompe con la estructura horizontal y vertical que presenta la Arquitectura general y surge como una propuesta que se acerca más a las necesidades del producto y a las particularidades de la estructura financiera (ver figura 2) que presenta la institución.

La aplicación tiene que coexistir e interactuar con las soluciones de software desarrolladas para los Registros Públicos y Mercantiles. Por tanto, la Arquitectura también incluye los mecanismos de integración con dichos sistemas y dada las características del entorno donde se desenvuelve, su estructura es lo suficientemente flexible como para soportar cualquier interacción con otro módulo o subsistema que pueda surgir sin repercutir significativamente en el código fuente de la misma.

2.4 Representación Arquitectónica

Como se ha mencionado la Arquitectura del sistema se presenta desde dos enfoques:

2.4.1 Enfoque Horizontal

Para dar respuesta a todos los procesos y funcionalidades que se quieren acometer, el sistema de Administración Financiera comprende 10 módulos que eventualmente pueden interactuar para satisfacer determinado requerimiento:

- Administración.
- Presupuesto.
- Contabilidad.
- Recaudación.
- Requisiciones.
- Compras y Servicios.
- Retenciones.
- Tesorería.
- Fondos en Anticipo.

- Fondos de Caja Chica.

Teniendo en cuenta las dimensiones del software, se hace imprescindible realizar iteraciones sobre el documento de Línea Base que puedan marcar hitos en el proceso de desarrollo y por tanto obtener una medida de la prioridad y el avance que se vaya alcanzando. A partir de este análisis y tomando como punto de partida los requisitos que incorporan los módulos de manera individual y en conjunto, se debe dividir el proceso de desarrollo en 3 iteraciones.

Sin embargo, debido a que este documento se centra en el resultado de la primera iteración, se hace hincapié solamente en los principales requisitos funcionales que se obtienen en la misma. En posteriores iteraciones se irán incorporando los restantes.

Iteración 1:

Comprende el desarrollo del módulo de Administración, Presupuesto, Recaudación y Contabilidad.

Según RN-AFDR-02 Documento de Requerimientos del módulo Administración, el sistema debe permitir:

- Generar las propiedades de la UAC.
- Relacionar las UEL que pertenecen a la UAC por su dependencia financiera.
- Generar las propiedades de la UAC.
- Relacionar las UEL que pertenecen a la UAD por su dependencia financiera.
- Capturar las UEL con sus datos.
- Relacionar las UEL con el codificador de oficinas de Registros y Notarías.
- Definir el año a utilizar
- Registra las UEL dependientes.
- Crear coordinadores a las UEL.
- Gestionar las monedas que se utilizarán en el sistema.
- Gestionar el Plan de Cuentas Patrimoniales establecido en el país.
- Gestionar el Clasificador Presupuestario.
- Gestionar el Clasificador Económico.

- Relacionar las cuentas Patrimoniales con las Presupuestarias.
- Relacionar las cuentas Patrimoniales con las Económicas.
- Gestionar el Codificador de Bienes y Servicios establecido.
- Relacionarlo con el de Unidad de Medida.
- Gestionar el codificador de Unidad de Medida.
- Gestionar el codificador de Ejercicios Fiscales.
- Gestionar el codificador de Períodos Contables por cada Ejercicio fiscal.
- Gestionar el codificador de bancos.
- Gestionar el codificador de sucursales bancarias por cada banco.
- Gestionar las monedas que se utilizarán en el sistema.

Según RN-AFDR-03 Documento de Requerimientos del módulo Presupuesto, el sistema debe permitir:

- Gestionar las fuentes de financiamiento.
- Crear un proyecto nuevo.
- Gestionar las acciones específicas asociadas al proyecto que se desea crear.
- Gestionar las UEL asociadas a cada Acción Específica.
- Gestionar las fuentes de financiamiento asociadas a cada Acción Específica.
- Gestionar las fuentes de financiamiento del proyecto que se esté creando.
- Crear una Acción Centralizada.
- Gestionar las acciones específicas asociadas a la Acción Centralizada que se desea crear.
- Gestionar los datos de la Acción Especifica que se desea crear.
- Gestionar las partidas presupuestarias asociadas a cada Acción Especifica.
- Capturar los Indicadores Generales del Anteproyecto de Presupuesto.
- Gestionar las Fuentes de Financiamiento del Anteproyecto de Presupuesto.
- Formular el Presupuesto por Proyecto.
- Formular el Presupuesto por Acción Centralizada
- Gestionar formulación de un proyecto seleccionado.
- Configurar las Acciones Específicas de un proyecto.
- Asignar las Fuentes de Financiamiento de una Acción Específica.

- Asignar las partidas presupuestarias a la UEL seleccionada para realizar la Asignación Presupuestaria del Gasto.
- Capturar la Distribución Física y Financiera para realizar la Distribución Anual de la UEL seleccionada.
- Gestionar formulación de una Acción Centralizada seleccionada.
- Configurar las Acciones Específicas de una Acción Centralizada.
- Asignar las Fuentes de Financiamiento de una Acción Específica.
- Gestionar una UEL.
- Modificar una partida presupuestaria.
- Realizar un desglose de la partida presupuestaria seleccionada por Bienes/Servicios, por Entes o por un monto general.
- Configurar las Acciones Específicas asociadas a cada proyecto.
- Configurar las unidades ejecutaras locales asociadas a esa Acción Específica.
- Configurar las Acciones Específicas asociadas a esa Acción Centralizada.
- Gestionar una modificación presupuestaria.
- Cambiar el estado a una modificación presupuestaria.
- Crear modificaciones presupuestarias según tipo de modificación.
- Gestionar por cada trimestre la programación de sus metas por proyectos o acciones centralizadas.
- Aprobar la Programación Trimestral de las Metas.
- Gestionar por cada trimestre la programación de la Ejecución Financiera por proyectos o acciones centralizadas.
- Aprobar la Programación Trimestral de la Ejecución Financiera.
- Seleccionar el Año Presupuestario al cual se le va hacer la programación de Ingresos y Desembolsos.
- Gestionar por cada mes y por Denominación el monto y la Cuota de compromisos pendientes.

Según RN-AFDR-04 Documento de Requerimientos del módulo Contabilidad, el sistema debe permitir:

- Gestionar los comprobantes contables (Adicionar, eliminar y modificar un comprobante contable).
- Conformar un comprobante que se encuentre en estado de edición.
- Anular un comprobante una vez que el mismo este en estado de confirmación.
- Registrar los asientos contables patrimoniales y presupuestarios.

- Generar el comprobante contable de cierre.

Según RN-AFDR-01 Documento de Requerimientos del módulo Recaudación, el sistema debe permitir:

- Recibir y registrar la Planilla Única Bancaria (PUB) correspondiente al trámite a realizar a partir de los Módulos Registrales y Notariales en cada una de las oficinas.
- Enviar información de las PUB emitidas a la Unidad de Recaudación.
- Verificar y Actualizar el estado de la PUB.
- Enviar información de las PUB caducadas a la Unidad de Recaudación.
- Enviar información de las PUB pagadas a los registros.
- La recepción de la confirmación del pago de la PUB en el banco.
- El envío de la confirmación del pago de la PUB a la oficina y a la Unidad de Contabilidad.
- Validar la PUB que fue pagada en el banco.
- Capturar los datos del pago de la PUB para ser tramitada.
- Generar el asiento contable de la contabilidad patrimonial.
- Generar el asiento contable para la ejecución del presupuesto de ingreso.
- Enviar la PUB tramitada a la Unidad de Recaudación.
- Recibir un resumen diario emitido por el banco de las PUB pagadas durante el día, la información se enmarca en la identificación del banco, la fecha, la cantidad y el monto total.
- Conciliar la relación de cada una de las PUB pagadas durante el día con el resumen diario emitido por el banco.
- Enviar una petición al banco solicitando la relación de todas las PUB pagadas en caso de no estar conciliado.
- Recibir del banco la relación de todas las PUB pagadas y comparar con las recepcionadas durante el día para determinar cuáles son las diferencias que existen, estas diferencias serán categorizadas para una mejor comprensión por parte del Responsable de Recaudación.
- Gestionar la PUB en la vista de diferencias.
- Gestionar los reintegros solicitados tanto parciales como totales.
- Cambiar el estado del reintegro una vez confirmado.
- Mostrar las PUB relacionadas con el usuario que solicita el reintegro en la oficina correspondiente.

- Cambiar el estatus de las PUB una vez confirmado el reintegro.
- Generar el asiento contable correspondiente.
- Recaudar las PUB pagadas que vencieron el tiempo para realizar el trámite.
- Calcular la diferencia entre lo recaudado y trasferido en tiempo y monto.

De manera general el sistema tiene que:

- Generar y visualizar los reportes asociados a cada módulo.
- Imprimir los reportes.
- Exportar los reportes a formato PDF, XLS y DOC.

Iteración 2:

Comprende el desarrollo del módulo Requisiciones, Compras y Servicios, Retenciones y Tesorería

Iteración 3:

Comprende el desarrollo de los módulos Fondos en Anticipo y Fondos de Caja Chica.

En la sección correspondiente a la Vista Lógica y la de Implementación se verá con más detalles las relaciones que se establecen entre cada uno de estos módulos y sus especificidades internas.

Esta distribución se basa fundamentalmente en las dependencias que se establecen entre los módulos y en alguna medida en las prioridades que tiene el cliente con cada uno de ellos. Precisamente, al final de cada iteración obtenemos un fragmento de funcionalidad del sistema completo agrupadas de manera tal que el proceso de implementación sea fluido e ininterrumpido. No obstante como se verá, siempre quedan conexiones y dependencias que necesitan resolverse, esto significa que la Arquitectura tiene que incluir algún mecanismo para solventar esta situación.

La solución radica en la utilización de interfaces que representen las funcionalidades de cada subsistema, de esta forma su implementación es transparente al módulo que las utiliza y pude cambiar y evolucionar tantas veces como sea necesario sin necesidad de involucrarlo.

Para enlazar las funcionalidades con su implementación, es decir, la interfaz con la (s) clases que la implementan, se hace uso de fachadas desarrolladas sobre Spring.net y la implantación de patrones como el Proxy buscando garantizar en todo momento el cumplimiento de uno de los principios de la programación, que tiene su base en el patrón GRASP Bajo Acoplamiento descritos en [LC2004]. Además se incluye el establecimiento de objetos falsos para simular posibles resultados ante diferentes llamadas de las funcionalidades.

Por otra parte, la distribución horizontal de la solución descansa sobre un subsistema donde se van acoplando cada módulo luego de haberlo terminado y haya pasado por la etapa de prueba. Los módulos se acoplarían como plug-ins[7] que puedan agregarse y quitarse de la solución según se necesite, de este modo se tiene en todo momento una versión estable e independiente del desarrollo. Igualmente, a medida que se vaya cumpliendo cada una de las iteraciones propuestas, se realizarán las pruebas de la integración que serían a este nivel de organización teniendo en cuenta los resultados que se esperan obtener de las funcionalidades previstas.

Como se puede ver la Arquitectura está basada en componentes y conectores. Esta distribución garantiza mantener un desarrollo estable, seguro y al mismo tiempo agiliza el proceso de implementación.

En el caso de la distribución vertical se mantiene esta misma idea e incorpora algunas que contribuyen al enriquecimiento de la solución y a garantizar el cumplimiento de los atributos de calidad que son objetivo del presente trabajo.

2.4.2 Enfoque Vertical

[7] Un plugin (o plug-in -en inglés "enchufar"-, también conocido como addin, add-in, addon o add-on) es una aplicación informática que interactúa con otra aplicación para aportarle una función o utilidad específica, generalmente muy específica, como por ejemplo servir como driver (controlador) en una aplicación, para hacer así funcionar un dispositivo en otro programa.

El enfoque vertical se refiere a la distribución de la Arquitectura para cada módulo. Este tipo de distribución puede dar cabida a varias configuraciones diferentes, en este caso particular se adoptan las que garantizan con más eficiencia los atributos de calidad que se desean priorizar, los cuales constituyen el objetivo fundamental de la investigación.

Siguiendo la idea anterior, este enfoque está separado en dos estructuras que presentan configuraciones diferentes. La primera distribución se corresponde con el modulo Común que tiene características muy peculiares dada su función dentro de la Arquitectura según la figura (ver figura 3), la segunda viene dada por la generalidad de la solución y está representada en la figura (ver figura 4).

En una arquitectura de n niveles como esta, las acciones de la aplicación están lógicamente divididas por funciones. Lo anterior se basa en un estilo y patrón multicapa como se puede ver en las figuras (ver figura 3 y 4).

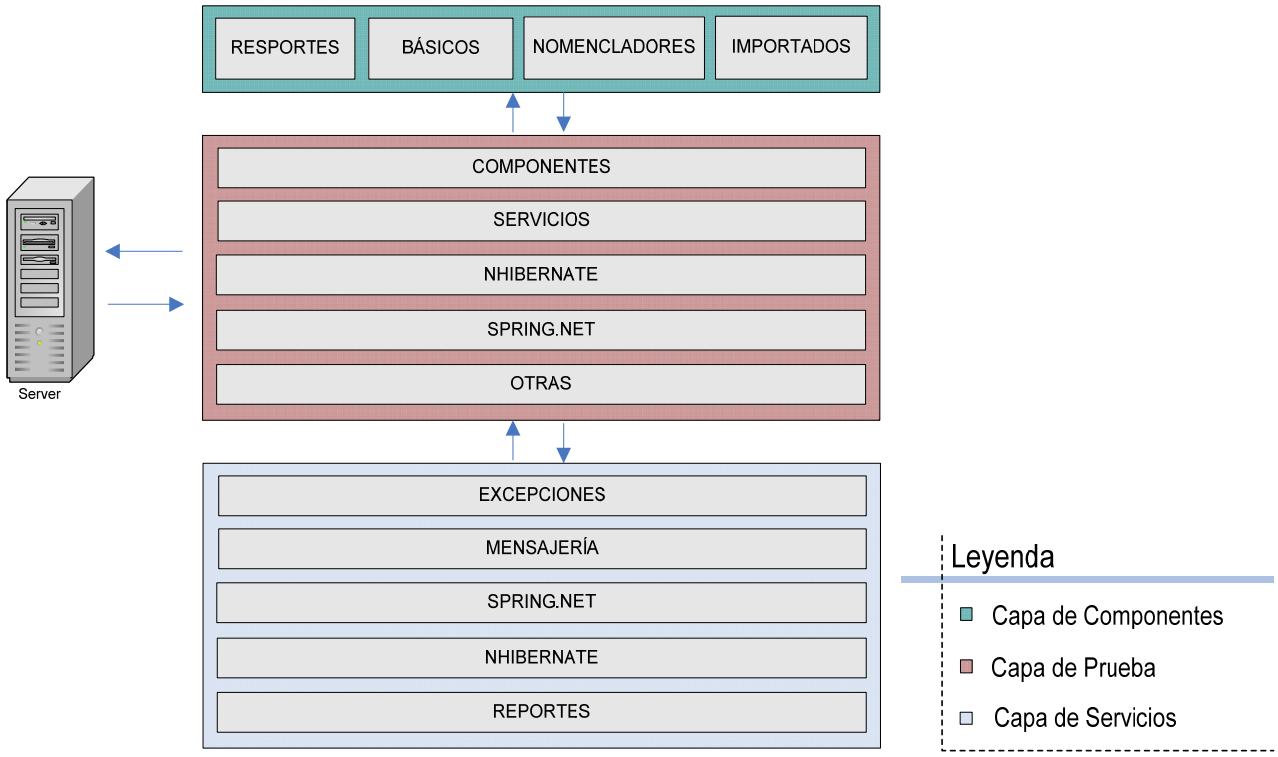


Figura 3 Representación del enfoque vertical de la Arquitectura módulo Común.

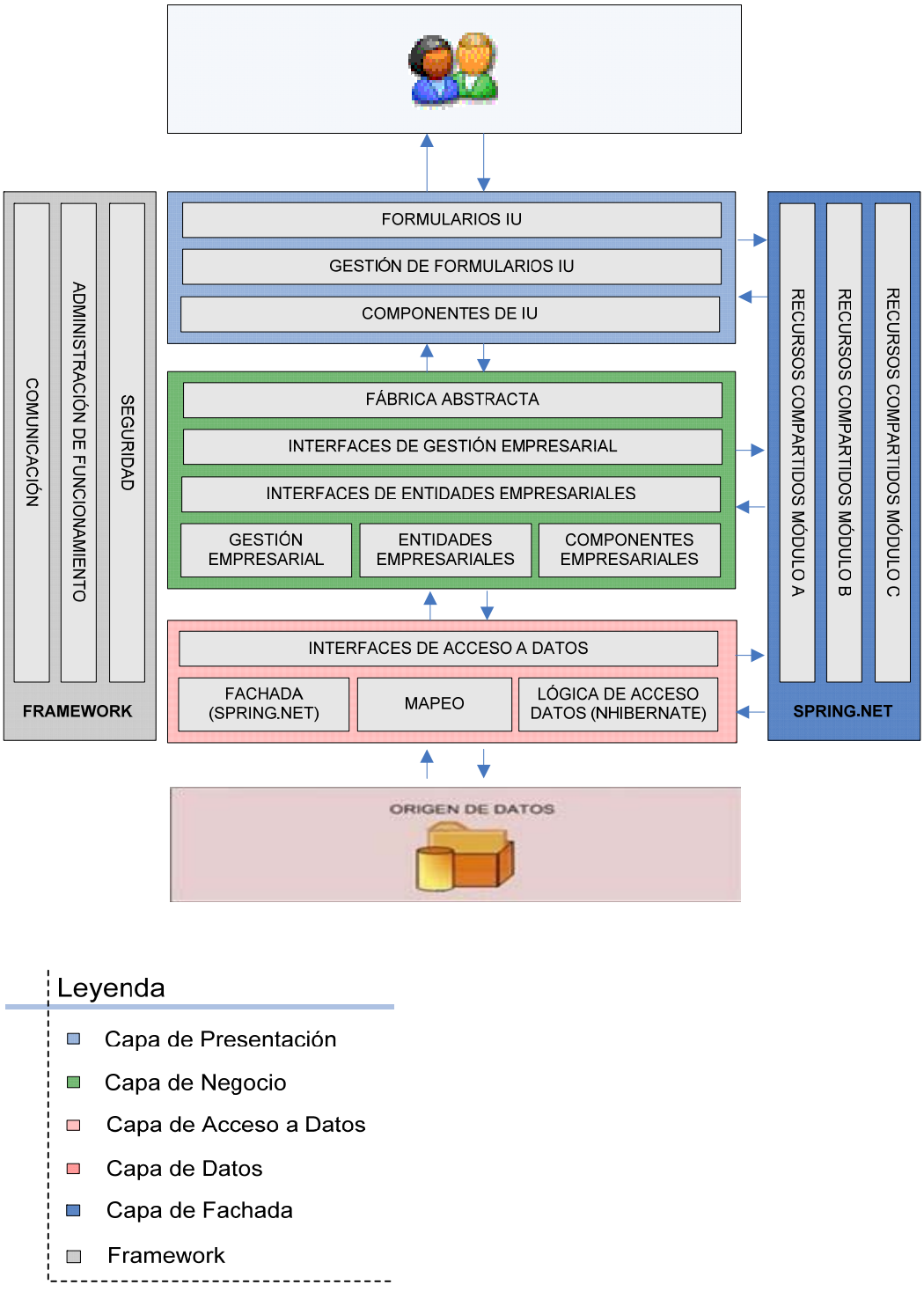


Figura 4 Representación del enfoque vertical de la Arquitectura general.

De manera general lo más representativo de esta distribución es la fuerte presencia que tienen los componentes y más aún los conectores, estos últimos resueltos en su mayoría con la utilización de interfaces y la implementación del Patrón Fachada, Proxy y la utilización del framework Spring.net como ya se habían mencionado. A continuación se detallan cada una de las capas y los componentes que conforman la perspectiva vertical de la Arquitectura:

- **Capa de Presentación:**

Formularios de Interfaz de Usuario.

Los formularios responden a un comportamiento y diseño estándar, todos los módulos tienen la misma forma con vistas a facilitar el trabajo con ellos y la estandarización del sistema completo.

Su uso se basa en la explotación de los Formularios de Interfaz de Usuario que realmente provee un framework (Framework Común) facilitando el desarrollo del sistema. Este framework se basa en acciones contempladas en la Gestión de Formularios de Interfaz de Usuario y cada acción se corresponde con funcionalidades de captura o visualización de los datos que tiene asociado un formulario cuya forma depende de la funcionalidad específica para la cual fue concebida dicha acción.

Gestión de Formularios de Interfaz de Usuario.

Su uso se basa en la explotación del componente gestión de interfaz que realmente provee el framework antes mencionado facilitando el desarrollo del sistema basado en acciones.

¿Qué es una acción?

- Cada acción debe tener sentido semántico propio y completo. Deben ser atómicas.
- Las acciones básicamente definen un bloque de código reusable por varias operaciones sobre el sistema.
- Las acciones están asociadas a vistas del sistema.
- Cada acción puede representar un estado del sistema que puede o no persistir.

- Una acción es una clase que representa precisamente la ejecución de alguna tarea concreta. Estas tareas no deben ser excesivamente complejas y la clase debe: Heredar de la clase **"Accion"** o "**AccionSegura"** (Framework Común).
- Propiedades en función del objetivo específico de la misma.
- Se le debe reescribir el método CrearForma con vistas a controlar lo que se desee del formulario asociado.
- Se deben programar dentro de la misma todos los eventos que puedan ocurrir y que se deseen utilizar a partir de la interacción con la forma visual.

Componentes de Interfaz de Usuarios.

Los componentes de interfaz de usuarios son recursos empleados para encapsular una función determinada y serán utilizados por más de un proceso en el módulo. El hecho de que estén en la Capa de Presentación se debe a que trabajan con los formularios directamente, ya sea de forma visual o con los datos que se encuentran relacionados en el mismo a través de otros componentes o controles. Cada componente puede incorporar clases de negocio e inclusive elementos de acceso a datos encapsulados y distribuidos según la estructura que tiene la Arquitectura general.

- **Capa Lógica de Negocio**

El diseño de esta capa depende directamente del negocio específico al que se refiera cada módulo.
El Negocio recibe datos y/o información capturada en las interfaces de usuario, gestiona o procesa la misma, de ser necesario solicitándola a la capa de Acceso a Datos y finalmente enviársela a la Presentación nuevamente para que esta la muestre al usuario en el punto donde se inició la petición.

Entidades del Negocio.

Las entidades empresariales son clases objeto - valor que representan los datos con los que se van a trabajar en cada uno de los procesos que se están automatizando.

Cada entidad es un elemento auto sustentado, es decir, se encarga de procesar sus propios datos o valores sin interactuar con los demás elementos del negocio, con esto se garantiza la independencia y el encapsulamiento de la información según la competencia que se tenga sobre la misma.

Gestión Empresarial.

Este ensamblado tiene como objetivo agrupar una serie de entidades con un fin común, lo anterior significa resolver determinadas funcionalidades donde intervienen una serie de datos que se encuentran en dichas clases objeto - valor. Las clases correspondientes a las funcionalidades se denominan Gestores.

Los Gestores serán clases estáticas puesto que solamente representan funcionalidades, por tanto no es necesario instanciarlos.

Componentes Empresariales.

Los componentes de negocio son recursos utilizados para encapsular una función determinada que será utilizada por más de un proceso de negocio en el módulo. Lógicamente estas funcionalidades son netamente de este nivel, es decir no tienen presentación pero si pueden utilizar recursos del acceso a datos.

Interfaces de Gestión y Entidades.

El pilar fundamental de la arquitectura radica en el uso de las interfaces como recurso que se utiliza para brindar funcionalidades que representan un nivel de abstracción. Este nivel es precisamente el que asegura el patrón Bajo Acoplamiento [LC2004] conjuntamente con otros elementos, algunos de los cuales ya se han visto.

La mayoría de las interfaces de la aplicación van a estar precisamente en el Negocio, puesto que aquí se encuentran en gran medida de la implementación de estas funcionalidades, ya sea en las Entidades, los Gestores o los Componentes de Gestión.

- **Capa Acceso a Datos**

Definir la estrategia de persistencia de una aplicación es una de las decisiones de Arquitectura más importantes. En una aplicación estándar más del 50% del código generado está relacionado con lógica de persistencia [RH2002].

Por tanto, el Acceso a Datos es la capa más crítica y sensible a cambios de la Arquitectura, pues controla todo lo concerniente a la información que se encuentra en la fuente de almacenamiento (Capa de Datos).

Al ser la capa inferior no conoce los niveles superiores, únicamente se limita al manejo de la información, ya sea para persistirla o proporcionarla para su procesamiento y propagación por la aplicación. Todo este manejo es responsabilidad de los Objetos de Acceso a Datos (DAOs por sus siglas en ingles).

Lógica de Acceso a Datos.

Todas las funcionalidades del Acceso a Datos radican en los DAOs, es decir este ensamblado implementa la totalidad de las operaciones de persistencia y obtención de datos explotando los recursos que brinda NHibernate framework que cumple perfectamente con el objetivo de este nivel, dígase trabajo con procedimientos almacenados y métodos de persistencia o consultas.

Fachada.

La fachada brinda una interfaz de alto nivel con la cual se va a interactuar cada vez que se necesite comunicarse, en este caso con el Acceso a Datos logrando una completa enajenación ante cualquier modificación que pueda ocurrir en la misma.

Esta estructura responde a la implementación de patrón Fachada, Proxy y está constituida por una clase que brinda el modulo Común y una estructura que enumera (*enum*[8]) los DAOs a los cuales se puede acceder desde el negocio.

Precisamente la utilidad del patrón en este nivel, radica en los elementos que se mencionaron anteriormente (criticidad, inestabilidad[9]), para su implementación se decidió el uso de Spring.net versión 1.1, framework que bajo el concepto de IoC resuelve perfectamente el problema planteado.

Mapeo.

Este elemento es de uso exclusivo de NHibernate como se verá, se decidió aislarlo en un ensamblado ya que básicamente son ficheros XML de configuración que son independientes de cualquier implementación, por tanto resulta muy útil tenerlos separados del código y así se evita recompilar toda una capa ante cualquier modificación en una de estas configuraciones.

Interfaces de Acceso a Datos.

Representan las funcionalidades que brinda este nivel, es decir las referidas a los DAOs. Su uso e importancia es la misma que las de la capa Lógica de Negocio.

- **Capa de Fachada**

La Capa de Fachada es una representación del patrón Proxy y Fachada a gran escala, este nivel es un recurso utilizado para mantener una representación de los demás módulos en el que se está implementando, siempre y cuando lo necesite. Es decir, cada módulo que se vaya a comunicar con el otro debe dar las funcionalidades que necesita (ver figura 4), las cuales se van a agrupar aquí. Esta distribución tiene su utilidad en el enfoque horizontal de la Arquitectura. Al emplear esta forma de

[8] La palabra clave **enum** se utiliza para declarar una enumeración, un tipo distinto que consiste en un conjunto de constantes con nombre denominado lista de enumeradores.

[9] Se refiere a que es una capa que está muy expensa a cambios.

comunicación entre los módulos se garantiza la abstracción y enajenación gracias a Spring.net, el cual ya se ha mencionado con anterioridad.

- **Capa de Datos**

Esta capa corresponde a los almacenes de datos. A ella pertenecen las bases de datos disponibles en los servidores de bases de datos. Esta capa es única y común a todos los módulos del sistema general. Su definición es la misma que la de la Arquitectura Base heredada de los Sistemas Registrales que ya se han mencionado (ver Anexo 2).

2.5 Elementos arquitectónicamente significativos.

Resultan elementos significativos en la Arquitectura el uso de frameworks/marcos de trabajo, los cuales brindan una serie de ventajas, principalmente porque comprenden funciones que resultan claves y muy útiles para lograr el cumplimiento del objetivo general.

Otros componentes importantes son los patrones implementados que en su conjunto brindan determinada funcionalidad que también da soporte a la arquitectura.

A partir de este análisis es importante reflejar cómo se lleva a cabo la implantación de aquellos elementos en la solución propuesta que mayor complejidad y dificultades puedan traer en el momento de utilizarlos.

2.5.1 NHibernate.

Para implementar NHibernate en la capa de Acceso a Datos se parte de que existe una Base de Datos Relacional y las tablas asociadas a cada clase persistente [BK2005] (estas clases son las entidades del negocio).

- Se crean los archivos mapping de cada entidad (ver figura 5 y 6).

- Se crea el archivo de configuración donde van a estar todos los datos de inicialización y conexión con la Base de Datos que necesita NHibernate (ver figura 7).

Todo lo concerniente a las funcionalidades de manipulación de la información para la persistencia y la obtención de los datos se maneja en los DAOs, para esto el framework propone sus propias implementaciones, las cuales se encuentran en el ensamblado principal NHibernate.dll (ver código 1). Las funcionalidades incluyen inserción, actualización, eliminación, etc. y el uso de procedimientos almacenados como un recurso importante en toda la lógica de acceso a datos.

En el caso de los procedimientos almacenados se tiene que hacer un archivo de mapeo XML independiente, donde justamente se relacionan todos estos procedimientos. Por otra parte, usando Oracle como gestor de Base de Datos (ver figura 8), luego de mapear los procedimientos el sistema está preparado para su uso y explotación a través de los recursos que en este sentido ofrece NHibernate (ver código 2).

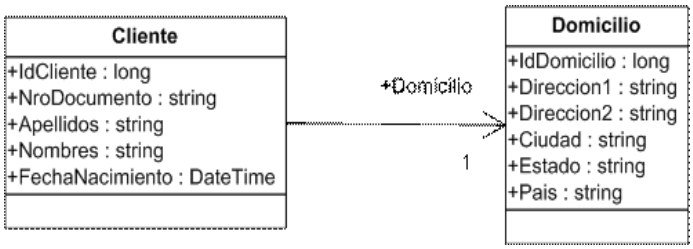


Figura 5. Representación de dos entidades con una relación de uno a muchos.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.0">
    <class name="EjemploNHibernateModeloDominio.Domicilio, EjemploNHibernateModeloDominio" table="DOMICILIO">
        <id name="IdDomicilio" column="ID_DOMICILIO" type="Int64" unsaved-value="0">
            <generator class="sequence">
                <param name="sequence">SEQ_DOMICILIO</param>
            </generator>
        </id>
        <property name="Direccion1" column="DIRECCION1" type="String" length="50"/>
        <property name="Direccion2" type="String" length="50"/>
        <property name="Ciudad" column="CIUDAD" type="String" length="50"/>
        <property name="Estado" column="ESTADO" type="String" length="50"/>
        <property name="Pais" column="PAIS" type="String" length="50"/>
    </class>
</hibernate-mapping>
```

```xml
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.0">
    <class name="EjemploNHibernateModeloDominio.Cliente, EjemploNHibernateModeloDominio" table="CLIENTE">
        <id name="IdCliente" column="ID_CLIENTE" type="Int64" unsaved-value="0">
            <generator class="sequence">
                <param name="sequence">SEQ_CLIENTE</param>
            </generator>
        </id>
        <property name="NroDocumento" column="NRO_DOCUMENTO" type="String" length="20"/>
        <property name="Apellidos" column="APELLIDOS" type="String" length="50"/>
        <property name="Nombres" column="NOMBRES" type="String" length="50"/>
        <property name="FechaNacimiento" column="FECHA_NACIMIENTO" type="DateTime"/>

        <many-to-one name="Domicilio" column="ID_DOMICILIO" cascade="all"/>
    </class>
</hibernate-mapping>
```

Figura 6. Ejemplo de los archivos de mapeo que genera la relación anterior.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="nhibernate" type="System.Configuration.NameValueSectionHandler, System,
      Version=1.0.5000.0,Culture=neutral, PublicKeyToken=b77a5c561934e089" />
    <section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler,log4net" />
  </configSections>

  <nhibernate>

    <add key="hibernate.connection.provider"
         value="NHibernate.Connection.DriverConnectionProvider"
    />

    <add key="hibernate.dialect"
         value="NHibernate.Dialect.OracleDialect"
    />

    <add
      key="hibernate.connection.driver_class"
      value="NHibernate.Driver.OracleClientDriver"
    />

    <add key="hibernate.connection.connection_string"
         value="Data Source=XE;User ID=user;Password=user;"
    />

  </nhibernate>
</configuration>
```

Figura 7. Archivo de configuración.

```
ISession session[10] = null;
 ITransaction transaction = null;
 try
 {
     session = sessionFactory.OpenSession();
     transaction = session.BeginTransaction();
     session.Metodo(parametro);
     transaction.Commit();
     MessageBox.Show("Cliente grabado correctamente");
     btnLimpiar_Click(sender, e);
```

[10] 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 nos permite manejar el contexto transaccional de nuestra lógica.

```
    }
    catch (Exception ex)
    {
        if (transaction != null) transaction.Rollback();
        MessageBox.Show(ex.Message);
    }
    finally
    {
        if (session != null) session.Close();
    }
}
```

……..

Código 1. Código que representa los pasos para la llamada a los métodos implementados por NHibernate.

Utilizando el método ISession.BeginTransaction() se obtiene un objeto del tipo ITransaction que representa la transacción que se utilizará para llamar al método que trabajará con los datos.

```
transaction = session.BeginTransaction();
```

Siguiendo el ejemplo, una vez iniciada la transacción, lo único que se tiene que hacer para salvar automáticamente el objeto Cliente y el objeto Domicilio asociado, es utilizar el método ISession.SaveOrUpdate(Object o) pasándole como argumento el objeto que se desea salvar.

```
session.SaveOrUpdate(cliente);
```

Utilizando el unsaved - value especificado en ambos mappings, NHibernate determina si tiene que insertar un nuevo objeto o actualizar uno existente. En una sola línea de código el framework resuelve en forma automática la grabación en ambas tablas (Cliente y Domicilio).

Luego de salvar el objeto, lo único que resta es confirmar la transacción en curso. En este sentido lo único que se tiene que hacer es invocar el método ITransaction.Commit().

```
transaction.Commit();
```

Si ocurre algún error, se deshacen todos los cambios utilizando el método ITransaction.Rollback();

```
catch (Exception ex)
{
    if (transaction != null) transaction.Rollback();
    MessageBox.Show(ex.Message);
}
```

Finalmente hay que asegurarse de cerrar siempre la sesión utilizando el método ISession.Close();

```
finally
{
    if (session != null) session.Close();
}
```

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.0">

    <store-procedure name="spObtenerClientes" procedure-name= "PKG_REPORTES.ObtenerClientes">

        <parameter name="io_AllRec"/>

        <return-list return-class="EjemploNHibernateModeloDominio.Cliente, EjemploNHibernateModeloDominio">

            <return-property name="IdCliente" num-col="0"/>
            <return-property name="NroDocumento" num-col="1"/>
            <return-property name="Apellidos" num-col="2"/>
            <return-property name="Nombres" num-col="3"/>
            <return-property name="FechaNacimiento" num-col="4"/>

            <return-object name="Domicilio">
                <return-property name="IdDomicilio" num-col="5"/>
                <return-property name="Direccion1" num-col="6"/>
                <return-property name="Direccion2" num-col="7"/>
                <return-property name="Ciudad" num-col="8"/>
                <return-property name="Estado" num-col="9"/>
                <return-property name="Pais" num-col="10"/>
            </return-object>

        </return-list>

    </store-procedure>

</hibernate-mapping>
```

Figura 8. Archivo de mapeo de los procedimientos almacenados.

……

1. session = sessionFactory.OpenSession();
   transaction = session.BeginTransaction();
   ArrayList clientes = session.CreateStoreProcedure("spObtenerClientes").List();
   transaction.Commit();

2. ArrayList clientes = session.CreateStoreProcedure("spObtenerClientesPorFechaNac")
   .SetParameter(inicial)
   .SetParameter(final)
   .List();

3. ArrayList clientes = session.CreateStoreProcedure("spObtenerClientesPorFechaNac")

```
                    .SetParameter(inicial)
                    .SetParameter(final)
                    .GetDataSet();

4. ArrayList clientes = session.CreateStoreProcedure("spEliminarClientes")
                    .SetParameter(idcliente)
                    .ExecuteNonQuery();
```

……..

Código 2. Ejemplos del uso de procedimientos almacenados con NHibernate, dependiendo del mapeo.

Se han descrito una serie de eventos que realiza NHibernate relacionados fundamentalmente con transacciones verticales. Resta definir como se realiza el trabajo con las transacciones horizontales las cuales involucran una serie de procesos que no deben persistir hasta que cada uno de ellos se ejecute completamente. Lo anterior significa que la información no se registra en la Base de Datos hasta que no se hayan realizado correctamente (sin Excepciones) todas las operaciones envueltas en la transacción.

Para el trabajo con estas transacciones y más aún con las funcionalidades básicas de NHibernate para Salvar, Eliminar, Actualizar, etc. se va a implementar en el módulo “ArquitecturaBase“ un componente de negocio que maneje todas estas funciones, las cuales trabajan en su totalidad con transacciones verticales. Así mismo, el negocio encapsulado en este componente encuentra su contraparte de Acceso a Datos, según la arquitectura vertical, la cual incluye además otros métodos para configuraciones, inicializaciones y precisamente el trabajo con transacciones. En consecuencia, cada vez que se necesite en cada módulo, trabajar con transacciones horizontales, solamente habría que acceder a este componente e interactuar directamente con los métodos dedicados a las transacciones embebiendo el código que se quiera en los mismos según se muestra (ver código 3).

```
……
objDaoComun.IniciarTransaction();
objDaoComun.EliminarColeccionObjetos(directoresAEliminar);
if (dependenciasAEliminar != null)
{
        foreach (IDependenciaFinancieraUACs objeto in dependenciasAEliminar)
        {
                if (objeto.ObjEstructuraFinanciera.IdEstructuraFinanciera != 0)
                        objDaoComun.Eliminar(objeto);
        }
}
objDaoComun.Salvar(objUAC);
if (objEstructura != null)
{
        objDaoComun.SalvarOActualizar(objEstructura);
        if (objEstructura.ColeccionDependenciasFinancieras != null)
        {
                foreach (IDependenciaFinancieraUACs objeto in
                    objEstructura.ColeccionDependenciasFinancieras)
                {
                        IDependenciaFinancieraUACs dependencia = new
                        EDependenciaFinancieraUACs(objEstructura,objeto.ObjUEL);
                        objDaoComun.SalvarOActualizar(dependencia);
                }
        }
}
objDaoComun.ReafirmarTransaction();
……
```

Código 3 Ejemplo del uso de transacciones horizontales.

2.5.2 Spring.net

Atendiendo a las ventajas y desventajas que se mencionaron en el Capítulo 1 referentes a este framework, se decidió implantar Spring.net en la capa de Acceso a Datos específicamente en su fachada y en la Capa Fachada de los módulos, con esto se garantiza que se cumplan todos los objetivos que se persiguen en cada una de ellas explotando las virtudes que tiene el framework. Es decir, con vistas a no sacrificar el rendimiento de la aplicación son estas dos capas las únicas en implantar esta variante, más aún por ser tan importantes y sensibles; donde resulta más útil sacrificar algo de rendimiento con vistas a ganar en desacoplamiento y abstracción de la información. A esto se le suma que en estos niveles no se manejan la mayoría de objetos y por tanto dependencias.

Para utilizar Spring.net primeramente se establecen todas sus configuraciones en el fichero app.config (ver figura 9), además se relacionan todos los objetos que se van a utilizar y sus dependencias.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<configSections>
   <sectionGroup name="spring">
        <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
        <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
   </sectionGroup>
</configSections>
<spring>
    <context>
      <resource uri="config://spring/objects" />
    </context>
<objects xmlns="http://www.springframework.net">

<object id="GtrContabilidad" type="Contabilidad.Negocio.Gestion.GtrContabilidad,
 Contabilidad.Negocio.Gestion">
    <constructor-arg index="0" ref="DaoComun"/>
    <constructor-arg index="1" ref="DaoOperacionesRegistroComp"/>
    <constructor-arg index="2" ref="DaoReportes"/>
</object>

<object id="DaoOperacionesRegistroComp"
 type="Contabilidad.Acceso.Datos.Daos.DaoOperacionesRegistroComp,
 Contabilidad.Acceso.Datos.Daos"/>

<object id="DaoReportes" type="Contabilidad.Acceso.Datos.Daos.DaoReportes,
 Contabilidad.Acceso.Datos.Daos"/>

<object id="DaoComun" type="ArquitecturaBase.Acceso.Datos.Daos.DaoComun,
 ArquitecturaBase.Acceso.Datos.Daos"/>
</objects>
</spring>
</configuration>
```

Figura 9. Archivo con las configuraciones de Spring.

Una estrategia utilizada para trabajar con este framework es el uso de interfaces, pues que con ellas se mantienen representaciones de objetos, o sea funcionalidades que serán instanciadas a partir de la configuración de un fichero XML y no en el código, logrando así un total desacoplamiento entre módulos y entre el negocio y la capa de Acceso a Datos a través de su fachada.

Es importante aclarar que el uso de las interfaces en casi la totalidad de la aplicación y no solamente en aquellos que formarán parte de la comunicación con otros módulos o capas se debe a que indudablemente este tipo de implementación ofrece múltiples ventajas[11] que en un momento determinado se pueden aprovechar, además garantizamos uniformidad y tener preparada la aplicación ante cualquier cambio en las peticiones y funcionalidades que se dan.

```
……..
IApplicationContext ctx = ContextRegistry.GetContext();
return ctx[“nombre objeto”];
……..
```

Código 3. Obteniendo el objeto del fichero de configuración.

Con este código se resuelve el problema fundamental de obtener el objeto correspondiente a la clase que implementa la interfaz puesto que solamente se necesitaría el nombre con el que se identifica en el fichero de configuración y este inyectaría las dependencias que se tienen establecidas previamente en el XML.

Es importante señalar que el uso de Spring.net no se limita a las funcionalidades propias del framework, pues contiene una serie de implementaciones e integraciones con otros marcos [PESSHCR2004-2006], en este sentido el que resulta más atractivo es NHibernate sobre todo para el manejo de transacciones y funcionalidades a través de ficheros XML. Esta línea se sigue como recomendaciones para futuras evoluciones de esta Arquitectura. En resumen, podemos obtener un archivo de configuración más completo y la aplicación quedaría más organizada estructuralmente. Es decir a medida que se quiera explotar aún más las ventajas que brinda el framework solamente habría que establecer las instrucciones necesarias en el XML y hacer la llamada correspondiente a sus ensamblados.

[11] El uso de interfaces es un mecanismo para abstraer los métodos a un nivel superior, múltiples objetos de clases diferentes pueden ser tratados como si fuesen de un mismo tipo común, donde este tipo viene indicado por el nombre del interfaz.

2.5.3 Patrón Proxy y Reflection.

La implantación de estos patrones se conjuga en el módulo ”ArquitecturaBase” para obtener la funcionalidad deseada, que en este caso se refiere a la integración de sistema con otros que pueden coexistir en el medio algunos de los cuales ya han sido mencionados.

La implementación está sustentada en un fichero de configuración XML (ver figura 10) donde se relacionan todas las funcionalidades que se desean compartir a través de ensamblados.

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

<Ensamblados>

      <GtrPlanillaUnicaBancaria>
            <ensamblado>Recaudacion.Negocio.Gestion</ensamblado>
            <clase>Recaudacion.Negocio.Gestion.GtrPlanillaUnicaBancaria</clase>
      </GtrPlanillaUnicaBancaria>

      <ConceptoRecaudacion>
            <ensamblado>Recaudacion.Negocio.Entidades</ensamblado>
            <clase>Recaudacion.Negocio.Entidades.ConceptoRecaudacion</clase>
      </ConceptoRecaudacion>

</Ensamblados>
```

Figura 10 Fichero XML de configuración para las funcionalidades que se brindan.

A partir de este XML por mecanismos de Reflection se instancian las funcionalidades que se encuentran en las interfaces. El encargado de llevar a cabo dicha operación es el Proxy que sirve de intermediario con las aplicaciones externas. El código (ver código 4) muestra como se lleva a cabo todo este proceso.

