**Universidad de las Ciencias Informáticas**
**Facultad 3**

# Título: Framework para la arquitectura de los sistemas registrales del proyecto Registros y Notarias Fase II.

**Trabajo de Diploma para optar por el título de Ingeniero Informático**

**Autores**: Earles Reyes Viada
Carlos Yaniel Jiménez Ramírez

**Tutores**: Ing. Yosvany Márquez Ruiz
Ing. Yunier Pérez Barroso

**Junio 2011**

**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 de ________ del año ________.

| | |
|---|---|
| Carlos Yaniel Jiménez Ramírez | Yosvany Márquez Ruiz |
| ____________________________ | ____________________________ |
| Firma de Autor | Firma de Tutor |
| Earles Reyes Viada | Yunier Pérez Barroso |
| ____________________________ | ____________________________ |
| Firma de Autor | Firma de Tutor |

AGRADECIMIENTOS

*Primero que todo agradezco a Yosvany, mejor tutor hay que mandarlo a hacer.*

*Mi más grande agradecimiento a mis padres, por enseñarme a caminar, por mostrarme el camino a seguir, porque todo lo que hago lleva puesto sus nombres.*

*A mis suegros, mis segundos padres.*

*Agradezco a la Revolución, a esta gran obra que es la Universidad de las Ciencias Informáticas, sin dudas mi mejor escuela.*

*A todos mis compañeros de aula y de proyecto en estos 5 años.*

*A Earles.*

*A Yunier.*

*Y no por último menos importante, a mi princesa, mi otra mitad, ella sabe cuánto significa para mi cada segundo a mi lado, cada gota del apoyo que me brinda.*

***Carlos Yaniel***

*Agradezco a Yosvany por ser un tutor de 5 puntos con ****.*

*A mis padres.*

*A mis hermanas y mis cuñados por hacer mi vida más fácil aquí en la universidad.*

*A Carlos.*

***Earles***

**DEDICATORIA**

*A mis padres y mi hermano.*

*A mi segunda familia.*

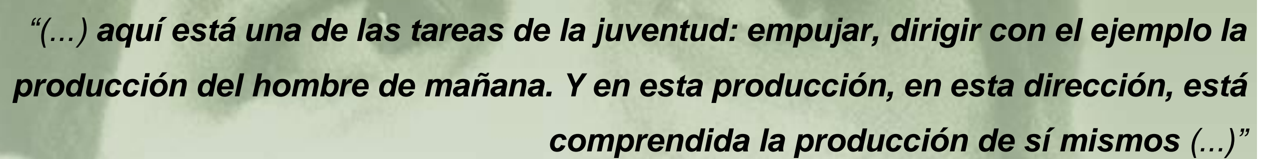
*A mi futura esposa.*

***Carlos Yaniel***

*A mis padres.*

*A Tati (Annelis)*

***Earles***



*“(...) **aquí está una de las tareas de la juventud: empujar, dirigir con el ejemplo la producción del hombre de mañana. Y en esta producción, en esta dirección, está comprendida la producción de sí mismos** (...)”*

***Ernesto “Che” Guevara***
***Mayo de 1964***

# RESUMEN

El término “framework”, cuya traducción aproximada sería “marco de trabajo”, es un concepto que los que se dedican al desarrollo de software utilizan, conocen o, como mínimo, lo han escuchado alguna vez. En general, con este término, se hace alusión a una estructura de software compuesta por componentes personalizables e intercambiables para el desarrollo de una aplicación. Además está asociado a un determinado tipo de aplicaciones, lo que implica que su alcance esté acotado; o sea que está ligado a un dominio concreto. En otras palabras, un framework se puede considerar como una aplicación genérica incompleta y configurable a la que se puede añadir las últimas piezas para construir una aplicación concreta.

El presente trabajo propone el proceso de desarrollo de un framework para sustentar la arquitectura del proyecto Registros y Notarias Fase II basada en el estilo arquitectónico (“Arquitectura en N-Capas Orientada al Dominio”). Diseñado para un tipo concreto de aplicación: aplicaciones empresariales complejas, con una vida relativamente larga y normalmente con un volumen de cambios evolutivos considerables.

Para el desarrollo de la investigación se empleó como metodología el Proceso Unificado de Desarrollo (RUP). Los artefactos se generaron usando como lenguaje de modelado el Lenguaje Unificado de Modelado (UML) y auxiliados por el Enterprise Architect como herramienta de Ingeniería de Software Asistida por Computadora (CASE). Para garantizar la calidad de los artefactos generados (especificación de los requisitos, diagramas de casos de uso del sistema y diagrama de clases del sistema) se aplicaron métricas, que arrojaron resultados positivos.

**PALABRAS CLAVES:** Framework, Requisitos, Casos de uso, Clases, Diseño.

## ÍNDICE

DECLARACIÓN DE AUTORÍA....II
AGRADECIMIENTOS....III
DEDICATORIA....IV
RESUMEN....VI
INTRODUCCIÓN....1
CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA....5
1.1. INTRODUCCIÓN....5
1.2. DEFINICIÓN DE FRAMEWORK....5
1.2.1. VENTAJAS DE LOS FRAMEWORK....6
1.2.2. CLASIFICACIÓN DE LOS FRAMEWORKS....6
1.2.3. FRAMEWORKS PARA LA INVERSIÓN DE CONTROL Y LA INYECCIÓN DE DEPENDENCIAS....7
1.2.4. FRAMEWORKS PARA LA PERSISTENCIA DE DATOS....10
1.3. ARQUITECTURA DE SOFTWARE....13
1.3.1. RELACIÓN DE ABSTRACCIÓN ENTRE ESTILOS Y PATRONES....13
1.3.2. ESTILOS ARQUITECTÓNICOS....14
1.3.3. PATRONES ARQUITECTÓNICOS....16
1.4. PATRONES DE DISEÑO....19
1.5. METODOLOGÍAS DE DESARROLLO DE SOFTWARE....21
1.5.1. CARACTERÍSTICAS DESEABLES DE UNA METODOLOGÍA....22
1.5.2. VENTAJAS DE TENER UNA BUENA METODOLOGÍA....23
1.5.3. METODOLOGÍAS TRADICIONALES....23
1.5.4. METODOLOGÍAS ÁGILES....24
1.5.5. RATIONAL UNIFIED PROCESS (RUP)....25
1.5.6. XP....28
1.5.7. JUSTIFICACIÓN DE LA METODOLOGÍA A UTILIZAR....29
1.6. EL LENGUAJE UNIFICADO DE MODELADO....30
1.7. HERRAMIENTAS CASE....32
1.7.1. RATIONAL ROSE ENTERPRISE EDITION....33
1.7.2. VISUAL PARADIGM FOR UML....34
1.7.3. ENTERPRISE ARCHITECT (EA):....34
1.8. CONCLUSIONES....36
CAPÍTULO 2: SOLUCIÓN PROPUESTA....37
2.1. INTRODUCCIÓN....37
2.2. PROPUESTA DEL SISTEMA....37
2.2.1. CAPAS HORIZONTALES....38
2.2.2. CAPAS VERTICALES....39
2.3. MODELO DEL SISTEMA....42
2.3.1. REQUISITOS DEL SISTEMA....42
2.3.2. MODELO DE CASOS DE USO DEL SISTEMA....46
2.4. DISEÑO....51
2.4.1. MODELO DE DISEÑO....51
2.5. CONCLUSIONES....54

CAPÍTULO 3: ANALISIS DE LOS RESULTADOS ........ 55
3.1. INTRODUCCIÓN ........ 55
3.2. MÉTRICAS DE LA CALIDAD DE LA ESPECIFICACIÓN ........ 56
3.3. MÉTRICA NOAS / NOS ........ 58
3.4. MÉTRICAS PARA EL MODELO DE DISEÑO ........ 64
3.4.1. LA SERIE DE MÉTRICAS CK ........ 64
3.4.2. MÉTRICAS PROPUESTAS POR LORENZ Y KIDD ........ 64
3.5. CONCLUSIONES ........ 66
CONCLUSIONES GENERALES ........ 67
RECOMENDACIONES ........ 68
BIBLIOGRAFÍA ........ 69

# INTRODUCCIÓN

El software constituye un ingrediente indispensable para el funcionamiento de una computadora. Está formado por una serie de instrucciones y datos, que permiten aprovechar todos los recursos de la misma, de manera que pueda resolver gran cantidad de problemas. La computadora es sólo un conglomerado de componentes electrónicos a los que el software le da vida, haciendo que funcionen de forma ordenada.

El software ha sido factor relevante en el desarrollo de las tecnologías y es actualmente una característica fundamental que debe tener toda organización que quiera ser más competitiva. Las empresas están obligadas a tomar decisiones cada vez más rápidas y precisas por lo que en el mundo la inmensa mayoría de estas controlan sus principales actividades por medio de sistemas informáticos de diversos grados de complejidad. La esencia de la utilidad del software para las empresas radica en que los sistemas informáticos, son capaces de controlar y gestionar la información de toda clase de procesos económicos y financieros, procesos de producción, proyectos inversionistas y cualquier otra actividad que se desee informatizar. Con el desarrollo de software el hombre agiliza, dinamiza, facilita y hace más humano y productivo su trabajo, pues la máquina se convierte en un auxiliar del hombre, liberándole de las tareas repetitivas.

A medida que evoluciona la sociedad y se desarrollan a un ritmo exponencial las tecnologías de la información y las comunicaciones, son más rigurosas las exigencias para la solución de problemas más complejos a través del software.

Las aplicaciones de gestión empresarial en la mayoría de los casos, llevan un correcto y ordenado control de productos, clientes, generan facturas y presupuestos y hacen búsquedas, entre otras tantas funcionalidades propias de un negocio posibilitando un ahorro de dinero y tiempo en Instituciones. De manera general, existen un conjunto de funciones comunes a ellas, entre las que se encuentran: manejar grandes volúmenes de datos que por lo general deben ser almacenados en diferentes medios, establecer niveles de seguridad para acceder y manipular dichos datos, implementar un sistema de mensajería para comunicarse con los usuarios, establecer un sistema de trazas que facilite las funciones de auditoría, entre muchas otras.

La República Bolivariana de Venezuela desde hace algún tiempo ha estado inmersa en una ardua tarea en cuanto a la modernización de su sistema Registral y Notarial para poder estar a la altura de muchos países con un gran desarrollo en este aspecto. Los primeros pasos para cumplir sus objetivos se evidencian en 1993 con la promulgación de la Ley del Registro Público, cuando el gobierno autoriza el inicio de un proceso de digitalización de documentos en los Registros Principales y Notarías Públicas de todo el país con el fin de mantener y conservar el patrimonio registral de cada oficina.

Posteriormente el 22 de diciembre del 2006 entró en vigor el nuevo Decreto Ley del Registro Público y del Notario decretada por la Asamblea Nacional de la República Bolivariana de Venezuela con el objetivo de otorgar seguridad jurídica y garantizar los principios de libertad contractual y de legalidad de los bienes, derechos, actos, contratos y negocios jurídicos mediante la automatización de sus procesos registrales y notariales para así ofrecer un mayor grado de seguridad jurídica a todos los ciudadanos en este país.

Como parte de los acuerdos de cooperación entre Cuba y la República Bolivariana de Venezuela nace un proyecto de software para automatizar los procesos de gestión del Servicio Autónomo de Registros y Notarías (SAREN), con el fin de que se pueda ofrecer garantía y seguridad jurídica de las operaciones que se realicen.

A lo largo de su desarrollo, este proyecto ha enfrentado problemas para construir las bases fundamentales que soportan la arquitectura de su solución en cada una de sus fases. Esta situación de manera general, se debe a que dicho proyecto aún no cuenta con un framework bien definido, que les permita aprovechar funciones comunes e importantes ya antes mencionadas que comparten las aplicaciones de gestión empresarial, pero además que permita otras funciones orientadas a soportar la arquitectura y por ende, a favorecer el desarrollo de la solución de software, como son:

- Facilitar los cambios en el sistema de una forma rápida y poco costosa debido a cambios ocasionales en los requisitos del sistema.
- Facilitar la reutilización del diseño, el código fuente o algunos de sus componentes aumentando la eficiencia y los niveles de productividad en el equipo de desarrollo, reduciendo los costos de producción.
- Favorecer el desarrollo de la solución al implementar de manera genérica los patrones de diseño más comunes.

- Facilitar el acoplamiento entre las diferentes partes que lo componen, ya sean paquetes o componentes, incluyendo además los elementos externos al sistema.
- Ayudar a la integridad de la información que pueda manejar el sistema, facilitando la seguridad de los mismos.
- Garantizar una alta capacidad de prueba lo que favorece a detectar y corregir fácilmente los errores.

Partiendo de la situación anterior, se identifica el siguiente **problema de investigación**: ¿Cómo contribuir al desarrollo de un framework que ayude a sustentar la arquitectura de la solución de software del proyecto Registros y Notarías Fase II?

Con vista a la solución del problema antes planteado se define como **objeto de estudio** el proceso de desarrollo de software.

El **objetivo general** del presente trabajo de diploma es realizar las etapas de Requisitos y Diseño de un framework que ayude a sustentar la arquitectura de la solución de software del proyecto Registros y Notarías Fase II.

Definiendo así el **campo de acción** como requisitos y diseño de framework para aplicaciones de gestión empresarial.

La **idea a defender** planteada es que con la realización de las etapas de requisitos y diseño de un framework se ayudaría a sustentar la arquitectura obtenida en el marco del proyecto Registros y Notarías en su Fase II.

**Objetivos Específicos:**

- Elaborar el marco teórico de la investigación.
- Obtener los artefactos de la etapa de requisitos para los componentes fundamentales del framework.
- Obtener los artefactos de diseño necesarios para los componentes fundamentales del framework.
- Evaluar la solución propuesta.

**Estructuración del contenido con una breve descripción de sus partes.**

El presente trabajo de diploma ha sido estructurado de la siguiente forma:

**Capítulo 1:** En este capítulo se realiza un estudio del proceso de desarrollo de software, por lo que se hace un análisis de las diferentes metodologías y tecnologías que se pueden emplear en la solución, seleccionando la más apropiada para llevar a cabo dicho proceso con éxito y calidad. Se realiza además un estudio del estado del arte de los principales frameworks que existen en el mundo, tanto libres como propietarios. Se realiza un estudio de los patrones de diseño, de arquitectura y estilos arquitectónicos que pueden ser empleados en la solución.

**Capítulo 2**: Descripción de la solución propuesta, se hace una descripción detallada de la solución que se propone que posibilita una mejor compresión. Se enuncian los requerimientos funcionales y no funcionales que debe presentar la solución a construir. Se determinan los casos de uso del sistema y los actores que interactuarán con éste, así como el diseño del sistema con los diagramas correspondientes a este flujo.

**Capítulo 3**: Este capítulo abordara acerca de las métricas fundamentales para validar la calidad de la especificación de los requisitos, los casos de usos, así como la calidad del diseño, dando una breve descripción de en qué consisten las mismas, como se aplican y cuáles fueron los resultados arrojados una vez aplicadas a nuestra solución.

# CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA

## 1.1. Introducción

En este capítulo se hace referencia al proceso de desarrollo de software, así como las metodologías de desarrollo de software, tecnologías y tendencias actuales que pueden ser útiles en el desarrollo de la propuesta de solución. Se define el concepto de framework, se aborda la importancia que tienen los mismos, así como un estudio del estado del arte de estos, de los patrones de diseño y de arquitectura, así como algunos estilos arquitectónicos.

## 1.2. Definición de Framework

El término “framework”, cuya traducción aproximada sería “marco de trabajo”, es un concepto que los que se dedican al desarrollo de software utilizan, conocen o, como mínimo, lo han escuchado alguna vez. Sin embargo, no es un concepto tan simple y mucho menos sencillo de definir. He aquí algunas definiciones:

- “…diseño abstracto orientado a objetos para un determinado tipo de aplicación, que se compone de una clase abstracta para cada componente principal del diseño; contendrá normalmente una librería de subclases que pueden ser utilizadas como componentes del diseño…” (Johnson, y otros, 1998)
- “…Es un conjunto cohesivo de interfaces y clases que colaboran para proporcionar los servicios de la parte central e invariable de un subsistema lógico…” (Larman, 2004)
- “…patrón arquitectónico que proporciona una plantilla extensible para aplicaciones dentro de un dominio…” (Booch, y otros, 1999)

En general, con el término framework, se hace alusión a una estructura de software compuesta por componentes personalizables e intercambiables para el desarrollo de una aplicación. Además está asociado a un determinado tipo de aplicaciones, lo que implica que su alcance esté acotado; o sea que está ligado a un dominio concreto. En otras palabras, un framework se puede considerar como una aplicación genérica incompleta y configurable a la que se puede añadir las últimas piezas para construir una aplicación concreta. Como todo software, es una mezcla de lo concreto y lo abstracto. Por último, es importante aclarar que no son ejecutables, para generar un ejecutable se debe instanciar el framework correspondiente al dominio que incluye al ejecutable.

