Universidad de las Ciencias Informáticas

Facultad 3

Título: “Sistema Automatizado para medir la calidad del diseño OO basado en métricas”.

Trabajo de Diploma para optar por el título de Ingeniero en Ciencias Informáticas

Autores:

Carlos Rafael Rodríguez Rodríguez

Yadira Almenares Matos

Tutor:

Ing. Johnny Pérez Acosta

La Habana, mayo de 2011. “Año 53 de la Revolución”.

Resumen

Las métricas de diseño Orientadas a Objeto (OO) son un recurso poderoso para validar artefactos generados durante el importante flujo de Diseño; no obstante resulta engorroso recopilar la información necesaria para aplicarlas, además del gasto de tiempo que implica.

El presente trabajo propone automatizar el uso de métricas de diseño OO, con el objetivo de contribuir en la reducción del tiempo de desarrollo y mejorar la calidad final de los productos. Para lograrlo se realiza el estudio de diferentes metodologías y herramientas de desarrollo, así como de métricas y patrones de diseño. Se realizan los artefactos más importantes de las actividades de Requerimientos, Diseño e Implementación. Finalmente se aplican diferentes pruebas para validar los artefactos obtenidos.

Como resultado se obtuvo un sistema que automatiza el cálculo de 13 métricas de diseño OO de las conocidas como CK y LK entre las que se encuentran: acoplamiento entre objetos (CBO), nivel de profundidad del árbol de herencia (DIT) y número de métodos reemplazados (NMO). Para calcularlas recupera la información necesaria del fichero XML de un diagrama de clases generado por una herramienta CASE. El sistema es extensible, permite la incorporación de plug-ins[1] para aplicar nuevas métricas y para interpretar XML de otras herramientas de modelado.

Palabras claves: calidad, diseño, herramientas CASE, métricas, patrones, plug-in.

[1]Enchufle, agregado. Se utiliza para nombrar fragmentos de código que no tienen funcionalidad por sí mismos, sino que necesitan ser agregados a una aplicación para poder ejecutarse.

Abstract

The object oriented design metrics are a powerful resource to validate artifacts generated during the important flow to design, however is cumbersome to collect the necessary information to apply, besides the time involved.

This document proposes to automate the use of object oriented design metrics, with the objective help reduce development time and improve final product quality. To achieve, it makes the study of different methodologies and development tools and metrics and design patterns. It made the most important artifacts of the activities of Requirements, Design and Implementation. Finally, apply different tests to validate the artifacts obtained.

The result was a system that automates the calculation of 13 metrics of object oriented design known as CK and LK, among which are: coupling between objects (CBO), level of depth of inheritance tree (DIT) and number of overridden methods (NMO).To calculate obtains the required information in an XML file of a class diagram generated by a CASE tool. The system is extensible, allows the addition of plug-ins to implement new metrics and interpret XML files from other modeling tools.

Keywords: CASE tools, design, metrics, patterns, quality, plug-in.

Índice

Resumen II

Abstract III

Introducción 1

Capítulo 1. Fundamentación Teórica. 6

1.1 Bases conceptuales....6

1.1.1 ¿Qué se entiende por norma o estándar?....6

1.1.2 Medición 7

1.1.3 Definición de calidad....7

1.1.4 Factores que determinan la calidad del software 8

1.1.5 El Diseño en el Paradigma Orientado a Objetos (POO)....9

1.2 Estado actual de la automatización de las métricas de diseño OO. 10

1.2.1 Herramientas de control de calidad....12

1.2.2 Herramientas de métricas y de gestión....14

1.2.3 Herramientas de análisis y diseño. 14

Conclusiones parciales del estado actual de la automatización de las métricas de diseño OO. 17

1.3 Métricas para la evaluación de Diseños Orientado a Objeto. 17

1.3.1 Conjunto de métricas CK 17

1.3.2 Conjunto de métricas LK....21

1.4 Patrones de Diseño Orientado a Objetos. 23
1.4.1 Patrones Creacionales 23
1.4.2 Patrones Estructurales 24
1.4.3 Patrones de Comportamiento 24
1.5 Metodologías de desarrollo de software 25
1.5.1 Rational Unified Process (RUP) 25
1.5.2 Agile Unified Process (AUP) 26
1.5.3 Extreme Programming (XP) 26
1.6 Estilos arquitectónicos 27
1.6.1 Estilos de Llamada y Retorno 28
1.7 Herramientas y Tecnologías para el desarrollo de software. 29
1.7.1 Lenguaje Unificado de Modelado (UML) 29
1.7.2 Herramientas CASE 29
1.7.3 Frameworks de desarrollo. 31
1.7.4 Entornos de Desarrollo Integrado (IDE) 32
1.7.5 Lenguaje de programación 33
Selección de metodología, estilo arquitectónico, herramientas y tecnologías para el desarrollo de la solución. 34
Capítulo 2. Solución propuesta. 35
2.1 Sistema Automatizado para medir la calidad del diseño OO basado en métricas. 35

2.2 Arquitectura del sistema 36

2.2.1 Modelo Vista Controlador (MVC) 37

2.2.2 Plug-in 37

2.2.3 Patrones de diseño 38

2.3 Modelo de dominio 40

2.4 Especificación de Requisitos de software 41

2.4.1 Requisitos funcionales 42

2.4.2 Requisitos no funcionales 43

2.5 Definición del modelo de casos de uso del sistema. 44

2.5.1 Actor del Sistema. 44

2.5.2 Descripción reducida de los casos de uso del sistema 44

2.5.3 Diagrama de casos de uso del sistema 47

2.6 Diseño del sistema 48

2.6.1 Modelo de diseño 48

2.6.2 Diagrama de clases del diseño 48

2.6.3 Diagrama de secuencia 49

2.7 Implementación de la solución 50

2.7.1 Estándares de implementación. 50

2.7.2 Tratamiento de errores 51

2.7.3 Seguridad 52

2.8 Conclusiones del capítulo 53

Capítulo 3. Análisis de resultados. 54

3.1 Resultado de las métricas orientadas a objetos. 54

3.1.1 Métricas propuestas por Chidamber y Kemerer. Aplicación al CU Gestionar XML. 54

3.1.2 Métricas propuestas por Lorenz y Kidd. Aplicación al CU Gestionar XML. 56

3.2 Resultados de las Pruebas de Caja Blanca y Caja Negra. 57

3.2.1 Pruebas de Caja Blanca. 57

3.2.2 Pruebas de Caja Negra 62

3.3 Conclusiones del capítulo 66

Conclusiones 67

Recomendaciones 68

Bibliografía 69

Índice de Figuras

Figura 1. Descripción de los procesos que ocurren en el sistema ----------36
Figura 2. Arquitectura del sistema. ----------37
Figura 3. Modelo de Dominio ----------41
Figura 4. Diagrama de Casos de Uso del Sistema ----------47
Figura 5. Diagrama de Clases del Diseño del CU Gestionar XML ----------49
Figura 6. Diagrama de Secuencia del CU Gestionar XML ----------50
Figura 7.Acoplamiento entre objetos. ----------55
Figura 8. Profundidad árbol de herencia. ----------55
Figura 9. Número de Método de Instancia Públicos. ----------56
Figura 10. Número de Variables Instancia. ----------56
Figura 11. Código de prueba ----------59
Figura 12. Grafo de flujo del método----------60

Índice de Tablas

Tabla 1. Resumen de herramientas que aplican métricas de diseño OO. 14
Tabla 2. Descripción reducida del CU Gestionar Proyecto 45
Tabla 3. Descripción reducida del CU Gestionar XML 45
Tabla 4. Descripción reducida del CU Aplicar Métricas 45
Tabla 5. Descripción reducida del CU Configurar Intervalos de Aceptación 46
Tabla 6. Descripción reducida del CU Mostrar Resultados 46
Tabla 7. Descripción reducida del CU Sincronizar modelo con XML fuente 46
Tabla 8. Descripción reducida del CU Cargar Plug-in 47
Tabla 9. Colaboraciones por clases. 55
Tabla 10. Herencias por clases. 55
Tabla 11. PIM por clases. 56
Tabla 12. Casos de prueba para el CU "Configurar Intervalos". 63
Tabla 13. Descripción de variables. 64
Tabla 14. Matriz de Datos Sección 1. 65
Tabla 15. Matriz de Datos Sección 2. 66

Introducción

Desde que el hombre creó sus primeros proyectos, un objetivo común ha estado presente en todos ellos: lograr que lo proyectado tenga la mayor calidad posible. Ante los crecientes avances de la ciencia esa aspiración es aún más relevante, a tal punto que, seguir las más estrictas normas de calidad establecidas internacionalmente es un imperativo si se pretende ser competitivo en el mercado internacional. Según el estándar 9001 de la Organización Internacional para la Estandarización (ISO), la calidad es el grado de acercamiento a las necesidades y expectativas de los consumidores. Cumpliendo las necesidades y expectativas de los consumidores, se consigue satisfacción en el consumidor, que este transmite a su entorno, generando más satisfacción.

Con el incremento de la producción de software hoy en día donde todas las empresas desean estar a la vanguardia, es una necesidad creciente elaborar productos de mayor calidad con menor costo y en el menor tiempo posible. Esto está condicionado por el hecho de que a los clientes no solo les interesa obtener la solución de software, sino que además el resultado esté basado en los más refinados estándares de calidad. La ISO 1926 considera calidad del software al conjunto de cualidades que lo caracterizan y determinan su utilidad y existencia; y puede expresarse en términos de eficiencia, corrección, confiabilidad, mantenibilidad, portabilidad, usabilidad, seguridad e integridad.

La calidad ha de ser medida durante todo el Proceso de Desarrollo de Software haciendo énfasis en las etapas iniciales del mismo; logrando así el ahorro de recursos, tiempo y sobre todo facilitando la creación de un software robusto desde su base.

El diseño es una etapa fundamental en el desarrollo de sistemas OO, por lo que es de gran importancia asegurar su calidad, evitando la propagación de errores en todo el proceso de desarrollo y el alto costo de su corrección. Para aumentar la calidad de un diseño OO es aconsejable aplicar las buenas prácticas o patrones de diseño que proveen soluciones predefinidas a problemas determinados.

Pero la aplicación de estos patrones no garantiza el éxito total del diseño, de manera que resulta imprescindible validarlo para saber si cumple con la calidad esperada y por consiguiente no afectará la calidad final del producto. Una de las técnicas empleadas para evaluar el diseño de software OO es el uso

de métricas. Las métricas son la medida cuantitativa del grado en que un sistema, componente o proceso posee un atributo dado, y tienen como objetivo asegurar la calidad del producto, estimar la efectividad del proceso y mejorar la calidad del trabajo realizado a nivel del proyecto (IEEE, 1993).

El uso manual de las métricas de diseño OO es complejo debido a la gran cantidad de información que se necesita para la ejecución de sus fórmulas, lo que influye en el tiempo y esfuerzo necesarios para su aplicación, por lo que en la mayoría de los casos los equipos de desarrollo no las emplean correctamente, desaprovechando las ventajas que estas proveen.

Para facilitar su uso estas métricas son implementadas por varias herramientas (independientes o integradas con otras) para evaluar la calidad del software. La mayoría de esas herramientas reciben como entrada el código fuente o binario obtenido después de la implementación, lo que no permite medir la calidad en las etapas iniciales del proceso de desarrollo, específicamente durante el diseño.

Existen otras herramientas, que si permiten evaluar el diseño siguiendo reglas del UML[2] que se encargan de verificar aspectos relacionados con la correcta conformación y uso de cada elemento, pero no cuentan con funcionalidades para realizar la medición del diseño basado en métricas.

Estos elementos están presentes actualmente en todo el mundo del desarrollo de software y llegan hasta la industria cubana que está en pleno proceso de consolidación. En ella existe un número creciente de centros destinados a la formación de recursos humanos y a la producción de software propiamente dicha. La Universidad de las Ciencias Informáticas (UCI) es uno de estos centros.

La UCI tiene como uno de sus objetivos crear productos, servicios informáticos y soluciones tecnológicas integrales, tanto para la informatización nacional como para la exportación. Para ello cuenta con varios proyectos productivos agrupados de acuerdo a temáticas afines en diferentes centros de desarrollo.

El Centro de Gobierno Electrónico (CEGEL) cuenta con varios proyectos de desarrollo de software en los que la breve carrera profesional de sus equipos de desarrollo provoca que algunos procesos no se ejecuten eficientemente. Aspectos como la evaluación de la calidad durante la etapa de diseño y la

---

[2] Lenguaje de Modelado Unificado, del inglés Unified Modeling Language. Ver sección 1.7.1.

aplicación de métricas OO aún tienen un tratamiento insuficiente. En esto incide la ausencia de políticas que rijan dicho proceso y la no utilización de herramientas que lo faciliten, lo que unido a lo engorroso que resulta aplicarlas manualmente trae consigo la inadecuada evaluación de la calidad en esta etapa del proceso de desarrollo de software, pudiendo acarrear un retraso en los cronogramas del proyecto y por consiguiente un aumento de los costos del mismo.

Por lo antes expuesto se define como **Problema de la investigación:** ¿Cómo contribuir a facilitar la evaluación basada en métricas de los diagramas de clases del diseño Orientado a Objetos en los proyectos del Centro de Gobierno Electrónico de la Universidad de las Ciencias Informáticas? La investigación tiene como **Objeto de Estudio:** el Proceso de Desarrollo de Software y dentro de este se define como **Campo de acción:** la evaluación de los diagramas de clase de diseño Orientado a Objeto basada en métricas.

Para dar respuesta al problema planteado se define como **Objetivo General:** Desarrollar un sistema que automatice la aplicación de métricas para evaluar la calidad del diseño de software Orientado a Objetos que contribuya a facilitar la evaluación de los diagramas de clases del diseño de los proyectos del Centro de Gobierno Electrónico de la Universidad de las Ciencias Informáticas, y como **Objetivos específicos:**

- ✓ Determinar la influencia del uso de las métricas en la evaluación de diseños Orientados a Objetos así como la eficacia de las herramientas que las implementan. Para definir la necesidad y vigencia de la investigación en cuestión.
- ✓ Seleccionar la metodología, herramientas y tecnologías que favorezcan el desarrollo del sistema.
- ✓ Obtener los artefactos correspondientes a la actividad de modelado. Permitiendo una mejor comprensión del negocio y determinar una solución viable para el desarrollo del sistema.
- ✓ Obtener el sistema informático que automatice la aplicación de las métricas para evaluar la calidad de los diagramas de clases del diseño.
- ✓ Evaluar la calidad del sistema desarrollado, aplicando métricas de diseño Orientado a Objetos así como pruebas de Caja Blanca y Caja Negra para determinar su utilidad y existencia.

Como **Idea a defender** se propone que: Con el desarrollo de un sistema que automatice la aplicación de métricas para evaluar la calidad del diseño de software Orientado a Objetos se contribuirá a facilitar la

evaluación de los diagramas de clases del diseño de los proyectos del Centro de Gobierno Electrónico de la Universidad de las Ciencias Informáticas.

Para dar cumplimiento a los objetivos planteados se trazan las siguientes **tareas de la investigación**:

1. Estudio de las métricas para el diseño de software Orientado a Objeto y selección de las que pudieran ser implementadas en el sistema.
2. Estudio del estado del arte sobre las tendencias en la automatización de las métricas de diseño de software Orientado a Objeto.
3. Selección de la metodología y herramientas que serán empleadas para el desarrollo del sistema.
4. Análisis de los patrones de diseño de software Orientado a Objeto para utilizarlos en el sistema con el fin de sugerir posibles soluciones en modelos de diseño.
5. Selección de la herramienta CASE[3] con la que será compatible el sistema en su primera versión.
6. Estudio de la estructura de los ficheros XML[4] generados por la herramienta CASE seleccionada para poder leer e interpretar los diagramas de clases generados.
7. Definición de los requerimientos funcionales y no funcionales del sistema a desarrollar.
8. Realización del Modelo de Casos de Uso del Sistema.
9. Definición de la arquitectura que tendrá el sistema.
10. Diseño de los diagramas de clases del sistema propuesto.
11. Diseño de los diagramas de secuencias del sistema propuesto.
12. Validación de los diagramas de clases del sistema propuesto mediante métricas de diseño OO.
13. Implementación del sistema propuesto.

[3] Las herramientas CASE (Ingeniería de Software Asistida por Computadora) son diversas aplicaciones informáticas destinadas a aumentar la productividad en el desarrollo de software reduciendo el costo de las mismas en términos de tiempo y dinero. Ayudan durante todo el ciclo de vida del software en tareas como realizar un diseño del proyecto e implementación de parte del código automáticamente con el diseño dado.

[4] Lenguaje de Marcas Extensible, del inglés eXtensibleMarkupLanguage. Es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C) que permite definir la gramática de lenguajes específicos. Se propone como un estándar para el intercambio de información estructurada entre diferentes plataformas.

14. Realización de pruebas unitarias al código del sistema obtenido.

Resultados esperados:

- ✓ Modelo de Dominio.
- ✓ Listado de requerimientos funcionales y no funcionales del sistema.
- ✓ Artefacto Modelo de diseño.
- ✓ Artefacto Realización de casos de usos.
- ✓ Código fuente del sistema.
- ✓ Resultados de las pruebas aplicadas al diseño.
- ✓ Resultados de las pruebas aplicadas al código.

El sistema creado contribuirá a la reducción del tiempo y esfuerzo requeridos para la aplicación de métricas OO en la evaluación de diseños de software y permitirá evaluar el diseño antes de avanzar a las siguientes etapas de desarrollo, evitando que en el futuro aparezcan debilidades en la mantenibilidad del sistema; por lo que fortalecerá el proceso de desarrollo de software.

Estructura del trabajo

El presente trabajo de diploma consta de tres capítulos y una breve introducción al tema de la Evaluación de Diseños de Software.

En el **capítulo 1** se realiza la fundamentación teórica de la propuesta y un estudio del estado del arte de la automatización de las métricas de diseño OO dentro de la Evaluación de Diseños Software.

En el **capítulo 2** se describe la solución propuesta; se desarrolla el análisis, diseño e implementación del sistema para la evaluación de diseños aplicando métricas OO.

En el **capítulo 3** se exponen el resultado de las pruebas hechas al diseño y la implementación del sistema.

Capítulo 1. Fundamentación Teórica.

En este capítulo se analizan brevemente un amplio número de temas con el objetivo de formular la fundamentación teórica de la presente investigación. Inicialmente se muestran varias definiciones y conceptos relacionados con la estandarización, la calidad y los factores que la determinan y el diseño como disciplina del Paradigma Orientado a Objetos (POO). Luego se presenta un estudio del estado del arte de la automatización de las métricas de diseño OO. De igual forma se estudian un grupo de métricas y patrones de diseño con el objetivo de comprender su funcionamiento y definir cuáles podrían ser empleados en la realización del trabajo. Finalmente se estudian las más relevantes metodologías de desarrollo, estilos arquitectónicos, herramientas y tecnologías con el fin de conformar la solución técnica del proyecto.

