Universidad de las Ciencias Informáticas
FACULTAD 6

Título: Plugin de la herramienta Visual Paradigm para la evaluación del diseño Orientado a Objeto.

Trabajo de Diploma para optar por el título de

Ingeniero en Ciencias Informáticas

Autor(es): Yurnelis Boizán del Pozo

Alexis Galano González

Tutor(es): Yanet Rosales Morales

Leonel Vila Pérez

Consultante: Armando Robert Lobo

La Habana

Junio 2012

"Año 54 de la Revolución"

"El hombre debe transformarse al mismo tiempo que la producción progresa; no realizaríamos una tarea adecuada si fuéramos tan solo productores de artículos, de materias primas y no fuéramos al mismo tiempo productores de hombres."

Ché

DECLARACIÓN DE AUTORÍA

Declaramos ser autores de la presente tesis y reconocemos a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

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

Yurnelis Boizán del Pozo _____ Firma del Autor

Alexis Galano González _____ Firma del Autor

Yanet Rosales Morales _____ Firma del Tutor

Leonel Vila Pérez _____ Firma del Tutor

Armando Robert Lob _____ Firma del Consultante

DATOS DE CONTACTO

Autor(es):

Yurnelis Boizán del Pozo. Universidad de las Ciencias Informáticas.2011

Correo de contacto: yboizan@estudiantes.uci.cu

Alexis Galano González. Universidad de las Ciencias Informáticas.2011

Correo de contacto: agalano@estudiantes.uci.cu

Tutor(es):

Ing. Yanet Rosales Morales. Ingeniero en Ciencias Informáticas de la Universidad de las Ciencias Informáticas (UCI) 2011. Miembro del Grupo de Análisis de Software de la Línea de Integración de Soluciones del Centro de Gestión de Datos (DATEC) de la Facultad 6.

Correo de contacto: yrmorales@uci.cu

Ing. Leonel Vila Pérez. Ingeniero en Ciencias Informáticas de Universidad de las Ciencias Informática (UCI) 2011. Miembro del Grupo de Desarrollo de la Línea de Integración de Soluciones del Centro de Gestión de Datos (DATEC) de la Facultad 6. Se desempeña como especialista de desarrollo del proyecto Sistema Integrado de Gestión de Datos (SIDAT).

Correo de contacto: lvila@uci.cu

Consultante:

Ing. Armando Robert Lobo: Ingeniero en Ciencias Informáticas de la Universidad de las Ciencias Informáticas (UCI) 2007. Miembro del Grupo de Arquitectura de Software de la Línea de Integración de Soluciones del Centro de Gestión de Datos (DATEC) de la Facultad 6. Se ha desempeñado durante su vida laboral como arquitecto principal de la solución Sistema Integrado de Gestión Estadística (SIGE) de la Plataforma de apoyo a la Toma de Decisiones (PATDSI). Actualmente se encuentra al frente del proyecto Sistema de Información de Gobierno, Módulo de Encuestas (SiGOB –Encuesta) como Gestor de Proyecto (líder) y como Arquitecto de Sistema.

Correo de contacto: arobert@uci.cu

AGRADECIMIENTOS

Yurnelis Boizán (NELLY)

Si tengo que darle las gracias a todo aquel que me apoyo en el tiempo de realización de la tesis, no me alcanzarían las 80 páginas para ponerlos a todos, pero bueno todos se merecen un pequeño reconocimiento de mi parte por aguantar mi mal humor, mi preguntadera y mi cara de amargada, por eso todos se merecen MILGRACIAS del tamaño del cielo. Sin más preámbulos, les agradezco:

- *A mis padres primero que todo por darme la vida, por educarme como lo han hecho y apoyarme en mi educación. Además de darme los mejores consejos y los peores regaños que nadie se puede imaginar a causa de mi cabecita porfiada.*
- *A mi familia, que aunque este lejos, me han apoyado en todo este tiempo.*
- *A mis amistades que tanto me han ayudado y han compartido buenos y malos momentos, en especial Marla Cuza, Ana Rosa, Claudia García, Ana Margarita, Juliet Mora, Ángel López, Katerine Cazanas.*
- *A mis compañeros de aula, que me han ayudado muchas veces cuando lo he necesitado.*
- *A mis tutores, que han sabido guiarme en el desarrollo de la tesis, en especial a la incansable Yanet Rosales Morales, que me ha regañado tal nivel que pensé que no me graduaba y Armando Robert Lobo.*
- *A los profesores que han pasado por el tribunal de tesis, los cuales han sido bastantes, y han sabido corregirme y guiarme en la tesis.*
- *A mi compañero de tesis Alexis Galano, por haber podido entenderme y haber desarrollado en conjunto conmigo este trabajo de diploma.*
- *A todos aquellos que han sido parte, de una forma u otra, en este trayecto de 5 años.*

Alexis Galano

A mi familia por el apoyo que me han dado a lo largo de la carrera para poder lograr mi objetivo.

- *En especial a mi mama por el sacrificio que ha hecho no solo al traerme al mundo, sino educarme para ser un hombre de bien.*
- *A mi abuela por tener tanta fe en mí, ya que siempre me inculco que con voluntad y determinación podemos lograr cualquier cosa.*
- *A mis amigos y compañeros que compartieron los momentos malos y buenos en la universidad, en especial a Yariel Gordillo, Víctor Manuel Estrada, Osmar Ruano, Julio E. Aguilar Barquié, Marcos A. Reyes Medina, Raúl A. Rondón, Adrian Rosales, José Rolando Lafurie, ruego me perdonen si se me queda alguien ya que son muchos.*
- *Quisiera por ultimo agradecer a mis tutores, Yanet Rosales Morales y Armando Robert Lobo por haber sido pilares fundamentales en la realización de este trabajo, el cual no fuera posible sin el gran apoyo que me brindaron.*

DEDICATORIA

Yurnelis Boizán (NELLY)

Dedico este trabajo de diploma y la gratitud de haberlo desarrollado con todo el sacrificio y amor que lleva a las personas que han hecho de mí lo que soy hoy:

- *A mis padres que son mi razón de ser: Mariluz del Pozo Betancourt y Horacio Boizán Romero.*
- *A mi abuelita Rosa Lora Peña.*
- *A mi hermana Yanet Boizán.*
- *A mis primos, en especial Leodenis Boizán.*
- *A toda mi familia, y a los que ya no están en ella.*
- *A mi hermanito "Javier Boizán del Pozo", que aunque no lo pude ver crecer, se que le hubiese gustado verme graduada con un título universitario.*

Alexis Galano

Dedico este trabajo a las personas más grandes en mi vida.

- *A mi abuela por confiar siempre en mi, por apoyarme siempre, por mantenerse siempre fuerte y prometerme que iba a estar ahí en el momento que me graduara y así fue.*
- *A mi mamá por el gran amor que me ha dado, por la educación que me ha inculcado, por el sacrificio que ha hecho para que yo pudiera seguir mi camino y por ser mi razón de ser.*

RESUMEN

La etapa de diseño constituye una de las fases fundamentales dentro del proceso de desarrollo del software. Para validar si el diseño realizado cumple con los estándares establecidos, una de las técnicas fundamentales es, la aplicación de métricas de calidad. Sin embargo, este proceso se dificulta por la inexistencia de herramientas que lo automaticen, requiriendo de personal calificado que lo realicen posteriormente a la etapa de diseño. Esto conlleva a la entrega tardía de los mismos, la insatisfacción del cliente, los cambios significativos en el software y la pérdida considerable de recursos disponibles. Se propone como solución informática un plugin para la herramienta Visual Paradigm for UML para la evaluación de las métricas de calidad de software en el diseño orientado a objetos, que permita automatizar el proceso de evaluación cuantitativo y cualitativo del diseño con su respectivo aporte a la elevación de la calidad de los productos desarrollados en la Universidad de las Ciencias Informáticas (UCI).

PALABRAS CLAVE:

Métricas, plugin, diseño orientado a objeto, calidad, software.

TABLA DE CONTENIDOS

Agradecimientos III

Dedicatoria V

Resumen VI

ÍNDICE DE FIGURAS 3

ÍNDICE DE TABLAS 4

Introducción 5

Capítulo 1: Métricas de calidad de software para sistemas orientado a objeto 9

Introducción 9

1.1 Calidad en el Software 9

1.2 Métricas de calidad de software 10

1.3. Características del diseño Orientado a Objeto 13

1.4 Métricas de calidad de software para el diseño Orientado a Objeto (OO). 17

1.4 Metodologías de desarrollo de software. 21

1.4.1 Open Unified (OpenUp) 21

1.5 Lenguaje Unificado de modelado (UML) 22

1.6 Lenguaje de programación 24

1.6.1 JAVA 24

1.7 Tecnologías y herramientas de desarrollo 25

1.7.1 Visual Paradigm Enterprise Edition 5.0 25

1.7.2 NetBeans IDE 6.9 25

Conclusiones parciales 27

Capítulo 2: Análisis y Diseño del Plugin 28

Introducción 28

2.1 Modelo de Dominio 28

2.2 Propuesta de Solución 29

2.3 Descripción de los Algoritmos para el Cálculo de las Métricas 31

2.4 Especificación de los Requisitos del sistema 33

2.4.1 Requisitos Funcionales 34

2.4.2 Requisitos No Funcionales 35

2.5 Modelo de Casos de Usos del Sistema 36

2.5.1 Actores del Sistema 36

2.5.2 Diagrama de Casos de Uso del Sistema 37

2.6. Modelo de Diseño 42

2.8. Diagrama de secuencia 46

Conclusiones parciales 48

Capítulo 3: Implementación y Pruebas del Plugin 49

Introducción 49

3.1 Modelo de Implementación 49

3.1.1 Diagrama de Componente 49

3.2 Ejemplos de códigos en el plugin. 52

3.3 Pruebas de Software 54

3.4. Casos de prueba 56

Conclusiones 62

Recomendaciones 63

REFERENCIAS BIBLIOGRÁFICAS 64

ANEXOS 67

GLOSARIO DE TERMINOS 69

ÍNDICE DE FIGURAS

Figura 1: Capas de la Metodología Open Up. 22
Figura 2: Proceso evolutivo de UML. 23
Figura 3: Evolución de Netbeans. 26
Figura 4: Modelo de Dominio. 28
Figura 5: Proceso de evaluación del diseño. 29
Figura 6: Reporte de evaluación. 30
Figura 7: Diagrama de Casos de Uso del Sistema. 38
Figura 8: Diagrama de Clases del Diseño - CU Evaluar Diseño. 43
Figura 9: Ejemplo del patrón Iterador en el plugin. 45
Figura 10: Ejemplo del Patrón Singleton en el plugin. 46
Figura 11: Diagrama de Secuencia - Escenario: Evaluar Diseño. 47
Figura 12: Diagrama de componentes para el CU Evaluar Diseño. 50
Figura 13: Ejemplo de código para Cargar Archivo con los criterios 52
Figura 14: Ejemplo de código del método Evaluar. 53
Figura 15: Ejemplo de código para una métrica. 54
Figura 16: Diagrama de Clases del Diseño - CU Definir Criterio 67
Figura 17: Diagrama de Secuencia Escenario: Definir Criterio 67
Figura 18: Diagrama de componentes para el CU Definir Criterio. 68
Figura 19: Logo de la aplicación. 68

ÍNDICE DE TABLAS

Tabla 1: Definición de los principales conceptos del Modelo de Dominio. 29
Tabla 2: Descripción de los Actores del Sistema. 37
Tabla 3: Descripción del caso de uso significativo Evaluar Diseño. 42
Tabla 4: Descripción de las clases relevantes del diseño. 44
Tabla 5: Descripción de los componentes más relevantes en el CU Evaluar Diseño. 52
Tabla 6: Secciones en el CU Evaluar Diseño. 57
Tabla 7: Variables definidas a partir de la interfaz Evaluar Diseño. 58
Tabla 8: Matriz de datos para el CU Evaluar Diseño. 61

INTRODUCCIÓN

La característica que distingue el modelo de sociedad que se viene perfilando en la actualidad es la "Sociedad del Conocimiento". Esta se encuentra relacionada con el empleo constante de innovaciones, tanto tecnológicas como organizativas, surgiendo de esta forma la llamada Revolución Informática.

El auge de la industria del software y la competencia entre las empresas por dominar el mercado, definieron la calidad y el tiempo como las variables fundamentales del proceso de desarrollo del software. La deficiencia en el cumplimiento de alguno de estos factores, ya sea por una incorrecta planificación del proceso de desarrollo o por un diseño menos usable, puede tener consecuencias desfavorables para una empresa.

Debido a la variedad en la calidad de los productos, se hizo necesaria la creación de estándares que definieran la calidad de un producto y la seguridad de que el mismo es óptimo para su propósito. Dichos estándares consistían en un conjunto de acuerdos documentados que contienen especificaciones técnicas u otros criterios precisos para ser usados constantemente, como reglas, lineamientos o definiciones de características. Para obtener una evaluación de calidad a través de dichos estándares, surgen las métricas de calidad de software encargadas de medir de una manera u otra, distintos atributos del software.

Cuba avanza en su informatización priorizando el uso social y colectivo de las Tecnologías de Información y las Comunicaciones (TICs), abriendo un universo de posibilidades a pesar del bloqueo económico, comercial y financiero que le es impuesto injustamente. La Universidad de las Ciencias Informáticas (UCI) se ha convertido en la institución que marcha a la vanguardia en el desarrollo de software, todo dentro del modelo de formación vinculado a la producción que se manifiesta en la relación existente entre las facultades docentes y los centros productivos con que cuenta la universidad.

Uno de los centros enmarcados en esta esfera productiva es el Centro de Tecnologías de Datos (DATEC), que tiene como objetivo proveer consultorías implícitas en la creación de nuevas tecnologías de base de datos a partir del procesamiento y análisis de la información. El mismo desarrolla un conjunto de activos informáticos para la producción de software sobre la base de un modelo de Línea de Producto de Software (LPS) que fortalece la tecnología de desarrollo, reduciendo en tiempo y elevando la calidad de sus productos.

DATEC consta de 4 líneas de productos de software: PostgreSQL, Almacenes de Datos, Bioinformática e Integración de Soluciones. Esta última desarrolla una serie de activos informáticos reutilizables y Orientado a Objeto, en su conjunto desplegados para integración de productos, los cuales en su mayoría son enriquecidos para servicios web.

Para el modelado de los productos informáticos desarrollados se utiliza la herramienta de Ingeniería de Software Asistida por Computadora (CASE del inglés *Computer Aided Software Engineering*) Visual Paradigm. Los modelos diseñados a través de Visual Paradigm constituyen abstracciones que ocultan las complejidades de las tecnologías, los cuales dichos diseños dan pauta a la creación y desarrollo de un producto de software, que puede ser considerado o no de alta calidad.

La etapa de diseño constituye una de las fases fundamentales dentro del proceso ingenieril de desarrollo del software. Esta contempla como tarea fundamental, la transformación de los requisitos de los usuarios en representaciones o diagramas, imprescindibles para llevar a cabo y con calidad, la construcción del software como producto final. Describe el producto arquitectónico global, los subsistemas que lo componen y la manera en que se asignan a los procesadores, la asignación de clases a subsistemas y el diseño de la interfaz de usuario.

El trabajo de los ingenieros de software en este ámbito es muy complejo, pues lograr un diseño robusto y a la vez exacto en cuanto a sus exigencias, requiere de gran esfuerzo y capacidad de abstracción. Para validar si el diseño realizado cumple con los estándares establecidos por el proyecto, una de las técnicas fundamentales es, la aplicación de métricas de calidad. Sin embargo, este proceso se dificulta por la inexistencia de herramientas que lo automaticen, requiriendo de personal calificado que lo realicen posteriormente a la etapa de diseño. Esto puede generar como consecuencia la entrega tardía de los mismos y por consiguiente la insatisfacción del cliente, ya que generalmente no se logra en el tiempo establecido implementar todos los requisitos explícitos contenidos en el modelo de análisis además de ajustarse a todos los implícitos que desea el cliente. Además un diseño sin la calidad requerida puede provocar la necesidad de realizar cambios en el software, causa frecuente de afectación en variables fundamentales dentro del proceso de desarrollo de software como son el tiempo, el esfuerzo, la calidad y el costo.

En función de lo antes expuesto se identifica el **Problema Científico**:

¿Cómo contribuir a la evaluación de las métricas de calidad de software para el diseño orientado a objetos de los artefactos elaborados en la herramienta Visual Paradigm?

Se define como **Objeto de Estudio**: Métricas de calidad del software.

Enmarcado en el **Campo de Acción**: Métricas de calidad del software para el diseño orientado a objetos aplicadas a los Patrones Generales de Software para Asignación de Responsabilidades (GRASP).

Se persigue con ello el **Objetivo general**:

Desarrollar un plugin de la herramienta Visual Paradigm para evaluar las métricas de calidad de software del diseño orientado a objetos aplicados a los Patrones Generales de Software para Asignación de Responsabilidades (GRASP).