```
……
public static object Instanciar(string nombreObjeto)
{
        try
        {
                Assembly asm = System.Reflection.Assembly.GetEntryAssembly();
                string nombreA = asm.FullName;
                nombreA = nombreA.Substring(0, nombreA.IndexOf(','));
                System.IO.Stream stream = asm.GetManifestResourceStream(nombreA +
                ".Ensamblados.xml");
                System.Xml.XmlDocument docXML = new System.Xml.XmlDocument();
                docXML.Load(stream);
                string nombreAsm = docXML["Ensamblados"][nombreObjeto]["ensamblado"].InnerText;
                Assembly asmPlugin =  Assembly.Load(nombreAsm);
                string nombreClase = docXML["Ensamblados"][nombreObjeto]["clase"].InnerText;
                System.Type tipo = asmPlugin.GetType(nombreClase);

                return tipo.GetConstructor(new Type[0] { }).Invoke(new object[0] { });

        }
        catch(Exception ex)
        {
                throw new Exception(ex.ToString());
        }
}
……
```

Código 4 Implementación del Proxy a través de mecanismos de Reflection.

2.6 Vista de Casos de Uso.

Según RUP en la Vista de Casos de Usos de la Arquitectura se relacionan aquellos que son arquitectónicamente significativos.

A continuación se presenta la Vista de Casos de Uso para la primera iteración de la Arquitectura:

- Administración.
- Presupuesto.
- Recaudación.
- Contabilidad.

2.6.1 Administración

El Módulo de Administración conforma la relación de todos los codificadores necesarios para el funcionamiento del resto de los módulos del sistema, así como la definición de los Indicadores Generales por cada uno de ellos.

Entre los codificadores fundamentales se encuentran: la Estructura Financiera que incluye la definición de la UAC, la gestión de las UAD y la creación de las UEL, además se configuran los Clasificadores de Cuentas Patrimoniales, Presupuestarias, Económico, el Catálogo de Bienes y Servicios y las relaciones que puedan establecerse entre ellos.

Teniendo en cuenta la descripción del módulo y RN-AFMCU-02 Documento de Caso de Uso del Sistema del módulo Administración, resultan Casos de Usos significativos para la Arquitectura:

- Definir Estructura Financiera de la UAC.
- Gestionar Unidad Administradora Desconcentrada.
- Gestionar Unidad Ejecutora Local.
- Gestionar Clasificador Económico.
- Gestionar Plan de Cuentas Patrimoniales.
- Gestionar Catálogo de Bienes y Servicios.
- Gestionar Clasificador Presupuestario.

En la figura (ver figura 12) se muestran estos casos de uso en la vista de Casos de Uso de la Arquitectura.

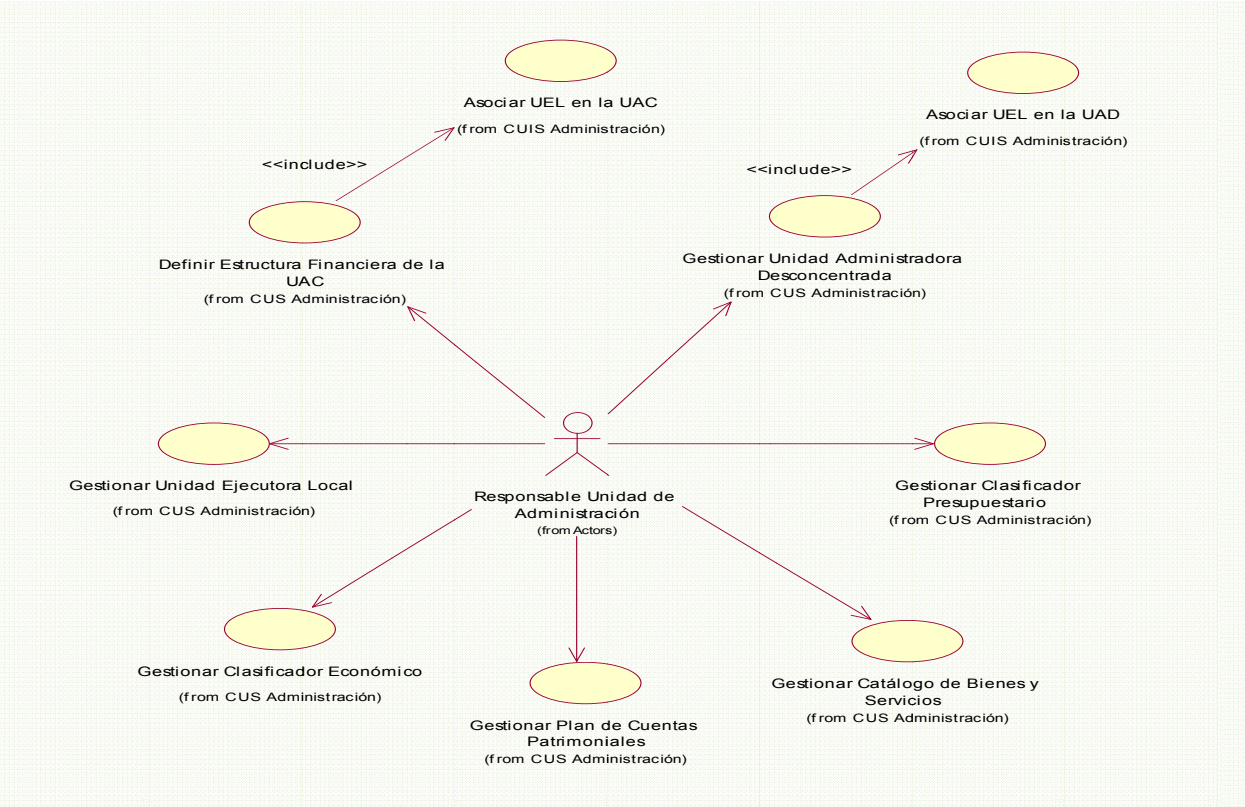


Figura 12 Vista de Casos de Uso módulo Administración.

Definir Estructura Financiera de la UAC: Consiste en gestionar en el sistema las propiedades de la UAC y su dependencia financiera.

Gestionar Unidad Administradora Desconcentrada: Permite crear, modificar y eliminar UADs en el sistema y relacionarlas con las oficinas.

Gestionar Unidad Ejecutora Local: Permite crear, modificar y eliminar UELs en el sistema y relacionarlas con las oficinas

Gestionar Clasificador Económico: Consiste en crear el Clasificador Económico con el cual se trabajará en el sistema.

Gestionar Plan de Cuentas Patrimoniales: Consiste en crear el Plan de Cuentas Patrimoniales con el cual se trabajará en el sistema

Gestionar Catálogo de Bienes y Servicios: Consiste en crear el Catálogo de Bienes y Servicios con sus propiedades.

Gestionar Clasificador Presupuestario: Permite crear el Clasificador Presupuestario vigente con el cual se trabajará en el sistema dando la posibilidad de gestionar dichas cuentas.

2.6.2 Presupuesto.

El Módulo de Presupuesto contempla todas las funcionalidades relacionadas con la Formulación, Modificación, Programación, Ejecución y Cierre del presupuesto para un año seleccionado a partir de la Estructura Financiera aprobada.

La Formulación se realiza por Proyectos y Acciones Centralizadas y cada uno de estos está compuesto a su vez por Acciones Específicas las cuales están detalladas por Unidades Ejecutoras Locales y Fuentes de Financiamiento.

Este proceso de Formulación comienza en las solicitudes de las Unidades Ejecutoras Locales de sus necesidades presupuestarias para la ejecución de sus metas y se consolida a nivel de Unidad Administradora Desconcentrada y Unidad Administradora Central.

El proceso de Modificación Presupuestaria se realiza una vez sancionada la ley del presupuesto por la existencia de Traspasos de Créditos Presupuestarios, Insubsistencia, Créditos Adicionales y Recorte de Créditos Presupuestarios al nivel de las Unidades Ejecutoras Locales.

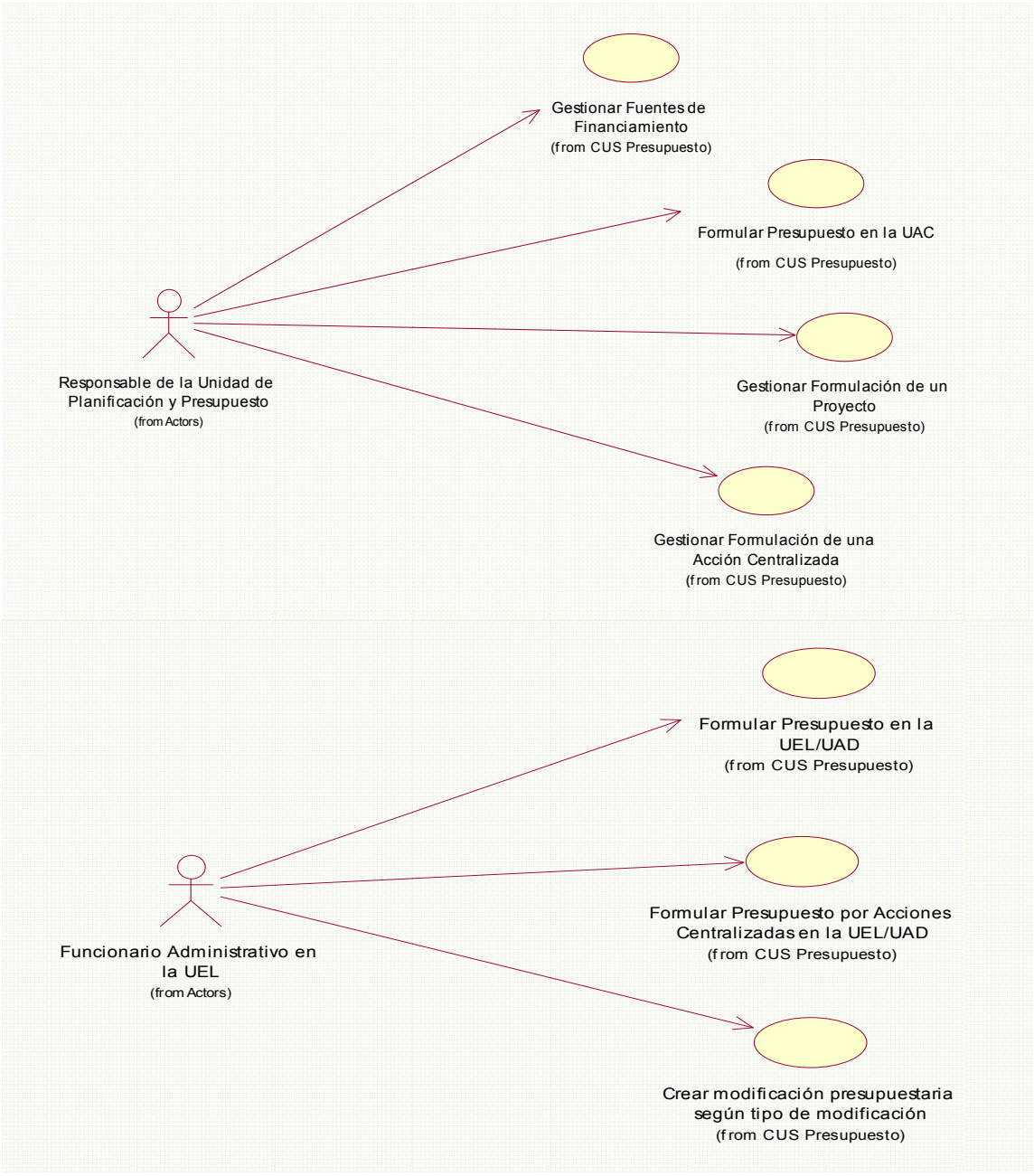
El Proceso de Programación se realiza por Trimestres, obteniéndose como resultado una tabla de programación mensual.

En la Ejecución y Cierre de Presupuesto se generan diferentes reportes donde se muestra la ejecución física y financiera de los proyectos/acciones centralizadas además de la ejecución financiera trimestral del presupuesto tanto del gasto como del ingreso.

Teniendo en cuenta la descripción del módulo y RN-AFDR-03 Documento de Caso de Uso del Sistema del módulo Presupuesto, resultan Casos de Usos significativos para la Arquitectura:

- Formular Presupuesto en la UAC.
- Gestionar Fuentes de Financiamiento.
- Gestionar Formulación de un Proyecto en la UAC.
- Gestionar Formulación de una Acción Centralizada en la UAC.
- Formular Presupuesto por Proyecto en la UEL/UAD.
- Formular Presupuesto por Acciones Centralizadas en la UEL/UAD.
- Crear Modificación Presupuestaria según tipo de modificación.
- Programar la Ejecución Financiera por Trimestre y Proyectos.
- Programar la Ejecución Financiera por Trimestre.
- Programar Ingresos y Desembolsos mensualmente.

La figura (ver figura 13) se muestran estos casos de uso del módulo en la vista de Casos de Uso de la Arquitectura.


Gestionar Fuentes de Financiamiento
(from CUS Presupuesto)
Formular Presupuesto en la UAC
(from CUS Presupuesto)
Responsable de la Unidad de Planificación y Presupuesto
(from Actors)
Gestionar Formulación de un Proyecto
(from CUS Presupuesto)
Gestionar Formulación de una Acción Centralizada
(from CUS Presupuesto)
Formular Presupuesto en la UEL/UAD
(from CUS Presupuesto)
Funcionario Administrativo en la UEL
(from Actors)
Formular Presupuesto por Acciones Centralizadas en la UEL/UAD
(from CUS Presupuesto)
Crear modificación presupuestaria según tipo de modificación
(from CUS Presupuesto)

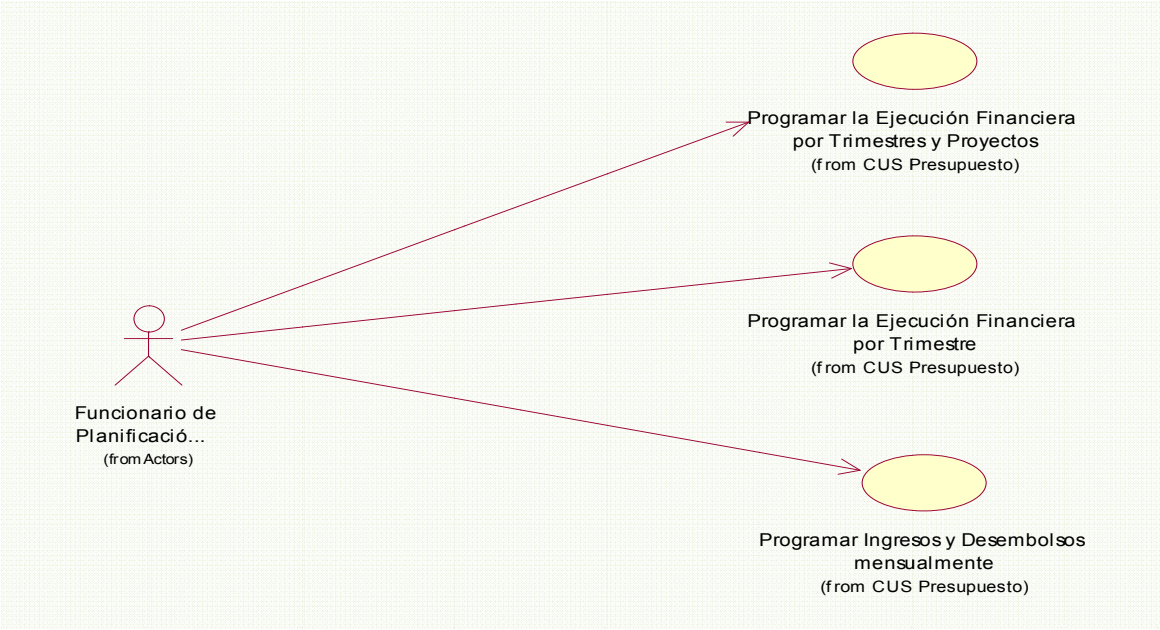


Figura 13 Vista de Casos de Uso módulo Presupuesto.

Formular Presupuesto en la UAC: Consiste en formular el anteproyecto de presupuesto a nivel central en la UAC.

Gestionar Fuentes de Financiamiento: Consiste en asociar las fuentes de financiamiento para crear el Ante-Proyecto de presupuesto en la UAC.

Gestionar Formulación de un Proyecto en la UAC: Consiste en configurar los datos necesarios para crear el presupuesto de un proyecto.

Gestionar Formulación de una Acción Centralizada en la UAC: Consiste en configurar los datos necesarios para crear el presupuesto de una Acción Centralizada.

Formular Presupuesto por Proyecto en la UEL/UAD: Consiste en la formulación del proyecto de presupuesto desde las UEL o las UAD.

Formular Presupuesto por Acciones Centralizadas en la UEL/UAD: Consiste en modificar las acciones centralizadas para el ante proyecto de presupuesto en la UEL/UAD.

Crear Modificación Presupuestaria según tipo de modificación: Consiste en crear la modificación presupuestaria dependiendo del tipo que esta sea.

Programar la Ejecución Financiera por Trimestre y Proyectos: Consiste en hacer una programación por trimestre del año que se va a presupuestar teniendo en cuenta las metas por proyectos.

Programar la Ejecución Financiera por Trimestre: Consiste en hacer una programación por trimestre del año que se va a presupuestar teniendo en cuenta la Ejecución Financiera por proyectos y por acciones centralizadas.

Programar Ingresos y Desembolsos mensualmente: Consiste en hacer una programación mensual de ingresos y desembolsos del año que se está presupuestando.

2.6.3 Recaudación.

El Módulo de Recaudación consiste en la recepción de los pagos que deben realizar los clientes de los Registros y Notarías por los trámites solicitados, en las oficinas de los bancos recaudadores correspondientes, mediante una Planilla Única Bancaria, diseñada de acuerdo a las necesidades de las oficinas.

Permite registrar el estatus de las Planillas Únicas Bancarias desde que son emitidas en las oficinas hasta que le dan continuidad al trámite después de pagada, así como el proceso de Reintegros y Recaudación por abandono de las mismas. Permite además evaluar la cantidad de Planillas Únicas Bancarias exoneradas, exentas y caducadas.

Registra y controla todos los Ingresos que se obtienen por Derechos de Registro Público, Mercantil, Principal y Notariales, Habilitación, Traslado, Inserción Anticipada del documento y Transporte.

Realiza las coordinaciones con los bancos recaudadores cuando exista algún problema de conciliación de las Planillas pagadas para la búsqueda de posibles soluciones.

Teniendo en cuenta la descripción y RN-AFMCU-01 Documento de Caso de Uso del Sistema del módulo Recaudación, resultan Casos de Usos significativos para la Arquitectura:

- Recibir PUB emitidas.
- Notificar PUB Caducada.
- Recibir PUB en Trámite.
- Recibir PUB para ser tramitada por autorización.
- Recibir confirmación de las PUB pagadas en banco.
- Recibir Resumen Diario de la PUB pagadas.
- Capturar Reintegros.
- Recaudar PUB por abandono.

La figura (ver figura 14) se muestran estos casos de uso del módulo de Recaudación en la vista de Casos de Uso de la Arquitectura.

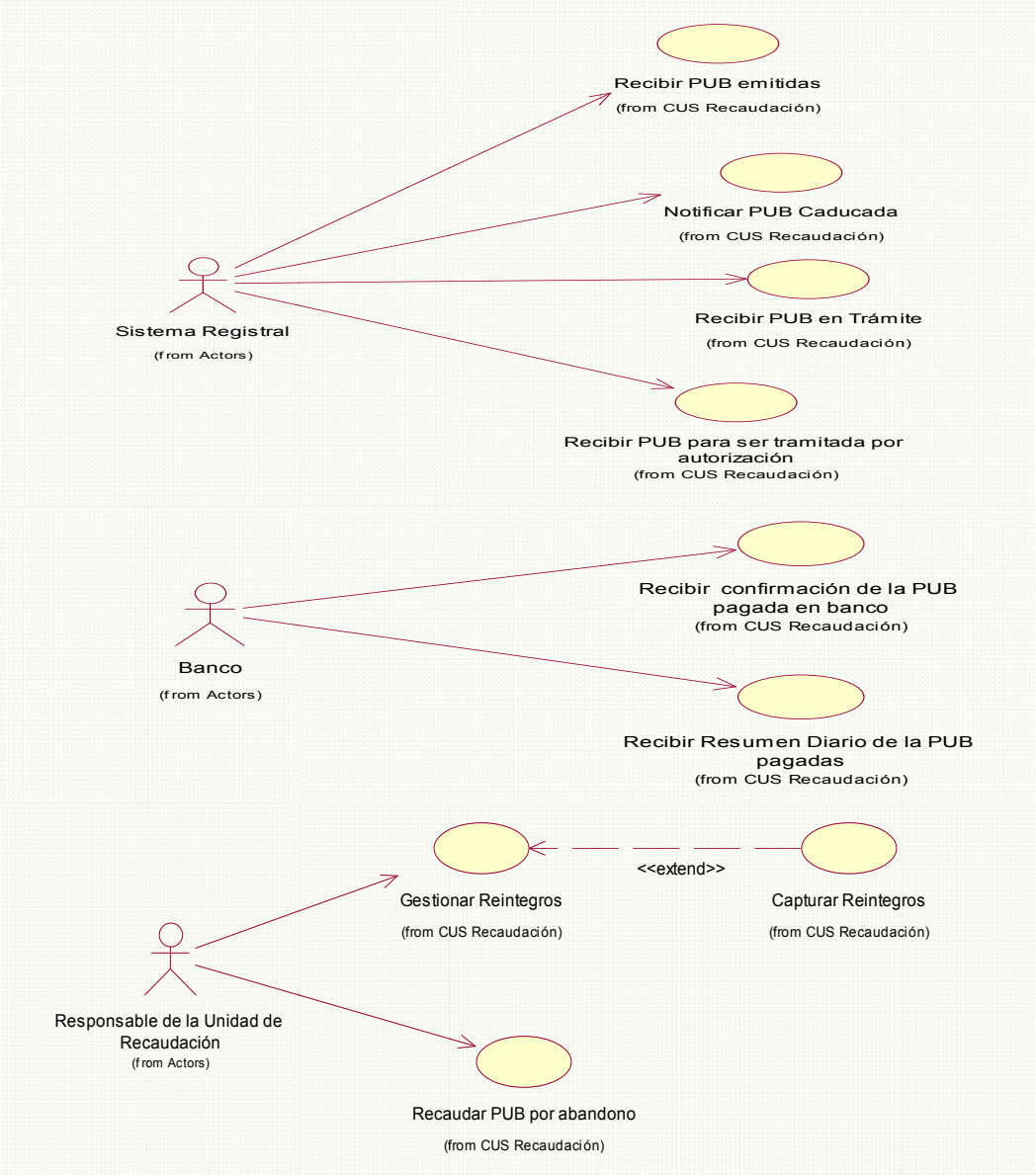


Figura 14 Vista de Casos de Uso módulo Recaudación.

Recibir PUB emitidas: Este caso de uso es iniciado por el actor sistema registral cuando envía la información de las PUB emitidas para ser registradas en la UR, estableciéndole un estado a las mismas.

Notificar PUB Caducada: Este caso de uso es iniciado por el actor sistema registral que le informa a la Unidad de Recaudación las PUB que han sido caducadas por no ser pagadas en el tiempo previsto.

Recibir PUB en Trámite: Cuando los usuarios presentan las PUB pagadas para darle continuidad al trámite se registra el pago de la misma y se envían a la Unidad de Recaudación.

Recibir PUB para ser tramitada por autorización: Consiste en registrar la PUB que es presentada para continuar el trámite con una previa autorización por parte de la autoridad competente.

Recibir confirmación de las PUB pagadas en banco: Este caso de uso se inicia cuando el Banco envía el recibo de la confirmación del pago de la PUB en el banco y es recibida en la UR donde se verifica que el Nro. Único de PUB y el Monto Cancelado de la notificación se corresponda con la PUB emitida, en caso contrario se registra la diferencia existente.

Recibir Resumen Diario de la PUB pagadas: Este caso de uso es iniciado por el Banco una vez que emite el resumen diario de las PUB pagadas durante el día a la UR, en donde se concilia con las recibidas durante el día para ver si hay diferencias entre las mismas, en caso de encontrarse alguna se solicita al Banco un resumen detallado para detectar donde existe la diferencia, realizando la comparación y registrando las diferencias encontradas.

Capturar Reintegros: Consiste en registrar los reintegros presentadas, una vez realizado el pago de la Planilla Única Bancaria.

Recaudar PUB por abandono: Consiste en mostrar las PUB por oficina que están pagadas y ha culminado el tiempo de realizar el trámite y de reclamar el monto depositado así como el reporte por oficina de las mismas.

2.6.4 Contabilidad.

En su conjunto ofrece la posibilidad de tener una Contabilidad actualizada, veraz y segura, que le permita obtener la información necesaria para lograr una administración más eficiente, si son aplicadas todas las funciones que en el mismo se han concebido.

En este módulo se muestra el registro de comprobantes, a partir de los cuales se controlan todas las operaciones contables generadas por el resto de los subsistemas y creadas en este, estos comprobantes son de tipo Patrimonial o Presupuestario. Otro de los procesos existentes es el cierre de las cuentas nominales el cual genera un comprobante, el resultado del cierre se asienta en una cuenta contable seleccionada.

Teniendo en cuenta la descripción y RN-AFMCU-04 Documento de Caso de Uso del Sistema del módulo Contabilidad, resultan Casos de Usos significativos para la Arquitectura:

- Gestionar Registro de Comprobantes Contables.
- Cerrar Cuentas Nominales.

La figura (ver figura 15) se muestran estos casos de uso del módulo en la vista de Casos de Uso de la Arquitectura.

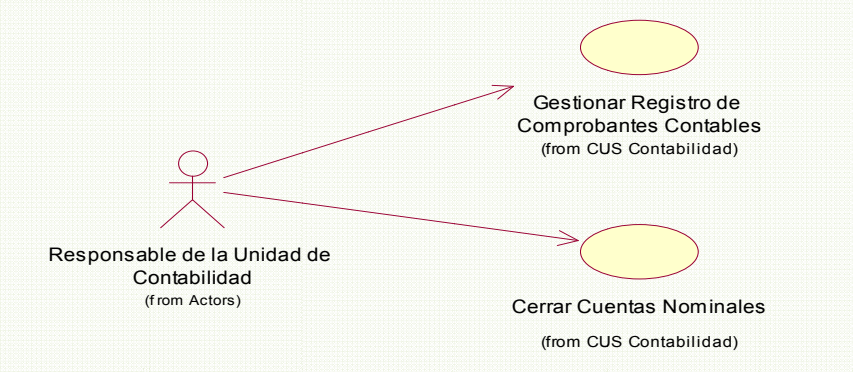


Figura 15 Vista de Casos de Uso módulo Contabilidad.

Gestionar Registro de Comprobantes Contables: Este caso de uso consiste en gestionar los comprobantes contables, visualizarlos en un registro tanto los manuales como los automatizados.

Cerrar Cuentas Nominales: Este caso de uso consiste en generar un comprobante para llevar el saldo de las cuentas nominales a cero.

2.7 Vista Lógica.

RUP propone para la Vista Lógica una representación del sistema a nivel de paquetes, subsistemas, clases significativas para la arquitectura, así como las principales relaciones que se establecen entre ellos.

Al igual que en la anterior, la vista lógica se corresponde a la primera iteración de la Arquitectura como se puede ver en la figura (ver figura 16).

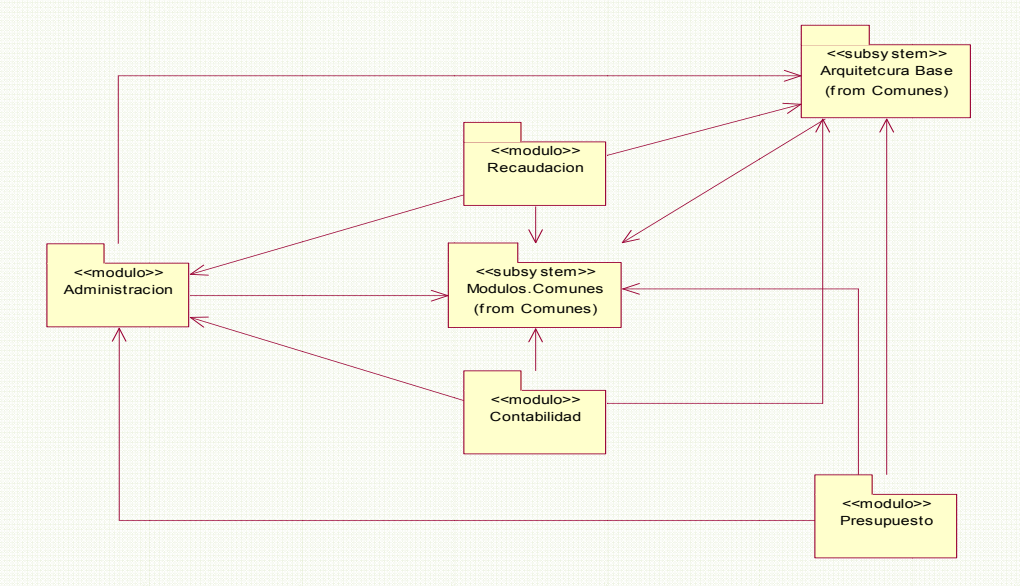


Figura16 Módulos correspondientes a la primera iteración de la Arquitectura del sistema.

En la figura anterior se agrupan los módulos en los paquetes donde se van a encapsular sus correspondientes funcionalidades. Así mismo se puede apreciar la presencia del subsistema Común y "ArquitecturaBase" que como se ha visto en secciones anteriores se refieren a algunas funcionalidades que dan soporte a la Arquitectura y como tal interactuarán con la totalidad de los paquetes y componentes que componen la solución.

Puesto que ya se tiene una visión general de esta iteración se hace necesario entrar en las particularidades de cada uno de estos paquetes. La figura (ver figura 17) muestra la distribución general correspondiente al modelo multicapas que se propone para cada uno de los módulos, con la excepción del Común cuya distribución se puede observar en la figura (ver figura 18).

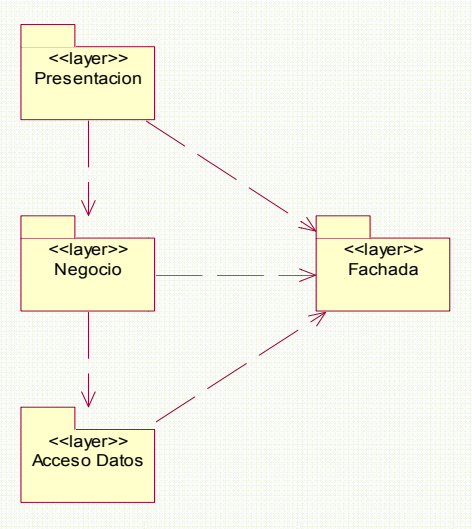


Figura 17 Representación del modelo multicapas módulos.

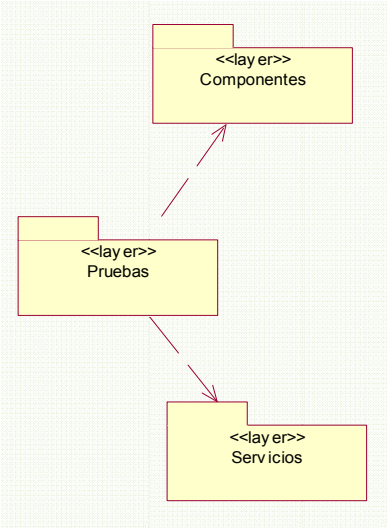


Figura 18 Representación del modelo multicapas módulo Común.

La estructura de paquetes de cada una de las capas que se aprecian en la figura (ver figura 17) se resume de la siguiente manera:

- **Presentación**:

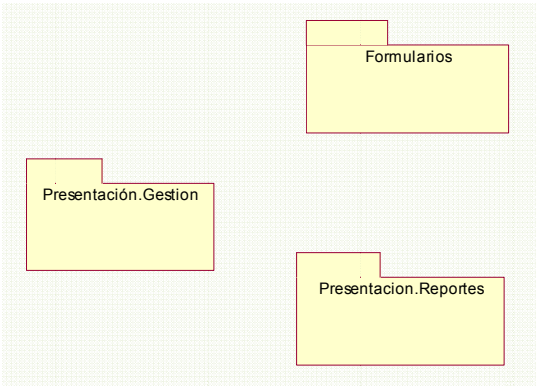


Figura 19 Estructura de la Capa de Presentación.

- El paquete correspondiente a los Formularios contiene las Interfaces de Usuario del módulo específico.
- El paquete correspondiente a la Gestión contiene las Acciones asociadas a cada una de las Interfaces de Usuario.
- El paquete correspondiente a los Reportes contiene los ficheros (.rpt) del Crystal Report para mostrar las estadísticas del sistema.

➢ **Negocio:**

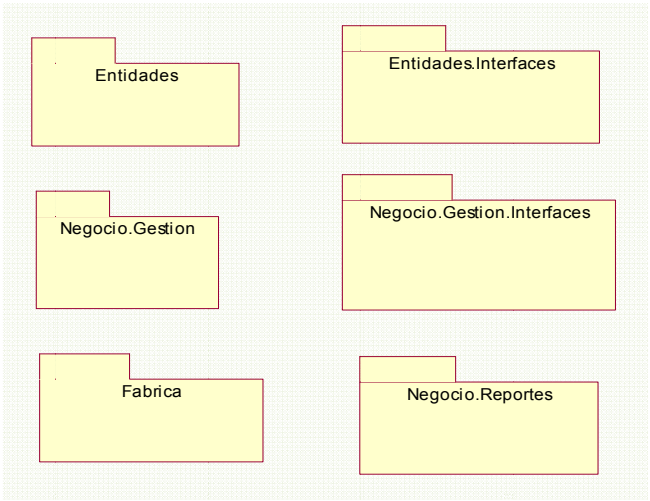


Figura 20 Estructura de la Capa de Negocio.

- El paquete correspondiente a las Entidades contiene las clases objeto-valor (clases persistentes).
- El paquete Entidades.Interfaces se corresponde con las interfaces que contienen las propiedades o los métodos que nos ofrecen las entidades.
- El paquete correspondiente a Negocio.Gestion contiene los Gestores de la aplicación, los cuales van a resolver la mayoría de las funcionalidades según los requisitos funcionales haciendo uso de las Entidades.

- El paquete de Gestion.Interfaces se corresponde con las interfaces que contienen los métodos/funcionalidades que ofrecen los Gestores.
- El paquete Fábrica contiene las funcionalidades que ofrece la Capa de Negocio a la Capa de Presentación a partir de una estructura de clases que en su conjunto dan vida el patrón Abstract Factory.
- Negocio.Reportes contiene las clases correspondientes a los datos que se quieren visualizar en los reportes.

➢ **Acceso a Datos:**

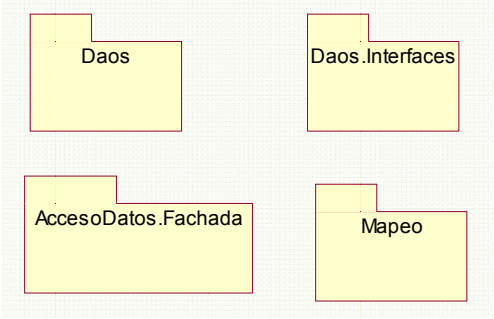


Figura 21 Estructura de la Capa de Acceso a Datos.

- El paquete correspondiente a los Daos contiene las clases que se encargan de las funcionalidades relacionadas con el acceso a datos, mayormente a través de procedenticos almacenados.
- Daos.Interfaces son las interfaces correspondientes a los métodos/funcionalidades que brindan los Daos.
- AccesoDatos.Fachada es un paquete que contiene una estructura específica de clases apoyada en Spring.net para que la Capa de Negocio interactué con los Daos.
- Mapeo contiene los ficheros XML de mapeo que utiliza NHibernate para la persistencia.

➢ **Fachada:**

Esta capa va a ser dinámica y por tanto puede presentar tantos paquetes como las funcionalidades que se quiera incorporar al módulo donde este implantada.

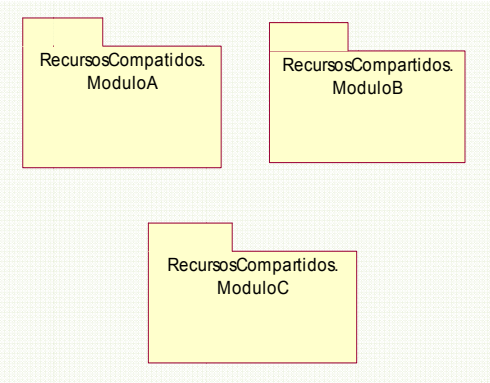


Figura 22 Estructura de la Capa de Fachada.

- Los paquetes correspondientes a los RecursosCompartidos contienen las funcionalidades que se quieren incorporar al módulo y toda la infraestructura para garantizarlas apoyándose fundamentalmente en el framework Spring.net.

En el caso del módulo Común, la estructura de paquetes de cada una de sus capas como se aprecian en la figura (ver figura 18) se resume de la siguiente manera:

- **Componentes:**

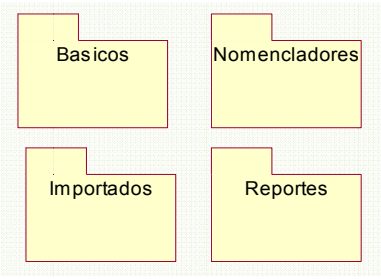


Figura 23 Estructura de la Capa de Componentes.

- El paquete Básicos contiene una librería de componentes que resultan básicos para el desarrollo del sistema fundamentalmente asociados a la Capa de Presentación.
- Nomencladores se refiere a los componentes nomencladores del sistema, es decir que trabajan con este tipo de datos que se encuentran en la Base de Datos.
- El paquete Importados contiene los componentes que se heredan de otras aplicaciones para ser reutilizados en la solución donde pueden ser modificados y actualizados por el equipo de desarrollo de acuerdo a las particularidades y necesidades de cada uno.
- Reportes contiene los componentes asociados al trabajo con las estadísticas del sistema.

➢ **Pruebas:**

Teniendo en cuenta que el módulo Común representa una fábrica de componentes y servicios para la Arquitectura se hace necesaria la presencia de una capa de Pruebas que sirva para validar y probar todos estos elementos.

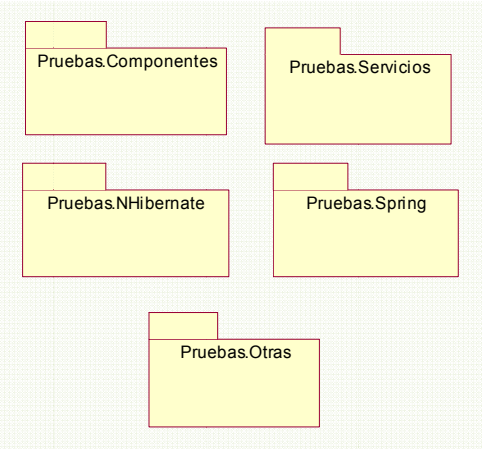


Figura 24 Estructura de la Capa de Pruebas.

- Pruebas.Componentes contiene toda la infraestructura para las pruebas que se van a realizar sobre la capa de componentes.

- Pruebas.Servicios contiene toda la infraestructura para las pruebas que se van a realizar sobre la capa de Servicios con la excepción de los framework open-source.
- Se crea un paquete Pruebas.NHibernate independiente para la pruebas sobre el framework open-source NHibernate puesto que el mismo requiere de una infraestructura más específica para verificar las modificaciones que se le realicen.
- Se crea un paquete Pruebas.Spring independiente para la pruebas sobre el framework open-source Spring.net puesto que este requiere también de una infraestructura más específica para verificar las modificaciones que se le puedan realizar.
- Pruebas.Otras contiene cualquier otro tipo de pruebas que se deseen realizar. Aquí se incluyen pruebas totales, pruebas que involucren otros elementos que no se encuentren en el módulo etc.

➢ **Servicios:**

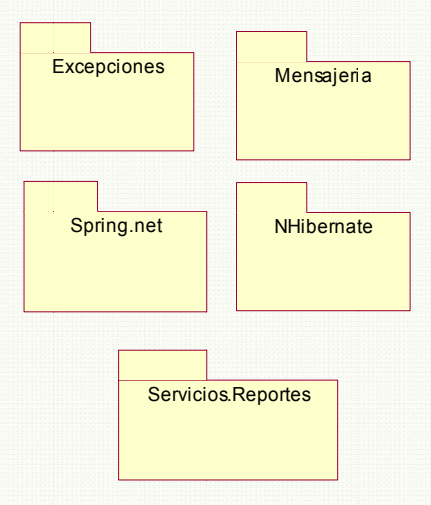


Figura 25 Estructura de la Capa de Servicios.

- Mensajería contiene la implementación de todo el servicio de mensajería del sistema, el cual se va a hacer centralizado en un fichero XML.

- Excepciones contiene la implementación de todo el servicio que garantiza un sencillo y adecuado trabajo con las excepciones del sistema, las cuales van a estar centralizadas en un fichero XML.
- Servicios.Reportes incluye la implementación de los servicios que ayuden al trabajo con los reportes.
- NHibernate este paquete contiene el código del framework NHibernate para realizarle modificaciones en caso de ser necesario.
- Spring.net este paquete contiene el código del framework Spring.net para realizarle modificaciones en caso de ser necesario

Para alcanzar un mayor grado de comprensión, a continuación se muestra en la figura (ver figura 26) la estructura que se ha ido viendo a nivel de clases y paquetes:

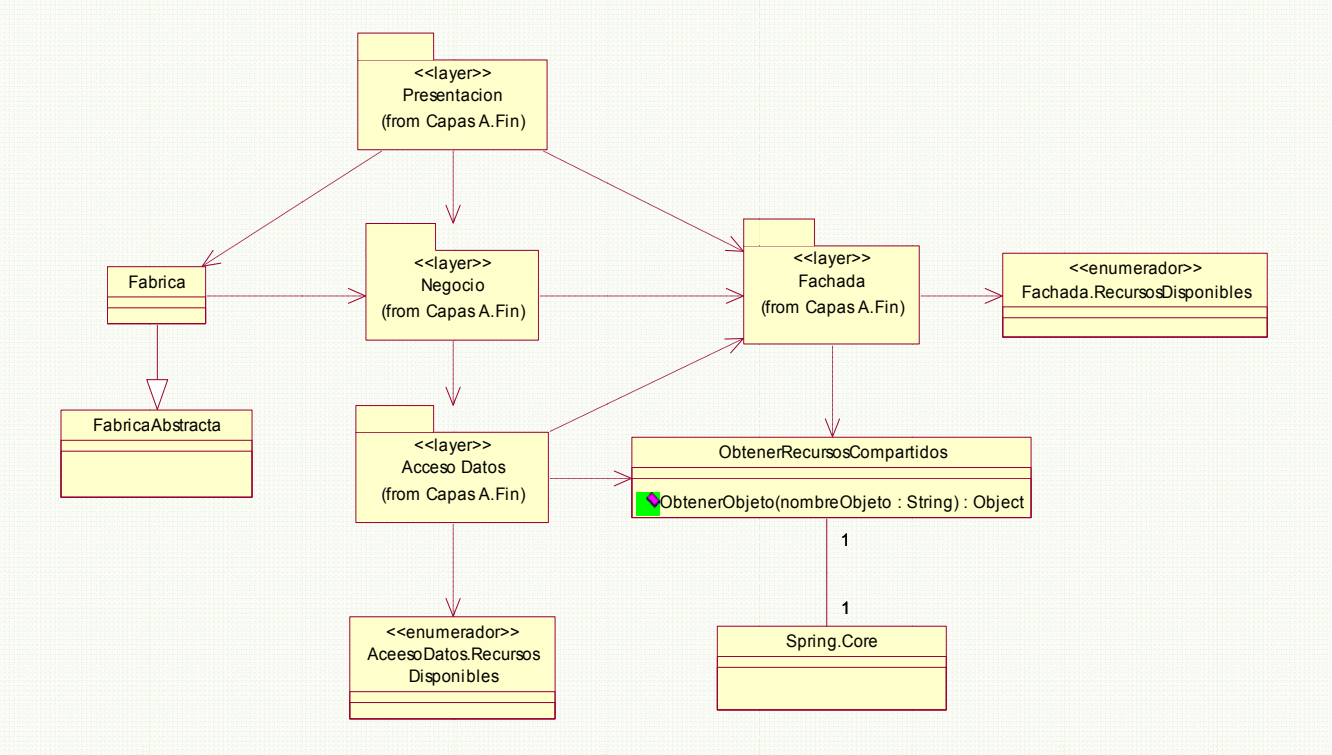


Figura 26 Estructura de clases para la Arquitectura planteada.

En la figura anterior se observa la presencia del patrón Abstract Factory que funciona de puente entre la Capa de Presentación y la de Negocio agrupando las clases de negocio en familias según las funcionalidades que representen. Por otra parte se puede ver la presencia de la Fachada como mecanismo de conexión con las funcionalidades que se encuentran en otros módulos según el enumerador (**enum**) que las relaciona y la clase que nos brinda el modulo Común (ObtenerRecursosCompartidos) que resuelve esta situación haciendo uso de Spring.net. Igualmente se resuelve la interacción de la Capa de Negocio con la de Acceso a Datos pero esta vez haciendo uso de la Fachada interna que presenta el Acceso a Datos.

2.8 Vista Implementación.

La vista de implementación del sistema contiene los componentes que se generan durante la compilación de cada uno de los paquetes y estructuras que se han ido viendo así como las dependencias que se establecen entre cada uno de ellos. Según RUP el propósito de esta vista es capturar las decisiones arquitectónicas para la implementación.

Puesto que la organización de componentes a partir de estas dependencias, ya sean de compilación, de inclusión es la misma para la totalidad del sistema, solamente se va a reflejar en esta vista aquellos módulos que introduzcan diferencias en los diagramas de componentes y que aporten mayor información al equipo de desarrollo.

Por tanto, la vista de implementación para la primera iteración se resume a los módulos de Administración y Presupuesto con las particularidades y especificidades de cada uno de ellos, además se incluye aunque en un menor grado el módulo “ArquitecturaBase”. No obstante para ganar en claridad y comprensión se hace alusión en caso de ser necesario al resto de los subsistemas.

El caso del módulo Administración es el más sencillo puesto que es el que menos dependencias genera en esta iteración. Como se puede ver en la figura (ver figura 27) existen dependencias con ficheros de configuración que necesita el framework Común, con otros que requiere NHibernate, Spring.net y finalmente los que pueda incorporar la solución relacionados fundamentalmente con el módulo Común. El

resto de las relaciones se obtienen de la Arquitectura vertical que deriva en los paquetes y finalmente en los componentes.

Por otra parte como se puede ver en la figura (ver figura 28) el módulo Presupuesto mantiene la misma estructura que el de Administración, solamente incorpora la Fachada puesto que requiere de algunas funcionalidades que en este caso se van a encontrar en Administración. Esta idea se mantiene para el caso de Contabilidad y Recaudación los cuales como se ha visto interactúan mayormente con Administración.

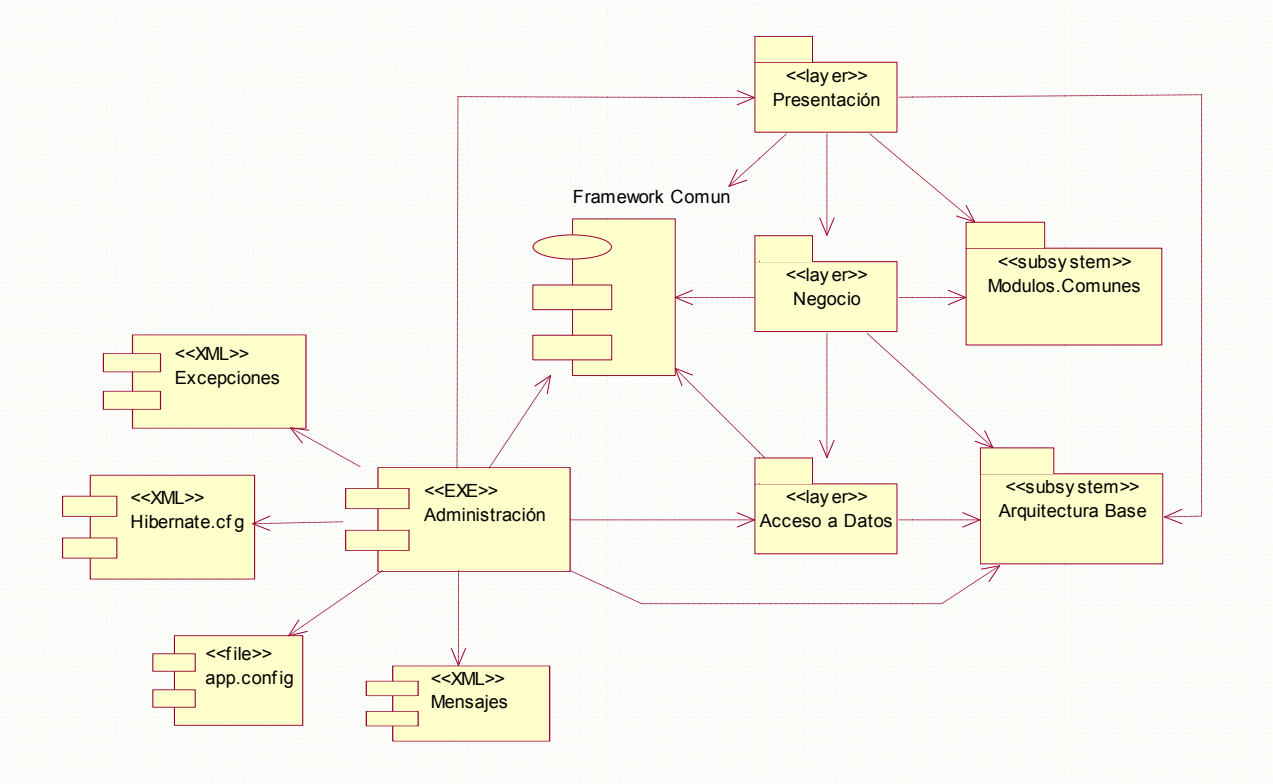


Figura 27 Representación de la Vista de Implementación módulo Administración.

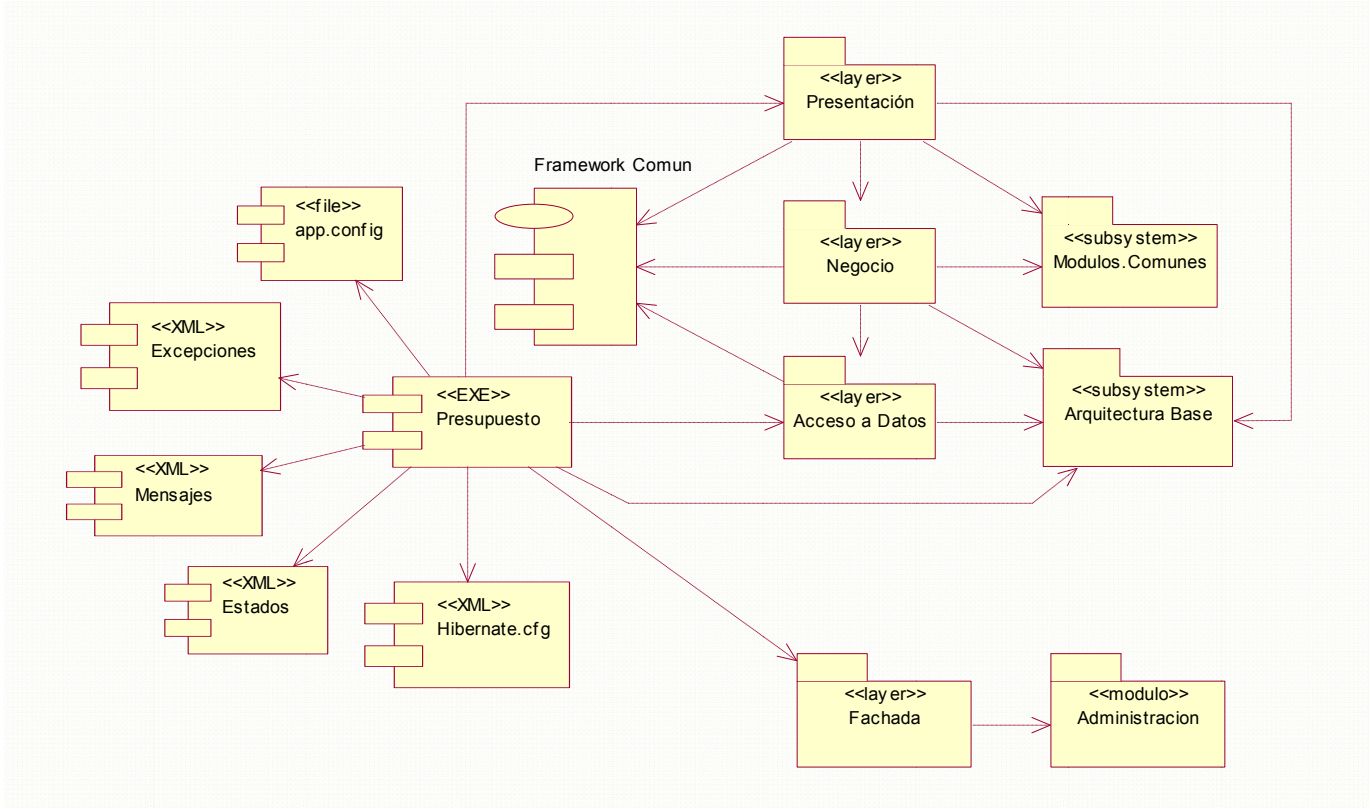


Figura 28 Representación de la Vista de Implementación módulo Presupuesto.

El caso del módulo Común es más específico. Su estructura como se ha visto en secciones anteriores es totalmente diferente del resto puesto que responde netamente a la Arquitectura del sistema. Sin embargo no genera mayores dependencias y la implementación que se realice sobre él no afecta el desempeño del sistema. Por tanto no cumple objetivo generar otra representación, en este documento, que no sea la que ya se obtuvo en la vista Lógica.

El módulo de “ArquitecturaBase” continúa con la misma línea de la solución como se puede ver en la figura (ver figura 29). No obstante solamente contiene aquellas funcionalidades concernientes al negocio que responden a la totalidad del sistema y lógicamente la infraestructura de integración del mismo. La Capa de Presentación no está presente puesto que la lógica relacionada con la interfaz de usuario está contemplada en los módulos que se van incorporando y en el framework (framework Común).

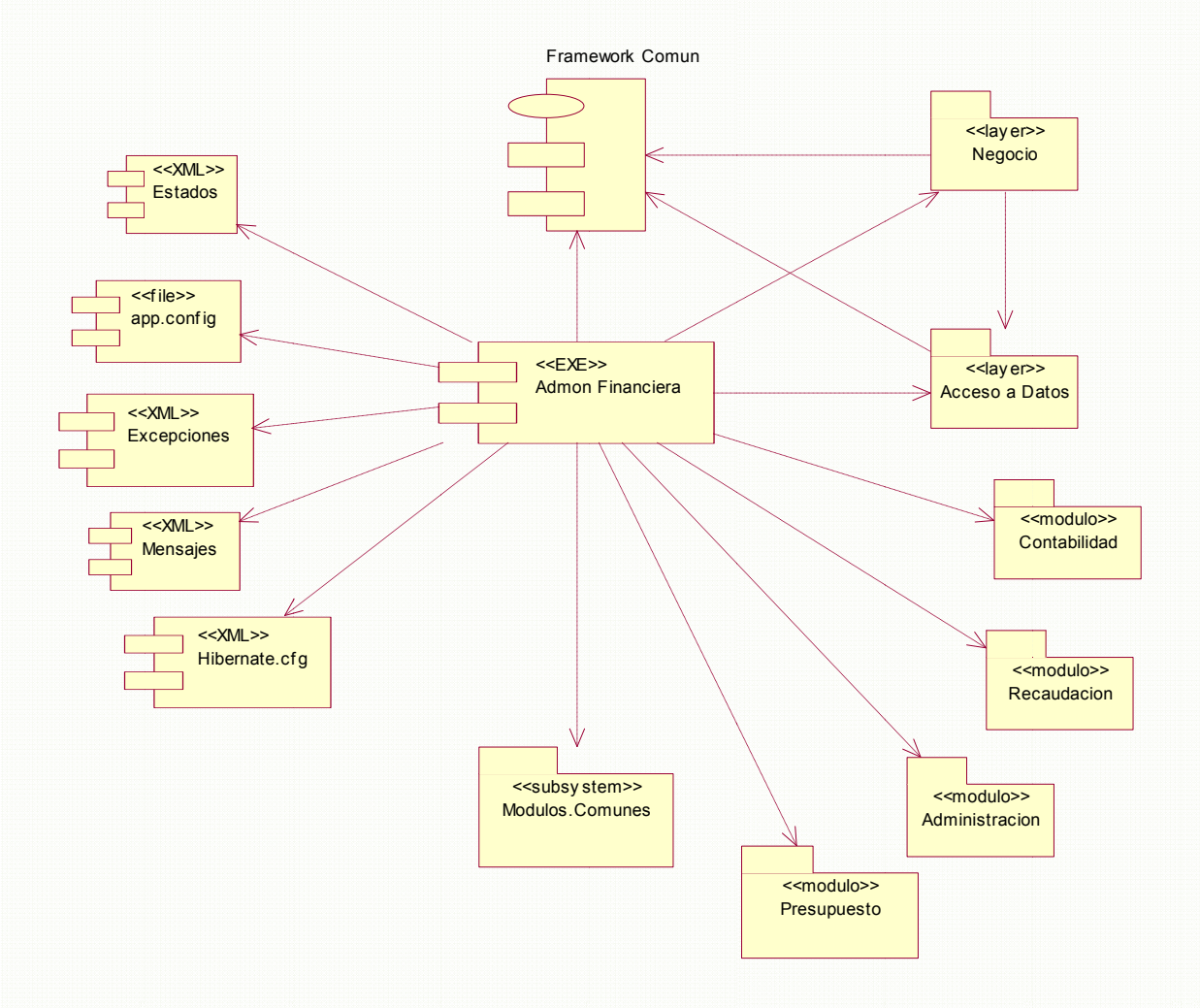


Figura 29 Representación de la Vista de Implementación módulo Arquitectura Base.

Con relación a los diagramas que se mostraron con anterioridad, se aprecian una serie de elementos que encapsulan las funcionalidades que a su vez se agrupan en otros componentes. Para ganar en claridad a continuación se visualizan cada uno de estos elementos con sus respectivos diagramas de componentes.

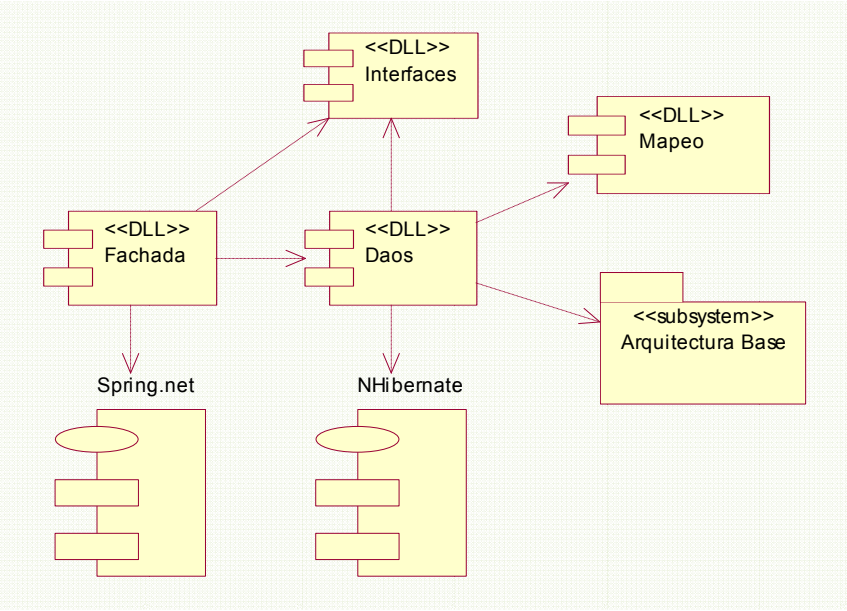


Figura 30 Vista de Implementación Capa de Acceso a Datos.

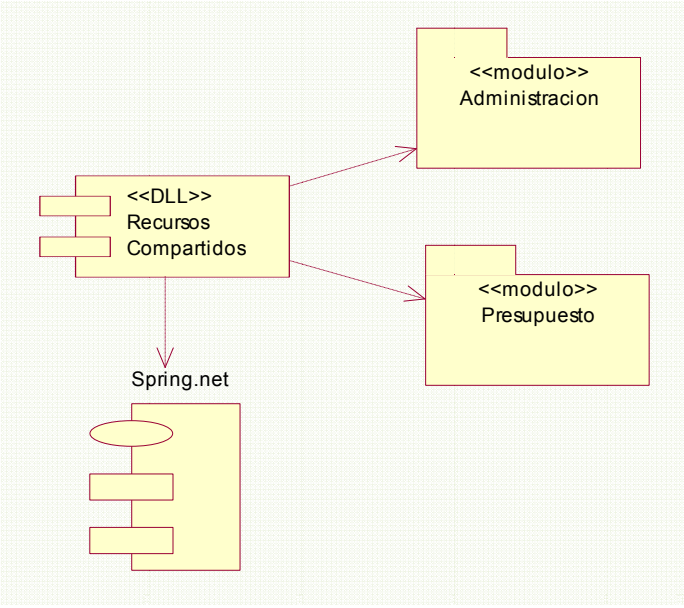


Figura 31 Vista de Implementación Capa de Fachada para un módulo que interactúe con Presupuesto y Administración.

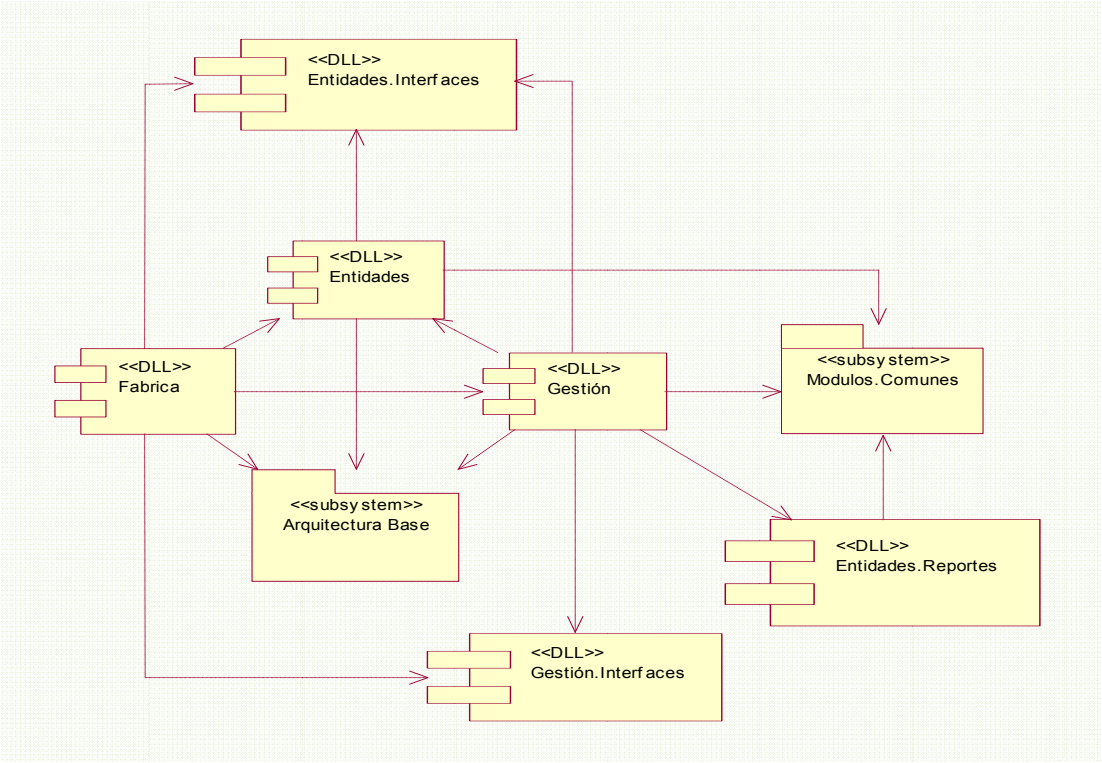


Figura 32 Vista de Implementación Capa de Negocio.

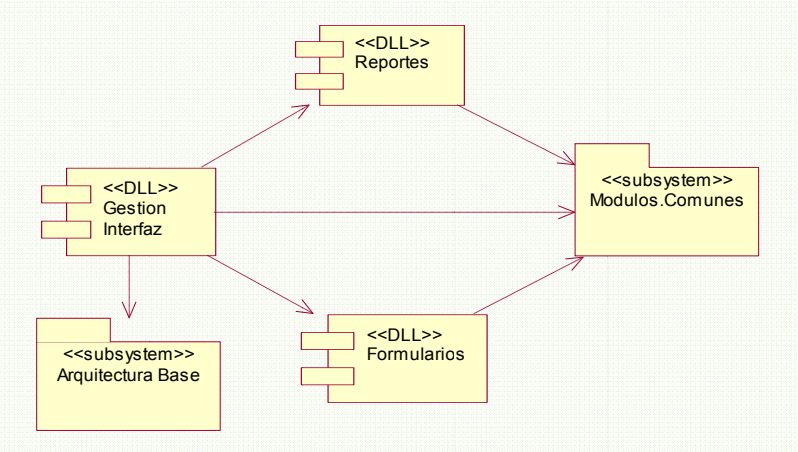


Figura 33 Vista de Implementación Capa de Presentación.

2.9 Vista de Despliegue.

Esta vista aborda la descripción de los nodos físicos para la mayoría de las configuraciones tanto para usuarios finales como para desarrolladores y probadores.

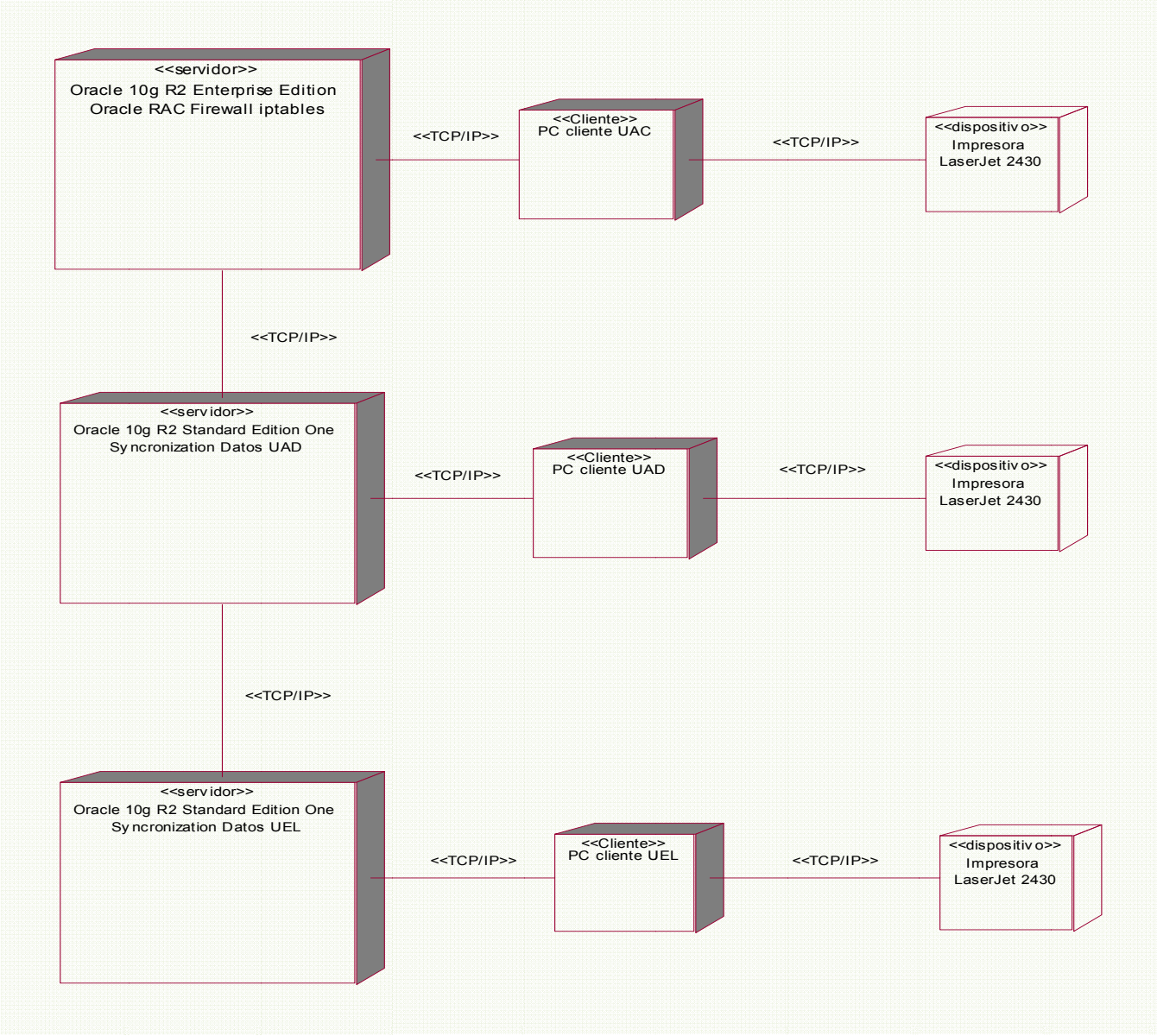


Figura 34 Vista de Despliegue del sistema Administración Financiera.

A continuación se describen los diferentes nodos y dispositivos que contiene la figura (ver figura 34):

Impresora: Este dispositivo garantiza las funcionalidades de impresión de reportes y documentos que genera la aplicación dependiendo de las funcionalidades de la máquina cliente que la utilice a través de la red.

PC Cliente: Este nodo se corresponde con la máquina donde se encuentra instalado el sistema de Administración Financiera con las funcionalidades que se correspondan con su localización en la estructura financiera para ese año, ya sea a nivel de UEL, UAD o UAC. En cada una de estas maquinas se va a encontrar instalado el Sistema Operativo Windows XP SP2 en Español.

Servidor Oracle 10g R2 Standard Edition UEL: Este es el servidor de base de datos que se encuentra en cada UEL. Está montado sobre el Sistema Operativo Windows Server 2003 y es el nodo al que la máquina cliente encuesta y solicita información. En caso de no ser satisfecha la solicitud, la misma se eleva al servidor correspondiente a la UAD a la cual está adscrita dicha UEL.

Servidor Oracle 10g R2 Standard Edition UAD: Este es el servidor de base de datos que se encuentra en cada UAD. Está montado sobre el Sistema Operativo Windows Server 2003 y es el nodo al que la máquina cliente o servidora de la UEL encuesta y solicita información. En caso de no ser satisfecha la solicitud, la misma se eleva al servidor que se encuentra en el Centro de Datos que representa a la UAC.

Servidor Oracle 10g R2 Enterprise Edition Oracle RAC Firewall iptables: Este es el servidor de base de datos que se encuentra en el Centro de Datos. Está montado sobre el Sistema Operativo RedHat AS v4.0 y es el nodo al que la máquina cliente o servidora de la UAD encuesta y solicita información. En este nivel es a donde van a llegar todas las solicitudes y por tanto apoyado en el mecanismo de réplica de datos aquí se encuentra registrada la totalidad de la información del sistema.

De manera general, para las UEL se utiliza una tecnología de red inalámbrica y entre ellas una red WAN con enlaces de alta velocidad y topología Estrella.

2.10 Conclusiones

La Arquitectura propuesta como se ha visto en las diferentes secciones de este capítulo tiene su base en l desarrollo de componentes y conectores. Cada uno de los elementos que incorpora se corresponde a uno de estos conceptos dependiendo del nivel de abstracción en el que nos encontremos.

Al culminar cada una de las iteraciones se obtienen actualizaciones de este documento que proporcionen mayor comprensión a los desarrolladores e interesados. Lo anterior incluye cuatro de las 5 vistas que propone la metodología utilizada y algunos conceptos/definiciones que puedan surgir en cada ciclo.

Igualmente para dar solución al objetivo general y los específicos en cada iteración se adoptan una serie de decisiones las cuales tiene su base en las restricciones y estrategias que presenta la arquitectura. Por tanto se hace necesario validar en primer lugar que los resultados sean los esperados y en segundo lugar que se respeten todas estas limitantes.

Capítulo 3

3. Resultados de la Arquitectura propuesta.

3.1 Introducción.

Los artefactos de arquitectura son decisivos en la calidad del software que se desarrolla. Su evaluación permite mitigar los diferentes riesgos asociados al desarrollo del software, mejorar la visión de los procesos críticos y validar las decisiones de diseño que se tomaron. Al ir evaluando los resultados obtenidos a partir de los objetivos que se propusieron se tiene la posibilidad de tomar acciones tempranas y valorar los atributos no funcionales (disponibilidad, desempeño, seguridad, interoperabilidad, mantenibilidad, etc.) sin esperar a que el software se construya.

Este capítulo constituye la rectificación y confirmación de todos los elementos que se han relacionado en la Línea Base propuesta. Para ello, se hace un análisis de los resultados obtenidos a partir de los escenarios arquitectónicos que se pueden encontrar y colapsen o provoquen un mal funcionamiento de la solución en un momento determinado.

3.2 Objetivos y Cualidades.

Los objetivos que se quieren alcanzar en la Arquitectura del sistema de Administración Financiera se refieren fundamentalmente a los atributos de calidad (Flexibilidad, Mantenibilidad, Escalabilidad, Integrabilidad, Interoperabilidad). Algunas de estas cualidades se refieren al diseño, sin embargo tienen su repercusión en la arquitectura y contribuyen a garantizar los objetivos.

En esta dirección es que se desarrolla la totalidad de la solución. Es decir estas cualidades marcan una línea entorno a la cual van a girar los procesos de desarrollo. A continuación se describe el significado de algunos de ellos para la Arquitectura.

Flexibilidad:

Se refiere a la posibilidad de variar el programa para adaptarse a requerimientos más amplios. Detrás de esta cualidad está implícita la idea de diseño para el cambio, para cumplir las metas de re-uso de código y arquitectura.

Permite personalizar el desenvolvimiento de nuevos componentes específicamente para atender las necesidades del sistema. Una arquitectura flexible permite poder modificar cualquiera de los demás componentes de la misma sin necesidad de cambios en las aplicaciones, ni en su interface con los demás elementos.

La flexibilidad en las aplicaciones, está dada por el concepto de "parametrización", donde se almacenan valores de configuración para gran cantidad de variables. En caso de requerir cambios en estas variables, basta con cambiarlas y estos se reflejan en el resto de la aplicación.

Reparabilidad y Mantenibilidad:

La Reparabilidad es la posibilidad de corregir los defectos del software con un limitado gasto de trabajo.

Por su parte la Mantenibilidad es similar a la anterior, pero no está vinculada a la solución de componentes sino a cambios que no aparecían en la especificación original o que fueron establecidos en forma incorrecta.

En el ciclo de vida de todo producto de software, el tiempo de mantenimiento es un componente importante del tiempo total, por lo que ambas cualidades son vitales en cualquier programa.

Escalabilidad:

La escalabilidad es la capacidad de un software o de un hardware de crecer, adaptándose a nuevos requisitos conforme cambian las necesidades del negocio.

3.3 Evaluación de los Resultados.

Si las decisiones arquitectónicas determinan los atributos de calidad del sistema, entonces es posible evaluar las decisiones arquitectónicas con respecto a su impacto sobre dichos atributos.

Muchos de estos atributos como los que ya se vieron, no pueden ser medidos directamente. Sin embargo, la experiencia señala que la Arquitectura de software propicia algunos de ellos (SEI, 2000; Bass et al., 1998). Por lo tanto, la arquitectura del software es clave para la calidad. Por esto, un análisis de la Arquitectura debe ser ejecutado para determinar cuán satisfactoria es para el propósito del sistema (Bass et al., 1998). Para realizar esta estimación de la calidad a partir de la Arquitectura, algunos métodos, tales como SAAM, ATAM, ABDM, ARID, etc., incorporan diferentes técnicas de evaluación. Estas técnicas incluyen los escenarios, la simulación, los modelos matemáticos, el prototipo, entre otros, los cuales permiten una evaluación temprana.

3.3.1 Software Architecture Analysis Method (SAAM)

Según Kazman et al. (2001), el Método de Análisis de Arquitecturas de Software (Software Architecture Analysis Method, SAAM) es el primero que fue ampliamente promulgado y documentado. El método fue originalmente creado para el análisis de la modificabilidad de una arquitectura, pero en la práctica ha demostrado ser muy útil para evaluar de forma rápida distintos atributos de calidad, tales como modificabilidad, portabilidad, escalabilidad e integrabilidad.

El método de evaluación SAAM se enfoca en la enumeración de un conjunto de escenarios que representan los cambios probables a los que estará sometido el sistema en el futuro. Como entrada principal, es necesaria alguna forma de descripción de la Arquitectura a ser evaluada. De acuerdo con Kazman, las salidas de la evaluación del método SAAM son las siguientes:

- Una proyección sobre la Arquitectura de los escenarios que representan los cambios posibles ante los que puede estar expuesto el sistema.

- Entendimiento de la funcionalidad del sistema, e incluso una comparación de múltiples arquitecturas con respecto al nivel de funcionalidad que cada una soporta sin modificación.

Con la aplicación de este método, si el objetivo de la evaluación es una sola arquitectura, se obtienen los lugares en los que la misma puede fallar, en términos de los requerimientos de modificabilidad. Para el caso en el que se cuenta con varias arquitecturas candidatas, el método produce una escala relativa que permite observar qué opción satisface mejor los requerimientos de calidad con la menor cantidad de modificaciones.

3.3.2 Architecture Tradeoff Analysis Method (ATAM)

Este método de evaluación obtiene su nombre no solo porque refleja cuán bien una arquitectura particular satisface las metas de calidad, sino que también provee ideas de cómo esas metas de calidad interactúan entre ellas y como realizan concesiones mutuas (tradeoff).

El método consta de nueve pasos, divididos en cuatro grupos:

- Presentación
- Investigación y Análisis.
- Pruebas.
- Informes.

Paso 1: Presentar el ATAM:

- Los pasos del ATAM en resumen.
- Las técnicas que serán utilizadas para la obtención y análisis.
- Las salidas de la evaluación.

Paso 2: Presentar las pautas del negocio:

- Las funciones más importantes del sistema.
- Toda restricción técnica.
- La mayoría de los stakeholders.
- Las guías de la arquitectura.

Paso 3: Presentar la arquitectura:

- Las restricciones técnicas.
- Otros sistemas.
- Propuestas arquitectónicas.

Paso 4: Identificar las propuestas arquitectónicas:

El ATAM centraliza el análisis de una arquitectura en el entendimiento de sus propuestas arquitectónicas, en este paso son capturadas por el equipo de evaluación pero no son analizadas

Paso 5: Generar el árbol de utilidad de los atributos de calidad:

Este paso es crucial, pues guía el resto del análisis. Sin esta guía, los evaluadores pueden perder su tiempo analizando aspectos de la arquitectura que no son de interés para los stakeholders.

Paso 6: Analizar las propuestas arquitectónicas.

En este paso se analizan los escenarios de las propuestas arquitectónicas.

Paso 7: Lluvia de ideas y priorización de escenarios:

Este paso consiste en la generación de nuevos escenarios para:

- Representar los intereses de los stakeholders que no hayan sido comprendidos.

Paso 8: Analizar las propuestas arquitectónicas:

En este paso el equipo de evaluación realiza las mismas actividades que en el paso 6, mapeando los escenarios recientemente generados con ranking más alto en los artefactos arquitectónicos.

Paso 9: Presentar los resultados.

- El documento de propuestas arquitectónicas.
- El conjunto de escenarios priorizados.
- El árbol de utilidad.
- Los riesgos descubiertos.
- Los no riesgos documentados.
- Los sensitivity points y tradeoff points encontrados.

3.3.3 Métodos de validación en la Arquitectura planteada

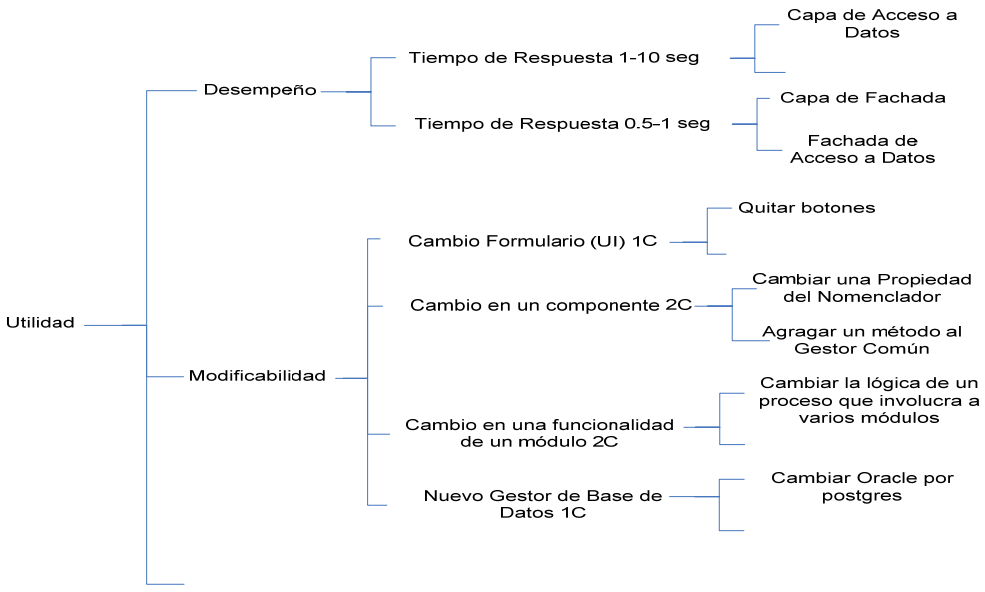
Si se tiene en cuenta que SAAM se centra fundamentalmente en el atributo de calidad modificabilidad y que además con su aplicación se garantiza que los interesados comprendan con mayor facilidad la arquitectura que se está evaluando, la documentación quede mejorada y que el esfuerzo y costo de los cambios se pueden estimar con anticipación, entonces resulta atractivo aplicarlo en la arquitectura propuesta. Sin embargo existen una serie de atributos y resultados que pueden quedar fuera de la validación si aplicáramos solamente el proceso anterior. Por tanto se decide implementar una variante propia que contempla algunas de los puntos comprendidos en los pasos de ATAM vinculadas con la propuesta que desarrolla SAAM. El hecho de seleccionar ATAM es precisamente porque este aborda aspectos y atributos que quedan más débiles en SAAM.

Para llevar a efecto esta variante se toma como punto de partida toda la descripción de la arquitectura que fue objetivo del Capítulo 2, la cual constituye una entrada para SAAM y favorece algunos de los pasos de ATAM. En este sentido se ofrecen las principales funcionalidades, algunas restricciones y una guía de la arquitectura. Además se hace un análisis de los principales escenarios arquitectónicos que tributan

directamente a situaciones de interés, si se tiene en cuenta los atributos de calidad que se quieren priorizar principalmente los que se relacionaron en la sección anterior.

Para complementar algunos de los pasos de ATAM se hace necesario además definir otros sistemas que interactúan con la solución. Tal es el caso de el módulo Mercantil y Público de la solución general de Registros y Notarias, estas aplicaciones se ven influenciadas mayormente en la interoperabilidad del software y forman parte de las restricciones técnicas conjuntamente con la integrabilidad entre los módulos propios del sistema de Administración Financiera y otras cuestiones relacionadas fundamentalmente con la comunicación y la navegación en el mismo.

Para ganar en claridad a continuación se hace una representación del árbol de utilidad referente a los atributos de calidad Desempeño y Modificabilidad. Aquí se introduce una nueva cualidad (Desempeño) que puede verse afectada en gran medida sobre todo por el uso de Spring.net y lógicamente con la utilización de NHibernate para el acceso a datos.



Leyenda

Seg Segundos que debe durar la operación.

C Componente Involucrado.

A continuación se describen los principales escenarios arquitectónicos que se pueden encontrar y se puntualizan todas las ideas que se han ido viendo.

Escenario	Cuando la Capa de Presentación le solicita información a la Capa de Negocio.
Objetivos del Negocio	Procesar alguna información.
Atributo de Calidad	Mantenibilidad, modificabilidad.
Estímulo	Obtener una colección de objetos.
Origen del Estímulo	Un formulario.
Elemento	Un botón.
Ambiente	Se está realizando un proceso de búsqueda.
Respuesta	Se muestra en el formulario la colección de objetos.
Medida de la Respuesta	Nivel de abstracción, repercusión.
Preguntas	¿Cuáles y cómo se afectan los componentes involucrados en la petición?
Anotación	Se requiere del uso de interfaces y la correcta implementación del patrón Abstract Factory.

Tabla 7 Escenario correspondiente a la comunicación entre la Capa de Presentación y Negocio.

La comunicación entre dos capas sin lugar a dudas marca un escenario que hay que tener en cuenta, precisamente porque en el mismo se encuentra el grado de dependencia que va a existir entre las capas involucradas. Por tal motivo mientras menos componentes se afecten en la operación más se favorecerá la mantenibilidad, modificabilidad y lógicamente la reparabilidad puesto que un cambio en alguna de los niveles implicados no resultará tan significativo.

En el caso de la comunicación entre Presentación y Negocio (ver tabla 7) se utiliza el patrón Abstract Factory para la conexión e interfaces que representan las funcionalidades del negocio. El patrón abstrae en alguna medida ambas capas aunque no en un 100%, esto se debe a que el Negocio y la Presentación no son tan críticos para la arquitectura. No obstante en caso de verse afectado algún componente de negocio solamente habría que actualizar la fabrica (Patrón Abstract Factory) puesto que la Presentación lo

único que conocerá es de esta y de las interfaces de negocio, facilitando así el mantenimiento y modificabilidad de la aplicación.

Por otra parte la Capa de Negocio y de Acceso a Datos (ver tabla 8) se comunican a través de una fachada que presenta la segunda, implementada sobre Spring.net. En este escenario si se le presta especial atención a tener el 100% de desacoplamiento puesto que la Capa de Datos es bastante crítica y propensa a cambios. Esta situación se resuelve con Spring.net que bajo su concepto de IoC y el uso de interfaces, limita la conexión entre ambas capas a un fichero XML permitiendo que la implementación del Acceso a Datos sea totalmente ajena al Negocio. Como se puede observar los componentes que se afectan en este nivel son mínimos y por tanto el sistema se hace altamente mantenible y flexible. Cualquier cambio y crecimiento en las implementaciones involucraría mayormente un fichero XML de configuración y no el código fuente.

Escenario	Cuando la Capa de Negocio le solicita información a la Capa de Acceso a Datos.
Objetivos del Negocio	Procesar algún dato para persistirlo u obtenerlo de la Base de Datos.
Atributo de Calidad	Mantenibilidad, modificabilidad, flexibilidad.
Estímulo	Obtener una colección de objetos.
Origen del Estímulo	Un gestor.
Elemento	Un método.
Ambiente	Se está realizando un proceso de búsqueda.
Respuesta	Se retorna la colección de objetos.
Medida de la Respuesta	Nivel de abstracción, repercusión.
Preguntas	¿Cuáles y cómo se afectan los componentes involucrados en la petición?
Anotación	Se requiere del uso de interfaces, la correcta implementación del patrón Fachada sobre Spring.net y el framework NHibernate para la interacción con la Base de Datos.

Tabla 8 Escenario correspondiente a la comunicación entre la Capa de Negocio y Acceso a Datos.

La comunicación entre los módulos de la solución (ver tabla 9) se realiza a través de una fachada la cual está implementada sobre Spring.net y mantiene la misma línea que la de la Capa de Datos aunque lógicamente en un nivel superior. Como ya se sabe, esta responsabilidad la tiene la Capa de Fachada que es donde van a radicar las implementaciones de los módulos que se necesiten cuando se vaya a acceder a sus funcionalidades, en este momento se le pide a la fachada que instancie el objeto.

Escenario	Cuando un módulo necesita comunicarse con otro para solicitar algún tipo de información o utilizar alguna funcionalidad.
Objetivos del Negocio	Llevar a cabo una operación que se salga del dominio del módulo.
Atributo de Calidad	Mantenibilidad, modificabilidad, flexibilidad, escalabilidad, integrabilidad.
Estímulo	Contabilizar.
Origen del Estímulo	Un gestor.
Elemento	Un método.
Ambiente	Se está llevando a cabo la contabilización de una operación contable.
Respuesta	Se registra la contabilización en un comprobante contable.
Medida de la Respuesta	Nivel de abstracción, repercusión, capacidad de incorporación de funcionalidades.
Preguntas	¿Cómo comparto los componentes de un módulo para llevar a cabo la operación? ¿Qué implica un cambio en alguno de estos componentes? ¿Cuáles componentes del módulo en que se está trabajando se ven afectados en el proceso?