1.1 Bases conceptuales

1.1.1 ¿Qué se entiende por norma o estándar?

Se considera como entidades de mayor reconocimiento internacional, por sus trabajos y esfuerzos realizados para la normalización, y reconocimiento de la Ingeniería del software a: la Organización Internacional para la Estandarización (ISO[5]), el Instituto de Ingenieros Eléctricos y Electrónicos (IEEE[6]) - Computer Society y el Instituto de Ingeniería de Software (SEI[7]).

Según ISO, un estándar es “un conjunto de acuerdos documentados que contienen especificaciones técnicas u otros criterios precisos para ser usados constantemente, como reglas, lineamientos o definiciones de características; con la finalidad de asegurar que los materiales, productos, procesos y servicios son óptimos para su propósito” (Álvarez, 2004).

Las normas o estándares definen los medios para que todos los procesos se realicen siempre de la misma forma, mientras no surjan ideas para mejorarlos. Son una guía para elevar la productividad y la calidad. Actúan como un modelo, patrón, ejemplo o criterio a seguir. Cada norma tiene un campo de

[5]Del inglés International Organization for Standarization.

[6] Del inglés Institute of Electrical and Electronics Engineers.

[7]Del inglés Software Engineering Institute.

validez que define la aplicación. Por esta razón, un mismo producto puede estar sujeto a varias normas.

El proceso de estandarización o normalización se basa en el trabajo conjunto de todas las partes involucradas: productores, profesionales, usuarios, administración pública, etc. (ISO/IEC, 1996). Además recoge las propuestas de todas las instituciones reconocidas internacionalmente como son los fabricantes, las asociaciones de consumidores, los juristas, los centros de investigación, las entidades de certificación e inspección.

1.1.2 Medición

Aunque los términos medida, medición y métricas se utilizan a menudo indistintamente, existe gran confusión a la hora de referirse a ellos. Dentro del contexto de la ingeniería del software, una medida "proporciona una indicación cuantitativa de extensión, cantidad, dimensiones, capacidad y tamaño de algunos atributos de un proceso o producto" (Pressman, 2002). La medición "es el proceso por el cual los números o símbolos son asignados a atributos o entidades en el mundo real tal como son descritos de acuerdo a reglas claramente definidas" (Fenton, 1991).

Por su parte las métricas pueden definirse como: "La continua aplicación de técnicas basadas en la medición al proceso de desarrollo de software y a sus productos para proveer información administrativa, significativa y oportuna, junto con el uso de esas técnicas para mejorar el proceso y sus productos" (Westfall, 1995).

1.1.3 Definición de calidad

No es difícil encontrar más de una decena de definiciones aportadas por instituciones, estudiosos y organizaciones propias del ámbito tradicional de la prestación de servicios o del suministro de bienes manufacturados.

La Real Academia Española de la Lengua (Española, 2001) define el concepto "calidad" como: Calidad (del lat. Qualitas, -atis y este calco del griego poiothz) .f. Propiedad o conjunto de propiedades inherentes a algo, que permiten juzgar su valor. 2. Buena calidad, superioridad o excelencia. 4. Condición o requisito que se pone en un contrato. 7. Importancia o gravedad de algo.

En el ámbito empresarial también se han esbozado algunos conceptos, tales como:

- ✓ Grado en el que un conjunto de características inherentes cumple con los requisitos (UNE-EN, 2000).
- ✓ El proceso de identificar, aceptar, satisfacer y superar constantemente las expectativas y necesidades de todos los colectivos humanos relacionados con la empresa (clientes, empleados, directivos, propietarios, proveedores y la comunidad) con respecto a los productos y servicios que proporciona (Andersen, 1995).

Para el caso específico del software Pressman (Pressman, 2002) define la calidad como la: Concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos, con los estándares de desarrollo explícitamente documentados, y con las características implícitas que se espera de todo software desarrollado profesionalmente.

1.1.4 Factores que determinan la calidad del software

Los factores de calidad son normalmente atributos de un alto nivel de abstracción como “fiabilidad”, “facilidad de uso”, “facilidad de mantenimiento”, etc.

Los modelos de McCall (McCall, y otros, 1977) y Boehm (Boehm, y otros, 1978) centrados en las características del producto software descomponen estos atributos en otros directamente mesurables. El modelo de calidad del software de McCall define tres aspectos o características importantes de un producto: características operacionales, de modificación (o revisión) y de transición; descomponiendo cada una en factores de calidad de alto nivel que determinan su rigor. Estos son a su vez descompuestos en criterios de calidad. Por último, estos son asociados a un conjunto de atributos directamente mensurables denominados métricas de calidad. Esto permite definir atributos en términos de otros más fáciles de medir. A continuación se describe el primer nivel de descomposición propuesto por McCall:

- ✓ **Características operacionales:**

Fiabilidad: Grado en que se esperaría que un programa desempeñe su función con la precisión requerida.

Eficiencia: Cantidad de código y de recursos de cómputo requeridos para que un programa realice sus funciones.

Facilidad de uso: Esfuerzo necesario para aprender, operar, y preparar las entradas e interpretar las salidas de un programa.

Integridad: Grado en que puede controlarse el acceso al software o a los datos por parte de las personas no autorizados.

Corrección: Grado en que el programa cumple con su especificación y satisface los objetivos que propuso el cliente.

- ✓ **Características de modificación:**

Facilidad de mantenimiento: Esfuerzo requerido para localizar y corregir un error en un programa.

Extensibilidad: Esfuerzo requerido para modificar un programa en operación.

Facilidad de prueba: Esfuerzo que demanda probar un programa de forma que se asegure que realiza la función requerida.

- ✓ **Características de transición:**

Portabilidad: Esfuerzo necesario para transferir un programa desde un entorno hardware y/o software a otro.

Reusabilidad: Grado en que un programa o componente software se puede reutilizar en otras aplicaciones.

Interoperabilidad: Esfuerzo requerido para acoplar un sistema a otro.

Para ser medidas, estas características se apoyan en los atributos internos del producto, aquellos que se pueden medir en términos del propio producto independientemente de su comportamiento. La medición de estos atributos se considera el fundamento para mejorar la calidad de los productos software.

1.1.5 El Diseño en el Paradigma Orientado a Objetos (POO)

Con el reciente auge de la computación en el mundo entero ha aumentado de forma exponencial la demanda de software, evidenciando la necesidad de encontrar técnicas y tecnologías de la Ingeniería de Software que sean cada vez más eficientes. En esa búsqueda de la solución nace el Paradigma

Orientado a Objetos, el mismo basa su filosofía en interacciones entre objetos, los cuales poseen características y funcionalidades propias que pueden ser utilizadas por otros objetos.

Dentro de los procesos a seguir en un proyecto de ingeniería de software se encuentra el Diseño de Software, el cual es definido por Pressman (Pressman, 2002) como proceso iterativo mediante el cual los requisitos se traducen en un "plano" para construir el software.

Todos los métodos de diseño buscan que el software cumpla con cuatro características fundamentales, a saber, abstracción, ocultamiento, independencia funcional y modularidad, pero solo el Diseño Orientado a Objetos (DOO) cuenta con un mecanismo que permite obtenerlas sin demasiada complejidad.

1.1.5.1 Características principales del Diseño Orientado a Objetos:

- ✓ Los objetos son abstracciones del mundo real o entidades del sistema que se administran entre ellas mismas.
- ✓ Los objetos son independientes y encapsulan el estado y la representación de información.
- ✓ La funcionalidad del sistema se expresa en términos de servicios de los objetos.
- ✓ Las áreas de datos compartidas son eliminadas.
- ✓ Los objetos se comunican mediante paso de parámetros.
- ✓ Los objetos pueden estar distribuidos y pueden ejecutarse en forma secuencial o en paralelo.

1.1.5.2 Ventajas del Diseño Orientado a Objetos:

- ✓ Fácil de mantener, los objetos representan entidades auto-contenidas.
- ✓ Los objetos son componentes reutilizables.
- ✓ Para algunos sistemas, puede haber un mapeo obvio entre las entidades del mundo real y los objetos del sistema.

1.2 Estado actual de la automatización de las métricas de diseño OO.

En las últimas décadas se ha trabajado en el área de desarrollo de sistemas para encontrar técnicas que permitan incrementar la productividad y el control de calidad en cualquier proceso de elaboración de software. Hoy en día la tecnología CASE reemplaza al papel y al lápiz por el ordenador para transformar la actividad de desarrollar software en un proceso automatizado.

Tomando como criterio principal la función, Pressman sugiere una taxonomía de herramientas CASE, de las que se analizan solo algunas categorías por pertenecer al dominio del problema planteado.

Herramientas de métricas y de gestión:

Las métricas o herramientas de medidas actuales se centran en características de procesos y productos. Las herramientas orientadas a la gestión se sirven de métricas específicas del proyecto (por ejemplo, LDC/persona-mes, defectos por punto de función) que proporcionan una indicación global de productividad o de calidad. Las herramientas con orientación técnica determinan las métricas técnicas que proporcionan una mejor visión de la calidad del diseño o del código.

Herramientas de control de calidad:

La mayor parte de las herramientas CASE que afirman tener como principal interés el control de calidad son en realidad herramientas de métricas que hacen una auditoria del código fuente para determinar si se ajusta o no a ciertos estándares del lenguaje. Otras herramientas extraen métricas técnicas en un esfuerzo por extrapolar la calidad del software que se está construyendo.

Herramientas de análisis y diseño:

Las herramientas de análisis y diseño hacen posible que el ingeniero del software cree modelos del sistema que vaya a construir. Al efectuar una comprobación de consistencia y validez de los modelos, estas herramientas proporcionan un cierto grado de visión en lo referente a la representación del análisis, y ayudan a eliminar errores antes de que se propaguen al diseño, o lo que es peor, a la propia implementación.

Siguiendo estas clasificaciones, se analizarán algunas herramientas pertenecientes a los dos primeros grupos así como las más utilizadas que se encuentran dentro del grupo de análisis y diseño. Con la intensión de determinar el nivel de aplicabilidad de las métricas de diseño OO en estas.

1.2.1 Herramientas de control de calidad.

A continuación se describen brevemente algunas herramientas o plug-ins que aplican métricas de diseño OO a partir de su combinación con los Entornos de Desarrollo Integrados (IDE[8]). Según las categorías antes mencionadas, estas herramientas se pueden considerar como de control de calidad.

1.2.1.1 Eclipse metrics plug-in.

Esta herramienta es un plug-in que se integra con el entorno de desarrollo Eclipse. Soporta un buen número de métricas y permite la introducción de los valores del intervalo de aceptación. Muestra una tabla con todas las métricas soportadas y un gráfico tridimensional en forma de nube de paquetes, mostrando las dependencias entre ellos. La configuración de este plug-in es una de sus mayores ventajas. Permite establecer recomendaciones en caso de encontrarse los resultados fuera del intervalo. Permite la exportación de los resultados a un documento XML (Metric1.3.6, 2005).

1.2.1.2 CyVix.

Herramienta autónoma desarrollada en Java. Calcula métricas a partir de archivos ***.class***. Permite cambiar los límites de complejidad ciclomática. Esta herramienta calcula un número reducido de métricas de métodos, clases y paquetes. Muestra también el árbol del proyecto. Exporta datos a HTML, XML y TXT (Cyvix, 2006).

1.2.1.3 RefactorIt.

RefactorIt es freeware[9] y está disponible como un plug-in de Eclipse, NetBeans, Sol UN Estudio, JDeveloper, JBuilder o como un programa independiente. Permite auditar el código implementado mostrando cuáles son las debilidades de un proyecto y cómo se pueden corregir. Provee la medición de métricas o refactorizaciones automáticas. Posibilita variar los intervalos de confianza de las métricas y decidir cuáles de ellas se desea calcular. Una vez calculadas las métricas, el plug-in muestra de forma organizada los valores y colorea de rojo aquellos valores que se salgan del rango para poderlos identificar de forma rápida (Refactorit, 2008).

---

[8]Del inglés Integrated Development Environment.Un IDE es un entorno de programación que ha sido empaquetado como un programa de aplicación, es decir, consiste en un editor de código, un compilador, un depurador y un constructor de interfaz gráfica. Los IDE pueden ser aplicaciones por sí solos o pueden ser parte de aplicaciones existentes. Ver sección 1.7.4.

[9] Define un tipo de software no libre que se distribuye sin costo, disponible para su uso y por tiempo ilimitado.

1.2.1.4 JDepend.

JDepend analiza ficheros Java binarios y calcula métricas de diseño para cada paquete de Java. Permite medir automáticamente la calidad de un diseño en términos de extensibilidad, reusabilidad y mantenimiento para gestionar dependencias de paquetes de forma efectiva. Posee interfaz gráfica, textual y XML para visualizar métricas de paquetes JAVA, dependencias y ciclos. La interfaz textual y la XML están pensadas para la integración de los resultados con otras herramientas. Puede ser usado junto a JUnit para crear clases de test que permiten comprobar automáticamente que el código es correcto en base a las métricas definidas (JDepend, 2005).

1.2.1.5 NDepend.

NDpend es una herramienta que simplifica el manejo de código fuente ***.NET***. Permite analizar la estructura del código, especificar reglas de diseño, refactorizar, hacer revisiones de código efectivas y comparar diferentes versiones de código. Facilita ver la estructura y las interrelaciones entre los diferentes componentes de un proyecto de software así como responder a preguntas sobre la calidad u otros aspectos del software en base a los numerosos indicadores o métricas incluidos. Además, permite realizar preguntas sobre el código utilizando CQL (CodeQueryLanguage[10]), un lenguaje de consultas similar a SQL. Es capaz de calcular varias métricas, las cuales pueden operar a diferentes niveles: espacios de nombres, ensamblados, tipos, métodos y campos (NDepend, 2004).

1.2.1.6 Tabla resumen.

La siguiente tabla muestra un resumen de las herramientas descritas anteriormente. Cabe destacar que todas cuentan con representación gráfica, aceptan intervalos y no permiten múltiples intervalos.

	Tipo	Tipo de Licencia	Entrada	Formato exportable	Métricas
Eclipse metricsplug-in	Plug-in (Eclipse)	Common Public	Código Fuente	XML	CK, LK, RM
Cyvis	Autónoma y Plug-in(Ant)	GPL	Códigos Binarios	HTML, XML y TXT	Otras

[10]Lenguaje de consulta de código.

RefactorIt	Autónoma y Plug-in	GPL	Código Fuente	CSV, XML, HTML y TXT	CK, LK, RM
JDepend	Autónoma y Plug-in	BSD License	Códigos Binarios	XML, HTML yTXT.	RM
NDepend	Autónoma y Plug-in	Open Source Prof.	Códigos Fuente y Binarios	XML, HTML, Excel y TXT	CK, LK, RM

Tabla 1. Resumen de herramientas que aplican métricas de diseño OO.

Como se puede apreciar estas herramientas presentan como principal desventaja que reciben como entrada código fuente y/o binario obtenido luego de iniciada la implementación, por lo que no garantizan la evaluación del diseño.

1.2.2 Herramientas de métricas y de gestión.

1.2.2.1 SDMetrics.

Software DesignMetrics (SDMetrics) es una herramienta para medir la calidad de los diseños UML. Calcula un amplio número de métricas, aproximadamente 120, entre ellas: acoplamiento entre componentes, tamaño de los paquetes, complejidad de las clases, etc. Comprueba 130 reglas de diseño UML, por ejemplo: detecta diseños incompletos, incorrectos, redundantes o inconsistentes; encuentra problemas de estilo como dependencias circulares, nombres no adecuados; etc. Cubre todos los tipos de diagrama UML (diagramas de clase, de secuencia, de casos de uso, etc.).Soporta todas las versiones de XMI. Permite comparar dos diseños y que los usuarios puedan definir nuevas métricas y reglas (Wüst, 2002).

Esta herramienta presenta como principal desventaja que es privativa.

1.2.3 Herramientas de análisis y diseño.

Por otro lado existe un grupo de herramientas que soportan las actividades de las disciplinas de análisis y diseño. Como parte de estas actividades incluyen algunas relacionadas con el aseguramiento de la calidad. Desde esta perspectiva serán analizadas algunas de las más utilizadas.

1.2.3.1 Enterprise Architect (EA).

Pruebas:

EA permite crear scripts de pruebas para elementos del modelo. Estas pruebas son:

- ✓ de unidad para elementos que están siendo estructuradas, por ejemplo clases y componentes
- ✓ de integración para evaluar cómo los componentes trabajan juntos
- ✓ de sistema para asegurar que el sistema reúne las necesidades de negocio
- ✓ de aceptación para probar la satisfacción del usuario
- ✓ de escenarios para probar la funcionalidad de la aplicación

Estas pruebas solo se pueden realizar a partir del código generado y no verifican la calidad de los artefactos generados durante el diseño.

Validación del modelo:

Otra de las funcionalidades de EA es la de evaluar los modelos comparándolos con reglas del UML conocidas. También verifica cualquier limitación definida dentro del modelo usando el Lenguaje de Limitación de Objeto (LLO).

Esas reglas se encuentran ordenadas en los siguientes grupos:

- ✓ (Elemento, Relación, Característica, Diagrama): Buena-Formación. Verifica si un elemento, relación, característica o diagrama está bien formado o no. Incluye verificaciones tales como si un elemento es válido en el UML y si un diagrama contiene elementos válidos o no dentro del mismo.
- ✓ Elemento: Composición. Verifica si un elemento de UML contiene hijos válidos, si contiene o no el número correcto de hijos válidos y si el elemento está perdiendo o no algún hijo requerido.
- ✓ (Elemento, Relación, Característica): Valida Propiedad. Verifica si el elemento en cuestión tiene o no las propiedades del UML correctas definidas para el mismo, y si contienen valores incorrectos o conflictivos.
- ✓ (Elemento, Relaciones, Característica): Conformidad OCL[11]. Valida un elemento en comparación con cualquier restricción definida en OCL.

[11] Lenguaje de Limitación de Objeto.

Como se evidencia, estas opciones de validación aunque son útiles no resuelven la problemática identificada. Un modelo puede pasar satisfactoriamente esta validación, lo que significaría que en su elaboración no han sido violadas las reglas del UML, pero esto no garantiza que el diseño cumpla con los requerimientos definidos y que lo haga siguiendo las buenas prácticas existentes.

Métricas:

EA provee un grupo de métricas relacionadas con la gestión del proyecto. Estas comprenden aspectos como: control de cambios, estabilidad, presupuesto, costo, gasto, iteraciones, la planificación, los datos reales, dotación de personal, la dinámica del equipo, la convergencia y los desechos de software entre otros. Pero no cuenta con ninguna funcionalidad que permita aplicar métricas para el diseño orientado a objetos.