Para su consecución se han planteado los siguientes **Objetivos específicos**:

1. Realizar un análisis del marco conceptual relacionado con las métricas de calidad de software del diseño Orientado a Objeto.
2. Realizar el análisis y diseño del plugin de la herramienta Visual Paradigm.
3. Realizar la implementación del plugin de la herramienta Visual Paradigm.
4. Realizar las pruebas funcionales al plugin de la herramienta Visual Paradigm.

Para la realización de los objetivos propuestos se han planteado las siguientes **Tareas de investigación**:

1. Análisis de las métricas de calidad de software para el diseño orientado a objetos, metodología, herramientas y tecnologías para el desarrollo del plugin.
2. Aplicación de la Ingeniería de Requisitos con el fin de evidenciar el análisis de la solución.
3. Descripción de la arquitectura base del plugin.
4. Definición de los componentes de diseño que se ajusten a la arquitectura de la herramienta Visual Paradigm.
5. Implementación de los componentes de diseño.
6. Evaluación de la aplicación mediante pruebas funcionales.
7. Documentación y corrección de las no conformidades identificadas en la ejecución de las pruebas.

Al finalizar esta investigación se espera obtener un Plugin de la herramienta Visual Paradigm UML para la evaluación de las métricas de calidad de software en el diseño orientado a objetos, el cual permitirá automatizar el proceso de evaluación cuantitativo y cualitativo (hasta donde las métricas permitan) del diseño orientado a objetos con su respectivo aporte a la elevación de la calidad de los productos del Departamento de Integración de Soluciones del centro DATEC.

Para garantizar estos resultados el presente trabajo está estructurado en 3 capítulos que abordan los elementos fundamentales de la investigación organizados de la siguiente manera:

Capítulo 1: Métricas de calidad de software para Sistemas Orientado a Objeto

El capítulo comprende un breve estudio de las Métricas de Calidad de Software en específico las que van dirigidas a sistemas orientados a objeto, su aplicación tanto en el mundo como en el centro DATEC. Además se aborda acerca de la metodología, herramientas y tecnologías propuestas para desarrollar la solución de la presente investigación.

Capítulo 2: Análisis y Diseño del Plugin

En este capítulo se describe todo lo referente a las funcionalidades que debe realizar el sistema, así como el funcionamiento del mismo. Además se plasma todo el proceso de negocio, como son: actores, casos de uso del sistema, así como diseños de clases referente a la descripción de la implementación y los requisitos tanto funcionales como no funcionales que se deben tener en cuenta para la realización del plugin. Se aborda a cerca de los patrones arquitectónicos y de diseño presentes en la implementación del plugin.

Capítulo 3: Implementación y Pruebas del Plugin

En este capítulo se presenta el modelo de implementación a través del diagrama de componentes. Además se realizan pruebas de caja negra al plugin, para comprobar su correcto funcionamiento mediante los casos de prueba.

CAPÍTULO 1: MÉTRICAS DE CALIDAD DE SOFTWARE PARA SISTEMAS ORIENTADO A OBJETO

Introducción

En el presente capítulo se tratarán los conocimientos básicos para el desarrollo del software a partir del estudio de las métricas de calidad orientadas a objetos (OO).El mismo refleja todo el proceso vinculado al Lenguaje Unificado de Modelado (UML) siendo de valiosa importancia para darle solución al problema planteado en la investigación. Además se expondrán las tecnologías y herramientas seleccionadas para su desarrollo.

1.1 Calidad en el Software

En la actualidad el software constituye la tecnología individual más importante del mundo. Nadie en las décadas anteriores pudo haber predicho que el mismo se convertiría en una tecnología indispensable en los negocios, la ciencia y la ingeniería, ni que estaría relacionado con sistemas de todo tipo: de transporte, telecomunicaciones, médicos, militares, industriales y de entretenimientos, entre otros.

“Un software se forma con: las instrucciones (programas de computadora) que al ejecutarse proporcionan las características, funciones y el grado de desempeño deseados, las estructuras de datos que permiten que los programas manipulen información de manera adecuada, y los documentos que describen la operación y el uso de los programas.” [1]

El tema de la calidad adquiere la total atención de todos los miembros de equipos de desarrollo debido al poder de solidez y fiabilidad con que garantiza la plena satisfacción de un cliente en relación con el producto o servicio solicitado. A partir de esto se puede apreciar una evolución en el mundo empresarial en relación con años atrás, en las cuales no se pensaba en toda una gama de características capaces de no solo cumplir con lo pedido por el cliente, sino que además permitiera adquirir el mejor producto de la manera más sencilla y descifrable posible.

La calidad de software constituye 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]

Realizar un software de alta calidad es un proceso complejo, pues requiere de actividades de control de calidad utilizadas para satisfacer los requisitos relativos de la misma. Estas actividades están centradas en dos objetivos fundamentales: mantener bajo control el proceso de desarrollo del producto y eliminar las causas de los defectos que el mismo presente en las diferentes fases del ciclo de vida. Además debe existir una valoración independiente que pueda demostrar que la organización es capaz de desarrollar productos y servicios de calidad cumpliendo con los pilares básicos de la certificación de

calidad. Estos pilares se basan en el uso de una metodología adecuada, un medio de valoración de dicha metodología y por consiguiente que la misma debe ser reconocida ampliamente por la industria.
A la hora de realizar un producto de calidad se debe pensar y meditar a cerca de una serie de características esenciales que el mismo debe poseer, tales como: mantenibilidad, la cual garantiza que el software debe ser diseñado de tal manera, que permita ajustarlo a los cambios en los requerimientos del cliente. Esta característica es primordial, debido al inevitable cambio del contexto en el que se desempeña un software. Además la confidencialidad juega un papel crucial pues en ella está implícita la seguridad y control de los fallos del producto. La eficiencia del mismo enmarcada en el eficiente uso de los recursos del sistema así como la capacidad de que el software sea utilizado sin un gran esfuerzo por los usuarios para los que fue diseñado, siendo esto un gran requisito de usabilidad.
Se requieren las herramientas precisas que ayuden al equipo para llevar adelante todas las tareas necesarias en relación a alcanzar los objetivos de calidad planteados y es muy importante también, disponer de personas preparadas técnicamente y liderados por al menos un profesional con experiencia. La calidad en el software debe ser catalogada y precisada con la mayor dedicación posible y es para esto que surgen las métricas de calidad del software, con el propósito de medir en términos confiables la calidad de los productos.

1.2 Métricas de calidad de software

El objetivo principal de la ingeniería del software es producir un sistema, aplicación o producto de alta calidad. Para lograr este objetivo, los ingenieros de software deben emplear métodos efectivos junto con herramientas modernas dentro del contexto de un proceso maduro de desarrollo del software. Al mismo tiempo, un buen ingeniero y administradores de la ingeniería del software deben medir si la alta calidad se va a llevar a cabo.
Desde los albores de la Informática se han tratado de medir de una manera u otra, distintos atributos del software. Estos atributos pueden ser: el tamaño, la complejidad, la frecuencia esperada de aparición de errores, cobertura de pruebas, o incluso atributos del proceso software como pueden ser la productividad.
Para obtener la evaluación de calidad del software a través de estos atributos, se deben utilizar medidas técnicas que evalúan con objetividad. A partir de la necesidad de estandarizar estas medidas, surgen las llamadas métricas de calidad, proporcionando una indicación de la efectividad de las actividades de control y de la garantía de calidad del software.
“Una métrica es una medida cuantitativa, del grado en que un sistema, componente o proceso posee un atributo determinado”. [1]

La medición es esencial para cualquier disciplina y la ingeniería de software no es una excepción. Las métricas de software se refieren a un amplio rango de medidas para el software de computadoras dentro del contexto de la planificación del proyecto de software, las métricas de calidad pueden ser aplicadas a organizaciones, procesos y productos los cuales directamente afectan a la estimación de costos.

Las métricas de software se definen como “La aplicación continua de mediciones basadas en técnicas para el proceso de desarrollo del software y sus productos para suministrar información relevante a tiempo, así el administrador junto con el empleo de estas técnicas mejorará el proceso y sus productos”. [2]

La importancia de las métricas radica en su posibilidad de establecer pronósticos y tendencias a partir de un determinado número de variables e indicadores científicos para la toma de decisiones. Su valor no reside solamente en la posibilidad de obtener resultados cuantitativos que apoyen la toma de decisiones, sino en su capacidad para estudiar la ciencia a nivel general como fenómeno social con el apoyo de las matemáticas.

Estas se catalogan de diversas formas, tales como:

- **De complejidad**: las cuales constituyen métricas que definen la medición de la complejidad: volumen, tamaño, nidaciones, y configuración.
- **De calidad**: son métricas que definen la calidad del software: exactitud, estructuración o modularidad, pruebas, mantenimiento.
- **De competencia**: son métricas que intentan valorar o medir las actividades de productividad de los programadores con respecto a su certeza, rapidez, eficiencia y competencia.
- **De desempeño**: las cuales miden la conducta de módulos y sistemas de un software, bajo la supervisión del SO o hardware.
- **Estilizadas**: son métricas de experimentación y de preferencia: estilo de código, convenciones, limitaciones, etc.
- **De Proyecto**: las medidas del proyecto de software son tácticas. Las métricas de proyectos y los indicadores derivados de ellos los utilizan tanto un administrador de proyectos como un equipo de software para adaptar el flujo de trabajo del proyecto y las actividades técnicas.
- **De Producto**: los productos del software son las salidas del proceso de producción del software. Éstas incluyen todos los artefactos entregados o documentos que son productos durante el ciclo de vida del software.

- **De Proceso**: las entidades de procesos de software incluyen actividades relacionadas con el software y eventos que usualmente son asociados con un factor de tiempo. Las métricas del proceso de software se utilizan para propósitos estratégicos.

Todas estas métricas contribuyen a la perfección del software, sobre todo las métricas de calidad seleccionadas para la elaboración del plugin. Es de suma importancia que se realice una exhaustiva medición de la calidad del software, ya que los productos son cada vez más complejos y sofisticados y por ende, la calidad que se exige para los mismos. A partir de esto se cree que las métricas son un buen medio para entender, monitorizar, controlar, predecir y probar el desarrollo software y los proyectos de mantenimiento.

Se han propuesto cientos de métricas para el software, pero no todas proporcionan suficiente soporte práctico para su desarrollo. Algunas demandan mediciones que son demasiado complejas, otras son tan esotéricas que pocos profesionales tienen la esperanza de entenderlas, y otras violan las nociones básicas intuitivas de lo que realmente es el software de alta calidad. Es por eso que se han definido una serie de atributos que deben acompañar a las métricas efectivas de software, por lo tanto la métrica obtenida y las medidas que conducen a ello deben cumplir con las siguientes características fundamentales:

- Simple y fácil de calcular: debería ser relativamente fácil de aprender a obtener la métrica y su cálculo no obligara a un esfuerzo o a una cantidad de tiempo inusuales.
- Empírica e intuitivamente persuasiva: la métrica debería satisfacer las nociones intuitivas del ingeniero de software sobre el atributo del producto en cuestión (por ejemplo: una métrica que mide la cohesión de un módulo debería aumentar su valor a medida que crece el nivel de cohesión).
- Consistente en el empleo de unidades y tamaños: el cálculo matemático de la métrica debería utilizar medidas que no lleven a extrañas combinaciones de unidades. Por ejemplo, multiplicando el número de personas de un equipo por las variables del lenguaje de programación en el programa resulta una sospechosa mezcla de unidades que no son intuitivamente concluyentes. [1]
- Patrones Generales de Software para Asignación de Responsabilidades (GRASP).

Según Pressman [’98], las métricas son la maduración de una disciplina, que ayudarán a la evaluación de los modelos de análisis y de diseño, en donde proporcionarán una indicación de la complejidad de diseños procedimentales y de código fuente, además de que las pruebas en el diseño sean más efectivas. Es por eso que se propone un proceso de medición en la disciplina de Diseño, definiendo métricas que puedan ser útiles para analizar la calidad del diseño de un producto de software, desarrollado bajo el paradigma orientado a objetos y en los diferentes lenguajes utilizados en la UCI.

1.3. Características del diseño Orientado a Objeto

El diseño es el centro de atención al final de la fase de elaboración y el comienzo de las iteraciones de construcción, siendo la primera de las tres actividades técnicas: diseño, generación de código y pruebas, que se requieren para construir y verificar el software. Su importancia se puede describir con una sola palabra -calidad-, ya que es el lugar en donde se fomentará, proporcionando las representaciones del software que se pueden evaluar en cuanto a este indicador. Es la única forma de convertir exactamente los requisitos de un cliente en un producto o sistema de software finalizado. Constituye la fase que no deberá saltarse nunca durante el desarrollo del software puesto a que sin él se corre el riesgo de construir un sistema inestable con grandes probabilidades de fallar cuando se lleven a cabo cambios en su sistema. Además de resultar muy difícil de comprobar y cuya calidad no puede evaluarse hasta muy avanzado el proceso, sin tiempo suficiente y con un elevado gasto monetario. El diseño convencional se centra en la arquitectura del software y en la definición de subsistemas, pero a partir de la necesidad de ver los problemas como objetos de la vida real, surge el Diseño Orientado a Objeto.

El diseño orientado a objeto (DOO) cumple una serie de requisitos que se concretan en la necesidad de la existencia de una arquitectura de software, en la que se especifiquen subsistemas que realicen funciones y provean soporte de infraestructura de objetos (clases). El mismo en similitud al diseño convencional realiza transformaciones en su conjunto para convertir el modelo de análisis en diseño. Para esto se apoya en cuatro capas principales que describen su total funcionamiento a partir de la colaboración entre objetos, tales como:

- “Diseño de subsistemas”, el cual contiene una representación de cada uno de los subsistemas, para permitir al software conseguir sus requisitos definidos por el cliente e implementar la infraestructura que soporte estos requerimientos.
- “Diseño de clases y objetos”, que está compuesto por jerarquía de clases que permiten al sistema ser creado usando generalizaciones y cada vez especializaciones más acertadas.
- “Diseño de mensajes” que contiene detalles del diseño que permite a cada objeto comunicarse con sus colaboradores.
- “Diseño de responsabilidades”, compuesto por estructuras de datos y diseños algorítmicos, para todos los atributos y operaciones de cada objeto. [3]

La principal virtud que tiene el diseño OO radica en su capacidad de definir y desarrollar diferentes conceptos que hacen atractivo y más robusto a los sistemas OO. Los cuales se definen a continuación:

- Abstracción

La abstracción es un mecanismo que permite al diseñador concentrarse en los detalles esenciales de un componente de programa (ya sean datos o procesos), prestando poca atención a los detalles de bajo nivel.

- Herencia

La herencia es un mecanismo que hace posible que los compromisos de un objeto se difundan a otros objetos. Se produce a lo largo de todos los niveles de la jerarquía de clases y apunta a reducir la cantidad de trabajo de mantenimiento del software necesario y aliviar así la carga de la actividad de prueba. La reutilización de software a través de la herencia apunta a producir un software mantenible, entendible y fiable.

- Ocultación de información

La ocultación de información suprime (u oculta) los detalles operacionales de un componente de programa. Solo se proporciona la información necesaria para acceder al componente a aquellos otros componentes que deseen acceder.

- Encapsulamiento

El encapsulamiento se define como “el empaquetamiento” (o enlazado) de una colección de elementos. Comprende las responsabilidades de una clase, incluyendo sus atributos (y otras clases para objetos agregados) además de operaciones y los estados de la clase.

- Polimorfismo

El polimorfismo representa un concepto de teoría de tipos, en el que un solo nombre (tal como una declaración de una variable), puede denotar instancias de muchas clases diferentes, en tanto en cuanto estén relacionadas por alguna superclase común. Cualquier objeto denotado por este nombre es capaz de responder a algún conjunto común de operaciones, de diversas formas. El polimorfismo, se vuelva más útil cuando existen muchas clases con los mismos protocolos. Con su utilización no son necesarias grandes sentencias clase, porque cada objeto conoce implícitamente su propio tipo.

- Modularidad

La modularidad es el único atributo del software que permite gestionar un programa. El software se divide en componentes nombrados y abordados por separado, llamados frecuentemente módulos, que se integran para satisfacer los requisitos del problema.

- Independencia funcional

Es la suma de la modularidad y de los conceptos de abstracción y ocultación de información. Alude a las técnicas de refinamiento que mejoran la independencia de módulos. Los módulos independientes son más fáciles de mantener (y probar) porque se limitan los efectos secundarios originados por

modificaciones de diseño/código, pues se reduce la propagación de errores y es posible utilizar módulos usables. [3]

En el diseño orientado a objetos resulta complicado descomponer el sistema en objetos (encapsulación, granularidad, dependencias, flexibilidad, reusabilidad, etc.), los patrones de diseño permiten identificar a los objetos apropiados de una manera mucho más sencilla y determinar la granularidad de los mismos. Estos a su vez permiten ante un problema reiterado ofrecer una solución para resolverlo. Describen el problema en forma sencilla, el contexto en que ocurre, los pasos a seguir, los puntos fuertes y débiles de la solución y si existen otros patrones asociados.