Anotación	Se requiere del uso de interfaces, la correcta implementación del patrón Fachada sobre Spring.net.

Tabla 9 Escenario correspondiente a la comunicación entre módulos.

Existen funcionalidades y servicios que son comunes para la solución en general (ver tabla 10), por tal motivo se decide organizarlos en componentes y servicios según sea el caso. Estos elementos se aplican a casi todos los niveles de abstracción y justamente al encapsular determinadas funcionalidades en ellos, se logra que dependiendo del nivel en que se encuentre, el desarrollo quede cohesionado y eficiente. Así

mismo se obtiene una gran capacidad para la reparabilidad y la mantenibilidad puesto que un componente es independiente de otro en el sentido de la responsabilidad que tiene que asumir en el negocio, lo que posibilita que ante cualquier cambio solamente se tendría que trabajar sobre el elemento que se ve afectado directamente, en ningún momento se compromete el resto del código y como resultado disminuye exponencialmente el gasto de trabajo.

Otra potencialidad que brindan es la reutilización de código, en efecto al tener encapsulada cierta funcionalidad, resulta muy útil la reutilización de las mismas con solo incorporar el componente en cuestión, lo que significa un adelanto significativo en la implementación.

Escenario	Cuando varios módulos necesitan los mismos servicios y funcionalidades.
Objetivos del Negocio	Capturar todas las excepciones de sistema centralizadamente y por ende bajo una misma línea. Validar todos los datos de los formularios de una manera óptima y fácil.
Atributo de Calidad	Mantenibilidad, modificabilidad, flexibilidad, escalabilidad, reusabilidad.
Estímulo	Recoger los datos de una interfaz y validar que no haya campos vacios. En tal caso lanzar una excepción.
Origen del Estímulo	Una acción.
Elemento	Un botón.
Ambiente	Se está construyendo un objeto con los datos de la interfaz de usuario.
Respuesta	Se construye el objeto con los datos capturados.
Medida de la Respuesta	Esfuerzo. Tiempo de trabajo.
Preguntas	¿Qué componentes utilizar? ¿Cómo configurar los componentes? ¿Cómo utilizar los servicios? ¿Dónde desarrollar los componentes?
Anotación	Se requiere de una capacitación y la correcta configuración de los componentes y servicios.

Tabla 10 Escenario correspondiente a utilización de componentes y servicios.

La infraestructura de integración (ver tabla 11) descrita se apoya en un fichero XML y en el uso de las interfaces a través del patrón Reflection. Por lo mismo, sin importar cuánto se repare modifique o crezca el sistema de Administración Financiera, esto no resultaría significativo para quien este interactuando con este.

En este nivel también existe un 100% de desacoplamiento puesto que el agente externo solamente conocerá de interfaces, la implementación de las mismas las garantizará la "ArquitecturaBase" a través de un proxy.

Escenario	Cuando el sistema tiene que interactuar con aplicaciones externas.
Objetivos del Negocio	Brindar los datos y funcionalidad que le soliciten al sistema.
Atributo de Calidad	Mantenibilidad, modificabilidad, interoperabilidad.
Estímulo	Solicitud de datos referentes a la recaudación.
Origen del Estímulo	Un sistema externo.
Elemento	Un método.
Ambiente	Se está llevando a cabo un proceso que necesita datos y funcionalidades de la recaudación controlados por nuestro sistema.
Respuesta	Se devuelven los datos solicitados.
Medida de la Respuesta	Nivel de abstracción, repercusión.
Preguntas	¿Cómo comparto los componentes de mi sistema para llevar a cabo la operación? ¿Qué implica un cambio en alguno de estos componentes?
Anotación	Se requiere del uso de interfaces, la correcta implementación del patrón Proxy y Reflection combinados.

Tabla 11 Escenario correspondiente a la integración.

Los resultados reflejan la escalabilidad en gran medida en el módulo "ArquitecturaBase", el cual facilita el crecimiento y adaptación a nuevos requerimientos del negocio puesto que como se ha visto proporciona la infraestructura donde descansan los módulos. Así mismo durante cada una de las iteraciones que se han

reflejado en este documento, la arquitectura va ganando en funcionalidades sin que esto sea traumático o significativo para la misma.

De manera general es importante destacar que el hardware según se vio en la vista de despliegue (ver figura 34) contribuye a la escalabilidad y robustez de la arquitectura si tenemos en cuenta que existe un servidor local en cada UEL que garantizará, en caso de perder conectividad, que se atiendan todas las peticiones que se puedan realizar sin verse afectado el sistema. También se asegura el respaldo de la información mediante mecanismos de réplica de datos y así mismo liberar la tensión y la sobrecarga de la red y el servidor.

Por otra parte, puesto que el proceso de implementación ya culminó la primera iteración, la arquitectura fue probada y sometida a diversas situaciones de stress que posibilitaron perfeccionarla y validarla en alguna medida, haciendo un análisis de los resultados esperados. Más aún si se tiene en cuenta que un método cuantitativo para hacerlo es la simulación de la misma.

A partir de todos los elementos que se han reflejado, se puede afirmar que la solución propuesta es totalmente flexible, modificable, reparable y mantenible. Igualmente los factores que aseguran estas cualidades influyen directa o indirectamente en la integrabilidad, interoperabilidad y la escalabilidad, aunque en este caso, el hardware también juega su papel.

3.4 Conclusiones.

Alcanzar un atributo de calidad puede tener un efecto positivo o negativo sobre otros atributos de calidad. Por tanto el Arquitecto de software tiene que tener claros aquellos que considere de mayor prioridad dependiendo de los resultados que desea obtener Algunos atributos de calidad deben ser diseñados y evaluados a nivel arquitectónico. Otros no son susceptibles de ser alcanzados a nivel arquitectónico, sin embargo en su conjunto tributan a darle solución a los objetivos que se persiguen.

Particularmente esta Línea Base garantiza la flexibilidad, la mantenibilidad y la escalabilidad por encima de otras cualidades que aunque algunas no dejan de estar presentes si adquieren un carácter secundario respecto a estas.

Conclusiones

La Arquitectura es una pieza importante en el ciclo de vida de un software, más aún teniendo en cuenta que la metodología seleccionada (RUP) se centra en esta para su desarrollo. Por tal motivo es importante prestar especial atención al estado de la documentación y las tecnologías existentes actualmente en el mundo referentes a esta materia. A partir de la investigación resultaron conceptos y tecnologías importantes:

- Los relacionados con la disciplina: el concepto de Arquitectura de software, conectores, componentes, vistas y notaciones.
- Los Patrones y Estilos de Arquitectura.
- La calidad arquitectónica y atributos de calidad.
- Framework de desarrollo existentes (NHibernate, Spring.net).

Así mismo, definir la Línea Base de la Arquitectura del sistema de Administración Financiera posibilitó llegar a un entendimiento entre todos los involucrados en el proceso de desarrollo, así como establecer un marco idóneo para representar todo los componentes y elementos que la conforman. En este sentido:

- Se describieron los conceptos, terminologías y el entorno donde se va a desenvolver la Arquitectura del sistema.
- Se describió la arquitectura desde dos enfoques Vertical y Horizontal.
- Se establecieron los componentes más significativos y como se implantan estos en la solución (Capas, Módulos, Clases).
- Se describieron las políticas de integración y los mecanismos que hay que asegurar para lograrla (Patrón Proxy, Reflection).
- Se describieron 4 de las 5 vistas propuestas por la metodología RUP (Vista de Casos de Uso, Vista Lógica, Vista de Implementación, Vista de Despliegue) acorde a los procesos que se encuentran en la solución

Cada uno de los artefactos y elementos que se mencionaron anteriormente se enmarcan en 3 iteraciones durante las cuales se va refinando este documento hasta obtener una versión final.

Por otra parte, después de haber obtenido un modelo para la arquitectura del sistema, fue necesario efectuar la validación del mismo para analizar si efectivamente respondía a los atributos de calidad que se desean obtener, para lograr el objetivo y como parte de este proceso:

- Se definieron cuales son los atributos de calidad que se desean priorizar en la Arquitectura del sistema (mantenibilidad, modificabilidad, flexibilidad).
- Se seleccionó como métodos para validar la Arquitectura propuesta SAAM y ATAM los cuales se conjugaron en una variante menos rigurosa pero que garantizara el objetivo fundamental de esta etapa.
- Se definieron los escenarios principales para la Arquitectura del sistema los cuales abarcan la comunicación entre capas, la utilización de componentes, la comunicación entre módulos y la integración con sistemas externos.

Recomendaciones

Con el propósito de ampliar y mejorar la documentación de la arquitectura presentada en este trabajo, se plantean las siguientes recomendaciones:

- No se mencionan características de los servidores como: balanza de carga, pool connection, seguridad de datos, mecanismos de protección contra ataques en la base de datos o modificación de los registros. Estos factores son muy importantes por tanto se recomienda incluirlos en futuras iteraciones o versiones de este documento.

- De manera general no se desarrollan los temas relacionados con la seguridad que entre otras cosas incluyen principios de la programación segura. Si bien se refleja en el documento que la seguridad se garantiza con la utilización del framework Común y algunos puntos heredados de la Arquitectura General si es importante incluir en este trabajo algunas particularidades específicas del sistema de Administración Financiera.

- Se recomienda manejar los temas relacionados con la encriptación, la transferencia de datos entre capas y entre subsistemas y siguiendo el punto anterior como se garantizan los atributos de calidad de seguridad mínimos en la arquitectura.

- Se recomienda continuar profundizando en la validación de la arquitectura propuesta. En este sentido sería bueno aplicar la totalidad del método ATAM puesto que este es mucho más completo que SAAM y por tanto se obtiene un resultado mucho más acabado.

Referencias Bibliográficas

Barbacci, M., Klein, M., Longstaff, T., & Weinstock, C. (1995). Quality Attributes. Carnegie Mellon University. Technical Report.

Bass, L., Barbacci, M., Carriere, J., Kazman, R., Klein, M., y Lipson, H. (1999). Attribute Based Architectural Styles. Software Engineering Institute, Carnegie Mellon University. Pittsburgh.

Bass, L., Clements, P., & Kazman, R. (1998). Software Architecture in practice. Addison-Wesley. Bass, L., Klein, M., & Bachmann, F. (2000). Quality Attribute Design Primitives. Software Engineering Institute, Carnegie Mellon University.

Canal, Carlos. (marzo 2005). Conferencia: “Arquitectura, marcos de trabajo y patrones”. Universidad de Málaga.

Larman, Craig [2004] (2005). Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd, Prentice Hall PTR.

[G95] Gamma, E., et al., Design Patterns: Elements of Reusable Object-Oriented Software. 1995.

Booch, G., Rumbaugh, J., & Jacobson, I. (1999). The UML Modeling Language User Guide. Adisson-Wesley Bosch, J. (2000). Design & Use of Software Architectures. Addison-Wesley. Bredemeyer, D., & Malan, R. (2002). The Visual Architecting Process. White Paper.

Longman, Inc. Lane, T. (1990). Studying Software Architecture Through Design Spaces and Rules. Technical report. The Computer Science Department, Carnegie Mellon University.

Larman, C. (1999). UML y Patrones: Introducción al análisis y diseño orientado a objetos. Prentice-Hall Hispanomericana. Losavio, F., Chirinos, L., Lévy, N., & Ramdane-Cherif, A. (2003). Quality Characteristics for Software Architecture. A ser publicado en el JOT 2003.

Perry, D., & Wolf, A. (1992). Foundations for the Study of Software Architecture. ACM Sigsoft - Software Engineering Notes, Vol 17, No. 4.

Pressman R. (2002) Ingeniería de Software. Un Enfoque Práctico. Quinta Edición. Mc Graw Hill. OTI. (2003) Eclipse Platform Technical Overview.

Rational Software Corporation. (1998). "Rational Unified Process: Best Practices for Software Development Teams".

Barbacci, M., Klein, M., Longstaff, T., & Weinstock, C. (1995). *Quality Attributes.* Carnegie Mellon University. Technical Report.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). Pattern – Oriented Software Architecture. A System of Patterns.

John Wiley & Sons, Inglaterra. Carriere, J., Kazman, R., Woods, S. (2000). Toward a Discipline of Scenariobased Architectural Engineering. Software Engineering Institute, Carnegie Mellon University.

Grady, R., & Caswell, D. (1987) Software Metrics: Establishing a company-Wide Program. Prentice Hall.

Kazman, R. (1996). Tool Support for Architecture Analysis and Design. Department of Computer Science, University of Waterloo.

Kazman, R., Clements, P., Klein, M. (2001). Evaluating Software Architectures. Methods and case studies. Addison Wesley. Kruchten, P. (1999). The Rational Unified Process. Reading, MA: Addison Wesley.

[KC94] Paul Kogut y Paul Clements. "Features of Architecture Description Languages". Borrador de un CMU/SEI Technical Report, Diciembre de 1994.

[KC95] Paul Kogut y Paul Clements. "Feature Analysis of Architecture Description Languages". En Proceedings of the Software Technology Conference (STC'95), Salt Lake City, Abril de 1995.

[Ves93] Steve Vestal. “A cursory overview and comparison of four Architecture Description Languages”. Technical Report, Honeywell Technology Center, Febrero de 1993.

[LV95] David Luckham y James Vera. “An Event-Based Architecture Definition Language”. IEEE Transactions on Software Engineering, pp. 717-734, Setiembre de 1995.

[SDK+95] Mary Shaw, Robert DeLine, Daniel Klein, Theodore Ross, David Young y Gregory Zelesnik. “Abstractions for Software Architecture and Tools to Support Them”. IEEE Transactions on Software Engineering, pp. 314-335, Abril de 1995.

[SG94] Mary Shaw y David Garlan. “Characteristics of Higher-Level Languages for Software Architecture”. Technical Report CMU-CS-94-210, Carnegie Mellon University, Diciembre de 1994.
[SG95] Mary Shaw y David Garlan. “Formulations and Formalisms in Software Architecture”. Springer-Verlag, Lecture Notes in Computer Science, Volumen 1000, 1995.

[Med96] Neno Medvidovic. “A classification and comparison framework for software Architecture Description Languages”. Technical Report UCI-ICS-97-02, 1996.

[Mon98] Robert Monroe. “Capturing software architecture design expertise with Armani”. Technical Report CMU-CS-163, Carnegie Mellon University, Octubre de 1998.

[GMW00] David Garlan, Robert Monroe y David Wile. “Acme: Architectural description of component-based systems”. Foundations of Component-Based Systems, Gary T. Leavens y Murali Sitaraman (eds), Cambridge University Press, pp. 47-68, 2000.

[MK96] Jeff Magee y Jeff Kramer. “Dynamic structure in software architectures”. En Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 3–14, San Fransisco, Octubre de 1996.

[RH2002] Red Hat. (2002). Hibernate. (Red Hat Americas;Red Hat EMEA;Red Hat APAC) Recuperado el 15 de febrero de 2008, de Hibernate: http://www.hibernate.org/343.html.

[BK2005] BAUER CHRISTIAN y KING GAVIN. "*Hibernate in Action*". A guide to the concepts and practices of object/relational mapping, 2005.

[PESSHCR2004-2006] Mark Pollack, Rick Evans, Aleksandar Seovic, Federico Spinazzi, Rob Harrop, Griffin Caprio, Choy Rim, The Spring Java Team. Spirng.net Reference Documentation. 2004-2006.

Anexos

Anexo 1. Red

Se propone la implementación de una red WAN corporativa utilizando una Red Privada Virtual (VPN) que aumente el nivel de seguridad de los datos. La implementación de dicha red VPN se propone a nivel de hardware con el objetivo de aliviar el procesamiento (demoras) que se incurren en soluciones a nivel de software.

Se propone la utilización de una topología de la red tipo estrella. La interacción entre los sistemas será utilizando la familia de protocolos TCP/IP (ver figura 1).

Se propone la implementación de una red de tecnología inalámbrica para el funcionamiento de las oficinas (ver figura 2).

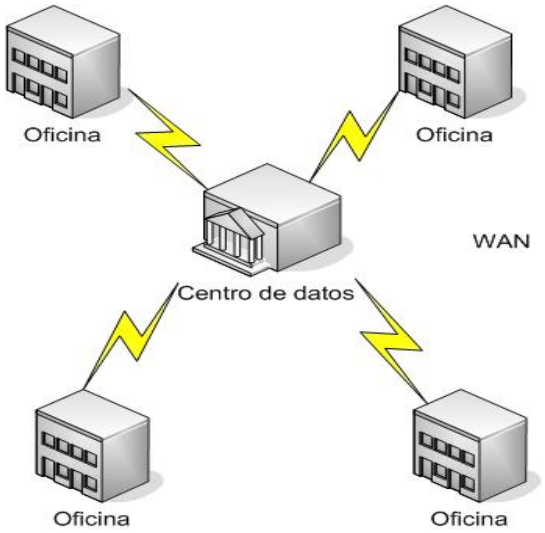


Figura 1 Arquitectura de red del sistema. Topología tipo estrella.

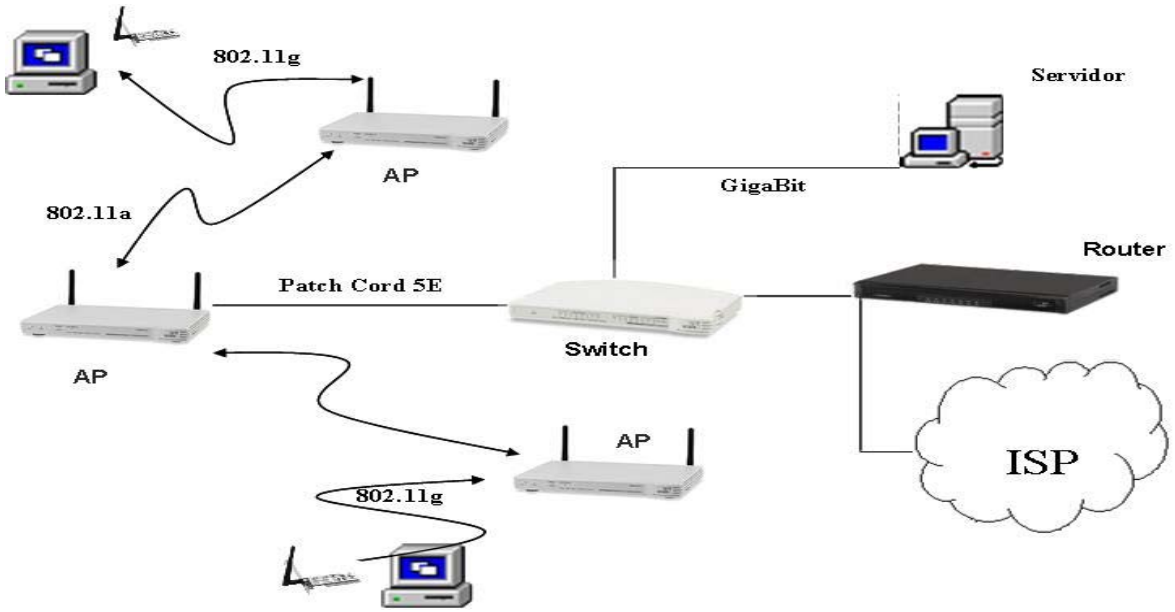


Figura 2 Arquitectura de red de las UE. Tecnología inalámbrica.

Anexo 2. Arquitectura respecto de los servidores de datos

La Arquitectura del sistema se ha diseñado basada en un modelo descentralizado (**ver figura 3**), lo que implica una comunicación intensa. De esta manera se requiere mantener una conexión a la red WAN de mediana velocidad que garantice la comunicación efectiva desde cada una de las UE hacia el centro de datos.

A pesar de esto existen momentos y pasos en los que las consultas se realizarán totalmente sobre entidades locales definidas para tal efecto, permitiendo de esta forma el normal funcionamiento de las UE en estado de desconexión con el centro de datos.

Para esto se han definido los siguientes elementos:

- Se mantendrá un cluster de Oracle a nivel central con capacidad de soportar todos los sistemas definidos.
- Existirán servidores locales con capacidad necesaria para el procesamiento de las solicitudes del conjunto de aplicaciones de las diferentes oficinas.
- Las aplicaciones siempre solicitarán los datos a través del servidor local.
- Desde cada servidor local se establecerá la conexión con servidores centrales para mantener la actualización de los datos en ambos sentidos.

Los servidores a nivel central poseerán Oracle Enterprise Edición 10g R2 Real Aplication Cluster, y en los servidores locales de las UE se poseerá el Oracle Standard Edition ONE versión 10g R2

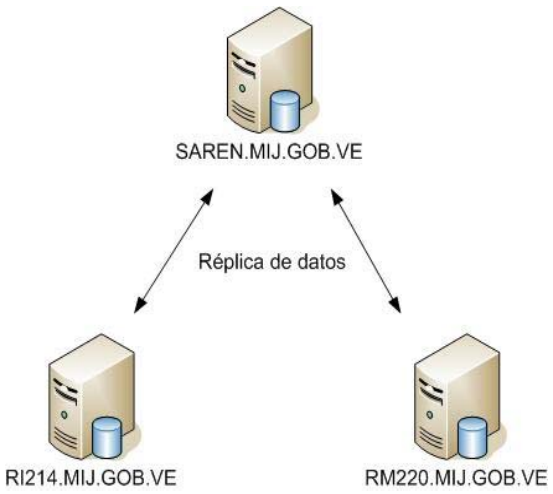


Figura 3 Modelo descentralizado.

Existen dos premisas para la comunicación entre las oficinas y el servidor central:

- Las oficinas deben poder realizar un elevado por ciento de sus funciones aunque no exista conexión con el servidor central.
- Minimizar el tiempo de atención al ciudadano evitando colas innecesarias y retrasos por el sistema.

- Las aplicaciones siempre encuestarán por defecto al servidor local. En caso de que no encuentren la información solicitada entonces la búsqueda tendrá lugar en el centro de datos.

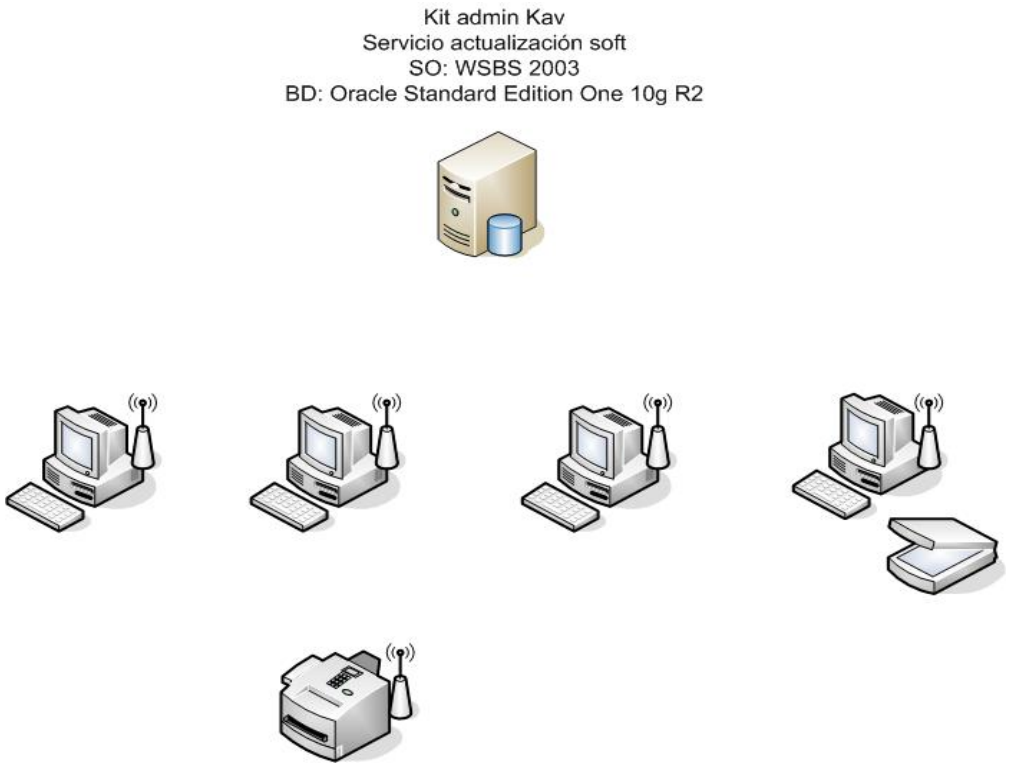


Figura 4 Arquitectura física de las oficinas.

Anexo 3. Despliegue

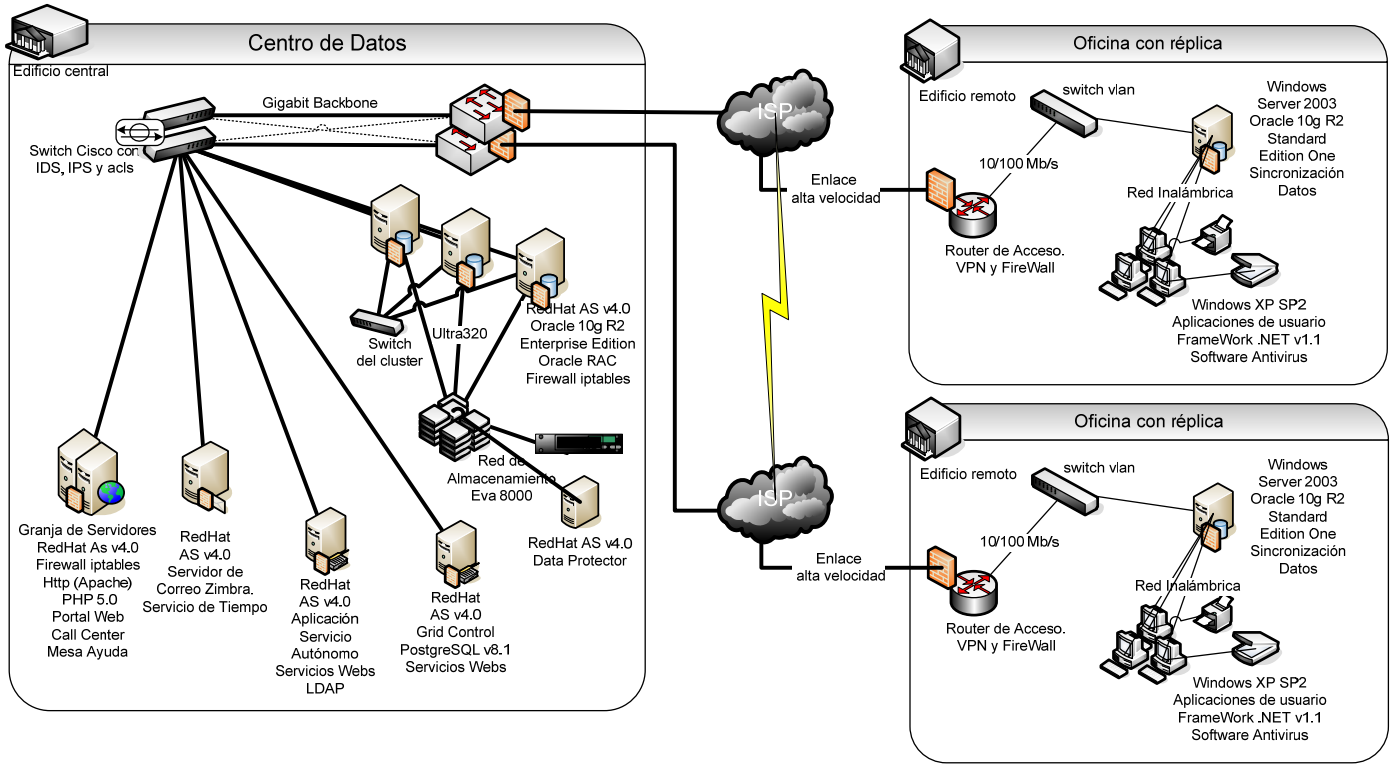


Figura 5 Despliegue de la solución de software para automatizar los Registros y Notarias de la República Bolivariana de Venezuela.

Glosario de Términos

Objetos falsos: Son objetos que simulan todos los posibles resultados contenidos en sus funcionalidades. La utilidad de estos objetos radica en que agilizan el proceso de implementación pues no es necesario esperar por cierta funcionalidad si de ante mano ya se le tiene representada en todas las vertientes posibles.

Atómico: El término se usa para definir la cualidad de un componente determinado de ser suficiente, indivisible.

Encapsular: Al empaquetamiento de las variables de un objeto con la protección de sus métodos se le llama encapsulamiento. Típicamente, el encapsulamiento es utilizado para esconder detalles de la puesta en práctica no importantes de otros objetos. Entonces, los detalles de la puesta en práctica pueden cambiar en cualquier tiempo sin afectar otras partes del programa. El término encapsular se refiere a la realización del encapsulamiento.

Abstracción/Enajenación: Se refiere al énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?", es decir representa el nivel de desconocimiento que tenga una funcionalidad de su implementación. El común denominador en la evolución de los lenguajes de programación, desde los clásicos o imperativos hasta los orientados a objetos, ha sido el nivel de abstracción del que cada uno de ellos hace uso.

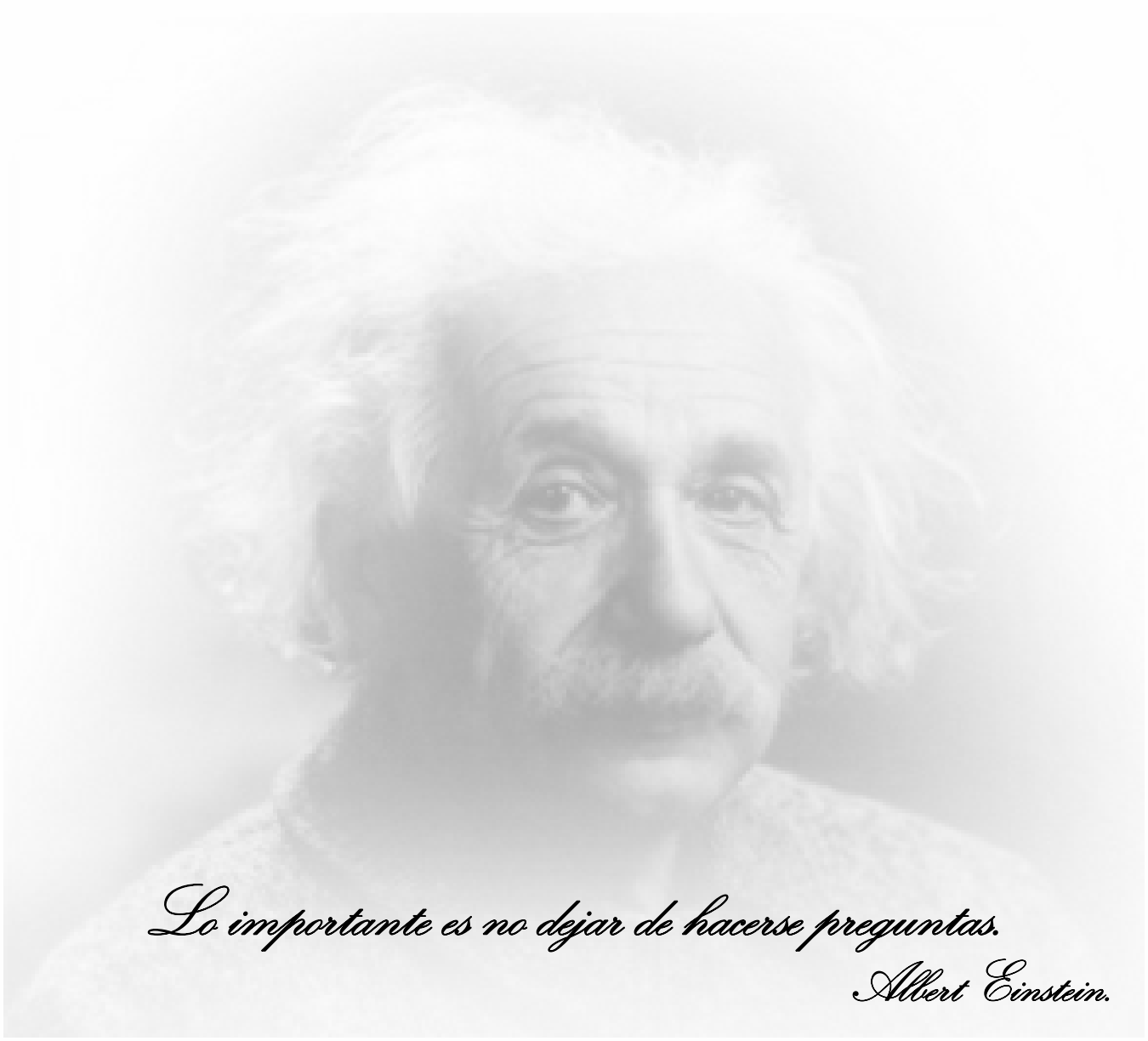
UNIVERSIDAD DE LAS CIENCIAS INFORMATICAS
FACULTAD 3

Arquitectura del sistema de Administración Financiera para SAREN.

TRABAJO DE DIPLOMA PARA OPTAR POR EL TÍTULO DE INGENIERO EN CIENCIAS INFORMÁTICAS.

Autor: Yunier Pérez Barroso.
Tutor: Ing. Yosvany Marquez Ruiz.

Ciudad de la Habana
Mayo 2008


Lo importante es no dejar de hacerse preguntas.
Albert Einstein.

DECLARACIÓN DE AUTORÍA

Declaro que soy el único autor de este trabajo y autorizo a la Facultad 3 de la Universidad de las Ciencias Informáticas a hacer uso del mismo en su beneficio.

Para que así conste firmo la presente a los ___ días del mes _____ del año _____.

_____ _____

Yunier Pérez Barroso Ing. Yosvany Márquez Ruiz

AGRADECIMIENTOS

A la Revolución y a la Universidad de las Ciencias Informáticas por hacerme partícipe de esta gran idea y permitirme aportar mi pequeño grano de arena a la misma.
A mi mamá y mi papá por su apoyo en todo momento, todo lo que soy se lo debo a ellos, a su ejemplo.
A mi hermano por su preocupación, por ayudarme a disminuir la carga cuando más me hace falta.
A tati por estar a mi lado siempre, por sopórtame tantas y tantas horas de cansancio.
A Maikel y Alejandro: sin ustedes hermanos la carga hubiera sido más pesada. De todo corazón gracias.
A Eddy por ayudarme tanto con tu conocimiento, con tus eternas críticas, pero sobre todo por ser mi camarada desde siempre.
Al Yosva, por ser mí amigo por encima de todo, por apoyarme en todas mis decisiones, por criticarme y porque trabajando juntos he aprendido muchas cosas.
A Yaumy,Henrik, Isma, Lulu por aconsejarme, ayudarme a crecer y permitirme contar con ustedes cuando necesitaba ayuda.
A Zolia y Gustavo, con ustedes empezó todo esto, gracias.
A todos los que de una forma u otra me han brindado su ayuda en los momentos difíciles: el piquete de la facultad 10, el proyecto Prisiones, a los buenos colegas de R&N, en fin a todos gracias también.

Con todo mi afecto, muchas gracias.

Yunier.

DEDICATORIA

A mis familiares.

RESUMEN

Las necesidades actuales que tiene toda organización para el logro de sus objetivos, demandan la construcción de grandes y complejos sistemas de software que requieren de la combinación de diferentes tecnologías y plataformas de hardware y software para alcanzar un funcionamiento acorde con dichas necesidades. Lo anterior, exige de los profesionales dedicados al desarrollo de software poner especial atención y cuidado al diseño de la Arquitectura sobre la cual estará soportado el funcionamiento de sus sistemas.

El presente trabajo, describe la Arquitectura del Sistema de Administración Financiera ofreciendo un marco idóneo para unificar los componentes que pertenecen a la solución bajo una misma línea; a través de los diferentes artefactos que resultan más representativos, acorde a la metodología RUP (Rational Unified Process), para el proceso de desarrollo. Además contiene un análisis de los resultados obtenidos a partir de los objetivos planteados y las restricciones arquitectónicas que se pueden encontrar y hacer que colapsen o provoquen un mal funcionamiento de la solución en un momento determinado.

ÍNDICE

Introducción

Las necesidades actuales que tiene toda organización para el logro de sus objetivos, demandan la construcción de grandes y complejos sistemas de software que requieren de la combinación de diferentes tecnologías y plataformas de hardware y software para alcanzar un funcionamiento acorde con dichas necesidades. Lo anterior, exige de los profesionales dedicados al desarrollo de software poner especial atención y cuidado al diseño de la Arquitectura sobre la cual estará soportado el funcionamiento de sus sistemas.

Un Sistema de Administración Financiera comprende la automatización de un conjunto de leyes, normas y procedimientos destinados a la obtención, asignación, uso, registro y evaluación de los recursos financieros, que tiene como propósito la eficiencia de la gestión de los mismos para la satisfacción de las necesidades colectivas.

La Administración Financiera del Sector Público en la República Bolivariana de Venezuela está integrada por un conjunto de subsistemas que deben operar en forma interrelacionada, atendiendo el mandato que en ese sentido establece la Ley Orgánica de la Administración Financiera del Sector Público (LOAFSP).

En el marco del Programa de Modernización de los Registros y Notarías de la República Bolivariana de Venezuela se decidió sustituir, en el Ministerio del Poder Popular para las Relaciones Interiores y de Justicia (MPPRIJ), el modelo financiero vigente para adoptar el que está reflejado en la LOAFSP; lo anterior incluye: establecer los niveles intermedios de administración de los fondos y una dirección descentralizada que fungen como unidades de ordenación de compromisos y pagos. Como parte de este proceso, se decidió la implementación de un software que controlará los procesos de Administración Financiera para el Servicio Autónomo de Registros y Notarias (SAREN), que es un ente descentralizado adscripto al MPPRIJ encargado de la dirección de los Registros y Notarias.

El sistema incluye el control y adecuado uso de los recursos financieros, así como el manejo eficiente y protección de los activos de SAREN cumpliendo y respetando lo establecido en la LOAFSP. Este es un elemento distintivo del nuevo modelo y fue el motivo fundamental que propició la no instauración de las aplicaciones que existían hasta el momento.

Por tanto, teniendo en cuenta que la institución SAREN no presenta en la actualidad una estructura financiera adecuada, los sistemas automatizados que están vigentes no se corresponden con las leyes, con la dinámica actual de la institución y que existen problemas de malversación, descontrol sobre todo de las finanzas y los recursos de SAREN, lo que ocasiona pérdidas de ingresos y desmoralización en la institución como ente del gobierno, resulta evidente que la construcción del software que resuelva toda esta realidad representa una salida necesaria a todos estos problemas.

Partiendo de esto, la situación problemática gira en torno a la construcción de esta solución de software y radica en *la inexistencia de una línea que defina la infraestructura que debe tener el Sistema de Administración Financiera en SAREN, que satisfaga las cualidades que se desean desarrollar en el mismo y centre el proceso de desarrollo en torno a ellas*. Esta situación lógicamente es el principal foco de atención de esta investigación de la cual se deriva el siguiente problema científico:

¿Qué elementos debe definir la Línea Base de la Arquitectura del Sistema de Administración Financiera en SAREN para que sea flexible, escalable, reutilizable y mantenible?

El objeto de estudio lo constituye el proceso de desarrollo del software de Administración Financiera que establece las fronteras del campo de acción el cual se centra en la arquitectura de software de este sistema.

Para dar solución a la problemática, el estudio realizado comprende en primer lugar la consulta de bibliografía relacionada con la Administración Financiera con el objetivo de obtener un mayor grado de familiarización con sus términos, lo anterior incluye la LOAFSP y la documentación concerniente a las características de SAREN principalmente la que tiene que ver con su estructura financiera. En segundo lugar se examinaron una serie de temas referentes a la arquitectura de software, dígase patrones, estilos, frameworks, lenguajes de descripción arquitectónica, herramientas así como buenas prácticas que están vigentes en la actualidad. Además se hizo un análisis de las características que debe tener un software para garantizar los objetivos que se plantean en la investigación, que comprende las bases para asegurar las funcionalidades y los temas referentes al desarrollo de las mismas, así como los factores afines con la integración entre los subsistemas que componen la solución. Todos estos puntos avalados en todo

momento por los resultados de la investigación y estudio en profundidad de la Arquitectura de Software y del rol de Arquitecto como máximo exponente de la misma.

Otras tareas importantes fueron las pruebas de los diversos escenarios[1] arquitectónicos obtenidos de los análisis previos, para identificar ventajas y desventajas con el fin de asegurar las características que se consideran más importantes y de mayor prioridad. El resultado de este proceso fue la obtención de una serie de modelos[2] arquitectónicos que finalmente se sometieron a un proceso de evaluación para obtener el mejor candidato.

La investigación se sustenta sobre la hipótesis de que si se define una adecuada línea base para la Arquitectura de un software, partiendo de sus requisitos funcionales y centrada en sus requerimientos arquitectónicos (requisitos no funcionales); entonces se obtendrá un sistema que sea flexible, escalable, reutilizable y mantenible.

El objetivo general que se persigue es definir la línea que deben seguir los procesos de análisis, diseño e implementación, evaluando las mejores prácticas y estilos vigentes en la actualidad referentes a la Arquitectura de software que permita a los desarrolladores y demás involucrados tener una idea clara de lo que se está implementando.

Para dar cumplimiento al objetivo general se han definido los siguientes objetivos específicos:

- Describir la terminología y los conceptos generales que están relacionados con la arquitectura de este sistema.
- Definir las políticas de integración que incluye la Línea Base de la Arquitectura para el sistema.
- Definir componentes, funcionalidades que validen y apuntalen la arquitectura propuesta así como las relaciones que se establecen entre ellos.
- Describir los patrones y frameworks utilizados en el sistema así como su implantación en el mismo.

[1] El nombre se aplica a la descripción de situaciones futuras de alguna magnitud.

[2] Propuesta, normalmente de carácter teórico-práctico, que tiene una serie de características que se consideran dignas de emular. Generalmente, el modelo ilustra una situación deseable para ser analizada y puesta en práctica en un contexto educativo similar, o bien adaptarla a otras características del entorno.

- Describir los elementos que conforman las 4+1 vistas de la arquitectura del sistema.
- Validar la arquitectura obtenida.

Para el cumplimiento de los objetivos descritos anteriormente la investigación paso por las siguientes etapas:

1. Estudio del estado del arte de los procesos relacionados con la Administración Financiera.
2. Estudio de los temas relacionados con la arquitectura de software, que incluye patrones, frameworks, estilos arquitectónicos así como métodos de evaluación arquitectónica.
3. Estudio de los métodos y políticas de integración con otros sistemas.
4. Selección de los frameworks y patrones a utilizar resultado del paso 2.
5. Etapa de pruebas a los modelos arquitectónicos obtenidos.
6. Establecimiento de las pautas de programación y seguridad para un desarrollo limpio, ordenado, sin agujeros ni vulnerabilidades.

La realización de esta investigación puede servir de guía y modelo a otros sistemas similares pues comprende un análisis de los requerimientos que debe cumplir un software de este tipo e incorpora elementos que pueden resultar útiles y prácticos para dar solución a determinados problemas que se dan comúnmente durante el desarrollo.

La Arquitectura obtenida, a pesar de la estructura que presenta, tiene en cuenta posibles limitaciones y agravantes que puede presentar cualquier equipo de desarrollo. En este sentido se logra una combinación de sus elementos de manera tal que cumple los objetivos y asegura los puntos más importantes que se tienen en cuenta durante el desarrollo de todo software, como son la abstracción, la flexibilidad, la escalabilidad, la mantenibilidad y el soporte.

Este trabajo está dividido en 3 capítulos fundamentales.

Capitulo 1: Fundamentación Teórica.

Definición del marco teórico de la investigación que comprende el estudio del arte de la Arquitectura de Software y del rol de arquitecto.

Capitulo 2: Línea Base.

Abarca la descripción del documento de Línea Base de la Arquitectura del Sistema de Administración Financiera que incluye la solución a las interrogantes que puedan surgir al respecto y otros puntos que complementan el trabajo para ganar en claridad y comprensión.

Capitulo 3: Resultados de la Arquitectura Propuesta.

Comprende el análisis de los resultados obtenidos con la Arquitectura propuesta, es decir una valoración de la misma a partir de los objetivos planteados.

Capítulo 1

1. Fundamentación Teórica.

1.1 Introducción.

La Arquitectura de software es un tema sumamente delicado, teniendo en cuenta que define la línea que deben seguir la mayoría de los elementos que intervienen en el desarrollo del software que se esté implementando. El siguiente capítulo aborda una serie de puntos referentes a los conceptos y temas relacionados con la Arquitectura a partir de una profunda investigación y experiencias anteriores que comprende un análisis del estado del arte y otros trabajos o tendencias que se siguen actualmente en el mundo. También se abordan terminologías relacionadas con la Administración Financiera del Sector Público fundamentalmente vinculadas a la Recaudación, al Presupuesto, la Contabilidad etc.

1.2 Arquitectura de software

La Arquitectura de Software es un área de investigación y práctica dentro de la Ingeniería de Software. En particular, la arquitectura de sistemas grandes ha sido objeto de un interés creciente durante la pasada década.

Esta materia no nació en forma espontánea, a partir de 1990 el término Arquitectura de Software comenzó a ganar aceptación y fue elemento de especial atención por parte de la industria y la comunidad científica. Este campo fue creado por necesidad, la realidad marcaba que los sistemas de software estaban creciendo, sistemas de cientos de miles de líneas, o incluso de millones de líneas se estaban volviendo comunes.

Existen tres razones por las cuales la Arquitectura de Software es importante para sistemas de software grandes y complejos:

- Es un vehículo de comunicación entre los stakeholders *(interesados)*. La Arquitectura de Software es una representación abstracta del sistema, a la que la mayoría de los stakeholders, sino todos, pueden utilizar como base para crear entendimiento mutuo, formar consensos y comunicarse entre ellos.
- Es una expresión de las decisiones tempranas del diseño. La Arquitectura de Software de un sistema es el artefacto que permite en forma temprana establecer prioridades entre los diferentes aspectos a ser analizados y es el artefacto con más influencia en la calidad del sistema. Los aspectos de calidad tales como desempeño, seguridad, mantenimiento, costo del esfuerzo del desarrollo actual y costo del esfuerzo del desarrollo futuro, están todos presentes en la Arquitectura.
- Es una abstracción del sistema reusable y transferible. La Arquitectura de Software constituye un modelo relativamente pequeño e intelectualmente comprensible de cómo el sistema está estructurado, y como sus componentes trabajan juntos. Este modelo es transferible entre sistemas. En particular, puede ser utilizado en otros sistemas con requerimientos similares, y puede promover reutilización a gran escala y en una línea de productos de software.

Se ha comentado sobre el surgimiento e importancia de la Arquitectura de Software, ahora resta definirla:

La Arquitectura del Software es la organización fundamental de un sistema formada por sus componentes, las relaciones entre ellos y el contexto en el que se implantarán, y los principios que orientan su diseño y evolución.

[IEEE Std 1471-2000]

Hay algunas implicancias claves en esta definición:

- La Arquitectura es una abstracción de un sistema o sistemas. Esta representa sistemas en términos de componentes abstractos que tienen propiedades externamente visibles y relaciones.
- Como la Arquitectura es abstracta, esta elimina la información local, los detalles de componentes privados no son arquitectónicos.
- Los sistemas están compuestos por muchas estructuras (comúnmente llamadas *vistas*). Una vista por si sola puede representar nada más que una arquitectura trivial. Es más, una arquitectura debería ser descrita por un conjunto de vistas que soporten las necesidades de su análisis y comunicación.

1.2.1 Objetivos

- Permite comprender y mejorar la estructura de las aplicaciones complejas.
- Permite reutilizar dicha estructura (o partes de ella) para resolver problemas similares.
- Posibilita planificar la evolución de la aplicación, identificando las partes mutables e inmutables de la misma, así como los costes de los posibles cambios.
- Permite analizar la corrección de la aplicación y su grado de cumplimiento respecto a los requisitos iniciales.
- Permite el estudio de alguna propiedad específica del dominio.

1.2.2 Características

- La Arquitectura parte del diseño de software.
- Nivel del diseño de software donde se definen la estructura y propiedades globales del sistema.
- Incluye sus componentes, las propiedades observables de dichos componentes y las relaciones que se establecen entre ellos.
- Un aspecto crítico: Una arquitectura errónea puede llevar a problemas incontables.
- Representación de alto nivel de la estructura del sistema describiendo las partes que lo integran.
- Puede incluir los patrones que supervisan la composición de sus componentes y las restricciones al aplicar los patrones.
- Trata aspectos del diseño y desarrollo que no pueden tratarse adecuadamente dentro de los módulos que forman el sistema.

1.2.3 ¿De qué se Ocupa?

- Diseño preliminar o de alto nivel.
- Organización a alto nivel del sistema, incluyendo aspectos como la descripción y análisis de propiedades relativas a su estructura y control global, los protocolos de comunicación y sincronización utilizados, la distribución física del sistema y sus componentes, etc.

- Otros aspectos relacionados con el desarrollo del sistema, su evolución y adaptación al cambio: composición, reconfiguración, reutilización, escalabilidad, mantenibilidad, etc.

1.2.4 ¿De qué no se ocupa?

- Diseño detallado.
- Diseño de algoritmos.
- Diseño de estructuras de datos.

1.3 El rol de Arquitecto de software

El rol de arquitecto de software tiene una serie de responsabilidades dentro del trabajo del proyecto. El arquitecto es el encargado de seleccionar la arquitectura más adecuada para el sistema que responda a las necesidades del usuario, a los requisitos funcionales y no funcionales y además debe ser capaz de lograr los resultados esperados bajo las restricciones dadas.

El arquitecto debe tomar decisiones críticas y cruciales para el sistema que indican muchas veces la dirección que se va a tomar en el proyecto con respecto al mantenimiento y funcionamiento de la aplicación. Para tomar estas decisiones el arquitecto debe apoyarse en el equipo de trabajo y en las restricciones más importantes y cada uno de los cambios que se realicen dejarlos bien explicados y documentados para que las demás personas puedan entender lo que se hizo.

RUP plantea varias responsabilidades para un arquitecto de software:

- El arquitecto posee la responsabilidad técnica del sistema y debe ser capaz de desarrollar e implementar todas las funcionalidades de la aplicación económicamente.
- El arquitecto debe trabajar en conjunto con todos los implicados en el proyecto para obtener experiencia de cada uno.
- Debe ser flexible en caso de existir algún cambio en la aplicación que influya en la arquitectura.

- El arquitecto obtiene una arquitectura sólida después de haber pasado por varias iteraciones del producto, ya en la fase de elaboración se debe tener la arquitectura base y estable, porque este es precisamente el hito de esta fase de RUP.
- En caso de tener un sistema complejo, RUP aconseja que exista un equipo de arquitectos para dividir las tareas porque la arquitectura en este caso puede llegar a ser demasiado compleja.
- El arquitecto debe tener conocimiento y experiencia en el desarrollo de sistemas, porque él es quien va a explicarle la arquitectura a cada uno de los miembros del equipo de trabajo.
- El arquitecto debe tener presente además que la arquitectura está dirigida por los casos de uso.
- Para el arquitecto la línea base de la arquitectura es el documento más importante en su trabajo.
- El arquitecto es responsable de seleccionar los estándares de codificación para el desarrollo del sistema.

1.4 Componentes, conectores y relaciones

Se entiende por *componentes* los bloques de construcción que conforman las partes de un sistema de software. A nivel de lenguajes de programación, pueden ser representados como módulos, clases, objetos o un conjunto de funciones relacionadas (Buschman et al., 1996). La noción de componente puede llegar a ser muy amplia: el término puede ser utilizado para especificar un conjunto de componentes.

Se distinguen tres tipos de componentes (Perry y Wolf, 1992), denominados también *elementos,* que son:

_ *Elementos de Datos:* contienen la información que será transformada.
_ *Elementos de Proceso*: transforman los elementos de datos.
_ *Elementos de Conexión:* llamados también *conectores,* que bien pueden ser elementos de datos o de proceso, y mantienen unidas las diferentes piezas de la Arquitectura.

Una *relación* es la conexión entre los componentes (Buschman et al., 1996). Puede definirse también como una abstracción de la forma en que los componentes interactúan en el sistema a través de los elementos de conexión. Es importante distinguir que una relación se concreta mediante conectores.

Según Bass et al. (1998), en virtud de que está conformado por componentes y relaciones entre ellos, todo sistema, por muy simple que sea, tiene asociada una Arquitectura. Sin embargo, no es necesariamente cierto que esta Arquitectura sea conocida por todos los involucrados en el desarrollo del mismo. Esto hace evidente la diferencia entre la Arquitectura del sistema y su descripción. Esta particularidad propone la importancia de la representación de una arquitectura.

Kazman et al. (2001) presentan la Arquitectura de software como el resultado de decisiones tempranas de diseño, necesarias antes de la construcción del sistema. Según Bass et al. (1998), uno de los aspectos importantes de una arquitectura de software es que, por ser un artefacto de diseño, direcciona atributos de calidad asociados al sistema. Kazman et al. (2001) proponen que las arquitecturas facilitan o inhiben estos atributos. Es por ello que se propone el estudio de los atributos de calidad asociados a la Arquitectura de un sistema de software, y cuál es su impacto sobre el mismo.

1.5 Calidad Arquitectónica.

Barbacci et al. (1995) establecen que el desarrollo de formas sistemáticas para relacionar atributos de calidad de un sistema a su arquitectura provee una base para la toma de decisiones objetivas sobre acuerdos de diseño y permite a los ingenieros realizar predicciones razonablemente exactas sobre los atributos del sistema que son libres de prejuicios y asunciones no triviales. El objetivo de fondo es lograr la habilidad de evaluar cuantitativamente y llegar a acuerdos entre múltiples atributos de calidad para alcanzar un mejor sistema de forma global.

1.5.1 Atributos de Calidad.

Según Barbacci et al. (1995) la *calidad de software* se define como el grado en el cual el software posee una combinación deseada de atributos. Tales atributos son requerimientos adicionales del sistema (Kazman et al., 2001), que hacen referencia a características que éste debe satisfacer, diferentes a los requerimientos funcionales.

Estas características o atributos se conocen con el nombre de *atributos de calidad*, los cuales se definen como las propiedades de un servicio que presta el sistema a sus usuarios (Barbacci et al. 1995).

A grandes rasgos, Bass et al. (1998) establece una clasificación de los atributos de calidad en dos categorías:

- *Observables vía ejecución*: aquellos atributos que se determinan del comportamiento del sistema en tiempo de ejecución. La descripción de algunos de estos atributos se presenta en la tabla 1.
- *No observables vía ejecución*: aquellos atributos que se establecen durante el desarrollo del sistema. La descripción de algunos de estos atributos se presenta en la tabla 2.