1.2.3.2 Visual Paradigm para UML (VP).

Pruebas:

Visual Paradigm permite modelar, escribir y documentar casos de pruebas de forma similar a como lo hace Enterprise Architect. Estos scripts verifican aspectos como: satisfacción del usuario, funcionalidad del sistema y como trabajan juntos los componentes. Pero no garantizan una evaluación del diseño directamente.

Validaciones de UML:

Por otro lado VP tiene implementadas algunas reglas del UML para validar de manera dinámica que determinados errores no se cometan durante el modelado. Estas reglas solo se limitan a verificar aspectos de la notación UML pero no evalúan otros factores de la calidad del modelo.

Métricas:

En un estudio realizado (Scribd, 2010)se asegura que Visual Paradigm apoya el uso de métricas, pero no especifica cómo ni cuáles. Fuera de ello no se encontró ninguna otra fuente que documente este aspecto. En el sitio oficial de Visual Paradigm tampoco existe información al respecto.

Conclusiones parciales del estado actual de la automatización de las métricas de diseño OO.

Lograr las más altas cotas de calidad es un requisito imprescindible para incursionar acertadamente en el mundo de los negocios actuales. Para lograr esos valores de calidad deben estar bien definidos los parámetros que permitan medir el grado en el que se han alcanzado. Según varios autores es imposible controlar o predecir lo que no se puede medir. En la producción de software el uso de las métricas es aún insuficiente. A pesar de que numerosos especialistas han abordado el tema dejando claros los beneficios que traería el uso de métricas, aún los ingenieros de software no han generalizado su uso. Las herramientas CASE dedicadas a la aplicación de métricas presentan varios inconvenientes; la mayor parte de ellas lo hacen a partir de código fuente, dejando atrás la importante etapa del diseño. Varias de ellas presentan además la limitante de ser privativas.

El estudio realizado en los epígrafes anteriores pone de manifiesto la necesidad de desarrollar una herramienta que permita la aplicación de métricas partiendo del diseño, garantizando así la calidad del software desde las etapas iniciales de su desarrollo.

1.3 Métricas para la evaluación de Diseños Orientado a Objeto.

Los modelos de diseño orientado a objetos son fundamentales para la creación de la base de los sistemas informáticos por lo que es de gran importancia validar la calidad de los mismos, evitando fallas durante todo el proceso de desarrollo de software.

Las métricas empleadas para medir la calidad del diseño OO están ajustadas a las características de la POO, por lo tanto se basan en el encapsulamiento, acoplamiento, cohesión, complejidad, polimorfismo y reutilización. Para medir el diseño existen numerosas métricas pero entre las más referenciadas se pueden encontrar la familia de métricas de diseño orientado a objetos propuesta por Chidamber y Kemerer y las orientadas a clases de Lorenz y Kidd.

1.3.1 Conjunto de métricas CK

Chidamber y Kemerer (Chidamber, y otros, 1994) establecen 6 métricas basadas en clases para medir cinco atributos básicos en el diseño orientado a objetos: acoplamiento, complejidad de una clase, reutilización, cohesión y herencia.

- ✓ Métodos ponderados por clase (Weighted Methods per Class –WMC)

- ✓ Profundidad del árbol de herencia (Depth of Inheritance Tree -DIT)
- ✓ Número de hijos (Number of children-NOC)
- ✓ Acoplamiento entre objetos (Coupling Between Object classes-CBO)
- ✓ Respuesta de una clase (Response for a class-RFC)
- ✓ Falta de cohesión en los métodos (Lack of cohesion in methods-LCOM)

1.3.1.1 Métodos ponderados por clase (Weighted Methods per Class -WMC)

WMC mide la complejidad de una clase (Chidamber, y otros, 1994) . Considérese una clase C_1 con los métodos, M_1, M_2..., M_n. Sea c_1, c_2,..., c_n la complejidad estática de los métodos. Entonces:

WMC = $\sum_{i=1}^{n} c_i$

Si todos los métodos son considerados de igual complejidad, entonces c_1 =1 y WMC=n (número de métodos).

Algunos autores simplifican esta métrica asignando 1 a cada método, teniendo nada más en cuenta la cantidad de métodos que tiene una clase. En este caso estaría midiendo el tamaño de una clase y no su complejidad, ya que una clase puede tener pocos métodos pero muy complejos y otra clase puede tener muchos métodos pero muy simples.

Las clases que presenten numerosos métodos complejos serán más complicadas de desarrollar y verificar, así como será más complejo el árbol de herencia. Además tiende a ser más específica de la aplicación, limitando su posibilidad de reutilización (Pressman, 2002). Por lo tanto es deseado obtener un número bajo de WMC. Esta métrica no será utilizada en la presente investigación debido a que en el diagrama de diseño de clases no se tiene una idea clara de la complejidad ciclomática de los métodos.

1.3.1.2 Profundidad del árbol de herencia (Depth of Inheritancetree -DIT).

Es la distancia desde una clase, a la clase raíz del árbol de herencia. Si la clase se encuentra en situación de herencia múltiple, el DIT será la longitud máxima hasta la raíz.

Chidamber y Kemerer (Chidamber, y otros, 1994) proponen esta métrica como medida de la complejidad de una clase, complejidad del diseño y el potencial de rehúso. Esto se debe a que las clases que estén más profundas en el árbol de herencia heredarán más métodos. Por lo tanto mientras

mayor sea el valor de DIT las clases serán más complejas de desarrollar y de mantener, la jerarquía será más profunda haciendo el diseño más trabajoso la probabilidad de detección de fallos será mayor. Por otra parte indica que se pueden reutilizar muchos métodos. Es deseado obtener un número bajo de DIT.

1.3.1.3 Número de hijos (Number of children-NOC).

NOC es el número de subclases inmediatas a una clase en el árbol de herencia, y mide la anchura de una jerarquía de clases.

Un alto valor del NOC indica la alta reutilización del código, pero también mide la posibilidad de haber creado abstracciones erróneas por el uso incorrecto de la herencia, dificultad de hacerle cambios a una clase base ya que afectaría a sus hijos, y es mayor el nivel de pruebas requerido (Chidamber, y otros, 1994). Indica también cuán influente puede ser una clase sobre el diseño del sistema. Por lo tanto sería favorable obtener un número bajo de NOC.

1.3.1.4 Acoplamiento entre objetos (Coupling between object classes-CBO).

El acoplamiento es la dependencia de una clase con varias clases del sistema. Existe dependencia cuando la clase utiliza métodos o atributos de las otras clases (Chidamber, y otros, 1994).

El CBO de una clase es el número de clases a las que ella está relacionada, sin tener en cuenta las relaciones por herencia, y mide la complejidad de la misma. El CBO alto no es deseado porque puede ser perjudicial para el diseño y mide la posible reutilización (mientras más independiente es una clase más fácil es de reutilizar). También un alto valor del mismo reduce el encapsulamiento y da medida de la complejidad de pruebas.

1.3.1.5 Respuesta de una clase (Response for a class-RFC)

RFC obtiene el tamaño del conjunto de respuesta para una clase (Chidamber, y otros, 1994). Este conjunto de respuesta de una clase consiste en el conjunto de métodos que se pueden ejecutar como respuesta a un mensaje recibido por un objeto de la clase (Pressman, 2002). Es el número de métodos locales más el número de métodos llamados por los métodos locales.

Existen variaciones en el cálculo: RFC y RFC’

RFC= M + R (Sólo considera el primer nivel del árbol de llamadas)

RFC'= M + R' (Considera el árbol completo de llamadas)

M = Número de métodos en una clase.

R = Número de métodos remotos llamados directamente por una clase.

R' = Número de métodos remotos llamados recursivamente por una clase considerando el árbol entero de llamadas.

RFC de una clase indica la complejidad de la misma, así como la del diseño. Si su valor es alto las pruebas al sistema y la corrección de errores son dificultosas. Por lo tanto es deseado obtener un número bajo de RFC.

Esta métrica no será utilizada en la presente investigación debido a que en el diagrama de diseño de clases no se conoce las llamadas internas que un método realiza a otro método.

1.3.1.6 Falta de cohesión en los métodos (Lack of cohesion in methods-LCOM)

Chidamber y Kemerer (Chidamber, y otros, 1994) definen LCOM como el número de grupos de métodos de una clase que no acceden a atributos comunes de la misma.

Considérese una clase C_1 con n métodos M_1, M_2,…, M_n. Sea $\{I_j\}$ = el conjunto de variables instancias por el método M_i. Hay n conjuntos tales que $\{I_1\}$,... $\{I_n\}$;

Sea $P = \{(I_i, I_j) \mid I_i \cap I_j = \emptyset\}$, y $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \emptyset\}$. Si todos los conjuntos n $\{I_1\}$,... $\{I_n\}$ son $\emptyset$, entonces $P = \emptyset$.

LCOM = $|P| - |Q|$, si $|P| > |Q|$ ó 0 en otro caso.

Un alto valor de LCOM no es deseado ya que implica falta de cohesión. Una baja cohesión incrementa la complejidad del diseño de clases y por tanto la facilidad de cometer errores durante el proceso de desarrollo. Estas clases pueden diseñarse mejor descomponiéndola en dos o más clases distintas aumentando la cohesión de las clases resultantes.

Esta métrica tampoco será automatizada en la herramienta pues en el diagrama de diseño de clases no se especifican los atributos de la clase a los cuales accederán cada uno de sus métodos.

1.3.2 Conjunto de métricas LK

Lorenz y Kidd (Lorenz, y otros, 1994) separan las métricas basadas en clases en tres grupos:

- ✓ Métricas de tamaño de la clase.
- ✓ Métricas de herencia.
- ✓ Métricas de las características internas de las clases.

Estas métricas están enfocadas a las características internas del diseño orientado a objeto y de esta manera, contribuyen a asegurar la mantenibilidad de los productos de software.

1.3.2.1 Métricas de tamaño

Número de Métodos de Instancia Públicos (PIM): Es el número total de métodos públicos de instancias, es decir los métodos que están disponibles como servicios para otras clases. Esta métrica mide la cantidad de responsabilidad que tiene una clase. Lorenz y Kidd (Lorenz, y otros, 1994) sugieren utilizar esta medida para ayudar en la estimación de la cantidad de trabajo necesario para desarrollar una clase. Es deseado obtener un número bajo de PIM.

Número de Métodos de Instancia (NIM): Se define como la suma de todos los métodos definidos para las instancias de una clase ya sean públicos, protegidos o privados. Esta medida fue definida por Lorenz y Kidd (Lorenz, y otros, 1994) como una medida del tamaño de la clase. Las clases más grandes son más complejas y difíciles de mantener, mientras que las más pequeñas tienden a ser más reutilizable. Es favorable obtener un valor bajo de NIM.

El Número de Variables de Instancia (NIV): Se determina por el número total de variables a nivel de instancia que tiene una clase. Un alto valor de NIV no es deseado debido a que un gran número de variables de instancia puede indicar demasiado acoplamiento con otras clases. Lorenz y Kidd (Lorenz, y otros, 1994) sugieren que las clases son más reutilizables cuando tienen menos variables de instancia.

El Número de Métodos de Clase (NCM): Es el número total de métodos a nivel de clase. Un método de clase es un método que es global para sus instancias. El número de métodos de la clase puede indicar la cantidad de elementos comunes que se manejan para todas las instancias. Este número generalmente debe ser relativamente pequeño en comparación con el número de métodos de instancia (Lorenz, y otros, 1994).

El Número de Variables de Clase (NVV): Es el total de variables de clases que tiene una clase. Los valores de las variables de clase son constantes y son compartidos por todos los objetos de la clase (variables estáticas-“static”). El número medio de variables de clase debe ser bajo. En general debe haber menos variables de clase que variables de instancia (Lorenz, y otros, 1994).

1.3.2.2 Métricas de herencia

El Número de Métodos Reemplazados (NMO): Es el número total de métodos que redefine una subclase. Una subclase se le permite redefinir un método que haya heredado de una de sus superclases. Esto se conoce como reemplazar el método. Se utiliza para medir la calidad del uso de la herencia. Los métodos reemplazados, especialmente en niveles muy profundos de la jerarquía de herencia, pueden indicar un problema en el diseño. Es deseado obtener un valor bajo de NMO.

El Número de Métodos Heredados (NMI): Es el número de métodos que hereda una subclase. Es deseado un número alto de NMI ya que indica la fuerza de la subclase en la especialización. También mide la calidad del uso de la herencia.

El Número de Métodos Añadidos (NMA): Se define como el número total de métodos que se definen en una subclase (Lorenz, y otros, 1994). Las subclases deben definir nuevos métodos, que extiende el comportamiento de las superclases. El número de nuevos métodos por lo general debería disminuir a medida que se mueven a través de las capas de la jerarquía. Igual que las anteriores mide la calidad de uso de la herencia.

El Índice de Especialización para una clase (SIX): Es el número de métodos redefinidos multiplicado por el nivel de anidamiento de la clase en la jerarquía y dividido entre el número total de métodos.

SIX = Nº de métodos redefinidos * Anidamiento en la jerarquía/ Nº total de métodos.

Mide el grado en que una subclase redefine el comportamiento de una superclase. SIX puede indicar cuándo hay demasiados métodos redefinidos, de tal forma que las abstracciones pueden no ser apropiadas y sea necesario reemplazar su comportamiento. Una subclase debería extender el comportamiento de la superclase con métodos nuevos más que reemplazar o borrar comportamiento a través de redefiniciones.

Lorenz y Kidd (Lorenz, y otros, 1994) proponen un valor del 15% para ayudar a identificar superclases que no tienen mucho en común con sus subclases. Cuanto más profundizamos en la jerarquía, más especializada ha de ser la subclase.

1.3.2.3 Métricas de características internas de una clase.

El Promedio de Parámetros por Método (APPM): Se determina como el cociente entre el número total de parámetros por método y el número total de métodos.

APMM = Nºtotal de parámetros por métodos/ Nºtotal de métodos.

1.4 Patrones de Diseño Orientado a Objetos.

Los patrones de diseño OO buscan codificar y hacer reutilizables un conjunto de principios a fin de diseñar aplicaciones de alta calidad. Se aplican en principio sólo en la fase de diseño, aunque en los últimos tiempos se ha comenzado a definir y aplicar patrones en las otras etapas del proceso de desarrollo, desde la concepción arquitectónica inicial hasta la implementación del código.

Los patrones de diseño OO tiene cuatro elementos importantes (Gamma, y otros, 1995): nombre del patrón, problema, solución y consecuencias.

Debido a la diversidad y variedad de los patrones de diseño es necesario agruparlos para su organización:

1.4.1 Patrones Creacionales

Abstraen el proceso de creación de instancias, haciendo un sistema independiente de cómo se crean, se componen y se representan sus objetos (Gamma, y otros, 1995).

- ✓ ***Factory Method (Método Fabricación):*** centraliza en una clase constructora la creación de objetos de un subtipo de un tipo determinado, ocultando al usuario la casuística para elegir el subtipo que crear.

- ✓ ***Abstract Factory (Fábrica Abstracta):*** permite trabajar con objetos de distintas familias de manera que estas no se mezclen entre sí y haciendo transparente el tipo de familia concreta que se esté usando.

- ***Builder (Constructor):*** abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto.
- ***Prototype (Prototipo):*** crea nuevos objetos clonándolos de una instancia ya existente.
- ***Singleton (Instancia Única):*** garantiza la existencia de una única instancia para una clase y la creación de un mecanismo de acceso global a dicha instancia.

1.4.2 Patrones Estructurales

Describen como las clases y objetos pueden ser combinados para formar grandes estructuras y proporcionar nuevas funcionalidades (Gamma, y otros, 1995).

- ***Adapter (Adaptador):*** adapta una interfaz para que pueda ser utilizada por una clase que de otro modo no podría utilizarla.
- ***Bridge (Puente):*** desacopla una abstracción de su implementación.
- ***Composite (Compuesto):*** permite tratar objetos compuestos y simples de forma homogénea.
- ***Decorator (Decorador):*** añade funcionalidad a una clase dinámicamente.
- ***Facade (Fachada)*:** provee de una interfaz unificada simple para acceder a una interfaz o grupo de interfaces de un subsistema.
- ***Proxy (Apoderado):*** mantiene un representante de un objeto.

1.4.3 Patrones de Comportamiento

Se utilizan para caracterizar el modo en que las clases y objetos interactúan, y se reparten la responsabilidad (Gamma, y otros, 1995).

- ***Interpreter (Interprete):*** define una gramática para un lenguaje dado, así como las herramientas necesarias para interpretarlo.
- ***Chain of Responsibility (Cadena de responsabilidad):*** permite establecer la línea que deben llevar los mensajes para que los objetos realicen la tarea indicada.
- ***Command (Comando):*** encapsula una operación en un objeto, permitiendo ejecutar dicha operación sin necesidad de conocer el contenido de la misma.

- ***Iterator (Iterador):*** permite realizar recorridos sobre objetos compuestos independientemente de la implementación de estos.
- ***Mediator (Mediador):*** define un objeto que coordine la comunicación entre objetos de distintas clases, pero que funcionan como un conjunto.
- ***Memento (Recuerdo):*** permite volver a estados anteriores del sistema.
- ***Observer (Observador):*** define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado, se notifiquen y actualicen automáticamente todos los objetos que dependen de él.
- ***State (Estado):*** permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.
- ***Visitor (Visitador):*** permite definir nuevas operaciones sobre una jerarquía de clases sin modificar las clases sobre las que opera.
- ***Strategy (Estrategia):*** define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables.

1.5 Metodologías de desarrollo de software

1.5.1 Rational Unified Process (RUP)

RUP es un proceso de desarrollo de software dirigido por casos de uso, centrado en la arquitectura, iterativo e incremental. Es una metodología orientada a objetos que utiliza UML[12] como lenguaje de modelado (Jacobson, y otros, 2000).

Divide en 4 fases el desarrollo del software:

- Inicio: el objetivo en esta etapa es determinar la visión del proyecto.
- Elaboración: aquí el objetivo es determinar la arquitectura óptima.
- Construcción: en ella el objetivo es obtener la capacidad operacional inicial.

---

[12] Lenguaje de Modelado Unificado.

- ✓ Transición: el objetivo es obtener la solución del proyecto.

Cada una de estas fases se desarrolla de forma iterativa, es decir, reproduce el ciclo de vida en cascada a menor escala. Los objetivos de una iteración se establecen en función de la evaluación de las iteraciones anteriores.

1.5.2 Agile Unified Process (AUP)

AUP es una versión simplificada de Rational Unified Process (RUP) que describe un enfoque simple y fácil de entender para el desarrollo de software usando técnicas ágiles y conceptos que aún se mantienen vigentes en RUP. Aplica técnicas ágiles como el Desarrollo Dirigido por Pruebas (TDD[13]), Desarrollo Dirigido por Modelado Ágil (AMDD[14]), administración ágil de cambios, y refactorización de bases de datos para mejorar la productividad. El ciclo de vida de AUP es en serie en lo grande e iterativo en lo pequeño, liberando entregables incrementales en el tiempo (Ambler, 2005).