"Un patrón es una solución a un problema de diseño no trivial que es efectiva y reusable." [1]

La utilización de patrones de diseño produce beneficios en el software, tales como:

- Contribuyen a reutilizar diseño, identificando aspectos claves de la estructura de un diseño que puede ser aplicado en una gran cantidad de situaciones. Esto reduce los esfuerzos de desarrollo y mantenimiento, mejora la seguridad, eficiencia y consistencia de nuestros diseños, además de proporcionar un considerable ahorro en la inversión.
- Mejoran (aumentan, elevan) la flexibilidad, modularidad y extensibilidad, factores internos e íntimamente relacionados con la calidad percibida por el usuario.
- Incrementan nuestro vocabulario de diseño, ayudándonos a diseñar desde un mayor nivel de abstracción.

Para la evaluación del diseño OO en la investigación se utilizarán los Patrones Generales de Software para Asignación de Responsabilidades (GRASP), ya que describen los principios fundamentales de diseño de objetos para la asignación de responsabilidades y poseen principal correlación con la programación OO, específicamente la compatibilidad con las métricas que se deben usar en la medición de la calidad en la investigación.

Los patrones GRASP se pueden clasificar en 5 tipos principales:

- **Experto:**

Es el principio básico de asignación de responsabilidades. Indica que la responsabilidad de la creación de un objeto o la implementación de un método, debe recaer sobre la clase que conoce toda la información necesaria para crearlo. De este modo se obtiene un diseño con mayor cohesión y así la información se mantiene encapsulada (disminución del acoplamiento).

Beneficios:

Se mantiene el encapsulamiento de la información, puesto que los objetos utilizan su propia información para llevar a cabo las tareas. Normalmente, esto conlleva un bajo acoplamiento, lo que da

lugar a sistemas más robustos y más fáciles de mantener. Se distribuye el comportamiento entre las clases que contienen la información requerida, por tanto, se estimula las definiciones de clases más cohesivas y "ligeras" que son más fáciles de entender y mantener. Se soporta normalmente una alta cohesión. [4]

- **Creador:**

Propone que un objeto tiene la responsabilidad de crear objetos de otra clase si agrega, contiene y usa exhaustivamente objetos de dicha clase. Además posee la información necesaria de inicializar esta última clase, es decir:
Asignar a la clase B la responsabilidad de crear una instancia de la clase A si se cumple una o más de los casos siguientes:

- B agrega objetos de A
- B contiene objetos de A
- B registra instancias de objetos de A
- B utiliza más estrechamente objetos de A
- B tiene los datos de inicialización que se pasará a un objeto de A cuando sea creado (por tanto, B es un Experto con respecto a la creación de A)

Si se asigna bien la responsabilidad de creación, el diseño puede soportar un bajo acoplamiento, mayor claridad, encapsulación y reutilización. [4]

- **Alta Cohesión:**

Cohesión funcional dentro de una clase es una medida que indica cuán relacionadas están las responsabilidades de una clase. Es un patrón evaluativo: entre más alta cohesión más fácil de entender, de cambiar, de reutilizar. No puede ser considerado aisladamente.
Una clase con baja cohesión hace muchas cosas no relacionadas, o hace demasiado trabajo. Tales clases no son convenientes, adolecen de los siguientes problemas:

- Difíciles de entender
- Difíciles de reutilizar
- Difíciles de mantener
- Delicadas, constantemente afectadas por los cambios.

Como regla empírica, una clase con alta cohesión tiene un número relativamente pequeño de métodos, con funcionalidad altamente relacionada, y no realiza mucho trabajo. Colabora con otros objetos para compartir el esfuerzo si la tarea es extensa. [4]

- **Controlador:**

Asigna la responsabilidad de recibir o manejar un mensaje de evento del sistema a una clase que representa una de las siguientes opciones:

- Representa el sistema global, dispositivo o subsistema (controlador de fachada).
- Representa un escenario de caso de uso en el que tiene lugar el evento del sistema, a menudo denominado Manejador, Coordinador o Sesión (controlador de sesión o de caso de uso).
- Utilice la misma clase controlador para todos los eventos del sistema en el mismo escenario de caso de uso.
- Informalmente, una sesión es una instancia de una conversación con un actor. Las sesiones pueden tener cualquier duración, pero se organizan a menudo en función de los casos de uso (sesiones de caso de uso).[4]

- **Bajo Acoplamiento:**

El acoplamiento es la medida de cuánto una clase está conectada (tiene conocimiento) de otras clases. Es un patrón evaluativo: un bajo acoplamiento permite que el diseño de clases sea más independiente. Reduce el impacto de los cambios y aumenta la reutilización. No puede ser considerado aisladamente. Puede no ser importante si la reutilización no es un objetivo. [4]

1.4 Métricas de calidad de software para el diseño Orientado a Objeto (OO).

El paradigma de la orientación a objetos posee características específicas que lo diferencia de enfoques más tradicionales como la programación estructurada. Esto implica que métricas aplicadas en programación estructurada pueden no ser válidas en orientación a objetos. Surgen así nuevas métricas que pretenden describir de manera adecuada estas nuevas características del software, tales como encapsulamiento, ocultamiento de la información, abstracción, herencia y polimorfismo.

Las mediciones pueden ser catalogadas en dos campos: medidas directas como una métrica de un atributo que no depende de ninguna métrica de otro atributo y medidas indirectas como una métrica de un atributo que se deriva de una o más métricas de otros atributos, formalizándose por medio de una

Ecuación, la cual es un algoritmo o cálculo que permite calcular una métrica a partir de valores predeterminados.

Gran parte del diseño orientado a objetos es subjetivo. Un diseñador experimentado sabe cómo puede caracterizar un sistema 00 para que se implemente de forma efectiva los requisitos del cliente. Pero a medida que los modelos de diseño 00 van creciendo de tamaño y complejidad, puede resultar beneficiosa una visión más objetiva de las características del diseño, tanto para el diseñador experimentado como para el menos experimentado, lo cual esto conlleva al uso de componentes cuantitativos, procurando la utilización de métricas OO.

Los objetivos principales de las métricas orientadas a objetos son los mismos que los existentes para las métricas surgidas para el software estructurado:

- Evaluar mejor la calidad del producto.
- Estimar la efectividad del proceso.
- Mejorar la calidad del trabajo realizado en el nivel del proyecto.

Constituye de gran utilidad el uso de métricas que ayuden a la detección de errores y a la mejora de la calidad en el diseño OO en fases tempranas, antes incluso de que dicho diseño se haya implementado en código. Por una parte, interesa saber si un determinado diseño se ajusta a un determinado nivel de calidad fijado de antemano y por otra parte, poder juzgar si el diseño de un producto software ya existente, tiene un nivel de calidad suficiente para ser susceptible de mejoras o bien merece la pena rehacer todo el producto desde sus fases iniciales.

A continuación se presenta la selección de las métricas para el diseño, teniendo en cuenta el enfoque orientado a objetos, el nivel de granularidad y la etapa del ciclo de vida en que se puede medir.

1 **Métrica Acoplamiento entre objetos (*Coupling Between Objects* – CBO) [Chidamber y Kemerer, 1994]**

Definición:

CBO de una clase es el número de clases a las cuales una clase está relacionada, sin tener con ella relaciones de herencia. Hay dependencia entre dos clases cuando una de ellas usa métodos o variables de la otra clase. Es consistente con las tradicionales definiciones de acoplamiento: “medida del grado de interdependencia entre módulos”. Esta métrica responde al patrón de Bajo Acoplamiento.

Valoración:

Los autores sugieren que sea un indicador del esfuerzo necesario para el mantenimiento y las pruebas. Cuanto más independiente es un objeto, más fácil es reutilizarlo en otra aplicación. Al reducir el acoplamiento se reduce la complejidad, se mejora la modularidad y se promueve el encapsulamiento. Una medida de acoplamiento es útil para determinar la complejidad de las pruebas

necesarias de distintas partes de un diseño. Cuanto mayor sea el acoplamiento entre objetos más rigurosas han de ser las pruebas.

2 Proporción de atributos heredados (*Attribute Inheritance Factor* - AIF) [Abreu y Melo, 1996]

Definición:

Es la proporción entre el número de atributos heredados y el número total de atributos. Esta métrica responde al patrón Creador.

Valoración:

Es un indicador de la capacidad de reutilización en un sistema.

3 Métodos ponderados por clase (*Weighted Methods per Class* - WMC) [Chidamber y Kemerer, 1994]

Definición:

Dada una clase C1 con los métodos M1,..., Mn y c1,..., cn la complejidad de los métodos, WMC se define como la sumatoria de las complejidades de cada método de una clase. Dado que en el diseño no es posible medir la complejidad de los métodos, son considerados de igual complejidad, entonces c1=1 y WMC=n (número de métodos). Esta métrica responde al patrón Alta Cohesión.

Valoración:

Describe la complejidad algorítmica de una clase en términos de las complejidades de todos sus métodos. Está relacionada a la calidad de la definición de complejidad de un método (ci). Los autores la simplifican asignando 1 a cada método, convirtiéndose así en un simple contador del número de métodos dentro de una clase. En este caso habría que considerarla como una medida del tamaño de una clase y no de complejidad, ya que una clase puede tener pocos métodos pero muy complejos y otra clase puede tener muchos métodos pero muy simples. Puede servir como un indicador de que una clase determinada necesite una descomposición adicional en varias clases.

4 Proporción de métodos heredados (*Method Inheritance Factor* - MIF) [Abreu y Melo, 1996]

Definición:

Es la proporción entre la suma de todos los métodos heredados en todas las clases y el número total de métodos (localmente definidos más los heredados) en todas las clases. Esta métrica responde al patrón Creador.

Valoración:

Es un indicador del nivel de reutilización. También se propone como ayuda para evaluar la cantidad de recursos necesarios a la hora de probar.

5 Número total de clases en el diseño (*Design size of classes* - DSC) [Bansiya y Davis]

Definición:

Es el número total de clases presentes en el diseño. Esta métrica responde al patrón Bajo Acoplamiento.

Valoración:

Es un indicador del nivel de reutilización. Da una visión de cuan un diseño está sobrecargado en cuanto a cantidad de clases se trata, pudiendo identificar la innecesaria creación de clases, asignándole responsabilidades que otras clases pueden realizar sin dificultad sin alterar el diseño y las posteriores pruebas en el software.

6 Número de métodos polimórficos(*Number of polymorphic methods* - NPM) [Bansiya y Davis]

Definición:

Es el número total de métodos polimórficos presentes en las clases. Esta métrica responde al patrón Creador.

Valoración:

De existir valores elevados de NPM, éstos mostrarán un uso inadecuado y declaraciones erróneas de abstracciones en el diseño.

7 Tamaño de clase (*Class Size* - TC) [Lorenz y Kidd]

Definición:

Es la suma del número total de operaciones (tanto operadores heredadas como privadas de la instancia) que están encapsuladas dentro de la clase y el número de atributos (tanto heredados como privados de la instancia) que están encapsulados en la clase. Esta métrica responde al patrón Bajo Acoplamiento y Alta Cohesión.

Valoración:

Si existen valores grandes de TC, éstos mostrarán que una clase puede tener demasiadas responsabilidades, lo cual reducirá la reusabilidad de la clase, y complicará la implementación y la comprobación. Por otra parte cuanto menor sea el valor medio para el tamaño, más probable es que las clases existentes dentro del sistema se puedan reutilizar ampliamente.

8 Acoplamiento de abstracción de datos (*Data abstraction coupling* - DAC)[Li y Henry,1993]

Definición:

El número de atributos en una clase que tienen como tipo otra clase. Esta métrica responde al patrón Bajo Acoplamiento.

Valoración:

Esta métrica mide la abstracción y el acoplamiento entre las clases. Valores altos de DAC indicarán innumerables declaraciones erróneas en la implementación, reduciendo la reusabilidad de las clases.

1.4 Metodologías de desarrollo de software.

El desarrollo de un buen software constituye una ardua tarea que debe contar con un equipo de desarrolladores dispuesto a asumir altas responsabilidades y que a su vez cuenten con un grupo de procedimientos que los apoyen en el logro del éxito del producto. Para lograr esto se emplean las metodologías de desarrollo de software.

"Una metodología de desarrollo de software es un conjunto de procedimientos, técnicas, herramientas y un soporte documental que ayuda a los desarrolladores a realizar un nuevo software". [5]

Las metodologías se pueden agrupar en dos grandes grupos, las tradicionales y las ágiles. Las metodologías ágiles dan mayor valor al individuo, a la colaboración con el cliente y al desarrollo incremental del software con iteraciones muy cortas, en estas el cliente llega a formar parte del equipo de trabajo. Las metodologías tradicionales se centran especialmente en el control del proceso, estableciendo rigurosamente las actividades involucradas, los artefactos que se deben producir, además de las herramientas y notaciones que se usarán.

1.4.1 Open Unified (OpenUp)

OpenUp es una variante ágil del Proceso Unificado que aplica el desarrollo iterativo e incremental dentro de la estructura del ciclo de vida. Se adopta al pragmatismo y la filosofía ágil que se centra en la colaboración natural del desarrollo de software. Constituye un proceso iterativo para el desarrollo de software mínimo, pues incluye el contenido del proceso fundamental, ya que puede ser manifestado como proceso íntegro para construir un sistema extensible debido a que puede ser utilizado como base para agregar o para adaptar más procesos.

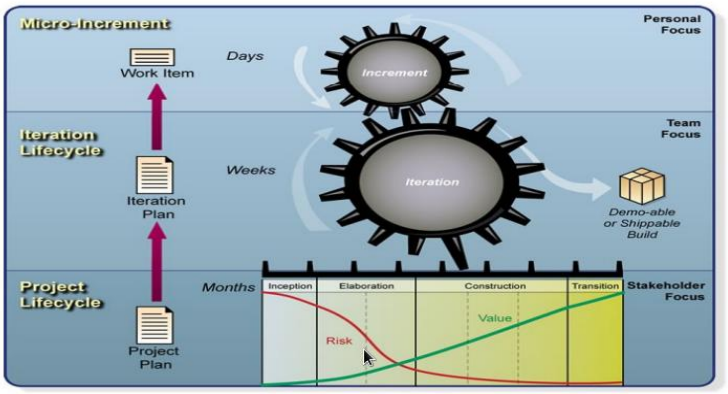


Figura 1: Capas de la Metodología Open Up.

Dicha metodología adopta como principios colaborar para alinear intereses y para compartir conocimiento. Se centra en balancear las prioridades para maximizar las necesidades de los stakeholders[1]. Esto permite el aporte de beneficios a los que lo utilizan ya que es apropiado para proyectos pequeños y de bajos recursos. Disminuye las probabilidades de fracaso en dichos proyectos e incrementa las probabilidades de éxito. Ayuda en la detección de errores temprano a través de un ciclo iterativo. Evita la elaboración de documentación, diagramas e iteraciones innecesarios requeridos en la metodología RUP y por ser una metodología ágil tiene un enfoque centrado al cliente y con iteraciones cortas.

Desde la perspectiva de los stakeholders del proyecto, OpenUp estructura el ciclo de vida en 4 fases: Inicio, Elaboración, Construcción y Transición. El ciclo de vida provee la visibilidad y los puntos de decisión tanto para stakeholders como miembros del equipo, permitiendo una vigilancia efectiva del proceso de desarrollo y facilitado la toma de decisiones apropiadas en cada instante. Consta de iteraciones planeadas con encajonamiento de tiempo en intervalos que no pasan de unas pocas semanas y centradas en producir de una manera predecible un incremento del valor para los stakeholders.

Teniendo en cuenta las características que presenta la metodología y en correspondencia con el centro DATEC, se decide utilizar OpenUP como guía en el proceso de desarrollo de software del sistema a desarrollar.

1.5 Lenguaje Unificado de modelado (UML)

Cualquier rama de ingeniería o arquitectura ha encontrado útil desde hace mucho tiempo la representación de los diseños de forma gráfica. Desde los inicios de la informática se han estado utilizando distintas formas de representar los diseños de una forma más bien personal o con algún modelo gráfico. La falta de estandarización en la manera de representar gráficamente un modelo impedía que los diseños gráficos realizados se pudieran compartir fácilmente entre distintos diseñadores.

Definición de UML:

[1] Stakeholder es un término inglés utilizado por primera vez por R. E. Freeman en su obra: “Strategic Management: A Stakeholder Approach”, (Pitman, 1984) para referirse a «quienes pueden afectar o son afectados por las actividades de una empresa»...