Atributo de Calidad	**Descripción**
Disponibilidad *(Availability)*	Es la medida de disponibilidad del sistema para el uso (Barbacci et al.,1995).
Confidencialidad *(Confidentiality)*	Es la ausencia de acceso no autorizado a la información (Barbacci et al.,1995).
Funcionalidad *(Functionality)*	Habilidad del sistema para realizar el trabajo para el cual fue concebido (Kazman et al., 2001).
Desempeño *(Performance)*	Es el grado en el cual un sistema o componente cumple con sus funciones designadas, dentro de ciertas restricciones dadas, como velocidad, exactitud o uso de memoria. (IEEE 610.12). Según Smith (1993), el desempeño de un sistema se refiere a aspectos temporales del comportamiento del mismo. Se refiere a capacidad de respuesta, ya sea el tiempo requerido para responder a aspectos específicos o el número de eventos procesados en un intervalo de tiempo. Según Bass et al. (1998), se refiere además a la cantidad de comunicación e interacción existente entre los componentes del sistema.

Confiabilidad *(Reliability)*	Es la medida de la habilidad de un sistema a mantenerse operativo a lo largo del tiempo (Barbacci et al., 1995).
Seguridad externa *(Safety)*	Ausencia de consecuencias catastróficas en el ambiente. Es la medida de ausencia de errores que generan pérdidas de información (Barbacci et al.,1995).
Seguridad Interna *(Security)*	Es la medida de la habilidad del sistema para resistir a intentos de uso no autorizados y negación del servicio, mientras se sirve a usuarios legítimos (Kazman et al., 2001).

Tabla 1. Descripción de atributos de calidad observables vía ejecución.

Es importante destacar que tener conocimiento de los atributos observables, no necesariamente implica que se satisfacen los atributos no observables vía ejecución. Por ejemplo, un sistema que satisface todos los requerimientos observables puede o no, ser costoso de desarrollar, así como también puede o no ser imposible de modificar. De igual manera, un sistema altamente modificable puede o no, arrojar resultados correctos.

Atributo de Calidad	Descripción
Configurabilidad *(Configurability)*	Posibilidad que se otorga a un usuario experto a realizar ciertos cambios al sistema (Bosch et al., 1999).
Integrabilidad *(Integrability)*	Es la medida en que trabajan correctamente componentes del sistema que fueron desarrollados separadamente al ser integrados. (Bass et al. 1998)
Integridad *(Integrity)*	Es la ausencia de alteraciones inapropiadas de la información (Barbacci et al., 1995).
Interoperabilidad *(Interoperability)*	Es la medida de la habilidad de que un grupo de partes del sistema trabajen con otro sistema. Es un tipo especial de *integrabilidad* (Bass et al. 1998)
Modificabilidad *(Modifiability)*	Es la habilidad de realizar cambios futuros al sistema. (Bosch et al. 1999).
Mantenibilidad	Es la capacidad de someter a un sistema a reparaciones y evolución (Barbacci et

(Maintainability)	al., 1995). Capacidad de modificar el sistema de manera rápida y a bajo costo (Bosch et al. 1999).
Portabilidad *(Portability)*	Es la habilidad del sistema para ser ejecutado en diferentes ambientes de computación. Estos ambientes pueden ser hardware, software o una combinación de los dos (Kazman et al., 2001).
Reusabilidad *(Reusability)*	Es la capacidad de diseñar un sistema de forma tal que su estructura o parte de sus componentes puedan ser reutilizados en futuras aplicaciones (Bass et al. 1998).
Escalabilidad *(Scalability)*	Es el grado con el que se pueden ampliar el diseño arquitectónico, de datos o procedimental (Pressman, 2002).
Capacidad de Prueba *(Testability)*	Es la medida de la facilidad con la que el software, al ser sometido a una serie de pruebas, puede demostrar sus fallas. Es la probabilidad de que, asumiendo que tiene al menos una falla, el software fallará en su próxima ejecución de prueba (Bass et al. 1998).

Tabla 2. Descripción de atributos de calidad no observables vía ejecución.

Bosch (2000) establece que los requerimientos de calidad se ven altamente influenciados por la Arquitectura del sistema. Al respecto, Bass et al. (1998) afirman que la calidad del sistema debe ser considerada en todas las fases del diseño, pero los atributos de calidad se manifiestan de maneras distintas a lo largo de estas fases. De esta forma, establecen que la arquitectura determina ciertos atributos de calidad del sistema, pero existen otros atributos que no dependen directamente de la misma.

Independientemente de esto, es importante tener en cuenta que no puede lograrse la satisfacción de ciertos atributos de calidad de manera aislada. Encontrar un atributo de calidad puede tener efectos positivos o negativos sobre otros atributos que, de alguna manera, también se desean alcanzar (Bass et al., 1998).

En su mayoría, los atributos de calidad se pueden organizar y descomponer de maneras diferentes, en lo que se conoce como *modelos de calidad*. Los modelos de calidad de software facilitan el entendimiento del proceso de la Ingeniería de Software (Pressman, 2002).

1.6 Estilos arquitectónicos y patrones.

De manera concreta, al diseñar una Arquitectura de software se deben crear y representar componentes que interactúen entre ellos y tengan asignadas tareas específicas, además de organizarlos de forma tal que se logren los requerimientos establecidos. Se puede partir con patrones de soluciones ya probados, con la intención de no comenzar de cero las propuestas y utilizar modelos que han funcionado. Estas soluciones probadas se conocen como estilos arquitectónicos, patrones arquitectónicos y patrones de diseño, que van de lo general a lo particular.

El estilo afecta a toda la Arquitectura de software y puede combinarse en la propuesta de solución. Por otra parte, un patrón arquitectónico se enfoca en dar solución a un problema en específico, de un atributo de calidad, y abarca solo parte de la Arquitectura. Un patrón de diseño ayuda a diseñar la estructura interna de un componente específico, es decir, su detalle. Aunque estos estilos y patrones se pueden adoptar, también pueden adaptarse con objeto de lograr alguna funcionalidad concreta esperada.

Bosch (2000) establecen que la imposición de ciertos estilos arquitectónicos mejora o disminuye las posibilidades de satisfacción de ciertos atributos de calidad del sistema. Con esto afirman que cada estilo propicia atributos de calidad, y la decisión de implementar alguno de los existentes depende de los requerimientos de calidad del sistema. De manera similar, plantean el uso de los patrones arquitectónicos y los patrones de diseño para mejorar la calidad del sistema. Al respecto, Buschmann et al. (1996) afirman que un criterio importante del éxito de los patrones – tanto arquitectónicos como de diseño - es la forma en que estos alcanzan de manera satisfactoria los objetivos de la Ingeniería de Software. Los patrones soportan el desarrollo, mantenimiento y evolución de sistemas complejos y de gran escala.

1.6.1 Estilos Arquitectónicos.

En la caracterización de David Garlan, Andrew Kompanek, Ralph Melton y Robert Monroe [GKM+96], también de Carnegie Mellon, se define el estilo como una entidad consistente en cuatro elementos:

1. Un vocabulario de elementos de diseño: componentes y conectores tales como tuberías, filtros, clientes, servidores, parsers, bases de datos, etcétera.
2. Reglas de diseño o restricciones que determinan las composiciones permitidas de esos elementos.
3. Una interpretación semántica que proporciona significados precisos a las composiciones.
4. Análisis susceptibles de practicarse sobre los sistemas construidos en un estilo, por ejemplo análisis de disponibilidad para estilos basados en procesamiento en tiempo real, o detección de abrazos mortales para modelos cliente-servidor.

Los estilos son entidades que ocurren en un nivel sumamente abstracto, puramente arquitectónico, que no coincide ni con la fase de análisis propuesta por la temprana metodología de modelado orientada a objetos (aunque sí un poco con la de diseño), ni con lo que más tarde se definirían como paradigmas de arquitectura, ni con los patrones arquitectónicos. A continuación se analizan estas tres discordancias una por una:

El análisis de la metodología de objetos, tal como se enuncia en [RBP+91] está muy cerca del requerimiento y la percepción de un usuario o cliente técnicamente agnóstico, un protagonista que en el terreno de los estilos no juega ningún papel. En arquitectura de software, los estilos surgen de la experiencia que el Arquitecto posee; de ningún modo vienen impuestos de manera explícita en lo que el cliente le pide.

Los paradigmas como la Arquitectura Orientada a Objetos (p.ej. CORBA), a componentes (COM, JavaBeans, EJB, CORBA Component Model) o a servicios, tal como se los define en [WF04], se relacionan con tecnologías particulares de implementación, un elemento de juicio que en el campo de los estilos se trata a un nivel más genérico y distante, si es que se llega a tratar alguna vez. Dependiendo de la clasificación que se trate, estos paradigmas tipifican más bien como sub-estilos de estilos más englobantes (peer-to-peer, distribuidos, etc) o encarnan la forma de implementación de otros estilos cualesquiera.

Los patrones arquitectónicos, por su parte, se han materializado con referencia a lenguajes y paradigmas también específicos de desarrollo, mientras que ningún estilo presupone o establece preceptivas al respecto. Si hay algún código en las inmediaciones de un estilo, será código del lenguaje de descripción arquitectónica o del lenguaje de modelado; de ninguna manera será código de lenguaje de programación. Lo mismo en cuanto a las representaciones visuales: los estilos se describen mediante simples cajas y líneas, mientras que los patrones suelen representarse en UML [Lar03].

¿Cuántos y cuáles son los estilos, entonces?

La tabla 3 resume los principales estilos arquitectónicos, los atributos de calidad que propician y los atributos que se ven afectados negativamente (atributos en conflicto), de acuerdo a Bass et al. (1998).

Estilo	Descripción	Atributos asociados	Atributos en conflicto
Datos Centralizados	Sistemas en los cuales cierto número de clientes accede y actualiza datos compartidos de un repositorio de manera frecuente.	Integrabilidad Escalabilidad Modificabilidad	Desempeño
Flujo de Datos	El sistema es visto como una serie de transformaciones sobre piezas sucesivas de datos de entrada. El dato ingresa en el sistema, y fluye entre los componentes, de uno en uno, hasta que se le asigne un destino final (salida o repositorio).	Reusabilidad Modificabilidad Mantenibilidad	Desempeño
Máquinas Virtuales	Simulan alguna funcionalidad que no es nativa al hardware o software sobre el que está implementado.	Portabilidad	Desempeño
Llamada y Retorno	El sistema se constituye de un programa principal que tiene el control del sistema y varios subprogramas que se comunican con éste mediante el uso de llamadas.	Modificabilidad Escalabilidad Desempeño	Mantenibilidad Desempeño

Componentes Independientes	Consiste en un número de procesos u objetos independientes que se comunican a través de mensajes.	Modificabilidad Escalabilidad	Desempeño Integrabilidad

Tabla 3. Estilos Arquitectónicos y Atributos de Calidad

1.6.2 Patrones

Buschmann et al. (1996) define *patrón* como una regla que consta de tres partes, la cual expresa una relación entre un contexto, un problema y una solución. En líneas generales, un patrón sigue el siguiente esquema:

- *Contexto.* Es una situación de diseño en la que aparece un problema de diseño
- *Problema*. Es un conjunto de fuerzas que aparecen repetidamente en el contexto
- *Solución*. Es una configuración que equilibra estas fuerzas. Esta abarca:

    - Estructura con componentes y relaciones.
    - Comportamiento a tiempo de ejecución: aspectos dinámicos de la solución, como la colaboración entre componentes, la comunicación entre ellos, etc.

Ventajas:

- Son soluciones simples y técnicas.
- Muy prácticos (deslumbran) y útiles.

Desventajas:

- Son soluciones concretas a problemas concretos.
- Libro de recetas, lo que hace difícil averiguar el patrón adecuado ante un problema concreto. (LePus)
- No dejan huella: En una implementación es difícil saber qué patrón se utilizó. (Ingeniería inversa).
- Facilitan la reutilización del diseño pero no tanto la de la implementación.

- No están formalizados (al menos no de forma simple).

Patrones de Arquitectura.

Partiendo de esta definición, propone los *patrones arquitectónicos* como descripción de un problema particular y recurrente de diseño, que aparece en contextos de diseño específico, y presenta un esquema genérico demostrado con éxito para su solución. El esquema de solución se especifica mediante la descripción de los componentes que la constituyen, sus responsabilidades y desarrollos, así como también la forma como estos colaboran entre sí.

Así mismo, Buschmann et al. (1996) plantean que los patrones arquitectónicos expresan el esquema de organización estructural fundamental para sistemas de software. Provee un conjunto de subsistemas predefinidos, especifica sus responsabilidades e incluye reglas y pautas para la organización de las relaciones entre ellos. Propone que son plantillas para Arquitecturas de software concretas, que especifican las propiedades estructurales de una aplicación - con amplitud de todo el sistema - y tienen un impacto en la Arquitectura de subsistemas. La selección de un patrón arquitectónico es, por lo tanto, una decisión fundamental de diseño en el desarrollo de un sistema de software.

La colección de patrones arquitectónicos debe ser estudiada en términos de factores de calidad e intereses, en anticipación a su uso. Esto quiere decir que un patrón puede ser analizado previamente, con la intención de seleccionar el que mejor se adapte a los requerimientos de calidad que debe cumplir el sistema. De manera similar, Barbacci et al. (1997) proponen que debe estudiarse la composición de los patrones, dado que ésta puede dificultar aspectos como el análisis, o poner en conflicto otros atributos de calidad. La tabla 4 presenta algunos patrones arquitectónicos, además de los atributos que propician y los atributos en conflicto, de acuerdo a Buschmann et al. (1996).

Patrón Arquitectónico	Descripción	Atributos asociados	Atributos en conflicto
Layers	Consiste en estructurar aplicaciones que pueden ser descompuestas en grupos de	Reusabilidad Portabilidad	Desempeño Mantenibilidad

	subtareas, las cuales se clasifican de acuerdo a un nivel particular de abstracción.	Facilidad de Prueba	
Pipes and Filters	Provee una estructura para los sistemas que procesan un flujo de datos. Cada paso de procesamiento está encapsulado en un componente filtro (*filter*). El dato pasa a través de conexiones (*pipes*), entre filtros adyacentes.	Reusabilidad Mantenibilidad	Desempeño
Blackboard	Aplica para problemas cuya solución utiliza estrategias no determinísticas. Varios subsistemas ensamblan su conocimiento para construir una posible solución parcial ó aproximada.	Modificabilidad Mantenibilidad Reusabilidad Integridad	Desempeño Facilidad de Prueba
Broker	Puede ser usado para estructurar sistemas de software distribuido con componentes desacoplados que interactúan por invocaciones a servicios remotos. Un componente *broker* es responsable de coordinar la comunicación, como el reenvío de solicitudes, así como también la transmisión de resultados y excepciones.	Modificabilidad Portabilidad Reusabilidad Escalabilidad Interoperabilidad	Desempeño
Model-View-Controler	Divide una aplicación interactiva en tres componentes. El modelo *(model)* contiene la información central y los datos. Las vistas *(view)* despliegan información al usuario. Los controladores *(controlers)* capturan la entrada del usuario. Las vistas y los controladores constituyen la interfaz del usuario.	Funcionalidad Mantenibilidad	Desempeño Portabilidad
Presentation-Abstraction-	Define una estructura para sistemas de software interactivos de agentes de	Modificabilidad Escalabilidad	Desempeño Mantenibilidad

Control	cooperación organizados de forma jerárquica. Cada agente es responsable de un aspecto específico de la funcionalidad de la aplicación y consiste de tres componentes: presentación, abstracción y control.	Integrabilidad	
Microkernel	Aplica para sistemas de software que deben estar en capacidad de adaptar los requerimientos de cambio del sistema. Separa un núcleo funcional mínimo del resto de la funcionalidad y de partes específicas pertenecientes al cliente.	Portabilidad Escalabilidad Confiablidad Disponibilidad	Desempeño
Reflection	Provee un mecanismo para sistemas cuya estructura y comportamiento cambia dinámicamente. Soporta la modificación de aspectos fundamentales como estructuras tipo y mecanismos de llamadas a funciones.	Modificabilidad	Desempeño

Tabla 4 Patrones arquitectónicos y atributos de calidad.

Con la intención de hacer una comparación clara entre estilo arquitectónico y patrón arquitectónico, la tabla 5 presenta las diferencias entre estos conceptos, construida a partir del planteamiento de Buschmann et al. (1996).

Estilo Arquitectónico	Patrón Arquitectónico
Sólo describe el esqueleto estructural y general *para aplicaciones*	Existen en varios rangos de escala, comenzando con patrones que definen la estructura básica de *una* aplicación
Son independientes del contexto al que puedan ser aplicados	Partiendo de la definición de *patrón*, requieren de la especificación de un contexto del problema

Cada estilo es independiente de los otros	Depende de patrones más pequeños que contiene, patrones con los que interactúa, o de patrones que lo contengan
Expresan técnicas de diseño desde una perspectiva que es independiente de la situación actual de diseño	Expresa un problema recurrente de diseño muy específico, y presenta una solución para él, desde el punto de vista del contexto en el que se presenta
Son una categorización de sistemas	Son soluciones generales a problemas comunes

Tabla 5. Diferencias entre estilo arquitectónico y patrón arquitectónico.

Patrones de Diseño:

Un *patrón de diseño* provee un esquema para refinar los subsistemas o componentes de un sistema de software, o las relaciones entre ellos. Describe la estructura comúnmente recurrente de los componentes en comunicación, que resuelve un problema general de diseño en un contexto particular (Buschman et al., 1996).

Son menores en escala que los patrones arquitectónicos, y tienden a ser independientes de los lenguajes y paradigmas de programación. Su aplicación no tiene efectos en la estructura fundamental del sistema, pero sí sobre la de un subsistema (Buschman et al., 1996), debido a que especifica a un mayor nivel de detalle, sin llegar a la implementación, el comportamiento de los componentes del subsistema. La tabla 6 presenta algunos patrones de diseño, junto a los atributos de calidad que propician y los atributos que entran en conflicto con la aplicación del patrón, según Buschmann et al. (1996).

Patrón de Diseño	Descripción	Atributos asociados	Atributos en conflicto
Whole-Part	Ayuda a constituir una colección de objetos	Reusabilidad	Desempeño

	que juntos conforman una unidad semántica.	Modificabilidad	
Master-Slave	Un componente maestro *(master)* distribuye el trabajo a los componentes esclavos *(slaves)*. El componente maestro calcula un resultado final a partir de los resultados arrojados por los componentes esclavos.	Escalabilidad Desempeño	Portabilidad
Proxy	Los clientes asociados a un componente se comunican con un representante de éste, en lugar del componente en sí mismo.	Desempeño Reusabilidad	Desempeño
Command Procesor	Separa las solicitudes de un servicio de su ejecución. Maneja las solicitudes como objetos separados, programa sus ejecuciones y provee servicios adicionales como el almacenamiento de los objetos solicitados, para permitir que el usuario pueda deshacer alguna solicitud.	Funcionalidad Modificabilidad Facilidad de Prueba	Desempeño
View Handler	Ayuda a manejar todas las vistas que provee un sistema de software. Permite a los clientes abrir, manipular y eliminar vistas. También coordina dependencias entre vistas y organiza su actualización.	Escalabilidad Modificabilidad	Desempeño
Forwarder-Receiver	Provee una comunicación transparente entre procesos de un sistema de software con un modelo de interacción punto a punto *(peer to peer).*	Mantenibilidad Modificabilidad Desempeño	Configurabilidad
Client-Dispatcher-Server	Introduce una capa intermedia entre clientes y servidores, es el componente despachador *(dispatcher)*. Provee una ubicación transparente por medio de un nombre de servicio, y esconde los detalles del	Configurabilidad Portabilidad Escalabilidad Disponibilidad	Desempeño Modificabilidad

	establecimiento de una conexión de comunicación entre clientes y servidores.		
Publisher-Subscriber	Ayuda a mantener sincronizados los componentes en cooperación. Para ello, habilita una vía de propagación de cambios: un editor (*publisher*) notifica a los suscriptores (*suscribers*) sobre los cambios en su estado.	Escalabilidad	Desempeño

Tabla 6. Patrones de diseño y atributos de calidad.

Puntualizar que la mayoría de estos patrones de diseño tiene su base en el catálogo de patrones contenido en el libro “Design Patterns: Elements of Reusable Object-Oriented Software”, también conocido como el LIBRO GOF (Gang-Of-Four Book) [G95]. El cual es considerado uno de los libros más vendidos del mundo de la programación orientación a objetos y recomendado para todos aquellos que se quieren dedicar en serio a la misma. Con la publicación de este libro, los patrones de software adquirieron una gran relevancia.

La figura (ver figura 1) presenta la relación de abstracción existente entre los conceptos de estilo arquitectónico, patrón arquitectónico y patrón de diseño. En ella se representa el planteamiento de Buschmann et al. (1996), que propone el desarrollo de arquitecturas de software como un sistema de patrones, y distintos niveles de abstracción.

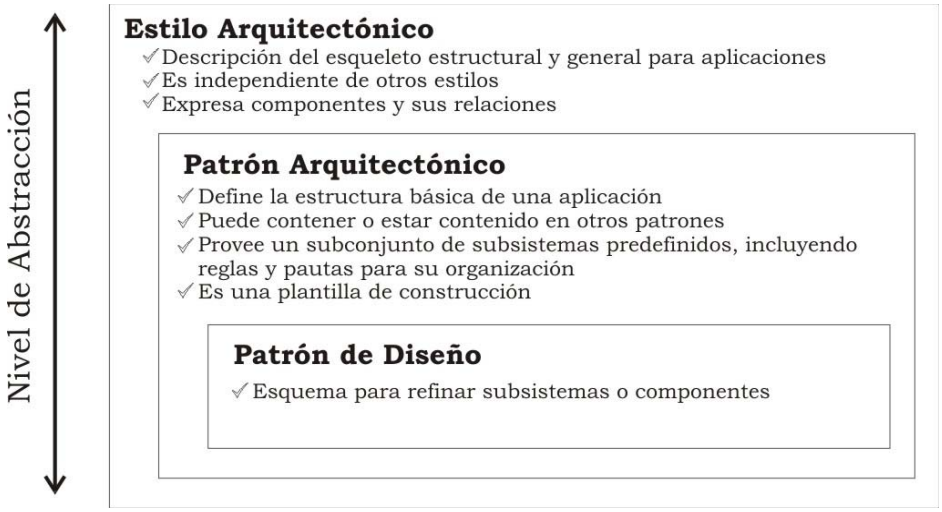


Figura 1. Relación de abstracción entre estilos y patrones

Los estilos y patrones ayudan al Arquitecto a definir la composición y el comportamiento del sistema de software, y una combinación adecuada de ellos permite alcanzar los requerimientos de calidad.

Ahora bien, la organización del sistema de software debe estar disponible para todos los involucrados en el desarrollo del sistema, ya que establece un mecanismo de comunicación entre los mismos. Tal objetivo se logra mediante la representación de la Arquitectura en formas distintas, obteniendo así una descripción de la misma de forma tal que puede ser entendida y analizada por todos los involucrados, con miras a obtener un sistema de calidad. Estas descripciones pueden establecerse a través de las *vistas arquitectónicas*, las *notaciones* como UML y los *lenguajes de descripción arquitectónica* (Bengtsson, 1999).

1.7 Vistas Arquitectónicas

De acuerdo al nivel de responsabilidad dentro del desarrollo de un sistema y la relación que se establezca con el mismo, son muchas las partes involucradas e interesadas en la Arquitectura de software (Kruchten, 1999), a saber:

- El *analista del sistema*, quien la utiliza para organizar y expresar claramente los requerimientos y entender las restricciones de tecnología y los riesgos.
- *Usuarios finales y clientes*, que necesitan conocer el sistema que están adquiriendo.
- El *gerente de proyecto*, que la utiliza para organizar el equipo y planificar el desarrollo.
- Los *diseñadores*, que lo utilizan para entender los principios subyacentes y localizar los límites de su propio diseño.
- Otras *organizaciones desarrolladoras* (si el sistema es abierto), que la utilizan para entender cómo interactuar con el sistema.
- Las *compañías subcontratadas*, que la utilizan para entender los límites de su sección de desarrollo.
- Los A*rquitectos*, quienes velan por la evolución del sistema y la reutilización de componentes.

Todas estas personas deben comunicarse de manera efectiva para discutir y razonar acerca de la Arquitectura, y así alcanzar las metas del desarrollo. En virtud de esto, Kruchten (1999) plantea que debe tenerse una representación del sistema que todos puedan comprender.

Una única representación de la Arquitectura del sistema resultaría demasiado compleja y poco útil para todos los involucrados, pues contendría mucha información irrelevante para la mayoría de estos. Es por ello que se plantea la necesidad de representaciones que contengan únicamente elementos que resultan de importancia para cierto grupo de involucrados.

Buschmann et al. (1996) establece que una *vista arquitectónica* representa un aspecto parcial de una arquitectura de software, que muestra propiedades específicas del sistema. Bass et al. (1998), haciendo uso indistinto de los términos estructura y vista, proponen que las *estructuras arquitectónicas* pueden definirse agrupando los componentes y conectores de acuerdo a la funcionalidad del sistema, sincronización y comunicación de procesos, distribución física, propiedades estáticas, propiedades dinámicas y propiedades de ejecución, entre otras.

Por su parte, Kruchten (1999) define una *vista arquitectónica* como una descripción simplificada o abstracción de un sistema desde una perspectiva específica, que cubre intereses particulares y omite

entidades no relevantes a esta perspectiva. Para la definición de una vista, Kruchten propone la identificación de ciertos elementos, que se mencionan a continuación:

- Punto de vista: involucrados e intereses de los mismos.
- Elementos que serán capturados y representados en la vista y las relaciones entre estos.
- Principios para organizar la vista.
- Forma en que se relacionan los elementos de una vista con otras vistas.
- Proceso a ser utilizado para la creación de la vista.

Kazman et al. (2001), Bass et al. (1998), Hofmeister et al. (2000) y Kruchten (1999), proponen, en función de las características del sistema o del proceso de desarrollo del mismo, distintas vistas arquitectónicas. Es importante resaltar que las vistas propuestas no son independientes entre sí, puesto que son perspectivas distintas de un mismo sistema (Kruchten, 1999). Por tal motivo, las vistas arquitectónicas deben estar coordinadas, de manera tal que al realizar cambios, estos se vean correctamente reflejados en las vistas afectadas, garantizando consistencia entre las mismas.

1.8 Notaciones

1.8.1 Unified Modeling Language (UML)

UML ha conseguido un rol importante en el proceso de desarrollo de software en la actualidad (Booch et al., 1999). La unificación del método de diseño y las notaciones, ha ampliado, entre otras cosas, el mercado de herramientas CASE que soportan el proceso de diseño de Arquitecturas de software. UML ofrece soporte para clases, clases abstractas, relaciones, comportamiento por interacción, empaquetamiento, entre otros. Estos elementos se pueden representar mediante nueve tipos de diagramas, que son: de clases, de objetos, de casos de uso, de secuencia, de colaboración, de estados, de actividades, de componentes y de despliegue.

Bengtsson (1999) presenta las características generales de UML, y las razones por las que resulta interesante su aplicación para efectos de la representación de una Arquitectura de software. Establece

que en UML existe soporte para algunos de los conceptos asociados a las arquitecturas de software, como los componentes, los paquetes, las librerías y la colaboración. UML permite la descripción de componentes en la Arquitectura de software en dos niveles; se puede especificar sólo el nombre del componente o especificar las clases o interfaces que implementan estos.

De igual forma, UML provee una notación para la descripción de la proyección de los componentes de software en el hardware. Esto corresponde a la vista física del modelo 4+1 (Kruchten, 1999). La proyección de los componentes de software permite a los Ingenieros de Software hacer mejores estimaciones cuando se intenta medir la calidad del sistema implementado, lo cual contribuye a la búsqueda de la mejor solución para un sistema específico. Esta notación puede ser extendida con mayor nivel de detalle y los componentes pueden ser conectados entre sí con el uso de las bondades del lenguaje UML.

Los patrones y frameworks están también soportados por UML, mediante el uso combinado de paquetes, componentes y colaboraciones, entre otros. Booch et al. (1999) proponen de forma detallada todos los aspectos que hacen de UML un lenguaje conveniente para la representación de las arquitecturas de software.

Sin embargo pese a que este lenguaje, por todas las facilidades que se han mencionado, puede ser utilizado para modelar arquitecturas, este no abarca la totalidad de los procesos y conceptos relacionados con esta disciplina por lo que resulta insuficiente y poco práctico aplicarlos en algunas de ellas. En este sentido, para satisfacer completamente el modelado de arquitecturas de software existen Lenguajes de Descripción de Arquitecturas (ADL).

1.8.2 Lenguajes de Descripción de Arquitectura (ADLs)

La literatura referida a los ADL es ya de magnitud respetable. Para desarrollar esta sección se han tomado en consideración los estudios previos de Kogut y Clements [KC94, KC95, Vestal [Ves93], Luckham y Vera [LV95], Shaw, DeLine y otros [SDK+95], Shaw y Garlan [SG94, SG95] y Medvidovic [Med96], así como la documentación de cada uno de los lenguajes. Algunos de los ADLs que se revisarán en este estudio son ACME, Darwin, Jacal y UniCon.

ACME.

El proyecto Acme comenzó a principios de 1995 en la Escuela de Ciencias de la Computación de la Universidad Carnegie Mellon. Hoy este proyecto se organiza en dos grandes grupos, que son el lenguaje Acme propiamente dicho y el Acme Tool Developer's Library (AcmeLib). De Acme se deriva, en gran parte, el ulterior estándar emergente ADML. Fundamental en el desarrollo de Acme ha sido el trabajado de destacados arquitectos y sistematizadores del campo, entre los cuales el más conocido es sin duda David Garlan, uno de los teóricos de arquitectura de software más activos en la década de 1990.

Objetivo principal: La motivación fundamental de Acme es el intercambio entre arquitecturas e integración de ADLs. Garlan considera que Acme es un lenguaje de descripción arquitectónica de segunda generación; podría decirse que es de segundo orden: un metalenguaje, una lengua franca para el entendimiento de dos o más ADLs, incluido Acme mismo. Con el tiempo, sin embargo, la dimensión metalingüística de Acme fue perdiendo prioridad y los desarrollos actuales profundizan su capacidad intrínseca como ADL puro.

Darwin.

Darwin es un lenguaje de descripción arquitectónica desarrollado por Jeff Magee y Jeff Kramer [MEDK95, MK96]. Darwin describe un tipo de componente mediante una interfaz consistente en una colección de servicios que son provistos (declarados por ese componente) o requeridos (o sea, que se espera ocurran en el entorno). Las configuraciones se desarrollan instanciando las declaraciones de componentes y estableciendo vínculos entre ambas clases de servicios.

Darwin soporta la descripción de arquitecturas que se reconfiguran dinámicamente a través de dos construcciones: instanciación tardía (lazy) y construcciones dinámicas explícitas. Utilizando instanciación lazy, se describe una configuración y se instancian componentes sólo en la medida en que los servicios que ellos provean sean utilizados por otros componentes. La estructura dinámica explícita, en cambio, se realiza mediante constructos de configuración imperativos. De este modo, la declaración de configuración deviene un programa que se ejecuta en tiempo de ejecución, antes que una declaración estática de la estructura.

Cada servicio de Darwin se modeliza como un nombre de canal, y cada declaración de binding es un proceso que trasmite el nombre del canal al componente que requiere el servicio. En una implementación generada en Darwin, se presupone que cada componente primitivo está implementado en algún lenguaje de programación, y que para cada tipo de servicio se necesita un ligamento (glue) que depende de cada plataforma. El algoritmo de elaboración actúa, esencialmente, como un servidor de nombre que proporciona la ubicación de los servicios provistos a cualquier componente que se ejecute.

Objetivo principal: Como su nombre lo indica, Darwin está orientado más que nada al diseño de arquitecturas dinámicas y cambiantes.

Jacal.

Es un lenguaje de descripción de arquitecturas de software de propósito general creado en la Universidad de Buenos Aires, por un grupo de investigación del Departamento de Computación de la Facultad de Ciencias Exactas y Naturales.

Objetivo principal – El objetivo principal de Jacal es lo que actualmente se denomina “animación” de arquitecturas. Esto es, poder visualizar una simulación de cómo se comportaría en la práctica un sistema basado en la arquitectura que se ha representado.

Más allá de este objetivo principal, el diseño de Jacal contempla otras características deseables en un ADL, como por ejemplo contar con una representación gráfica que permita a simple vista transmitir la arquitectura del sistema, sin necesidad de recurrir a información adicional. Para este fin, se cuenta con un conjunto predefinido (extensible) de conectores, cada uno con una representación distinta.

UniCon.

UniCon (Universal Connector Support) es un ADL desarrollado por Mary Shaw y otros [SDK+95]. Proporciona una herramienta de diseño para construir configuraciones ejecutables basadas en tipos de componentes, implementaciones y “conexiones expertas” que soportan tipos particulares de conectores. UniCon se asemeja a Darwin en la medida en que proporciona herramientas para desarrollar

configuraciones ejecutables de caja negra y posee un número fijo de tipos de interacción, pero el modelo de conectores de ambos ADLs es distinto.

Oficialmente se define como un ADL cuyo foco apunta a soportar la variedad de partes y estilos que se encuentra en la vida real y en la construcción de sistemas a partir de sus descripciones arquitectónicas. UniCon es el ADL propio del proyecto Vitruvius, cuyo objetivo es elucidar un nivel de abstracción de modo tal que se pueda capturar, organizar y tornar disponible la experiencia colectiva exitosa de los arquitectos de software.

Objetivo principal: El propósito de UniCon es generar código ejecutable a partir de una descripción, a partir de componentes primitivos adecuados. UniCon se destaca por su capacidad de manejo de métodos de análisis de tiempo real a través de RMA (Rate Monotonic Analysis).

A pesar de las potencialidades de los ADLs para representar arquitecturas de una manera más completa. El presente trabajo opta por UML puesto que es el lenguaje establecido por el proyecto general, abarca la mayoría de la representación arquitectónica que el sistema necesita y además porque UML es el lenguaje del que mayor conocimiento existe por parte del equipo de arquitectura.

1.9 Modelos y arquitecturas de referencia

Los modelos particularizan un estilo imponiendo una serie de restricciones sobre el mismo y realizando una descomposición y definición estándar de componentes.

- OSI (*Open System Interconection*) que particulariza el estilo de organización en capas, con 7 niveles.
- RM-ODP para el diseño de sistema distribuidos y abiertos (ISO/ITU-T, 1995). Hace transparente la heterogeneidad de plataformas, sistemas operativos, lenguajes. Se basa en el estilo de componentes independientes.
- CCM, COM, JavaBeans - Sistemas basados en el intercambio de eventos.

1.9.1 Arquitectura JEE

En la Arquitectura JEE se contemplan cuatro capas, en función del tipo de servicio y contenedores:

- Capa de **cliente**, también conocida como capa de presentación o de aplicación. Nos encontramos con componentes Java (applets o aplicaciones) y no-Java (HTML, JavaScript, etc.).
- Capa **Web**. Intermediario entre el cliente y otras capas. Sus componentes principales son los servlets y las JSP. Aunque componentes de capa cliente (applets o aplicaciones) pueden acceder directamente a la capa EJB, lo normal es que Los servlets/JSPs pueden llamar a los EJB.
- Capa **Enterprise JavaBeans**. Permite a múltiples aplicaciones tener acceso de forma concurrente a datos y lógica de negocio. Los EJB se encuentran en un servidor EJB, que no es más que un **servidor de objetos distribuidos**. Un EJB puede conectarse a cualquier capa, aunque su misión esencial es conectarse con los sistemas de información empresarial
  (un gestor de base de datos, ERP, etc.)
- Capa de **sistemas de información empresarial**.

La visión de la Arquitectura es un **esquema lógico**, no físico. Cuando hablamos de capas nos referimos sobre todo a **servicios** diferentes (que pueden estar físicamente dentro de la misma máquina e incluso compartir servidor de aplicaciones y JVM).

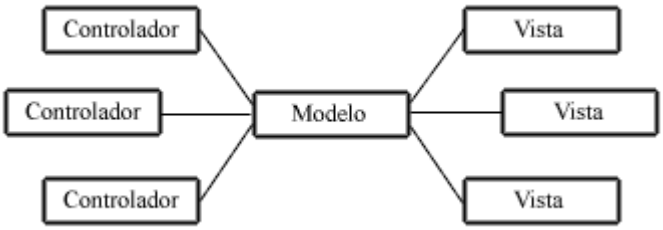
1.9.2 Arquitectura MVC

La Arquitectura MVC (*Model/View/Controller*) fue introducida como parte de la versión Smalltalk-80[3]. Fue diseñada para reducir el esfuerzo de programación necesario en la implementación de sistemas múltiples y sincronizados de los mismos datos. Sus características principales son que el Modelo, las Vistas y los Controladores se tratan como entidades separadas; esto hace que cualquier cambio producido en el Modelo se refleje automáticamente en cada una de las Vistas.

[3] Del lenguaje de programación Smalltalk.

Este modelo de Arquitectura se puede emplear en sistemas de representación gráfica de datos, o en sistemas CAD[4], en donde se presentan partes del diseño con diferente escala de aumento, en ventanas separadas.

En la siguiente figura, vemos la Arquitectura MVC en su forma más general. Hay un Modelo, múltiples Controladores que manipulan ese Modelo, y hay varias Vistas de los datos del Modelo, que cambian cuando cambia el estado de ese Modelo.



Este modelo de arquitectura presenta varias ventajas:

- Hay una clara separación entre los componentes de un programa; lo cual nos permite implementarlos por separado
- Hay un API[5] muy bien definido; cualquiera que use el API, podrá reemplazar el Modelo, la Vista o el Controlador, sin aparente dificultad.
- La conexión entre el Modelo y sus Vistas es dinámica; se produce en tiempo de ejecución, no en tiempo de compilación.

Al incorporar el modelo de arquitectura MVC a un diseño, las piezas de un programa se pueden construir por separado y luego unirlas en tiempo de ejecución. Si uno de los Componentes, posteriormente, se observa que funciona mal, puede reemplazarse sin que las otras piezas se vean afectadas. Este escenario contrasta con la aproximación monolítica típica de muchos programas Java. Todos tienen un

[4] Del inglés **C**omputer **A**ided **D**esign. CAD es todo sistema informático destinado a asistir al diseñador en su tarea específica.

[5] Interfaz de programación de aplicaciones (Applications Programming Interface): una serie de funciones que están disponibles para realizar programas para un cierto entorno.

Frame que contiene todos los elementos, un controlador de eventos, un montón de cálculos y la presentación del resultado. Ante esta perspectiva, hacer un cambio aquí no es nada trivial.

1.10 Frameworks/ Marcos de trabajo

La conferencia de la Universidad de Málaga "Arquitectura, marcos de trabajo y patrones", Canal (2005) establece una serie de características y clasificaciones para los frameworks:

- Definen una arquitectura adaptada a las particularidades de un determinado dominio de aplicación, definiendo de forma abstracta una serie de componentes y sus interfaces y estableciendo las reglas y mecanismos de interacción entre ellos.
- Suele incluirse la implementación de algunos de los componentes e incluso varias implementaciones alternativas.
- El usuario

    - Selecciona, instancia, extiende y reutiliza los componentes del marco.
    - Completa la Arquitectura con nuevos componentes específicos dentro de las relaciones estructurales del marco.

- Básicamente se presentan como un diseño reutilizable de todo o parte de un sistema, representado por un conjunto de componentes abstractos y la forma en que los componentes interactúan.
- Una alternativa es verlos como un esqueleto de una aplicación que debe ser adaptado por el programador según sus necesidades concretas.
- Un marco de trabajo define el patrón arquitectónico que relaciona los componentes de un sistema.
- Presentan dos niveles:

    - Especificación de la Arquitectura marco.
    - Implementación del marco de trabajo (normalmente un lenguaje orientado a objetos).

- Al desarrollo de aplicaciones a partir de un marco de trabajo se le denomina extensión del marco:

- Puntos de entrada (hot spots): Componentes o procedimientos cuya implementación final depende de la aplicación concreta.
- Definidos por el diseñador del marco para que sean la forma natural de la extensión del mismo.

Dependiendo de la extensión tenemos:

Marcos de trabajo de caja blanca

Que se extienden mediante herencia, concretamente mediante la implementación de las clases y métodos abstractos definidos como puntos de entrada. Se tiene acceso al código del marco y se permite reutilizar la funcionalidad de sus clases mediante herencia y redefinición de métodos.

Marcos de trabajo de caja de cristal

Admiten la inspección de su estructura e implementación, pero no su modificación ni extensión, excepto en los puntos de entrada.

Marcos de trabajo de caja gris

Los puntos de entrada no son simplemente métodos abstractos, de los que se declara meramente su signatura, sino que se definen por medio de un lenguaje de especificación de alto nivel los requisitos que deben cumplirse a la hora de implementarse.

Marcos de trabajo de caja negra

Su instanciación se realiza por medio de composición y delegación, en lugar de utilizar la herencia. Los puntos de entrada se definen por medio de interfaces que deben implementar los componentes que extiendan el marco.

Dependiendo de su aplicabilidad tenemos:

Marcos de trabajo horizontal

Válidos para todos los dominios de aplicación concretados en un aspecto del sistema.
Infraestructuras de comunicación, interfaces de usuario, entornos visuales, etc.

Marcos de trabajo distribuido (Middelware Application Frameworks) o Plataforma de Componentes Distribuidos para integrar componentes distribuidos.

Aíslan las dificultades conceptuales y técnicas del desarrollo de aplicaciones distribuidas basadas en componentes (CORBA, Active X/OLE/COM, JavaBeans, ACE, Hector, Aglets).

Marcos de trabajo vertical

Desarrollados específicamente para un dominio de aplicación.
Telecomunicaciones, fabricación, servicios telemáticos y multimedia, etc.

Ventajas de la utilización de marcos de trabajo:

- Son composicionales.
- Son el grado más alto de reutilización dentro del desarrollo de software.
- El diseño arquitectónico se reutiliza.
- Reducción de costes/Mejora de la calidad.
- Están intrínsecamente unidos a los componentes ya que además de proporcionar funcionalidad de los componentes permiten la composición entre ellos de forma consistente.

1.10.1 Spring.net

Es un framework/marco de trabajo que ayuda a construir aplicaciones empresariales sobre la plataforma .NET, basado en el framework Spring de Java del cual hereda los principales conceptos, tales como Inyección de Dependencia y Programación Orientada a Aspectos (AOP por sus siglas en ingles).

La mayoría de las aplicaciones empresariales están compuestas por una variedad de capas físicas y por un conjunto de funcionalidades. Independientemente de la Arquitectura empleada y atendiendo a la gran variedad de objetos internos y externos, que se manejan en cada nivel, la utilización de spring.net resulta una buena decisión.

Ventajas

Precisamente, al utilizar Spring.net se aseguran una serie de aspectos muy importantes en las aplicaciones, aspectos que comprende el framework y no requieren casi ningún código por parte del equipo de desarrollo.

- Implementación de Patrones de Diseño: La utilización de patrones en una arquitectura resulta una decisión muy acertada y eficiente puesto que constituyen buenas prácticas y soluciones a problemas muy comunes en el desarrollo de cualquier aplicación. Spring.net implementa una gran variedad de estos patrones de diseño de una forma bastante sencilla y práctica. Por ejemplo tenemos *Factory, Abstract Factory, Builder, Decorator, and Service Locator* por solo mencionar algunos.