Como se observa en la figura 2, AUP consta de 4 fases (Inicio, Elaboración, Construcción y Transición) y 7 disciplinas (Modelado, Implementación, Prueba, Despliegue, Configuración de la Administración, Gestión del proyecto y Entorno).

Entre los principales roles que intervienen en un proyecto AUP se encuentran: Modelador Ágil, Desarrollador, Administrador del proyecto, Examinador y Administrador de pruebas. Estos roles desempeñan diferentes actividades en una o varias disciplinas, entre esas actividades están: modelado de requerimientos, modelado de la arquitectura, prototipado de interfaces de usuario, análisis y diseño por modelo de lluvia de ideas.

1.5.3 Extreme Programming (XP)

XP es la primera metodología ágil y la que le dio conciencia al movimiento actual de metodologías ágiles. Está centrada en potenciar las relaciones interpersonales como clave para alcanzar el éxito. Para lograrlo, promueve el trabajo en equipo, se preocupa por el aprendizaje de los desarrolladores y propicia un buen clima de trabajo. Se basa en la retroalimentación continua cliente-equipo de desarrollo, la simplicidad en las soluciones implementadas y la disposición para asumir los cambios. Es

[13] Del inglés Test Driven Development. Permite escribir los casos de prueba antes de comenzar a escribir el código.

[14] Del inglés Agile Model Driven Development. Permite crear modelos ágiles que son lo suficientemente buenos para impulsan el desarrollo.

especialmente adecuada para proyectos con requisitos imprecisos y muy cambiantes, donde existe un alto riesgo técnico (Amaro Calderón, y otros, 2007).

El ciclo de desarrollo consta (de manera general) de los siguientes pasos:

1. El cliente define el valor de negocio a implementar.
2. El programador estima el esfuerzo necesario para su implementación.
3. El cliente selecciona qué construir, según sus prioridades y las restricciones de tiempo.
4. El programador construye ese valor de negocio.
5. Vuelve al paso 1.

Algunos de los artefactos desarrollados en un proyecto XP son: las historias de usuarios (para especificar los requisitos del software), las tarjetas C-R-C (para definir las clases, sus responsabilidades y colaboradores) y las tareas de programación. Entre los principales roles que propone se encuentran: Programador, Cliente y Encargado de pruebas.

Además propone varias prácticas con el fin de lograr el éxito final del proyecto, estas son:

Pruebas, refactorización, programación en pareja, cliente in-situ, estándares de programación, 40 horas por semana, integración continua, propiedad colectiva del código, diseño simple, metáfora, entregas pequeñas, el juego de la planificación.

1.6 Estilos arquitectónicos

La definición más usada de Arquitectura de Software (AS) es de la IEEE Std 1471-2000, que plantea: “La Arquitectura del Software es la organización fundamental de un sistema formada por sus componentes, las relaciones entre ellos y el contexto en el que se implantarán, y los principios que orientan su diseño y evolución” (IEEE, 2000).

La primera definición explícita de estilos arquitectónicos fue propuesta por Dewayne Perry y Alexander Wolf en octubre de 1992 (Wolf, y otros, 1992). Los estilos casi siempre se usan combinados; cada capa o componente puede ser internamente de un estilo diferente.

Según Shaw y Garlan (Shaw, y otros, 1996) los estilos se agrupan en cinco clases fundamentales y unos diez ejemplares. A continuación se describen algunos de los estilos de la clasificación Llamada y Retorno por su importancia en la elaboración de la solución.

1.6.1 Estilos de Llamada y Retorno

Son los estilos más generalizados en sistemas de gran escala. Permite dividir la arquitectura en dos partes, la primera representa la interfaz del usuario y la segunda contiene la lógica de negocio. Según lo expuesto por Reynoso (Reynoso, 2004) esta familia de estilos enfatiza la modificabilidad y la escalabilidad.

1.6.1.1 Modelo-Vista-Controlador (MVC)

Separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos: Modelo, Vista y Controlador (Burbeck, 1992). MVC se usa frecuentemente para construir aplicaciones web.

Reynoso expone como sus principales ventajas: el soporte de múltiples vistas y la amplia adaptación al cambio. Al mismo tiempo menciona como desventajas: el aumento de la complejidad de la solución como consecuencia de nuevos niveles de no direccionamiento y los costos de actualizaciones frecuentes: desacoplar el modelo de la vista no significa que los desarrolladores del modelo puedan ignorar la naturaleza de las vistas.

1.6.1.2 Arquitecturas en Capas

Este es un estilo estructurado jerárquicamente por capas que suelen ser entidades complejas, compuestas de varios paquetes o subsistemas, donde cada una proporciona servicios a la inmediatamente superior y actúan sobre los operadores de la inmediatamente inferior.

Al decir de Reynoso: “el estilo soporta un diseño basado en niveles de abstracción crecientes, lo cual a su vez permite a los implementadores la partición de un problema complejo en una secuencia de pasos incrementales. En segundo lugar, el estilo admite muy naturalmente optimizaciones y refinamientos. En tercer lugar, proporciona amplia reutilización.”

Por otro lado puede resultar difícil definir qué componentes ubicar en cada una de las capas. En ocasiones no se logra la contención del cambio en las mismas capas y se requiere una cascada de cambios, lo que acarrea una pérdida de eficiencia. Otra de sus posibles desventajas es el trabajo innecesario por parte de capas más internas o la redundancia entre varias de ellas.

1.6.1.3 Arquitecturas Orientadas a Objetos

Los componentes de este estilo son las instancias de los tipos de datos abstractos y se basan en principios Orientados a Objetos. Son asimismo las unidades de modelado, diseño e implementación; los objetos y sus interacciones son el centro de las incumbencias en el diseño de la arquitectura.

En este estilo, un objeto es ante todo una entidad reutilizable en el entorno de desarrollo. Además permite modificar la implementación de un objeto sin afectar a sus clientes, así como descomponer problemas en colecciones de agentes en interacción. Entre sus limitaciones están el hecho de que, para poder interactuar con otro objeto a través de una invocación de procedimiento, se debe conocer su identidad y que cuando se modifica un objeto se deben modificar también todos los objetos que lo invocan.

1.7 Herramientas y Tecnologías para el desarrollo de software.

1.7.1 Lenguaje Unificado de Modelado (UML)

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 (Rumbaugh, y otros, 1999). Su objetivo es representar el conocimiento acerca de los sistemas que se pretenden construir y las decisiones tomadas durante su desarrollo, tanto los representados por diagramas estáticos (Casos de Uso, diagrama de clases, etc.) como los dinámicos (Diagramas de actividades, interacción, etc.).

1.7.2 Herramientas CASE

Las herramientas CASE ayudan a los gestores y practicantes de la ingeniería del software en todas las actividades asociadas a los procesos de software. Automatizan las actividades de gestión de proyectos, gestionan todos los productos de los trabajos elaborados a través del proceso, y ayudan a los ingenieros en el trabajo de análisis, diseño y codificación. Las herramientas CASE se pueden integrar dentro de un entorno sofisticado (Pressman, 2002).

Con este enfoque que persigue mejorar la calidad del software e incrementar la productividad en el proceso de desarrollo, se plantean los siguientes objetivos:

- ✓ Permitir la aplicación práctica de metodologías, lo que resulta muy difícil sin emplear herramientas.

- ✓ Facilitar la realización de prototipos y el desarrollo conjunto de aplicaciones.
- ✓ Simplificar el mantenimiento del software.

1.7.2.1 Visual Paradigm

Visual Paradigm para UML es una herramienta profesional que soporta el ciclo de vida completo del desarrollo de software. Cuenta con un editor de detalles de casos de uso. Permite representar todos los tipos de diagramas de clases, generar código y documentación desde diagramas. Soporta notación UML 2.x, tiene capacidad de ingeniería inversa y directa así como importar y exportar ficheros XMI. Proporciona abundante información sobre UML. Presenta licencia gratuita y comercial. Es fácil de instalar y actualizar, es compatible entre ediciones. Además admite otras herramientas y plug-in de modelado UML para varias plataformas.

1.7.2.2 Enterprise Architect

Enterprise Architect proporciona un conjunto de funcionalidades que cubren todo el proceso de desarrollo de software 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. EA soporta UML como lenguaje para definir los distintos modelos de un proyecto.

EA soporta la generación e ingeniería inversa de código fuente para muchos lenguajes, incluyendo C++, C#, Java, Delphi, VB.Net, Visual Basic, Action Script y PHP. También se puede exportar rápidamente la documentación de los modelos creados en formato RTF (Rich Text Format) y a Word para su posterior personalización. Permite exportar e importar proyectos en formato XMI. Trae implementados los 24 patrones GoF[15].

1.7.2.3 Rational Rose

Es un instrumento operativo conjunto que utiliza el Lenguaje Unificado (UML) como medio para facilitar la captura de dominio de la semántica, la arquitectura y el diseño. Este software tiene la capacidad de: crear, ver, modificar y manipular los componentes de un modelo.

[15] Gang of Four. Nombre con el que es conocido el grupo de autores que los creó. Patrones de diseño orientado a objeto. Se clasifican en Estructurales, Creacionales y de Comportamiento.

No es gratuito, provee ingeniería de código (directa e inversa) para una amplia gama de lenguajes de programación y de bases de datos. Admite la integración con otras herramientas de desarrollo (IDEs) y trae implementados 20 de los patrones GoF. Requiere de otras herramientas para soportar la disciplina de Pruebas y para generar documentación.

1.7.3 Frameworks[16] de desarrollo.

1.7.3.1 Plataforma .Net

Microsoft.NET Framework es una plataforma de desarrollo de software que brinda la posibilidad de conectar una gran variedad de tecnologías de uso personal y de negocios. Está constituido por compiladores/traductores de diferentes lenguajes de programación, bibliotecas de clases, herramientas de desarrollo y una máquina virtual encargada de la ejecución del código fuente (Microsoft Coportation, 2005).

Entre sus características más notorias se encuentran:

- ✓ Existen más de 30 lenguajes adaptados a .Net, desde los más conocidos como C# (C Sharp), Visual Basic o C++, hasta otros lenguajes menos conocidos como Perl o Cobol. Esta característica posibilita que sea utilizado por una gran diversidad de programadores.
- ✓ Posee una gran biblioteca de clases básicas, permitiendo la creación de cualquier tipo de aplicación en tiempos increíblemente cortos, dándole a su vez mayor robustez y estabilidad.
- ✓ La máquina virtual provee a las aplicaciones de un entorno de ejecución seguro para el código.
- ✓ La plataforma .NET es hoy una de las más aceptadas por la comunidad de desarrolladores, universidades y empresas de software; además marcha a la cabeza en los adelantos de su área. Esta plataforma permite realizar aplicaciones sin necesidad de pagar licencia por su uso.

1.7.3.2 Plataforma J2EE

J2EE (Java 2 Enterprise Edition) plataforma que define como estándar un grupo de especificaciones, que deben ser seguidas para desarrollar el producto, para ello es necesario adquirir una serie de

---

[16] Marco de trabajo. Estructura conceptual y tecnológica de soporte, definida normalmente con artefactos o módulos de software concretos, a partir de la cual otro proyecto de software puede ser organizado y desarrollado.

recursos como el JRE (Java Runtime Environment), JDK (librerías), servidores web, entornos de programación, entre otros, que en su conjunto permitirán desarrollar las aplicaciones. Soporta un único lenguaje que es Java, utilizado para el desarrollo de todos los componentes y compilado por un código intermedio denominado Byte Codes que se ejecuta en un entorno de ejecución llamado JRE para transformar el lenguaje intermedio a código propio de la máquina en la que se corre la aplicación. Brinda soporte para múltiples sistemas operativos, debido a la portabilidad, o posibilidad de ejecutar las aplicaciones desarrolladas en cualquier sistema operativo y/o máquina del mercado. Utiliza múltiples productos lanzados al mercado que ofrecen entornos de desarrollo adecuados, tales como NetBeans de Sun, Visual Café de WebGain, Eclipse, entre otros. De este modo, se ha desarrollado a un nivel exponencial la plataforma y los clientes tienen la posibilidad de escoger entre una gran cantidad de opciones.

1.7.4 Entornos de Desarrollo Integrado (IDE)

1.7.4.1 Sharpdevelop

SharpDevelop es un IDE gratuito bajo una licencia de código abierto para los lenguajes de C#, VB.NET, Boo, IronPython, IronRuby y F#, que trabaja con diferentes plataformas como .Net y Mono. Tiene diversas funcionalidades entre las cuales se encuentra que permite editar código con facilidad ya que dispone, al igual que el entorno de Microsoft, de una función que autocompleta las funciones y muestra las diversas opciones de cada una de ellas. También permite editar formularios rápidamente y de forma visual. Además tiene un completo entorno de depuración donde se puede ejecutar los programas paso a paso, y asignar puntos de ruptura en el código.

Además de estas características básicas de cualquier IDE moderno brinda herramientas de mucha utilidad para cualquier programador, como son:

- ✓ Diseño de Formas Windows.
- ✓ Migración entre C# y Visual Basic .NET.
- ✓ NUnit integrada para pruebas.
- ✓ Previsualización de documentación XML.
- ✓ Analizador para ensamblado FxCorp.
- ✓ Integración eficiente para aprovechar las computadoras multi-core.

- ✓ Soporte de Subversión integrado.
- ✓ Generación automática de documentación Sandcastle y SHFB.

1.7.4.2 Visual C# Express Edition

Visual C# Express es una herramienta que facilita la creación de aplicaciones orientadas a objetos utilizando .NET Framework y el lenguaje de programación C#. Es sencilla de instalar y se destaca por su facilidad de uso, permitiendo mejorar la productividad. (Microsoft Coportation, 2005)

Otras características de Visual C# 2008 Express Edition son:

- ✓ Entorno de programación estable y fácil de usar.
- ✓ Diseño de aplicaciones de Windows Presentation Foundation (WPF).
- ✓ Posibilidad de usar gran cantidad de código predefinido.
- ✓ Soporte para realizar diagnósticos y optimización del sistema.

1.7.4.3 Visual Studio 2005

Microsoft Visual Studio es un entorno de desarrollo integrado para sistemas operativos Windows que soporta varios lenguajes de programación tales como C++, C#, J#, ASP.NET y Visual Basic, e integrado con la plataforma .Net Framework que le proporciona acceso a tecnologías claves para el desarrollo simplificado de las aplicaciones. Es un conjunto completo de herramientas de desarrollo que permite diseñar, crear, depurar e implementar aplicaciones Web ASP.NET, Servicios Web XML, aplicaciones de escritorio y aplicaciones móviles, y admite que dichas aplicaciones se comuniquen entre sí y que sean desarrolladas en diferentes lenguajes. De forma general se puede decir que este IDE es uno de los más aceptados por el mercado mundial independientemente de sus elevados costos de licencia. (Microsoft Coportation, 2005)

1.7.5 Lenguaje de programación

C# o C Sharp es un lenguaje de programación que está diseñado y optimizado para la plataforma .NET. Es un lenguaje orientado a objetos, simple, elegante y con una gran seguridad en el tratamiento de tipos. Cuenta con una serie de facilidades que le brinda la integración a la plataforma .NET y permite la implementación de un amplio grupo de patrones de arquitectura. (Microsoft Coportation, 2005)

Selección de metodología, estilo arquitectónico, herramientas y tecnologías para el desarrollo de la solución.

Tomando como base lo analizado en los epígrafes anteriores se define la siguiente propuesta de solución técnica, que regirá el desarrollo de este trabajo de diploma en sus fases posteriores.

Como metodología de desarrollo de software se empleará Agile Unified Process (AUP) por combinar las ventajas de RUP (robusta) con técnicas agiles. Esta metodología permite seleccionar entre todos los artefactos de RUP solo aquellos que se ajusten a las necesidades del equipo y a pesar de ser una metodología ágil presta gran importancia a la arquitectura del sistema.

Se analizaron los principales estilos arquitectónicos que se encuentran bajo la clasificación Llamada y Retorno y se propone emplear el MVC. Este estilo permite la construcción de sistemas escalables, débilmente acoplados y de fácil evolución. La separación de sus capas facilita el trabajo en equipo, sustituir la implementación de una capa sin afectar al resto del sistema y el mantenimiento en caso de errores al estar la estructura y el flujo de la aplicación definidos de forma más clara.

Para el modelado se selecciona el lenguaje UML 2.1 por ser el lenguaje que emplea la metodología seleccionada. Este lenguaje será empleado en la herramienta CASE Enterprise Architect 7.0. Esta herramienta está diseñada para ayudar a construir software robusto y fácil de mantener. Es multiusuario, con seguridad y administración de permisos incorporada. Permite generación de código fuente e ingeniería inversa para el lenguaje a utilizar en las fases posteriores del ciclo de desarrollo.

Entre los frameworks o plataformas estudiadas se selecciona Microsoft.NET 4.0, por las facilidades que brinda para la comunicación entre las aplicaciones. Hace mucho más sencillo el desarrollo de aplicaciones al encontrarse integrado en él un conjunto de lenguajes, herramientas y servicios que facilitan en gran medida el trabajo. Se propone como IDE SharpDevelop 4.0 debido a que es perfectamente compatible con la plataforma seleccionada. Es una herramienta gratis y de código abierto, y brinda funcionalidades que facilitan el desarrollo de aplicaciones y que proporcionan un mejor trabajo con C#, lenguaje de programación con el que el equipo de desarrollo está altamente familiarizado característica que lo convierte en la opción más viable para la implementación del sistema.

Capítulo 2. Solución propuesta.

En el presente capítulo se presenta una descripción de la solución propuesta para automatizar la aplicación de métricas de diseño orientado a objetos. Como parte de ello se describen los principales procesos y se presenta la arquitectura definida para el sistema, especificando que patrones de diseño se emplean. Así mismo se definen los artefactos comprendidos en la fase de Modelado de la metodología escogida y argumentada en el capítulo 1. Entre ellos se encuentra el modelo de dominio, para determinar los principales conceptos que intervienen en los procesos y sus relaciones. Se especifican los requerimientos tanto funcionales como no funcionales, el diagrama de caso de usos del sistema y los diagramas de clases del diseño y sus diagramas de secuencias. Finalmente se analizan aspectos relacionados a la implementación de la solución como son los estándares empleados y el tratamiento que reciben los errores y la seguridad.

2.1 Sistema Automatizado para medir la calidad del diseño OO basado en métricas.

El sistema propuesto simula los procedimientos que en la práctica se llevan a cabo para aplicar las métricas. A grandes rasgos esos procedimientos son:

- ✓ Recopilación de la información.
- ✓ Selección de las métricas a aplicar.
- ✓ Definición de los intervalos de aceptación para cada métrica.
- ✓ Cálculo de las métricas.
- ✓ Análisis de resultados (comparar contra intervalos).