“El lenguaje para modelado unificado (Unifed Modeling Languaje), es un lenguaje para la especificación, visualización, construcción y documentación de los artefactos de un proceso de sistema intensivo.” [6]

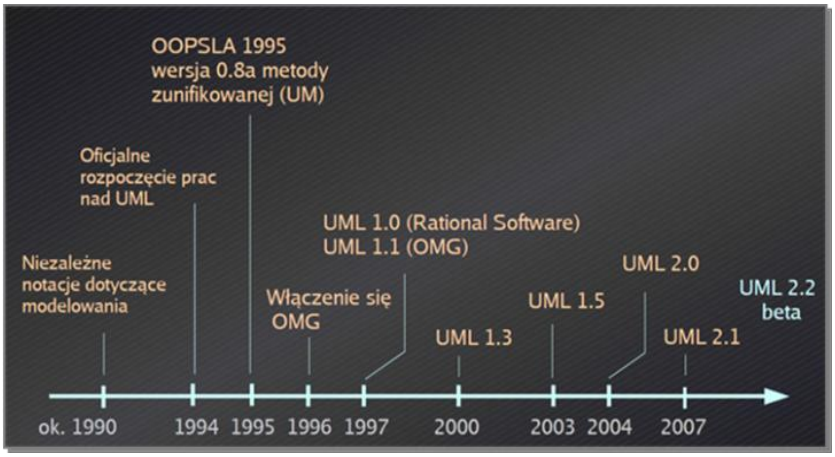


Figura 2: Proceso evolutivo de UML.

Proporciona una forma estándar de escribir los planos de un sistema. Cubre tanto las cosas conceptuales, tales como procesos del negocio y funciones del sistema, como las cosas concretas, tales como las clases escritas en un lenguaje de programación específico. Además está relacionada con esquemas de bases de datos y componentes software reutilizables.

UML está pensado en modelos para sistemas complejos con gran cantidad de software, el lenguaje es lo suficientemente expresivo como para modelar sistemas que no son informáticos, como flujos de trabajo en una empresa, diseño de la estructura de una organización y por supuesto, en el diseño de hardware. El mismo no tiene propietario y está abierto para todos.

Entre sus principales ventajas están:

- Estar apoyado por el Grupo de Gestión de Objetos (OMG - Object Management Group) como la notación estándar para el desarrollo de proyectos informáticos.
- Es útil para el desarrollo de modelaje visual de cualquier proyecto no solo informático y más aun es estándar.
- Promueve la reutilización.
- Su unificación permite que sea interpretado por cualquier analista en cualquier parte del mundo.
- Su independencia, ya que no depende de una herramienta por lo cual el análisis y Diseño que este notado en UML podrá ser implementado en cualquier lenguaje.

- Fácil de interpretar las necesidades e interacciones entras las clases, objetos por lo cual se ve el sistema desde lo más amplio hasta su detalle, proporcionando una documentación más interactiva con los programadores.

Teniendo en cuenta las características que presenta el lenguaje unificado de modelado y en correspondencia con el centro DATEC, se decide utilizar UML en su versión 2.0

1.6 Lenguaje de programación

Un lenguaje de programación es un idioma artificial diseñado para expresar cómputos que pueden ser llevadas a cabo por máquinas como las computadoras. Pueden usarse para crear programas que controlen el comportamiento físico y lógico de una máquina, para expresar algoritmos con precisión, o como modo de comunicación humana. Está formado por un conjunto de símbolos, reglas sintácticas y semánticas que definen su estructura además del significado de sus elementos y expresiones. [7]

1.6.1 JAVA

Java es un lenguaje de programación orientado a objetos, desarrollado por Sun Microsystems a principios de los años 90. El lenguaje en sí mismo toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria. Con respecto a la memoria, su gestión no es un problema ya que ésta es gestionada por el propio lenguaje y no por el programador.

Fue diseñado para crear software altamente fiable. Para ello proporciona numerosas comprobaciones en compilación y en tiempo de ejecución. Sus características de memoria liberan a los programadores de una familia entera de errores (la aritmética de punteros), ya que se ha prescindido por completo los punteros y la recolección de basura elimina la necesidad de liberación explícita de memoria.

Este constituye un lenguaje de alto nivel y sus características más importantes son:

- Lenguaje orientado a objetos.
- Lenguaje sencillo.
- Independiente de plataforma.
- Brinda un gran nivel de seguridad.
- Capacidad multihilo.
- Gran rendimiento.
- Creación de aplicaciones distribuidas.

- Su robustez o lo integrado que tiene el protocolo TCP/IP[2] lo que lo hace un lenguaje ideal para Internet.

Para la implementación del plugin se decide utilizar java, por ser el lenguaje de programación propuesto por el API de desarrollo de la herramienta "Visual Paradigm", teniendo en cuenta las características que presenta el mismo.

1.7 Tecnologías y herramientas de desarrollo

1.7.1 Visual Paradigm Enterprise Edition 5.0

Como herramienta para el modelado del diseño de la solución se ha seleccionado Visual Paradigm ya que soporta el ciclo de vida completo del desarrollo de software: análisis y diseño orientados a objetos, construcción, pruebas y despliegue. [8] Presenta un diseño centrado en casos de uso y proporciona a los desarrolladores de software una interfaz simple y amigable, con muchas opciones tales como: diversidad de idiomas y generación de código para varios lenguajes de programación. Posee facilidad para la instalación y actualización, así como compatibilidad entre sus ediciones. También facilita la interoperabilidad con otras herramientas Case y la mayoría de los principales entornos de desarrollo integrados (IDE). Presenta licencia gratuita cuando es usada para el sistema operativo Linux.

Posee características gráficas muy cómodas que facilitan la realización de los diagramas de modelado como son:

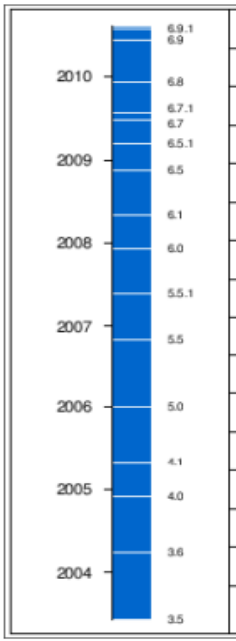
- Facilita la interoperabilidad con otras herramientas CASE como Rational Rose.
- Se integra con diversos entornos de desarrollo como: NetBeans (de Sun), Eclipse (de IBM), JDeveloper (de Oracle), JBuilder (de Borland).
- Está disponible en varias ediciones: Enterprise, Professional, Community, Standard, Modeler y Personal.
- Genera código y realiza ingeniería inversa para diferentes lenguajes de programación como: Java, C++, CORBA IDL, PHP, XML Schema y ADA.
- En adición se genera código para C#, Visual Basic.net, Object Definition Lenguaje (ODL), Flash Action Script, Delphi, Perl y Python.
- Se integra con el Visio para importar imágenes del mismo para realizar los diagramas de despliegue. Además exporta e importa los diagramas en el estándar XML 1. [8]

1.7.2 NetBeans IDE 6.9

[2] TCP/IP es la base del Internet que sirve para enlazar computadoras que utilizan diferentes sistemas operativos, incluyendo PC, minicomputadoras y computadoras centrales sobre redes de área local y área extensa

El NetBeans es un entorno de desarrollo integrado (IDE) modular y basado en estándares, escrito en el lenguaje de programación Java. Es una aplicación de código abierto ("open source") diseñada para el desarrollo de aplicaciones fácilmente portables entre las distintas plataformas. El mismo está pensado para escribir, compilar, depurar y ejecutar programas. Dispone de soporte para crear interfaces gráficas de forma visual, desarrollo de aplicaciones web, control de versiones, colaboración entre varias personas, creación de aplicaciones compatibles con teléfono móvil y resaltado de sintaxis.
La plataforma ofrece servicios comunes a las aplicaciones de escritorio, permitiéndole al desarrollador enfocarse en la lógica específica de su aplicación. Entre las características de la plataforma están:

- Administración de las interfaces de usuario (ej. menús y barras de herramientas).
- Administración de las configuraciones del usuario.
- Administración del almacenamiento (guardando y cargando cualquier tipo de dato).
- Administración de ventanas.
- Framework basado en asistentes (diálogo paso a paso).[6]



Versión	Fecha de Lanzamiento
NetBeans 6.9.1	4 de agosto de 2010
NetBeans 6.9	15 de Junio de 2010
NetBeans 6.8	10 de Diciembre de 2009
NetBeans 6.7.1	27 de Julio de 2009
NetBeans 6.7	29 de Junio de 2009
NetBeans 6.5	Noviembre 25 de 2008
NetBeans 6.1	Abril 28 de 2008
NetBeans 6.0	Diciembre 3 de 2007
NetBeans 5.5.1	Mayo 24 de 2007
NetBeans 5.5	Octubre 30 de 2006
NetBeans 5.0	Enero de 2006
NetBeans 4.1	Mayo de 2005
NetBeans 4.0	Diciembre de 2004
NetBeans 3.6	Abril de 2004
NetBeans 3.5	Junio de 2003

Figura 3: Evolución de Netbeans.

Todas las funciones del IDE son provistas por módulos. Cada módulo provee una función bien definida, tales como el soporte de Java, edición, o soporte para el sistema de control de versiones. NetBeans contiene todos los módulos necesarios para el desarrollo de aplicaciones Java en una sola descarga, permitiéndole al usuario comenzar a trabajar inmediatamente. Cada vez que evoluciona ofrece funcionalidades más seguras y mayor soporte al lenguaje, posibilitando el uso de nuevas

versiones de las cuales el NetBeans 6.9 constituye una de las más actualizadas la cual será utilizada en el desarrollo de la investigación.
Se decide utilizar en correspondencia con el centro como herramienta para el desarrollo del plugin, el IDE NetBeans 6.9 por las características que presenta.

Conclusiones parciales

Luego del estudio realizado en cuanto a los términos apropiados al problema, la evolución de UML enfocado al proceso de desarrollo de software; el análisis de las metodologías y tecnologías necesarias para el desarrollo de la herramienta, y teniendo en cuenta principalmente las exigencias y necesidades del centro, se define, OpenUP como metodología de desarrollo de software ya que es una metodología ágil para proyectos de corta duración diseñada para pequeños equipos de trabajo. Utilizando como lenguaje de modelado UML adoptado por la Universidad como estándar para el desarrollo de software. Se emplea Java como lenguaje de programación, propuesto por el API de desarrollo de la herramienta Visual Paradigm designada la misma como herramienta de modelado en su versión 5.0, adoptado en el Centro DATEC en su entorno tecnológico. Además, luego de un estudio profundo en cuanto a diseño y calidad de software se refiere, entre la variedad de métricas para evaluar la calidad del mismo, se seleccionaron 8 para la evaluación del diseño orientado a objeto, siendo el objetivo principal del plugin.

CAPÍTULO 2: ANÁLISIS Y DISEÑO DEL PLUGIN

Introducción

Un proceso de desarrollo de software tiene como propósito la producción eficaz y eficiente de un producto de software que reúna los requisitos del cliente. Es por ello que para lograr un mejor entendimiento de las características del sistema en el presente capítulo, se explican los principales conceptos y definiciones el modelo de dominio. A partir de esta representación se enfatiza en una propuesta de solución para el desarrollo del problema científico de la investigación. Luego se pretende describir los requerimientos funcionales garantizando la plena interpretación de la generación de los artefactos implícitos en cada una de las fases de desarrollo. Además se evidencian los diagramas de clases e interacción identificando los casos de usos arquitectónicamente significativos.

2.1 Modelo de Dominio

El modelo de dominio es una representación visual de los conceptos u objetos del mundo real significativos para un problema o área de interés. Se representa en UML con un diagrama de clases en el que se muestran conceptos u objetos del dominio del problema, asociaciones entre las clases conceptuales y atributos de estas. En la fase de inicio se determinó que los procesos del negocio no están claramente definidos, por tanto se decide representar los conceptos que definen la situación real del sistema mediante un Modelo de Dominio y de esta forma utilizar un vocabulario común que ayude a usuarios, clientes, desarrolladores e interesados a entender el contexto en que se ubica el sistema, logrando una captura correcta de los requisitos.

Diagrama conceptual del dominio.

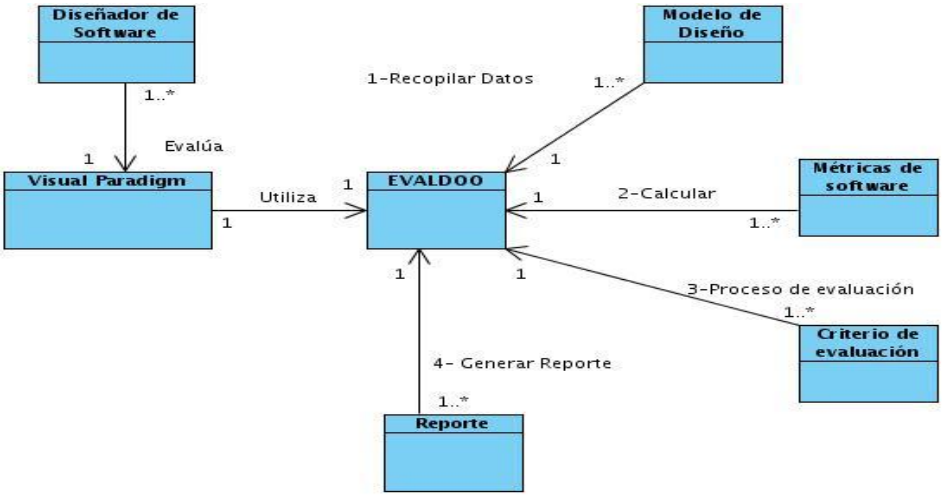


Figura 4: Modelo de Dominio.

Definición de Conceptos del Modelo de Dominio

Elementos	Definiciones
Diseñador de software:	Persona facultada para utilizar el plugin asistido a la herramienta “Visual Paradigm”.
Visual Paradigm:	Herramienta CASE que da soporte al modelado visual con UML 2.1.
EVALDOO:	Plugin integrado a la herramienta “Visual Paradigm”, la cual contiene funcionalidades que permiten a partir de un diagrama del diseño OO evaluar métricas de calidad.
Modelo de Diseño:	Representa los diagramas de clases a los cuales se le realizarán las evaluaciones de las métricas.
Métricas de software:	Métricas a ser utilizadas en el plugin para efectuar la evaluación.
Criterio de evaluación:	Representa el indicador para cada métrica por el cual será regida la evaluación.
Reporte:	Archivo generado por el plugin, el cual una vez que es evaluado el diseño se genera para su comprensión, ya que está conformado por información relevante respecto a la solución.

Tabla 1: Definición de los principales conceptos del Modelo de Dominio.

2.2 Propuesta de Solución

Se propone realizar un plugin para la herramienta Visual Paradigm, la cual estará orientada a la evaluación de los modelos de diseño Orientados a Objeto pasando por tres procesos esenciales para su cumplimiento.

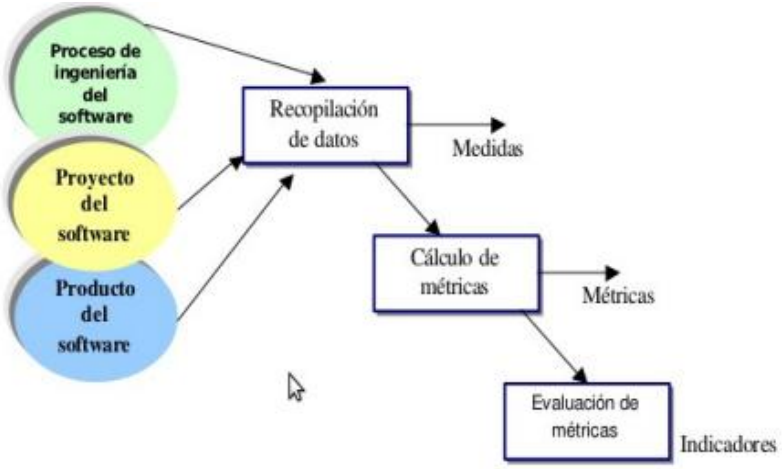


Figura 5: Proceso de evaluación del diseño.

Dichos procesos transcurren a partir del proceso de ingeniería del software, en el cual se obtienen todos los artefactos necesarios para su comprensión. Luego el producto es sometido a una evaluación a través de métricas definidas para garantizar la correcta utilización de los elementos del diseño. Este proceso comienza con la recopilación de datos necesarios que se correspondan con las medidas que establece la métrica seleccionada por la persona encargada de realizar la evaluación del diseño, tales como acceso a atributos, métodos o clases. Luego de obtener estos valores se procede al cálculo de dichas métricas con la información recopilada. Posteriormente se evaluará cada indicador de las métricas dando una determinación de su cumplimiento, arrojando a una solución factible y óptima según las expectativas de un buen o mal diseño a través de un reporte. Como se muestra en la siguiente figura:

Figura 6: Reporte de evaluación.

2.3 Descripción de los Algoritmos para el Cálculo de las Métricas

El proceso de cálculo de las métricas se realiza a partir de los algoritmos definidos y previamente documentados por los autores vinculados la investigación de dichas y otras métricas, de las cuales fueron seleccionados algunos algoritmos para el desarrollo del plugin.
A continuación se explican dichos algoritmos partiendo de las definiciones previamente investigadas.

1) **Acoplamiento entre objetos (*Coupling Between Objects* - CBO) [Chidamber y Kemerer, 1994]**

Esta métrica representa el número de clases a las cuales una clase está relacionada, sin tener con ella relaciones de herencia.

Fórmula: $CBOi = Cl$

Donde:

CRSH: Representa cada una de las clases que presentan relaciones de dependencia, excluyendo la relación de herencia.

TC: Número total de clases.