- Inversión del Control (IoC)[6]: Delega la responsabilidad de instanciar los objetos en un archivo de configuración XML, esto resulta muy útil puesto que evita cualquier dependencia que pueda existir entre los objetos, más si son de capas o aplicaciones externas. El nombre viene precisamente de que los objetos generalmente son instanciados antes de ser utilizados en el momento en que se carga el XML.
- Basado en Programación Orientada a Aspectos (AOP): Ayuda a modificar dinámicamente el modelo estático para incluir el código requerido para cumplir los requerimientos secundarios sin tener que modificar el modelo estático original. Mejor aún, normalmente se puede tener este código adicional en una única localización en vez de tenerlo repartido por el modelo existente, como se haría si estuviéramos trabajando Orientado a Objetos (OO).

[6] Cuando hablamos acerca de Inversión de control: *"La pregunta es, ¿Qué aspecto del control se invierte?"*. Después de platicar sobre el término Inversión de control se sugiere renombrar el nombre del patrón, o al menos darle un nombre más explicito, y se comenzó a usar el término *inyección de dependencia*.

Desventajas

- La configuración de Spring está inflada. Configurar mediante XML es muy laborioso e introduce errores al no tener una herramienta para hacerlo.
- Perdida de las ventajas del tipado fuerte al configurar con XML.
- Spring no es ligero. Es decir tiende a aumenta el tiempo de respuesta de la aplicación lo que repercute en el rendimiento de la misma. Esto se debe a que en el momento de la llamada el framework interpreta todas las instancias que tiene configuradas en el fichero.
- 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.

1.10.2 NHibernate

Utilizar un framework/ marco de trabajo de Object Relational Mapping (ORM) para resolver la lógica de persistencia es una técnica madura que ha demostrado ser extremadamente superior a las técnicas tradicionales basadas en el uso de APIs como ADO.NET.

Esta sección pretende argumentar esta idea a partir el uso de NHibernate, un framework de ORM open-source para .NET basado en el excelente framework Hibernate surgido en la comunidad Java en el año 2002.

Ventajas.

Una estrategia elegante y compatible con las buenas prácticas de diseño pregonadas en estos últimos 10 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, en este caso NHibernate ofrece entre otras las siguientes ventajas:

- Transparente: Los objetos del dominio desconocen todo lo concerniente a 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 puede cargar jerarquías de objetos en forma polimórfica.
- Soporte de los 3 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: Permite cargar objetos que solo contengan la clave del objeto completo.
- Soporte de caching: En el contexto de una transacción, puedo disminuir la cantidad de veces que voy 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, etc. simplemente cambiando la configuración correspondiente.

1.11 Evaluando una Arquitectura de Software

Un vez generada y documentada la Arquitectura de software, ésta debe evaluarse para verificar que cumpla con todos los requerimientos; específicamente con los atributos de calidad establecidos. Dicha evaluación puede realizarse mediante técnicas cualitativas, como cuestionarios o escenarios, o a través de técnicas cuantitativas, como simulaciones o modelos matemáticos. En la literatura existen diferentes métodos de evaluación para verificar desde múltiples atributos de calidad hasta algunos en específico.

Ejemplos de estos métodos de evaluación son ATAM, ABAS, SAAM, SNA, ALMA, RMA, teoría de colas, teoría de confiabilidad, entre otras.

Cuanto más temprano se encuentre un problema en un proyecto de software, mejor. El costo de arreglar un error durante las fases de requerimientos o diseño, es mucho menor al costo de arreglar ese mismo error en la fase de verificación. Dado que la Arquitectura es un producto temprano de la fase de diseño, esta tiene un profundo efecto en el sistema y en el proyecto.

Una mala Arquitectura puede llevar a un proyecto al fracaso. Todos los requerimientos de calidad pueden quedar insatisfechos.

La Arquitectura también determina la estructura del proyecto: configuración, agenda y presupuesto, alcance, entre otros aspectos. Es mejor cambiar la Arquitectura antes que otros artefactos, que están basados en ella, se estabilicen.

Realizar una evaluación de la arquitectura es la manera más económica de evitar desastres.

1.11.1 ¿Cuándo una Arquitectura puede ser evaluada?

Generalmente, la evaluación de la Arquitectura ocurre después que esta ha sido especificada, pero antes que empiece la implementación. En un proceso iterativo y/o incremental, la evaluación se puede realizar al final de cada ciclo. Sin embargo, uno de los atractivos de la evaluación de arquitecturas es que se puede efectuar en cualquier etapa de la vida de una arquitectura. En particular, existen dos variaciones útiles: temprana y tardía.

1.11.2 Evaluación temprana

La evaluación no tiene porque esperar a que la Arquitectura este totalmente especificada. Esta puede ser utilizada en cualquier etapa del proceso de creación de la Arquitectura, para examinar las decisiones arquitectónicas ya tomadas y decidir entre las opciones que están pendientes.

Por supuesto, la completitud y fidelidad de la evaluación es directamente proporcional a la completitud y fidelidad de la descripción de la Arquitectura.

1.11.3 Evaluación tardía

Esta variación toma lugar no solo cuando la Arquitectura está terminada, también cuando la implementación esta completa. Este caso ocurre cuando la organización hereda un sistema legado. La técnica para evaluar un sistema legado es la misma que para evaluar un sistema recién nacido. Una evaluación es útil para entender el sistema legado, y saber si este cumple con los requerimientos de calidad y comportamiento.

En general, una evaluación debe realizarse cuando hay suficiente de la Arquitectura como para justificarlo. Una buena regla sería: *realizar una evaluación cuando el equipo de desarrollo empieza a tomar decisiones que dependen de la Arquitectura y el costo de deshacerlas sobrepasa al costo de realizar una evaluación*.

1.11.4 ¿Qué resultado produce la evaluación de una Arquitectura?

En términos concretos, la evaluación de la Arquitectura produce un informe, la forma y contenido del mismo varía según el método utilizado. En particular, produce repuestas a dos tipos de preguntas:

- ¿Es esta Arquitectura adecuada para el sistema para la cual fue diseñada?
- ¿Cuál de dos o más Arquitecturas propuestas es la más adecuada para el sistema?

Decimos que una Arquitectura es adecuada cuando cumple dos criterios:

- El sistema resultante de ella cumple con los objetivos de calidad. No todas las propiedades de calidad del sistema son resultado directo de la Arquitectura, pero muchas lo son.
- El sistema puede ser construido con los recursos con que se cuenta: el plantel, el presupuesto, el sistema legado (si hay), entre otros. Esto es, la Arquitectura es construible.

Un resultado que también produce la evaluación de una Arquitectura es la captura y priorización de las metas que la arquitectura debe cumplir para poder ser considerada adecuada.

La evaluación de una Arquitectura no produce resultados cuantitativos. No es de interés, por ejemplo, evaluar el performance en cantidad de transacciones por segundo a esta altura, dado que el sistema no está construido aún. Lo que interesa, en un espíritu de mitigación de riesgos, es aprender cómo un atributo de calidad es afectado por una decisión de diseño arquitectónico, para que de esta forma se pueda estudiar con cuidado dicha decisión.

Una evaluación Arquitectónica dice si una arquitectura es adecuada respecto a un conjunto de metas, y problemática con respecto a otro conjunto de metas. En ocasiones, las metas pueden ser contradictorias entre ellas, o algunas ser más importantes que otras. El director de proyecto es quien deberá tomar la decisión si la Arquitectura evalúa bien o mal en las distintas áreas. La evaluación ayuda a encontrar debilidades, no dirá “si” o “no”, “bien” o “mal”, “6 en 10”, dirá donde están los riesgos.

1.12 La Administración Financiera

La Administración Financiera es la técnica que tiene por objeto la obtención, control y el adecuado uso de recursos financieros que requiere una empresa, así como el manejo eficiente y protección de los activos de la empresa.

Es la disciplina que se encarga del estudio de la teoría y de su aplicación en el tiempo y en el espacio, sobre la obtención de recursos, su asignación, distribución y minimización del riesgo en las organizaciones a efectos de lograr los objetivos que satisfagan a la coalición imperante.

1.12.1 Objetivos.

- Planear el crecimiento de la empresa, tanto táctica como estratégica.
- Captar los recursos necesarios para que la empresa opere en forma eficiente. Asignar recursos de acuerdo con los planes y necesidades de la empresa.

- Optimizar los recursos financieros.
- Minimizar la incertidumbre de la inversión. Maximización de las utilidades Maximización del Patrimonio Neto Maximización del Valor Actual Neto de la Empresa Maximización de la Creación de Valor.

1.12.2 Funciones

La mayor parte de las decisiones empresariales se miden en términos financieros. La importancia de la función administrativa financiera depende del tamaño de la empresa, en compañías pequeñas, la función financiera la desempeña el departamento de contabilidad. Al crecer una empresa es necesario un departamento separado ligado al presidente de la compañía (o director general) por medio de un vicepresidente de finanzas, conocido como gerente financiero. El tesorero y el contralor se reportan al vicepresidente de finanzas. El tesorero coordina las actividades financieras, tales como: planeación financiera y percepción de fondos, administración del efectivo, desembolsos de capital, manejo de créditos y administración de la cartera de inversiones. El contador se ocupa de actividades contables, administración fiscal, procesamiento de datos así como la contabilidad financiera y de costos. La función administrativa financiera está muy ligada con la economía y la contaduría.

1.13 Propuesta de Solución: Arquitectura Software Administración Financiera

Para darle solución a la situación problemática planteada, se define la Línea Base de la Arquitectura para establecer la infraestructura que soporta los procesos lógicos que se llevan a cabo durante el desarrollo del Sistema de Administración Financiera de SAREN en la República Bolivariana de Venezuela. Específicamente esta propuesta abarca diez módulos principales que responden al negocio y dos independientes que introduce la arquitectura para el desarrollo de componentes, servicios y la integración.

La arquitectura obtenida está dirigida a alcanzar un sistema flexible, escalable, mantenible. Para lograr estos objetivos se apoya en una serie de recursos. A continuación se relacionan algunos de ellos:

- Implementación del Patrón Fachada.
- Implementación del Patrón Proxy.

- Implementación del Patrón MVC.
- Implementación del Patrón Reflection.
- Utilización de un estilo y patrón multicapa.
- Utilización del framework Spring.net.
- Utilización del framework NHibernate.

La incorporación de todos estos elementos es muy particular de la solución, con su utilización se busca respetar las prioridades fundamentales del cliente y la Arquitectura. En este sentido se tienen en cuenta los requisitos funcionales, los no funcionales y las disponibilidades del proyecto. Los aportes más significativos son:

- Utilizar spring.net de una forma parcial, centrado fundamentalmente en su filosofía de inyección de dependencias para lograr una mayor abstracción entre las capas.
- Agregar funcionalidades y métodos a NHibernate, relacionados fundamentalmente con el trabajo con procedimientos almacenados, que resulten prácticos para el desarrollo gracias a que es open-source.
- Hacer uso de las interfaces como recurso para mantener la menor dependencia posible y facilitar posteriores mantenimientos al software.

1.14 Conclusiones

Una vez descritos los procesos que se analizan dentro del marco de este trabajo, se pude concluir que la arquitectura de software es muy importante para garantizar un flujo de trabajo adecuado y preciso. En efecto, durante su definición se tienen en cuenta una serie de aspectos que contemplan el aseguramiento de los requisitos funcionales y no funcionales del software; en este caso vinculado a los procesos relacionados con la Administración Financiera del sector público en SAREN.

El diseño de una Arquitectura de software debe considerarse una parte fundamental, crítica e imprescindible en el desarrollo de un sistema de software, ya que es precisamente en esta fase en donde recae toda la creatividad, experiencia y creación de la propuesta de solución que más se adecue a las necesidades del cliente y le permita lograr sus objetivos.

Se trata de un concepto que nació hace ya varios años, no obstante, emerge recientemente como concepto formal, como un proceso de Ingeniería. En general, la mayoría no tiene un método bien definido para desarrollar la industria de software y aunque no es una tarea sencilla el adoptar la creación de una Arquitectura de software, se requiere romper paradigmas en la forma de trabajo de las personas. Los profesionales de la industria de software y específicamente, quienes están dedicados al diseño de sistemas, deben capacitarse ampliamente en el campo de la Arquitectura de software para cumplir con esta importante etapa del ciclo de vida de un sistema.

Capítulo 2

2. Línea Base.

2.1 Introducción.

Este capítulo tiene como propósito describir los puntos más significativos para la Arquitectura del software de Administración Financiera. Es decir, ofrecer un marco idóneo para unificar los componentes que pertenecen a la solución bajo una misma línea a través de los diferentes artefactos que resultan más representativos, acorde a la metodología RUP (Rational Unified Process), para el proceso de desarrollo. Lo anterior incluye 4 de las 5 vistas arquitectónicas donde se recogen casos de usos, paquetes, subsistemas, clases, componentes y demás elementos que avalen y apoyen la solución.

2.1 Alcance

La Arquitectura del software de Administración Financiera incluye patrones arquitectónicos, patrones de diseño y el uso de frameworks/marcos de trabajo, que proporcionan una serie de funcionalidades que representan significativos pasos de avance en el desarrollo del sistema. Asimismo se propone un desarrollo basado en componentes, conectores y subsistemas para su desarrollo y mantenimiento.

La Línea Base abarca además los temas referentes a la integración entre los módulos que componen la solución así como mecanismos, estructuras y políticas que son necesarios para la correcta ejecución de la misma. La Arquitectura se presenta desde dos enfoques: uno **Horizontal** que comprende la comunicación entre los módulos y las prioridades para su desarrollo atendiendo a las funcionalidades que se quieran ir obteniendo en cada iteración y otra **Vertical** que se centra en las especificidades que tiene la Arquitectura para cada módulo de manera individual.

Independientemente de que el software en general abarca 10 módulos que responden al negocio, este documento se limita solamente a una primera iteración que incluye 4 de ellos. No obstante en las restantes se tiene que ir refinando y actualizando el documento hasta obtener una versión final.

2.2 Concepciones Generales.

La solución de software está dirigida a la automatización de los procesos administrativos y financieros según la distribución adoptada actualmente a tal efecto en SAREN (Servicio Autónomo de Registros y Notarias) en la República Bolivariana de Venezuela (ver figura 2).

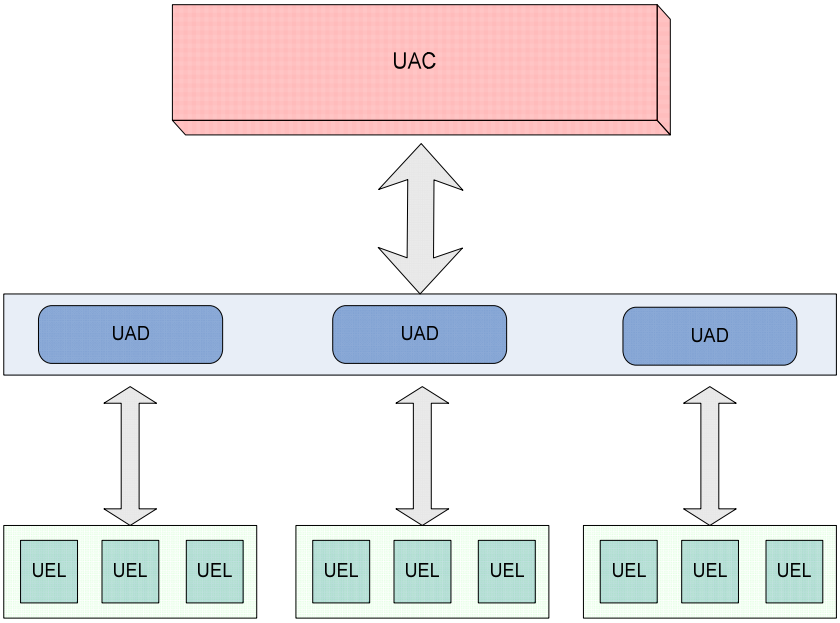


Figura 2. Estructura Financiera de SAREN.

Algunos conceptos y términos que se han utilizado derivan una serie de palabras que resultan claves para comprender el proceso y las diferentes situaciones que se puedan encontrar en este documento.

2.2.1 Definiciones, Acrónimos y Abreviaturas

UE: Unidad Ejecutora.
UAC: Unidad Administradora Central.
UAD: Unidad Administradora Desconcentrada.
UEL: Unidad Ejecutora Local.
SAREN: Servicio Autónomo de Registros y Notarías.
LOAFSP: Ley Orgánica de la Administración Financiera del Sector Público.
MPPRIJ: Ministerio del Poder Popular para las Relaciones Interiores y de Justica.

2.3 Descripción Arquitectónica

El Sistema de Administración Financiera consta de 10+2 módulos, lo anterior significa que son diez módulos que responden a los requisitos funcionales (Negocio) y dos que funcionan independientes ("MóduloComún" y "ArquitecturaBase"). El primero se utiliza para la construcción y desarrollo de los servicios y componentes que apuntalan la Arquitectura. El segundo es donde se implementan todos los componentes de negocio que involucren la totalidad de la solución y los mecanismos de integración del sistema según la Arquitectura propuesta.

El Sistema está concebido para desempeñarse como una aplicación de escritorio y desarrollado sobre la plataforma .Net. Cada uno de los subsistemas y funcionalidades que lo componen estarán disponible en la aplicación dependiendo de los requerimientos y competencia que tenga la UE donde se esté implantando. Es decir, en las UEL, UAD o UAC, la aplicación brindará y garantizará la configuración e infraestructura necesaria para establecer y definir los elementos de la solución que son del dominio de la misma (requisitos funcionales y no funcionales).

El sistema forma parte de la solución general para automatizar los Registros y Notarias en la República Bolivariana de Venezuela. Por tanto, toma como punto de partida la Arquitectura Base General de dicha solución y de ella hereda algunos elementos tales como: la topología de red (ver Anexo 1) y su infraestructura, la Arquitectura de Datos (ver Anexo 2), los mecanismos de seguridad (basada en roles), el lenguaje de programación (C Sharp), algunos puntos del modelo de despliegue (ver Anexo 3) y la utilización de un framework que sustenta el desarrollo basado en capas.

Sin embargo debido a la magnitud del Sistema de Administración Financiera y a partir de experiencias pasadas, la Línea Base para este software rompe con la estructura horizontal y vertical que presenta la Arquitectura general y surge como una propuesta que se acerca más a las necesidades del producto y a las particularidades de la estructura financiera (ver figura 2) que presenta la institución.

La aplicación tiene que coexistir e interactuar con las soluciones de software desarrolladas para los Registros Públicos y Mercantiles. Por tanto, la Arquitectura también incluye los mecanismos de integración con dichos sistemas y dada las características del entorno donde se desenvuelve, su estructura es lo suficientemente flexible como para soportar cualquier interacción con otro módulo o subsistema que pueda surgir sin repercutir significativamente en el código fuente de la misma.

2.4 Representación Arquitectónica

Como se ha mencionado la Arquitectura del sistema se presenta desde dos enfoques:

2.4.1 Enfoque Horizontal

Para dar respuesta a todos los procesos y funcionalidades que se quieren acometer, el sistema de Administración Financiera comprende 10 módulos que eventualmente pueden interactuar para satisfacer determinado requerimiento:

- Administración.
- Presupuesto.
- Contabilidad.
- Recaudación.
- Requisiciones.
- Compras y Servicios.
- Retenciones.
- Tesorería.
- Fondos en Anticipo.

- Fondos de Caja Chica.

Teniendo en cuenta las dimensiones del software, se hace imprescindible realizar iteraciones sobre el documento de Línea Base que puedan marcar hitos en el proceso de desarrollo y por tanto obtener una medida de la prioridad y el avance que se vaya alcanzando. A partir de este análisis y tomando como punto de partida los requisitos que incorporan los módulos de manera individual y en conjunto, se debe dividir el proceso de desarrollo en 3 iteraciones.

Sin embargo, debido a que este documento se centra en el resultado de la primera iteración, se hace hincapié solamente en los principales requisitos funcionales que se obtienen en la misma. En posteriores iteraciones se irán incorporando los restantes.

Iteración 1:

Comprende el desarrollo del módulo de Administración, Presupuesto, Recaudación y Contabilidad.

Según RN-AFDR-02 Documento de Requerimientos del módulo Administración, el sistema debe permitir:

- Generar las propiedades de la UAC.
- Relacionar las UEL que pertenecen a la UAC por su dependencia financiera.
- Generar las propiedades de la UAC.
- Relacionar las UEL que pertenecen a la UAD por su dependencia financiera.
- Capturar las UEL con sus datos.
- Relacionar las UEL con el codificador de oficinas de Registros y Notarías.
- Definir el año a utilizar
- Registra las UEL dependientes.
- Crear coordinadores a las UEL.
- Gestionar las monedas que se utilizarán en el sistema.
- Gestionar el Plan de Cuentas Patrimoniales establecido en el país.
- Gestionar el Clasificador Presupuestario.
- Gestionar el Clasificador Económico.

- Relacionar las cuentas Patrimoniales con las Presupuestarias.
- Relacionar las cuentas Patrimoniales con las Económicas.
- Gestionar el Codificador de Bienes y Servicios establecido.
- Relacionarlo con el de Unidad de Medida.
- Gestionar el codificador de Unidad de Medida.
- Gestionar el codificador de Ejercicios Fiscales.
- Gestionar el codificador de Períodos Contables por cada Ejercicio fiscal.
- Gestionar el codificador de bancos.
- Gestionar el codificador de sucursales bancarias por cada banco.
- Gestionar las monedas que se utilizarán en el sistema.

Según RN-AFDR-03 Documento de Requerimientos del módulo Presupuesto, el sistema debe permitir:

- Gestionar las fuentes de financiamiento.
- Crear un proyecto nuevo.
- Gestionar las acciones específicas asociadas al proyecto que se desea crear.
- Gestionar las UEL asociadas a cada Acción Específica.
- Gestionar las fuentes de financiamiento asociadas a cada Acción Específica.
- Gestionar las fuentes de financiamiento del proyecto que se esté creando.
- Crear una Acción Centralizada.
- Gestionar las acciones específicas asociadas a la Acción Centralizada que se desea crear.
- Gestionar los datos de la Acción Especifica que se desea crear.
- Gestionar las partidas presupuestarias asociadas a cada Acción Especifica.
- Capturar los Indicadores Generales del Anteproyecto de Presupuesto.
- Gestionar las Fuentes de Financiamiento del Anteproyecto de Presupuesto.
- Formular el Presupuesto por Proyecto.
- Formular el Presupuesto por Acción Centralizada
- Gestionar formulación de un proyecto seleccionado.
- Configurar las Acciones Específicas de un proyecto.
- Asignar las Fuentes de Financiamiento de una Acción Específica.

- Asignar las partidas presupuestarias a la UEL seleccionada para realizar la Asignación Presupuestaria del Gasto.
- Capturar la Distribución Física y Financiera para realizar la Distribución Anual de la UEL seleccionada.
- Gestionar formulación de una Acción Centralizada seleccionada.
- Configurar las Acciones Específicas de una Acción Centralizada.
- Asignar las Fuentes de Financiamiento de una Acción Específica.
- Gestionar una UEL.
- Modificar una partida presupuestaria.
- Realizar un desglose de la partida presupuestaria seleccionada por Bienes/Servicios, por Entes o por un monto general.
- Configurar las Acciones Específicas asociadas a cada proyecto.
- Configurar las unidades ejecutaras locales asociadas a esa Acción Específica.
- Configurar las Acciones Específicas asociadas a esa Acción Centralizada.
- Gestionar una modificación presupuestaria.
- Cambiar el estado a una modificación presupuestaria.
- Crear modificaciones presupuestarias según tipo de modificación.
- Gestionar por cada trimestre la programación de sus metas por proyectos o acciones centralizadas.
- Aprobar la Programación Trimestral de las Metas.
- Gestionar por cada trimestre la programación de la Ejecución Financiera por proyectos o acciones centralizadas.
- Aprobar la Programación Trimestral de la Ejecución Financiera.
- Seleccionar el Año Presupuestario al cual se le va hacer la programación de Ingresos y Desembolsos.
- Gestionar por cada mes y por Denominación el monto y la Cuota de compromisos pendientes.

Según RN-AFDR-04 Documento de Requerimientos del módulo Contabilidad, el sistema debe permitir:

- Gestionar los comprobantes contables (Adicionar, eliminar y modificar un comprobante contable).
- Conformar un comprobante que se encuentre en estado de edición.
- Anular un comprobante una vez que el mismo este en estado de confirmación.
- Registrar los asientos contables patrimoniales y presupuestarios.

- Generar el comprobante contable de cierre.

Según RN-AFDR-01 Documento de Requerimientos del módulo Recaudación, el sistema debe permitir:

- Recibir y registrar la Planilla Única Bancaria (PUB) correspondiente al trámite a realizar a partir de los Módulos Registrales y Notariales en cada una de las oficinas.
- Enviar información de las PUB emitidas a la Unidad de Recaudación.
- Verificar y Actualizar el estado de la PUB.
- Enviar información de las PUB caducadas a la Unidad de Recaudación.
- Enviar información de las PUB pagadas a los registros.
- La recepción de la confirmación del pago de la PUB en el banco.
- El envío de la confirmación del pago de la PUB a la oficina y a la Unidad de Contabilidad.
- Validar la PUB que fue pagada en el banco.
- Capturar los datos del pago de la PUB para ser tramitada.
- Generar el asiento contable de la contabilidad patrimonial.
- Generar el asiento contable para la ejecución del presupuesto de ingreso.
- Enviar la PUB tramitada a la Unidad de Recaudación.
- Recibir un resumen diario emitido por el banco de las PUB pagadas durante el día, la información se enmarca en la identificación del banco, la fecha, la cantidad y el monto total.
- Conciliar la relación de cada una de las PUB pagadas durante el día con el resumen diario emitido por el banco.
- Enviar una petición al banco solicitando la relación de todas las PUB pagadas en caso de no estar conciliado.
- Recibir del banco la relación de todas las PUB pagadas y comparar con las recepcionadas durante el día para determinar cuáles son las diferencias que existen, estas diferencias serán categorizadas para una mejor comprensión por parte del Responsable de Recaudación.
- Gestionar la PUB en la vista de diferencias.
- Gestionar los reintegros solicitados tanto parciales como totales.
- Cambiar el estado del reintegro una vez confirmado.
- Mostrar las PUB relacionadas con el usuario que solicita el reintegro en la oficina correspondiente.

- Cambiar el estatus de las PUB una vez confirmado el reintegro.
- Generar el asiento contable correspondiente.
- Recaudar las PUB pagadas que vencieron el tiempo para realizar el trámite.
- Calcular la diferencia entre lo recaudado y trasferido en tiempo y monto.

De manera general el sistema tiene que:

- Generar y visualizar los reportes asociados a cada módulo.
- Imprimir los reportes.
- Exportar los reportes a formato PDF, XLS y DOC.

Iteración 2:

Comprende el desarrollo del módulo Requisiciones, Compras y Servicios, Retenciones y Tesorería

Iteración 3:

Comprende el desarrollo de los módulos Fondos en Anticipo y Fondos de Caja Chica.

En la sección correspondiente a la Vista Lógica y la de Implementación se verá con más detalles las relaciones que se establecen entre cada uno de estos módulos y sus especificidades internas.

Esta distribución se basa fundamentalmente en las dependencias que se establecen entre los módulos y en alguna medida en las prioridades que tiene el cliente con cada uno de ellos. Precisamente, al final de cada iteración obtenemos un fragmento de funcionalidad del sistema completo agrupadas de manera tal que el proceso de implementación sea fluido e ininterrumpido. No obstante como se verá, siempre quedan conexiones y dependencias que necesitan resolverse, esto significa que la Arquitectura tiene que incluir algún mecanismo para solventar esta situación.

La solución radica en la utilización de interfaces que representen las funcionalidades de cada subsistema, de esta forma su implementación es transparente al módulo que las utiliza y pude cambiar y evolucionar tantas veces como sea necesario sin necesidad de involucrarlo.

Para enlazar las funcionalidades con su implementación, es decir, la interfaz con la (s) clases que la implementan, se hace uso de fachadas desarrolladas sobre Spring.net y la implantación de patrones como el Proxy buscando garantizar en todo momento el cumplimiento de uno de los principios de la programación, que tiene su base en el patrón GRASP Bajo Acoplamiento descritos en [LC2004]. Además se incluye el establecimiento de objetos falsos para simular posibles resultados ante diferentes llamadas de las funcionalidades.

Por otra parte, la distribución horizontal de la solución descansa sobre un subsistema donde se van acoplando cada módulo luego de haberlo terminado y haya pasado por la etapa de prueba. Los módulos se acoplarían como plug-ins[7] que puedan agregarse y quitarse de la solución según se necesite, de este modo se tiene en todo momento una versión estable e independiente del desarrollo. Igualmente, a medida que se vaya cumpliendo cada una de las iteraciones propuestas, se realizarán las pruebas de la integración que serían a este nivel de organización teniendo en cuenta los resultados que se esperan obtener de las funcionalidades previstas.

Como se puede ver la Arquitectura está basada en componentes y conectores. Esta distribución garantiza mantener un desarrollo estable, seguro y al mismo tiempo agiliza el proceso de implementación.

En el caso de la distribución vertical se mantiene esta misma idea e incorpora algunas que contribuyen al enriquecimiento de la solución y a garantizar el cumplimiento de los atributos de calidad que son objetivo del presente trabajo.

2.4.2 Enfoque Vertical

[7] Un plugin (o plug-in -en inglés "enchufar"-, también conocido como addin, add-in, addon o add-on) es una aplicación informática que interactúa con otra aplicación para aportarle una función o utilidad específica, generalmente muy específica, como por ejemplo servir como driver (controlador) en una aplicación, para hacer así funcionar un dispositivo en otro programa.

El enfoque vertical se refiere a la distribución de la Arquitectura para cada módulo. Este tipo de distribución puede dar cabida a varias configuraciones diferentes, en este caso particular se adoptan las que garantizan con más eficiencia los atributos de calidad que se desean priorizar, los cuales constituyen el objetivo fundamental de la investigación.

Siguiendo la idea anterior, este enfoque está separado en dos estructuras que presentan configuraciones diferentes. La primera distribución se corresponde con el modulo Común que tiene características muy peculiares dada su función dentro de la Arquitectura según la figura (ver figura 3), la segunda viene dada por la generalidad de la solución y está representada en la figura (ver figura 4).

En una arquitectura de n niveles como esta, las acciones de la aplicación están lógicamente divididas por funciones. Lo anterior se basa en un estilo y patrón multicapa como se puede ver en las figuras (ver figura 3 y 4).

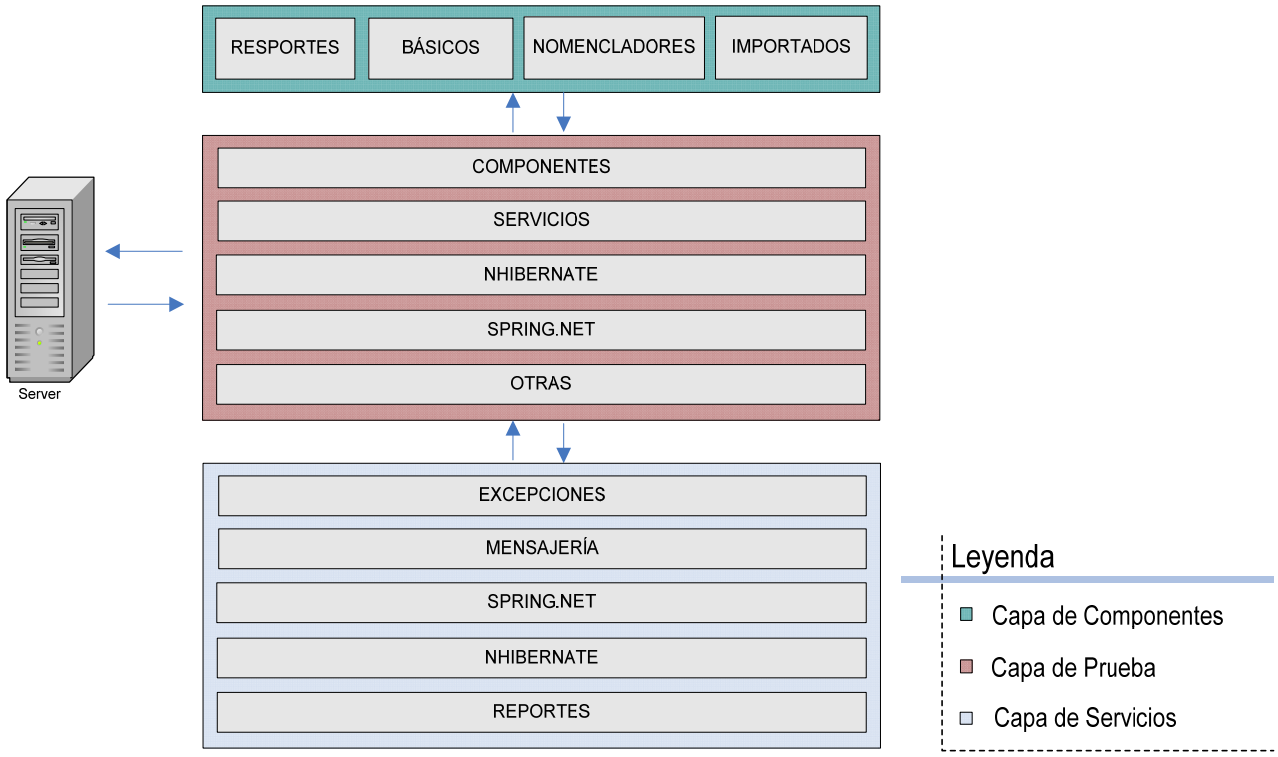


Figura 3 Representación del enfoque vertical de la Arquitectura módulo Común.

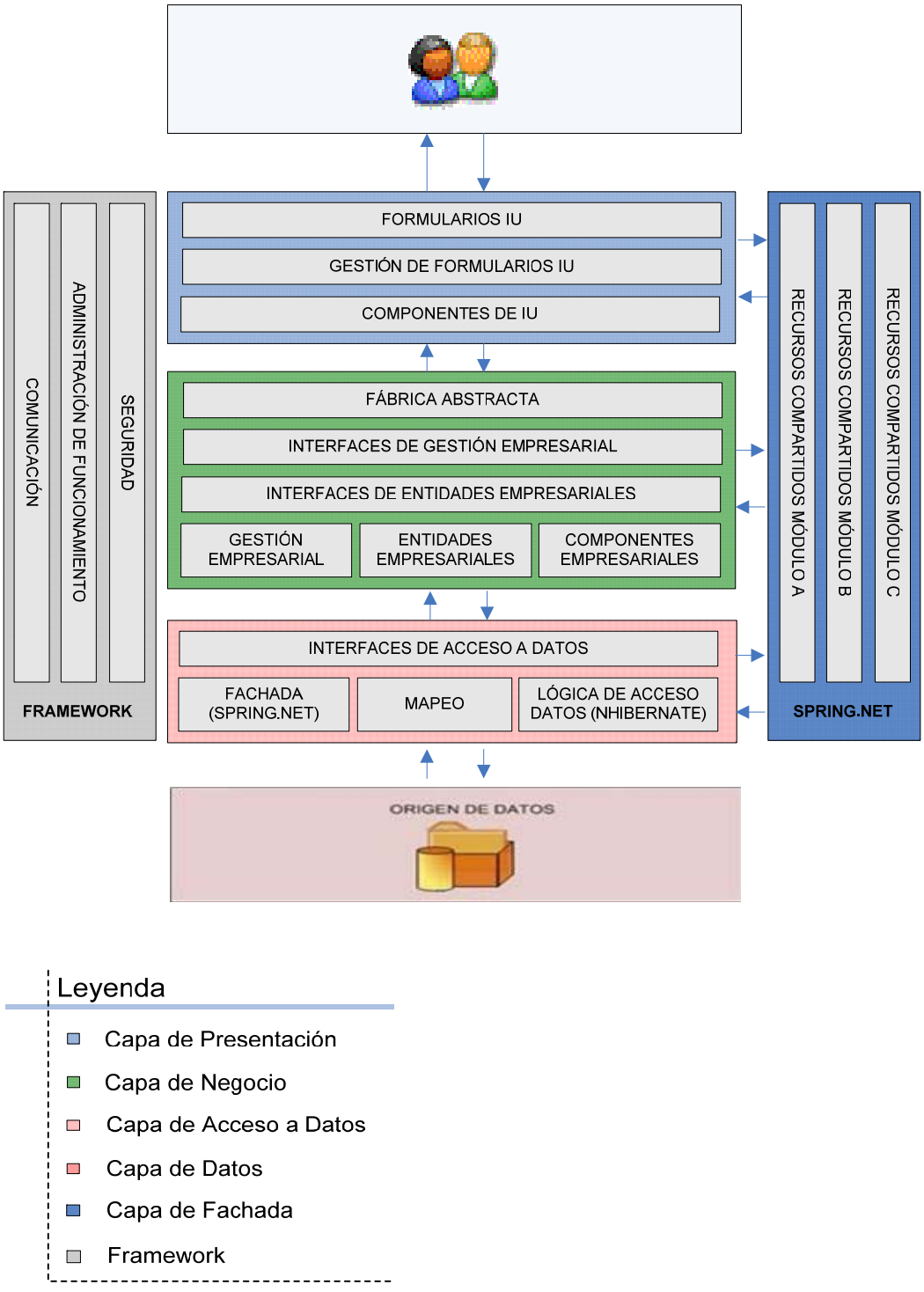


Figura 4 Representación del enfoque vertical de la Arquitectura general.

De manera general lo más representativo de esta distribución es la fuerte presencia que tienen los componentes y más aún los conectores, estos últimos resueltos en su mayoría con la utilización de interfaces y la implementación del Patrón Fachada, Proxy y la utilización del framework Spring.net como ya se habían mencionado. A continuación se detallan cada una de las capas y los componentes que conforman la perspectiva vertical de la Arquitectura:

- **Capa de Presentación:**

Formularios de Interfaz de Usuario.

Los formularios responden a un comportamiento y diseño estándar, todos los módulos tienen la misma forma con vistas a facilitar el trabajo con ellos y la estandarización del sistema completo.

Su uso se basa en la explotación de los Formularios de Interfaz de Usuario que realmente provee un framework (Framework Común) facilitando el desarrollo del sistema. Este framework se basa en acciones contempladas en la Gestión de Formularios de Interfaz de Usuario y cada acción se corresponde con funcionalidades de captura o visualización de los datos que tiene asociado un formulario cuya forma depende de la funcionalidad específica para la cual fue concebida dicha acción.

Gestión de Formularios de Interfaz de Usuario.

Su uso se basa en la explotación del componente gestión de interfaz que realmente provee el framework antes mencionado facilitando el desarrollo del sistema basado en acciones.

¿Qué es una acción?

- Cada acción debe tener sentido semántico propio y completo. Deben ser atómicas.
- Las acciones básicamente definen un bloque de código reusable por varias operaciones sobre el sistema.
- Las acciones están asociadas a vistas del sistema.
- Cada acción puede representar un estado del sistema que puede o no persistir.

- Una acción es una clase que representa precisamente la ejecución de alguna tarea concreta. Estas tareas no deben ser excesivamente complejas y la clase debe: Heredar de la clase **"Accion"** o "**AccionSegura"** (Framework Común).
- Propiedades en función del objetivo específico de la misma.
- Se le debe reescribir el método CrearForma con vistas a controlar lo que se desee del formulario asociado.
- Se deben programar dentro de la misma todos los eventos que puedan ocurrir y que se deseen utilizar a partir de la interacción con la forma visual.

Componentes de Interfaz de Usuarios.

Los componentes de interfaz de usuarios son recursos empleados para encapsular una función determinada y serán utilizados por más de un proceso en el módulo. El hecho de que estén en la Capa de Presentación se debe a que trabajan con los formularios directamente, ya sea de forma visual o con los datos que se encuentran relacionados en el mismo a través de otros componentes o controles. Cada componente puede incorporar clases de negocio e inclusive elementos de acceso a datos encapsulados y distribuidos según la estructura que tiene la Arquitectura general.

- **Capa Lógica de Negocio**

El diseño de esta capa depende directamente del negocio específico al que se refiera cada módulo.
El Negocio recibe datos y/o información capturada en las interfaces de usuario, gestiona o procesa la misma, de ser necesario solicitándola a la capa de Acceso a Datos y finalmente enviársela a la Presentación nuevamente para que esta la muestre al usuario en el punto donde se inició la petición.

Entidades del Negocio.

Las entidades empresariales son clases objeto - valor que representan los datos con los que se van a trabajar en cada uno de los procesos que se están automatizando.

Cada entidad es un elemento auto sustentado, es decir, se encarga de procesar sus propios datos o valores sin interactuar con los demás elementos del negocio, con esto se garantiza la independencia y el encapsulamiento de la información según la competencia que se tenga sobre la misma.

Gestión Empresarial.

Este ensamblado tiene como objetivo agrupar una serie de entidades con un fin común, lo anterior significa resolver determinadas funcionalidades donde intervienen una serie de datos que se encuentran en dichas clases objeto - valor. Las clases correspondientes a las funcionalidades se denominan Gestores.

Los Gestores serán clases estáticas puesto que solamente representan funcionalidades, por tanto no es necesario instanciarlos.

Componentes Empresariales.

Los componentes de negocio son recursos utilizados para encapsular una función determinada que será utilizada por más de un proceso de negocio en el módulo. Lógicamente estas funcionalidades son netamente de este nivel, es decir no tienen presentación pero si pueden utilizar recursos del acceso a datos.

Interfaces de Gestión y Entidades.

El pilar fundamental de la arquitectura radica en el uso de las interfaces como recurso que se utiliza para brindar funcionalidades que representan un nivel de abstracción. Este nivel es precisamente el que asegura el patrón Bajo Acoplamiento [LC2004] conjuntamente con otros elementos, algunos de los cuales ya se han visto.

La mayoría de las interfaces de la aplicación van a estar precisamente en el Negocio, puesto que aquí se encuentran en gran medida de la implementación de estas funcionalidades, ya sea en las Entidades, los Gestores o los Componentes de Gestión.

- **Capa Acceso a Datos**

Definir la estrategia de persistencia de una aplicación es una de las decisiones de Arquitectura más importantes. En una aplicación estándar más del 50% del código generado está relacionado con lógica de persistencia [RH2002].

Por tanto, el Acceso a Datos es la capa más crítica y sensible a cambios de la Arquitectura, pues controla todo lo concerniente a la información que se encuentra en la fuente de almacenamiento (Capa de Datos).

Al ser la capa inferior no conoce los niveles superiores, únicamente se limita al manejo de la información, ya sea para persistirla o proporcionarla para su procesamiento y propagación por la aplicación. Todo este manejo es responsabilidad de los Objetos de Acceso a Datos (DAOs por sus siglas en ingles).

Lógica de Acceso a Datos.

Todas las funcionalidades del Acceso a Datos radican en los DAOs, es decir este ensamblado implementa la totalidad de las operaciones de persistencia y obtención de datos explotando los recursos que brinda NHibernate framework que cumple perfectamente con el objetivo de este nivel, dígase trabajo con procedimientos almacenados y métodos de persistencia o consultas.

Fachada.

La fachada brinda una interfaz de alto nivel con la cual se va a interactuar cada vez que se necesite comunicarse, en este caso con el Acceso a Datos logrando una completa enajenación ante cualquier modificación que pueda ocurrir en la misma.

Esta estructura responde a la implementación de patrón Fachada, Proxy y está constituida por una clase que brinda el modulo Común y una estructura que enumera (*enum*[8]) los DAOs a los cuales se puede acceder desde el negocio.

Precisamente la utilidad del patrón en este nivel, radica en los elementos que se mencionaron anteriormente (criticidad, inestabilidad[9]), para su implementación se decidió el uso de Spring.net versión 1.1, framework que bajo el concepto de IoC resuelve perfectamente el problema planteado.

Mapeo.

Este elemento es de uso exclusivo de NHibernate como se verá, se decidió aislarlo en un ensamblado ya que básicamente son ficheros XML de configuración que son independientes de cualquier implementación, por tanto resulta muy útil tenerlos separados del código y así se evita recompilar toda una capa ante cualquier modificación en una de estas configuraciones.

Interfaces de Acceso a Datos.

Representan las funcionalidades que brinda este nivel, es decir las referidas a los DAOs. Su uso e importancia es la misma que las de la capa Lógica de Negocio.

- **Capa de Fachada**

La Capa de Fachada es una representación del patrón Proxy y Fachada a gran escala, este nivel es un recurso utilizado para mantener una representación de los demás módulos en el que se está implementando, siempre y cuando lo necesite. Es decir, cada módulo que se vaya a comunicar con el otro debe dar las funcionalidades que necesita (ver figura 4), las cuales se van a agrupar aquí. Esta distribución tiene su utilidad en el enfoque horizontal de la Arquitectura. Al emplear esta forma de

[8] La palabra clave **enum** se utiliza para declarar una enumeración, un tipo distinto que consiste en un conjunto de constantes con nombre denominado lista de enumeradores.

[9] Se refiere a que es una capa que está muy expensa a cambios.

comunicación entre los módulos se garantiza la abstracción y enajenación gracias a Spring.net, el cual ya se ha mencionado con anterioridad.

- **Capa de Datos**

Esta capa corresponde a los almacenes de datos. A ella pertenecen las bases de datos disponibles en los servidores de bases de datos. Esta capa es única y común a todos los módulos del sistema general. Su definición es la misma que la de la Arquitectura Base heredada de los Sistemas Registrales que ya se han mencionado (ver Anexo 2).

2.5 Elementos arquitectónicamente significativos.

Resultan elementos significativos en la Arquitectura el uso de frameworks/marcos de trabajo, los cuales brindan una serie de ventajas, principalmente porque comprenden funciones que resultan claves y muy útiles para lograr el cumplimiento del objetivo general.

Otros componentes importantes son los patrones implementados que en su conjunto brindan determinada funcionalidad que también da soporte a la arquitectura.

A partir de este análisis es importante reflejar cómo se lleva a cabo la implantación de aquellos elementos en la solución propuesta que mayor complejidad y dificultades puedan traer en el momento de utilizarlos.

2.5.1 NHibernate.

Para implementar NHibernate en la capa de Acceso a Datos se parte de que existe una Base de Datos Relacional y las tablas asociadas a cada clase persistente [BK2005] (estas clases son las entidades del negocio).

- Se crean los archivos mapping de cada entidad (ver figura 5 y 6).

- Se crea el archivo de configuración donde van a estar todos los datos de inicialización y conexión con la Base de Datos que necesita NHibernate (ver figura 7).

Todo lo concerniente a las funcionalidades de manipulación de la información para la persistencia y la obtención de los datos se maneja en los DAOs, para esto el framework propone sus propias implementaciones, las cuales se encuentran en el ensamblado principal NHibernate.dll (ver código 1). Las funcionalidades incluyen inserción, actualización, eliminación, etc. y el uso de procedimientos almacenados como un recurso importante en toda la lógica de acceso a datos.

En el caso de los procedimientos almacenados se tiene que hacer un archivo de mapeo XML independiente, donde justamente se relacionan todos estos procedimientos. Por otra parte, usando Oracle como gestor de Base de Datos (ver figura 8), luego de mapear los procedimientos el sistema está preparado para su uso y explotación a través de los recursos que en este sentido ofrece NHibernate (ver código 2).

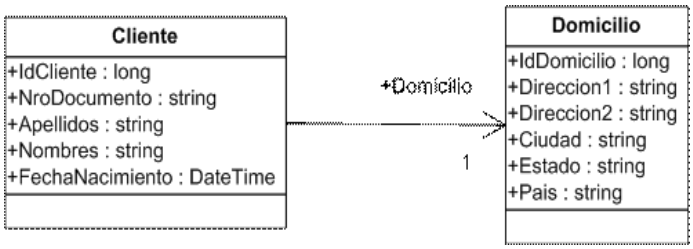


Figura 5. Representación de dos entidades con una relación de uno a muchos.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.0">
    <class name="EjemploNHibernateModeloDominio.Domicilio, EjemploNHibernateModeloDominio" table="DOMICILIO">
        <id name="IdDomicilio" column="ID_DOMICILIO" type="Int64" unsaved-value="0">
            <generator class="sequence">
                <param name="sequence">SEQ_DOMICILIO</param>
            </generator>
        </id>
        <property name="Direccion1" column="DIRECCION1" type="String" length="50"/>
        <property name="Direccion2" type="String" length="50"/>
        <property name="Ciudad" column="CIUDAD" type="String" length="50"/>
        <property name="Estado" column="ESTADO" type="String" length="50"/>
        <property name="Pais" column="PAIS" type="String" length="50"/>
    </class>
</hibernate-mapping>
```

```xml
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.0">
    <class name="EjemploNHibernateModeloDominio.Cliente, EjemploNHibernateModeloDominio" table="CLIENTE">
        <id name="IdCliente" column="ID_CLIENTE" type="Int64" unsaved-value="0">
            <generator class="sequence">
                <param name="sequence">SEQ_CLIENTE</param>
            </generator>
        </id>
        <property name="NroDocumento" column="NRO_DOCUMENTO" type="String" length="20"/>
        <property name="Apellidos" column="APELLIDOS" type="String" length="50"/>
        <property name="Nombres" column="NOMBRES" type="String" length="50"/>
        <property name="FechaNacimiento" column="FECHA_NACIMIENTO" type="DateTime"/>

        <many-to-one name="Domicilio" column="ID_DOMICILIO" cascade="all"/>
    </class>
</hibernate-mapping>
```