Para lograrlo, el sistema recopila la información desde un XML generado por la herramienta de modelado empleada y que es cargado por el usuario. Dicho XML representa la estructura de un Diagrama de Clases del Diseño e incluye detalles sobre las clases y sus componentes.

Así mismo el sistema muestra las métricas disponibles para aplicar permitiendo seleccionar las deseadas y configurar sus intervalos de aceptación. De igual modo calcula las métricas seleccionadas

y muestra los resultados luego de compararlos con los intervalos definidos anteriormente, indicado las deficiencias encontradas.

Figura 1. Descripción de los procesos que ocurren en el sistema

2.2 Arquitectura del sistema

La Arquitectura de Software es una disciplina reciente en el mundo del desarrollo de software pero se ha demostrado que representa un elemento de vital importancia dentro del ciclo de desarrollo del mismo. Establece todos los fundamentos para que el equipo de desarrollo trabaje en una línea común que permita alcanzar los objetivos y necesidades del sistema.

Se requiere de una arquitectura robusta, que guíe el proceso de desarrollo y que defina de manera abstracta los componentes que lleven a cabo alguna tarea, sus interfaces y la comunicación entre ellos. Debe ser bien definida y no presentar dificultades, para evitar que el sistema desarrollado no cumpla con los requerimientos determinados y fracase cuando se encuentre en explotación.

Con el objetivo de no incurrir en fallos y construir los cimientos de un sistema robusto, seguro, reutilizable, buscando siempre un bajo acoplamiento y alta cohesión entre los componentes del mismo, se han definido una serie de patrones arquitectónicos que son aplicados atendiendo a disímiles características del software.

2.2.1 Modelo Vista Controlador (MVC)

El patrón arquitectónico MVC presenta múltiples ventajas ya mencionadas en el capítulo 1. Es un estilo muy común para construir software de gestión debido a las facilidades que brinda, pues permite la reutilización y la independencia entre las capas, facilita la estandarización, la utilización de los recursos y la administración.

Las capas definidas en el sistema propuesto son:

- ✓ **Vista**: Es donde se encuentran los formularios del sistema, permitiendo el intercambio de información del usuario con la aplicación.
- ✓ **Modelo**: Contiene las entidades representativas del negocio.
- ✓ **Controlador o Capa lógica**: Contiene los subsistemas y clases encargadas de realizar la mayoría de las funcionalidades que solicita el cliente. En esta capa se encontrará el ensamblado "Extensible.dll" que es el que define el comportamiento de los posibles ensamblados adicionados al sistema en forma de plug-in. Y contiene además los plug-in implementados por terceros.

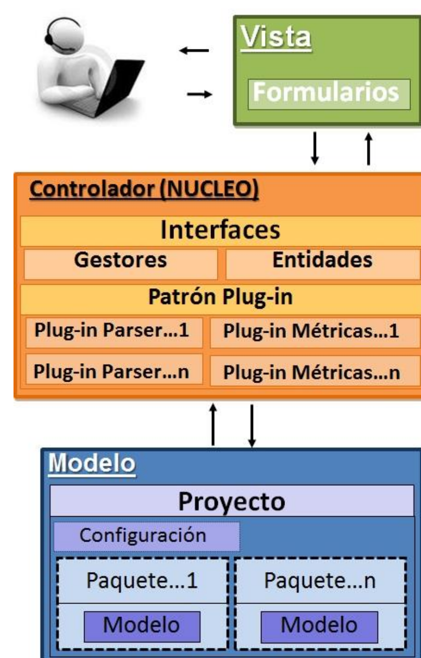

Figura 2. Arquitectura del sistema.

2.2.2 Plug-in

El patrón Plug-in permite extender el comportamiento del sistema de forma dinámica creando un objeto de instancia de una interfaz en tiempo de ejecución. Tiene como requisito que la clase que se instancia, implemente una determinada interfaz para poder tratar las distintas clases plug-in por igual.

Con este patrón el comportamiento modificado (el plug-in) se conecta a una clase abstracta parcial, que, a su vez, se conecta a la clase principal. El plug-in utiliza esta interfaz para aplicar métodos llamados por la clase principal.

Tiene como ventaja que permite conectar nuevas clases a la aplicación sin necesidad de modificar el código fuente original, facilitando una mayor modularidad en el programa y la posibilidad de que un tercero añada nuevas funcionalidades (Microsoft Coportation, 2005). En el sistema propuesto se utiliza

para que se puedan agregar nuevas métricas de diseño de software OO y traductores de XML compatibles con disímiles herramientas de modelado.

2.2.3 Patrones de diseño

Para el desarrollo de la herramienta propuesta se utilizaron patrones de diseño GoF. A continuación se hará un resumen de cuáles y cómo fueron utilizados.

2.2.3.1 Singleton (Instancia Única)

El objetivo de este patrón es garantizar que una clase sólo tenga una única instancia, proporcionando un punto de acceso global a la misma (Gamma, y otros, 1995). Este patrón se utiliza cuando debe haber únicamente una instancia de una clase y debe ser claro su acceso para los clientes. Aunque también es utilizado en ocasiones en que la "Instancia Única" debe ser especializada mediante herencia y los clientes deben poder usar la instancia extendida sin modificar su código. Como es de esperar esto aporta muchas ventajas como son:

- ✓ El acceso a la "Instancia Única" está más controlado.
- ✓ Se reduce el espacio de nombres (frente al uso de variables globales).
- ✓ Permite refinamientos en las operaciones y en la representación, mediante la especialización por herencia de "Solitario".
- ✓ Es fácilmente modificable para permitir más de una instancia y, en general, para controlar el número de las mismas (incluso si es variable).

Este patrón es utilizado en las clases controladoras del diseño, debido a que su función es controlar todos los eventos del sistema, y sería beneficioso tener una instancia única de dichas clases manteniendo así su estado durante todo el período de ejecución.

2.2.3.2 Abstract Factory (Fábrica Abstracta)

El propósito de este patrón es proporcionar una interfaz para crear familias de objetos relacionados o dependientes entre sí, sin especificar sus clases concretas (Gamma, y otros, 1995). Este patrón puede ser utilizado por diferentes motivos, entre ellos se pueden encontrar los siguientes:

- ✓ Cuando un sistema debe ser independiente de la creación, composición y representación de sus objetos.

- ✓ El sistema debe ser configurado con una familia de productos entre varias.
- ✓ Se quiere proporcionar una librería de clases de objetos de la que se desea revelar solo sus interfaces y no la implementación.
- ✓ Cuando una familia de objetos relacionados está diseñada para ser usada en conjunto.

Fábrica Abstracta potencia el encapsulamiento aislando a los clientes de las implementaciones, aumenta la flexibilidad del diseño permitiendo sustituir fácilmente una familia de objetos y refuerza la consistencia al obligar a la utilización de una familia de objetos a la vez.

Este patrón se emplea para encapsular la creación de los objetos del negocio, abstrayendo a la vista de la forma en que son instanciadas las entidades. Permite que el manejo de dichos objetos sea más flexible y desacopla la vista del modelo.

2.2.3.3 Composite (Compuesto)

Los objetos son estructurados en forma de árbol para representar jerarquías de parte-todo. Permite gestionar objetos complejos e individuales de forma uniforme (Gamma, y otros, 1995). Este patrón es utilizado cuando se quiere:

- ✓ Representar jerarquías de objetos.
- ✓ Obviar las diferencias entre las composiciones de objetos y los objetos individuales.

El patrón Composite permite que los objetos primitivos estén compuestos por objetos complejos, que a su vez pueden ser compuestos, y así de manera recurrente. Le permite facilidades al cliente ya que pueden tratar uniformemente a las estructuras compuestas y a los objetos individuales. También facilita añadir nuevos tipos de componentes pero trae como desventaja que sea más difícil restringirlos en determinado compuesto.

Se utiliza en el diseño de las clases del modelo, debido a que todas las estructuras que se encuentran en un diagrama de clases constituyen elementos de dicho diagrama con la particularidad que estos elementos pueden ser simples como el caso de los atributos y parámetros de una función; o compuestos como por ejemplo una clase que constituye un elemento en sí y a su vez está conformada por otro elementos simples o compuestos, como son los atributos y funciones respectivamente.

2.2.3.4 Strategy (Estrategia)

Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables (Gamma, y otros, 1995). Este patrón es usado para configurar una clase con un determinado comportamiento de entre muchos posibles. También se usa cuando se necesita que un algoritmo sea implementado de diferentes formas y/o para evitar exponer estructuras de datos complejas y dependientes de este.

Además cuando una clase define muchos comportamientos y estos se representan como múltiples sentencias condicionales en sus operaciones, sería factible hacer de cada una de estas ramas condicionales una nueva clase con un comportamiento en particular bien definido. Strategy tiene como ventaja que permite variar el algoritmo independientemente de su contexto, haciéndolo más fácil de cambiar, comprender y extender.

Es utilizado para la creación de las métricas de diseño y de los traductores de XML que serán incorporados al sistema en forma de plug-in. Garantiza que las estrategias (diferentes implementaciones de la función aplicar métrica o traducir modelo) puedan ser manejadas de una forma única.

2.3 Modelo de dominio

El CEGEL posee una estructura en la que los procesos docentes y productivos están interrelacionados, dándole un gran peso a la producción de software. En este contexto el uso de métricas para evaluar diseños de software es irregular y está limitado por la ausencia de políticas que rijan dicho proceso; además no se emplean herramientas que lo automaticen. Esto trae consigo que el uso de métricas sea en la actualidad una opción y no una regla para los equipos de desarrollo.

Estas condiciones no permitieron identificar con claridad los elementos que conforman el proceso por lo que se decidió crear un modelo de dominio cuyo propósito fundamental es definir una terminología común y sentar las bases del entendimiento del desarrollo del sistema.

El Modelo de Dominio es un artefacto que muestra las clases conceptuales significativas en un dominio, las relaciones entre estas clases y sus atributos. No contiene conceptos propios de un sistema de software sino de la realidad física (Larman, 2001).

A continuación se muestra el modelo del dominio del sistema:

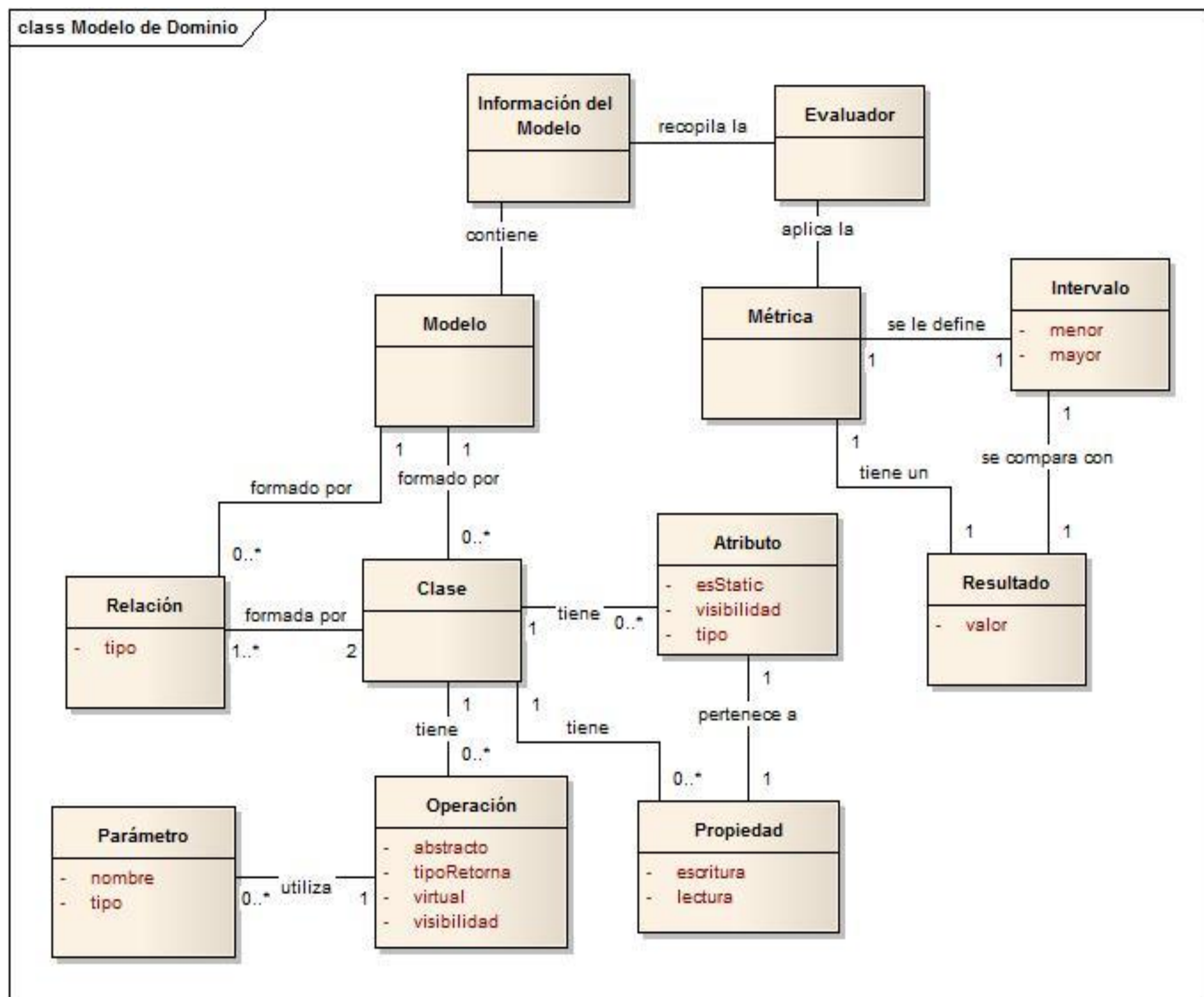

Figura 3. Modelo de Dominio

2.4 Especificación de Requisitos de software

El propósito general de la captura de requisitos es obtener una descripción correcta de lo que debe de hacer el sistema y delimitar su alcance. Los requisitos juegan un papel importante durante el ciclo de vida de un proyecto.

Para su obtención se aplicó la técnica de tormentas de ideas, con el objetivo de comprender las necesidades existentes; en ellas cada miembro del equipo de desarrollo expuso sus criterios.

Luego de ser capturadas fueron enumeradas y clasificadas en requisitos funcionales y no funcionales, todas las acciones que el sistema debería ser capaz de realizar.

2.4.1 Requisitos funcionales

Según la IEEE 610.32 los requerimientos funcionales son capacidades o condiciones que el sistema debe cumplir. Dependen de las necesidades de los clientes, constituyendo un paso fundamental para la satisfacción de dichos clientes y un mejor entendimiento para el equipo de desarrollo (IEEE, 1998).

Para el sistema propuesto se registraron los siguientes requisitos funcionales:

RF 1. Cargar el XML correspondiente al diagrama de clases del diseño generado por la herramienta CASE.

RF 2. Traducir el XML cargado al modelo de clases del sistema.

RF 3. Mostrar los elementos del modelo cargado.

RF 4. Mostrar las métricas candidatas a aplicar al modelo.

RF 5. Aplicar métrica Profundidad del árbol de herencia (DIT).

RF 6. Aplicar métrica Número de hijos (NOC).

RF 7. Aplicar métrica Acoplamiento entre objetos (CBO).

RF 8. Aplicar métrica Número de Métodos de Instancia Públicos (PIM).

RF 9. Aplicar métrica Número de Métodos de Instancia (NIM).

RF 10. Aplicar métrica Número de Variables de Instancia (NIV).

RF 11. Aplicar métrica Número de Métodos de Clase (NCM).

RF 12. Aplicar métrica Número de Variables de Clase (NVV).

RF 13. Aplicar métrica Número de Métodos Reemplazados (NMO).

RF 14. Aplicar métrica Número de Métodos Heredados (NMI).

RF 15. Aplicar métrica Número de Métodos Añadidos (NMA).

RF 16. Aplicar métrica Índice de Especialización (SIX).

RF 17. Aplicar métrica Promedio de Parámetros por Método (APPM).

RF 18. Configurar los intervalos de aceptación para cada métrica.

RF 19. Comparar los intervalos de aceptación con el resultado de las métricas.

RF 20. Mostrar los resultados obtenidos por la aplicación de las métricas.

RF 21. Crear proyecto.

RF 22. Abrir proyecto.

RF 23. Salvar proyecto.
RF 24. Sincronizar un modelo nuevo con un proyecto creado.
RF 25. Cargar plug-in.

2.4.2 Requisitos no funcionales

Los requerimientos no funcionales son propiedades o cualidades que el producto debe tener. Están vinculados generalmente a requisitos funcionales, es decir una vez se conozca lo que el sistema debe hacer se puede determinar con facilidad cómo ha de comportarse y qué cualidades debe tener (IEEE, 1998).

El levantamiento de requerimientos para el sistema propuesto arrojó los siguientes requisitos no funcionales:

2.4.2.1 Requerimientos de Seguridad

- ✓ El sistema debe comprobar que los usuarios que deseen introducir un plug-in posean los permisos necesarios.
- ✓ Se aplicarán las reglas de la “programación segura” mediante el tratamiento de excepciones.
- ✓ El sistema permitirá además la verificación sobre acciones irreversibles; es decir, se le solicitará al usuario la confirmación al realizar operaciones como la eliminación.

2.4.2.2 Requerimiento de Software

- ✓ Se utilizará como Framework .Net 4.0.
- ✓ Se utilizará como lenguaje de programación C#.
- ✓ Se utilizará para el modelado la herramienta Enterprise Architect 7.0.
- ✓ Se utilizará como IDE SharpDevelop 4.0

2.4.2.3 Requerimiento de Escalabilidad

- ✓ El sistema debe ser extensible.

2.4.2.4 Requerimientos de Apariencia o Interfaz Externa

- ✓ El sistema tendrá un menú de acciones para que el usuario pueda disponer de él en la medida de sus necesidades, visible todo el tiempo para propiciar el fácil acceso a las funcionalidades del software.
- ✓ El sistema permitirá claridad en los servicios que brinda, con términos asociados a los procesos de evaluación de proyectos de software propiciando que el usuario tenga un buen entendimiento de las utilidades que brinda.

2.5 Definición del modelo de casos de uso del sistema.

La arquitectura de un sistema está condicionada entre otras cosas por los casos de uso. Éstos son artefactos narrativos que describen el comportamiento del sistema desde el punto de vista de los usuarios. Los diagramas de casos de usos del sistema representan gráficamente a los procesos y su interacción con los actores.