### 1.2.1. Ventajas de los framework

Actualmente los frameworks revisten gran importancia para construir grandes sistemas de software. Su desarrollo está ganando rápidamente una amplia aceptación puesto que promueven la reutilización del diseño y del código fuente , simplifican el desarrollo de una aplicación encapsulando operaciones complejas en instrucciones sencillas y mediante la automatización de algunos patrones utilizados para resolver las tareas comunes, así como proporcionan estructura a todos los documentos y códigos fuentes del sistema, forzando al equipo de desarrollo a crear documentos y códigos más legibles y más fáciles de mantener. (Markiewicz, y otros, 2001)

La utilización de un framework en el desarrollo de una aplicación implica un cierto coste inicial de aprendizaje, aunque a largo plazo con seguridad facilitará tanto el desarrollo como el mantenimiento de la aplicación, además dinamiza el proceso de diseño y como los frameworks están alineados con los requisitos del negocio, el diseño se simplifica. Mejoran el desarrollo y las pruebas proporcionando los siguientes beneficios: mayor productividad del desarrollador, menos codificación, mayor consistencia, resultados inmediatos, mayor flexibilidad. Si se usan de forma adecuada el software puede responder fácilmente a los cambios en los requerimientos tanto del negocio como de tecnología. (Markiewicz, y otros, 2001)

### 1.2.2. Clasificación de los frameworks

En (Taligent, Corp, 1994) se da una clasificación de los frameworks, agrupándolos en tres categorías. La primera categoría propuesta son los “Frameworks de aplicación”, los cuales encapsulan una capa de funcionalidad horizontal que se puede aplicar en la construcción de una gran variedad de aplicaciones. Uno de los ejemplos más completos de este tipo de frameworks los constituyen los que implementan las interfaces gráficas de usuario (GUI). En otra categoría están los “Frameworks de soporte”, estos proporcionan servicios básicos a nivel de sistema. Por último están los “Frameworks de dominio”, dentro de los que se encuentra el framework que proponemos desarrollar en el presente trabajo. Los mismos son aplicables a un dominio de aplicación o a una línea de productos, tales como aplicaciones bancarias, tráfico, entre otras. Implementan una capa de funcionalidad vertical. Estos frameworks son y deben ser los más numerosos, y su evolución deberá ser también la más rápida, pues deben adaptarse a las áreas de negocio para las que fueron diseñados. Este

tipo de framework puede hacer uso de las dos categorías mencionadas anteriormente con el objetivo de proporcionar un marco de trabajo completo y robusto para el desarrollo de aplicaciones dentro del dominio para el cual fueron diseñados. Lo que permite a las organizaciones desarrollar software de más calidad y reducir el tiempo de salida al mercado de dicho producto, aumentando así la productividad.

Actualmente existen en el mercado numerosos frameworks y plataformas para crear aplicaciones, además de un gran número de herramientas orientadas a la capa de presentación, capa de negocio, capa de datos y/o servicios. Nos referiremos a continuación a algunos de los frameworks más usados orientados específicamente a la persistencia de datos, la inversión de control y la inyección de dependencias y explicaremos nuestra propuesta de incluir algunos de ellos en el desarrollo de nuestra solución.

### 1.2.3. Frameworks para la Inversión de Control y la Inyección de Dependencias.

**Patrón Inversión de Control (IoC):**

Delegamos a un componente o fuente externa, la función de seleccionar un tipo de implementación concreta de las dependencias de nuestras clases. En definitiva este patrón describe técnicas para soportar una arquitectura de tipo “plug-in” donde los objetos pueden buscar instancias de otros objetos que requieren y de los cuales dependen.

**Patrón Inyección de Dependencias (Dependence Injection, DI):**

Es realmente un caso especial de IoC. Es un patrón en el que se suplen objetos o dependencias a una clase en lugar de ser la propia clase quien cree los objetos o dependencias que necesita.

El objetivo es lograr un bajo acoplamiento entre los objetos de nuestra aplicación. Con este patrón de diseño, los objetos no crean o buscan sus dependencias (objetos con los cuales colabora) sino que éstas son dadas al objeto. El contenedor (la entidad que coordina cada objeto en el sistema) es el encargado de realizar este trabajo al momento de instanciar el objeto. Se invierte la responsabilidad en cuanto a la manera en que un objeto obtiene la referencia a otro objeto.

De esta manera, los objetos conocen sus dependencias por su interfaz. Así la dependencia puede ser intercambiada por distintas implementaciones a través del contenedor.

La IoC y la DI añaden flexibilidad y conllevan a "tocar" el menor código posible según avanza el proyecto, añaden comprensión y mantenibilidad al mismo.

El principio de "Única Responsabilidad" (Single Responsibility Principle) establece que cada Objeto debe tener una única responsabilidad. El concepto fue introducido por Robert C Martin y establece que una responsabilidad es una razón para cambiar y concluye diciendo que una clase debe tener solo una razón para cambiar. Este principio está ampliamente aceptado por la industria del desarrollo y en definitiva favorece a diseñar y desarrollar clases pequeñas con una única responsabilidad. Esto está directamente relacionado con el numero de dependencias (objetos de los que depende) cada clase. Haciendo uso de DI en el constructor, por sistema nos vemos obligados a declarar todas las dependencias de objetos en el constructor. Así pues, DI es también una forma de guía que nos conduce a realizar buenos diseños e implementaciones en el desarrollo, además de ofrecernos un desacoplamiento que podemos utilizar para inyectar diferentes ejecuciones de forma transparente.

En resumen, IoC y DI favorecen el desacoplamiento, brindan mayor flexibilidad, facilitan la mantenibilidad de nuestras aplicaciones y la realización de las pruebas unitarias.

**Unity**

Unity es actualmente el framework ligero de Microsoft más completo para implementar Inversión de Control e Inyección de Dependencias.

Algunas de sus características:

Proporciona un mecanismo para construir o ensamblar instancias de objetos, los cuales pueden contener otras instancias de objetos dependientes.

Se soporta jerarquía de contenedores. Un contenedor puede tener contenedores hijo, lo cual permite que las consultas de localización de objetos pasen de los contenedores hijo a los contenedores padre.

Se puede obtener la información de configuración de sistemas estándar de configuración, como ficheros XML, y utilizarlos para configurar el contenedor.

**Castle-Windsor**

Castle es un proyecto de código abierto que aspira a simplificar el desarrollo de aplicaciones empresariales. Es uno de los mejores frameworks para implementar Inversión de Control e Inyección de Dependencias. Ofrece un conjunto de herramientas (trabajo en conjunto o por separado) y la integración con otros proyectos de código abierto.

**Spring.Net**

Spring.Net es un framework de código abierto para la plataforma .NET basado en el framework Spring de Java. Es uno de los mejores frameworks con Programación Orientada a Aspectos. Ofreciendo también capacidades de Inversión de Control. Es un framework liviano y no intrusivo: generalmente los objetos que programamos no tienen dependencias en clases específicas de Spring. Proporciona apoyo integral de infraestructura para el desarrollo de aplicaciones empresariales sobre la plataforma .Net. Decidimos seleccionar el framework de Spring por su ventaja de ser de código abierto, la amplia gama de patrones que ofrece y su interrelación con los frameworks NHibernate y NUnit.

Una de las razones que justifican el uso de Spring.net es precisamente su trabajo con las transacciones. Como se ha mencionado, dicho framework proporciona un Administrador de Transacciones el cual tiene las siguientes ventajas:

Garantiza un amplio manejo transaccional a través de las diferentes APIs (Application Programming Interface) de este tipo tales como ADO.NET, Enterprise Services, System.Transactions, and NHibernate.

Cuenta con instrucciones para el manejo de transacciones que soportan cualquiera de las tecnologías antes mencionadas.

Proporciona una simple interfaz con las funcionalidades que abarcan la mayoría de los eventos que se pueden encontrar durante el trabajo con transacciones.

**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, y Service Locator por solo mencionar algunos.

Inversión del Control (IoC): 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).

### 1.2.4. Frameworks para la persistencia de datos.

Utilizar un marco de trabajo de Mapeo Relacional de Objetos (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.

Utilizar un framework ORM 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 mapeo 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, entre otros. simplemente cambiando la configuración correspondiente.

Entity framework y NHibernate son dos de los frameworks más usados para la persistencia de datos.

**Entity Framework**

Entity Framework es un conjunto de tecnologías de ADO.NET que permiten el desarrollo de aplicaciones de software orientadas a datos. Los arquitectos y programadores de aplicaciones orientadas a datos se han enfrentado a la necesidad de lograr dos objetivos muy diferentes. Deben modelar las entidades, las relaciones y la lógica de los problemas empresariales que resuelven, y también deben trabajar con los motores de datos que se usan para almacenar y recuperar los datos. Los datos pueden abarcar varios sistemas de almacenamiento, cada uno con sus propios protocolos; incluso las aplicaciones que funcionan con un único sistema de almacenamiento deben equilibrar los requisitos del sistema de almacenamiento con respecto a los requisitos de escribir un código de aplicación eficaz y fácil de mantener.

Entity Framework permite a los programadores trabajar con datos en forma de objetos y propiedades específicos del dominio, por ejemplo, con clientes y direcciones, sin tener que pensar en las tablas de las bases de datos subyacentes y en las columnas en las que se almacenan estos datos. Con Entity Framework, los desarrolladores de software pueden trabajar en un nivel más alto de abstracción cuando tratan con datos, y puede crear y

mantener aplicaciones orientadas a datos con menos código que en las aplicaciones tradicionales.

**NHibernate**

La utilización de este framework en la solución está supeditada al uso de Spring.net puesto que ambos se encuentran integrados.

NHibernate es un framework de código abierto para el mapeo de objetos a bases de datos relacionales. Es desarrollado activamente, completo y se utiliza en miles de proyectos exitosos. Nace producto de la conversión de Hibernate de lenguaje Java a C# para su integración en la plataforma .NET. Se maneja la persistencia de datos.NET desde y hacia una base de datos relacional subyacente. Dada una descripción XML de sus entidades y relaciones, NHibernate genera automáticamente SQL para cargar y almacenar los objetos. Si lo desea, puede describir los metadatos de mapeo de atributos en el código fuente.

NHibernate soporta persistencia transparente, sus clases de objetos no tienen que seguir un modelo de programación restrictiva. Las clases persistentes no son necesarias para implementar cualquier interfaz o heredar de una clase base especial. Esto hace posible diseñar la lógica de negocio utilizando objetos simples de .NET y lenguaje orientado a objetos.

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

Además de que NHibernate le saca cierta ventaja a Entity Framework porque lleva más años en el mercado, es una tecnología más probada y evolucionada, Entity Framework es un software privativo por lo que no permite su extensibilidad ni modificar algunos de los comportamientos que ofrece. Como ya señalamos anteriormente, es NHibernate es software libre, distribuido bajo los términos de la LGPL (Licencia Pública General Menor de GNU). Por lo que decidimos incluir en el diseño de nuestro framework el uso de NHibernate, unido a que está desarrollado especialmente para la plataforma .Net, sobre la cual proponemos desarrollar la solución propuesta.

## 1.3. Arquitectura de Software

A continuación se exponen conceptos fundamentales relacionados con la arquitectura del Proyecto Registros y Notarias Fase II, a la cual el framework debe sustentar.

### 1.3.1. Relación de abstracción entre estilos y patrones

Los estilos arquitectónicos son la herramienta básica de un arquitecto a la hora de dar forma a la arquitectura de una aplicación. Un estilo arquitectónico se puede entender como un conjunto de principios que definen a alto nivel un aspecto de la aplicación y viene definido por un conjunto de componentes, un conjunto de conexiones entre dichos componentes y un conjunto de restricciones sobre cómo se comunican dos componentes cualesquiera conectados. Estos pueden organizarse en torno al aspecto de la aplicación sobre el que se centran. Los principales aspectos son: comunicaciones, despliegue, dominio, interacción y estructura.

Lo normal en una arquitectura es que no se base en un solo estilo arquitectural, sino que combine varios de dichos estilos arquitecturales para obtener las ventajas de cada uno. Es importante entender que los estilos arquitecturales son indicaciones abstractas de cómo dividir en partes el sistema y de cómo esas partes deben interactuar. En las aplicaciones reales los estilos arquitectónicos se "instancian" en una serie de componentes e interacciones concretas. Esa es la principal diferencia existente entre un estilo arquitectural y un patrón de diseño. Los patrones de diseño son descripciones estructurales y funcionales de cómo resolver de forma concreta un determinado problema mediante orientación a objetos, mientras que los estilos arquitecturales son descripciones abstractas y no están atados a ningún paradigma de programación específico.

### 1.3.2. Estilos arquitectónicos.

**En Capas**

Se basa en una distribución jerárquica de los roles y las responsabilidades para proporcionar una división efectiva de los problemas a resolver. Los roles indican el tipo y la forma de interacción con otras capas y las responsabilidades la funcionalidad que implementan.

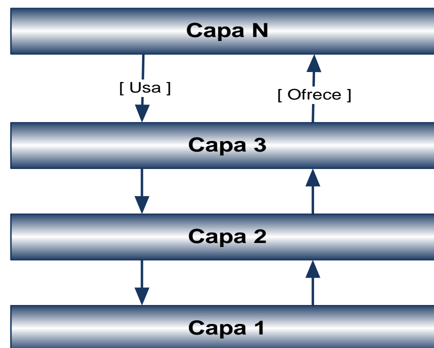



**Figura 1.1** Arquitectura en Capas

Algunos de sus beneficios:

- Atracción ya que los cambios se realizan a alto nivel y se puede incrementar o reducir el nivel de abstracción que se usa en cada capa del modelo.
- Aislamiento ya que se pueden realizar actualizaciones en el interior de las capas sin que esto afecte al resto de sistema.
- Rendimiento ya que distribuyendo las capas en distintos niveles físicos se puede mejorar la escalabilidad, la tolerancia a fallos y el rendimiento.

Es recomendable usar este estilo cuando las aplicaciones son complejas y el alto nivel de diseño requiere la separación para que los distintos equipos puedan concentrarse en distintas áreas de funcionalidad, cuando ya se tiene construidas capas de una aplicación anterior, que pueden reutilizarse o integrarse o cuando ya se tiene aplicaciones que expongan su lógica de negocio a través de interfaces de servicio.

**Orientada al Dominio**

Se basa sobre todo en la importancia del Dominio del Negocio, sus elementos y comportamientos y las relaciones entre ellos. Está muy indicado para diseñar e implementar aplicaciones empresariales complejas.

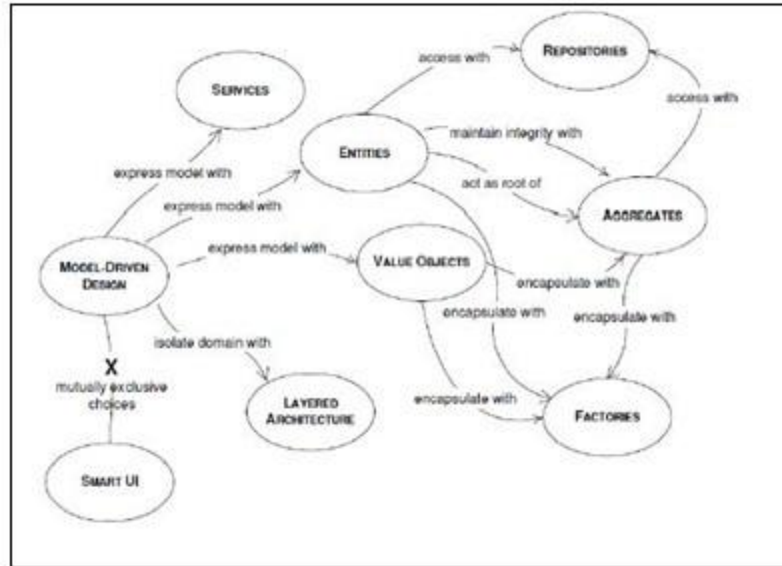



**Figura 1.2** Arquitectura Orientada al Dominio

Algunos de sus beneficios:

- **Comunicación:** Todas las partes de un equipo de desarrollo pueden usar el modelo de dominio y las entidades que define para comunicar conocimiento del negocio y requerimientos, haciendo uso de un lenguaje común.
- **Extensibilidad:** La Capa del Dominio es el corazón del software y por lo tanto estará completamente desacoplada de las capas de infraestructura, siendo más fácil así extender y evolucionar la tecnología del software.
- **Mejora las Pruebas:** Facilita las pruebas y los cambios, debido a que la tendencia de diseño es a desacoplar los objetos de las diferentes capas de la arquitectura, lo cual facilita los cambios y las pruebas.