2) **Proporción de atributos heredados (*Attribute Inheritance Factor* - AIF) [Abreu y Melo, 1996]**

Esta métrica constituye la proporción entre el número de atributos heredados y el número total de atributos.

Fórmula: $AIF = \frac{\sum_{i=1}^{TC} A}{\sum_{i=1}^{TC} A}$

Donde: $A_a = A_d + A_i$

A_a*:* Número de atributos disponibles.

A_d(Número de atributos definidos.

A_i*:* Número de atributos heredados.

TC: Número total de clases.

3) **Métodos ponderados por clase (*Weighted Methods per Class* - WMC) [Chidamber y Kemerer, 1994]**

Esta métrica representa la sumatoria de las complejidades de cada método de una clase. Atendiendo a que en el diseño no es posible medir la complejidad de un método, se estima conveniente asignar complejidad 1 a cada uno de los mismos, transformando la métrica en un simple contador de métodos.

Fórmula: $WMC = \sum_{i=}^{n}$

Donde:

Número de métodos para esa clase.

4) Proporción de métodos heredados (*Method Inheritance Factor* - MIF) [Abreu y Melo, 1996]

Esta métrica constituye la proporción entre la suma de todos los métodos heredados en todas las clases y el número total de métodos.

Formula:

$$MIF = \frac{\sum_{i=1}^{TC} M}{\sum_{i=1}^{TC} M}$$

Donde:

$$M_a(C_i) = M_d(C_i) + M_i$$

M_a(Número de métodos disponibles.

M_d(Número de métodos definidos.

M_i(Número de métodos heredados.

TC Número total de clases.

5) Número total de clases en el diseño (*Design size of classes* - DSC) [Bansiya y Davis]

Esta métrica constituye la sumatoria de las clases presentes en el diseño.

Formula:

$$DSC = \sum_{i=1}^{TC}$$

Donde:

CD Representa cada una de las clases presentes en el diseño.

TC Representa el número total de clases.

6) **Número de métodos polimórficos (*Number of polymorphic methods* - NPM) [Bansiya y Davis]**

Esta métrica representa el número total de métodos polimórficos presentes en las clases.

Formula:

$$NPM = \sum_{i=1}^{TC}$$

Donde:

MP Representa cada uno de los métodos polimórficos presentes en cada una de las clases.

7) **Tamaño de clase (*Class Size* - TC) [Lorenz y Kidd]**

Esta métrica es la suma del número total de operaciones (tanto operadores heredadas como privadas de la instancia) que están encapsuladas dentro de la clase y el número de atributos (tanto heredados como privados de la instancia) que están encapsulados en la clase.

Formula:

$$TC = \sum_{i=1}^{TC} OPH + \sum_{i=1}^{TC}$$

Donde:

OHP Representa cada uno de los métodos heredados y privados dentro de la clase.

AHP Representa cada uno de los atributos heredados y privados dentro de la clase.

TC Representa el número total de clases.

8) **Acoplamiento de abstracción de datos (*Data abstraction coupling* - DAC) [Li y Henry, 1993]**

Esta métrica es el número de atributos que tienen como tipo otra clase.

Formula:

$$DAC = \sum_{i=1}^{TC}$$

Donde:

ATipC Representa cada uno de los atributos que tienen como tipo de dato otra clase.

2.4 Especificación de los Requisitos del sistema

A partir de la descripción de las clases más importantes dentro del contexto del sistema representadas en el Modelo de Dominio se realizó el levantamiento de requisitos. Este flujo de trabajo de requerimientos ayuda a establecer y mantener el acuerdo con los clientes ó los interesados en la aplicación. Proporciona a los desarrolladores del sistema una mejor comprensión de los requisitos y define las fronteras del software. Establece una base para planificar el contenido técnico de las

iteraciones, además de definir una interfaz para el usuario enfocado en las necesidades y metas planteadas.

A continuación se hace referencia a los requisitos funcionales y no funcionales que se encuentran plasmados en el documento de Especificación de requisitos del plugin.

2.4.1 Requisitos Funcionales

Los requisitos funcionales (RF) son capacidades o condiciones que el sistema debe cumplir. Definen las funciones que el sistema será capaz de realizar. Expresan la naturaleza del funcionamiento del sistema, cómo interactúa el sistema con su entorno y cuáles van a ser su estado y funcionamiento. [9]

Los requisitos funcionales deben:

- Estar redactados de tal forma que sean comprensibles para usuarios sin conocimientos técnicos avanzados (de Informática).
- Especificar el comportamiento externo del sistema y evitar, en la medida de lo posible, establecer características de su diseño.
- Priorizarse (al menos, se ha de distinguir entre requisitos obligatorios y requisitos deseables).

Para el desarrollo de este sistema se han definido los siguientes requisitos funcionales de acuerdo a las características que presenta el plugin, teniendo en cuenta las condiciones que debe cumplir agrupados por patrones de casos de uso.

- **RF1** Seleccionar lenguaje de programación
  - **RF1.1** Seleccionar lenguaje de programación
- **RF2** Evaluar Diseño:
  - **RF2.1:** Validar existencia de diagramas de clases OO
  - **RF2.2:** Seleccionar Métricas a utilizar
  - **RF2.3:** Buscar datos según criterios de evaluación
  - **RF2.4:** Calcular métrica
  - **RF2.5:** Evaluar métrica
  - **RF2.6:** Evaluar Diseño
  - **RF2.7:** Generar Reporte
  - **RF2.8:** Mostrar reporte
- **RF3** Exportar reporte
  - **RF3.1:** Exportar reporte
- **RF4** Definir criterios:
  - **RF4.1**: Definir métrica

RF4.2: Guardar criterio

RF4.3: Guardar Archivo de Consulta

- **RF5** Intercambiar criterio

  RF5.1: Intercambiar criterio de evaluación

2.4.2 Requisitos No Funcionales

Los requisitos no funcionales (RNF) son propiedades o cualidades que el producto debe tener. Estas propiedades o cualidades se refieren a las características que hacen al producto atractivo, usable, rápido o confiable. Por lo general los requisitos no funcionales son fundamentales en el éxito del producto; normalmente están vinculados a los requisitos funcionales, es decir, una vez que se conoce lo que el sistema debe hacer se puede determinar cómo ha de comportarse, qué cualidades o propiedades debe tener. [9]

Los requisitos no funcionales:

Han de especificarse cuantitativamente, siempre que sea posible para que se pueda verificar su cumplimiento.

Se identificaron los siguientes requisitos no funcionales asumiendo los de la herramienta Visual Paradigm*,* ya que el plugi*n* por sí solo no cumple funcionalidad*:*

- **RNF 1:** Requisitos de Software

  RNF1.1 Se utilizará para la utilización del plugin la herramienta Visual Paradigm.

- **RNF 2:** Requisitos de Hardware

  Se requiere para la completa ejecución del plugin:

  RNF2.1 procesador a 3, 0 GHz o superior.

  RNF2.2 Mínimo 512 MB de RAM (Random Access Memory, por sus siglas en inglés), pero se recomienda 1,0 GB.

  RNF2.3 Un mínimo de 500 MB de espacio en disco.

- **RNF 3:** Restricciones del diseño y la implementación

  RNF3.1 Se hace uso de la herramienta Visual Paradigm en su versión 8.0 e IDE NetBeans 7.1.

  El lenguaje de programación que será usado para la implementación es Java siguiendo el paradigma de la Programación Orientada a Objeto.

- **RNF 4:** Requisitos de Usabilidad

RNF4.1 Facilidad de uso por parte de los usuarios: La interfaz debe ser lo más descriptiva posible, permitiendo que las operaciones a realizar por los usuarios estén bien descritas, de manera que se puedan entender claramente.

RNF4.2 La aplicación debe permitir el uso de teclas rápidas.

- **RNF 5:** Requisitos de Soporte

  RNF5.1 Proveer un manual de usuario.

- **RNF 6:** Requisitos de Portabilidad

  RNF6.1 El plugin una vez integrado a la herramienta visual Paradigm, podrá ser instalado y disponer del mismo en diferentes sistemas operativos por ser Visual Paradigm una herramienta multiplataforma.

2.5 Modelo de Casos de Usos del Sistema

Luego de realizar el proceso de levantamiento de requisitos del sistema expuestos anteriormente se realizó el modelo de casos de uso del sistema. El cual permite a partir de la captura de los requisitos describirlos en el proceso de desarrollo de software de forma legible y precisa, además de simplificar la construcción de los modelos de objetos y servir de base para las pruebas del sistema. Representa las relaciones existentes entre actores y casos de uso, donde la confección de los CU representa los RF definidos anteriormente a través de la aplicación de patrones de CU. En el diagrama de CU correspondiente al plugin a desarrollar se pone de manifiesto el patrón Extensión <extend>, el cual alega su utilización a partir de que un caso de uso pueda incluir el comportamiento de otro caso de uso bajo determinadas condiciones, siempre con la incertidumbre de que dicha acción puede o no ocurrir, el patrón Inclusión <include>, el mismo permite dividir las redundancias y reutilizar los CU, siendo de carácter obligatorio su ejecución y el patrón Múltiples Actores, el cual delega que un actor puede desarrollar varias funciones en un sistema.

A continuación se exponen los conceptos principales en correspondencia a este modelo, a partir de la descripción y representación de los mismos.

2.5.1 Actores del Sistema

Un actor es un usuario del sistema. Esto incluye usuarios humanos y otros sistemas computacionales, el mismo usa un Caso de Uso para ejecutar una porción de trabajo de valor para el negocio. El conjunto de casos de uso al que un actor tiene acceso define rol en el sistema y el alcance de su acción. Además son generalmente responsables de realizar actividades que serán automatizadas en el futuro sistema. [10].

El plugin cuenta con tres actores esenciales de los cuales se realiza una breve descripción a continuación:

Actor	Objetivo
Diseñador	Persona autorizada a realizar las opciones de plugin, tanto evaluar el diseño como definir criterio de métricas.
Evaluador	Es el que se encarga de realizar la evaluación en el sistema.
Especialista de métricas	Es el que se encarga de definir los criterios de cada métrica.

Tabla 2: Descripción de los Actores del Sistema.

2.5.2 Diagrama de Casos de Uso del Sistema

Los diagramas de casos de uso documentan el comportamiento de un sistema desde el punto de vista del usuario. Por lo tanto determinan los requisitos funcionales del sistema, es decir, representan las funciones que un sistema puede ejecutar. [11]

Las características principales de los casos de uso delegan que están expresados desde el punto de vista del actor, se documentan con texto informal describiendo la interacción, se describen tanto lo que hace el actor como lo que hace el sistema cuando interactúa con él, aunque el énfasis está puesto en la interacción, son iniciados por un único actor y están acotados al uso de una funcionalidad claramente definida del sistema.

Su ventaja principal es la facilidad para interpretarlos, lo que hace que sean especialmente útiles en la comunicación con el cliente, y de vital importancia para la representación de las funciones de un sistema.

A partir de los requisitos funcionales identificados anteriormente se muestra a continuación el diagrama de casos de uso de la aplicación:

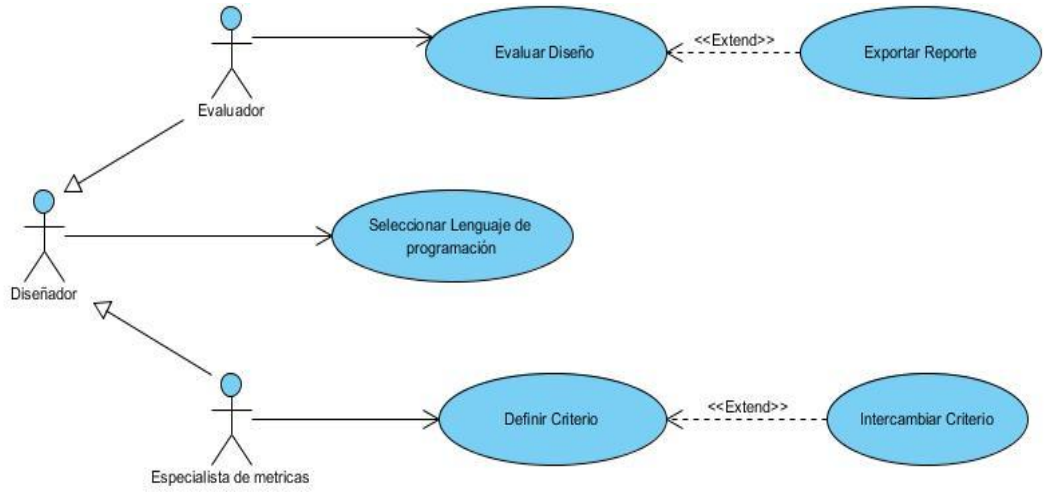


Figura 7: Diagrama de Casos de Uso del Sistema.

Descripción de un caso de uso significativo

Objetivo	Evaluar el diseño Orientado a Objeto de la herramienta Visual Paradigm.	
Actores	Evaluador: Inicia el caso de uso con la selección de las métricas a utilizar para la evaluación de los diagramas de clases en la herramienta Visual Paradigm.	
Resumen	El caso de uso comienza con la selección de las métricas a utilizar, una vez que se selecciona la opción “Evaluar”, el sistema procede a evaluar cada uno de los diagramas de clases existentes en la herramienta.	
Complejidad	Alta	
Prioridad	Crítico	
Precondiciones	Debe existir al menos un diagrama de clase en el Visual Paradigm. Al menos debe estar seleccionada una métrica.	
Postcondiciones	Se evalúa el diseño.	
Flujo de eventos		
Flujo básico “Evaluar Diseño”		
	Actor	**Sistema**
1	Da clic derecho en el context (sección de área de trabajo en la herramienta).	Muestra un menú con las opciones: - Evaluar Diseño - Definir Criterio.

2	Selecciona la opción que desee ejecutar.	
3		En dependencia de la operación indicada por el Evaluador realiza las siguientes acciones: • Si el evaluador desea seleccionar lenguaje de programación, se ejecuta el CU "Seleccionar lenguaje de programación".(Ver descripción de CU) • Si el evaluador desea evaluar diseño, se ejecuta la sección "Evaluar Diseño" que a su vez permite las siguientes opciones: - Validar existencia de diagramas de clases OO, ver Sección "Validar existencia de diagramas de clases OO". - Seleccionar métricas a utilizar, ver Sección: "Seleccionar métricas a utilizar". - Buscar datos según criterios de evaluación, ver Sección: "Buscar datos según criterios de evaluación". - Calcular métrica, ver Sección: "Calcular métrica". - Evaluar métrica, ver Sección: "Evaluar métrica". - Generar reporte de métricas, ver Sección "Generar Reporte". - Mostrar reporte de métricas, ver Sección "Mostrar Reporte". • Si el evaluador desea Definir el criterio de alguna métrica, se ejecuta el CU

		"Definir Criterio".(Ver descripción de CU) - Si el evaluador desea Exportar el reporte de métricas, se ejecuta el CU "Exportar Reporte".(Ver descripción de CU) Si selecciona la opción "Cancelar" ver paso 1 del Flujo Alterno.
Flujos alternos		
1ª Cancelar evaluación		
	Actor	**Sistema**
1		Cancela la petición de evaluación de diseño y definición de criterio y cierra el menú.
Sección 1: "Validar existencia de diagramas de clases OO"		
Flujo básico "Validar existencia de diagramas de clases OO"		
	Actor	**Sistema**
1	Da clic en el context para desplegar e plugin.	Despliega el plugin mostrando un menú con las opciones que presenta el mismo, mientras exista algún diagrama de clases OO. En caso que no exista al menos 1 diagrama, ver paso 1 del Flujo alterno.
Flujos alternos		
1ª No existe ningún diagrama de clases OO.		
	Actor	**Sistema**
1		Aparecen las opciones del plugin deshabilitadas.
Sección 2: "Seleccionar Métricas a utilizar"		
Flujo básico "Seleccionar métricas a utilizar"		
	Actor	**Sistema**
1		Muestra una interfaz con las métricas disponibles.
2	Selecciona las métricas que va a utilizar para llevar a cabo la evaluación del diseño.	Captura cada una de las opciones que selecciona. De no seleccionar al menos 1 métrica ver paso

		1 del Flujo Alterno.
Flujos alternos		
1ª No selecciona ninguna métrica.		
	Actor	**Sistema**
1		Muestra un mensaje informando que debe seleccionar al menos una métrica.
Sección 3: "Buscar datos según criterios de evaluación"		
Flujo básico "Buscar datos según criterios de evaluación"		
	Actor	**Sistema**
1		Busca los datos referentes a las métricas en los diagramas de clases existentes.
Sección 4: "Calcular métrica".		
Flujo básico "Calcular métrica".		
	Actor	**Sistema**
1		Calcula matemáticamente los valores de cada métrica según lo recopilado en los diagramas a partir de los criterios obtenidos.
Sección 5: "Evaluar métrica".		
Flujo básico "Evaluar métrica".		
	Actor	**Sistema**
1		Evalúa los valores obtenidos en el cálculo de cada métrica a partir de los intervalos definidos para cada una de ellas en función de si se encuentra en el rango de evaluación o no.
Sección 6: "Evaluar Diseño"		
Flujo básico "Evaluar Diseño"		
	Actor	**Sistema**
1	Selecciona la opción "Evaluar".	
2		Procede a realizar la evaluación de los diagramas existentes.
3		Genera un reporte con los datos de la evaluación de cada una de las métricas en el diseño.