Los actores representan un rol que realiza una persona, una organización o un sistema automatizado. Todo caso de uso es iniciado por un actor del sistema; en el sistema se detectaron los siguientes actores:

2.5.1 Actor del Sistema.

Actor: Usuario

Descripción: Representa los usuarios que utilizaran el sistema creado. Inicia todos los casos de uso del sistema excepto el CU Cargar Plug-in.

Actor: LoadSistema

Descripción: Se define este actor ficticio porque es el que inicializa el caso de uso Cargar Plug-in, cuando la aplicación es ejecutada.

2.5.2 Descripción reducida de los casos de uso del sistema

CASO DE USO:	Gestionar Proyecto

Resumen:	El caso de uso inicia cuando el Usuario decide crear, salvar o cargar un proyecto. El sistema solicita los datos necesarios para cada caso, el usuario los introduce y ejecuta la operación indicada.
REFERENCIAS	
Actores:	Usuario
Requisitos:	RF 21, RF 22, RF 23

Tabla 2. Descripción reducida del CU Gestionar Proyecto

CASO DE USO:	Gestionar XML
Resumen:	El caso de uso se inicia cuando el usuario selecciona la opción Cargar Modelo en el menú principal. EL sistema carga un XML de la dirección especificada, lo interpreta y lo muestra en forma de elementos (clases).
REFERENCIAS	
Actores:	Usuario
Requisitos:	RF 1, RF 2, RF 3, RF 23

Tabla 3. Descripción reducida del CU Gestionar XML

CASO DE USO:	Aplicar Métricas
Resumen:	El caso de uso inicia cuando el usuario elige la opción Aplicar Métricas, el sistema brinda la posibilidad de seleccionar las métricas que se pueden aplicar al diagrama de clases del diseño, el usuario selecciona las métricas deseadas, el sistema las aplica al modelo y muestra un resultado.
REFERENCIAS	
Actores:	Usuario
Requisitos:	RF5, RF6, RF7, RF8, RF9, RF10, RF11, RF12, RF13, RF14, RF15, RF16, RF17

Tabla 4. Descripción reducida del CU Aplicar Métricas

CASO DE USO:	Configurar Intervalos de Aceptación

Resumen:	El caso de uso inicia cuando el usuario elige la opción Configurar Intervalos de Aceptación, el sistema muestra las métricas disponibles permitiendo que se configuren los intervalos de aceptación para cada una de ellas, el usuario establece dichos intervalos y guarda la configuración.
REFERENCIAS	
Actores:	Usuario
Requisitos:	RF23

Tabla 5. Descripción reducida del CU Configurar Intervalos de Aceptación

CASO DE USO:	Mostrar resultados
Resumen:	El caso de uso inicia cuando el usuario elige la opción Aplicar Métricas, el sistema aplica las métricas al modelo, compara el resultado obtenido con los intervalos de aceptación y muestra un dictamen.
REFERENCIAS	
Actores:	Usuario
Requisitos:	RF19, RF20.

Tabla 6. Descripción reducida del CU Mostrar Resultados

CASO DE USO:	Sincronizar modelo con XML fuente
Resumen:	El caso de uso se inicia cuando el usuario selecciona la opción Sincronizar Modelo en el menú principal. EL sistema carga un XML de la dirección especificada, lo interpreta, lo compara con el modelo correspondiente que se encuentra en memoria y aplica los cambios.
REFERENCIAS	
Actores:	Usuario
Requisitos:	RF 1, RF 2, RF 3, RF 23, RF 24

Tabla 7. Descripción reducida del CU Sincronizar modelo con XML fuente

CASO DE USO:	Cargar Plug-in

Resumen:	El caso de uso se inicia cuando el sistema es ejecutado. El sistema carga los Plug-in incorporados. Los Plug-in deben implementar las interfaces definidas, una para los traductores de XML y la otra para las métricas.
REFERENCIAS	
Actores:	LoadSistema.
Requisitos:	RF 25.

Tabla 8. Descripción reducida del CU Cargar Plug-in

2.5.3 Diagrama de casos de uso del sistema

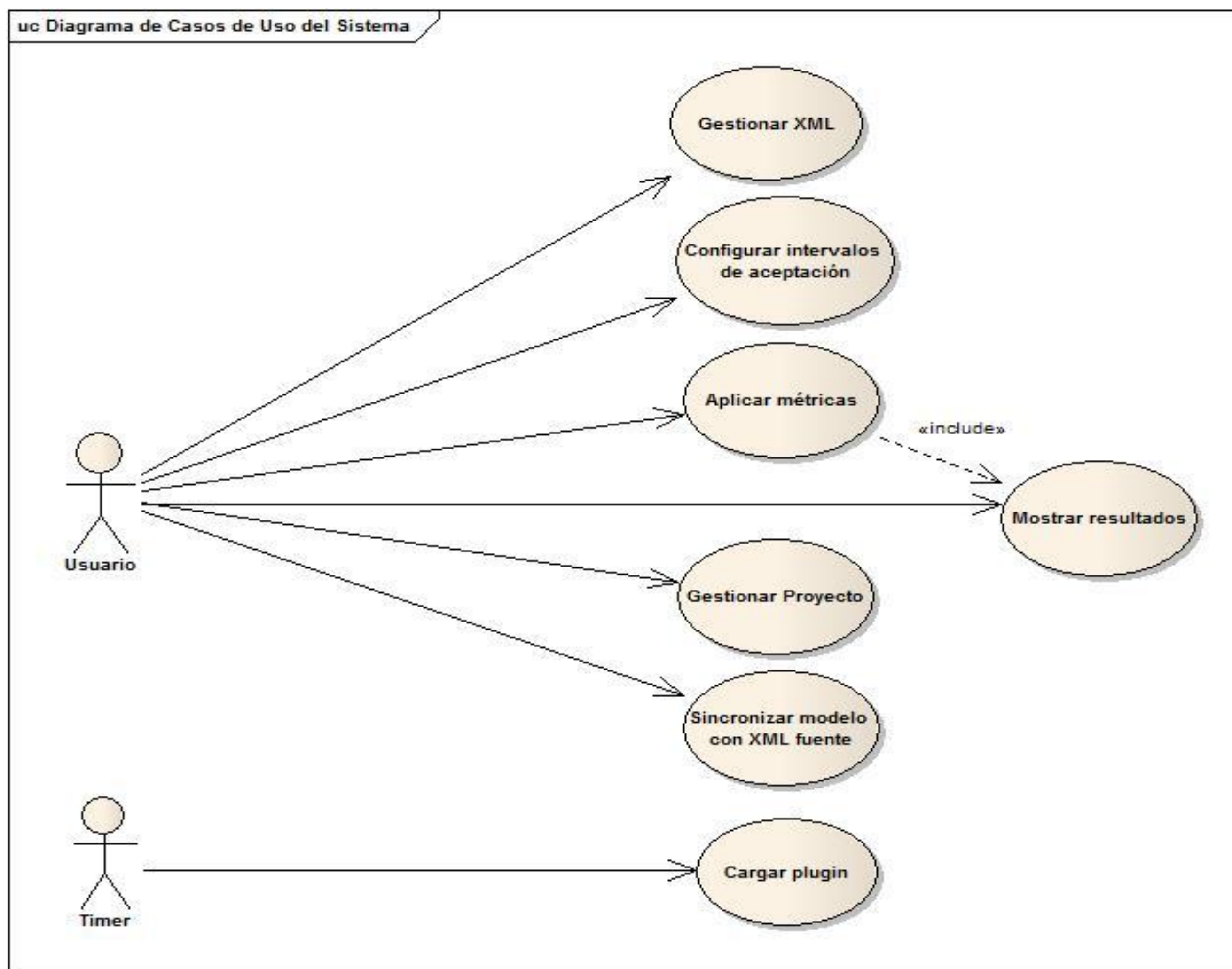

Figura 4. Diagrama de Casos de Uso del Sistema

2.6 Diseño del sistema

El diseño es una de las actividades técnicas necesarias para la elaboración del software. El diseño propuesto debe cumplir en totalidad con los requerimientos del sistema, debe ser capaz de facilitar las mejoras del software, debe especificarse, de forma tal que sea entendible por otros diseñadores y no diseñadores, servir como guía para los demás flujos de la ingeniería de software y permitir la comprobación del sistema fácilmente.

2.6.1 Modelo de diseño

En el modelo de diseño se identifican las clases que modelarán el problema, sus interfaces y jerarquía de herencia, además de establecer las relaciones claves entre las mismas. También se identifican los componentes que conforman el sistema, agrupándolos en subsistemas según su funcionalidad y propósito.

2.6.2 Diagrama de clases del diseño

Los diagramas de clases son diagramas de estructura estática que muestran un conjunto de clases, interfaces, colaboraciones y relaciones. Se utilizan para modelar la vista de diseño estática mostrando cómo puede ser construido el sistema. Los diagramas de clases del diseño describen la realización de los casos de uso y al mismo tiempo constituyen una abstracción del modelo de implementación y el código fuente, es una entrada esencial a las actividades de implementación. Con ellos se crea el diseño conceptual de la información que se manejará en el sistema, y los componentes que se encargaran del funcionamiento y la relación entre uno y otro.

En la siguiente imagen se muestra el diagrama de clases del Caso de Uso Gestionar XML.

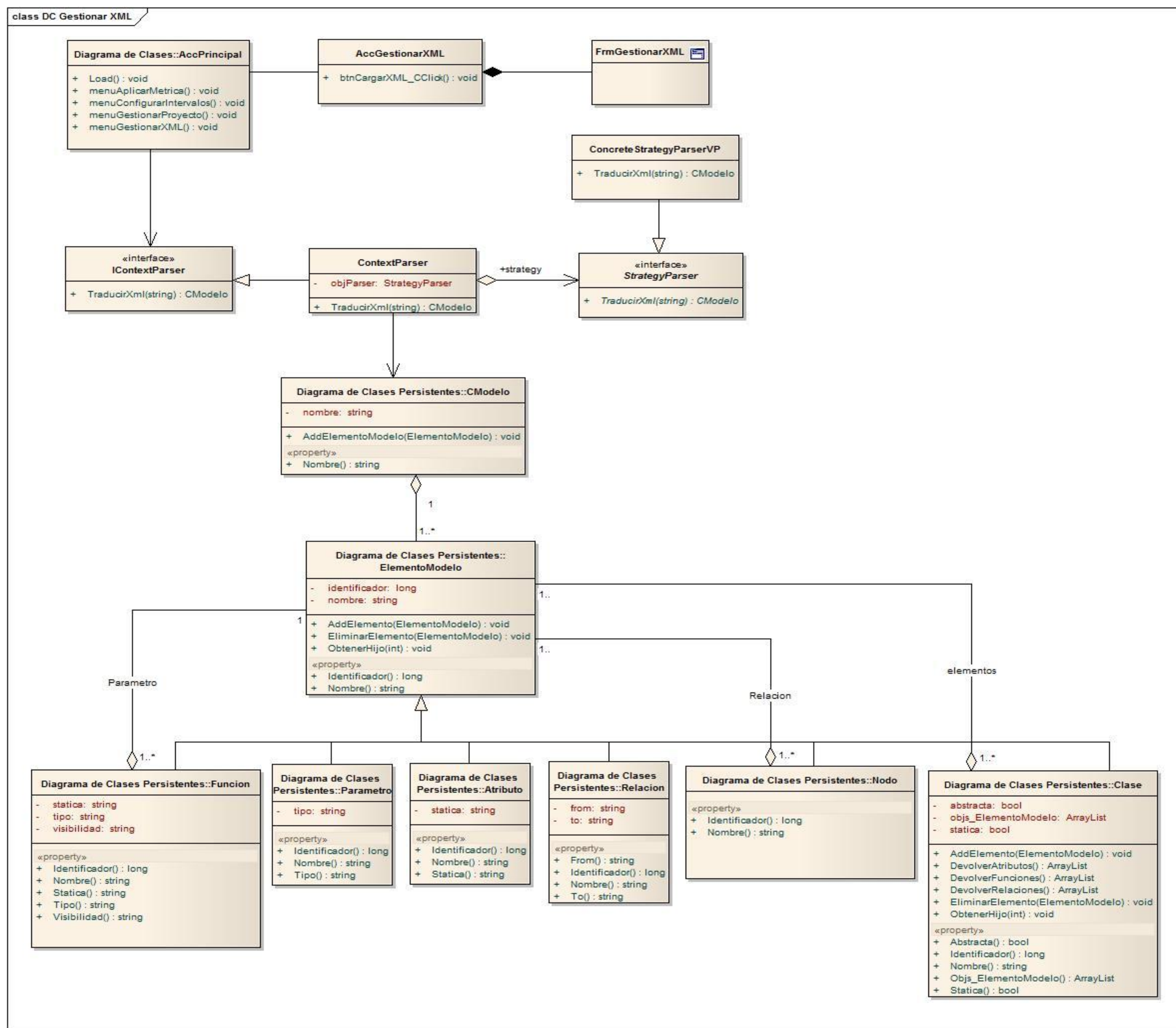

Figura 5. Diagrama de Clases del Diseño del CU Gestionar XML

2.6.3 Diagrama de secuencia

Los diagramas de secuencias muestran las interacciones entre los objetos o subsistemas mediante transferencia de mensajes, destacando la ordenación temporal de dichos mensajes. Es factible realizar

un diagrama de secuencia por cada escenario debido a que ayuda a la claridad en la realización del caso de uso.

A continuación se muestra a modo de ejemplo el diagrama de secuencia del Caso de Uso Gestionar XML.

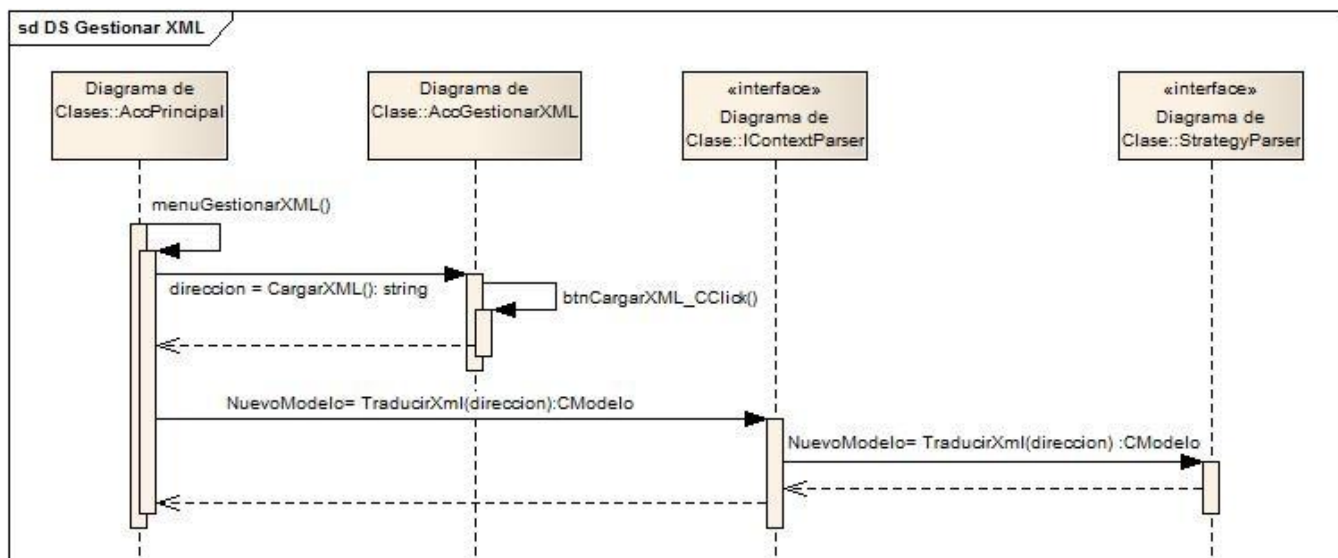

Figura 6. Diagrama de Secuencia del CU Gestionar XML

2.7 Implementación de la solución

La implementación es el principal flujo de trabajo en la fase de construcción. Se implementan los elementos del diseño en términos de implementación, obteniéndose los componentes necesarios para el funcionamiento de la aplicación. Está determinado por el lenguaje de programación y tiene como objetivo llevar a cabo la implementación de cada una de las clases significativas del diseño. En este flujo de trabajo se define la organización del código en términos de los subsistemas de implementación organizados en capas.

2.7.1 Estándares de implementación.

Los estándares de implementación son pautas de programación que están enfocadas a la estructura y apariencia física del código fuente, que facilita la lectura, comprensión y mantenimiento del mismo.

Dichos estándares definen la nomenclatura de las variables, objetos, método y funciones, teniendo en cuenta el orden y legibilidad del código escrito.

A continuación se presentan las pautas seguidas para la codificación del sistema propuesto:

- ✓ Los nombres de las clases deben de iniciar con letra mayúscula, utilizando la notación Camello, y si éstas tienen más de una palabra, cada palabra nueva debe iniciar con mayúscula y sin utilizar separadores entre ellas.
- ✓ Las clases correspondientes a los formularios comenzarán con la palabra "Form" y las controladoras con la palabra "Gestor", teniendo a continuación una palabra que permita una interpretación objetiva y cumpliendo con la norma establecida anteriormente. Ejemplo: "FormSeleccionarMetrica" y "GestorProyecto".
- ✓ Los nombres de las variables deben iniciar con letra minúscula, y si éstas tienen más de una palabra, cada palabra nueva debe iniciar con mayúscula y sin utilizar separadores entre ellas.
- ✓ En caso de que el tipo la variable constituya una clase de dominio deberá tener el prefijo "obj", teniendo a continuación un nombre descriptivo de la clase y cumpliendo con la norma establecida anteriormente. Ejemplo: "objProyecto".
- ✓ Los nombres de las funciones siguen la notación Camello descrita anteriormente y deberán brindar una interpretación explícita del contexto para el cual fue creado.
- ✓ Los nombres de los componentes que conforman las interfaces visuales tendrán un prefijo referente al tipo de componente seguido de una palabra que defina la acción a realizar o campo que representa. Ejemplo: "btnAdicionar".
- ✓ Se debe comentar todo lo que se haga en el código. Dichos comentarios deben ser claros y precisos, de forma tal que se entienda el propósito de lo que se está desarrollando.

2.7.2 Tratamiento de errores

En el sistema propuesto se tiene en cuenta el tratamiento de errores, para favorecer el correcto funcionamiento del mismo y la satisfacción del usuario. Se evitan, minimizan y tratan los posibles errores, con el fin de garantizar la integridad y confiabilidad de la información.

Para el tratamiento de errores se validan los datos que son introducidos por el usuario al sistema. Se valida que el formato de los datos sea el esperado y se identifican cada uno de los puntos en los que puede dar el error. Los mismos son capturados y gestionados en los métodos implementados, en el cual se definen los posibles errores y cómo deben ser mostrados de manera entendible para el usuario. Además este manejo de errores se realiza en dependencia del tipo de error lanzado.