Es recomendable usar este estilo en aplicaciones complejas con mucha lógica de negocio, con Dominios complejos y donde se desea mejorar la comunicación y minimizar los malos entendidos en la comunicación del equipo de desarrollo. Es también una aproximación ideal escenarios empresariales grandes y complejos que son difíciles de manejar con otras técnicas.

**Estilo arquitectónico del Proyecto Registros y Notarías Fase II**

El estilo arquitectónico que se seleccionó en el Proyecto Registros y Notarías Fase II es el de "Arquitectura en N-Capas Orientada al dominio". Esta Arquitectura está diseñada para un tipo

concreto de aplicaciones: 'Aplicaciones empresariales complejas', con una vida relativamente larga y normalmente con un volumen de cambios evolutivos considerables. Por lo tanto, en estas aplicaciones es muy importantes todo lo relativo al mantenimiento de la aplicación, facilidad de actualización o sustitución de tecnologías y frameworks por otras versiones más modernas o incluso por otros diferentes y que todo esto se pueda realizar con el menor impacto posible sobre el resto de la aplicación. En definitiva, que los cambios de tecnologías de infraestructura de una aplicación no afecten a capas de alto nivel de la aplicación, especialmente que afecten el mínimo posible a la capa del "Dominio de la Aplicación".

En las aplicaciones complejas, el comportamiento de las reglas del negocio (lógica del Dominio) está sujeto a muchos cambios y es muy importante poder modificar, construir y realizar pruebas sobre dichas capas de lógica del dominio de una forma fácil e independiente. Debido a esto, un objetivo importante es tener el mínimo acoplamiento entre el Modelo del Dominio (lógica y reglas del negocio) y el resto de las capas (Capas presentación, persistencia de datos, entre otras.).

Así pues, las razones por las que es importante hacer uso de una "Arquitectura N- Capas Orientada al Dominio" es especialmente en los casos donde el comportamiento del negocio a automatizar (lógica del dominio) está sujeto a muchos cambios y evoluciones. En este caso disponer de un "Modelo de Dominio" disminuirá el coste total de dichos cambios y a medio plazo, el coste total de la propiedad será mucho menor que si la aplicación hubiera sido desarrollada de una forma más acoplada, porque los cambios no tendrán tanto impacto. En definitiva, el tener todo el comportamiento del negocio que puede estar cambiando encapsulado en una única área del software, disminuye drásticamente la cantidad de tiempo que se necesita para realizar un cambio, porque será realizado en un solo sitio y podrá ser convenientemente probado de forma aislada, aunque esto, por supuesto dependerá de cómo se halla desarrollado. El poder aislar tanto como sea posible dicho código del Modelo del Dominio disminuye las posibilidades de tener que realizar cambios en otras áreas de la aplicación lo cual puede afectar con nuevo problemas. Esto es de vital importancia si se desea reducir y mejorar los ciclos de estabilización y puesta en producción de las soluciones.

### 1.3.3. Patrones arquitectónicos.

Los patrones arquitectónicos son la descripción de un problema particular y recurrente de diseño, que aparece en contextos de diseño específicos 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 (Buschmann, y otros, 1996). Aunque existen varios patrones arquitectónicos solo haremos referencia a dos de ellos por ser los únicos que a los efectos de la investigación tienen mayor significación.

**Modelo-Vista-Controlador:**

Divide una aplicación interactiva en tres componentes. El modelo contiene la información central y los datos. Las vistas despliegan información al usuario. Los controladores capturan la entrada del usuario. Las vistas y los controladores constituyen la interfaz del usuario.

**Atributos Asociados:**

- Funcionalidad
- Mantenibilidad

**Atributos en Conflictos:**

- Desempeño
- Portabilidad

**Modelo-Vista-Presentación**

El patrón arquitectónico "Modelo-Vista-Presentación" (MVP), es una evolución de MVC y el Forms&Controller, tratando de tomar lo mejor de cada uno de ellos, las ideas subyacentes de MVP son muy parecidas a las del MVC. De cualquier forma que MVP es diseñado para ser más utilizable y comprensible. MVP tiene mucho menos implementaciones que MVC, una implementación conocida MVP para .NET es "Microsoft User Interface Process Application Block" (UIPAB).

### 1.3.4. Atributos de Calidad

Según (Barbacci, y otros, 2005) 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, y otros, 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, y otros, 2005).

### Atributos de Calidad deseables en una aplicación:

**Disponibilidad:** Es la medida de disponibilidad del sistema para el uso (Barbacci, y otros, 2005).

**Confidencialidad:** Es la ausencia de acceso no autorizado a la información (Barbacci, y otros, 2005).

**Funcionalidad:** Habilidad del sistema para realizar el trabajo para el cual fue concebido (Kazman, y otros, 2001).

**Desempeño:** 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, 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 (Kazman, y otros, 2001), se refiere además a la cantidad de comunicación e interacción existente entre los componentes del sistema.

**Confiabilidad:** Es la medida de la habilidad de un sistema a mantenerse operativo a lo largo del tiempo (Barbacci, y otros, 2005).

**Seguridad externa:** Ausencia de consecuencias catastróficas en el ambiente. Es la medida de ausencia de errores que generan pérdidas de información (Barbacci, y otros, 2005).

**Seguridad Interna:** 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 (Barbacci, y otros, 2005).

**Configurabilidad:** Posibilidad que se otorga a un usuario experto a realizar ciertos cambios al sistema (Booch, y otros, 1999).

**Integrabilidad:** Es la medida en que trabajan correctamente componentes del sistema que fueron desarrollados separadamente al ser integrados. (Bass, y otros, 1998).

**Integridad:** Es la ausencia de alteraciones inapropiadas de la información (Barbacci, y otros, 2005).

**Interoperabilidad:** 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, y otros, 1998).

**Modificabilidad:** Es la habilidad de realizar cambios futuros al sistema. (Booch, y otros, 1999).

**Mantenibilidad:** Es la capacidad de someter a un sistema a reparaciones y evolución (Barbacci, y otros, 2005) Capacidad de modificar el sistema de manera rápida y a bajo costo (Booch, y otros, 1999).

**Portabilidad:** 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, y otros, 2001).

**Reusabilidad:** 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, y otros, 1998)

**Escalabilidad:** Es el grado con el que se pueden ampliar el diseño arquitectónico, de datos o procedimental (Pressman, 2005)

**Capacidad de Prueba:** 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, y otros, 1998).

## 1.4. Patrones de Diseño

El planteamiento de formalizar soluciones a distintas situaciones, de modo que puedan ser entendidas por otros profesionales, es a lo que se llama patrones. Por tanto, un patrón no es más que la descripción detallada de una solución adecuada a un problema concreto.

Un patrón es un modelo que podemos seguir para realizar algo. Los patrones surgen de la experiencia de seres humanos de tratar de lograr ciertos objetivos. Capturan la experiencia existente y probada para promover buenas prácticas. (Oktaba). Los patrones permiten y han permitido en diferentes áreas del conocimiento humano rehusar la esencia de la solución de un problema al enfrentar nuevos problemas similares.

A continuación se realizará un estudio de los diferentes patrones de diseño más utilizados en la actualidad, se expondrá en qué consiste cada uno para facilitar su comprensión y los beneficios que aporta a nuestra aplicación la utilización de los mismos.

Un patrón de diseño nombra, abstrae e identifica los aspectos claves de un diseño estructurado común, que lo hace útil para la creación de diseños orientados a objetos reutilizables. Los patrones de diseño identifican las clases participantes y las instancias, sus papeles y colaboraciones, y la distribución de responsabilidades. Cada patrón de diseño se enfoca sobre un particular diseño orientado a objetos. Se describe cuándo se aplica, las características de otros diseños y las consecuencias y ventajas de su uso. (Martínez Juan) Un patrón de diseño es una descripción de clases y objetos comunicándose entre sí, adaptada para resolver un problema de diseño general en un contexto particular. Los patrones de diseño se pueden clasificar atendiendo a dos criterios, el primero el propósito, refleja que hace un patrón. El segundo el alcance, especifica si el patrón se aplica especialmente a las clases o a los objetos. Esta clasificación se muestra en la tabla (VER ANEXO 6)

**Patrones GRASP**

Los patrones GRASP describen los principios fundamentales de la asignación de responsabilidades a objetos, expresados en forma de patrones. El nombre se eligió para indicar la importancia de captar estos principios, si se quiere diseñar eficazmente el software orientado a objetos. Los patrones GRASP básicos se refieren a cuestiones y aspectos fundamentales del diseño, ellos son:

**Experto:** Asignar una responsabilidad al experto de información: la clase que cuenta con la información necesaria para cumplir la responsabilidad. Expresa simplemente la intuición de que los objetos hacen cosas relacionadas con la información que poseen. Este patrón permite conservar el encapsulamiento, ya que los objetos se valen de su propia información para hacer lo que se les pide. Esto soporta un bajo acoplamiento. Este patrón propicia además que el comportamiento se distribuya entre las clases que cuentan con la información requerida, alentando con ello definiciones de clases sencillas y más cohesivas, o sea que además brinda soporte a una alta cohesión.

**Creador:** Este patrón guía la asignación de responsabilidades relacionadas con la creación de objetos. El propósito fundamental de este patrón es encontrar un creador que debemos

conectar con el objeto producido en cualquier evento. Este patrón brinda soporte a un bajo acoplamiento.

**Bajo Acoplamiento:** Este es un principio que se debe recordar siempre que se vaya a diseñar algo. Este patrón estimula asignar una responsabilidad de modo que su colocación no incremente el acoplamiento tanto que produzca los resultados negativos propios de un alto acoplamiento. El bajo acoplamiento soporta el diseño de clases más independientes, que reducen el impacto de los cambios y también reutilizables, que acrecientan la oportunidad de una mayor productividad.

**Alta cohesión:** Este es también un principio que se debe tener en cuenta en todas las decisiones de diseño. (Booch, y otros, 1999) Señala que se da una alta cohesión cuando los elementos de un componente, una clase por ejemplo, "colaboran para producir algún comportamiento bien definido". Una clase de alta cohesión posee un número relativamente pequeño de responsabilidades, con una importante funcionalidad relacionada y poco trabajo por hacer. Colabora con otros objetos para compartir el esfuerzo si la tarea es grande. Este patrón mejora la calidad y facilidad con que se puede entender el diseño, se genera un bajo acoplamiento y permite fomentar la reutilización.

**Controlador:** La mayor parte de los sistemas reciben eventos de entrada externa, los cuales generalmente incluyen una interfaz gráfica para el usuario (IGU) operado por una persona. En estos casos hay que elegir un controlador que maneje esos eventos de entrada. Este patrón se utiliza porque las operaciones del sistema deberían manejarse en la capa de dominio de los objetos, y no en las de interfaz, presentación o aplicación.

## 1.5. Metodologías de desarrollo de software

Hoy en día el aumento de la exigencia de la industria del software en términos de productividad, calidad y mantenibilidad de los sistemas, ha llevado a las organizaciones a la necesidad de reducir el esfuerzo en el desarrollo del proyecto, el tiempo en el ciclo de vida y el costo sin afectar la calidad, además de satisfacer los requerimientos y los plazos de entrega exigidos por los clientes. (García Ávila, 2000) Quedando así evidenciado la necesidad de la implantación de metodologías de desarrollo con el fin de solucionar la problemática que resulta de la escasa documentación de los sistemas y de la falta de comunicación con los usuarios durante el proceso de desarrollo, lo que genera productos que no responden totalmente a las necesidades de los usuarios. En este sentido, la ingeniería de

software define la necesidad de utilizar un conjunto de métodos, procedimientos, técnicas y herramientas que facilitan la construcción de un sistema informático en el tiempo requerido y con calidad.

(Pressman, 2005) Plantea que en el proceso de desarrollo de software lo que ocurre es que:

- No se utilizan eficazmente las metodologías modernas de desarrollo del software ni las herramientas de Ingeniería del Software asistida por computadora (CASE), que son más importantes que el hardware más moderno para conseguir una buena calidad y productividad.
- No existe una adecuada descripción formal y detallada del ámbito de la información, funciones, rendimiento, interfaces, ligaduras de diseño y criterios de validación por una mala comunicación entre clientes y analistas.
- No se recogen datos sobre el proceso de desarrollo del software.
- No se garantiza la calidad desde el inicio del proyecto, ni se aplica la revisión técnica formal que es un filtro de calidad muy efectivo para encontrar defectos en el software.

Todos estos problemas se pueden evitar realizando una buena gestión de los proyectos de software, lo que se traduce en la selección de una buena metodología.

### 1.5.1. Características deseables de una metodología

- Existencia de reglas predefinidas.
- Cobertura total del ciclo de desarrollo.
- Planificación y control.
- Comunicación efectiva.
- Utilización sobre un abanico amplio de proyectos.
- Fácil formación.
- Herramientas CASE.
- Actividades que mejoren el proceso de desarrollo.
- Soporte al mantenimiento.
- Soporte de la reutilización de software. (García Rubio, y otros, 2007)

### 1.5.2. Ventajas de tener una buena metodología

Mejora de los procesos de desarrollo.

- Todas las personas del proyecto trabajan bajo un marco común.
- Estandarización de conceptos, actividades y nomenclaturas.
- Actividades de desarrollo apoyadas por procedimientos y guías.
- Resultados del desarrollo predecibles.
- Uso de herramientas de ingeniería de software.
- Planificación de actividades en base a un conjunto de tareas definidas y a la experiencia en otros proyectos.
- Recopilación de mejores prácticas para proyectos futuros.

Mejora de los productos.

- Se asegura que los productos cumplen con los objetivos de calidad propuestos.
- Detección temprana de errores.
- Se garantiza la trazabilidad de los productos a lo largo del desarrollo.

Mejora de las relaciones con el cliente.

- El cliente percibe el orden en los procesos.
- Facilita al cliente el seguimiento de la evolución del proyecto.
- Se establecen mecanismos para asegurar que los productos desarrollados cumplen con las expectativas del cliente. (López Barrio, 2005)

Existen en la actualidad múltiples metodologías que pueden dividirse en dos categorías: Tradicionales y Ágiles. A continuación se realizará un estudio de las metodologías que se usan en el desarrollo de software, y luego de tener un conocimiento previo se hará la selección de la metodología que guiará el proceso de desarrollo del presente trabajo.

### 1.5.3. Metodologías Tradicionales

Las metodologías tradicionales se centran especialmente en el control del proceso, estableciendo rigurosamente las actividades involucradas, los artefactos que se deben

producir, los roles, las herramientas y notaciones que se usarán, incluyendo modelado y documentación detallada. Estas propuestas han demostrado ser efectivas y necesarias en un gran número de proyectos, pero también han presentado problemas en otros muchos. Estas metodologías se caracterizan por ser rígidas y dirigidas por la documentación que se genera en cada una de las actividades desarrolladas. Este esquema "tradicional" para abordar el desarrollo de software es efectivo en proyectos de gran envergadura donde por lo general la gestión es el principal problema, por lo que se necesita un proceso gestionado de forma estricta.

**Ejemplos de metodologías tradicionales:**

- **SPICE:** metodología de desarrollo en la que se definen un conjunto de buenas prácticas para la mejora gradual de los procesos de ingeniería.
- **Rational Unified Process (RUP):** define un ciclo de vida iterativo priorizando el uso de lenguajes de modelado, casos de uso y centrado en la arquitectura.
- **Microsoft Solution Framework (MSF):** metodología flexible e interrelacionada con una serie de conceptos, modelos y prácticas de uso. Se centra en los modelos de proceso y de equipo dejando en un segundo plano las elecciones tecnológicas. (López Barrio, 2005)

### 1.5.4. Metodologías Ágiles

Las metodologías ágiles están especialmente orientadas para proyectos pequeños. Constituyen una solución con una elevada simplificación, a pesar de ello no renuncian a las prácticas esenciales para asegurar la calidad del producto. Las características de los proyectos para los cuales las metodologías ágiles han sido especialmente pensadas se ajustan a un amplio rango de proyectos industriales de desarrollo de software; aquellos en los cuales los equipos de desarrollo son pequeños, con plazos reducidos, requisitos volátiles y nuevas tecnologías. El Manifiesto Ágil es un documento que resume la filosofía ágil.