		Termina el caso de uso.
Sección 7 "Generar Reporte"		
Flujo básico "Generar Reporte"		
	Actor	**Sistema**
1		Genera el reporte con los datos recolectados de la evaluación para cada métrica.
Sección 8 "Mostrar Reporte"		
Flujo básico "Mostrar Reporte"		
	Actor	**Sistema**
		Muestra el reporte generado en una interfaz con todos los datos del sistema.
Relaciones		
	CU Extendidos	Exportar Reporte en el CU Evaluar Diseño.
Requisitos funcionales	RF2.1 RF2.2 RF2.3 RF2.4 RF2.5 RF2.6 RF2.7 RF2.8	

Tabla 3: Descripción del caso de uso significativo Evaluar Diseño.

Para obtener la descripción detallada de la realización de los casos de uso ver el expediente digital Especificación de casos de uso.

2.6. Modelo de Diseño

El diseño constituye una representación ingenieril significativa en la construcción del software. Es un proceso que tributa en demasía al proceso de desarrollo de software para transformar los requisitos de los usuarios en un producto de software finalizado. Durante esta fase, cuando la arquitectura es estable y los requisitos están bien entendidos, el centro de atención se desplaza a la implementación siendo de vital importancia la especificación de la estructura del sistema, la cual es definida a través del modelo de diseño.

El modelo de diseño es un modelo de objetos que describe la realización de casos de uso, y sirve como una abstracción del modelo de aplicación y su código fuente. Se utiliza como parte esencial para las actividades en ejecución y prueba, que se basa en el análisis y los requisitos de la arquitectura del sistema. Representa los componentes de aplicación, determina su colocación adecuada y el uso dentro de la arquitectura en general del sistema.

Diagrama de clases del diseño

Un diagrama de clases constituye el pilar básico del modelado con UML, siendo utilizado tanto para mostrar lo que el sistema puede hacer (análisis), como para mostrar cómo puede ser construido (diseño). Al igual que los demás diagramas, puede contener notas y restricciones. Además representa las clases que serán utilizadas dentro del sistema y las relaciones que existen entre ellas. Sirve para visualizar las relaciones entre las clases que involucran el sistema, las cuales pueden ser asociativas, de herencia, de uso y de convencimiento. Un diagrama de clases está compuesto por los siguientes elementos: Clase: atributos, métodos y visibilidad. Relaciones: Herencia, Composición, Agregación, Asociación y Uso. [12]

Partiendo de la descripción detallada de los casos de uso del sistema, se modelaron los diagramas de clases del diseño.

A continuación se muestra el diagrama de clases diseño del caso de uno de los casos de usos más significativos del sistema: Evaluar Diseño:

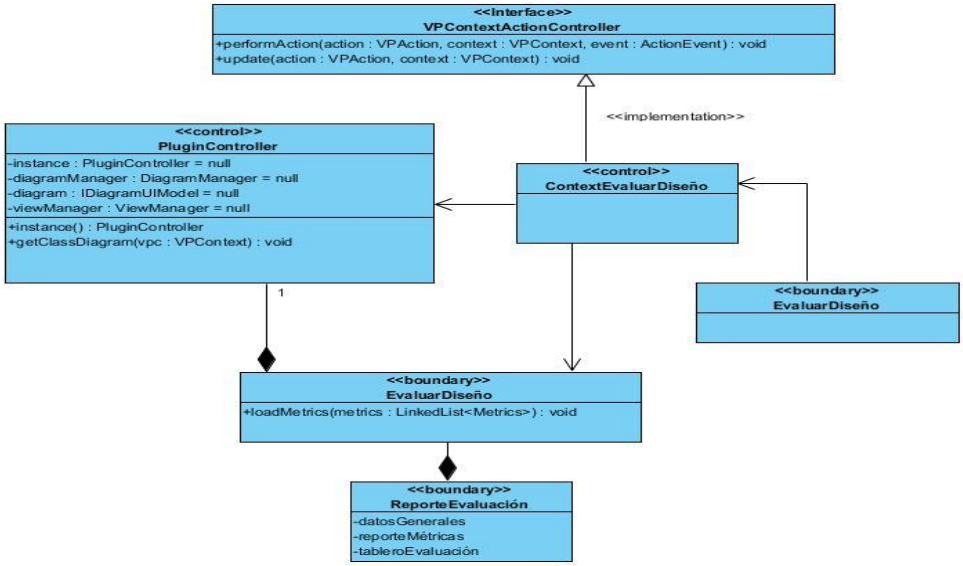


Figura 8: Diagrama de Clases del Diseño - CU Evaluar Diseño.

Descripción de las clases relevantes del diseño:

Clase	**Descripción**
VPContextActionController	Es la clase que permite actualizar cada acción realizada a través del contexto, así como ejecutar las acciones en el propio contexto.
ContextEvaluarDiseño	Es la clase que permite la captura en el contexto de la acción de realizar la evaluación, así como la ejecución del formulario.

PluginController	Es la clase que mantiene el control de todas las acciones que se realizan.
EvaluarDiseño	Formulario que permite la manipulación de cada uno de los elementos de las métricas a la hora de realizar la evaluación.
ReporteEvaluacion	Formulario que contiene y manipula los datos del reporte a generar una vez que se realiza la evaluación.

Tabla 4: Descripción de las clases relevantes del diseño.

Patrones utilizados

Diseñar lo que se pretende construir, teniendo en cuenta los elementos de calidad, así como el propio método de diseño constituyen las bases fundamentales si se quiere lograr una alta calidad en el producto de software final. De manera que el diseño es una etapa que no deberá saltarse u omitirse en un proceso de construcción de software, se apoya de patrones para hacer más eficiente su aplicación. Los patrones no constituyen una teoría y mucho menos un lenguaje de programación, sencillamente son la experiencia obtenida en el proceso de desarrollo de software, probada y que funciona realmente. Estos tienen una fuerte relación con los atributos de calidad, ya que es a partir de ellos que se tipifican y deben su propuesta de solución.

Para que una solución sea considerada, un patrón debe poseer ciertas características, entre las cuales se encuentra la comprobación de su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reusable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

La arquitectura del plugin está basada en paquetes integrados a la herramienta Visual Paradigm, la cual provee el mecanismo de integración del plugin a través de un openapi.jar, librería que está ubicada en el paquete de instalación de la herramienta, dentro del directorio “lib/openapi.jar, implementando la arquitectura MVC (Modelo Vista Controlador).

Estos paquetes en el IDE Netbeans están estructurados de la siguiente manera: el primero, a la configuración del *plugin,* conformado por un conjunto de clases que permiten cargar y configurar el *plugin*, las mismas son Plugin.xml y Plugin.java. El segundo contiene las acciones del *plugin* definidas a nivel de herramientas y de contexto. Dichas clases en dependencia de la acción, implementan interfaces asociadas a las acciones VPActionController y VPContextAction. Ambas clases definen el performAction el cual permite ejecutar acciones referentes al evento onClick de las mismas. El tercer paquete contiene los diálogos que se desea mostrar, lo cual esto se realiza mediante el método performAction asociado a la clase de acción correspondiente al dialogo.

Muchos son los patrones de diseño utilizados en el proceso de desarrollo del software, entre los que se encuentran los patrones Generales de Asignación de Responsabilidades GRASP, evidenciándose en la implementación el Controlador y Creador descritos a continuación:

- Controlador

El patrón Controlador cumple su función en la clase Plug_inControllers, dicha clase posee todas la acciones fundamentales dentro del plug-in, como la realización de cada una de las evaluaciones y la captura de los elementos del diagrama.

- Creador

El patrón creador se encuentra la clase GenerarReporte (Interfaz) la cual hace instancias de plug_inControllers, así como de la clase Evaluación para acceder a cada una de las evaluaciones realizadas.

Además se utilizan los patrones “La Banda de los cuatro” (*Gang of Four* - GOF). Estos patrones resuelven problemas específicos de diseño, y vuelven al diseño orientado a objetos más flexible, elegante y extremadamente reutilizable. Ayudan a los diseñadores a reutilizar diseños exitosos basando nuevos diseños en experiencia previa.

Para la implementación del plugin se seleccionaron los patrones Singleton e Iterador, con el fin de solucionar y optimizar los posibles problemas presentes en cuanto a la relación con los objetos y demás situaciones solucionables, los cuales serán descritos a continuación.

- Iterator (Iterador)

```
clases = new IClass[classes.size()];
Iterator<IClass> iter = classes.iterator();
while(iter.hasNext()){
  clases[i] = iter.next();
  i++;

}
```

Figura 9: Ejemplo del patrón Iterador en el plugin.

Descripción:

El patrón Iterador, es de tipo comportamiento a nivel de objetos que proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna. Permite realizar recorridos sobre objetos compuestos independientemente de la implementación de estos. Con la aplicación de este patrón se incrementa la flexibilidad, dado que para permitir nuevas formas de recorrer una estructura basta con modificar el iterador en uso, cambiarlo por otro o definir

uno nuevo. Además se facilitan el paralelismo y la concurrencia, puesto que, como cada iterador tiene consciencia de su estado en cada momento, es posible que dos o más iteradores recorran una misma estructura simultánea o solapadamente. [3]

¿Cuándo utilizarlo?

Se debe utilizar cuando se quiere acceder a los elementos de un objeto agregado sin mostrar su representación interna, cuando se quieren permitir recorridos múltiples en objetos agregados y cuando se quiera proporcionar una interfaz uniforme para recorrer diferentes estructuras de agregación.

¿Cómo funciona?

Para la implementación del *plugin* el patrón iterador al igual que el método de fabricación, se utiliza directamente en la implantación, para iterar sobre el proyecto, obteniendo una colección de elementos y captar de esta forma información a través de operadores secuenciales.

- Singleton (Solitario)

```
public static Plug_inControllers instance(
  if (instance==null){
      instance = new Plug_inControllers();
    }
  return instance;
}
```

Figura 10: Ejemplo del Patrón Singleton en el plugin.

Descripción:

Es de tipo creacional, a nivel de objetos. Su propósito es garantizar que una clase sólo tenga una única instancia, proporcionando un punto de acceso global a la misma. El acceso a la “Instancia Única” es controlado. 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). [3]

¿Cuándo utilizarlo?

Se utiliza cuando debe haber únicamente una instancia de una clase y debe ser claro su acceso para los clientes.

¿Cómo funciona?

El patrón Singleton o Solitario es implementado por la clase PluginController.java que permite la creación de objetos a través de instancias, manteniendo la consistencia entre los objetos.

2.8. Diagrama de secuencia

El Diagrama de Secuencia muestra las clases que participan para la realización de un caso de uso siguiendo la notación del Lenguaje de Modelado Unificado (UML). Cada clase participante etiqueta una línea temporal. La comunicación entre las clases participantes para cada responsabilidad individual se muestra utilizando mensajes. Cada mensaje está compuesto por el nombre de un servicio y sus argumentos. [13]

Los diagramas de secuencia tienen dos características que los distinguen de los diagramas de colaboración. En primer lugar, está la línea de vida, la cual es la línea discontinua vertical que representa la existencia de un objeto a lo largo de un período de tiempo. En segundo lugar el foco de control, el cual es un rectángulo delgado y estrecho que representa el período de tiempo durante el cual un objeto ejecuta una acción, bien sea directamente o a través de un procedimiento subordinado.

En el presente diagrama se muestran el flujo de acciones que ocurren en la aplicación al ejecutar la acción “Evaluar Diseño”, para evaluar el diseño dentro del caso de uso del mismo nombre.

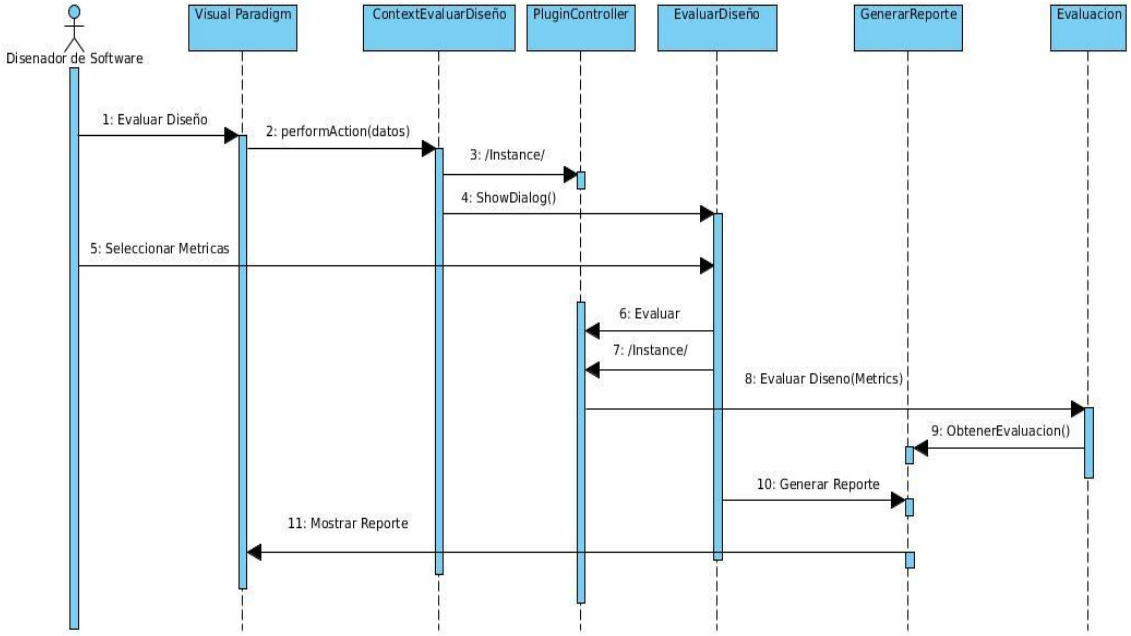


Figura 11: Diagrama de Secuencia - Escenario: Evaluar Diseño.

El proceso de evaluación del diseño Orientado a objeto en la herramienta Visual Paradigm se realiza a partir de que se captura el diagrama de clases. Para esto se crea una instancia del PluginController con la cual se captura la acción en el contexto para la evaluación del diseño. Luego se realiza una instancia de la clase *VPContext* descrita en el openapi.jar, la cual permite obtener el diagrama activo con todos sus componentes mediante el método *getDiagram ()*. Posteriormente se muestra la interfaz con las métricas disponibles y se realiza otra instancia al PluginController permitiendo la ejecución de los métodos pertenecientes a las métricas que fueron seleccionadas por el diseñador de software para

la evaluación. Seguidamente se procede a efectuar las últimas funciones del plugin, en el cual luego de realizar el proceso de evaluación, se obtienen de la clase Evaluación, el listado de todas las evaluaciones realizadas que posteriormente serán mostradas en un reporte.

Conclusiones parciales.

Luego del estudio del presente capítulo se expusieron los conceptos esenciales en el contexto del sistema a través del modelo de dominio, arrojando posteriormente a una propuesta de solución transcurriendo por 3 procesos esenciales tales como “Recopilación de datos, Cálculo de métricas y Evaluación de métricas”. Se realizó el levantamiento de requisitos del sistema concluyendo con la identificación de 14 requisitos funcionales agrupados por patrones de caso de uso dando lugar al diagrama de caso de uso con todas sus relaciones. Luego de un análisis de los requisitos identificados se obtuvo el modelo de diseño como vía principal para visualizar las relaciones entre las clases involucradas en el sistema. Fueron definidos para la implementación del plugin los patrones Generales de Asignación de Responsabilidades (Grasp) y los patrones Gang of Four (GOF). Se realizó la descripción de los algoritmos para el cálculo de cada una de las 8 métricas definidas en el capitulo anterior y los diagramas de secuencia presentes en el sistema.

CAPÍTULO 3: IMPLEMENTACIÓN Y PRUEBAS DEL PLUGIN

Introducción

En el capítulo correspondiente a las disciplinas de implementación y pruebas, se analizarán los artefactos principales como es el caso del Modelo de Implementación poniendo en práctica el diseño de la solución. Se comenzará a implementar el sistema en términos de componentes y se describirán las pruebas a realizar, con el objetivo de comprobar las funcionalidades del plugin en los diferentes escenarios, para de esta forma verificar en todos los casos que los resultados de las pruebas sean los esperados.

3.1 Modelo de Implementación

El Modelo de Implementación hace más entendible el trabajo a los desarrolladores, pues describe cómo los elementos del modelo de diseño, se implementan en términos de componentes, ficheros de código fuente, ejecutables, entre otros. Describe cómo se organizan los componentes de acuerdo a los mecanismos de estructuración y modularización disponibles en el entorno de implementación y lenguajes de implementación empleados, también cómo dependen los componentes unos de otros. El mismo está compuesto por los diagramas activos de despliegue y componentes.