2.7.3 Seguridad

La seguridad es un aspecto a tener presente durante el desarrollo de todo sistema informático. Más aún cuando se desea construir un software que podrá ser extendido con tan solo colocar un ensamblado en uno de sus directorios. La facilidad brindada por el sistema para ejecutar códigos desarrollados por terceros podría convertirse en una amenaza potencial a la integridad y confidencialidad de los datos del cliente. Por este motivo se toman una serie de medidas con vistas a minimizar los riesgos que esto representa.

- ✓ Firma de los ensamblados de la herramienta: Con la firma de los ensamblados se garantiza que cada uno de estos ensamblados tengan un nombre único (nombre seguro). De esta forma se evitaría la suplantación o modificación de los ensamblados por parte de un tercero. Certificándose así la autenticidad de los mismos.

- ✓ Verificación de la autenticidad de los plug-in: Se verifica antes de cargar un plug-in que esté firmado con la misma clave que se ha firmado el resto del sistema. Evitándose la carga de un plug-in desconocido que pudiera comprometer la integridad del sistema.

- ✓ Creación de un nuevo Dominio de Aplicación para la ejecución de los plug-ins: Siempre que un sistema admite código de terceros está expuesto a afectaciones originadas por este, ya sea por errores de programación o secuencias de código mal intencionadas. Para minimizar estos riesgos se crea un nuevo Dominio de Aplicación (AppDomain) en el cual serán ejecutados los plug-ins. Esto garantiza que la ejecución de los plug-ins no se realice en el mismo entorno en que se ejecuta la aplicación base. Esta ejecución independiente permite establecer nuevas políticas de seguridad que definirán un entorno controlado donde se ejecutarán los plug-ins y al mismo tiempo garantiza que los errores que puedan producirse en ellos, causados por una mala implementación, no afecten la aplicación base.

2.8 Conclusiones del capítulo

Con el desarrollo de este capítulo se generaron los siguientes artefactos relacionados con el flujo de Modelación de la metodología AUP:

- ✓ La arquitectura del sistema, determinada por los patrones arquitectónicos MVC y Plug-in.
- ✓ El Modelo de Dominio, permitiendo que se sentaran las bases del entendimiento del desarrollo del sistema.
- ✓ La especificación de los requerimientos funcionales y no funcionales, permitiendo determinar lo que debe de hacer el sistema y su alcance.
- ✓ El Modelo de Casos de Uso del Sistema para el cual se desarrolló la descripción de los actores y casos de uso del sistema.
- ✓ El Modelo de Diseño, desarrollando los diagramas de clases del diseño y los diagramas de secuencia.

Todo esto permitió obtener una mayor comprensión del sistema y la definición de los principios que guiaron la implementación y organización de la misma. Dentro del flujo de implementación se describió el tratamiento de errores y la seguridad que fue llevada a cabo para un mejor funcionamiento de dicha aplicación. Así como la obtención del sistema automatizado para medir la calidad del diseño OO basado en métricas, el cual permite aplicar 13 métricas de diseño OO.

Capítulo 3. Análisis de resultados.

Valorar si el diseño obtenido se ajusta al nivel de calidad requerido es importante para poder conocer la efectividad de los procesos que han sido modelados y si requieren gran esfuerzo para su implementación. De igual modo el proceso de pruebas es clave a la hora de detectar errores o fallas. Los objetivos principales de la realización de una prueba son tener un buen caso de prueba que sirva para descubrir un error no descubierto antes. Por tal motivo en el presente capítulo se analizan los resultados obtenidos de la aplicación de un conjunto de métricas al diseño y la realización de casos de pruebas al código del sistema.

3.1 Resultado de las métricas orientadas a objetos.

A continuación se aplican un conjunto de métricas[17] OO con el objetivo de determinar el grado de calidad y fiabilidad del diseño propuesto en el capítulo anterior. Es necesario plantear que los valores mostrados constituyen estimaciones conservadoras debido a que generalmente durante la fase de implementación se hace necesario incluir nuevas funcionalidades y atributos. Se analizarán los resultados obtenidos por algunas métricas para el caso de uso Gestionar XML por ser uno de los de mayor responsabilidad en el sistema.

3.1.1 Métricas propuestas por Chidamber y Kemerer. Aplicación al CU Gestionar XML.

3.1.1.1 Acoplamiento entre objetos (CBO).

Para aplicar esta métrica es necesario conocer la cantidad de clases con las que se relaciona una clase excluyendo las relaciones por herencia.

Como se muestra en la figura el mayor por ciento de las clases no están acopladas y el por ciento restante presentan una acoplamiento relativamente bajo, permitiendo que aumente el grado de reutilización de las clases existentes. Este resultado también ayuda a que las pruebas o modificaciones que sean necesarias, resulten fáciles de ejecutar.

[17] Las métricas usadas son las mismas que se implementan en el sistema propuesto cuyo funcionamiento fue descrito en la sección 1.3.

Nombre de la Clases	No. Colaboraciones
AccGestionarXML	0
AccPrincipal	2
Clase	1
CModelo	1
ContextParser	2
ElementoModelo	0
Funcion	1
Atributo	0
Parametro	0
Relacion	0
Nodo	1
ConcreteStrategyParserVP	0

Tabla 9. Colaboraciones por clases.

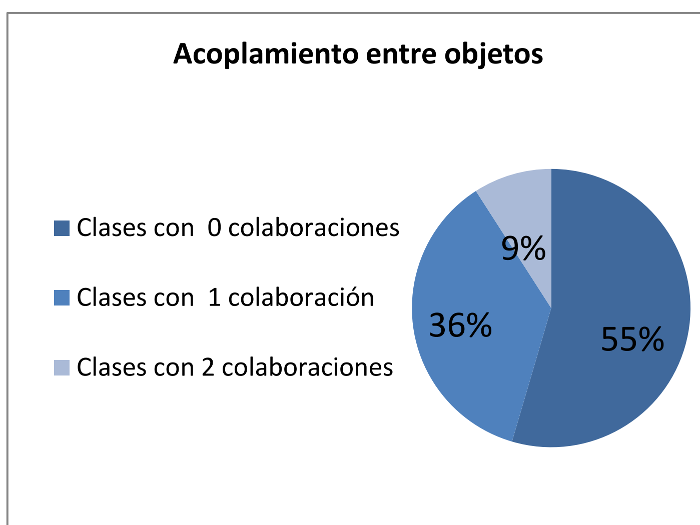

Figura 7.Acoplamiento entre objetos.

3.1.1.2 Profundidad del árbol de herencia (DIT).

Para aplicar esta métrica es necesario conocer la cantidad de relaciones de herencia que tiene una clase. Seguidamente se muestran los datos recopilados y su representación gráfica.

Clases	Profundidad árbol herencia
AccGestionarXML	0
AccPrincipal	0
Clase	1
CModelo	0
ContextParser	0
ElementoModelo	0
Funcion	1
Atributo	1
Parametro	1
Relacion	1
Nodo	1
ConcreteStrategyParserVP	0

Tabla 10. Herencias por clases.

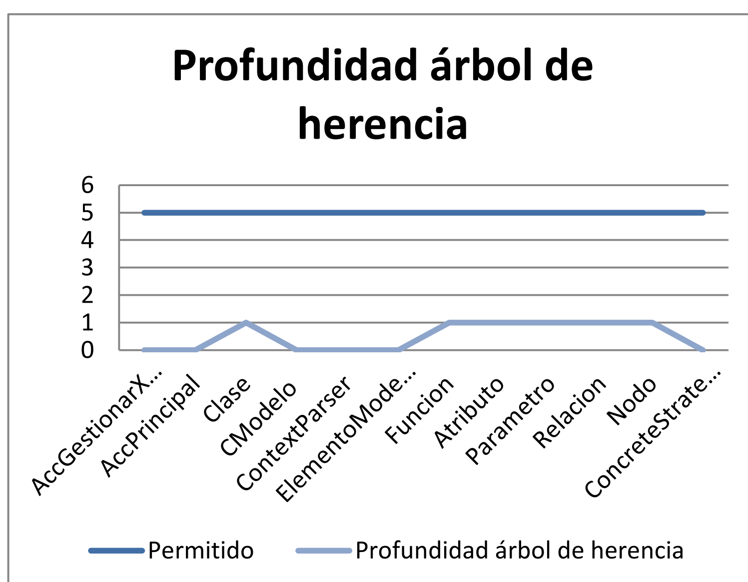

Figura 8. Profundidad árbol de herencia.

La imagen muestra en color claro el nivel que tiene cada una de las clases en el árbol de herencia. Las que tienen valor cero indican que no tienen relaciones de herencia o que están en el nivel cero del

árbol. Las restantes poseen valor 1 que es inferior al 5 sugerido por la bibliografía consultada. Estos resultados indican que las clases serán fáciles de desarrollar y de mantener.

3.1.2 Métricas propuestas por Lorenz y Kidd. Aplicación al CU Gestionar XML.

3.1.2.1 Número Métodos Instancia Públicos (PIM).

Para calcular esta métrica se necesita la información de la cantidad de métodos de instancia públicos que tiene cada clase. A continuación se muestran y representan gráficamente los resultados.

Número Clases	No. Met. Instancia Públicos
4	1
0	2
6	3
0	4
1	5
1	6

Tabla 11. PIM por clases.

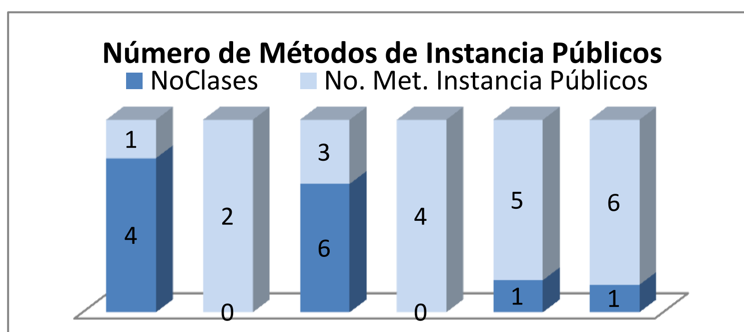

Figura 9. Número de Método de Instancia Públicos.

Los resultados obtenidos son satisfactorios. Diez de las doce clases modeladas tienen tres o menos métodos de instancia públicos lo que es sin dudas un resultado bajo. Las otras dos clases tienen valores que aunque son más elevados que los demás, también resultan bajos. Estas dos últimas clases fueron revisadas y se comprobó que sus métodos son todos necesarios para el correcto funcionamiento del sistema.

3.1.2.2 Número de Variables Instancia (NIV).

Se necesita conocer el número de variables de instancia de una clase para calcular esta métrica. A continuación se exponen los resultados obtenidos y se representan gráficamente.

Número Clases	No. Var. Instancia
3	0
2	1
2	2
2	3
1	4
2	5

Tabla 13. NIV por clases.

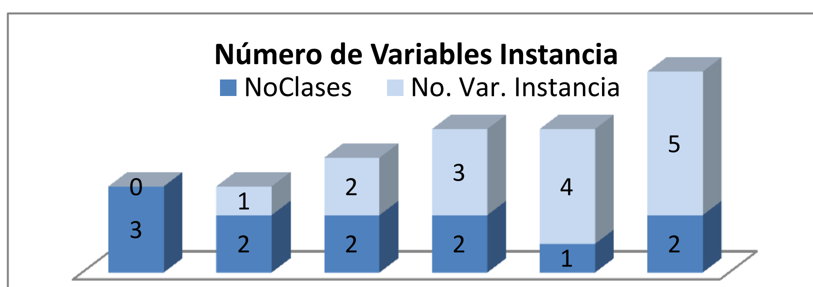

Figura 10. Número de Variables Instancia.

El resultado evidencia que los valores de esta métrica para cada una de las clases son permisibles. Solo dos de las doce clases modeladas tienen cinco variables de instancia que aunque es el mayor, es bajo en comparación con la cantidad que suelen tener las clases. Estas dos clases tienen 5 variables instancia que resultan imprescindibles para la creación de los elementos del modelo, dos de las cuales son heredadas de su superclase.

3.2 Resultados de las Pruebas de Caja Blanca y Caja Negra.

Las pruebas de software son un elemento crítico para la garantía de calidad del software y representa una revisión final de las especificaciones, del diseño y de la codificación (Pressman, 2002). Para realizar estas pruebas se diseñan casos de pruebas con el objetivo fundamental de determinar si las características propias del software son completamente satisfactorias. Dichos casos de prueba proporcionan una descripción de puntos importantes de observación, con resultados positivos o negativos para lograr que la mayoría de los requisitos de la aplicación sean debidamente probados. Entre los principales métodos de prueba se encuentran las pruebas de Caja Blanca y de Caja Negra, utilizadas para validar las funcionalidades del sistema creado.

3.2.1 Pruebas de Caja Blanca.

En las pruebas de caja blanca se comprueban los caminos lógicos del software proponiendo casos de prueba en los que se ejerciten conjuntos específicos de condiciones y/o bucles. Se puede examinar el estado del programa en varios puntos para determinar si el estado real coincide o no con el esperado.

Existen varios tipos de pruebas de caja blanca, ellos son:

- ✓ Prueba de Condición: Es un método de diseño de casos de prueba que ejercita las condiciones lógicas contenidas en el módulo de un programa.
- ✓ Prueba de Flujo de Datos: Se seleccionan caminos de prueba de un programa de acuerdo con la ubicación de las definiciones y los usos de las variables del programa.
- ✓ Prueba de Bucles: Es una técnica de prueba de caja blanca que se centra exclusivamente en la validez de las construcciones de bucles.
- ✓ Prueba del Camino Básico: Esta técnica permite obtener una medida de la complejidad lógica de un diseño y usar esta medida como guía para la definición de un conjunto básico.

Para validar el sistema se selecciona la prueba del camino básico debido a que permite obtener una medida de la complejidad lógica del código de cada método, programa o módulo dado. Es además una de las más eficientes en cuanto a cobertura de código, pues logra que se ejecuten todos los bucles en sus límites operacionales.

Para aplicar esta técnica debe seguirse el siguiente algoritmo de trabajo:

- ✓ Para aplicar la técnica del camino básico se debe introducir la notación para la representación del flujo de control, representado por un grafo de flujo en el cual:
  - ✓ Cada nodo del grafo corresponde a una o más sentencias de código fuente.
  - ✓ Todo segmento de código de cualquier programa se puede traducir a un grafo de flujo.
- ✓ A partir del diseño o del código fuente, se dibuja el grafo de flujo asociado.
- ✓ Se calcula la complejidad ciclomática del grafo.
- ✓ Se determina un conjunto básico de caminos independientes.
- ✓ Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.
- ✓ Los casos de prueba obtenidos del conjunto básico garantizan que durante la prueba se ejecute por lo menos una vez cada sentencia del programa.

Para dibujar el grafo se emplean tres componentes principalmente, a saber:

- ✓ Nodo: son los círculos representados en el grafo de flujo, el cual representa una o más secuencias del procedimiento, donde un nodo corresponde a una secuencia de procesos o a una sentencia de decisión. Los nodos que no están asociados se utilizan al inicio y final del grafo.
- ✓ Aristas: son constituidas por las flechas del grafo, iguales a las representadas en un diagrama de flujo y constituyen el flujo de control del procedimiento. Las aristas terminan en un nodo, aun cuando el nodo no representa la sentencia de un procedimiento.

- ✓ Regiones: son las áreas delimitadas por las aristas y nodos donde se incluye el área exterior del grafo, como una región más. Las regiones se enumeran siendo la cantidad de regiones equivalente a la cantidad de caminos independientes del conjunto básico de un procedimiento.

3.2.1.1 Aplicación de la técnica del camino básico al método AplicarMetricasAPPM.

En la siguiente figura se muestra el código del método AplicarMetricasAPPM y en lado derecho se visualiza la enumeración de las sentencias de dicho código.

```
public ArrayList AplicarMetricas(CModelo objModelo)
{
    int cantMetodos;                                                              1
    int cantParametros;                                                           1
    DetalleResultado objDetalleResult;                                            1
    ArrayList resultados = new ArrayList();                                       1

    for (int i = 0; i < objModelo.Elem_ElementoModelo.Count; i++)                 2
    {
        objDetalleResult = new DetalleResultado();                                3
        cantMetodos = 0;                                                          3
        cantParametros = 0;                                                       3

        for (int j = 0; j < (objModelo.Elem_ElementoModelo[i] as Clase).List_ElementoModelo.Count; j++)   4
          {
            if((objModelo.Elem_ElementoModelo[i] as Clase).List_ElementoModelo[j] is Funcion)   5
            {
                cantMetodos ++;                                                   6
                cantParametros += ((objModelo.Elem_ElementoModelo[i] as Clase).List_ElementoModelo[j] as Funcion).Param_ElementoModelo.Count;6
            }                                                                     7

          }
        objDetalleResult.CClase = objModelo.Elem_ElementoModelo[i] as Clase;      8
        objDetalleResult.Valor = cantParametros/cantMetodos;                      8
        resultados.Add(objDetalleResult);                                         8

    }
    return resultados;                                                            9
}
```

Figura 11. Código de prueba

En la figura # 15 aparece el grafo de flujo confeccionado a partir del código anterior, cada nodo corresponde a la numeración de cada una de las sentencias, este grafo muestra todas las posibles rutas a seguir durante la ejecución del método AplicarMetricasAPPM.

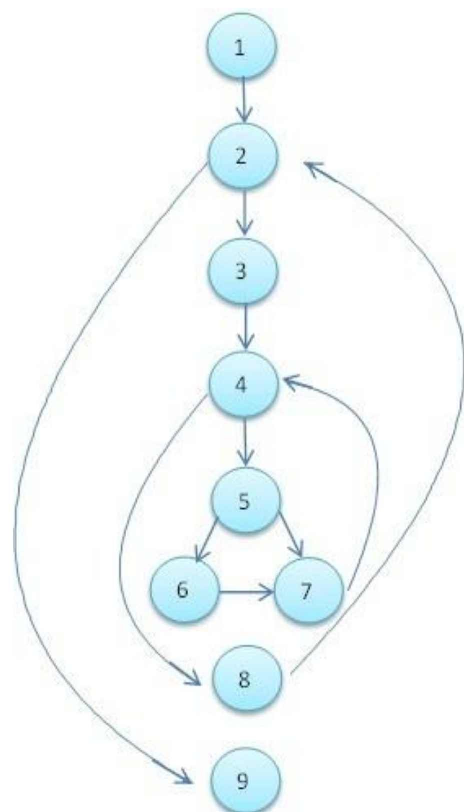

Figura 12. Grafo de flujo del método

Complejidad ciclomática del grafo.

Luego de haber construido el grafo se realiza el cálculo de la complejidad ciclomática mediante tres fórmulas, las cuales tienen que mostrar el mismo resultado para asegurar que el cálculo de la complejidad es correcto.