**Ejemplos de Metodologías Ágiles:**

- **XP o Programación Extrema:** metodología ligera de desarrollo de software, que surge como respuesta y posible solución a los problemas derivados del cambio en los requerimientos.

- **Scrum:** metodología ágil que se basa en la práctica del desarrollo iterativo e incremental. Aumenta significativamente la productividad y reduce el tiempo de desarrollo. Está pensada para equipos de desarrollo pequeños. Define ciclos de desarrollo o *sprint* y roles como *ScrumMaster, ProductOwner* y *stakeholders.*

  **Agile Unified Process:** es una versión simplificada del Rational Unified Process (RUP), describe de una manera simple y fácil de entender la forma de desarrollar aplicaciones de software de negocio usando técnicas ágiles y conceptos que aún se mantienen válidos en RUP.

### 1.5.5. Rational Unified Process (RUP)

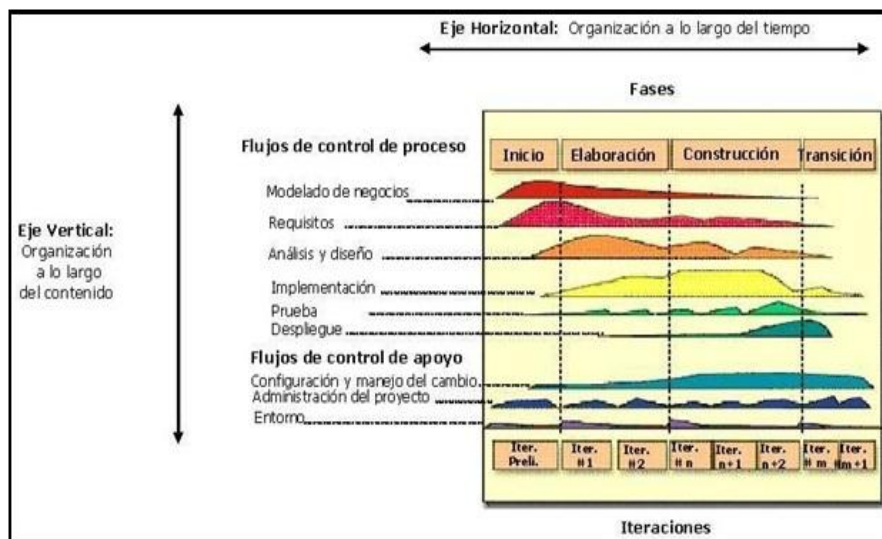



**Figura 1.3** RUP

Como ilustra la Figura 1.3 el Proceso Unificado de está pensado en dos dimensiones. El mismo se repite a lo largo de una serie de ciclos que constituyen la vida de un sistema. Cada ciclo concluye con una versión del producto para los clientes. Cada ciclo consta de cuatro fases. Cada fase termina con un hito. (Rumbaugh, y otros, 2004)

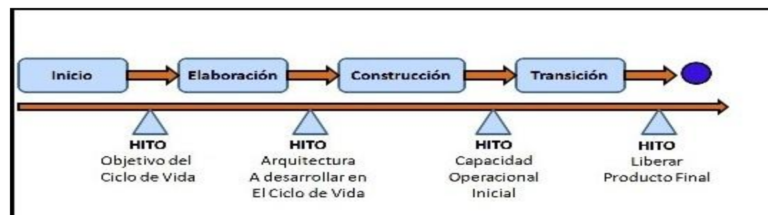



**Figura 1.4** Hitos y Fases de RUP

Como se observa en la figura 1.4 en RUP se han agrupado las actividades en grupos lógicos definiéndose 9 flujos de trabajo principales. Los 6 primeros son conocidos como flujos de ingeniería y los tres últimos como de apoyo.

El RUP está basado en componentes. Utiliza el estándar de modelado visual, el Lenguaje Unificado de Modelado (UML), y sus tres pilares fundamentales son que es un proceso dirigido por casos de uso, centrado en la arquitectura e iterativo e incremental.

**Un proceso dirigido por casos de uso**

Los Casos de Uso son una técnica de captura de requisitos que fuerza a pensar en términos de importancia para el usuario y no sólo en términos de funciones que sería bueno contemplar. Se define un Caso de Uso como un fragmento de funcionalidad del sistema que proporciona al usuario un valor añadido. Los Casos de Uso representan los requisitos funcionales del sistema. (Kruchten, 2000)

Los casos de uso no son solo una herramienta para especificar los requisitos de un sistema. También guían su diseño, implementación y prueba; guían el proceso de desarrollo. Basándose en el modelo de casos de uso, los desarrolladores crean una serie de modelos de diseño e implementación que llevan a cabo los casos de uso. Los desarrolladores revisan cada uno de los sucesivos modelos para que sean conformes al modelo de casos de uso. Los ingenieros de prueba prueban la implementación para garantizar que los componentes del modelo de implementación implementan correctamente los casos de uso. De este modo los casos de uso no solo inician el proceso de desarrollo, sino que les proporcionan un hilo conductor. (Rumbaugh, y otros, 2004)

**Proceso centrado en la arquitectura**

La arquitectura de un sistema es la organización o estructura de sus partes más relevantes, lo que permite tener una visión común entre todos los involucrados (desarrolladores y usuarios) y una perspectiva clara del sistema completo, necesaria para controlar el desarrollo. (Kruchten, 2000) Podemos afirmar que RUP está centrado en la arquitectura, y esto se debe fundamentalmente a que los casos de uso solamente no son suficientes. Se necesitan más cosas para conseguir un sistema de trabajo. Cada producto tiene tanto una función como una forma. Ninguna es suficiente por sí misma. Estas dos fuerzas deben equilibrarse para obtener un producto con éxito. En esta situación, la función corresponde a los casos de uso y la forma

a la arquitectura. La arquitectura nos da una clara perspectiva del sistema completo, necesaria para controlar el desarrollo. Se necesita una arquitectura para: comprender el sistema, organizar el desarrollo, fomentar la reutilización y hacer evolucionar el sistema. El desarrollo de un producto software comercial supone un gran esfuerzo que puede durar entre varios meses hasta posiblemente un año o más. Es práctico dividir el trabajo en partes más pequeñas o mini-proyectos.

**¿Por qué un desarrollo iterativo e incremental?**

Para obtener un software mejor. Para cumplir los hitos principales y secundarios con los cuales se controla el desarrollo.

Los conceptos - dirigido por casos de uso, centrado en la arquitectura e iterativo e incremental- son de igual importancia. La arquitectura proporciona la estructura sobre la cual guiar las iteraciones, mientras que los casos de uso definen los objetivos y dirigen el trabajo de cada iteración. La eliminación de una de las tres ideas reduciría drásticamente el valor de esta metodología. Es como un taburete de tres patas. Sin una de ellas el taburete se cae. RUP identifica seis de las llamadas mejores prácticas con las que define una forma efectiva de trabajar para los equipos de desarrollo de software. Ellas son:

**Desarrollo de software iterativo:** Para los sofisticados sistemas de software que se desarrollan en la actualidad es necesario un enfoque iterativo, que permita una mayor comprensión del problema a través de sucesivos refinamientos, y lograr una solución eficaz a través de múltiples iteraciones. Este enfoque iterativo del desarrollo propicia que en cada etapa del ciclo de desarrollo se aborden los temas de mayor riesgo, reduciendo significativamente el perfil de riesgo del proyecto.

**Administrar requerimientos:** El Proceso Unificado describe como licitar, organizar, documentar y seguir los cambios de las funcionalidades y restricciones requeridas. Las nociones de caso de uso y escenario han demostrado ser una excelente vía de capturar los requerimientos funcionales y de asegurarnos que estos facilitan el diseño, la implementación y las pruebas; lo que hace más probable que el sistema final satisfaga las necesidades de los usuarios finales.

**Desarrollo basado en componentes:** El Proceso Unificado apoya el desarrollo de software basado en componentes. Los componentes son módulos no triviales, subsistemas que

cumplen una clara función. El Proceso Unificado proporciona un enfoque sistemático para la definición de la arquitectura utilizando componentes tanto nuevos como existentes, estos son unidos en una arquitectura bien definida. Esta característica en un proceso de desarrollo permite que el sistema se vaya creando a medida que se obtienen o se desarrollan sus componentes.

**Modelado Visual:** Las abstracciones visuales ayudan a comunicar los diferentes aspectos del software, ver cómo los elementos del sistema se comunican entre sí, asegurarse que los componentes están acordes con el código, mantener la coherencia entre el diseño y la implementación, y promover la comunicación inequívoca.

**Verificación continua de la calidad:** Poco rendimiento de las aplicaciones y poca fiabilidad son algunos de los factores que inhiben la aceptación de las aplicaciones de software actuales. Por lo tanto, la calidad debe revisarse con respecto a los requisitos sobre la base de la fiabilidad, funcionalidad, el rendimiento de la aplicación y el rendimiento del sistema. La evaluación de la calidad se construye en el proceso, en todas las actividades, con la participación de todos los miembros del equipo, utilizando criterios y mediciones objetivas, y que no sean tratados como una ocurrencia tardía o una actividad realizada por un grupo separado.

**Gestión de los cambios:** Poder realizar el seguimiento de los cambios es esencial en un entorno en el que el cambio es inevitable. El Proceso Unificado de Desarrollo describe como controlar, seguir y supervisar los cambios para lograr un desarrollo iterativo exitoso. Proporciona una guía para establecer espacios de trabajo seguros para cada desarrollador, proporcionando aislamiento de los cambios realizados en otros lugares de trabajo y controlando los cambios en todos los artefactos de software.

### 1.5.6. XP

Es una metodología ligera de desarrollo de software, que surge como respuesta y posible solución a los problemas derivados del cambio en los requerimientos. En la mayoría de los casos se plantea como una metodología a emplear en los proyectos con riesgos de requisitos muy cambiantes. Se plantea que «Todo en el software cambia. Los requisitos cambian. El diseño cambia. El negocio cambia. La tecnología cambia. El equipo cambia. Los miembros del equipo cambian. El problema no es el cambio en sí mismo, puesto que se sabe que el

cambio va a suceder; el problema es la incapacidad de adaptarse a dicho cambio cuando éste tiene lugar». XP es una metodología orientada al cliente y de iteraciones cortas.

**Lo fundamental de XP es:**

- La comunicación: Entre los usuarios y los desarrolladores.
- La simplicidad: Al desarrollar y codificar los módulos del sistema.
- La retroalimentación: Concreta y frecuente del equipo de desarrollo, el cliente y los usuarios finales.

### Fases de la metodología XP (VER ANEXO 7)

XP es un proceso muy orientado a la implementación, en el que se genera poca documentación y en que la funcionalidad exacta del sistema final no se define nunca formal y contractualmente. Es por eso que este método es más aplicable para desarrollos internos.

### 1.5.7. Justificación de la metodología a utilizar.

Después de estudiar ambas metodologías se decidió seleccionar la metodología RUP para guiar el proceso de desarrollo, pues a pesar de ser una metodología tradicional, o sea, rígida y dirigida por la documentación que se genera en cada una de las actividades desarrolladas, es una metodología muy exitosa y utilizada actualmente en proyectos de gran envergadura, además posee muchas características que ayudaron a su selección, entre ellas podemos citar:

- Establece rigurosamente las actividades involucradas, los roles, las herramientas y notaciones que se usarán, incluyendo modelado y documentación detallada, así como los artefactos que se deben producir, esto último la convierte en una de las metodologías más importantes para alcanzar un grado de certificación en el desarrollo del software.
- Este proceso tiene tres características que lo hacen único: Dirigido por casos de uso, centrado en la arquitectura e iterativo e incremental.
- Contempla seis de las llamadas buenas prácticas para el desarrollo de un software.
- Aumenta la productividad del equipo, pues proporciona un acceso común a todos los miembros del proyecto al mismo conocimiento base, por lo que sin importar en qué

fase del Proceso Unificado se encuentre algún miembro trabajando todos comparten el mismo lenguaje, proceso y visión de cómo desarrollar software.

- Se emplea fundamentalmente en proyectos que se desarrollarán a largo plazo.
- Es una metodología orientada al proceso.

Como el objetivo general de este trabajo de diploma es realizar las etapas de Requisitos y Diseño de un framework para la arquitectura de los sistemas registrales del proyecto Registros y Notarias Fase II, se decidió seleccionarla en lugar de XP, pues este último es un proceso muy orientado a la implementación, por lo que para lograr un análisis y diseño exitoso esta es más adecuada.

Con respecto a la metodología RUP el Dr. Xavier Ferré Grau (Ferrer Grau, 2005) en su tesis de doctorado afirma que el proceso que está tomando mayor atención en la Ingeniería de Software en la actualidad es el Proceso Unificado y que esto es debido a que sus impulsores son los mejores metodólogos del desarrollo orientado a objetos de la década del 90, James Rumbaugh, Ivar Jacobson y Grady Booch. Sostiene además que adopta un verdadero enfoque iterativo y que es el más aplicado en proyectos reales. Alega que Rational Unified Process (RUP) es una particularización del modelo de proceso representado por el Proceso Unificado.

Junto al Lenguaje Unificado de Modelado (UML) constituye la metodología estándar más utilizada para el análisis, implementación y documentación de sistemas orientados a objetos. (Tejera Hernández, y otros, 2007)

## 1.6. El Lenguaje Unificado de Modelado

El modelado visual es el proceso de tomar la información de un modelo y representarla gráficamente utilizando un conjunto de elementos gráficos estándares. El modelado visual facilita la comunicación entre usuarios, desarrolladores, analistas, probadores, gerentes y todos los participantes en el proyecto. (Boggs, y otros, 2002)

El Lenguaje Unificado de Modelado (UML) es un lenguaje de modelado visual que se usa para especificar, visualizar, construir y documentar artefactos de un sistema de software. Captura decisiones y conocimiento sobre los sistemas que se deben construir. Se usa para entender, diseñar, hojear, configurar, mantener, y controlar la información sobre tales sistemas. Está pensado para usarse con todos los métodos de desarrollo, etapas del ciclo de

vida, dominios de aplicación y medios. El UML cumple con la función de servir de enlace entre quien tiene la idea del sistema y el desarrollador, ya que le ayuda a capturar la idea de un sistema para comunicarla posteriormente a quien está involucrado en su proceso de desarrollo; esto se lleva a cabo mediante un conjunto de símbolos y diagramas. Cada diagrama tiene fines distintos dentro del proceso de desarrollo.

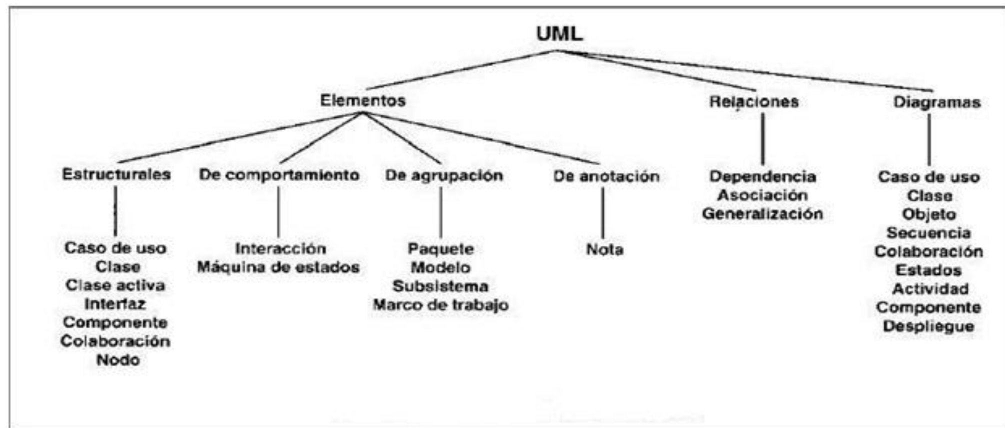



**Figura 1.5** UML

UML proporciona una organización en el proceso de diseño de forma tal que analistas, clientes, desarrolladores y otras personas involucradas en el desarrollo del sistema lo comprendan y convengan con él. El UML es un lenguaje para construir modelos; no guía al desarrollador en la forma de realizar análisis y diseño orientado a objetos, ni le indica cuál proceso de desarrollo adoptar. Los autores del lenguaje UML -Booch, Jacobson y Rumbaugh- hicieron un excelente servicio a la comunidad de la tecnología orientada a objetos al crear un lenguaje estandarizado de modelado que es elegante, expresivo y flexible.