Figura 6. Ejemplo de los archivos de mapeo que genera la relación anterior.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="nhibernate" type="System.Configuration.NameValueSectionHandler, System,
      Version=1.0.5000.0,Culture=neutral, PublicKeyToken=b77a5c561934e089" />
    <section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler,log4net" />
  </configSections>

  <nhibernate>

    <add key="hibernate.connection.provider"
         value="NHibernate.Connection.DriverConnectionProvider"
    />

    <add key="hibernate.dialect"
         value="NHibernate.Dialect.OracleDialect"
    />

    <add
      key="hibernate.connection.driver_class"
      value="NHibernate.Driver.OracleClientDriver"
    />

    <add key="hibernate.connection.connection_string"
         value="Data Source=XE;User ID=user;Password=user;"
    />

  </nhibernate>
</configuration>
```

Figura 7. Archivo de configuración.

```
ISession session[10] = null;
 ITransaction transaction = null;
 try
 {
     session = sessionFactory.OpenSession();
     transaction = session.BeginTransaction();
     session.Metodo(parametro);
     transaction.Commit();
     MessageBox.Show("Cliente grabado correctamente");
     btnLimpiar_Click(sender, e);
```

[10] 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 nos permite manejar el contexto transaccional de nuestra lógica.

```
    }
    catch (Exception ex)
    {
        if (transaction != null) transaction.Rollback();
        MessageBox.Show(ex.Message);
    }
    finally
    {
        if (session != null) session.Close();
    }
}
```

……..

Código 1. Código que representa los pasos para la llamada a los métodos implementados por NHibernate.

Utilizando el método ISession.BeginTransaction() se obtiene un objeto del tipo ITransaction que representa la transacción que se utilizará para llamar al método que trabajará con los datos.

```
transaction = session.BeginTransaction();
```

Siguiendo el ejemplo, una vez iniciada la transacción, lo único que se tiene que hacer para salvar automáticamente el objeto Cliente y el objeto Domicilio asociado, es utilizar el método ISession.SaveOrUpdate(Object o) pasándole como argumento el objeto que se desea salvar.

```
session.SaveOrUpdate(cliente);
```

Utilizando el unsaved - value especificado en ambos mappings, NHibernate determina si tiene que insertar un nuevo objeto o actualizar uno existente. En una sola línea de código el framework resuelve en forma automática la grabación en ambas tablas (Cliente y Domicilio).

Luego de salvar el objeto, lo único que resta es confirmar la transacción en curso. En este sentido lo único que se tiene que hacer es invocar el método ITransaction.Commit().

```
transaction.Commit();
```

Si ocurre algún error, se deshacen todos los cambios utilizando el método ITransaction.Rollback();

```
catch (Exception ex)
{
    if (transaction != null) transaction.Rollback();
    MessageBox.Show(ex.Message);
}
```

Finalmente hay que asegurarse de cerrar siempre la sesión utilizando el método ISession.Close();

```
finally
{
    if (session != null) session.Close();
}
```

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.0">

    <store-procedure name="spObtenerClientes" procedure-name= "PKG_REPORTES.ObtenerClientes">

        <parameter name="io_AllRec"/>

        <return-list return-class="EjemploNHibernateModeloDominio.Cliente, EjemploNHibernateModeloDominio">

            <return-property name="IdCliente" num-col="0"/>
            <return-property name="NroDocumento" num-col="1"/>
            <return-property name="Apellidos" num-col="2"/>
            <return-property name="Nombres" num-col="3"/>
            <return-property name="FechaNacimiento" num-col="4"/>

            <return-object name="Domicilio">
                <return-property name="IdDomicilio" num-col="5"/>
                <return-property name="Direccion1" num-col="6"/>
                <return-property name="Direccion2" num-col="7"/>
                <return-property name="Ciudad" num-col="8"/>
                <return-property name="Estado" num-col="9"/>
                <return-property name="Pais" num-col="10"/>
            </return-object>

        </return-list>

    </store-procedure>

</hibernate-mapping>
```

Figura 8. Archivo de mapeo de los procedimientos almacenados.

……

1. session = sessionFactory.OpenSession();
   transaction = session.BeginTransaction();
   ArrayList clientes = session.CreateStoreProcedure("spObtenerClientes").List();
   transaction.Commit();

2. ArrayList clientes = session.CreateStoreProcedure("spObtenerClientesPorFechaNac")
   .SetParameter(inicial)
   .SetParameter(final)
   .List();

3. ArrayList clientes = session.CreateStoreProcedure("spObtenerClientesPorFechaNac")

```
                    .SetParameter(inicial)
                    .SetParameter(final)
                    .GetDataSet();

4. ArrayList clientes = session.CreateStoreProcedure("spEliminarClientes")
                    .SetParameter(idcliente)
                    .ExecuteNonQuery();
```

……..

Código 2. Ejemplos del uso de procedimientos almacenados con NHibernate, dependiendo del mapeo.

Se han descrito una serie de eventos que realiza NHibernate relacionados fundamentalmente con transacciones verticales. Resta definir como se realiza el trabajo con las transacciones horizontales las cuales involucran una serie de procesos que no deben persistir hasta que cada uno de ellos se ejecute completamente. Lo anterior significa que la información no se registra en la Base de Datos hasta que no se hayan realizado correctamente (sin Excepciones) todas las operaciones envueltas en la transacción.

Para el trabajo con estas transacciones y más aún con las funcionalidades básicas de NHibernate para Salvar, Eliminar, Actualizar, etc. se va a implementar en el módulo “ArquitecturaBase“ un componente de negocio que maneje todas estas funciones, las cuales trabajan en su totalidad con transacciones verticales. Así mismo, el negocio encapsulado en este componente encuentra su contraparte de Acceso a Datos, según la arquitectura vertical, la cual incluye además otros métodos para configuraciones, inicializaciones y precisamente el trabajo con transacciones. En consecuencia, cada vez que se necesite en cada módulo, trabajar con transacciones horizontales, solamente habría que acceder a este componente e interactuar directamente con los métodos dedicados a las transacciones embebiendo el código que se quiera en los mismos según se muestra (ver código 3).

```
……
objDaoComun.IniciarTransaction();
objDaoComun.EliminarColeccionObjetos(directoresAEliminar);
if (dependenciasAEliminar != null)
{
        foreach (IDependenciaFinancieraUACs objeto in dependenciasAEliminar)
        {
                if (objeto.ObjEstructuraFinanciera.IdEstructuraFinanciera != 0)
                        objDaoComun.Eliminar(objeto);
        }
}
objDaoComun.Salvar(objUAC);
if (objEstructura != null)
{
        objDaoComun.SalvarOActualizar(objEstructura);
        if (objEstructura.ColeccionDependenciasFinancieras != null)
        {
                foreach (IDependenciaFinancieraUACs objeto in
                    objEstructura.ColeccionDependenciasFinancieras)
                {
                        IDependenciaFinancieraUACs dependencia = new
                        EDependenciaFinancieraUACs(objEstructura,objeto.ObjUEL);
                        objDaoComun.SalvarOActualizar(dependencia);
                }
        }
}
objDaoComun.ReafirmarTransaction();
……
```

Código 3 Ejemplo del uso de transacciones horizontales.

2.5.2 Spring.net

Atendiendo a las ventajas y desventajas que se mencionaron en el Capítulo 1 referentes a este framework, se decidió implantar Spring.net en la capa de Acceso a Datos específicamente en su fachada y en la Capa Fachada de los módulos, con esto se garantiza que se cumplan todos los objetivos que se persiguen en cada una de ellas explotando las virtudes que tiene el framework. Es decir, con vistas a no sacrificar el rendimiento de la aplicación son estas dos capas las únicas en implantar esta variante, más aún por ser tan importantes y sensibles; donde resulta más útil sacrificar algo de rendimiento con vistas a ganar en desacoplamiento y abstracción de la información. A esto se le suma que en estos niveles no se manejan la mayoría de objetos y por tanto dependencias.

Para utilizar Spring.net primeramente se establecen todas sus configuraciones en el fichero app.config (ver figura 9), además se relacionan todos los objetos que se van a utilizar y sus dependencias.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<configSections>
   <sectionGroup name="spring">
        <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
        <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
   </sectionGroup>
</configSections>
<spring>
    <context>
      <resource uri="config://spring/objects" />
    </context>
<objects xmlns="http://www.springframework.net">

<object id="GtrContabilidad" type="Contabilidad.Negocio.Gestion.GtrContabilidad,
 Contabilidad.Negocio.Gestion">
    <constructor-arg index="0" ref="DaoComun"/>
    <constructor-arg index="1" ref="DaoOperacionesRegistroComp"/>
    <constructor-arg index="2" ref="DaoReportes"/>
</object>

<object id="DaoOperacionesRegistroComp"
 type="Contabilidad.Acceso.Datos.Daos.DaoOperacionesRegistroComp,
 Contabilidad.Acceso.Datos.Daos"/>

<object id="DaoReportes" type="Contabilidad.Acceso.Datos.Daos.DaoReportes,
 Contabilidad.Acceso.Datos.Daos"/>

<object id="DaoComun" type="ArquitecturaBase.Acceso.Datos.Daos.DaoComun,
 ArquitecturaBase.Acceso.Datos.Daos"/>
</objects>
</spring>
</configuration>
```

Figura 9. Archivo con las configuraciones de Spring.

Una estrategia utilizada para trabajar con este framework es el uso de interfaces, pues que con ellas se mantienen representaciones de objetos, o sea funcionalidades que serán instanciadas a partir de la configuración de un fichero XML y no en el código, logrando así un total desacoplamiento entre módulos y entre el negocio y la capa de Acceso a Datos a través de su fachada.

Es importante aclarar que el uso de las interfaces en casi la totalidad de la aplicación y no solamente en aquellos que formarán parte de la comunicación con otros módulos o capas se debe a que indudablemente este tipo de implementación ofrece múltiples ventajas[11] que en un momento determinado se pueden aprovechar, además garantizamos uniformidad y tener preparada la aplicación ante cualquier cambio en las peticiones y funcionalidades que se dan.

```
……..
IApplicationContext ctx = ContextRegistry.GetContext();
return ctx[“nombre objeto”];
……..
```

Código 3. Obteniendo el objeto del fichero de configuración.

Con este código se resuelve el problema fundamental de obtener el objeto correspondiente a la clase que implementa la interfaz puesto que solamente se necesitaría el nombre con el que se identifica en el fichero de configuración y este inyectaría las dependencias que se tienen establecidas previamente en el XML.

Es importante señalar que el uso de Spring.net no se limita a las funcionalidades propias del framework, pues contiene una serie de implementaciones e integraciones con otros marcos [PESSHCR2004-2006], en este sentido el que resulta más atractivo es NHibernate sobre todo para el manejo de transacciones y funcionalidades a través de ficheros XML. Esta línea se sigue como recomendaciones para futuras evoluciones de esta Arquitectura. En resumen, podemos obtener un archivo de configuración más completo y la aplicación quedaría más organizada estructuralmente. Es decir a medida que se quiera explotar aún más las ventajas que brinda el framework solamente habría que establecer las instrucciones necesarias en el XML y hacer la llamada correspondiente a sus ensamblados.

[11] El uso de interfaces es un mecanismo para abstraer los métodos a un nivel superior, múltiples objetos de clases diferentes pueden ser tratados como si fuesen de un mismo tipo común, donde este tipo viene indicado por el nombre del interfaz.

2.5.3 Patrón Proxy y Reflection.

La implantación de estos patrones se conjuga en el módulo ”ArquitecturaBase” para obtener la funcionalidad deseada, que en este caso se refiere a la integración de sistema con otros que pueden coexistir en el medio algunos de los cuales ya han sido mencionados.

La implementación está sustentada en un fichero de configuración XML (ver figura 10) donde se relacionan todas las funcionalidades que se desean compartir a través de ensamblados.

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

<Ensamblados>

      <GtrPlanillaUnicaBancaria>
            <ensamblado>Recaudacion.Negocio.Gestion</ensamblado>
            <clase>Recaudacion.Negocio.Gestion.GtrPlanillaUnicaBancaria</clase>
      </GtrPlanillaUnicaBancaria>

      <ConceptoRecaudacion>
            <ensamblado>Recaudacion.Negocio.Entidades</ensamblado>
            <clase>Recaudacion.Negocio.Entidades.ConceptoRecaudacion</clase>
      </ConceptoRecaudacion>

</Ensamblados>
```

Figura 10 Fichero XML de configuración para las funcionalidades que se brindan.

A partir de este XML por mecanismos de Reflection se instancian las funcionalidades que se encuentran en las interfaces. El encargado de llevar a cabo dicha operación es el Proxy que sirve de intermediario con las aplicaciones externas. El código (ver código 4) muestra como se lleva a cabo todo este proceso.

```
……
public static object Instanciar(string nombreObjeto)
{
        try
        {
                Assembly asm = System.Reflection.Assembly.GetEntryAssembly();
                string nombreA = asm.FullName;
                nombreA = nombreA.Substring(0, nombreA.IndexOf(','));
                System.IO.Stream stream = asm.GetManifestResourceStream(nombreA +
                ".Ensamblados.xml");
                System.Xml.XmlDocument docXML = new System.Xml.XmlDocument();
                docXML.Load(stream);
                string nombreAsm = docXML["Ensamblados"][nombreObjeto]["ensamblado"].InnerText;
                Assembly asmPlugin =  Assembly.Load(nombreAsm);
                string nombreClase = docXML["Ensamblados"][nombreObjeto]["clase"].InnerText;
                System.Type tipo = asmPlugin.GetType(nombreClase);

                return tipo.GetConstructor(new Type[0] { }).Invoke(new object[0] { });

        }
        catch(Exception ex)
        {
                throw new Exception(ex.ToString());
        }
}
……
```

Código 4 Implementación del Proxy a través de mecanismos de Reflection.

2.6 Vista de Casos de Uso.

Según RUP en la Vista de Casos de Usos de la Arquitectura se relacionan aquellos que son arquitectónicamente significativos.

A continuación se presenta la Vista de Casos de Uso para la primera iteración de la Arquitectura:

- Administración.
- Presupuesto.
- Recaudación.
- Contabilidad.

2.6.1 Administración

El Módulo de Administración conforma la relación de todos los codificadores necesarios para el funcionamiento del resto de los módulos del sistema, así como la definición de los Indicadores Generales por cada uno de ellos.

Entre los codificadores fundamentales se encuentran: la Estructura Financiera que incluye la definición de la UAC, la gestión de las UAD y la creación de las UEL, además se configuran los Clasificadores de Cuentas Patrimoniales, Presupuestarias, Económico, el Catálogo de Bienes y Servicios y las relaciones que puedan establecerse entre ellos.

Teniendo en cuenta la descripción del módulo y RN-AFMCU-02 Documento de Caso de Uso del Sistema del módulo Administración, resultan Casos de Usos significativos para la Arquitectura:

- Definir Estructura Financiera de la UAC.
- Gestionar Unidad Administradora Desconcentrada.
- Gestionar Unidad Ejecutora Local.
- Gestionar Clasificador Económico.
- Gestionar Plan de Cuentas Patrimoniales.
- Gestionar Catálogo de Bienes y Servicios.
- Gestionar Clasificador Presupuestario.

En la figura (ver figura 12) se muestran estos casos de uso en la vista de Casos de Uso de la Arquitectura.

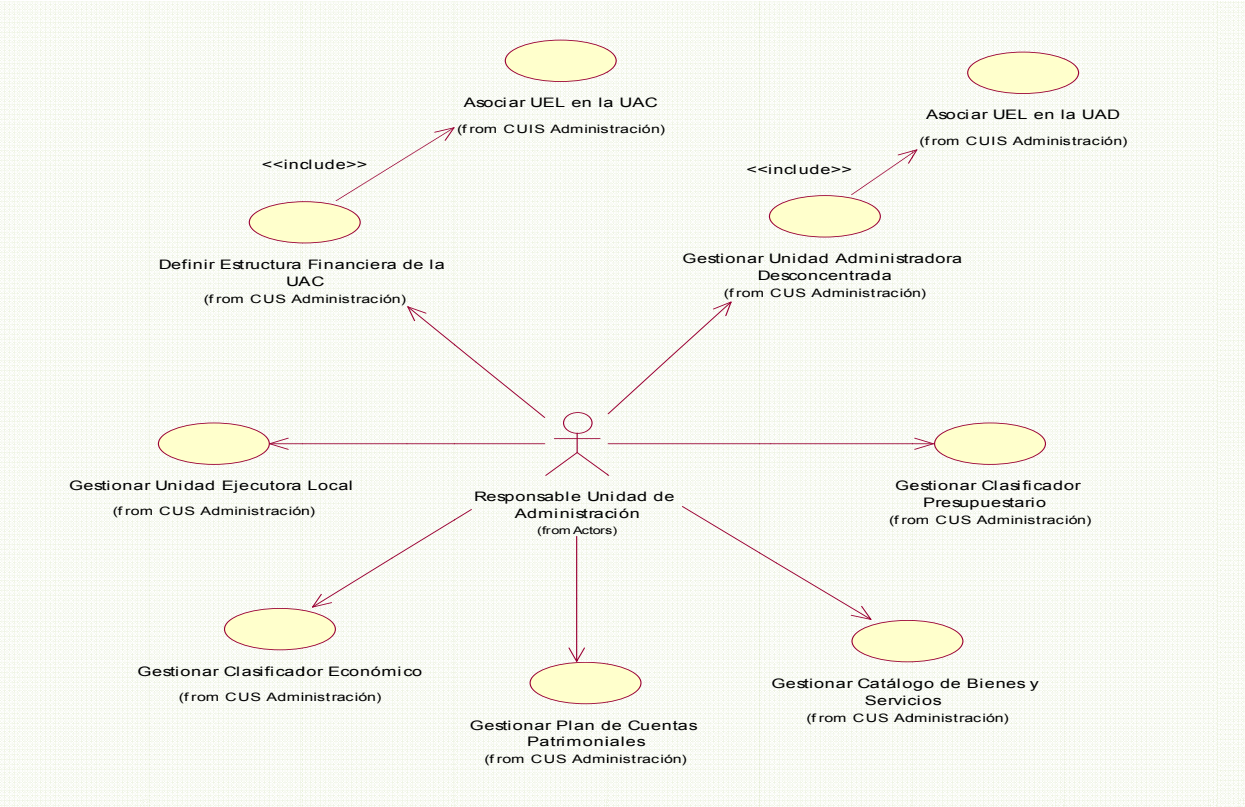


Figura 12 Vista de Casos de Uso módulo Administración.

Definir Estructura Financiera de la UAC: Consiste en gestionar en el sistema las propiedades de la UAC y su dependencia financiera.

Gestionar Unidad Administradora Desconcentrada: Permite crear, modificar y eliminar UADs en el sistema y relacionarlas con las oficinas.

Gestionar Unidad Ejecutora Local: Permite crear, modificar y eliminar UELs en el sistema y relacionarlas con las oficinas

Gestionar Clasificador Económico: Consiste en crear el Clasificador Económico con el cual se trabajará en el sistema.

Gestionar Plan de Cuentas Patrimoniales: Consiste en crear el Plan de Cuentas Patrimoniales con el cual se trabajará en el sistema

Gestionar Catálogo de Bienes y Servicios: Consiste en crear el Catálogo de Bienes y Servicios con sus propiedades.

Gestionar Clasificador Presupuestario: Permite crear el Clasificador Presupuestario vigente con el cual se trabajará en el sistema dando la posibilidad de gestionar dichas cuentas.

2.6.2 Presupuesto.

El Módulo de Presupuesto contempla todas las funcionalidades relacionadas con la Formulación, Modificación, Programación, Ejecución y Cierre del presupuesto para un año seleccionado a partir de la Estructura Financiera aprobada.

La Formulación se realiza por Proyectos y Acciones Centralizadas y cada uno de estos está compuesto a su vez por Acciones Específicas las cuales están detalladas por Unidades Ejecutoras Locales y Fuentes de Financiamiento.

Este proceso de Formulación comienza en las solicitudes de las Unidades Ejecutoras Locales de sus necesidades presupuestarias para la ejecución de sus metas y se consolida a nivel de Unidad Administradora Desconcentrada y Unidad Administradora Central.

El proceso de Modificación Presupuestaria se realiza una vez sancionada la ley del presupuesto por la existencia de Traspasos de Créditos Presupuestarios, Insubsistencia, Créditos Adicionales y Recorte de Créditos Presupuestarios al nivel de las Unidades Ejecutoras Locales.

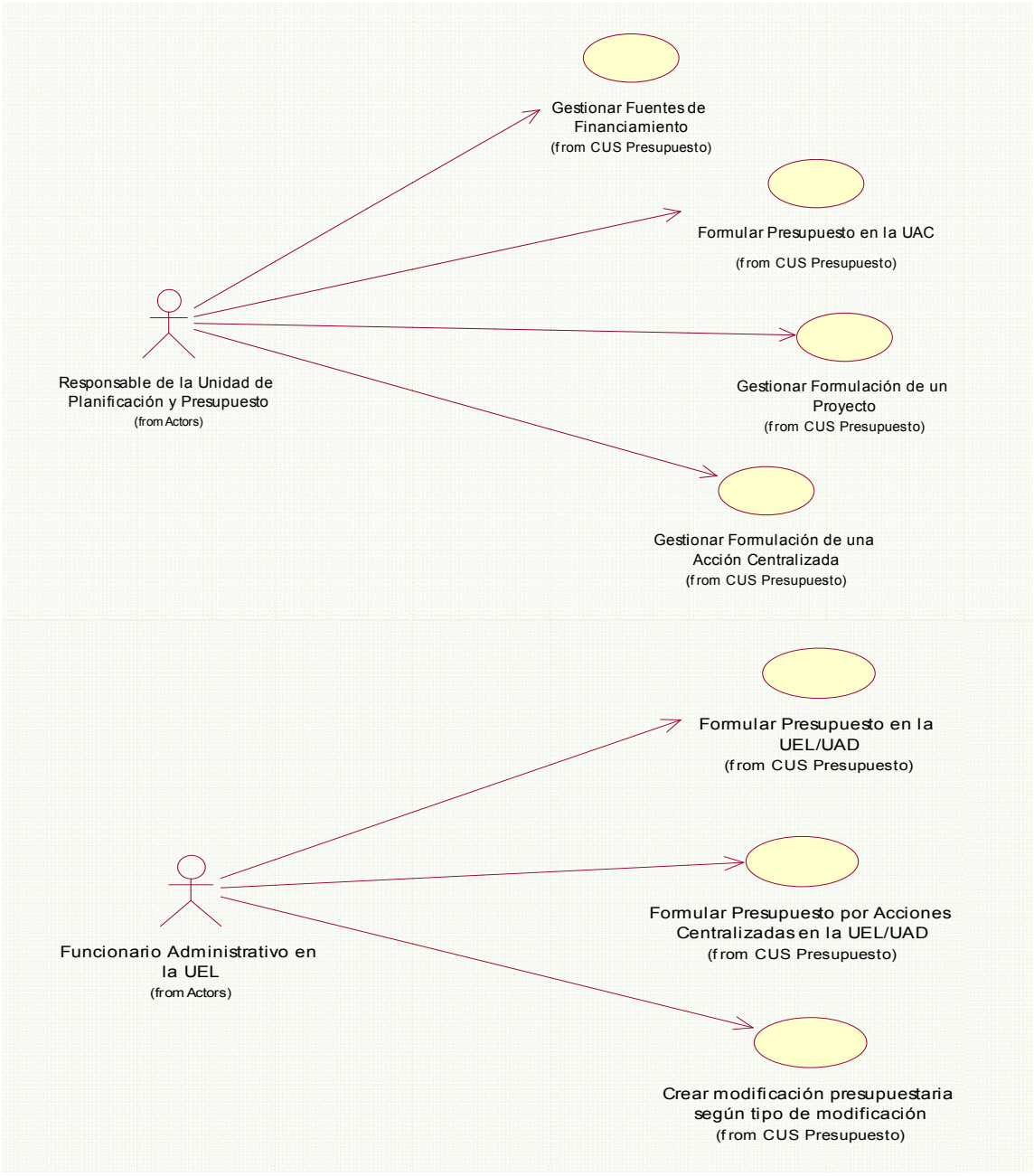
El Proceso de Programación se realiza por Trimestres, obteniéndose como resultado una tabla de programación mensual.

En la Ejecución y Cierre de Presupuesto se generan diferentes reportes donde se muestra la ejecución física y financiera de los proyectos/acciones centralizadas además de la ejecución financiera trimestral del presupuesto tanto del gasto como del ingreso.

Teniendo en cuenta la descripción del módulo y RN-AFDR-03 Documento de Caso de Uso del Sistema del módulo Presupuesto, resultan Casos de Usos significativos para la Arquitectura:

- Formular Presupuesto en la UAC.
- Gestionar Fuentes de Financiamiento.
- Gestionar Formulación de un Proyecto en la UAC.
- Gestionar Formulación de una Acción Centralizada en la UAC.
- Formular Presupuesto por Proyecto en la UEL/UAD.
- Formular Presupuesto por Acciones Centralizadas en la UEL/UAD.
- Crear Modificación Presupuestaria según tipo de modificación.
- Programar la Ejecución Financiera por Trimestre y Proyectos.
- Programar la Ejecución Financiera por Trimestre.
- Programar Ingresos y Desembolsos mensualmente.

La figura (ver figura 13) se muestran estos casos de uso del módulo en la vista de Casos de Uso de la Arquitectura.


Gestionar Fuentes de Financiamiento
(from CUS Presupuesto)
Formular Presupuesto en la UAC
(from CUS Presupuesto)
Responsable de la Unidad de Planificación y Presupuesto
(from Actors)
Gestionar Formulación de un Proyecto
(from CUS Presupuesto)
Gestionar Formulación de una Acción Centralizada
(from CUS Presupuesto)
Formular Presupuesto en la UEL/UAD
(from CUS Presupuesto)
Funcionario Administrativo en la UEL
(from Actors)
Formular Presupuesto por Acciones Centralizadas en la UEL/UAD
(from CUS Presupuesto)
Crear modificación presupuestaria según tipo de modificación
(from CUS Presupuesto)

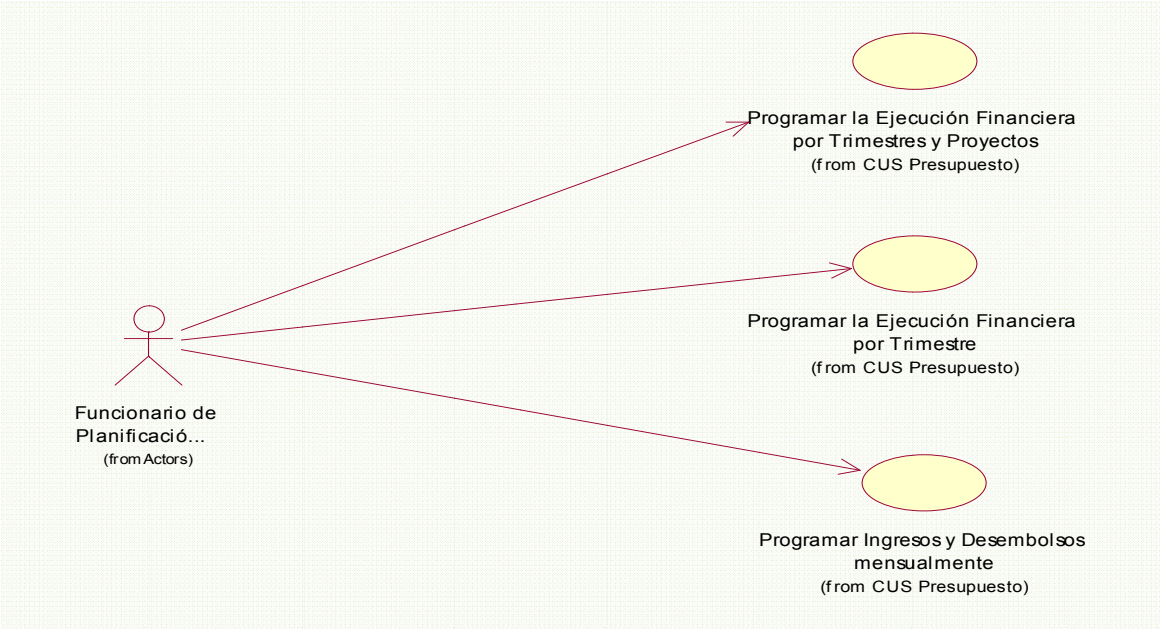


Figura 13 Vista de Casos de Uso módulo Presupuesto.