Fórmulas para calcular complejidad ciclomática:

1. **V (G) = (A – N) + 2** Donde “A” es el número de aristas y “N” el de nodos. Para los valores A = 11 y N = 9, **V (G) = 4**

2. **V (G) = P + 1** Donde “P” es la cantidad total de nodos predicados (son los nodos de los cuales parten dos o más aristas). Para P = 3, **V (G) = 4**

3. **V (G) = R** Donde “R” es la cantidad total de regiones. **V (G) = 4**

El cálculo efectuado mediante las tres fórmulas ha dado el mismo valor, por lo que puede afirmarse que la complejidad ciclomática del código es 4, lo que significa que existen cuatro vías independientes por donde el flujo puede circular. Ese valor representa el límite mínimo del número total de casos de pruebas para el procedimiento tratado.

Caminos básicos:

Camino 1: 1-2-3-4-5-6-7-4-8-2-9

Camino 2: 1-2-3-4-5-7-4-8-2-9

Después de haber obtenido los caminos básicos del flujo, se elabora los casos de pruebas para el procedimiento, se debe realizar al menos un caso de prueba por cada camino básico. Para realizarlos es necesario cumplir con las siguientes exigencias:

- ✓ Descripción: Se hace la entrada de datos necesaria, validando que los parámetros obligatorios pasen nulos al procedimiento o no se entre algún dato erróneo.
- ✓ Condición de ejecución: Se especifica cada parámetro para que cumpla una condición deseada para ver el funcionamiento del procedimiento.
- ✓ Entrada: Se muestran los parámetros que entran al procedimiento.
- ✓ Resultados esperados: Se expone el resultado que el procedimiento espera.

La realización de los casos de prueba se ejemplificará con la descripción de un caso de prueba para el camino básico #1.

Caso de prueba para el camino básico #1:

Descripción:

Los datos de entrada cumplirán con los siguientes requisitos:

El parámetro de entrada será un objeto de la clase modelo. La lista de elementos de dicho modelo no debe de estar vacía y todos los elementos deben de ser clases. Estas clases deben de tener una lista de elementos, formada entre otras cosas por funciones y estas a su vez por una lista de parámetros.

Condición de ejecución:

La lista de elementos del modelo tendrá longitud 2, es decir estará formado por dos clases. Cada clase estará constituida por 4 funciones y cada función por 2 parámetros.

Entrada:

listaElem_ElementoModelo del modelo = clase1, clase2

listaElementos de clases (una lista para cada clase) = funcion1, funcion2, funcion3, funcion4.

listaElementos de funciones (una lista para cada función de cada clase) = parametro1, parametro2.

Resultados esperados:

Se espera que devuelva una lista de resultados con longitud 2. En la primera posición estaría la clase1 con valor igual 2 y en la segunda la clase2 con valor igual 2.

Luego de aplicar distintos casos de pruebas a todos los caminos básicos, se pudo comprobar que el flujo de trabajo del método está correcto, pues cumple con las condiciones necesarias planteadas. De la misma manera se realizaron casos de prueba a las principales funciones del sistema y los resultados obtenidos fueron satisfactorios, validando el código del sistema y garantizando la calidad del mismo.

3.2.2 Pruebas de Caja Negra

Las pruebas de caja negra representan las pruebas que se realizan a la interfaz del sistema. Los casos de pruebas validan las funciones del software, que los datos de entrada se acepten de forma adecuada y que el resultado obtenido sea el correcto. Para desarrollar la prueba de caja negra existen varias técnicas, entre ellas están:

- ✓ Métodos de pruebas basados en grafos: Este método está dirigido a las relaciones que existen entre los objetos del programa y su comportamiento.
- ✓ Partición equivalente: Esta técnica divide el campo de entrada en clases de datos que tienden a ejercitar determinadas funciones del software.
- ✓ Análisis de valores límite: Prueba la habilidad del programa para manejar datos que se encuentran en los límites aceptables.
- ✓ Prueba de la tabla ortogonal: Suministra un método sistemático y eficiente para probar sistemas con un número reducido de parámetros de entrada.

Se utilizará la técnica de particiones equivalentes debido a que es una de las más efectivas pues permite examinar los valores válidos e inválidos de las entradas en la aplicación y descubre de forma inmediata errores existentes. A continuación se muestra un caso de prueba de uno de los caso de uso más significativos del sistema “Configurar Intervalos”.

3.2.2.1 Aplicación de la técnica de particiones equivalentes CU “Configurar Intervalos”

Descripción General.

El caso de uso inicia cuando el usuario elige la opción Configurar Intervalos de Aceptación, el sistema muestra las métricas disponibles permitiendo que se configuren los intervalos de aceptación para cada una de ellas, el usuario establece dichos intervalos y guarda la configuración.

Condiciones de Ejecución.

Debe haberse creado o abierto un proyecto, donde se guardará la configuración de los intervalos.

Secciones a probar en el Caso de Uso.

Nombre de la sección	Escenarios de la sección	Descripción de la funcionalidad
1. Crear intervalos de aceptación.	1.1. Guardar configuración de intervalos.	Se guarda correctamente los intervalos de aceptación por cada métrica.
	1.2. Existen datos incompletos.	Luego de haber introducido los datos, el sistema los verifica y valida, de haber alguna celda de texto vacía muestra un mensaje indicándolo.
	1.3. Existen datos incorrectos.	Cuando el usuario intenta escribir datos incorrectos el sistema no lo permite. Para cada campo el sistema permite escribir solo valores correctos.
	1.4. No está creado el proyecto.	Luego de haber introducido los datos, el sistema los verifica y valida, en caso de que no exista un proyecto creado, lanza un mensaje de error.
2. Modificar intervalos de aceptación.	2.1. Modificar configuración de	Modificar los intervalos de aceptación de manera satisfactoria.
	2.2. Existen datos incompletos.	Luego de haber introducido los datos, el sistema los verifica y valida, de haber alguna celda de texto vacía muestra un mensaje indicándolo.
	EC 2.3. Existen datos incorrectos.	Cuando el usuario intenta escribir datos incorrectos el sistema no se lo permite. Para cada campo el sistema permite escribir solo valores correctos.

Tabla 12. Casos de prueba para el CU "Configurar Intervalos".

Descripción de variables:

No	Nombre de campo	Clasificación	Valor Nulo	Descripción
1	Menor valor	Campo de texto	No	Solo números positivos.

2	Mayor Valor	Campo de texto	No	Solo números positivos.

Tabla 13. Descripción de variables.

Matriz de Datos:

Sección 1 Crear intervalos de aceptación.

Id del escenario	EC 1.1	EC 1.2	EC 1.3	EC 1.4
Escenario	Guardar configuración de intervalos	Existen celdas vacías.	Existen datos incorrectos	No está creado el proyecto
Menor valor	1	1	1	1
Mayor valor	5	(vacío)	Valor 1	5
Respuesta del Sistema	El sistema guarda la configuración de los intervalos y se cierra la ventana.	El sistema muestra un mensaje informando que debe llenar el campo correspondiente.	El sistema no permite escribir caracteres que no sean números en este campo.	El sistema muestra un mensaje de error indicando que debe crear o cargar un proyecto.
Resultado de la Prueba	Satisfactoria.	Satisfactoria.	Satisfactoria	Satisfactoria

FlujoCentral	En el menú principal en Guardar proyecto o Configuración, se muestra una ventana con todas las métricas y la opción de introducir para cada una de ellas un valor menor y uno mayor del rango de intervalo de aceptación. Se da clic en el botón guardar.	En el menú principal en Guardar proyecto o en Configuración, se muestra una ventana con todas las métricas y la opción de introducir para cada una de ellas un valor menor y uno mayor del rango de intervalo de aceptación. Se da clic en el botón guardar dejando datos incompletos.	En el menú principal en Guardar proyecto o en Configuración, se muestra una ventana con todas las métricas. El usuario intenta escribir caracteres que no sean números en los campos de los valores.	Se activa la opción de crear o cargar un proyecto.

Tabla 14. Matriz de Datos Sección 1.

Sección 2 Modificar intervalos de aceptación.

Id del escenario	EC 1.1	EC 1.2	EC 1.3
Escenario	Modificar configuración	Existen celdas vacías.	Existen datos incorrectos
Menor valor	1	1	1
Mayor valor	5	(vacío)	Valor 1
Respuesta del Sistema	El sistema guarda la configuración de los intervalos y se cierra la ventana.	El sistema muestra un mensaje informando que debe llenar el campo correspondiente.	El sistema no permite escribir caracteres que no sean números en este campo.
Resultado de la Prueba	Satisfactoria.	Satisfactoria.	Satisfactoria

Flujo Central	En el menú principal en Configuración, se muestra una ventana con todas las métricas y la opción de modificar para cada una de ellas el valor menor y el mayor del rango de intervalo de aceptación. Se da clic en el botón guardar.	En el menú principal en Configuración, se muestra una ventana con todas las métricas y la opción de modificar para cada una de ellas el valor menor y el mayor del rango de intervalo de aceptación. Se da clic en el botón guardar dejando datos incompletos.	En el menú principal en Configuración, se muestra una ventana con todas las métricas. El usuario intenta escribir caracteres que no sean números en los campos de los valores.

Tabla 15. Matriz de Datos Sección 2.

Resultados de las pruebas de caja negra.

Las pruebas de caja negra fueron realizadas a los casos de uso más significativos del sistema en varias iteraciones. Dichas pruebas arrojaron resultados exitosos y los errores detectados fueron resueltos en su totalidad. Por tanto el sistema está apto para su utilización y cumple con todas las funcionalidades para las que fue concebido.

3.3 Conclusiones del capítulo

Durante el desarrollo de este capítulo se aplicaron las métricas para evaluar el diseño y se realizaron las pruebas de caja blanca y caja negra al sistema, analizándose los resultados obtenidos. Se puede concluir que el diseño no tiene alta complejidad estructural y que posee una calidad aceptable, evidenciándose en los resultados de las métricas aplicadas. Además los resultados satisfactorios arrojados por las pruebas demostraron que el sistema cuenta con las características y funcionalidades para las que fue creado. Esta serie de validaciones dan como resultado la obtención de un producto final con calidad.

Conclusiones

En el presente trabajo se propuso un Sistema Automatizado para medir la calidad del diseño OO basado en métricas, enfocado en la disciplina de Diseño con el objetivo de contribuir a reducir los tiempos de desarrollo y mejorar la calidad final del producto. Se cumplieron los objetivos propuestos y se arribó a las siguientes conclusiones:

- ✓ Se realizó un estudio del estado del arte de la automatización de las métricas de diseño OO, analizándose diferentes herramientas existentes en el mundo, logrando identificar las principales tendencias y concluyendo que no existe ninguna que solucione el problema identificado en esta investigación.
- ✓ Se analizaron distintas metodologías de desarrollo, estilos y patrones arquitectónicos, herramientas CASE, frameworks y entornos de desarrollo, así como los patrones y las métricas de diseño OO. Esto permitió concebir la fundamentación teórica en la que se sustenta el desarrollo del trabajo.
- ✓ Se caracterizaron los aspectos que conforman el proceso de aplicación de métricas de diseño OO, permitiendo su comprensión con vistas al desarrollo de la aplicación.
- ✓ Se construyeron el Modelo de Casos de Uso del Sistema y el Modelo de Diseño, para los cuales se desarrolló la descripción de los actores y casos de uso del sistema así como los diagramas de clases del diseño y los diagramas de secuencia.
- ✓ Se obtuvo un sistema automatizado para medir la calidad del diseño OO basado en métricas, el cual permite aplicar 13 métricas de diseño OO a la información extraída del XML de un diagrama de clases generado por una herramienta CASE.
- ✓ Se realizó un análisis y validación de la calidad tanto del diseño como de la implementación del sistema a través de la aplicación de diferentes métricas de calidad al diseño y la realización de varios tipos de pruebas sobre el código que ayudaron a corroborar el estado de los artefactos obtenidos durante el desarrollo.
- ✓ El sistema creado es totalmente extensible, permitiendo la incorporación de plug-ins tanto para aplicar nuevas métricas como para interpretar XML de otras herramientas de modelado.

Recomendaciones

Una vez concluido el trabajo se formulan las siguientes recomendaciones:

- ✓ Aplicar el Sistema Automatizado para medir la calidad del diseño OO basado en métricas a los proyectos del CEGEL.
- ✓ Promover la creación de plug-ins que permitan la compatibilidad del sistema con nuevas herramientas CASE e incrementar el número de métricas soportadas.
- ✓ Continuar el desarrollo del sistema de forma tal que permita una gestión más amplia y precisa de los proyectos que en él son analizados.

Bibliografía

Álvarez, J.L.M. 2004. *Aplicación de un Sistema Experto para el desarrollo de Sistema Evaluador del modelo Capability Maturity Model (CMM) niveles dos y tres.* México : Universidad de las Américas, 2004.

Amaro Calderón, Sarah Dámaris y Valverde Rebaza, Jorge Carlos. 2007. *Metodologías Ágiles.* Trujillo, Perú : Universidad Nacional de Trujillo. Facultad de Ciencias Físicas y Matemáticas. Escuela de Informática., 2007.

Ambler, Scott W. 2005. *The Agile Unified Process (AUP).* [En línea] Septiembre de 2005. [Citado el: 15 de Enero de 2011.] http://www.ambysoft.com/unifiedprocess/agileUP.html.

Andersen, Arthur. 1995. *La Calidad en España.* Madrid : s.n., 1995.

Archer, Tom. 2001. *C# a fondo.* Madrid : McGraw-Hill, 2001.

Boehm, B.W. y Kaspar, S.R., et al. 1978. *Characteristics of Software Quality.* s.l. : TRW Series of Software Technology, 1978.

Burbeck, Steve. 1992. *Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC).* 1992.

Buschman, F, y otros. 2000. *Pattern Oriented Software Architecture: Patterns for Concurrent and Networked.* Estados Unidos : John Wiley & Sons, 2000.

Chidamber, Shyam R. y Kemerer, Chris F. 1994. *A Metrics Suite for Object Oriented Design.* s.l. : IEEE Transactions on Software Engineering., 1994. vol. 20, no. 6.

Cyvix. 2006. Cyvix. [En línea] SourceForge.net, 2006. [Citado el: 17 de Noviembre de 2010.] http://cyvis.sourceforge.net/.

Española, Real Academia. 2001. *Diccionario de la lengua española 22" Edición.* Madrid : s.n., 2001.

Fenton, Norman E. 1991. *Software Metrics: A Rigorous Approach.* London, UK, UK : Chapman & Hall, Ltd., 1991. ISBN:0442313551.

Gamma, R., y otros. 1995. *Design Patterns. Elements of Reusable Object-Oriented Software.* s.l. : Addison Wesley Professional Computing Series, 1995.

Grimán, A., y otros. 2003. *Quality oriented Architectural Approaches for Enterprise Systems.* 2003.

IEEE. 2000. *1471-2000 - IEEE Recommended Practice for Architectural Description for Software-Intensive Systems.* 2000.

—. 1998. *Guide to Software Requirements Specifications. ANSI/IEEE Standard 830-1998.* 1998.

—. 1993. *IEEE Software EngineeringStandards. Standard 610.12-1990, 1993.* 1993.

Información general y conceptual sobre .NET Framework. *MSDN.* [En línea] Microsoft. [Citado el: 9 de Febrero de 2011.] http://msdn.microsoft.com/es-es/library/zw4w595w.aspx.

ISO/IEC. 1991. *Evaluación de producción del software – Características de calidad y guías para su uso.* 1991. 9126.

—. 1996. *ISO/IEC Guide 2. Standardization and related activities. General vocabulary .* 1996.

Jacobson, Ivar, Booch, Grady y Rumbaugh, James. 2000. *El Proceso Unificado de Desarrollo de Software.* Madrid : Addison Wesley, 2000.

JDepend. 2005. JDepend. [En línea] Clarkware Consulting, Inc, 2005. [Citado el: 18 de Noviembre de 2010.] http://clarkware.com/software/JDepend.html.

Larman, Craig. 2001. *UML y Patrones. Una introduccón al análisis y diseño orientado a objeto y al proceso unificado.* s.l. : Prentice Hall, 2001. 2da Edición.

Lorenz, M. y Kidd, J. 1994. *Object Oriented Metrics.* Englewood, New Jersey : Prentice Hall, 1994.

McCall, J.A. y Richards, P.K. y Walters, G.F. 1977. *Factors in software quality.* Estados Unidos : s.n., 1977.

Metric1.3.6. 2005. Metric 1.3.6. [En línea] SourceForge.net, 2005. [Citado el: 17 de Noviembre de 2010.] http://metrics.sourceforge.net/.

Microsoft Coportation. 2005. MSDN. [En línea] 2005. http://msdn.microsoft.com/es-es.

NDepend. 2004. NDepend. [En línea] Syncfusion, Inc., 2004. [Citado el: 18 de Noviembre de 2010.] http://www.ndepend.com/.

Pressman, Roger S. 2002. *Ingeniería del Software. Un enfoque práctico.* s.l. : McGraw Hill, 2002.

Refactorit. 2008. Refactorit. [En línea] SourceForge.net, 2008. [Citado el: 17 de Noviembre de 2010.] http://refactorit.sourceforge.net/.

Reynoso, Carlos Billy. 2004. *Introducción a la Arquitectura de Software.* s.l. : UNIVERSIDAD DE BUENOS AIRES, 2004.

Ruiz, A., y otros. 2001. *Tratamiento Automático de Requisitos de Calidad en Sistemas Multiorganizacionales Basados en la Web.* Sevilla : Actas de las Jornadas de Ingeniería de Requisitos Aplicada, 2001.

Rumbaugh, James, Jacobson, Ivar y Booch, Grady. 1999. *El Lenguaje Unificado de Modelado.* s.l. : Addison Wesley, 1999.

Scribd. 2010. Scribd. *Scribd.* [En línea] 2010.

SharpDevelop. [En línea] http://www.sharpdevelop.net/opensource/sd/.

Shaw, Mary y Garlan, David. 1996. *Software Architecture: Perspectives on an Emerging Discipline.* s.l. : Prentice Hall, 1996.

Stephen, Albin. 2003. *The Art of Software Architecture: Design methods and techniques.* Nueva York : s.n., 2003.

UNE-EN. 2000. *ISO 9000 Sistemas de gestión de la calidad. Fundamentos y vocabulario.* Madrid : s.n., 2000.

Westfall, Linda L. 1995. *Software Metrics That Meet Your Information Needs.* s.l. : Annual Quality Congress, Cincinnati OH, Vol. 49, No. 0, May 1995, pp. 889-897, 1995.

Wolf, Alexander y Dewayne, Perry. 1992. *Foundations for the study of software architecture.* 1992.

Wüst, Jürgen. 2002. SDMetrics. *SDMetrics.* [En línea] Diciembre de 2002. [Citado el: 10 de Enero de 2011.] http://www.sdmetrics.com/index.html.