Prescindiendo de la aceptación que pueda tener, este lenguaje recibió la aprobación de facto en la industria, pues sus creadores representan métodos muy difundidos de la primera generación del análisis y diseño orientado a objeto. Muchas organizaciones dedicadas al desarrollo de software y los proveedores de herramientas de CASE (Computer Aided Software Engineering) lo adoptaron, y muy probablemente se convierta en el estándar mundial que utilizarán los desarrolladores de herramientas CASE. Algunas de las propiedades de UML como lenguaje de modelado estándar son:

- Concurrencia: es un lenguaje distribuido y adecuado a las necesidades actuales y futuras de conectividad.

- Ampliamente utilizado por la industria desde su adopción por OMG.
- Reemplaza a decenas de notaciones empleadas con otros lenguajes.
- Modela estructuras complejas.
- Las estructuras más importantes que soportan tienen su fundamento en las tecnologías orientadas a objetos, tales como objetos, clases, componentes y nodos.
- Emplea operaciones abstractas como guía para variaciones futuras, añadiendo variables si es necesario.
- Comportamiento del sistema: casos de uso, diagramas de secuencia y de colaboraciones, que sirven para evaluar el estado de las máquinas.

Se decidió utilizar UML pues la metodología seleccionada (RUP) lo emplea de manera eficiente, y de esta forma permite evolucionar adecuadamente en el diseño e implementación de un sistema informático.

### 1.7. Herramientas CASE

El incremento del desarrollo de software hizo que se creara un soporte automatizado para el desarrollo y mantenimiento de software. Este es el llamado "ingeniería del software asistida por computadora".

Una de las razones para la creación de las herramientas CASE fue el incremento en la velocidad de desarrollo de los sistemas, permitiendo a los analistas tener más tiempo para el análisis y diseño, y minimizar el tiempo para codificar y probar. Una herramienta CASE es un producto computacional enfocado a apoyar una o más técnicas dentro de un método de desarrollo de software. De acuerdo con Kendall la ingeniería de sistemas asistida por ordenador es la aplicación de tecnología informática a las actividades, las técnicas y las metodologías propias de desarrollo. Su objetivo es acelerar el proceso para el que han sido diseñadas, en el caso de CASE para automatizar o apoyar una o más fases del ciclo de vida del desarrollo de sistemas.

Las herramientas CASE son usadas en algunas de las fases de desarrollo de sistemas de información, incluyendo análisis, diseño y programación. El objetivo fundamental de las herramientas CASE es proveer un lenguaje para describir el sistema general que sea suficientemente explícito para generar todos los programas necesarios. Son muchos los

beneficios que pueden proveer las herramientas CASE en todas las etapas del proceso de desarrollo de software, algunas de ellas son:

- Hacer el trabajo de diseño de software más fácil y agradable.
- Verificar el uso de todos los elementos en el sistema diseñado.
- Ayudar en la documentación del sistema.
- Ayudar en la creación de relaciones en las bases de datos.
- Generar estructuras de código.
- Reducción del costo de producción de software.

El uso de las herramientas CASE permite una mejora en la calidad de los desarrollos realizados, pero para lograr esto además de la propia herramienta CASE es necesario contar con una organización y una metodología de trabajo. Estas herramientas permiten la reutilización del código, la portabilidad del software y la estandarización de la documentación.

### 1.7.1. Rational Rose Enterprise Edition

Rational Rose Enterprise Edition es una herramienta CASE que soporta el modelado visual con UML, ofreciendo distintas perspectivas del sistema. Da soporte al Proceso Unificado de Rational (RUP). Algunas de sus características son:

- Diseño dirigido por modelos que redundan en una mayor productividad de los desarrolladores.
- Diseño centrado en casos de uso y enfocado al negocio, que generan un software de mayor calidad.
- Cubre todo el ciclo de vida de un proyecto: concepción y formalización del modelo, construcción de los componentes, transición a los usuarios y certificación de las distintas fases y entregables.
- Solo permite trabajar sobre el sistema operativo Windows.
- Esta herramienta propone la utilización de cuatro tipos de modelo para realizar un diseño del sistema, utilizando una vista estática y otra dinámica de los modelos del sistema, uno lógico y otro físico. Permite crear y refinar estas vistas creando de esta

forma un modelo completo que representa el dominio del problema y el sistema de software.

- Desarrollo Iterativo
- Generador de Código

### 1.7.2. Visual Paradigm for UML

Visual Paradigm 6.0 es una herramienta CASE orientada a UML, soporta el ciclo de vida completo del desarrollo de software: análisis y diseño orientados a objetos, construcción, pruebas y despliegue. Permite dibujar todos los tipos de diagramas de clases, código inverso, generar código desde diagramas y generar documentación. Visual Paradigm genera toda la documentación de lo que se hace cumpliendo con estándares establecidos. El análisis textual es una técnica útil y práctica para la captura de los requisitos del sistema y la identificación de las clases candidatas, Visual Paradigm es una de las pocas herramientas CASE que soporta el análisis textual. Tiene disponibilidad en múltiples plataformas y en múltiples versiones. Esta característica es muy importante pues por ejemplo el Rational Rose, que es una herramienta muy recomendada y además profesional, tiene una desventaja en su contra pues obliga al usuario a desarrollar en máquinas con el sistema operativo Windows, mientras que el Visual Paradigm está disponible para varios sistemas operativos como Windows, Linux, Unix.

### 1.7.3. Enterprise Architect (EA):

Es una herramienta CASE para el diseño y construcción de sistemas de software.

Beneficios de EA:

- Soporta la especificación de UML 2.0, que describe un lenguaje visual por el cual se pueden definir mapas o modelos de un proyecto, y los diagramas de UML 2.1.
- Es una herramienta progresiva que cubre todos los aspectos del ciclo de desarrollo, proporcionando una trazabilidad completa desde la fase inicial del diseño a través del despliegue y mantenimiento.
- También provee soporte para pruebas, mantenimiento y control de cambio.
- Sus modelos se pueden exportar en formato RTF para una personalización y presentación final.

- Realiza ingeniería directa e inversa de código fuente en Action Script, C++, C#, Delphi, Java, Python, PHP, VB.NET, Visual Basic.
- Tiene una alta capacidad de sincronización de código.
- Posee soporte para Corba también disponible como plug-in libre.
- Posee plug-ins para vincular EA a Visual Studio.NET y Eclipse.
- Posee una interfaz de usuario intuitiva.
- Realiza ingeniería inversa para sistemas de Base de Datos muy populares como Oracle, SQL Server, MySQL, Access, PostgreSQL, entre otros.
- Permite generar tablas del Modelo de Base de Datos, columnas, claves, llaves foráneas, entre otras.
- Permite generar los scripts DLL para crear las estructuras de Base de Datos.
- Posibilita el desarrollo distribuido, es decir, es multiusuario.
- Soporta diferentes repositorios basados en DBMS, incluyendo Oracle, SQL Server, My SQL, PostGreSQL.
- Soporta importar/exportar archivos XMI para manejar la distribución y actualización de frameworks y otros paquetes basados en la estructura del modelo.
- Soporta control de versiones de repositorios.
- Soporta importar archivos .JAR en java y ensamblados .NET.
- Posee un perfil incorporado para XSD para simplificar el desarrollo de esquemas XML usando UML.
- Ingeniería Directa de esquema XML desde Modelos UML
- Posee gran velocidad, estabilidad y buen rendimiento.

Por todas estas características antes mencionadas, las grandes ventajas que brinda Enterprise Architect y la vasta experiencia existente por años de utilización de esta herramienta en el Proyecto Registros y Notarias se decidió su selección.

## 1.8. Conclusiones

La calidad en el desarrollo y mantenimiento de software se ha convertido en uno de los principales objetivos del desarrollo de software a nivel mundial, y la Universidad de las Ciencias Informáticas se hace partícipe de esta tendencia. Es por ello que en aras de construir un framework para aplicaciones de gestión empresarial con la calidad requerida, este capítulo se ha dedicado a realizar un estudio sobre distintos frameworks, metodologías de desarrollo de software, así como las distintas herramientas CASE, patrones de casos de uso, de diseño y arquitectura y distintos estilos arquitectónicos. También se dejaron claros conceptos relacionados con frameworks, arquitecturas, entre otros. De este estudio resultó la selección de la metodología RUP, el Lenguaje Unificado de Modelado como lenguaje de modelado y la herramienta CASE Enterprise Architect. De esta forma se ha dado cumplimiento al primer objetivo específico de este trabajo de diploma.

# CAPÍTULO 2: SOLUCIÓN PROPUESTA

## 2.1. Introducción

El presente capítulo tiene como objetivo hacer una valoración de las principales características del framework a desarrollar, el cual debe contribuir a sustentar la arquitectura del Proyecto Registros y Notarias en su Fase II. Se hace mención a las fases de la metodología de desarrollo utilizada para la implementación del framework que se propone y se exponen los artefactos generados durante el transcurso de las mismas.

## 2.2. Propuesta del sistema

La propuesta de solución del presente trabajo de diploma se basa en diseñar un framework que ayude a sustentar la arquitectura obtenida en el Proyecto Registros y Notarias en su Fase II, la cual está basada en el estilo arquitectónico: “Arquitectura en N-Capas Orientada al Dominio”, como se puede ver en la siguiente figura:

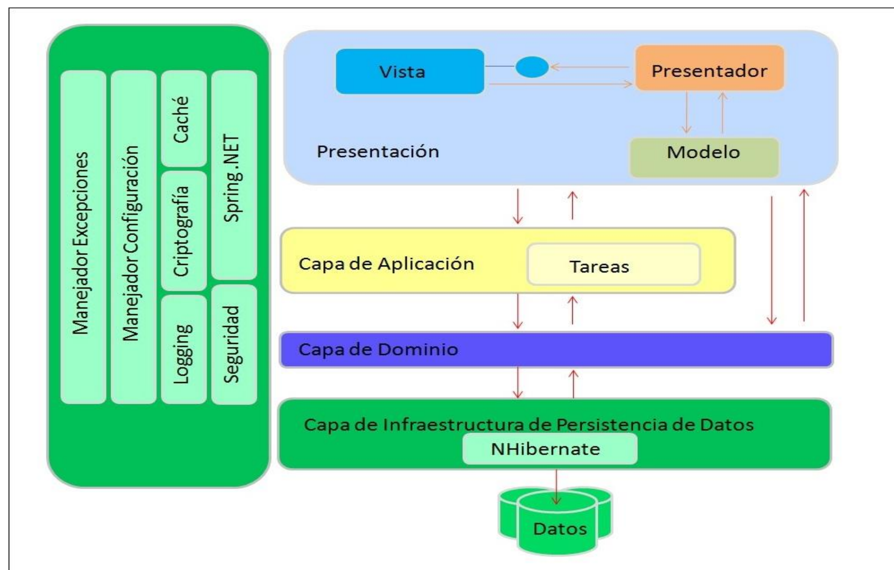



**Figura 2.1** Arquitectura Proyecto RN Fase II

Como se puede apreciar en la figura 2.1, esta arquitectura está compuesta por varias capas horizontales y verticales, las cuales por un lado garantizan el cumplimiento de los atributos de calidad de la arquitectura y por otro lado proveen un amplio espectro de funcionalidades que facilitan la implementación del software de gestión empresarial. Es por esta razón, que el framework que se propone debe contar con un conjunto de características y funciones que

ayuden a satisfacer dichos atributos de calidad y respondan a cada una de las funcionalidades con las que cuenta la arquitectura tanto en las capas horizontales como las capas verticales.

### 2.2.1. Capas Horizontales

**Presentación**

El patrón (MVP) separa el modelo del dominio, la presentación y las acciones basadas en la interacción con el usuario en tres clases separadas. La vista le delega a su presentador toda la responsabilidad del manejo de los eventos del usuario. El presentador se encarga de actualizar el modelo cuando surge un evento en la vista, pero también es responsable de actualizar a la vista cuando el modelo le indica que ha cambiado. El modelo no conoce la existencia del presentador. Por lo tanto, si el modelo cambia por acción de algún otro componente que no sea el presentador, debe disparar un evento para que éste se entere.

A la hora de implementar este patrón, se identifican los siguientes componentes:

- **IVista:** es la interfaz con la que el Presentador se comunica con la vista.
- **Vista:** vista que implementa la interfaz IVista y se encarga de manejar los aspectos visuales. Mantiene una referencia a su Presentador el cual le delega la responsabilidad del manejo de los eventos.
- **Presentador:** contiene la lógica para responder a los eventos y manipula el estado de la vista mediante una referencia a la interfaz IVista. El Presentador utiliza el modelo para saber cómo responder a los eventos. El presentador es responsable de establecer y administrar el estado de una vista.
- **Modelo:** Está compuesto por los objetos que conocen y manejan los datos dentro de la aplicación. Por ejemplo, pueden ser las clases que conforman el modelo del negocio.

Incluir el diseño de este componente en el framework propuesto posibilita parcialmente el cumplimiento del objetivo general de este trabajo de diploma, pues brinda sustento a la arquitectura en términos de funcionalidad y mantenibilidad debido a que su implementación incorpora dichos atributos de calidad.

**Persistencia de Datos**

En esta capa se encapsula la lógica requerida para acceder a las fuentes de dados requeridos por la aplicación. Centraliza por tanto la funcionalidad común de acceso a datos de forma que la aplicación pueda disponer de un mejor mantenimiento y desacoplamiento entre la tecnología con respecto a la lógica del Dominio. Si se hace uso de tecnologías base tipo ORM (Object/Relational Mapping Frameworks), se simplifica el trabajo. (Nilson, 2006) Para el framework que se está desarrollando se propone el NHibernate el cual se acopla muy bien con el Spring.NET y brinda a las aplicaciones atributos de calidad como funcionalidad, desempeño, configurabilidad, modificabilidad y mantenibilidad, ayudando a sustentar la arquitectura propuesta para el Proyecto Registros y Notarías Fase II.

### 2.2.2. Capas Verticales

**Spring.net:**

Sobre este framework se construyen la mayoría de los elementos arquitectónicamente significativos que se pueden encontrar, tomando como premisa sus principales potencialidades las cuales tributan significativamente a la arquitectura, tales como Inyección de Dependencia, Programación Orientada a Aspectos, interoperabilidad e integración con otros framework utilizados así como el hecho de ser un framework de software libre dando la posibilidad de tener acceso a su código fuente.

Para el correcto funcionamiento del framework y garantizar la explotación máxima de sus funcionalidades, se debe configurar el fichero App.config con los diferentes recursos y dependencias, dichos ficheros estarán distribuidos por cada una de las capas en dependencia del grado de responsabilidad que esta tiene con el objetivo de ganar en cohesión y organización del código.

Este framework al igual que NHibernate, brinda sustento a la arquitectura en términos de funcionalidad, desempeño, configurabilidad, integrabilidad y modificabilidad, así como con la adopción de buenas prácticas con la reutilización de patrones fundamentales como son la Inyección de Dependencias y la Inversión de Control, los cuales añaden comprensión, flexibilidad y mantenibilidad a la arquitectura.

**Net4Log:**

Para los servicios de logeo se selecciono el Log4Net de la fundación de software apache. Es fácil de usar, código abierto y está muy bien documentado. Hay otra gran cantidad de servicio de logeo pero no son de código abierto.

Log4Net es una librería de logeo que nos permite logear fácilmente mensajes de nuestras aplicaciones a diferentes tipos de almacenamiento. Por ejemplo: ficheros de texto, MS SQL, Oracle, SQL Lite, SMTP, y otros. En caso de algún problema con la aplicación es de mucha ayuda para saber donde está localizado. Por lo que el sistema gana en cuanto a trazabilidad, y es más fácil revisar los estados del software y corregir errores, garantizando así atributos de calidad de la arquitectura del proyecto Registros y Notarías Fase II muy importantes como la mantenibilidad y la capacidad de prueba.

**Criptografía**

La criptografía es todo acerca de conservar secretos. Esto puede ser algo tan simple como proteger la contraseña o tan complicado como encubrir las transacciones financieras de un cliente de ojos indiscretos. La técnica usada para encubrir fue llamada encriptar. La encriptación se divide en tres categorías

Hashes: Un hash es el cifrado de un solo sentido para una situación en la que el valor tiene que estar asegurado pero no tiene que ser descifrado. Los valores hash son números que representan una cadena o valor de la matriz de bytes. En su formato original, es mucho menor que el valor que representan. Ellos son generados por un algoritmo que hace que sea muy poco probable que dos cadenas den como resultado el mismo valor hash. Los valores hash se utilizan para los artículos tales como contraseñas.