Formular Presupuesto en la UAC: Consiste en formular el anteproyecto de presupuesto a nivel central en la UAC.

Gestionar Fuentes de Financiamiento: Consiste en asociar las fuentes de financiamiento para crear el Ante-Proyecto de presupuesto en la UAC.

Gestionar Formulación de un Proyecto en la UAC: Consiste en configurar los datos necesarios para crear el presupuesto de un proyecto.

Gestionar Formulación de una Acción Centralizada en la UAC: Consiste en configurar los datos necesarios para crear el presupuesto de una Acción Centralizada.

Formular Presupuesto por Proyecto en la UEL/UAD: Consiste en la formulación del proyecto de presupuesto desde las UEL o las UAD.

Formular Presupuesto por Acciones Centralizadas en la UEL/UAD: Consiste en modificar las acciones centralizadas para el ante proyecto de presupuesto en la UEL/UAD.

Crear Modificación Presupuestaria según tipo de modificación: Consiste en crear la modificación presupuestaria dependiendo del tipo que esta sea.

Programar la Ejecución Financiera por Trimestre y Proyectos: Consiste en hacer una programación por trimestre del año que se va a presupuestar teniendo en cuenta las metas por proyectos.

Programar la Ejecución Financiera por Trimestre: Consiste en hacer una programación por trimestre del año que se va a presupuestar teniendo en cuenta la Ejecución Financiera por proyectos y por acciones centralizadas.

Programar Ingresos y Desembolsos mensualmente: Consiste en hacer una programación mensual de ingresos y desembolsos del año que se está presupuestando.

2.6.3 Recaudación.

El Módulo de Recaudación consiste en la recepción de los pagos que deben realizar los clientes de los Registros y Notarías por los trámites solicitados, en las oficinas de los bancos recaudadores correspondientes, mediante una Planilla Única Bancaria, diseñada de acuerdo a las necesidades de las oficinas.

Permite registrar el estatus de las Planillas Únicas Bancarias desde que son emitidas en las oficinas hasta que le dan continuidad al trámite después de pagada, así como el proceso de Reintegros y Recaudación por abandono de las mismas. Permite además evaluar la cantidad de Planillas Únicas Bancarias exoneradas, exentas y caducadas.

Registra y controla todos los Ingresos que se obtienen por Derechos de Registro Público, Mercantil, Principal y Notariales, Habilitación, Traslado, Inserción Anticipada del documento y Transporte.

Realiza las coordinaciones con los bancos recaudadores cuando exista algún problema de conciliación de las Planillas pagadas para la búsqueda de posibles soluciones.

Teniendo en cuenta la descripción y RN-AFMCU-01 Documento de Caso de Uso del Sistema del módulo Recaudación, resultan Casos de Usos significativos para la Arquitectura:

- Recibir PUB emitidas.
- Notificar PUB Caducada.
- Recibir PUB en Trámite.
- Recibir PUB para ser tramitada por autorización.
- Recibir confirmación de las PUB pagadas en banco.
- Recibir Resumen Diario de la PUB pagadas.
- Capturar Reintegros.
- Recaudar PUB por abandono.

La figura (ver figura 14) se muestran estos casos de uso del módulo de Recaudación en la vista de Casos de Uso de la Arquitectura.

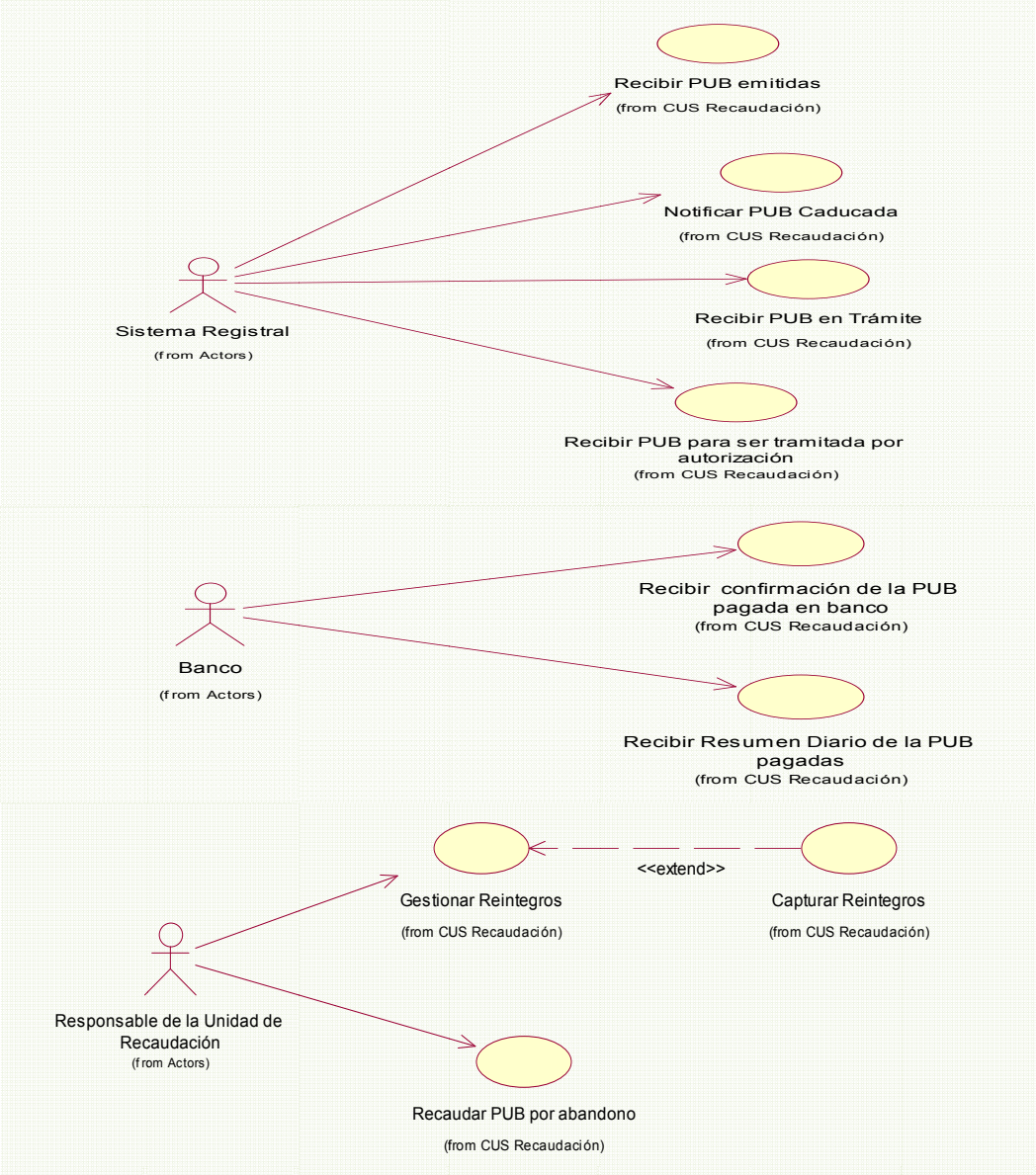


Figura 14 Vista de Casos de Uso módulo Recaudación.

Recibir PUB emitidas: Este caso de uso es iniciado por el actor sistema registral cuando envía la información de las PUB emitidas para ser registradas en la UR, estableciéndole un estado a las mismas.

Notificar PUB Caducada: Este caso de uso es iniciado por el actor sistema registral que le informa a la Unidad de Recaudación las PUB que han sido caducadas por no ser pagadas en el tiempo previsto.

Recibir PUB en Trámite: Cuando los usuarios presentan las PUB pagadas para darle continuidad al trámite se registra el pago de la misma y se envían a la Unidad de Recaudación.

Recibir PUB para ser tramitada por autorización: Consiste en registrar la PUB que es presentada para continuar el trámite con una previa autorización por parte de la autoridad competente.

Recibir confirmación de las PUB pagadas en banco: Este caso de uso se inicia cuando el Banco envía el recibo de la confirmación del pago de la PUB en el banco y es recibida en la UR donde se verifica que el Nro. Único de PUB y el Monto Cancelado de la notificación se corresponda con la PUB emitida, en caso contrario se registra la diferencia existente.

Recibir Resumen Diario de la PUB pagadas: Este caso de uso es iniciado por el Banco una vez que emite el resumen diario de las PUB pagadas durante el día a la UR, en donde se concilia con las recibidas durante el día para ver si hay diferencias entre las mismas, en caso de encontrarse alguna se solicita al Banco un resumen detallado para detectar donde existe la diferencia, realizando la comparación y registrando las diferencias encontradas.

Capturar Reintegros: Consiste en registrar los reintegros presentadas, una vez realizado el pago de la Planilla Única Bancaria.

Recaudar PUB por abandono: Consiste en mostrar las PUB por oficina que están pagadas y ha culminado el tiempo de realizar el trámite y de reclamar el monto depositado así como el reporte por oficina de las mismas.

2.6.4 Contabilidad.

En su conjunto ofrece la posibilidad de tener una Contabilidad actualizada, veraz y segura, que le permita obtener la información necesaria para lograr una administración más eficiente, si son aplicadas todas las funciones que en el mismo se han concebido.

En este módulo se muestra el registro de comprobantes, a partir de los cuales se controlan todas las operaciones contables generadas por el resto de los subsistemas y creadas en este, estos comprobantes son de tipo Patrimonial o Presupuestario. Otro de los procesos existentes es el cierre de las cuentas nominales el cual genera un comprobante, el resultado del cierre se asienta en una cuenta contable seleccionada.

Teniendo en cuenta la descripción y RN-AFMCU-04 Documento de Caso de Uso del Sistema del módulo Contabilidad, resultan Casos de Usos significativos para la Arquitectura:

- Gestionar Registro de Comprobantes Contables.
- Cerrar Cuentas Nominales.

La figura (ver figura 15) se muestran estos casos de uso del módulo en la vista de Casos de Uso de la Arquitectura.

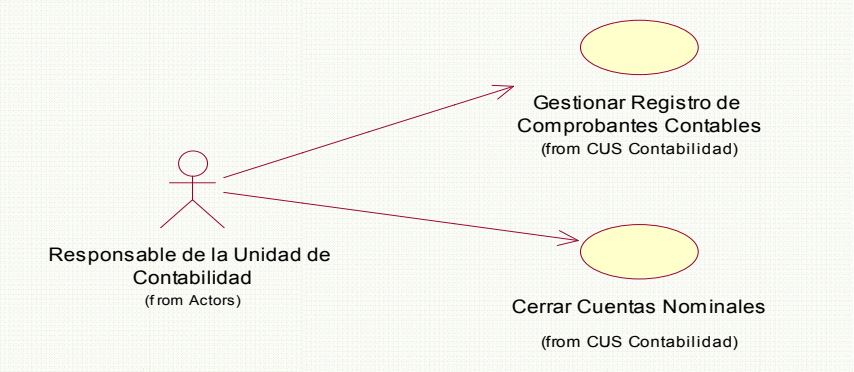


Figura 15 Vista de Casos de Uso módulo Contabilidad.

Gestionar Registro de Comprobantes Contables: Este caso de uso consiste en gestionar los comprobantes contables, visualizarlos en un registro tanto los manuales como los automatizados.

Cerrar Cuentas Nominales: Este caso de uso consiste en generar un comprobante para llevar el saldo de las cuentas nominales a cero.

2.7 Vista Lógica.

RUP propone para la Vista Lógica una representación del sistema a nivel de paquetes, subsistemas, clases significativas para la arquitectura, así como las principales relaciones que se establecen entre ellos.

Al igual que en la anterior, la vista lógica se corresponde a la primera iteración de la Arquitectura como se puede ver en la figura (ver figura 16).

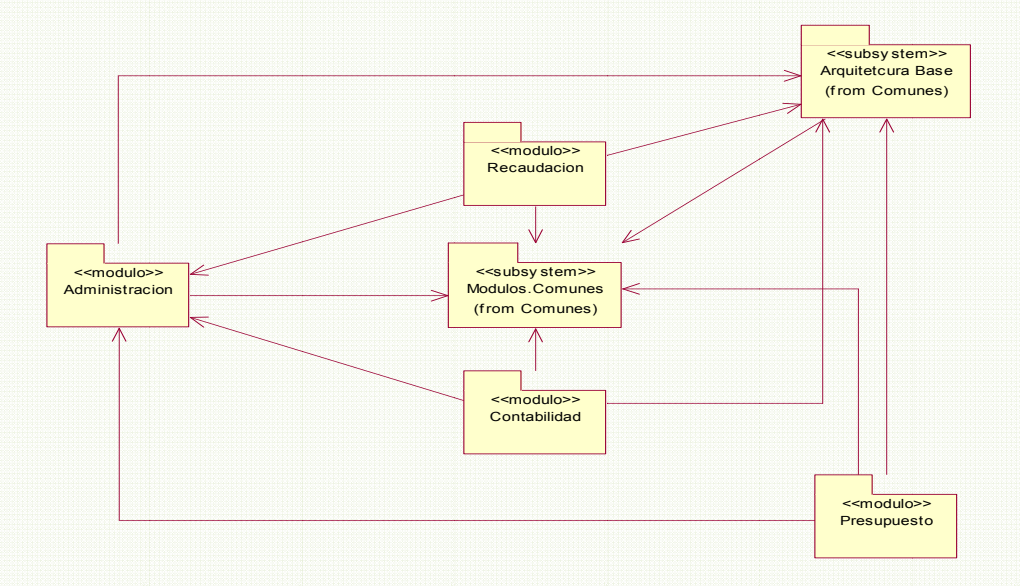


Figura16 Módulos correspondientes a la primera iteración de la Arquitectura del sistema.

En la figura anterior se agrupan los módulos en los paquetes donde se van a encapsular sus correspondientes funcionalidades. Así mismo se puede apreciar la presencia del subsistema Común y "ArquitecturaBase" que como se ha visto en secciones anteriores se refieren a algunas funcionalidades que dan soporte a la Arquitectura y como tal interactuarán con la totalidad de los paquetes y componentes que componen la solución.

Puesto que ya se tiene una visión general de esta iteración se hace necesario entrar en las particularidades de cada uno de estos paquetes. La figura (ver figura 17) muestra la distribución general correspondiente al modelo multicapas que se propone para cada uno de los módulos, con la excepción del Común cuya distribución se puede observar en la figura (ver figura 18).

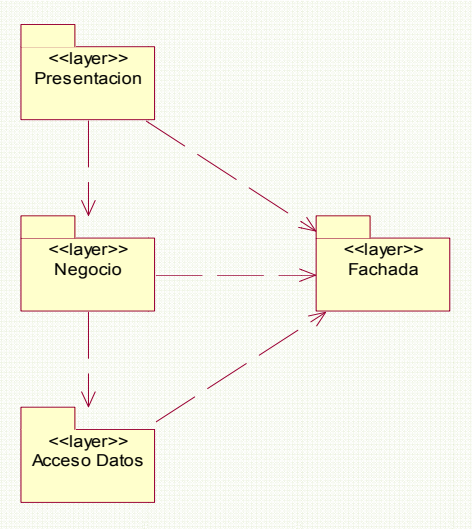


Figura 17 Representación del modelo multicapas módulos.

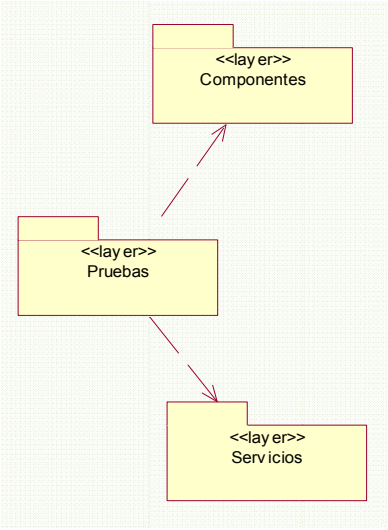


Figura 18 Representación del modelo multicapas módulo Común.

La estructura de paquetes de cada una de las capas que se aprecian en la figura (ver figura 17) se resume de la siguiente manera:

- **Presentación**:

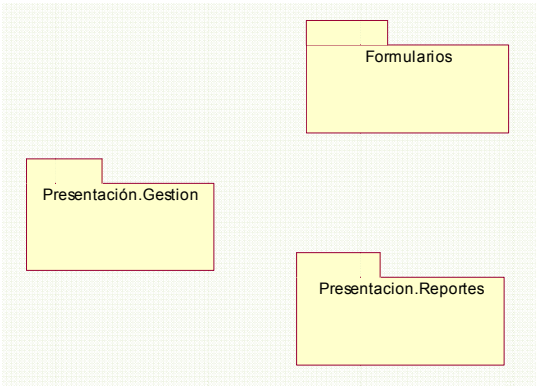


Figura 19 Estructura de la Capa de Presentación.

- El paquete correspondiente a los Formularios contiene las Interfaces de Usuario del módulo específico.
- El paquete correspondiente a la Gestión contiene las Acciones asociadas a cada una de las Interfaces de Usuario.
- El paquete correspondiente a los Reportes contiene los ficheros (.rpt) del Crystal Report para mostrar las estadísticas del sistema.

➢ **Negocio:**

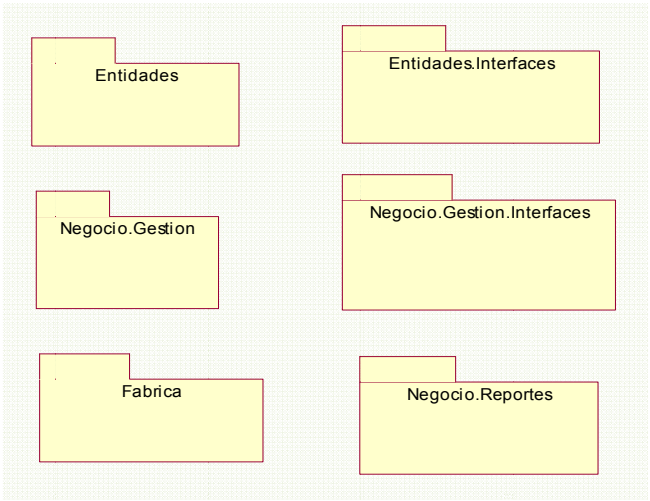


Figura 20 Estructura de la Capa de Negocio.

- El paquete correspondiente a las Entidades contiene las clases objeto-valor (clases persistentes).
- El paquete Entidades.Interfaces se corresponde con las interfaces que contienen las propiedades o los métodos que nos ofrecen las entidades.
- El paquete correspondiente a Negocio.Gestion contiene los Gestores de la aplicación, los cuales van a resolver la mayoría de las funcionalidades según los requisitos funcionales haciendo uso de las Entidades.

- El paquete de Gestion.Interfaces se corresponde con las interfaces que contienen los métodos/funcionalidades que ofrecen los Gestores.
- El paquete Fábrica contiene las funcionalidades que ofrece la Capa de Negocio a la Capa de Presentación a partir de una estructura de clases que en su conjunto dan vida el patrón Abstract Factory.
- Negocio.Reportes contiene las clases correspondientes a los datos que se quieren visualizar en los reportes.

➢ **Acceso a Datos:**

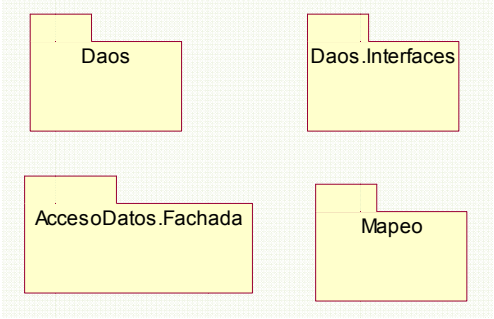


Figura 21 Estructura de la Capa de Acceso a Datos.

- El paquete correspondiente a los Daos contiene las clases que se encargan de las funcionalidades relacionadas con el acceso a datos, mayormente a través de procedenticos almacenados.
- Daos.Interfaces son las interfaces correspondientes a los métodos/funcionalidades que brindan los Daos.
- AccesoDatos.Fachada es un paquete que contiene una estructura específica de clases apoyada en Spring.net para que la Capa de Negocio interactué con los Daos.
- Mapeo contiene los ficheros XML de mapeo que utiliza NHibernate para la persistencia.

➢ **Fachada:**

Esta capa va a ser dinámica y por tanto puede presentar tantos paquetes como las funcionalidades que se quiera incorporar al módulo donde este implantada.

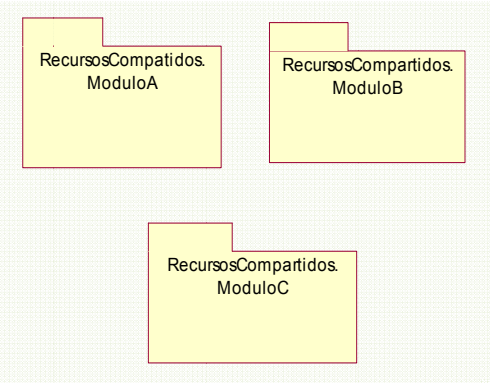


Figura 22 Estructura de la Capa de Fachada.

- Los paquetes correspondientes a los RecursosCompartidos contienen las funcionalidades que se quieren incorporar al módulo y toda la infraestructura para garantizarlas apoyándose fundamentalmente en el framework Spring.net.

En el caso del módulo Común, la estructura de paquetes de cada una de sus capas como se aprecian en la figura (ver figura 18) se resume de la siguiente manera:

- **Componentes:**

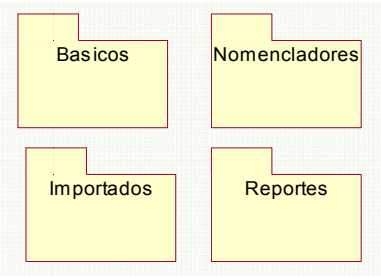


Figura 23 Estructura de la Capa de Componentes.

- El paquete Básicos contiene una librería de componentes que resultan básicos para el desarrollo del sistema fundamentalmente asociados a la Capa de Presentación.
- Nomencladores se refiere a los componentes nomencladores del sistema, es decir que trabajan con este tipo de datos que se encuentran en la Base de Datos.
- El paquete Importados contiene los componentes que se heredan de otras aplicaciones para ser reutilizados en la solución donde pueden ser modificados y actualizados por el equipo de desarrollo de acuerdo a las particularidades y necesidades de cada uno.
- Reportes contiene los componentes asociados al trabajo con las estadísticas del sistema.

➢ **Pruebas:**

Teniendo en cuenta que el módulo Común representa una fábrica de componentes y servicios para la Arquitectura se hace necesaria la presencia de una capa de Pruebas que sirva para validar y probar todos estos elementos.

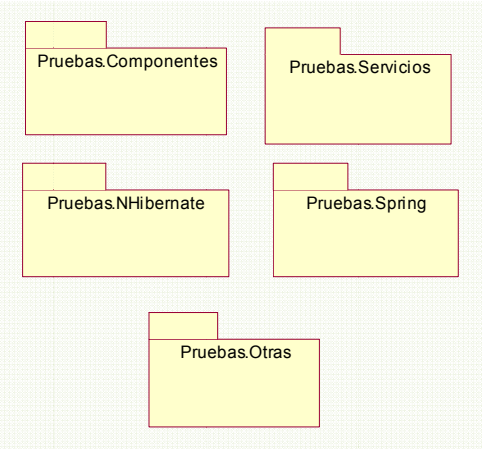


Figura 24 Estructura de la Capa de Pruebas.

- Pruebas.Componentes contiene toda la infraestructura para las pruebas que se van a realizar sobre la capa de componentes.

- Pruebas.Servicios contiene toda la infraestructura para las pruebas que se van a realizar sobre la capa de Servicios con la excepción de los framework open-source.
- Se crea un paquete Pruebas.NHibernate independiente para la pruebas sobre el framework open-source NHibernate puesto que el mismo requiere de una infraestructura más específica para verificar las modificaciones que se le realicen.
- Se crea un paquete Pruebas.Spring independiente para la pruebas sobre el framework open-source Spring.net puesto que este requiere también de una infraestructura más específica para verificar las modificaciones que se le puedan realizar.
- Pruebas.Otras contiene cualquier otro tipo de pruebas que se deseen realizar. Aquí se incluyen pruebas totales, pruebas que involucren otros elementos que no se encuentren en el módulo etc.

➢ **Servicios:**

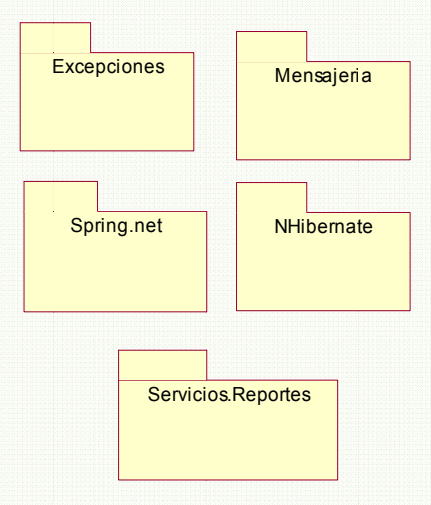


Figura 25 Estructura de la Capa de Servicios.

- Mensajería contiene la implementación de todo el servicio de mensajería del sistema, el cual se va a hacer centralizado en un fichero XML.

- Excepciones contiene la implementación de todo el servicio que garantiza un sencillo y adecuado trabajo con las excepciones del sistema, las cuales van a estar centralizadas en un fichero XML.
- Servicios.Reportes incluye la implementación de los servicios que ayuden al trabajo con los reportes.
- NHibernate este paquete contiene el código del framework NHibernate para realizarle modificaciones en caso de ser necesario.
- Spring.net este paquete contiene el código del framework Spring.net para realizarle modificaciones en caso de ser necesario

Para alcanzar un mayor grado de comprensión, a continuación se muestra en la figura (ver figura 26) la estructura que se ha ido viendo a nivel de clases y paquetes:

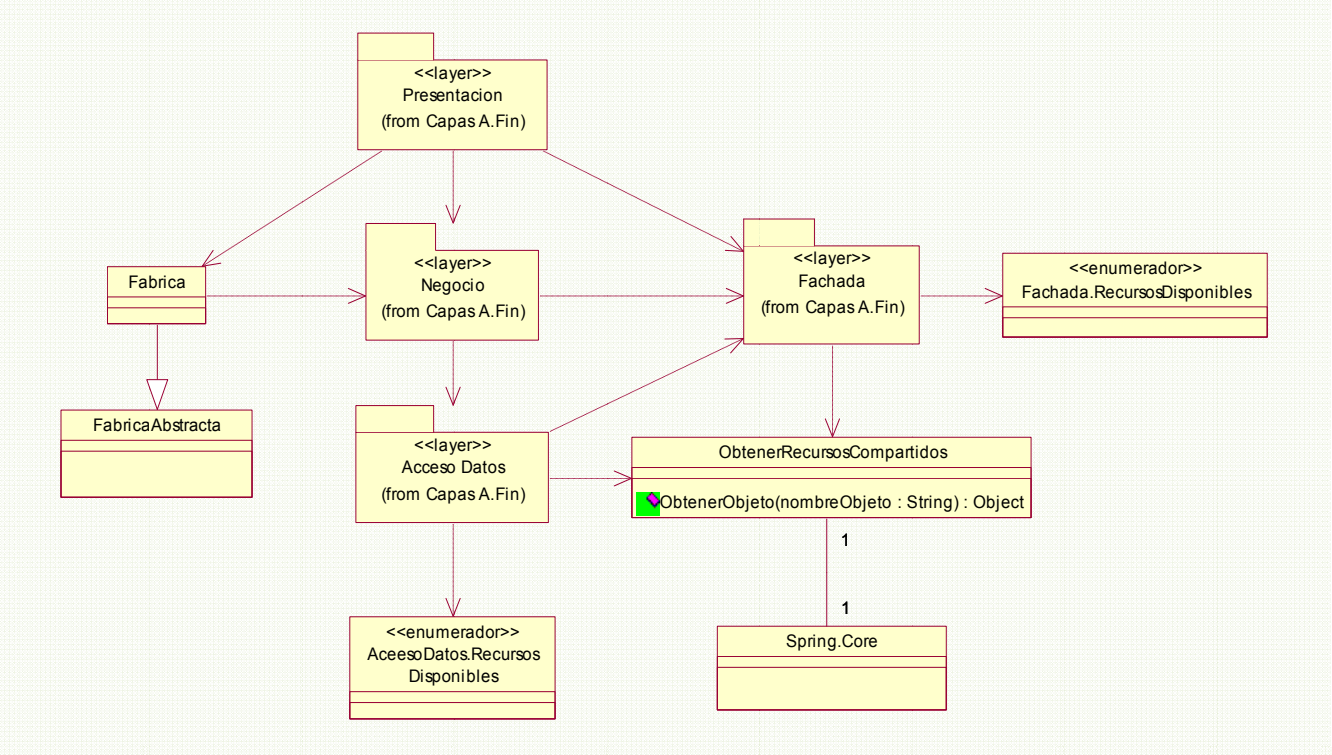


Figura 26 Estructura de clases para la Arquitectura planteada.

En la figura anterior se observa la presencia del patrón Abstract Factory que funciona de puente entre la Capa de Presentación y la de Negocio agrupando las clases de negocio en familias según las funcionalidades que representen. Por otra parte se puede ver la presencia de la Fachada como mecanismo de conexión con las funcionalidades que se encuentran en otros módulos según el enumerador (**enum**) que las relaciona y la clase que nos brinda el modulo Común (ObtenerRecursosCompartidos) que resuelve esta situación haciendo uso de Spring.net. Igualmente se resuelve la interacción de la Capa de Negocio con la de Acceso a Datos pero esta vez haciendo uso de la Fachada interna que presenta el Acceso a Datos.

2.8 Vista Implementación.

La vista de implementación del sistema contiene los componentes que se generan durante la compilación de cada uno de los paquetes y estructuras que se han ido viendo así como las dependencias que se establecen entre cada uno de ellos. Según RUP el propósito de esta vista es capturar las decisiones arquitectónicas para la implementación.

Puesto que la organización de componentes a partir de estas dependencias, ya sean de compilación, de inclusión es la misma para la totalidad del sistema, solamente se va a reflejar en esta vista aquellos módulos que introduzcan diferencias en los diagramas de componentes y que aporten mayor información al equipo de desarrollo.

Por tanto, la vista de implementación para la primera iteración se resume a los módulos de Administración y Presupuesto con las particularidades y especificidades de cada uno de ellos, además se incluye aunque en un menor grado el módulo “ArquitecturaBase”. No obstante para ganar en claridad y comprensión se hace alusión en caso de ser necesario al resto de los subsistemas.

El caso del módulo Administración es el más sencillo puesto que es el que menos dependencias genera en esta iteración. Como se puede ver en la figura (ver figura 27) existen dependencias con ficheros de configuración que necesita el framework Común, con otros que requiere NHibernate, Spring.net y finalmente los que pueda incorporar la solución relacionados fundamentalmente con el módulo Común. El

resto de las relaciones se obtienen de la Arquitectura vertical que deriva en los paquetes y finalmente en los componentes.

Por otra parte como se puede ver en la figura (ver figura 28) el módulo Presupuesto mantiene la misma estructura que el de Administración, solamente incorpora la Fachada puesto que requiere de algunas funcionalidades que en este caso se van a encontrar en Administración. Esta idea se mantiene para el caso de Contabilidad y Recaudación los cuales como se ha visto interactúan mayormente con Administración.

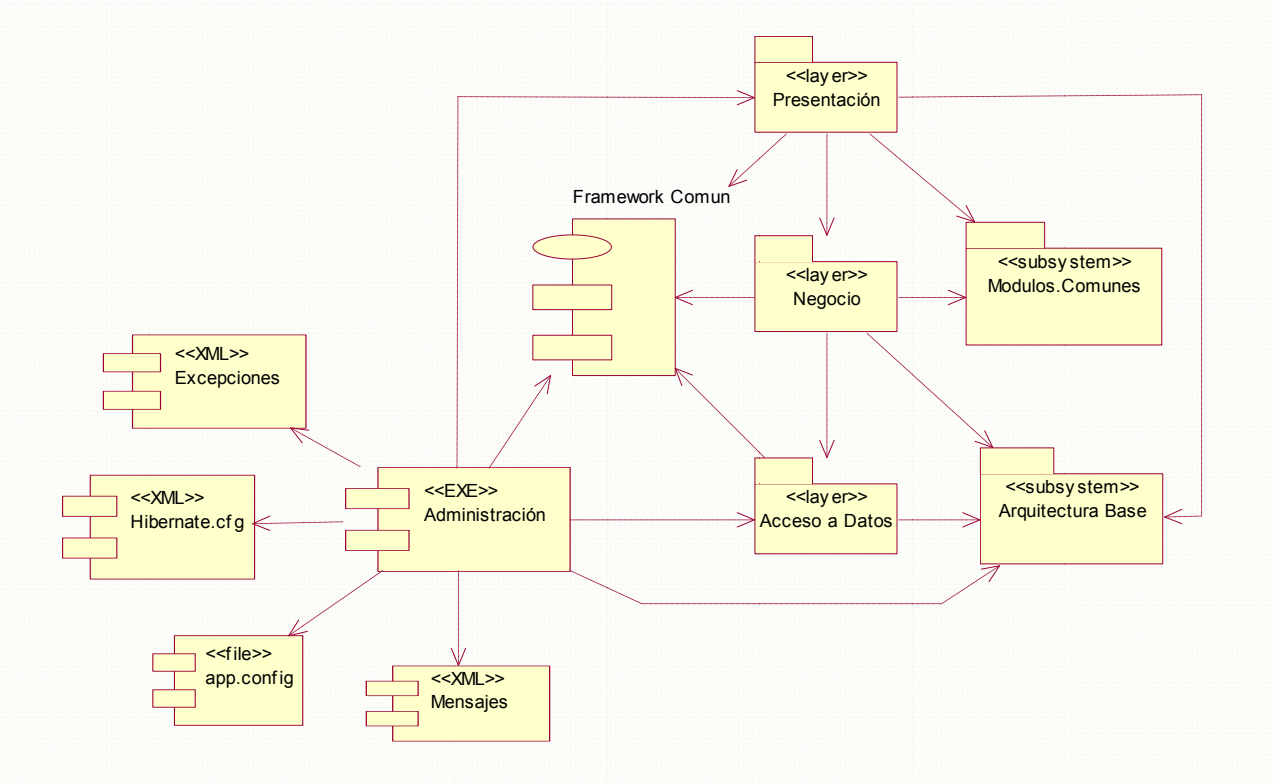


Figura 27 Representación de la Vista de Implementación módulo Administración.

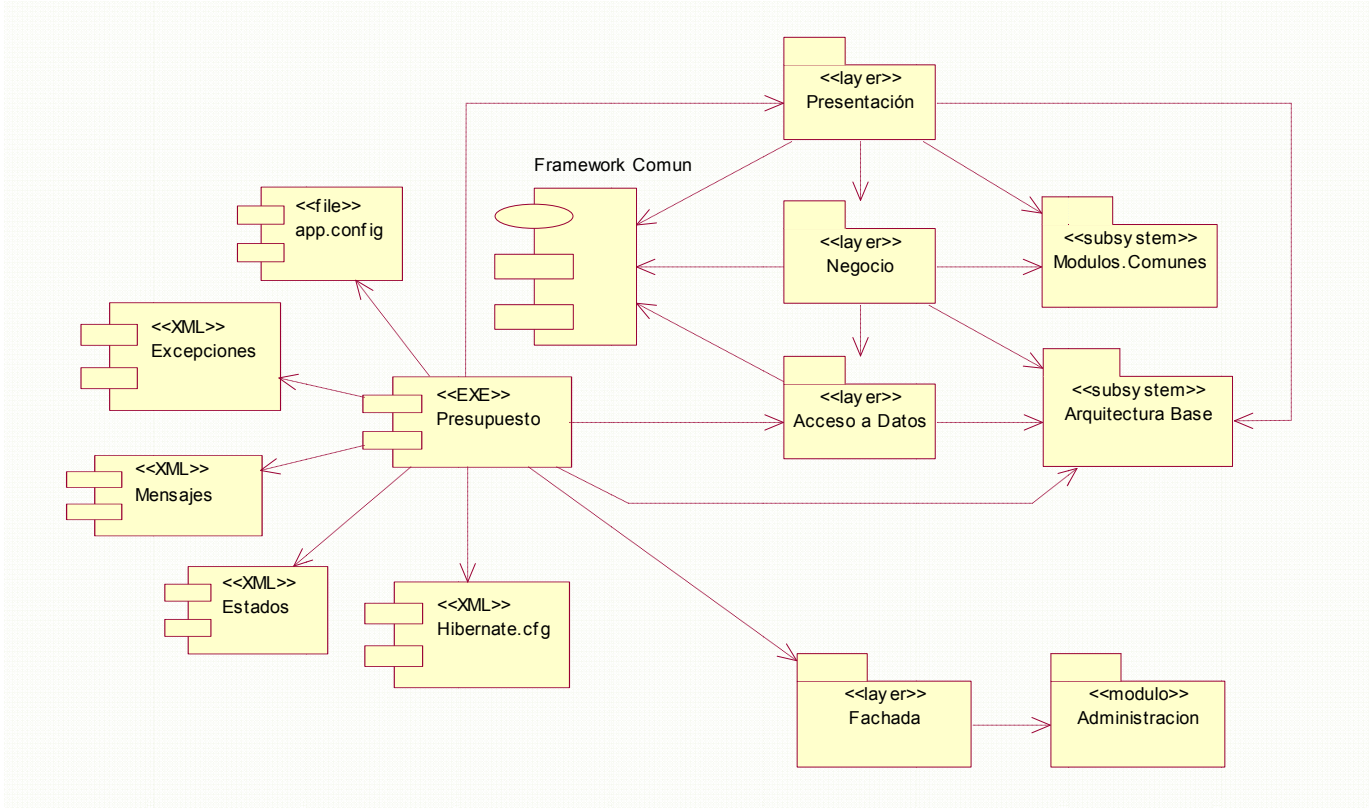


Figura 28 Representación de la Vista de Implementación módulo Presupuesto.

El caso del módulo Común es más específico. Su estructura como se ha visto en secciones anteriores es totalmente diferente del resto puesto que responde netamente a la Arquitectura del sistema. Sin embargo no genera mayores dependencias y la implementación que se realice sobre él no afecta el desempeño del sistema. Por tanto no cumple objetivo generar otra representación, en este documento, que no sea la que ya se obtuvo en la vista Lógica.

El módulo de “ArquitecturaBase” continúa con la misma línea de la solución como se puede ver en la figura (ver figura 29). No obstante solamente contiene aquellas funcionalidades concernientes al negocio que responden a la totalidad del sistema y lógicamente la infraestructura de integración del mismo. La Capa de Presentación no está presente puesto que la lógica relacionada con la interfaz de usuario está contemplada en los módulos que se van incorporando y en el framework (framework Común).

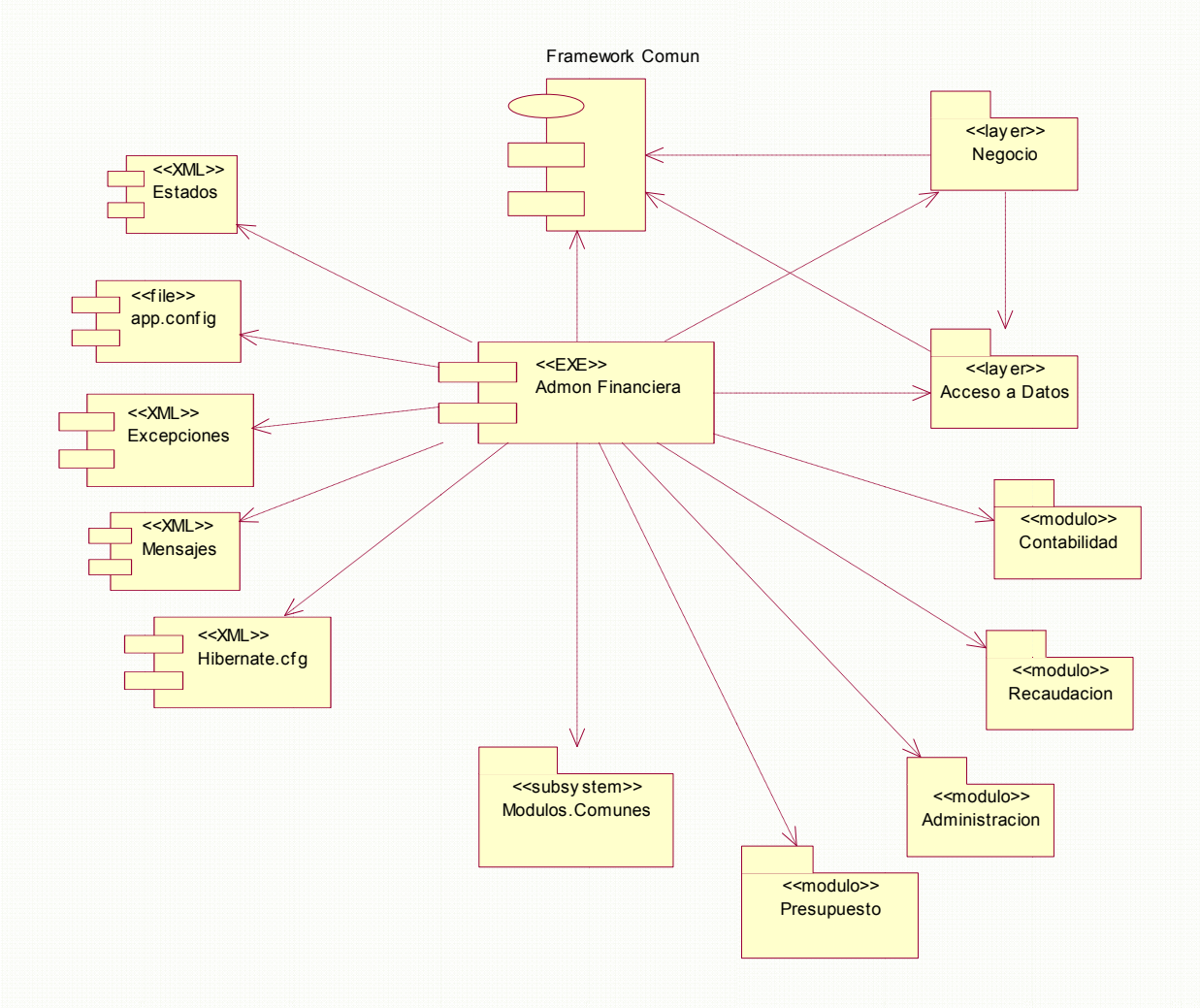


Figura 29 Representación de la Vista de Implementación módulo Arquitectura Base.

Con relación a los diagramas que se mostraron con anterioridad, se aprecian una serie de elementos que encapsulan las funcionalidades que a su vez se agrupan en otros componentes. Para ganar en claridad a continuación se visualizan cada uno de estos elementos con sus respectivos diagramas de componentes.

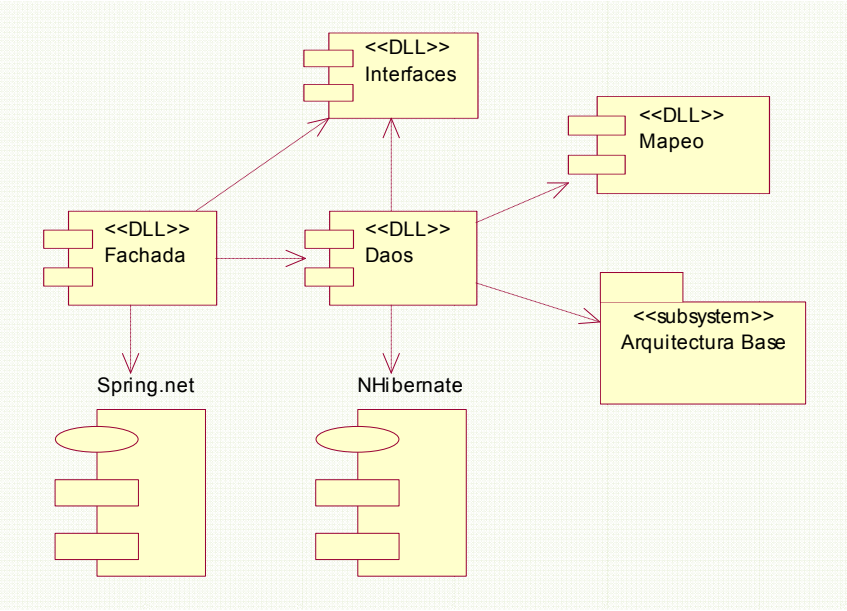


Figura 30 Vista de Implementación Capa de Acceso a Datos.

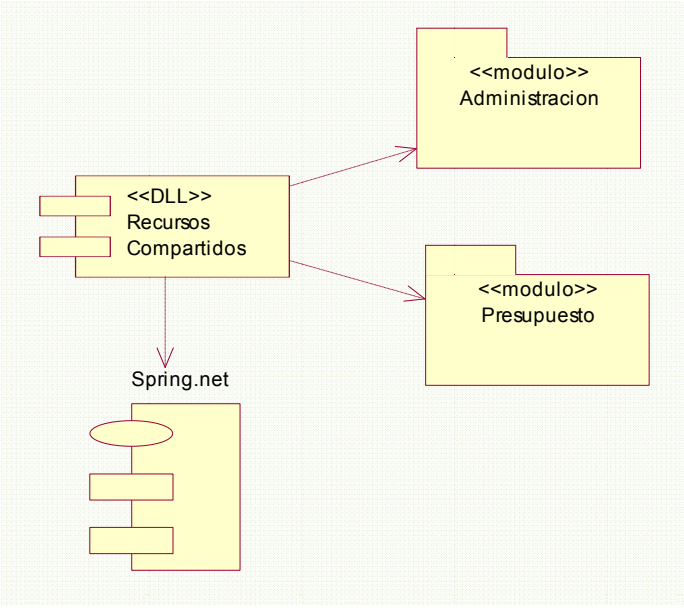


Figura 31 Vista de Implementación Capa de Fachada para un módulo que interactúe con Presupuesto y Administración.

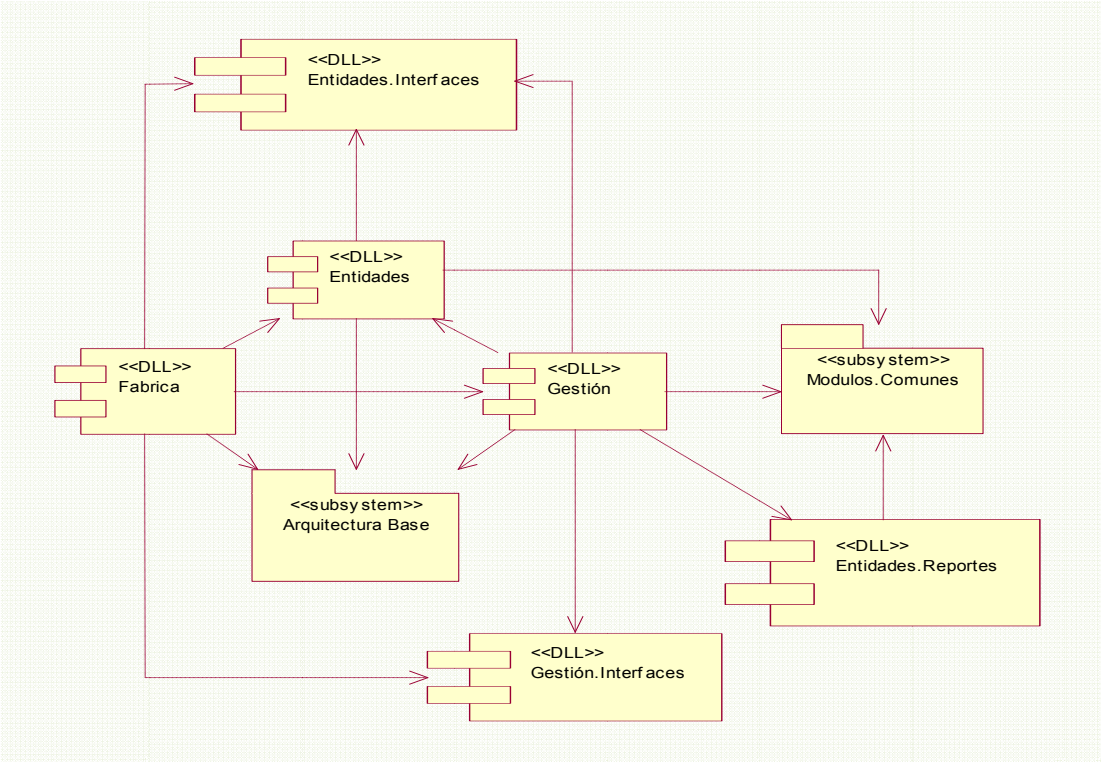


Figura 32 Vista de Implementación Capa de Negocio.

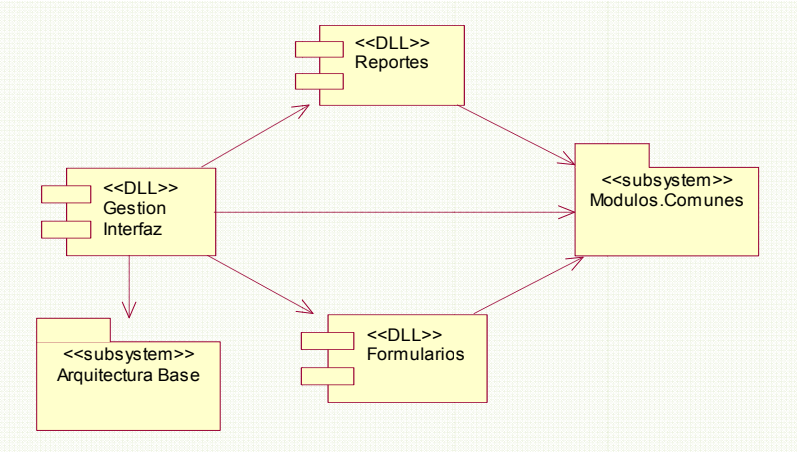


Figura 33 Vista de Implementación Capa de Presentación.

2.9 Vista de Despliegue.

Esta vista aborda la descripción de los nodos físicos para la mayoría de las configuraciones tanto para usuarios finales como para desarrolladores y probadores.