3.1.1 Diagrama de Componente

Un componente es el empaquetamiento físico de los elementos de un modelo, como son las clases en el modelo de diseño. Un diagrama de componente muestra como el sistema está dividido en componentes y las dependencias entre ellos. Provee una vista arquitectónica de alto nivel del sistema, ayudando a los desarrolladores a visualizar el camino de la implementación y permitiendo tomar decisiones respecto a las tareas de implementación. Muestra las opciones de realización incluyendo código fuente, binario y ejecutable. Los componentes representan todos los tipos de elementos software que entran en la fabricación de aplicaciones informáticas, archivos, paquetes, bibliotecas cargadas dinámicamente, etc.

Existen diferentes tipos de componentes, como son:

- **Executable:** Especifica un componente que se puede ejecutar en un nodo.
- **Library:** Especifica una biblioteca de objetos estática o dinámica.
- **Table:** Especifica un componente que representa una tabla de una base de datos.
- **File:** Especifica un componente que representa un documento que contiene código fuente o datos.
- **Document:** Especifica un componente que representa un documento.
- **Folder:** Especifica un paquete que contiene en su interior una serie componentes.

El presente modelo de implementación describe cada uno de los componentes asociados al diseño de clases propuesto para la construcción del plugin para la herramienta de modelado Visual Paradigm, así como la relación de dependencia entre los componentes que la integran, proporcionando una vista de la arquitectura de paquetes del plugin.

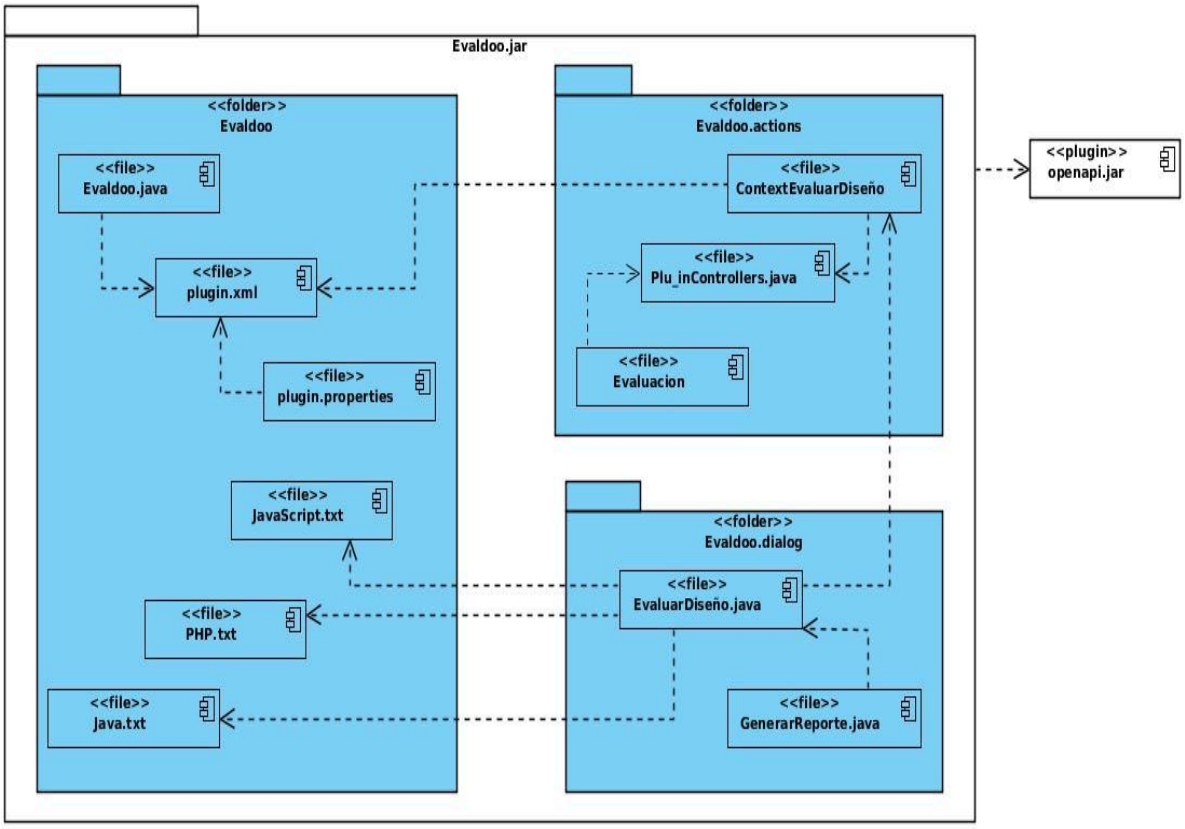


Figura 12: Diagrama de componentes para el CU Evaluar Diseño.

Descripción de los componentes más relevantes.

Nombre del plugin: “Evaldoo” representa la aplicación “Evaluador de diseño Orientado a Objeto”.

Componentes	**Descripción**
Paquete Evaldoo	Paquete que agrupa las clases asociadas a la configuración del plugin: - plugin.xml - Evaldoo.java - plugin.properties

	• JavaScript.txt, • Java.txt • PHP.txt.
Paquete Evaldoo.actions	Paquete que agrupa las clases referentes a las acciones que son accesibles a través contexto (área de diseño del diagrama en VP) o del menú herramientas. • ContextEvaluarDiseño.java • Plug_inController.java • Evaluación.java
Paquete Evaldoo.dialog	Paquete que agrupa los formularios presentes en la aplicación como es el caso de EvaluarDiseño.java y GenerarReporte.java.
Evaldoo.java	Carga y descarga el plugin mediante la clase VPPluginInfo que provee el openapi.jar, capturando la información definida en el archivo plugin.xml.
Plugin.xml	Su función consiste en la configuración del plugin, define el nombre del plugin, descripción, proveedor, librerías a utilizar y las acciones a ejecutar.
Plugin.properties	Define propiedades asociados a los eventos y acciones del plugin.
JavaScript.txt	Archivo que contiene el criterio de cada una de las métricas de forma numérica para el lenguaje javaScript.
PHP.txt	Archivo que contiene el criterio de cada una de las métricas de forma numérica para el lenguaje php.
Java.txt	Archivo que contiene el criterio de cada una de las métricas de forma numérica para el lenguaje java.
PluginController.java	Clase que define las acciones o funcionalidades que permite el plugin.
ContextEvaluarDiseño.java	Clase que permite la captura en el contexto de la acción de realizar la evaluación, así como la ejecución del formulario.
Evaluación.java	Clase que contiene todas las evaluaciones realizadas para

	posteriormente ser mostradas en el reporte.
GenerarReporte.java	Formulario que contiene toda la información necesaria para generar el reporte de la evaluación en la herramienta.

Tabla 5: Descripción de los componentes más relevantes en el CU Evaluar Diseño.

3.2 Ejemplos de códigos en el plugin.

Una vez que es iniciado el plugin se muestran las diferentes funcionalidades para la cual está diseñado, entre las cuales se encuentra Evaluar Diseño. Para el correcto funcionamiento de dicha opción se deben definir con anterioridad las funcionalidades de seleccionar las métricas y el lenguaje a utilizar y una vez que proceda a efectuarse el sistema carga un archivo el cual contiene de acuerdo al lenguaje de programación seleccionado, el criterio de dichas métricas, como se muestra en el código.

Esta funcionalidad recibe como parámetro el archivo que contiene los criterios de cada una de las métricas en el lenguaje estimado, los cuales dichos valores están plasmados en un rango constituido por un valor inicial y un valor final.

```
public void CargarFile(String file) throws FileNotFoundException{
    float [] uno = new float[16];
    String arch = null;
    String [] spli = null;
    File fil = new File(file);
    FileReader fr = new FileReader(fil);
    BufferedReader br = new BufferedReader(fr);
    if (fil!=null){
    try {
        arch = br.readLine();
        spli = arch.split(" ");
        br.close();
    } catch (IOException ex) {
        Logger.getLogger(DefinirCriterio.class.getName()).log(Level.SEVERE, null, ex);
    }
    }
    for(int i = 0; i<spli.length;i++){
     uno[i]= Float.parseFloat(spli[i]);
    }
    // Valores Iniciales
    cboiS.setValue(uno[0]);
    aifiS.setValue(uno[1]);
    wmciS.setValue(uno[2]);
    daciS.setValue(uno[3]);
    mifiS.setValue(uno[4]);
    dsciS.setValue(uno[5]);
    npmiS.setValue(uno[6]);
    tciS.setValue(uno[7]);
```

Figura 13: Ejemplo de código para Cargar Archivo con los criterios.

Una vez que se cargan los valores de las métricas se ejecuta el método Evaluar, el cual hace una comparación con las métricas que fueron seleccionadas y las que están disponibles en el listado inicial,

que al encontrar alguna similitud en las mismas, realiza una llamada a los métodos correspondientes de dichas métricas.

```
public void Evaluar(){
String[] metricas = metricasS.getItems();
plc.setMetricas(metricas);
    for(int i = 0; i<metricas.length;i++){
     if(metricas[i].equals("(CBO): Acoplamiento Entre Objetos")){
        plc.CBO();
     }
     if(metricas[i].equals("(AIF): Proporción de Atributos Heredados")){
        plc.AIF();
     }
     if(metricas[i].equals("(WMC): Métodos Ponderados por Clase")){
        plc.WMC();
     }
     if(metricas[i].equals("(DAC): Acoplamiento de Abstracción de Datos")){
        plc.DAC();
     }
     if(metricas[i].equals("(MIF): Proporción de Métodos Heredados")){
        plc.MIF();
     }
     if(metricas[i].equals("(DSC): Número Total de Clases en el Diseño")){
        plc.DSC();
     }
     if(metricas[i].equals("(NPM): Número de Métodos Polimórficos")){
        plc.NPM();
     }
     if(metricas[i].equals("(TC): Tamaño de la Clase")){
        plc.TC();
     }
    }
```

Figura 14: Ejemplo de código del método Evaluar.

Se ejecuta el código correspondiente a las métricas seleccionadas para la evaluación En el ejemplo que se presenta se corresponde a la métrica CBO o acoplamiento entre objetos, la cual establece la sumatoria de las clases a la cual una clase esta relacionada sin tener con ella relaciones de herencia. Al obtenerse el resultado para dicha métrica, se realiza una comparación con sus criterios con el fin de arrojar a un resultado, el cual de acuerdo al código emite el criterio de un diseño mas o menos usable de acuerdo a las expectativas de que el mismo posea un nivel alto o bajo de acoplamiento.

```
public void CBO() {
int cont = 0;
float[] crit = getCriterios();
try {
    for (int i = 0; i < arrClases.length; i++) {
        Iterator asoc = arrClases[i].fromRelationshipIterator();
        while (asoc.hasNext()) {
            IRelationship asore = (IRelationship) asoc.next();
            if (asore.getModelType().compareTo("Dependency")==0) {
                cont++;
            }
        }
        if (cont < crit[0] || cont > crit[8]) {
            Evaluacion eva = new Evaluacion("CBO", arrClases[i].getName(), cont, EvaluacionTipo.Alerta, "No existe un Bajo Acoplam
            evaluaciones.add(eva);
        } else if (cont == crit[0] || cont == crit[8]) {
            Evaluacion eva = new Evaluacion("CBO", arrClases[i].getName(), cont, EvaluacionTipo.Regular, "Regular");
            evaluaciones.add(eva);
        } else {
            Evaluacion eva = new Evaluacion("CBO", arrClases[i].getName(), cont, EvaluacionTipo.Bien, "Bien");
            evaluaciones.add(eva);
        }
        cont = 0;
    }
} catch (Exception e) {
    Util.dumpException(e);
}

}
```

Figura 15: Ejemplo de código para una métrica.

3.3 Pruebas de Software

Para determinar la calidad de un producto de software se deben efectuar medidas y desarrollar actividades que permitan comprobar el grado de cumplimiento de las especificaciones iniciales del sistema, lo cual es aquí donde las pruebas de software desempeñan un papel fundamental. Estas se definen como una actividad en la cual un sistema o uno de sus componentes se ejecutan en circunstancias previamente especificadas, los resultados se observan y registran con el fin de realizar una evaluación de algún aspecto.

Objetivos de las pruebas:

La prueba de software es una etapa imprescindible durante todo el proceso de desarrollo, pues una vez que se genera código fuente, el software debe ser probado para descubrir y corregir el máximo de errores posibles antes de su entrega al cliente. Un buen caso de prueba es aquel que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces, de ahí que se hace necesario el cumplimiento de sus objetivos, expuestos a continuación, ya que este proceso posibilita no solo la detección de errores en el código sino la documentación necesaria que facilita que este código pueda ser reutilizado:

- Planificar las pruebas necesarias en cada iteración.
- Diseñar e implementar las pruebas creando los casos de prueba que especifican qué probar, creando los procedimientos de prueba que especifican cómo realizar las pruebas y creando, si es posible, componentes de prueba ejecutables para automatizar las pruebas.
- Realizar las diferentes pruebas y manejar los resultados de cada prueba sistemáticamente.[1]

Las construcciones en las que se detectan defectos son probadas de nuevo y posiblemente devueltas a otro flujo de trabajo, como diseño o implementación, de forma que los defectos importantes puedan ser arreglados.

En la evaluación dinámica del sistema se aplicó el nivel de trabajo Pruebas de Desarrollador, el cual está diseñado e implementado por el equipo de desarrollo. Cada nivel de prueba engloba una técnica de prueba específica según los atributos de calidad que se deseen verificar con las pruebas al software. La técnica de prueba seleccionada es la Prueba funcional, la cual tiene como principal objetivo asegurar el trabajo apropiado de los requisitos funcionales, incluyendo la navegación, entrada de datos, procesamiento y obtención de resultados. A través del método de Caja Negra, también conocido como Pruebas de Comportamiento, se ejecutará cada caso de uso usando datos válidos e inválidos, para verificar que los resultados esperados ocurran cuando se usen datos válidos y se desplieguen los mensajes apropiados de error.

Para confeccionar los casos de prueba de Caja Negra se utilizó el criterio de la técnica de Partición de Equivalencia, donde se divide el campo de entrada en clases de datos que tienden a ejercitar determinadas funciones del software. En esencia, esta técnica intenta dividir el dominio de entrada de un programa en un número finito de variables de equivalencia. De tal modo que se pueda asumir razonablemente que una prueba realizada con un valor representativo de cada variable es equivalente a una prueba realizada con cualquier otro valor de dicha variable. Las variables de equivalencia representan un conjunto de estados válidos y no válidos para las condiciones de entrada de un programa. Se definen dos tipos de variables de equivalencia: las válidas, que representan entradas válidas al programa, y las no válidas, que representan valores de entrada erróneos, aunque pueden existir valores no relevantes a los que no sea necesario proporcionar un valor real.

Método de Caja Negra

Se centra en los requisitos funcionales, permitiendo al ingeniero del software derivar conjuntos de condiciones de entrada que ejerciten completamente todos los requisitos funcionales de un programa. Estas pruebas se llevan a cabo sobre la interfaz del software y son completamente indiferentes al

comportamiento interno y la estructura del programa. Los casos de prueba de Caja Negra pretenden demostrar que las funciones del software son operativas, que la entrada se acepta de forma adecuada, que se produce una salida correcta, y que se mantiene la integridad de la información externa.

3.4. Casos de prueba

Un caso de prueba se diseña según las funcionalidades descritas en los casos de uso. El propósito que se persigue con este artefacto es lograr una comprensión común de las condiciones específicas que la solución debe cumplir. Se parte de la descripción de los casos de uso del sistema, como apoyo para las revisiones. Cada planilla de caso de prueba recoge la especificación de un caso de uso, dividido en secciones y escenarios, detallando las funcionalidades descritas en él y describiendo cada variable que recoge el caso de uso en cuestión. Además quedan plasmados las revisiones realizadas al caso de prueba; así como un registro de todo aquello que no corresponde a la calidad del software. Se efectuaron los casos de pruebas a los 6 casos de uso del sistema, plasmándose en el expediente de proyecto en la fase de Pruebas. (Ver planilla Diseño de Casos de Prueba)
A continuación se presenta la tabla de secciones a probar en el caso de uso Evaluar Diseño:

Nombre de sección	**Escenarios**	**Descripción**	**Flujo donde empieza**
SC1: Validar existencia de diagramas de clases OO	EC 1.1 Validar existencia de diagramas de clases OO	El sistema verifica que existe al menos un diagrama de clases en la herramienta para poder habilitar las opciones que permite el plugin, de lo contrario aparecen deshabilitadas y muestra un mensaje indicando que el diagrama está vacío.	Flujo Principal
SC2: Seleccionar Métricas a utilizar	EC 2.1 Seleccionar Métricas a utilizar	El diseñador de software decide seleccionar las métricas a utilizar en el plugin y el sistema verifica que se seleccione al menos 1, de lo contrario envía un mensaje indicando que no se ha seleccionado alguna métrica.	Flujo Principal

	EC 2.2 Mostrar mensaje de error.	El sistema muestra un mensaje informando que no se ha seleccionado métrica alguna.	Flujo Alterno
SC3: Buscar datos según criterios de evaluación	EC 3.1 Buscar datos según criterios de evaluación	El sistema busca los datos referentes a las métricas seleccionadas en los diagramas de clases existentes.	Flujo Principal
SC4 : Calcular métrica	EC 4.1 Calcular métrica	El sistema calcula los valores para cada métrica seleccionada.	Flujo Principal
SC5: Evaluar métrica	EC 5.1 Evaluar métrica	El sistema evalúa cada métrica de acuerdo a los indicadores establecidos para cada métrica.	Flujo Principal
SC6: Evaluar Diseño	EC 6.1 Evaluar Diseño	El sistema emite un resultado a partir de los datos obtenidos en los diagramas de clases y el cálculo realizado para cada métrica.	Flujo Principal
SC7: Generar Reporte	EC 7.1 Generar Reporte	El sistema genera un reporte constituido por las evaluaciones referentes a las métricas que fueron utilizadas en la evaluación.	Flujo Principal
SC8: Mostrar reporte	EC 8.1 Mostrar reporte	El sistema muestra reporte que fue generado a través de una interfaz.	Flujo Principal

Tabla 6: Secciones en el CU Evaluar Diseño.