**Cifrado simétrico:** Este tipo de cifrado utiliza la misma clave para cifrar y descifrar los datos. El peligro de cifrado simétrico es que la clave debe ser entregada al usuario final antes de que pueda ser utilizado y puede ser interceptada. Si bien el hecho de que ambas partes utilizan la misma clave es su riesgo para la seguridad propia, este tipo de cifrado tiene sus beneficios. La ventaja más obvia es que es más rápido que el cifrado asimétrico.

**Cifrado asimétrico:** Este tipo de cifrado utiliza una clave pública y otra privada. Esto es más seguro que la alternativa simétrica, ya que sólo la clave privada puede descifrar el mensaje

cifrado con la clave pública, sin embargo, también es más lento. El ejemplo clásico de este tipo de cifrado Secure Sockets Layer (SSL), que se utiliza para asegurar tráfico HTTP.

La protección de datos (DPAPI) proporciona un medio para proteger las claves privadas, credenciales y los datos confidenciales para el uso de aplicaciones. DPAPI se usa para proteger las claves de cifrado, lo que lo hace a través de la entropía. La entropía es una clave secundaria que se utiliza para garantizar que los usuarios no puedan descifrar una clave por su propia cuenta.

**Seguridad**

La seguridad tiende a ser un área de desarrollo de aplicaciones que a menudo se pasa por alto o es implementada pobremente. Esto es parcialmente debido a que los calendarios de los proyectos son cada vez más limitados, pero también debido a la complejidad cada vez mayor del desarrollo de aplicaciones. Este problema se agrava por las miles de maneras en que los equipos de desarrollo históricamente han manejado la creación de sistemas de acceso. El componente de seguridad se introduce al diseño de nuestro framework para simplificar y estandarizar la forma de autorizar a los usuarios y el acceso de forma segura a la información en cache dentro de la aplicación.

Con este componente los desarrolladores que usen el framework podrán implementar las funcionalidades comunes relacionadas con la seguridad en sus aplicaciones. El componente de seguridad permitirá manejar cuestiones como la autenticación y autorización de usuarios contra una base de datos, recuperación del rol y la información del perfil y el almacenamiento en caché de la información de perfiles de usuario. Además reduce la necesidad de escribir código repetitivo para realizar tareas comunes en lo referente a la seguridad y ayuda a mantener la coherencia de las prácticas de seguridad en nuestras aplicaciones.

Tanto el componente de criptografía como el de seguridad brindan fortaleza a la arquitectura en términos de atributos de calidad fundamentales como son disponibilidad, confidencialidad, seguridad interna e integridad, características estas que dan sustento a la arquitectura.

**Caching**

El almacenamiento en cache de los datos es adecuado para cualquiera de las situaciones siguientes:

- Se debe acceder varias veces a datos o estáticos o a datos que es poco probable que sufran cambios.
- El manejo de los datos se vuelve costoso en términos de creación, acceso o transporte.
- Los datos siempre deberán estar disponibles incluso cuando la fuente, por ejemplo un servidor, no esté disponible.

El desarrollo de este componente permitirá a los desarrolladores que usen el framework incorporar una memoria caché local en sus aplicaciones. Esto, entre otras cosas, permite mejorar atributos muy importantes de la aplicación y por consiguiente darle sustento a la arquitectura dando así cumplimiento al objetivo general de este trabajo. Dichos atributos son:

- **Rendimiento:** manteniendo los datos cerca de donde se usan. Esto evita incurrir, para cada llamada, en los costos de creación, procesamiento y transporte de los datos.
- **Escalabilidad:** almacenar datos en una cache local ayuda a ahorrar recursos e incrementa la escalabilidad a medida que aumentan las demandas de la aplicación.
- **Disponibilidad:** puesto que si se mantiene una cache local la aplicación puede sobrevivir a fallos del sistema tales como la latencia de red, problemas de servicios Web, y los fallos de hardware.

## 2.3. Modelo del sistema

En este epígrafe se especifican los requisitos tanto funcionales como no funcionales que tendrá el sistema que se quiere construir, así como el modelamiento del sistema en términos de casos de uso.

### 2.3.1. Requisitos del sistema

La IEEE Standard Glossary of Software Engineering Terminology define un requerimiento como:

- Condición o capacidad que necesita un usuario para resolver un problema o lograr un objetivo.
- Condición o capacidad que tiene que ser alcanzada o poseída por un sistema o componente de un sistema para satisfacer un contrato, estándar, u otro documento impuesto formalmente.

El propósito fundamental de la captura de los requisitos es guiar el desarrollo hacia el sistema correcto. Esto se consigue mediante una descripción de los requisitos del sistema

suficientemente buena como para que pueda llegarse a un acuerdo entre el cliente (incluyendo a los usuarios) y los desarrolladores sobre qué debe y qué no debe hacer el sistema. (Rumbaugh, y otros, 2004)

Para capturar los requisitos se utilizó la técnica de entrevistas. Esta técnica fue aplicada a los desarrolladores y arquitectos de algunos proyectos productivos en la UCI, cuya experiencia oscila entre los 6 meses y los 3 años. Los resultados arrojados fueron los siguientes:

**Requisitos funcionales**

Son capacidades o condiciones que el sistema debe cumplir. Para cumplir los objetivos de este sistema el mismo debe ser capaz de:

RF1 - Reportar excepción.

**Descripción**: el framework debe ser capaz de reportarle a la aplicación que lo utilice cada una de las excepciones lanzadas por el sistema.

**Prioridad:** Alta.

RF2 - Describir excepción.

**Descripción**: el framework debe ser capaz de guardar cada una de las excepciones reportadas de acuerdo al formato que haya sido definido en la configuración del componente.

**Prioridad:** Media.

RF3 - Guardar excepción.

**Descripción**: el framework debe ser capaz de describir cada una de los atributos de todas las excepciones reportadas para que estas sean guardadas posteriormente.

**Prioridad:** Alta.

RF4 - Mostrar excepción.

**Descripción**: el framework debe ser capaz de mostrarle al usuario cada una de las excepciones reportadas.

**Prioridad:** Alta.

RF5 - Manejar Configuración.

**Descripción**: El framework debe permitir manejar toda la configuración de cada uno de sus componentes fundamentales.

**Prioridad:** Alta.

RF6 - Guardar logs.

**Descripción**: El framework debe permitir guardar los logs que se generen con cada uno de los mensajes de error de la aplicación.

**Prioridad:** Alta.

RF7 - Obtener logs.

**Descripción**: El framework debe permitir obtener todos los logs que han sido previamente guardados para su utilización en las pruebas del sistema.

**Prioridad:** Alta.

RF8 - Realizar almacenamiento en caché de datos.

**Descripción**: El framework debe ser capaz de proveer al sistema de una memoria caché local de datos para los datos estáticos que son utilizados frecuentemente por la aplicación.

**Prioridad:** Media.

RF9 - Obtener Información de Tarea.

**Descripción**: El framework debe permitir obtener toda la información de una tarea que el usuario ha solicitado iniciar para su posterior ejecución.

**Prioridad:** Alta.

RF10 - Navegar a una vista.

**Descripción**: El framework debe permitir dentro de la ejecución de una tarea, navegar a las vistas correspondientes a cada uno de los presentadores.

**Prioridad:** Alta.

RF11 - Autenticar usuario.

**Descripción**: El framework debe brindar la funcionalidad de autenticación de usuarios a la aplicación.

**Prioridad:** Alta.

RF12 - Autorizar usuario.

**Descripción**: El framework debe brindar la funcionalidad de autorización de usuarios a la aplicación.

**Prioridad:** Alta.

RF13 - Encriptar Datos.

**Descripción**: El framework debe brindar la funcionalidad de encriptar los datos que así lo requieran como contraseñas, datos de configuración, entre otros.

**Prioridad:** Media.

RF14 - Desencriptar Datos.

**Descripción**: El framework debe brindar permitir Desencriptar los datos que han sido previamente encriptados.

**Prioridad:** Media.

**Requisitos no funcionales**

Los requisitos no funcionales son propiedades o cualidades que el producto debe tener. Debe pensarse en estas propiedades como las características que hacen al producto atractivo, usable, rápido o confiable.

**Requisitos de software**

RNF-1 Sistema Operativo Microsoft Windows 2003 o superior; Sistema Operativo Linux.

RNF-2 Plataforma .NET (2.0) o superior, Plataforma Mono 1.9 o superior con GTK# 2.10 o superior.

**Requisitos de hardware**

RNF-3 Se requiere un mínimo de 256 MB de RAM, 512 MB recomendado, y 1.3 GHz de velocidad de procesamiento.

**Requisitos de usabilidad**

RNF-4 Podrá ser usado fácilmente por cualquier desarrollador, personal de soporte u otra persona con vasta experiencia en el uso de herramientas informáticas, con un mínimo de capacitación y experiencia.

**Requisitos de portabilidad**

RNF-5 Una de las mayores ventajas que tendrá el sistema es su portabilidad ya que el mismo podrá correr en cualquier plataforma, Windows o Linux.

**Restricciones de diseño e implementación**

RNF-6 El sistema será implementado en la plataforma .NET para luego ser migrado a la plataforma MONO.

**Rendimiento**

RNF-7 El sistema debe requerir un consumo mínimo de recursos.

Debe tener tiempos de respuesta rápidos garantizando de esta forma la agilidad del sistema.

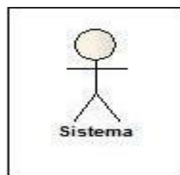
### 2.3.2. Modelo de casos de uso del sistema

Un modelo de casos de uso es un modelo del sistema que contiene actores, casos de uso y sus relaciones.

El modelo de casos de uso permite que los desarrolladores y los clientes lleguen a un acuerdo sobre los requisitos, es decir sobre las condiciones y posibilidades que debe cumplir el sistema. El modelo de casos de uso describe lo que hace el sistema para cada tipo de usuario. Cada usuario se representa mediante uno o más actores. (Rumbaugh, y otros, 2004)

**Actores del sistema**

Los actores representan terceros fuera del sistema que colaboran con el sistema. Al identificar los actores del sistema se identifica el entorno externo del sistema.



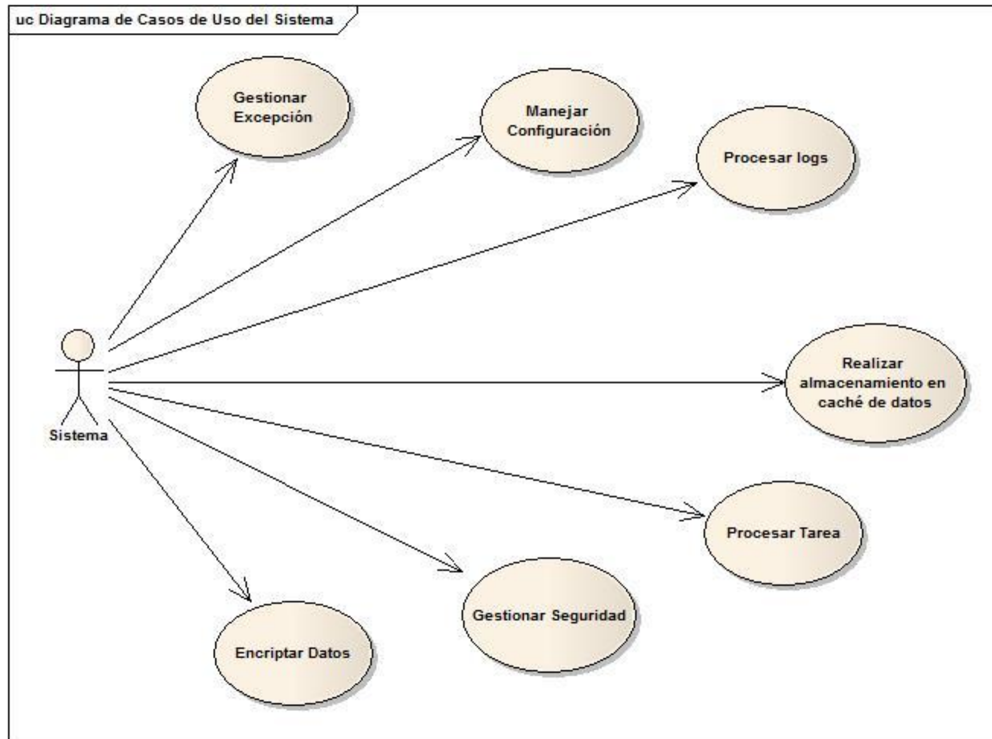

**2.2** Actor del Sistema

Cada forma en que los actores usan el sistema se representa con un caso de uso. Los casos de uso son fragmentos de funcionalidad que el sistema ofrece para aportar un resultado de valor para sus actores. Un caso de uso especifica una secuencia de acciones que el sistema puede llevar a cabo interactuando con sus actores, incluyendo alternativas dentro de la secuencia. (Rumbaugh, y otros, 2004). Los casos de uso son el componente clave del modelado. Su propósito es ilustrar como un sistema permite a un actor cumplir una meta, ilustrando todos los posibles caminos apropiados que ellos pueden tomar para cumplirla, así como las situaciones que podrían hacerlo fallar.

**2.3** Caso de Uso del Sistema

**Diagrama de Casos de uso del sistema.**

El Diagrama de Casos de uso describe cómo interactúan los actores y los casos de uso y como se relacionan entre sí los casos de uso. (Rumbaugh, y otros, 2004) Es un modelo de las funciones del sistema y su entorno, y sirve como un contrato entre el cliente y los desarrolladores. El modelo de casos de uso del sistema es usado como una entrada fundamental a las actividades en el análisis, diseño y las pruebas. La figura 2.4 muestra el Diagrama de Casos de Uso del Sistema propuesto.




**2.4** Diagrama de Casos de Uso del Sistema

**Especificación de los casos de uso del sistema**

En este epígrafe se presenta una breve descripción de cada uno de los casos de uso del sistema, presentando: nombre del caso de uso, actor y la descripción del mismo.

| **Nombre del CU** | Gestionar Excepción |
|---|---|
| **Actor** | Sistema(Inicia) |
| **Descripción:** Este caso de uso consiste en darle tratamiento a todas las excepciones que sean lanzadas por el sistema, o sea, describir la excepción, verificar si ya fue tratada, guardarla y mostrarla al usuario. | |

| **Nombre del CU** | Manejar Configuración |
| --- | --- |
| **Actor** | Sistema(Inicia) |
| **Descripción:** Este caso de uso consiste en manejar todas las secciones de configuración de cada uno de los componentes registradas por el usuario, y en los que se incluyen propiedades esenciales en las políticas de funcionamiento de un componente y parámetros de uso general. | |

| **Nombre del CU** | Procesar logs |
| --- | --- |
| **Actor** | Sistema(Inicia) |
| **Descripción:** Este caso de uso consiste en procesar todos los logs de mensajes de errores del sistema, o sea, guardarlos, leerlos y así facilitar el mantenimiento y las pruebas del sistema. | |

| **Nombre del CU** | Realizar almacenamiento en caché de datos |
| --- | --- |
| **Actor** | Sistema(Inicia) |
| **Descripción:** Este caso de uso consiste en incorporar una caché de datos local al sistema y gestionar todo lo relacionado con la misma: qué tipo de objetos deben persistir en caché, tiempo de persistencia, eliminación de los mismos de la caché al caducar, obtener los objetos de la caché cuando deban ser usados, guardarlos y obtenerlos de forma segura. | |

| **Nombre del CU** | Procesar Tarea |
|---|---|
| **Actor** | Sistema(Inicia) |
| **Descripción:** Este caso de uso consiste en implementar el patrón Modelo-Vista-Presentador, o sea, cada una de las acciones de los usuarios son tratadas como una tarea, por lo que se debe realizar la tarea gestionando toda la información que requiera la misma. | |

| **Nombre del CU** | Gestionar Seguridad |
|---|---|
| **Actor** | Sistema(Inicia) |
| **Descripción:** Este caso de uso consiste en gestionar toda la seguridad del sistema, o sea, la forma en que los usuarios se autentican y son autorizados a acceder a la aplicación y toda la información de la misma. También se encarga de la seguridad de los objetos en cache. | |

| **Nombre del CU** | Encriptar Datos |
|---|---|
| **Actor** | Sistema(Inicia) |
| **Descripción:** Este caso de uso consiste en la encriptación de los distintos datos que requieren ser encriptados utilizando los métodos más usados de encriptación existentes | |

## 2.4. Diseño

En el diseño se modela el sistema y se encuentra su forma para que soporte todos los requisitos-incluyendo los requisitos no funcionales y otras restricciones –que le suponen. Una entrada esencial en el diseño es el resultado del análisis, esto es el modelo de análisis. Los propósitos del diseño son:

- Adquirir una comprensión en profundidad de los aspectos relacionados con los requisitos no funcionales y restricciones relacionadas con los lenguajes de programación, componentes reutilizables, sistemas operativos, tecnologías de distribución y concurrencia, tecnologías de interfaz de usuario y tecnologías de gestión de transacciones.
- Crear una entrada apropiada y un punto de partida para actividades de implementación subsiguientes capturando los requisitos o subsistemas individuales, interfaces y clases.
- Ser capaces de descomponer los trabajos de implementación en partes más manejables que puedan ser llevadas a cabo por diferentes equipos de desarrollo, teniendo en cuenta la posible concurrencia.
- Ser capaces de visualizar y reflexionar sobre el diseño utilizando una notación común. (Rumbaugh, y otros, 2004)

### 2.4.1. Modelo de diseño

El modelo de diseño es un modelo de objetos que describe la realización física de los casos de uso, centrándose en como los requisitos funcionales y no funcionales, junto con otras restricciones relacionadas con el entorno de implementación, tienen impacto en el sistema a considerar. El modelo de diseño sirve de abstracción a la implementación, y es de ese modo, utilizado como una entrada fundamental a las actividades de implementación.