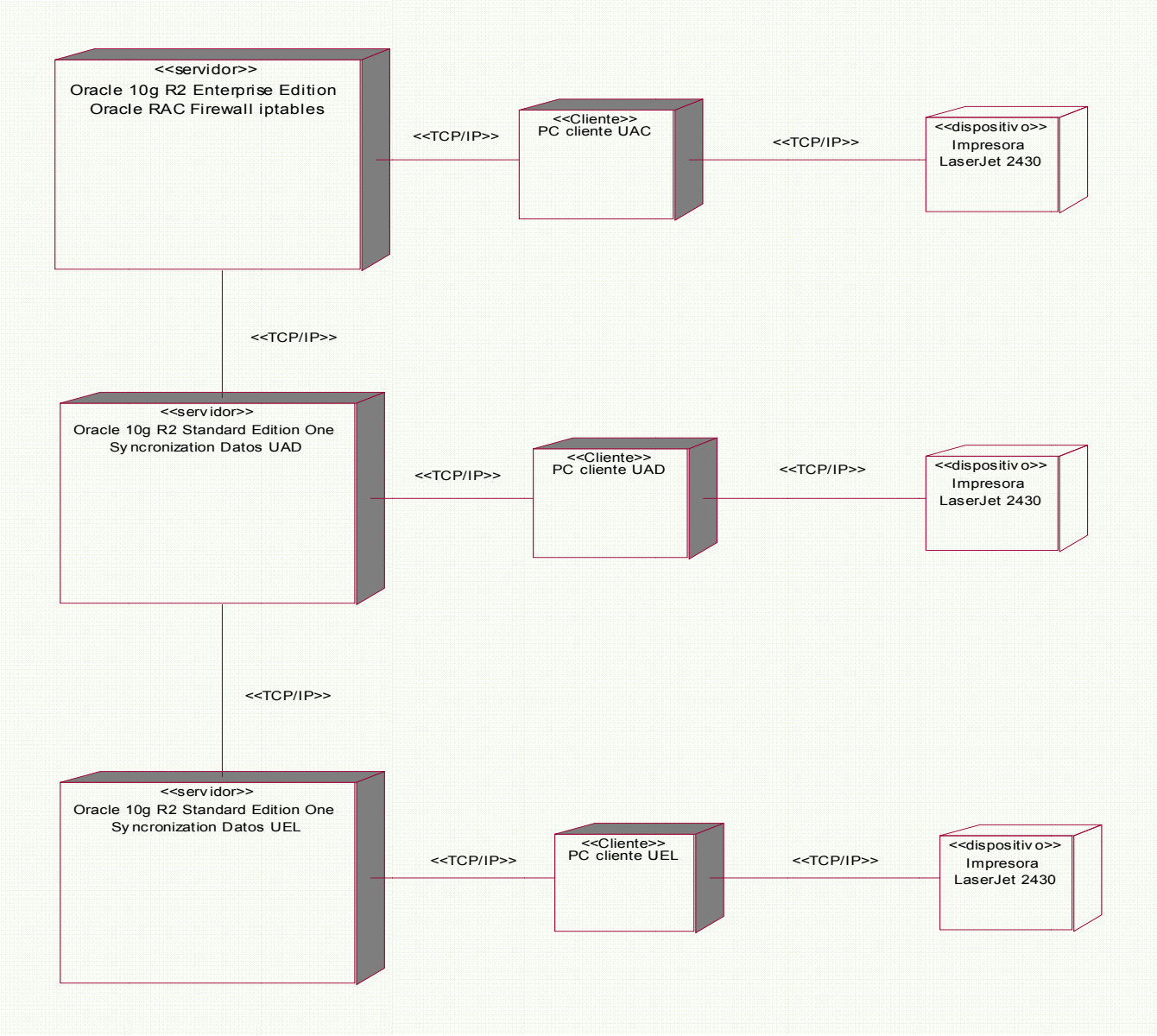


Figura 34 Vista de Despliegue del sistema Administración Financiera.

A continuación se describen los diferentes nodos y dispositivos que contiene la figura (ver figura 34):

Impresora: Este dispositivo garantiza las funcionalidades de impresión de reportes y documentos que genera la aplicación dependiendo de las funcionalidades de la máquina cliente que la utilice a través de la red.

PC Cliente: Este nodo se corresponde con la máquina donde se encuentra instalado el sistema de Administración Financiera con las funcionalidades que se correspondan con su localización en la estructura financiera para ese año, ya sea a nivel de UEL, UAD o UAC. En cada una de estas maquinas se va a encontrar instalado el Sistema Operativo Windows XP SP2 en Español.

Servidor Oracle 10g R2 Standard Edition UEL: Este es el servidor de base de datos que se encuentra en cada UEL. Está montado sobre el Sistema Operativo Windows Server 2003 y es el nodo al que la máquina cliente encuesta y solicita información. En caso de no ser satisfecha la solicitud, la misma se eleva al servidor correspondiente a la UAD a la cual está adscrita dicha UEL.

Servidor Oracle 10g R2 Standard Edition UAD: Este es el servidor de base de datos que se encuentra en cada UAD. Está montado sobre el Sistema Operativo Windows Server 2003 y es el nodo al que la máquina cliente o servidora de la UEL encuesta y solicita información. En caso de no ser satisfecha la solicitud, la misma se eleva al servidor que se encuentra en el Centro de Datos que representa a la UAC.

Servidor Oracle 10g R2 Enterprise Edition Oracle RAC Firewall iptables: Este es el servidor de base de datos que se encuentra en el Centro de Datos. Está montado sobre el Sistema Operativo RedHat AS v4.0 y es el nodo al que la máquina cliente o servidora de la UAD encuesta y solicita información. En este nivel es a donde van a llegar todas las solicitudes y por tanto apoyado en el mecanismo de réplica de datos aquí se encuentra registrada la totalidad de la información del sistema.

De manera general, para las UEL se utiliza una tecnología de red inalámbrica y entre ellas una red WAN con enlaces de alta velocidad y topología Estrella.

2.10 Conclusiones

La Arquitectura propuesta como se ha visto en las diferentes secciones de este capítulo tiene su base en l desarrollo de componentes y conectores. Cada uno de los elementos que incorpora se corresponde a uno de estos conceptos dependiendo del nivel de abstracción en el que nos encontremos.

Al culminar cada una de las iteraciones se obtienen actualizaciones de este documento que proporcionen mayor comprensión a los desarrolladores e interesados. Lo anterior incluye cuatro de las 5 vistas que propone la metodología utilizada y algunos conceptos/definiciones que puedan surgir en cada ciclo.

Igualmente para dar solución al objetivo general y los específicos en cada iteración se adoptan una serie de decisiones las cuales tiene su base en las restricciones y estrategias que presenta la arquitectura. Por tanto se hace necesario validar en primer lugar que los resultados sean los esperados y en segundo lugar que se respeten todas estas limitantes.

Capítulo 3

3. Resultados de la Arquitectura propuesta.

3.1 Introducción.

Los artefactos de arquitectura son decisivos en la calidad del software que se desarrolla. Su evaluación permite mitigar los diferentes riesgos asociados al desarrollo del software, mejorar la visión de los procesos críticos y validar las decisiones de diseño que se tomaron. Al ir evaluando los resultados obtenidos a partir de los objetivos que se propusieron se tiene la posibilidad de tomar acciones tempranas y valorar los atributos no funcionales (disponibilidad, desempeño, seguridad, interoperabilidad, mantenibilidad, etc.) sin esperar a que el software se construya.

Este capítulo constituye la rectificación y confirmación de todos los elementos que se han relacionado en la Línea Base propuesta. Para ello, se hace un análisis de los resultados obtenidos a partir de los escenarios arquitectónicos que se pueden encontrar y colapsen o provoquen un mal funcionamiento de la solución en un momento determinado.

3.2 Objetivos y Cualidades.

Los objetivos que se quieren alcanzar en la Arquitectura del sistema de Administración Financiera se refieren fundamentalmente a los atributos de calidad (Flexibilidad, Mantenibilidad, Escalabilidad, Integrabilidad, Interoperabilidad). Algunas de estas cualidades se refieren al diseño, sin embargo tienen su repercusión en la arquitectura y contribuyen a garantizar los objetivos.

En esta dirección es que se desarrolla la totalidad de la solución. Es decir estas cualidades marcan una línea entorno a la cual van a girar los procesos de desarrollo. A continuación se describe el significado de algunos de ellos para la Arquitectura.

Flexibilidad:

Se refiere a la posibilidad de variar el programa para adaptarse a requerimientos más amplios. Detrás de esta cualidad está implícita la idea de diseño para el cambio, para cumplir las metas de re-uso de código y arquitectura.

Permite personalizar el desenvolvimiento de nuevos componentes específicamente para atender las necesidades del sistema. Una arquitectura flexible permite poder modificar cualquiera de los demás componentes de la misma sin necesidad de cambios en las aplicaciones, ni en su interface con los demás elementos.

La flexibilidad en las aplicaciones, está dada por el concepto de "parametrización", donde se almacenan valores de configuración para gran cantidad de variables. En caso de requerir cambios en estas variables, basta con cambiarlas y estos se reflejan en el resto de la aplicación.

Reparabilidad y Mantenibilidad:

La Reparabilidad es la posibilidad de corregir los defectos del software con un limitado gasto de trabajo.

Por su parte la Mantenibilidad es similar a la anterior, pero no está vinculada a la solución de componentes sino a cambios que no aparecían en la especificación original o que fueron establecidos en forma incorrecta.

En el ciclo de vida de todo producto de software, el tiempo de mantenimiento es un componente importante del tiempo total, por lo que ambas cualidades son vitales en cualquier programa.

Escalabilidad:

La escalabilidad es la capacidad de un software o de un hardware de crecer, adaptándose a nuevos requisitos conforme cambian las necesidades del negocio.

3.3 Evaluación de los Resultados.

Si las decisiones arquitectónicas determinan los atributos de calidad del sistema, entonces es posible evaluar las decisiones arquitectónicas con respecto a su impacto sobre dichos atributos.

Muchos de estos atributos como los que ya se vieron, no pueden ser medidos directamente. Sin embargo, la experiencia señala que la Arquitectura de software propicia algunos de ellos (SEI, 2000; Bass et al., 1998). Por lo tanto, la arquitectura del software es clave para la calidad. Por esto, un análisis de la Arquitectura debe ser ejecutado para determinar cuán satisfactoria es para el propósito del sistema (Bass et al., 1998). Para realizar esta estimación de la calidad a partir de la Arquitectura, algunos métodos, tales como SAAM, ATAM, ABDM, ARID, etc., incorporan diferentes técnicas de evaluación. Estas técnicas incluyen los escenarios, la simulación, los modelos matemáticos, el prototipo, entre otros, los cuales permiten una evaluación temprana.

3.3.1 Software Architecture Analysis Method (SAAM)

Según Kazman et al. (2001), el Método de Análisis de Arquitecturas de Software (Software Architecture Analysis Method, SAAM) es el primero que fue ampliamente promulgado y documentado. El método fue originalmente creado para el análisis de la modificabilidad de una arquitectura, pero en la práctica ha demostrado ser muy útil para evaluar de forma rápida distintos atributos de calidad, tales como modificabilidad, portabilidad, escalabilidad e integrabilidad.

El método de evaluación SAAM se enfoca en la enumeración de un conjunto de escenarios que representan los cambios probables a los que estará sometido el sistema en el futuro. Como entrada principal, es necesaria alguna forma de descripción de la Arquitectura a ser evaluada. De acuerdo con Kazman, las salidas de la evaluación del método SAAM son las siguientes:

- Una proyección sobre la Arquitectura de los escenarios que representan los cambios posibles ante los que puede estar expuesto el sistema.

- Entendimiento de la funcionalidad del sistema, e incluso una comparación de múltiples arquitecturas con respecto al nivel de funcionalidad que cada una soporta sin modificación.

Con la aplicación de este método, si el objetivo de la evaluación es una sola arquitectura, se obtienen los lugares en los que la misma puede fallar, en términos de los requerimientos de modificabilidad. Para el caso en el que se cuenta con varias arquitecturas candidatas, el método produce una escala relativa que permite observar qué opción satisface mejor los requerimientos de calidad con la menor cantidad de modificaciones.

3.3.2 Architecture Tradeoff Analysis Method (ATAM)

Este método de evaluación obtiene su nombre no solo porque refleja cuán bien una arquitectura particular satisface las metas de calidad, sino que también provee ideas de cómo esas metas de calidad interactúan entre ellas y como realizan concesiones mutuas (tradeoff).

El método consta de nueve pasos, divididos en cuatro grupos:

- Presentación
- Investigación y Análisis.
- Pruebas.
- Informes.

Paso 1: Presentar el ATAM:

- Los pasos del ATAM en resumen.
- Las técnicas que serán utilizadas para la obtención y análisis.
- Las salidas de la evaluación.

Paso 2: Presentar las pautas del negocio:

- Las funciones más importantes del sistema.
- Toda restricción técnica.
- La mayoría de los stakeholders.
- Las guías de la arquitectura.

Paso 3: Presentar la arquitectura:

- Las restricciones técnicas.
- Otros sistemas.
- Propuestas arquitectónicas.

Paso 4: Identificar las propuestas arquitectónicas:

El ATAM centraliza el análisis de una arquitectura en el entendimiento de sus propuestas arquitectónicas, en este paso son capturadas por el equipo de evaluación pero no son analizadas

Paso 5: Generar el árbol de utilidad de los atributos de calidad:

Este paso es crucial, pues guía el resto del análisis. Sin esta guía, los evaluadores pueden perder su tiempo analizando aspectos de la arquitectura que no son de interés para los stakeholders.

Paso 6: Analizar las propuestas arquitectónicas.

En este paso se analizan los escenarios de las propuestas arquitectónicas.

Paso 7: Lluvia de ideas y priorización de escenarios:

Este paso consiste en la generación de nuevos escenarios para:

- Representar los intereses de los stakeholders que no hayan sido comprendidos.

Paso 8: Analizar las propuestas arquitectónicas:

En este paso el equipo de evaluación realiza las mismas actividades que en el paso 6, mapeando los escenarios recientemente generados con ranking más alto en los artefactos arquitectónicos.

Paso 9: Presentar los resultados.

- El documento de propuestas arquitectónicas.
- El conjunto de escenarios priorizados.
- El árbol de utilidad.
- Los riesgos descubiertos.
- Los no riesgos documentados.
- Los sensitivity points y tradeoff points encontrados.

3.3.3 Métodos de validación en la Arquitectura planteada

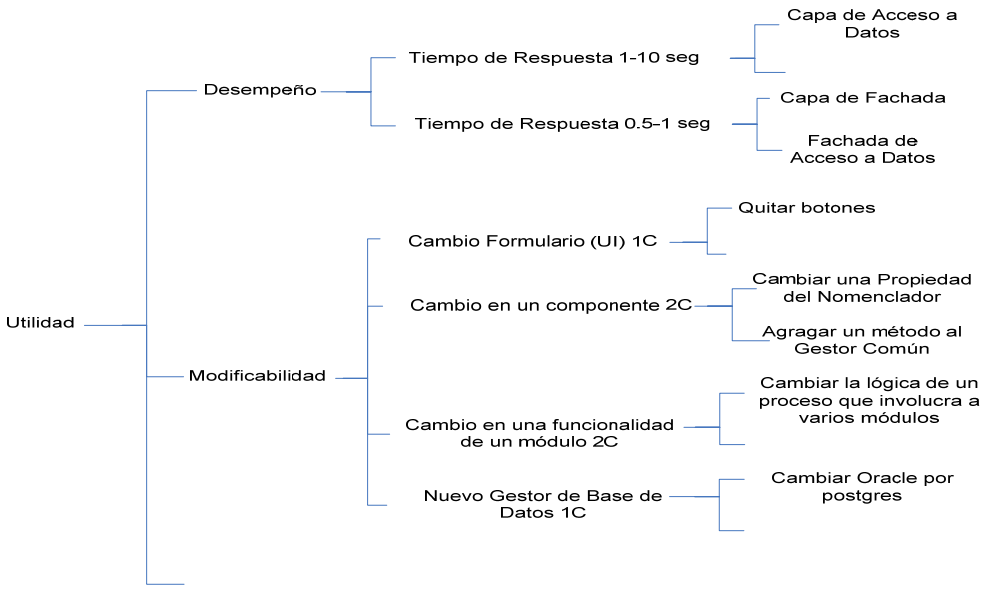
Si se tiene en cuenta que SAAM se centra fundamentalmente en el atributo de calidad modificabilidad y que además con su aplicación se garantiza que los interesados comprendan con mayor facilidad la arquitectura que se está evaluando, la documentación quede mejorada y que el esfuerzo y costo de los cambios se pueden estimar con anticipación, entonces resulta atractivo aplicarlo en la arquitectura propuesta. Sin embargo existen una serie de atributos y resultados que pueden quedar fuera de la validación si aplicáramos solamente el proceso anterior. Por tanto se decide implementar una variante propia que contempla algunas de los puntos comprendidos en los pasos de ATAM vinculadas con la propuesta que desarrolla SAAM. El hecho de seleccionar ATAM es precisamente porque este aborda aspectos y atributos que quedan más débiles en SAAM.

Para llevar a efecto esta variante se toma como punto de partida toda la descripción de la arquitectura que fue objetivo del Capítulo 2, la cual constituye una entrada para SAAM y favorece algunos de los pasos de ATAM. En este sentido se ofrecen las principales funcionalidades, algunas restricciones y una guía de la arquitectura. Además se hace un análisis de los principales escenarios arquitectónicos que tributan

directamente a situaciones de interés, si se tiene en cuenta los atributos de calidad que se quieren priorizar principalmente los que se relacionaron en la sección anterior.

Para complementar algunos de los pasos de ATAM se hace necesario además definir otros sistemas que interactúan con la solución. Tal es el caso de el módulo Mercantil y Público de la solución general de Registros y Notarias, estas aplicaciones se ven influenciadas mayormente en la interoperabilidad del software y forman parte de las restricciones técnicas conjuntamente con la integrabilidad entre los módulos propios del sistema de Administración Financiera y otras cuestiones relacionadas fundamentalmente con la comunicación y la navegación en el mismo.

Para ganar en claridad a continuación se hace una representación del árbol de utilidad referente a los atributos de calidad Desempeño y Modificabilidad. Aquí se introduce una nueva cualidad (Desempeño) que puede verse afectada en gran medida sobre todo por el uso de Spring.net y lógicamente con la utilización de NHibernate para el acceso a datos.



Leyenda

Seg Segundos que debe durar la operación.

C Componente Involucrado.

A continuación se describen los principales escenarios arquitectónicos que se pueden encontrar y se puntualizan todas las ideas que se han ido viendo.

Escenario	Cuando la Capa de Presentación le solicita información a la Capa de Negocio.
Objetivos del Negocio	Procesar alguna información.
Atributo de Calidad	Mantenibilidad, modificabilidad.
Estímulo	Obtener una colección de objetos.
Origen del Estímulo	Un formulario.
Elemento	Un botón.
Ambiente	Se está realizando un proceso de búsqueda.
Respuesta	Se muestra en el formulario la colección de objetos.
Medida de la Respuesta	Nivel de abstracción, repercusión.
Preguntas	¿Cuáles y cómo se afectan los componentes involucrados en la petición?
Anotación	Se requiere del uso de interfaces y la correcta implementación del patrón Abstract Factory.

Tabla 7 Escenario correspondiente a la comunicación entre la Capa de Presentación y Negocio.

La comunicación entre dos capas sin lugar a dudas marca un escenario que hay que tener en cuenta, precisamente porque en el mismo se encuentra el grado de dependencia que va a existir entre las capas involucradas. Por tal motivo mientras menos componentes se afecten en la operación más se favorecerá la mantenibilidad, modificabilidad y lógicamente la reparabilidad puesto que un cambio en alguna de los niveles implicados no resultará tan significativo.

En el caso de la comunicación entre Presentación y Negocio (ver tabla 7) se utiliza el patrón Abstract Factory para la conexión e interfaces que representan las funcionalidades del negocio. El patrón abstrae en alguna medida ambas capas aunque no en un 100%, esto se debe a que el Negocio y la Presentación no son tan críticos para la arquitectura. No obstante en caso de verse afectado algún componente de negocio solamente habría que actualizar la fabrica (Patrón Abstract Factory) puesto que la Presentación lo

único que conocerá es de esta y de las interfaces de negocio, facilitando así el mantenimiento y modificabilidad de la aplicación.

Por otra parte la Capa de Negocio y de Acceso a Datos (ver tabla 8) se comunican a través de una fachada que presenta la segunda, implementada sobre Spring.net. En este escenario si se le presta especial atención a tener el 100% de desacoplamiento puesto que la Capa de Datos es bastante crítica y propensa a cambios. Esta situación se resuelve con Spring.net que bajo su concepto de IoC y el uso de interfaces, limita la conexión entre ambas capas a un fichero XML permitiendo que la implementación del Acceso a Datos sea totalmente ajena al Negocio. Como se puede observar los componentes que se afectan en este nivel son mínimos y por tanto el sistema se hace altamente mantenible y flexible. Cualquier cambio y crecimiento en las implementaciones involucraría mayormente un fichero XML de configuración y no el código fuente.

Escenario	Cuando la Capa de Negocio le solicita información a la Capa de Acceso a Datos.
Objetivos del Negocio	Procesar algún dato para persistirlo u obtenerlo de la Base de Datos.
Atributo de Calidad	Mantenibilidad, modificabilidad, flexibilidad.
Estímulo	Obtener una colección de objetos.
Origen del Estímulo	Un gestor.
Elemento	Un método.
Ambiente	Se está realizando un proceso de búsqueda.
Respuesta	Se retorna la colección de objetos.
Medida de la Respuesta	Nivel de abstracción, repercusión.
Preguntas	¿Cuáles y cómo se afectan los componentes involucrados en la petición?
Anotación	Se requiere del uso de interfaces, la correcta implementación del patrón Fachada sobre Spring.net y el framework NHibernate para la interacción con la Base de Datos.

Tabla 8 Escenario correspondiente a la comunicación entre la Capa de Negocio y Acceso a Datos.

La comunicación entre los módulos de la solución (ver tabla 9) se realiza a través de una fachada la cual está implementada sobre Spring.net y mantiene la misma línea que la de la Capa de Datos aunque lógicamente en un nivel superior. Como ya se sabe, esta responsabilidad la tiene la Capa de Fachada que es donde van a radicar las implementaciones de los módulos que se necesiten cuando se vaya a acceder a sus funcionalidades, en este momento se le pide a la fachada que instancie el objeto.

Escenario	Cuando un módulo necesita comunicarse con otro para solicitar algún tipo de información o utilizar alguna funcionalidad.
Objetivos del Negocio	Llevar a cabo una operación que se salga del dominio del módulo.
Atributo de Calidad	Mantenibilidad, modificabilidad, flexibilidad, escalabilidad, integrabilidad.
Estímulo	Contabilizar.
Origen del Estímulo	Un gestor.
Elemento	Un método.
Ambiente	Se está llevando a cabo la contabilización de una operación contable.
Respuesta	Se registra la contabilización en un comprobante contable.
Medida de la Respuesta	Nivel de abstracción, repercusión, capacidad de incorporación de funcionalidades.
Preguntas	¿Cómo comparto los componentes de un módulo para llevar a cabo la operación? ¿Qué implica un cambio en alguno de estos componentes? ¿Cuáles componentes del módulo en que se está trabajando se ven afectados en el proceso?
Anotación	Se requiere del uso de interfaces, la correcta implementación del patrón Fachada sobre Spring.net.

Tabla 9 Escenario correspondiente a la comunicación entre módulos.

Existen funcionalidades y servicios que son comunes para la solución en general (ver tabla 10), por tal motivo se decide organizarlos en componentes y servicios según sea el caso. Estos elementos se aplican a casi todos los niveles de abstracción y justamente al encapsular determinadas funcionalidades en ellos, se logra que dependiendo del nivel en que se encuentre, el desarrollo quede cohesionado y eficiente. Así

mismo se obtiene una gran capacidad para la reparabilidad y la mantenibilidad puesto que un componente es independiente de otro en el sentido de la responsabilidad que tiene que asumir en el negocio, lo que posibilita que ante cualquier cambio solamente se tendría que trabajar sobre el elemento que se ve afectado directamente, en ningún momento se compromete el resto del código y como resultado disminuye exponencialmente el gasto de trabajo.

Otra potencialidad que brindan es la reutilización de código, en efecto al tener encapsulada cierta funcionalidad, resulta muy útil la reutilización de las mismas con solo incorporar el componente en cuestión, lo que significa un adelanto significativo en la implementación.

Escenario	Cuando varios módulos necesitan los mismos servicios y funcionalidades.
Objetivos del Negocio	Capturar todas las excepciones de sistema centralizadamente y por ende bajo una misma línea. Validar todos los datos de los formularios de una manera óptima y fácil.
Atributo de Calidad	Mantenibilidad, modificabilidad, flexibilidad, escalabilidad, reusabilidad.
Estímulo	Recoger los datos de una interfaz y validar que no haya campos vacios. En tal caso lanzar una excepción.
Origen del Estímulo	Una acción.
Elemento	Un botón.
Ambiente	Se está construyendo un objeto con los datos de la interfaz de usuario.
Respuesta	Se construye el objeto con los datos capturados.
Medida de la Respuesta	Esfuerzo. Tiempo de trabajo.
Preguntas	¿Qué componentes utilizar? ¿Cómo configurar los componentes? ¿Cómo utilizar los servicios? ¿Dónde desarrollar los componentes?
Anotación	Se requiere de una capacitación y la correcta configuración de los componentes y servicios.

Tabla 10 Escenario correspondiente a utilización de componentes y servicios.

La infraestructura de integración (ver tabla 11) descrita se apoya en un fichero XML y en el uso de las interfaces a través del patrón Reflection. Por lo mismo, sin importar cuánto se repare modifique o crezca el sistema de Administración Financiera, esto no resultaría significativo para quien este interactuando con este.

En este nivel también existe un 100% de desacoplamiento puesto que el agente externo solamente conocerá de interfaces, la implementación de las mismas las garantizará la "ArquitecturaBase" a través de un proxy.

Escenario	Cuando el sistema tiene que interactuar con aplicaciones externas.
Objetivos del Negocio	Brindar los datos y funcionalidad que le soliciten al sistema.
Atributo de Calidad	Mantenibilidad, modificabilidad, interoperabilidad.
Estímulo	Solicitud de datos referentes a la recaudación.
Origen del Estímulo	Un sistema externo.
Elemento	Un método.
Ambiente	Se está llevando a cabo un proceso que necesita datos y funcionalidades de la recaudación controlados por nuestro sistema.
Respuesta	Se devuelven los datos solicitados.
Medida de la Respuesta	Nivel de abstracción, repercusión.
Preguntas	¿Cómo comparto los componentes de mi sistema para llevar a cabo la operación? ¿Qué implica un cambio en alguno de estos componentes?
Anotación	Se requiere del uso de interfaces, la correcta implementación del patrón Proxy y Reflection combinados.

Tabla 11 Escenario correspondiente a la integración.

Los resultados reflejan la escalabilidad en gran medida en el módulo "ArquitecturaBase", el cual facilita el crecimiento y adaptación a nuevos requerimientos del negocio puesto que como se ha visto proporciona la infraestructura donde descansan los módulos. Así mismo durante cada una de las iteraciones que se han

reflejado en este documento, la arquitectura va ganando en funcionalidades sin que esto sea traumático o significativo para la misma.

De manera general es importante destacar que el hardware según se vio en la vista de despliegue (ver figura 34) contribuye a la escalabilidad y robustez de la arquitectura si tenemos en cuenta que existe un servidor local en cada UEL que garantizará, en caso de perder conectividad, que se atiendan todas las peticiones que se puedan realizar sin verse afectado el sistema. También se asegura el respaldo de la información mediante mecanismos de réplica de datos y así mismo liberar la tensión y la sobrecarga de la red y el servidor.

Por otra parte, puesto que el proceso de implementación ya culminó la primera iteración, la arquitectura fue probada y sometida a diversas situaciones de stress que posibilitaron perfeccionarla y validarla en alguna medida, haciendo un análisis de los resultados esperados. Más aún si se tiene en cuenta que un método cuantitativo para hacerlo es la simulación de la misma.

A partir de todos los elementos que se han reflejado, se puede afirmar que la solución propuesta es totalmente flexible, modificable, reparable y mantenible. Igualmente los factores que aseguran estas cualidades influyen directa o indirectamente en la integrabilidad, interoperabilidad y la escalabilidad, aunque en este caso, el hardware también juega su papel.

3.4 Conclusiones.

Alcanzar un atributo de calidad puede tener un efecto positivo o negativo sobre otros atributos de calidad. Por tanto el Arquitecto de software tiene que tener claros aquellos que considere de mayor prioridad dependiendo de los resultados que desea obtener Algunos atributos de calidad deben ser diseñados y evaluados a nivel arquitectónico. Otros no son susceptibles de ser alcanzados a nivel arquitectónico, sin embargo en su conjunto tributan a darle solución a los objetivos que se persiguen.

Particularmente esta Línea Base garantiza la flexibilidad, la mantenibilidad y la escalabilidad por encima de otras cualidades que aunque algunas no dejan de estar presentes si adquieren un carácter secundario respecto a estas.

Conclusiones

La Arquitectura es una pieza importante en el ciclo de vida de un software, más aún teniendo en cuenta que la metodología seleccionada (RUP) se centra en esta para su desarrollo. Por tal motivo es importante prestar especial atención al estado de la documentación y las tecnologías existentes actualmente en el mundo referentes a esta materia. A partir de la investigación resultaron conceptos y tecnologías importantes:

- Los relacionados con la disciplina: el concepto de Arquitectura de software, conectores, componentes, vistas y notaciones.
- Los Patrones y Estilos de Arquitectura.
- La calidad arquitectónica y atributos de calidad.
- Framework de desarrollo existentes (NHibernate, Spring.net).

Así mismo, definir la Línea Base de la Arquitectura del sistema de Administración Financiera posibilitó llegar a un entendimiento entre todos los involucrados en el proceso de desarrollo, así como establecer un marco idóneo para representar todo los componentes y elementos que la conforman. En este sentido:

- Se describieron los conceptos, terminologías y el entorno donde se va a desenvolver la Arquitectura del sistema.
- Se describió la arquitectura desde dos enfoques Vertical y Horizontal.
- Se establecieron los componentes más significativos y como se implantan estos en la solución (Capas, Módulos, Clases).
- Se describieron las políticas de integración y los mecanismos que hay que asegurar para lograrla (Patrón Proxy, Reflection).
- Se describieron 4 de las 5 vistas propuestas por la metodología RUP (Vista de Casos de Uso, Vista Lógica, Vista de Implementación, Vista de Despliegue) acorde a los procesos que se encuentran en la solución

Cada uno de los artefactos y elementos que se mencionaron anteriormente se enmarcan en 3 iteraciones durante las cuales se va refinando este documento hasta obtener una versión final.

Por otra parte, después de haber obtenido un modelo para la arquitectura del sistema, fue necesario efectuar la validación del mismo para analizar si efectivamente respondía a los atributos de calidad que se desean obtener, para lograr el objetivo y como parte de este proceso:

- Se definieron cuales son los atributos de calidad que se desean priorizar en la Arquitectura del sistema (mantenibilidad, modificabilidad, flexibilidad).
- Se seleccionó como métodos para validar la Arquitectura propuesta SAAM y ATAM los cuales se conjugaron en una variante menos rigurosa pero que garantizara el objetivo fundamental de esta etapa.
- Se definieron los escenarios principales para la Arquitectura del sistema los cuales abarcan la comunicación entre capas, la utilización de componentes, la comunicación entre módulos y la integración con sistemas externos.

Recomendaciones

Con el propósito de ampliar y mejorar la documentación de la arquitectura presentada en este trabajo, se plantean las siguientes recomendaciones:

- No se mencionan características de los servidores como: balanza de carga, pool connection, seguridad de datos, mecanismos de protección contra ataques en la base de datos o modificación de los registros. Estos factores son muy importantes por tanto se recomienda incluirlos en futuras iteraciones o versiones de este documento.

- De manera general no se desarrollan los temas relacionados con la seguridad que entre otras cosas incluyen principios de la programación segura. Si bien se refleja en el documento que la seguridad se garantiza con la utilización del framework Común y algunos puntos heredados de la Arquitectura General si es importante incluir en este trabajo algunas particularidades específicas del sistema de Administración Financiera.

- Se recomienda manejar los temas relacionados con la encriptación, la transferencia de datos entre capas y entre subsistemas y siguiendo el punto anterior como se garantizan los atributos de calidad de seguridad mínimos en la arquitectura.

- Se recomienda continuar profundizando en la validación de la arquitectura propuesta. En este sentido sería bueno aplicar la totalidad del método ATAM puesto que este es mucho más completo que SAAM y por tanto se obtiene un resultado mucho más acabado.

Referencias Bibliográficas

Barbacci, M., Klein, M., Longstaff, T., & Weinstock, C. (1995). Quality Attributes. Carnegie Mellon University. Technical Report.

Bass, L., Barbacci, M., Carriere, J., Kazman, R., Klein, M., y Lipson, H. (1999). Attribute Based Architectural Styles. Software Engineering Institute, Carnegie Mellon University. Pittsburgh.

Bass, L., Clements, P., & Kazman, R. (1998). Software Architecture in practice. Addison-Wesley. Bass, L., Klein, M., & Bachmann, F. (2000). Quality Attribute Design Primitives. Software Engineering Institute, Carnegie Mellon University.

Canal, Carlos. (marzo 2005). Conferencia: “Arquitectura, marcos de trabajo y patrones”. Universidad de Málaga.

Larman, Craig [2004] (2005). Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd, Prentice Hall PTR.

[G95] Gamma, E., et al., Design Patterns: Elements of Reusable Object-Oriented Software. 1995.

Booch, G., Rumbaugh, J., & Jacobson, I. (1999). The UML Modeling Language User Guide. Adisson-Wesley Bosch, J. (2000). Design & Use of Software Architectures. Addison-Wesley. Bredemeyer, D., & Malan, R. (2002). The Visual Architecting Process. White Paper.

Longman, Inc. Lane, T. (1990). Studying Software Architecture Through Design Spaces and Rules. Technical report. The Computer Science Department, Carnegie Mellon University.

Larman, C. (1999). UML y Patrones: Introducción al análisis y diseño orientado a objetos. Prentice-Hall Hispanomericana. Losavio, F., Chirinos, L., Lévy, N., & Ramdane-Cherif, A. (2003). Quality Characteristics for Software Architecture. A ser publicado en el JOT 2003.

Perry, D., & Wolf, A. (1992). Foundations for the Study of Software Architecture. ACM Sigsoft - Software Engineering Notes, Vol 17, No. 4.

Pressman R. (2002) Ingeniería de Software. Un Enfoque Práctico. Quinta Edición. Mc Graw Hill. OTI. (2003) Eclipse Platform Technical Overview.

Rational Software Corporation. (1998). "Rational Unified Process: Best Practices for Software Development Teams".

Barbacci, M., Klein, M., Longstaff, T., & Weinstock, C. (1995). *Quality Attributes.* Carnegie Mellon University. Technical Report.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). Pattern – Oriented Software Architecture. A System of Patterns.

John Wiley & Sons, Inglaterra. Carriere, J., Kazman, R., Woods, S. (2000). Toward a Discipline of Scenariobased Architectural Engineering. Software Engineering Institute, Carnegie Mellon University.

Grady, R., & Caswell, D. (1987) Software Metrics: Establishing a company-Wide Program. Prentice Hall.

Kazman, R. (1996). Tool Support for Architecture Analysis and Design. Department of Computer Science, University of Waterloo.

Kazman, R., Clements, P., Klein, M. (2001). Evaluating Software Architectures. Methods and case studies. Addison Wesley. Kruchten, P. (1999). The Rational Unified Process. Reading, MA: Addison Wesley.

[KC94] Paul Kogut y Paul Clements. "Features of Architecture Description Languages". Borrador de un CMU/SEI Technical Report, Diciembre de 1994.

[KC95] Paul Kogut y Paul Clements. "Feature Analysis of Architecture Description Languages". En Proceedings of the Software Technology Conference (STC'95), Salt Lake City, Abril de 1995.

[Ves93] Steve Vestal. “A cursory overview and comparison of four Architecture Description Languages”. Technical Report, Honeywell Technology Center, Febrero de 1993.

[LV95] David Luckham y James Vera. “An Event-Based Architecture Definition Language”. IEEE Transactions on Software Engineering, pp. 717-734, Setiembre de 1995.

[SDK+95] Mary Shaw, Robert DeLine, Daniel Klein, Theodore Ross, David Young y Gregory Zelesnik. “Abstractions for Software Architecture and Tools to Support Them”. IEEE Transactions on Software Engineering, pp. 314-335, Abril de 1995.

[SG94] Mary Shaw y David Garlan. “Characteristics of Higher-Level Languages for Software Architecture”. Technical Report CMU-CS-94-210, Carnegie Mellon University, Diciembre de 1994.
[SG95] Mary Shaw y David Garlan. “Formulations and Formalisms in Software Architecture”. Springer-Verlag, Lecture Notes in Computer Science, Volumen 1000, 1995.

[Med96] Neno Medvidovic. “A classification and comparison framework for software Architecture Description Languages”. Technical Report UCI-ICS-97-02, 1996.

[Mon98] Robert Monroe. “Capturing software architecture design expertise with Armani”. Technical Report CMU-CS-163, Carnegie Mellon University, Octubre de 1998.

[GMW00] David Garlan, Robert Monroe y David Wile. “Acme: Architectural description of component-based systems”. Foundations of Component-Based Systems, Gary T. Leavens y Murali Sitaraman (eds), Cambridge University Press, pp. 47-68, 2000.

[MK96] Jeff Magee y Jeff Kramer. “Dynamic structure in software architectures”. En Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 3–14, San Fransisco, Octubre de 1996.

[RH2002] Red Hat. (2002). Hibernate. (Red Hat Americas;Red Hat EMEA;Red Hat APAC) Recuperado el 15 de febrero de 2008, de Hibernate: http://www.hibernate.org/343.html.

[BK2005] BAUER CHRISTIAN y KING GAVIN. "*Hibernate in Action*". A guide to the concepts and practices of object/relational mapping, 2005.

[PESSHCR2004-2006] Mark Pollack, Rick Evans, Aleksandar Seovic, Federico Spinazzi, Rob Harrop, Griffin Caprio, Choy Rim, The Spring Java Team. Spirng.net Reference Documentation. 2004-2006.

Anexos

Anexo 1. Red

Se propone la implementación de una red WAN corporativa utilizando una Red Privada Virtual (VPN) que aumente el nivel de seguridad de los datos. La implementación de dicha red VPN se propone a nivel de hardware con el objetivo de aliviar el procesamiento (demoras) que se incurren en soluciones a nivel de software.

Se propone la utilización de una topología de la red tipo estrella. La interacción entre los sistemas será utilizando la familia de protocolos TCP/IP (ver figura 1).

Se propone la implementación de una red de tecnología inalámbrica para el funcionamiento de las oficinas (ver figura 2).

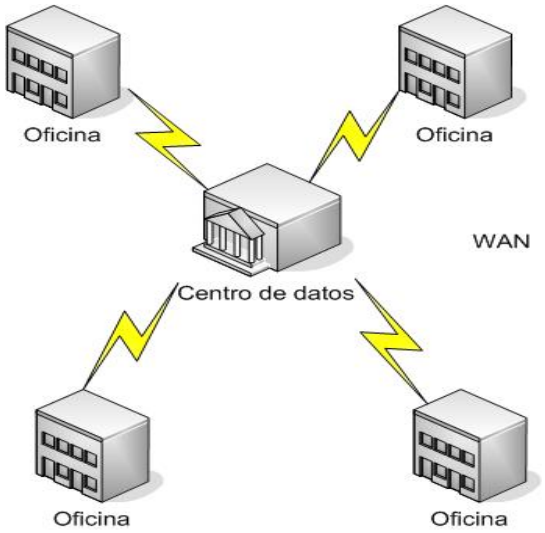


Figura 1 Arquitectura de red del sistema. Topología tipo estrella.

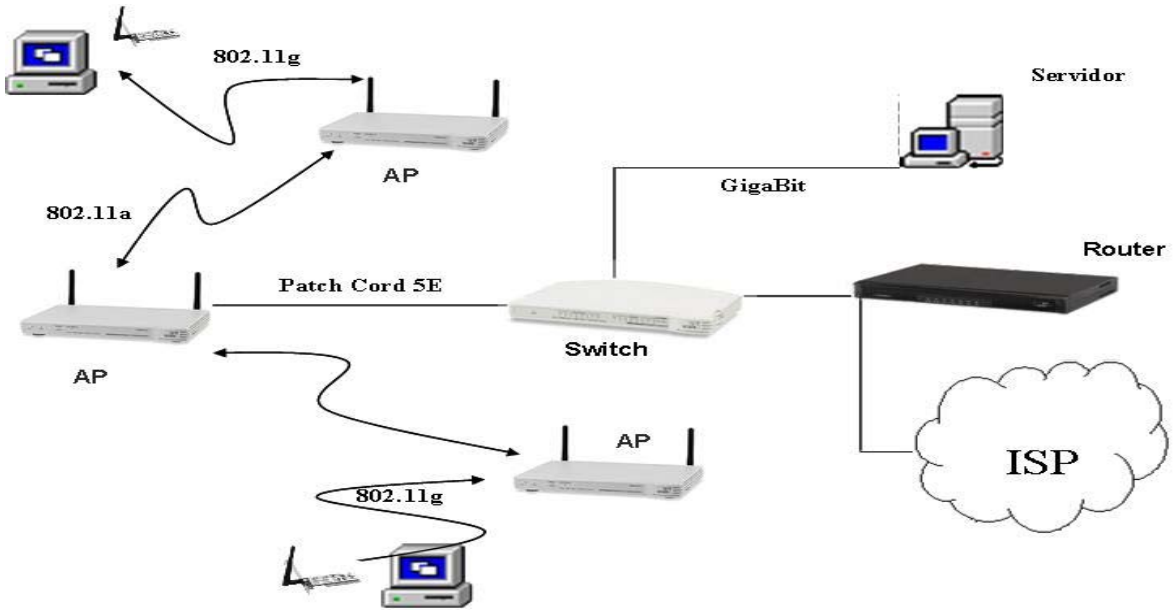


Figura 2 Arquitectura de red de las UE. Tecnología inalámbrica.

Anexo 2. Arquitectura respecto de los servidores de datos

La Arquitectura del sistema se ha diseñado basada en un modelo descentralizado (**ver figura 3**), lo que implica una comunicación intensa. De esta manera se requiere mantener una conexión a la red WAN de mediana velocidad que garantice la comunicación efectiva desde cada una de las UE hacia el centro de datos.

A pesar de esto existen momentos y pasos en los que las consultas se realizarán totalmente sobre entidades locales definidas para tal efecto, permitiendo de esta forma el normal funcionamiento de las UE en estado de desconexión con el centro de datos.

Para esto se han definido los siguientes elementos:

- Se mantendrá un cluster de Oracle a nivel central con capacidad de soportar todos los sistemas definidos.
- Existirán servidores locales con capacidad necesaria para el procesamiento de las solicitudes del conjunto de aplicaciones de las diferentes oficinas.
- Las aplicaciones siempre solicitarán los datos a través del servidor local.
- Desde cada servidor local se establecerá la conexión con servidores centrales para mantener la actualización de los datos en ambos sentidos.

Los servidores a nivel central poseerán Oracle Enterprise Edición 10g R2 Real Aplication Cluster, y en los servidores locales de las UE se poseerá el Oracle Standard Edition ONE versión 10g R2

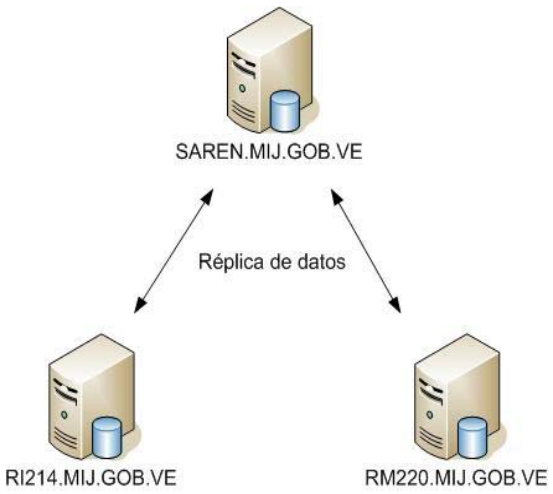


Figura 3 Modelo descentralizado.

Existen dos premisas para la comunicación entre las oficinas y el servidor central:

- Las oficinas deben poder realizar un elevado por ciento de sus funciones aunque no exista conexión con el servidor central.
- Minimizar el tiempo de atención al ciudadano evitando colas innecesarias y retrasos por el sistema.

- Las aplicaciones siempre encuestarán por defecto al servidor local. En caso de que no encuentren la información solicitada entonces la búsqueda tendrá lugar en el centro de datos.

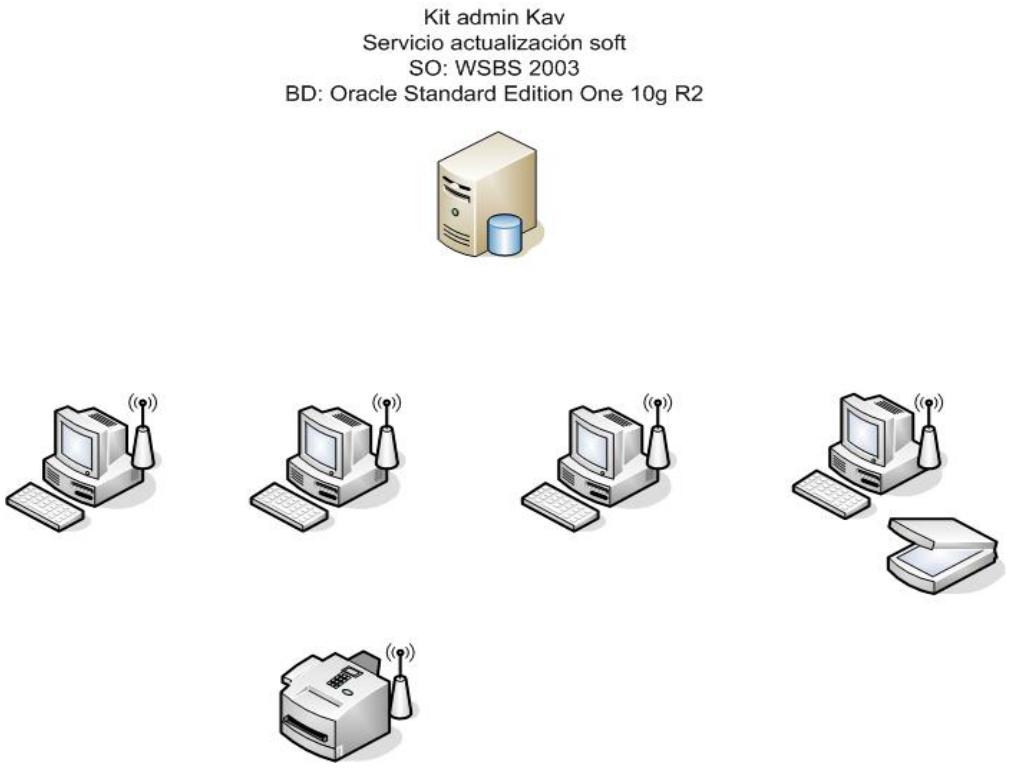


Figura 4 Arquitectura física de las oficinas.

Anexo 3. Despliegue

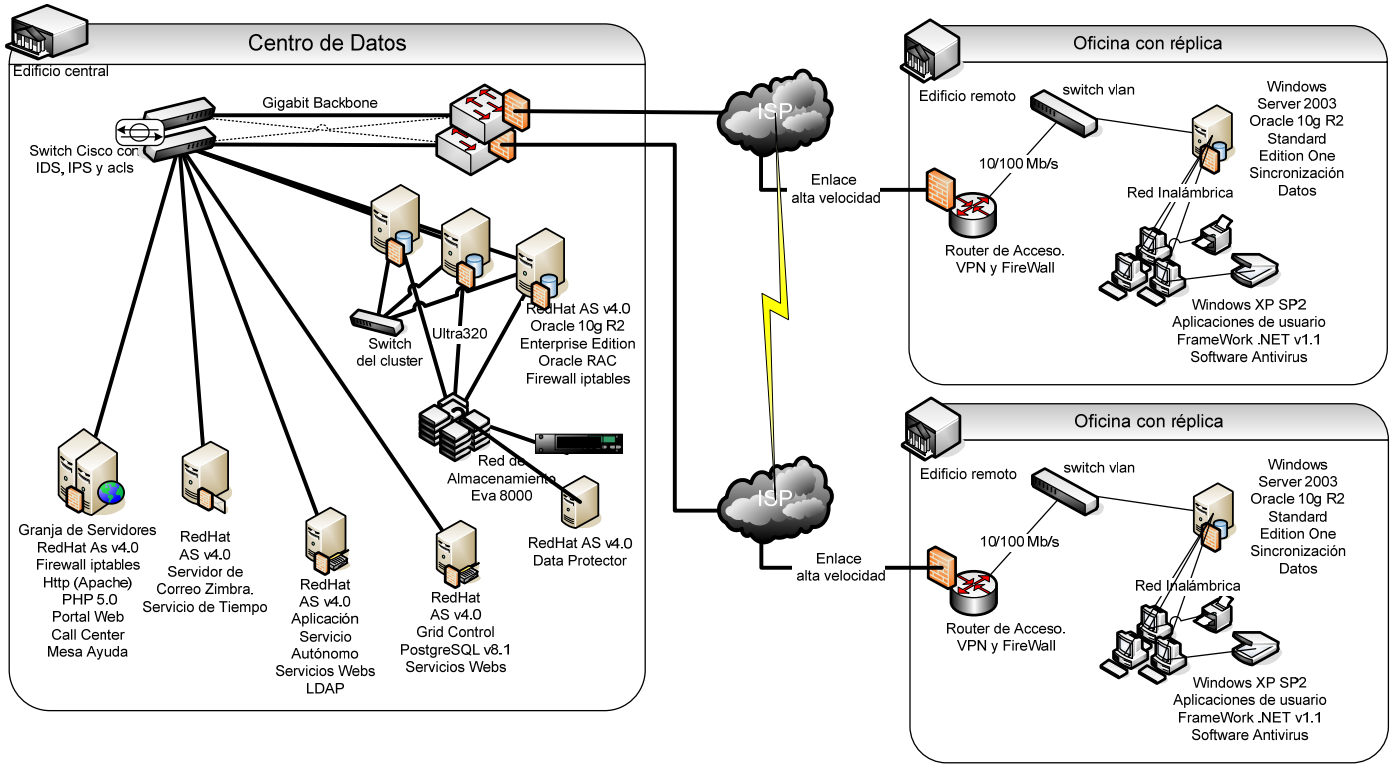


Figura 5 Despliegue de la solución de software para automatizar los Registros y Notarias de la República Bolivariana de Venezuela.

Glosario de Términos

Objetos falsos: Son objetos que simulan todos los posibles resultados contenidos en sus funcionalidades. La utilidad de estos objetos radica en que agilizan el proceso de implementación pues no es necesario esperar por cierta funcionalidad si de ante mano ya se le tiene representada en todas las vertientes posibles.

Atómico: El término se usa para definir la cualidad de un componente determinado de ser suficiente, indivisible.

Encapsular: Al empaquetamiento de las variables de un objeto con la protección de sus métodos se le llama encapsulamiento. Típicamente, el encapsulamiento es utilizado para esconder detalles de la puesta en práctica no importantes de otros objetos. Entonces, los detalles de la puesta en práctica pueden cambiar en cualquier tiempo sin afectar otras partes del programa. El término encapsular se refiere a la realización del encapsulamiento.

Abstracción/Enajenación: Se refiere al énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?", es decir representa el nivel de desconocimiento que tenga una funcionalidad de su implementación. El común denominador en la evolución de los lenguajes de programación, desde los clásicos o imperativos hasta los orientados a objetos, ha sido el nivel de abstracción del que cada uno de ellos hace uso.