#### Realización de caso de uso-diseño

Una realización de caso de uso-diseño es una colaboración en el modelo de diseño que describe cómo se realiza un caso de uso específico, y como se ejecuta, en términos de clases de diseño y sus objetos. Proporciona una traza directa a una realización de casos de uso-análisis en el modelo de análisis. Una realización de caso de uso-diseño tiene una

descripción textual del flujo de eventos, diagramas de clases que muestran sus clases de diseño participantes, y diagramas de interacción que muestran las realizaciones de un flujo o escenario concreto de un caso de uso en términos de interacción entre objetos del diseño.

**Diagrama de clases del diseño**

En este diagrama se representan las clases participantes en las realizaciones de caso de uso, subsistemas y sus relaciones. También puede darse el caso de algunas operaciones, atributos y asociaciones sobre una clase específica que son relevantes.

**Clases del diseño**

Una clase del diseño es una abstracción sin costuras de una clase o construcción similar en la implementación del sistema. Esta abstracción es sin costuras en el siguiente sentido:

- El lenguaje utilizado para especificar una clase de diseño es lo mismo que el lenguaje de programación. Consecuentemente las operaciones, parámetros, atributos, tipos y demás son especificados utilizando la sintaxis del lenguaje de programación elegido.
- La visibilidad de los atributos y las operaciones de una clase de diseño se especifica con frecuencia.
- Las relaciones de aquellas clases de diseño implicadas con otras clases, a menudo tienen un significado directo cuando la clase es implementada.

Los métodos tienen correspondencia directa con el correspondiente método en la implementación de las clases.

**Diagrama de clases del diseño propuesto. (****VER ANEXO 1****)**

**Diagramas de interacción**

La secuencia de acciones en un caso de uso comienza cuando un actor invoca el caso de uso mediante el envío de algún tipo de mensaje al sistema. En el interior del sistema tendremos algún objeto de diseño que recibe el mensaje del actor. Después el objeto de diseño llama a algún otro objeto, y de esta manera los objetos implicados interactúan para realizar y llevar a cabo el caso de uso. En el diseño es preferible representar esto con diagramas de secuencia ya que nuestro centro de atención principal es encontrar secuencias de interacciones detalladas y ordenadas en el tiempo.

**Diagramas de secuencia del diseño propuesto (Ver ANEXO 2)**

## 2.5. Conclusiones

En el presente capítulo utilizando como basamento las características del sistema, que fueron definidas en el capítulo anterior, se obtuvo la especificación de los requisitos tanto funcionales como no funcionales del framework propuesto, el diagrama de casos de uso del sistema, así como los artefactos de diseño necesarios para cada uno de sus componentes fundamentales. Además quedo demostrado como cada componente al incorporar importantes atributos de calidad de una aplicación, sustenta la arquitectura del proyecto Registros y Notarias Fase II. Con la realización de este capítulo, hemos dado cumplimiento de forma parcial a nuestros objetivos.

# CAPÍTULO 3: ANALISIS DE LOS RESULTADOS

## 3.1. Introducción

Lord Kelvin en una ocasión dijo: "Cuando pueda medir lo que está diciendo y expresarlo con números, ya conoce algo sobre ello; cuando no pueda medir, cuando no pueda expresar lo que dice con números, su conocimiento es precario y deficiente: puede ser el comienzo del conocimiento, pero en sus pensamientos, apenas está avanzando hacia el escenario de la ciencia."

La medición es fundamental para cualquier disciplina de ingeniería, y la ingeniería del software no es una excepción. La medición permite tener una visión más profunda, proporcionando un mecanismo para la evaluación objetiva. (Pressman, 2005) En el presente capítulo se realiza un análisis de los resultados, tomando como principal basamento las métricas para determinar la calidad de la especificación de los requisitos, del DCUS y del diseño.

**Métricas**

El IEEE Standard Glossary of Software Engineering Terms (IEEE, 1993) define métrica como una medida cuantitativa del grado en que un sistema, componente o proceso posee un atributo dado. Una medida proporciona una indicación cuantitativa de la extensión, cantidad, dimensiones, capacidad o tamaño de algunos atributos de un proceso o producto. Hay cuatro razones para medir los procesos del software, los productos y los recursos:

- Caracterizar
- Evaluar
- Predecir
- Mejorar

Caracterizar es importante para comprender mejor los procesos, los productos, los recursos y los entornos y para establecer las líneas base para las comparaciones con evaluaciones futuras. Evaluar permite determinar el estado con respecto al diseño. También permite valorar la consecución de los objetivos de calidad y evaluar el impacto de la tecnología y las mejoras del proceso. Predecir ayuda a planificar. Se hace esto porque se quiere establecer objetivos alcanzables para el coste, planificación, y calidad de manera que se puedan aplicar los recursos apropiados. Por último la medición se realiza para mejorar, permite recoger la

información cuantitativa que ayuda a identificar obstáculos, problemas de raíz, ineficiencias y otras oportunidades para mejorar la calidad del producto y el rendimiento del proceso. (Pressman, 2005)

### 3.2. Métricas de la calidad de la especificación

Davis y sus colegas (Pressman, 2005) proponen una lista de características que pueden emplearse para valorar la calidad del modelo de análisis y la correspondiente especificación de requisitos: especificidad (ausencia de ambigüedad), compleción, corrección, comprensión, capacidad de verificación, consistencia interna y externa, capacidad de logro, concisión, trazabilidad, capacidad de modificación, exactitud y capacidad de reutilización.

Aunque muchas de estas características parecen de naturaleza cualitativa, Davis sugiere que todas pueden representarse usando una o más métricas.

Asumimos que hay $n_r$ requisitos en una especificación, tal como:

$$n_r = n_f + n_{nf}$$

Donde: $n_f$ **:** es el número de requisitos funcionales

$n_{nf}$ **:** es el número de requisitos no funcionales.

**Especificidad** Para determinar la especificidad (ausencia de ambigüedad) de los requisitos. Davis sugiere una métrica basada en la consistencia de la interpretación de los revisores para cada requisito:

$$Q_i = n_{ui} \;/\; n_r$$

Donde: $n_{ui}$ es el número de requisitos para los que todos los revisores tuvieron interpretaciones idénticas.

Cuanto más cerca este de 1 el valor de Q, menor será la ambigüedad de la especificación. Para evaluar la métrica de la especificidad de los requisitos se realizaron dos revisiones por varios revisores, específicamente 10 analistas pertenecientes al proyecto Registros y Notarías. El objetivo principal de estas revisiones era obtener los requisitos con el mayor grado de claridad posible y sin ambigüedades.

**Resultados:**

$n_r = n_f + n_{nf}$

$n_r$ = 14 + 7

$n_r$ = 21

**Primera Revisión**

$Q_i = n_{ui} / n_r$

$Q_i$ = 17 / 21

$Q_i$ = 0.81

**Segunda Revisión**

$Q_i = n_{ui} / n_r$

$Q_i$ = 21 / 21

$Q_i$ = 1

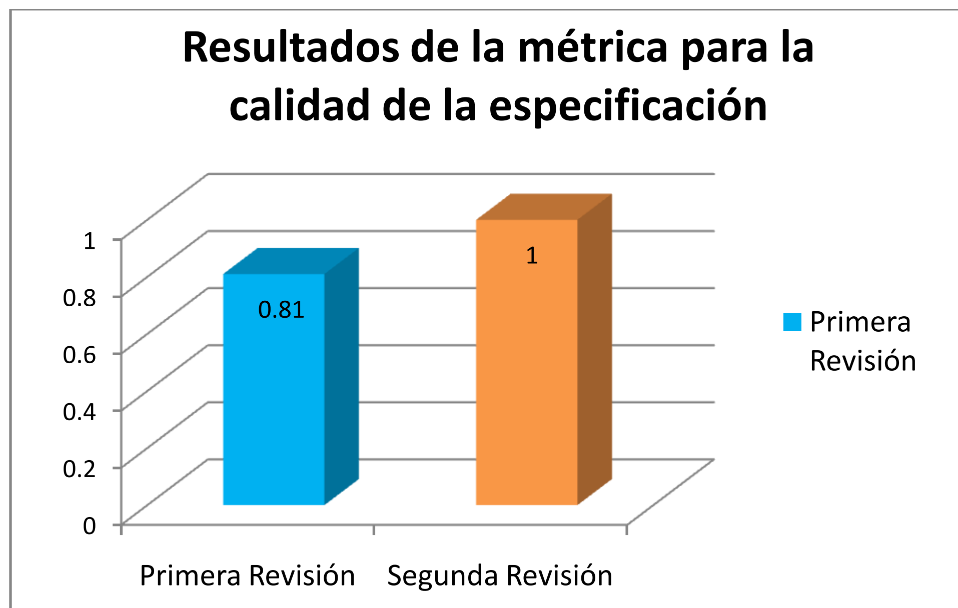



**Figura 3.1** Resultados de la métrica para la calidad de la especificación

Luego de realizada la primera revisión se detectaron algunos requisitos con problemas de redacción, que dificultaban la interpretación de los revisores. Se realizaron los cambios necesarios y para la segunda revisión todos los revisores tuvieron la misma interpretación 21 requisitos de un total de 21, lo que significa que todos los requisitos están especificados de forma clara y sin ambigüedades.

**Matriz de Trazabilidad**

Seguidamente se muestra la matriz de trazabilidad de los Requisitos vs Casos de Uso, de manera que permite conocer qué requisitos serán incluidos en qué casos de uso, y si un requisito sufre cambios, entonces se conocerá qué caso de uso deberá ser cambiado. En dicha matriz se aprecia como resultado que todos los requisitos fueron asociados al menos a un caso de uso para cubrir una determinada funcionalidad del Sistema.

| | Autenticar usuario | Autorizar usuario | Describir excepción | Desencriptar datos | Encriptar datos | Guardar excepción | Guardar logs | Manejar configuración | Mostrar excepción | Navegar a una vista | Obtener información de tarea | Obtener logs | Realizar almacenamiento en caché | Reportar excepción |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Encriptar Datos | | | | X | X | | | | | | | | | |
| Gestionar Excepción | | | X | | | X | | | X | | | | | X |
| Gestionar Seguridad | X | X | | | | | | | | | | | | |
| Manejar Configuración | | | | | | | | X | | | | | | |
| Procesar logs | | | | | | | X | | | | | X | | |
| Procesar Tarea | | | | | | | | | | X | X | | | |
| Realizar almacenamiento en caché de datos | | | | | | | | | | | | | X | |

**Figura 3.2** Matriz de Trazabilidad

## 3.3. Métrica NOAS / NOS

**NOAS:** Number of Actor Steps- Número de pasos del actor.

**NOS:** Number of Steps- Número de pasos.

Esta heurística se basa en la idea de que un caso de uso sirve para expresar una interacción actor–sistema. Por ello, el número de pasos de actor y el de pasos del sistema deben estar en torno al 50%, considerando también la posibilidad de que existan pasos de inclusión o extensión en los que se realice otro caso de uso. Las situaciones que llevan a esta métrica fuera del rango habitual son:

- El hecho de obviar la participación del sistema, por lo que el caso de uso resulta incompleto. Es la situación más habitual.

- El hecho de haber desglosado demasiado las acciones de un actor determinado. En este caso aparecen varios pasos seguidos, del mismo actor, lo cual podría haberse evitado uniéndolos en uno solo, separando las acciones por comas en el texto del paso.

El rango habitual de esta métrica es [30%,60%]. Un valor alto de la misma puede estar condicionado por el hecho de que el caso de uso no incluye todo lo que debe hacer el sistema para alcanzar el objetivo de este o demasiado desglose de los pasos del actor. Un valor bajo de la misma puede estar condicionado por el hecho de que no incluye todo lo que debe hacer el actor para alcanzar el objetivo del caso de uso, demasiado desglose de los pasos del sistema. (Bernárdez, y otros, 2008)

| Nombre del CU | NOAS | NOS | Valor de la métrica |
|---|---|---|---|
| Gestionar Excepción | **6** | **13** | **46.2%** |
| Manejar Configuración | **10** | **19** | **52.6%** |
| Procesar loggs | **3** | **7** | **42.9%** |
| Realizar almacenamiento en caché de datos | **5** | **10** | **50%** |
| Procesar Tarea | **8** | **15** | **53.3%** |
| Gestionar Seguridad | **7** | **15** | **46.7%** |
| Encriptar Datos | **4** | **8** | **50%** |

**Tabla 3.1** Aplicación de la métrica NOAS/NOS al DCUS

Esta métrica fue aplicada a cada uno de los casos de uso del sistema, como se muestra en la tabla anterior, para cada uno de los casos de uso el valor de esta métrica está en el rango habitual, con valores bastante cercanos al 50%, lo que implica que los mismos están bastante completos pues no se obvian participaciones del actor o del sistema, o sea, que se incluye todo lo que debe hacer tanto el actor como el sistema, para lograr el objetivo del caso de uso; y los pasos del actor y del sistema tienen un desglose adecuado. En el caso de las acciones del actor, si realiza varias, estas están separadas por comas, lo que constituye un solo paso.

Se aplicaron además un conjunto de métricas orientadas a objetos para evaluar las siguientes propiedades de calidad: **Correctitud y Complejidad.**

**Correctitud:** Grado en que las interacciones actor / sistema soportan adecuadamente el proceso del negocio.

**Complejidad:** Grado de claridad en la presentación de los elementos que describen el contexto y la claridad del sistema.

Estas métricas fueron aplicadas en dos revisiones realizadas:

**Primera Revisión**

| **Factor** | **Métricas Asociadas** | **Valor** |
| --- | --- | --- |
| **Factor Correctitud** | | |
| Factor 1. ¿Representa el caso de uso requisitos comprensibles por el usuario? | Métrica1. Grado en que los requisitos representados por el caso de uso son comprensibles por el usuario.<br><br>Métrica 2. Número de casos de uso en que los requisitos representados no son comprensibles por el usuario. | Total de requisitos: 21<br>Cantidad de requisitos que no son comprensibles por el usuario: 0<br>Representa: 0%<br>Total de casos de Uso: 7<br>Número de casos de uso en que los requisitos representados no son comprensibles por el usuario: 0<br>Representa: 0% |
| Factor 2. ¿Las interacciones definidas describen la funcionalidad requerida del sistema? | Métrica 3. Número de casos de uso que deben ser modificados para adecuarlos a la funcionalidad del sistema | Total de casos de Uso: 7 Número de casos de uso que deben ser modificados para adecuarlos a la funcionalidad del sistema : 1<br>Representa: 14.3 % |
| Factor 3. Las interacciones definidas introducen mejoras al proceso actual. | Métrica 4. Número de casos de uso que deben ser modificados para mejorar el | Total de casos de Uso: 7 Número de casos de uso que deben ser modificados para |

| | | |
|---|---|---|
| | proceso actual | mejorar el proceso actual: 2 Representa: 28,6% |
| | | **89.28%** |

**Tabla 3.2** Factor Correctitud Primera Revisión

| **Factor** | **Métricas Asociadas** | **Valor** |
|---|---|---|
| **Factor Complejidad** | | |
| Factor 4. ¿Los elementos dentro del diagrama están adecuadamente ubicados de manera que facilitan su interpretación? | Métrica 5. Número de elementos del diagrama que requieren reubicación | Total de casos de Uso: 7. Número de casos de uso que requieren reubicación de manera que faciliten su interpretación: 1 Representa: 14.3 % |
| | | **85,7%** |

**Tabla 3.3** Factor Complejidad Primera Revisión

**Segunda Revisión**

| **Factor** | **Métricas Asociadas** | **Valor** |
|---|---|---|
| **Factor Correctitud** | | |
| Factor 1. ¿Representa el caso de uso requisitos comprensibles por el usuario? | Métrica1. Grado en que los requisitos representados por el caso de uso son comprensibles por el usuario.<br><br>Métrica 2. Número de casos de uso en que los requisitos representados no son comprensibles por el usuario. | Total de requisitos: 21 Cantidad de requisitos que no son comprensibles por el usuario: 0 Representa: 0%<br>Total de casos de Uso: 7 Número de casos de uso en que los requisitos representados no son comprensibles por el |

| | | |
|---|---|---|
| | | usuario: 0<br>Representa: 0% |
| Factor 2. ¿Las interacciones definidas describen la funcionalidad requerida del sistema? | Métrica 3. Número de casos de uso que deben ser modificados para adecuarlos a la funcionalidad del sistema | Total de casos de Uso: 7 Número de casos de uso que deben ser modificados para adecuarlos a la funcionalidad del sistema : 0<br>Representa: 0 % |
| Factor 3. Las interacciones definidas introducen mejoras al proceso actual. | Métrica 4. Número de casos de uso que deben ser modificados para mejorar el proceso actual | Total de casos de Uso: 7 Número de casos de uso que deben ser modificados para mejorar el proceso actual: 0 Representa: 0 % |
| | | **100 %** |

**Tabla 3.4** Factor Correctitud Segunda Revisión

| **Factor** | **Métricas Asociadas** | **Valor** |
|---|---|---|
| **Factor Complejidad** | | |
| Factor 4. ¿Los elementos dentro del diagrama están adecuadamente ubicados de manera que facilitan su interpretación? | Métrica 5. Número de elementos del diagrama que requieren reubicación | Total de casos de Uso: 7. Número de casos de uso que requieren reubicación de manera que faciliten su interpretación: 0<br>Representa: 0 % |
| | | **100 %** |

**Tabla 3.5** Factor Complejidad Segunda Revisión

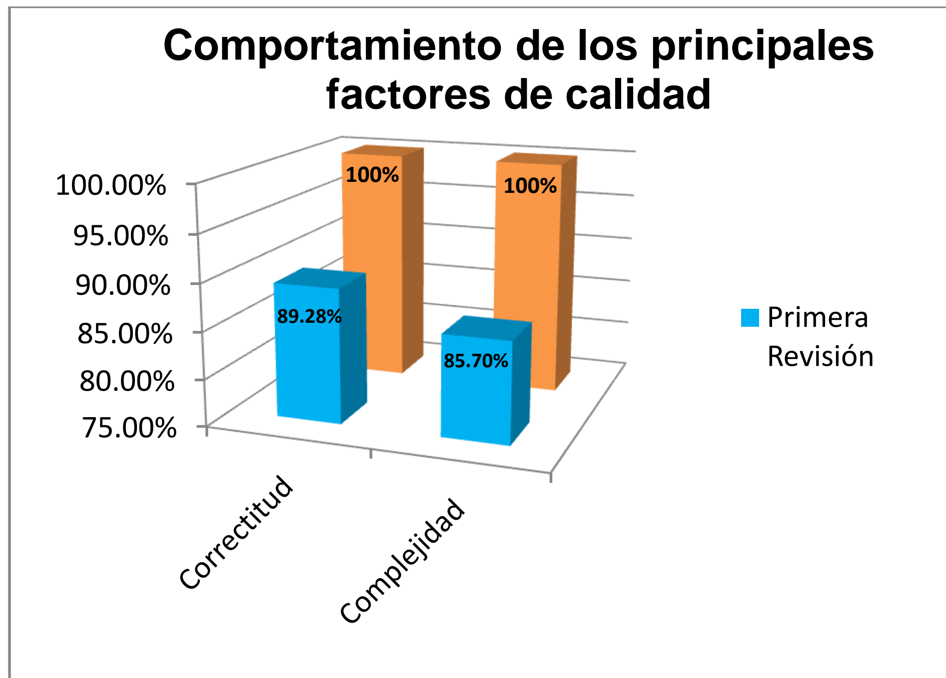



**Figura 3.3** Comportamiento de los principales factores de calidad

El gráfico de la figura 3.3 permite observar las mejorías en el DCUS de la segunda revisión con respecto a la primera, pues luego de haber aplicado las métricas en la primera revisión se obtuvieron los siguientes resultados:

El factor correctitud se vio afectado pues había casos de uso casos que debían ser modificados para adecuarlos a la funcionalidad del sistema.

El factor complejidad se vio afectado pues había casos de uso que requerían reubicación en el DCUS para que facilitaran su interpretación.

Teniendo en cuenta los resultados de la primera revisión se llevaron a cabo algunos cambios en el DCUS, como parte de los mismos fue necesario eliminar casos de uso, así como redefinir algunos ya existentes y agregar nuevas funcionalidades al sistema. Después de aplicar estos cambios se llevó a cabo una segunda revisión en la que ninguno de los factores de calidad se vieron afectados pues de manera general los casos de uso representan una interacción observable por un actor, existe una adecuada separación entre el flujo básico y los flujos alternos; y los nombres de los casos de uso son correctos, pues en todos los casos el mismo es una expresión verbal que describe alguna funcionalidad relevante en el contexto del usuario.

## 3.4. Métricas para el Modelo de Diseño

### 3.4.1. La serie de métricas CK.

Uno de los conjuntos de métricas más ampliamente referenciados han sido los propuestos por Chidamber y Kemerer. Normalmente conocidas como la serie de métricas CK, los autores han propuesto seis métricas basadas en clases para sistemas OO. De esas seis métricas se aplicará la siguiente:

**Árbol de profundidad de herencia (APH)**

Esta métrica se define como la máxima longitud del nodo a la raíz del árbol. A medida que el APH crece es posible que clases de más bajos niveles hereden muchos métodos. Esto conlleva a dificultades potenciales cuando se intenta predecir el comportamiento de una clase. Una jerarquía de clases profunda (el APH largo) también conduce a una complejidad de diseño mayor. Los autores sugieren un umbral de 6 niveles como indicador de un abuso en la herencia en distintos lenguajes de programación.

**Resultado:** A partir de los datos obtenidos después de aplicar la métrica APH se obtuvo que los niveles más altos de herencia sean de 3, lo cual se encuentra dentro del umbral definido para determinar que el diseño no es complejo, no existe un alto acoplamiento y no es de difícil mantenimiento.

### 3.4.2. Métricas Propuestas por Lorenz y Kidd

**Tamaño de clase (TC)**

El tamaño general de una clase puede medirse determinando las siguientes medidas:

- Total de operaciones (tanto heredadas como privadas de la instancia), que se encapsulan dentro de la clase.
- El número de atributos (atributos tanto heredados como privados de la instancia), encapsulados por la clase.

Un TC grande afecta los indicadores de calidad definidos para esta métrica por los especialistas:

**Reutilización:** reduce la reutilización de la clase.

**Implementación:** complica la implementación.

**Responsabilidad:** la clase debe tener bastante responsabilidad.

Las medidas o umbrales para los parámetros de calidad han sido una polémica a nivel mundial en el diseño de sistemas. Algunos especialistas plantean umbrales para estas métricas según se muestra en la tabla 3.6, estos fueron los aplicados en el diseño de este sistema.

| **No de Operaciones y/o atributos** | |
|---|---|
| **TC** | **Umbral** |
| Pequeño | **<=20** |
| Medio | **>20 y <=30** |
| Grande | **>30** |

**Tabla 3.6** Umbrales para el Tamaño de Clase

**Resultado:** Esta métrica fue aplicada en cada uno de los diagramas de clases del diseño agrupados por casos de uso, arrojando los siguientes resultados:

| **Diagrama de Clases del Diseño** | **Cantidad de clases** | **Clasificación de acuerdo a umbrales de la tabla 3.6** |
|---|---|---|
| Almacenar Objetos en Cache | 17 | Medio |
| Encriptar Datos | 12 | Pequeño |
| Gestionar Configuración | 21 | Medio |
| Gestionar Seguridad del Sistema | 17 | Pequeño |
| Iniciar Tarea | 16 | Pequeño |
| Mostrar Mensaje | 7 | Pequeño |
| Navegar a una Vista | 11 | Pequeño |
| Obtener Información de Tarea | 13 | Pequeño |
| Tratar Excepción | 7 | Pequeño |

**Tabla 3.7** Resultados

Estos valores demuestran que los indicadores de calidad reutilización, implementación y responsabilidad no se ven afectados.

## 3.5. Conclusiones

En este capítulo se aplicaron métricas para evaluar la especificación de los requisitos, el DCUS, así como el diseño del sistema, por lo que se puede concluir que:

- Tanto la especificación de los requisitos como el DCUS fueron realizados con calidad. Existe una trazabilidad entre los mismos, pues todos los requisitos están representados en algún caso de uso.
- El sistema presenta un bajo acoplamiento, es de fácil mantenimiento y el diseño del mismo no es complejo, pues la profundidad de los niveles de herencia están acorde con los umbrales.
- Al tener de forma general tamaños de clases pequeños se garantiza el cumplimiento de los indicadores de calidad reutilización, implementación y responsabilidad.

De esta forma ha quedado evaluada la solución obtenida, cumpliendo así con el último de los objetivos específicos propuestos.

## CONCLUSIONES GENERALES

En este trabajo de diploma quedó elaborado un marco teórico de la investigación, se obtuvieron los requisitos y los artefactos de diseño necesarios para los componentes fundamentales del framework y se evaluó la solución obtenida. Dando cumplimiento así a todos los objetivos específicos y al objetivo general propuesto, ya que se obtuvo el diseño de un framework que ayuda a sustentar la arquitectura del Proyecto Registros y Notarias Fase II con cada uno de sus componentes, garantizando el cumplimiento de los atributos de calidad propuestos en la misma.

## RECOMENDACIONES

Se recomienda:

- Continuar con el desarrollo del framework propuesto en sus etapas de implementación y prueba según se propone en la metodología RUP.
- Una vez terminado el producto, se use en los proyectos productivos de la Universidad con el objetivo de sustentar sus arquitecturas.

## BIBLIOGRAFÍA

**Barbacci, M., y otros. 2005.** *Quality Attributes.* s.l. : Carnegie Mellon University. , 2005.

**Bass, L. y Clements, P. 1998.** *Software Architecture in practice.* s.l. : Carnegie Mellon University, 1998.

**Bauer, F. L. 1972.** *Software Engineering Information Processing. Amsterdam.* s.l. : North Holland Publishing Co., 1972.

**Bernárdez, B. y Durán, A. 2008.** *Una propuesta para la verificación de requisitos basada en métricas. Revista de Procesos y Métricas de las Tecnologías de la Información.* 2008.

**Boggs, Wendy y Boggs, Michael. 2002.** *UML with Rational Rose.* 2002.

**Booch, G., Rumbaugh, J. y Jacobson, I. 1999.** *The UML Modeling Language User Guide.* 1999.

**Buschmann, Frank y Meunier, Regine. 1996.** *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* s.l. : Wiley & Sons, 1996.

**Cañar, David.** [En línea] [Citado el: 10 de 2 de 2011.] http://es.wikibooks.org/wiki/Programaci%C3%B3n:C_sharp_NET.

**Ciberaula. 2006.** ¿Qué es Java? [En línea] 2006. [Citado el: 10 de 2 de 2011.] http://java.ciberaula.com/articulo/que_es_java/.

**Evans, Eric. 2004.** *Domain-Driven Design: Tackling Complexity in the Heart of Software.* s.l. : Addison-Wesley, 2004.

**Ferrer Grau, Xavier. 2005.** *Tesis de Doctorado Marco de Integración de la usabilidad en el Proceso de Desarrollo de Software.* s.l. : Universidad Politécnica de Valencia, Facultad de Informática, 2005.

**Fowler, Martin. 2006.** *Patterns of Enterprise Application Architecture.* s.l. : Addison-Wesley, 2006.

**García Ávila, Lourdes. 2000.** *Modelo de evaluación de la calidad para el análisis y diseño orientado a objetos de sistemas informáticos.* s.l. : Universidad Central de las Villas : Tesis de Doctorado, 2000.

**García Rubio, Félix Oscar y Bravo Santos, Crescencio. 2007.** Metodologías de Desarrollo de Softaware. [En línea] 2007. [Citado el: 5 de 12 de 2010.] http://alarcos.inf-cr.uclm.es/per/fgarcia/isoftware/doc/tema9_2xh.pdf.

**IEEE. 1993.** IEEE Standars Collection: Software Engineering. 1993. Ingeniería de Software y su relación con las Herramientas Case. [En línea] 1993. [Citado el: 14 de 1 de 2011.] http://catarina.udlap.mx/u_dl_a/tales/documentos/lis/rea_c_ji/capitulo2.pdf.

**Johnson, Ralph E. y Foote, Brian. 1998.** *Designing Reusable Classes.* s.l. : Journal of Object-Oriented Programming, 1998.

**Kazman, R. y Clements, P. 2001.** *Attribute Based Architectural Styles.* s.l. : Software Engineering Institute, Carnegie Mellon University. Pittsburgh., 2001.

**Kruchten, P. 2000.** *The Rational Unified Process: An Introduction.* s.l. : Addison Wesley, 2000.

**Larman, Craig. 2004.** *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. 3ra ed.* s.l. : Addison Wesley Professional, 2004.

**López Barrio, C. 2005.** *Metodología de desarrollo: Programación Extrema.* s.l. : Programa de Doctorado Ingeniería de Sistemas Electrónicos para Entornos Inteligentes, 2005.

**Markiewicz, Marcus E. y Lucena, Carlos. 2001.** *Object Oriented Framework Development.* 2001.

**Martínez Juan, Francisco Javier.** *Guías de Construcción de Software en Java con Patrones de Diseño.* s.l. : Oviedo : Universidad de Oviedo.

**Nilson, Jimmy. 2006.** *Applying Domain-Driven-Design and Patterns with examples in C# and .Net.* s.l. : Addison-Wesley, 2006.

**Oktaba, Hanna.** Introducción a Patrones. Facultad de Ciencias. UNAM. [En línea] [Citado el: 19 de 1 de 2011.] http://www.mcc.unam.mx/~cursos/Algoritmos/javaDC99-2/patrones.html.

**Pressman, Roger S. 2005.** *Ingeniería de Software. Un enfoque práctico.* s.l. : Madrid : Mc Graw-Hill Interamericana de España S.A.Quinta Edición, 2005.

**Rumbaugh, James, Jacobson, Ivar y Booch, Grady. 2004.** *El Proceso Unificado de Desarrollo de Software.* s.l. : La Habana : Félix Varela, 2004.

**Seco, José Antonio González. 2002.** EL LENGUAJE DE PROGRAMACIÓN C#. [En línea] 2002. [Citado el: 11 de 2 de 2011.] http://www.programacion.com/tutorial/csharp/3/.

**Sequera, Richard A.** Lenguaje C . [En línea] [Citado el: 10 de 2 de 2011.] http://www.monografias.com/trabajos4/lenguajec/lenguajec.shtml.

**Sierra, Fco. Javier Ceballos. 2001.** EL LENGUAJE DE PROGRAMACIÓN C#. [En línea] 2001. [Citado el: 11 de 2 de 2011.] http://www.agapea.com/El-lenguaje-de-programacion-C--n11766i.htm.

**Taligent, Corp. 1994.** *Building Object-Oriented frameworks.* s.l. : Cupertino,CA, 1994.

**Tejera Hernández, Dayana Caridad y Sánchez Echevarría, Leidy Bárbara. 2007.** *Ingeniería de Requisitos para el desarrollo del Sistema de Inventario Almacén (SIGIA) Módulo Nomencladores.* s.l. : La Habana, 2007.

**Zelkovitz, M. V. 1976.** *Principles of Software Engineering and Design.* s.l. : Prentice Hall, 1976.