Partiendo de esta descripción, se detallan las variables que se encuentran en las interfaces asociadas al caso de uso.

No	Nombre del campo	Clasificación	Valor Nulo	Descripción
V1	Métricas	String	No	Cadena de caracteres alfabéticos con el cual se obtienen los nombres de las métricas a seleccionar.
V2	Lenguaje	String	No	Cadena de caracteres alfabéticos con el cual se obtienen los nombres

				de los lenguajes de programación disponibles.

Tabla 7: Variables definidas a partir de la interfaz Evaluar Diseño.

Esta descripción posibilitó que se realizara una matriz de datos, donde se evaluó y probó la validez de cada uno de los datos introducidos en el sistema, específicamente en la sección que se estuvo probando. Utilizando un juego de datos válidos e inválidos se identificó el empleo de la técnica de partición de equivalencia, descritos a continuación:

Juego de datos: V: indica válido, I: indica inválido, NA: que no es necesario proporcionar un valor del dato en este caso, ya que es irrelevante.

Matriz de Datos:

Id EC	Escenario	Variables		Respuesta del sistema	Resultado de prueba	Flujo Central
		V1	V2			
1.1	Validar existencia de diagramas de clases OO.	NA	NA	Valida la existencia de diagramas de clases en la herramienta para poder habilitar las opciones que permite el plugin.	Satisfactoria	1 El diseñador de software da clic en el contexto. 2 Si existe algún diagrama de clases, el sistema se habilita y activa las opciones que contiene.
2.1	Seleccionar Métricas a utilizar.	V	NA	Captura las métricas que fueron seleccionadas por el diseñador de software.	Satisfactoria	1 El diseñador de software selecciona las métricas que va a utilizar en la

						evaluación. 2 El sistema captura las métricas seleccionadas.
2.2	Mostrar mensaje de error.	V	NA	Muestra un mensaje de error informando que debe seleccionar al menos una métrica.	Satisfactoria	1. El sistema muestra un mensaje informando que debe seleccionar al menos una métrica.
3.1	Seleccionar lenguaje de programación.	NA	V	Captura el lenguaje que fue seleccionado para posteriormente escoger los criterios de acuerdo al mismo.	Satisfactoria	1 El diseñador de software selecciona el lenguaje que quiere para la evaluación. 2 El sistema captura el lenguaje seleccionado.
3.2	Mostrar mensaje de error.	NA	V	Muestra un mensaje de error informando que debe seleccionar el lenguaje.	Satisfactoria	1. El sistema muestra un mensaje de error informando que debe seleccionar el lenguaje de programación a utilizar.
4.1	Buscar datos según criterios de	NA	NA	Obtiene los datos de acuerdo a los criterios de las	Satisfactoria	1 El sistema busca los valores en los diagramas de

	evaluación.			métricas en cada uno de los diagramas de clases.			clases de acuerdo a la descripción de la métrica.
5.1	Calcular métrica.	NA	NA	Calcula cada métrica a partir de los indicadores definidos en las mismas.	Satisfactoria	1	El sistema calcula las métricas que fueron seleccionadas a partir de los indicadores definidos en las mismas y los valores obtenidos en los diagramas.
6.1	Evaluar métrica.	NA	NA	Evalúa las métricas a partir de los indicadores calculados para las métricas y los criterios definidos para ellas.	Satisfactoria	1	El sistema evalúa cada métrica seleccionada a partir de los indicadores definidos como criterio de evaluación en las mismas y el cálculo efectuado de las métricas.

7.1	Evaluar Diseño	V	V	El sistema evalúa finalmente el diseño, emitiendo un criterio de un buen o mal diseño a partir de los criterios de las métricas y los datos obtenidos en los distintos diagramas.	Satisfactoria	1	El sistema evalúa el diseño una vez que son evaluadas las métricas seleccionadas, emitiendo un resultado sobre la base de un buen o mal diseño.

Tabla 8: Matriz de datos para el CU Evaluar Diseño.

Una vez que se aplican las pruebas funcionales se arrojan diferentes resultados. La eficiencia del sistema esta dado en los errores que a partir de dichas pruebas se pueden encontrar y a la vez corregir, minimizándole tiempo y costo al producto y obteniendo un software satisfactorio. El plugin desarrollado, después de aplicarle las pruebas arrojó 5 no conformidades significativas, las cuales fueron resueltas, culminando el presente trabajo con resultados satisfactorios.

Conclusiones parciales

Como resultado de este capítulo se obtuvo la implementación del sistema en términos de componentes, proporcionando solución a los requisitos especificados en el capítulo anterior. Además, se estructuraron las clases del diseño en paquetes de componentes mostrando la organización y sus dependencias entre los componentes que conforman el sistema. Una vez concebida la estructura y la implementación del sistema, se realizaron los casos de pruebas para validar la completitud de los requerimientos, obteniéndose resultados satisfactorios.

CONCLUSIONES

1. Luego de un estudio profundo a cerca de las métricas de calidad de software para el diseño orientado a objeto, se expusieron las características de las métricas de calidad, su importancia y se seleccionaron 8 para el desarrollo del plugin.
2. A partir del análisis y diseño realizado en el plugin, se expusieron los conceptos esenciales del sistema los cuales ayudaron a la comprensión del mismo, así como la definición de la estructura principal del plugin, proporcionando la entrada apropiada y el punto de partida para las actividades de implementación.
3. Se realizó la implementación correspondiente obteniendo un plugin con las funcionalidades necesarias para garantizar que se realicen correctamente las evaluaciones de los diseños realizados por los desarrolladores en el centro DATEC a partir de las métricas seleccionadas.
4. Se realizaron las pruebas funcionales al plugin para validar las funcionalidades que fueron implementadas, las cuales demostraron que los indicadores de calidad cumplieron satisfactoriamente con los requisitos propuestos, garantizando su correcto funcionamiento.

RECOMENDACIONES

Luego de haber analizado los resultados del presente trabajo de diploma, surgen algunas ideas que podrían ser incorporadas en un futuro con el objetivo de fortalecer el sistema desarrollado, por lo que se recomienda:

1. Realizar la implementación de nuevas métricas para el diseño OO.
2. Realizar un estudio de métricas de calidad para el diseño convencional e identificar métricas que puedan ser incorporadas en futuras versiones del plugin.
3. Realizar un estudio de métricas de calidad para ser aplicadas en los artefactos generados por el Visual Paradigm en todo el proceso de desarrollo de software.

REFERENCIAS BIBLIOGRÁFICAS

1. **Pressman, Roger.** Ingeniería del Software: Un enfoque práctico. Sexta Edición. México DF: McGraw-Hill, 2006.

2. http://catarina.udlap.mx/u_dl_a/tales/documentos/lis/gonzalez_d_h/capitulo2.pdf. [En línea]

3. Pressman, R. S. Ingeniería del software. Un enfoque práctico. Quinta Edición. S.l.: McGraw-Hill, 2002.

4. **Grasp.** http://germanlescano.wordpress.com/tag/grasp/.[En línea]

5. **Kruchten, Philippe.** El proceso unificado racional: Una introducción. 2004. ISBN 0-321-19770-4.

6. **netbeans.org.** http://netbeans.org/index_es.html. [En línea]

7.**Ortiz,KadirHector.http**://www.eumed.net/libros/2009c/583/Representacion%20del%20Modelo%20de%20Objetos%20de%20Dominio.htm. [En línea]

8. **visual-paradigm.** http://www.visual-paradigm.com/product/vpsuite/. [En línea]

9. **requirements.** http://elvex.ugr.es/idbis/db/docs/design/2-requirements.pdf. [En línea]

10. **casos de uso.** http://www.exa.unicen.edu.ar/catedras/modysim/teoria/casos_de_uso_a.pdf. [En línea]

11. **DiagramaCasosDeUso.** http://www2.uah.es/jcaceres/capsulas/DiagramaCasosDeUso.pdf. [En línea]

12. http://egdamar877.blogspot.com/2009/05/expocicion.html. [En línea]

13. http://wer.inf.puc-rio.br/wer02/zip/Transformacion_Espec%287%29.pdf. [En línea]

BIBLIOGRAFÍA

1. **Pressman, Roger.** Ingeniería del Software: Un enfoque práctico. Sexta Edición. México DF: McGraw-Hill, 2006.

2. http://catarina.udlap.mx/u_dl_a/tales/documentos/lis/gonzalez_d_h/capitulo2.pdf. [En línea]

3. Pressman, R. S. Ingeniería del software. Un enfoque práctico. Quinta Edición. S.l.: McGraw-Hill, 2002.

4. **Grasp.** http://germanlescano.wordpress.com/tag/grasp/.[En línea]

5. **Kruchten, Philippe.** El proceso unificado racional: Una introducción. 2004. ISBN 0-321-19770-4.

6. **netbeans.org.** http://netbeans.org/index_es.html. [En línea]

7.**Ortiz,KadirHector.http**://www.eumed.net/libros/2009c/583/Representacion%20del%20Modelo%20de%20Objetos%20de%20Dominio.htm. [En línea]

8. **visual-paradigm.** http://www.visual-paradigm.com/product/vpsuite/. [En línea]

9. **requirements.** http://elvex.ugr.es/idbis/db/docs/design/2-requirements.pdf. [En línea]

10. **casos de uso.** http://www.exa.unicen.edu.ar/catedras/modysim/teoria/casos_de_uso_a.pdf. [En línea]

11. **Diagrama Casos De Uso.** http://www2.uah.es/jcaceres/capsulas/DiagramaCasosDeUso.pdf. [En línea]

12. http://egdamar877.blogspot.com/2009/05/expocicion.html. [En línea]

13. http://wer.inf.puc-rio.br/wer02/zip/Transformacion_Espec%287%29.pdf. [En línea]

14. Pressman, R. S. Ingeniería del software. Un enfoque práctico. Cuarta Edición. S.l.: McGraw-Hill, 1998.

15. **conceptos-de-programación.** http://www.scribd.com/doc/82893398/Conceptos-de-Programacion. [En línea]

16. **Arregui, Juan José Olmedilla.** *Revisión Sistemática de Métricas de Diseño Orientado a Objetos.* Universidad Politécnica de Madrid, Facultad de Informática: s.n., Septiembre 2005.

17. **Negro, Pablo Ariel.** *Umbrales para métricas Orientadas a Objeto.* Facultad de Tecnología Informática, Universidad Abierta Interamericana. : s.n., agosto 2008.

18. Pedro Jesús Vázquez Escudero, María N. Moreno García, Francisco J. García Peñalvo. *MÉTRICAS ORIENTADAS A OBJETOS, Informe Técnico.* s.l. : DPTOIA-IT, 2001-002.

19. **Pablo Negro, Roxana Giandini, – ASSE (36 JAIIO07).** *Umbrales para Métricas Orientadas A Objetos -.* Universidad Abierta Interamericana, LIFIA, UNLP, Mar del Plata. Argentina: s.n., Agosto 2007.

20. **Michele Lanza, Radu Marinescu.** Object Oriented Metrics in Practice.

21. **Lovelle, Juan Manuel Cueva.** *Calidad del software.* Madrid, España: s.n.

22. http://kasyles.blogspot.com/2008/09/openup-como-alternativa-metodolgica.html. [En línea]

23. http://www.slideshare.net/samith/metodologia-upen-up-3439131. [En línea]

24. http://www.scribd.com/doc/2080534/UML. [En línea]

25. **Ernández, Enrique.** El lenguaje unificado de Modelado.

26. **Métricas e indicadores.** http://www.ciw.cl/recursos/Charla_Metricas_Indicadores.pdf. [En línea]

27. http://www.slideshare.net/americajuarez/metricas01. [En línea]

28. **Sommerville, Ian.** Ingenieria del Software. 7ma Edición. 2005. pág. 687. ISBN: 8478290745.

ANEXOS

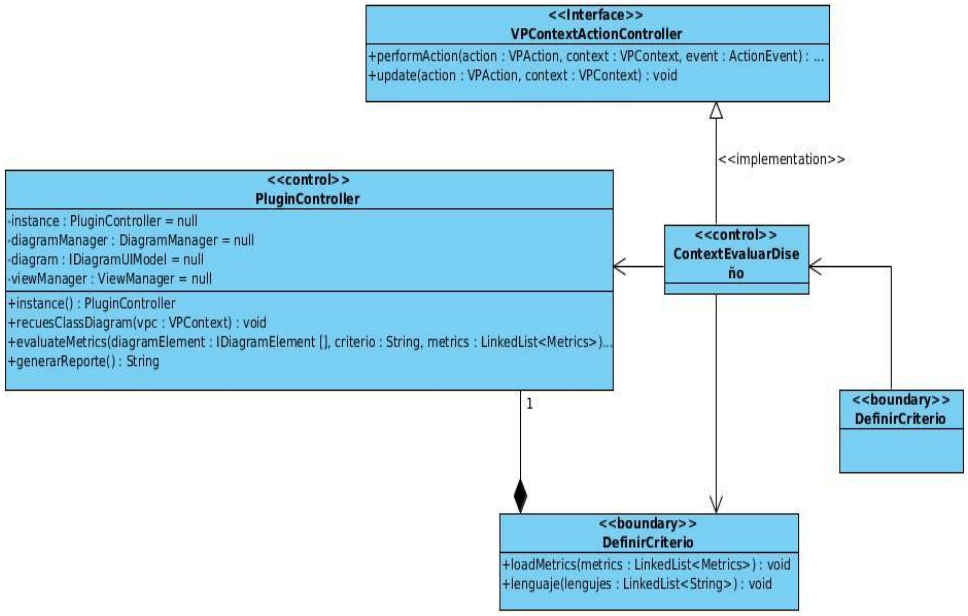


Figura 16: Diagrama de Clases del Diseño - CU Definir Criterio

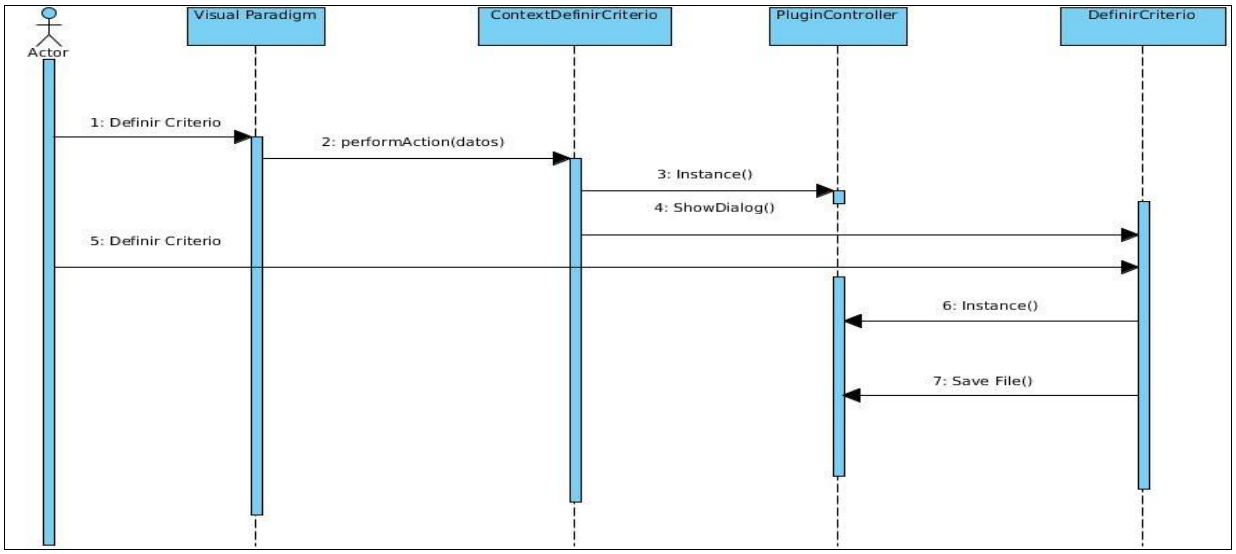


Figura 17: Diagrama de Secuencia Escenario: Definir Criterio

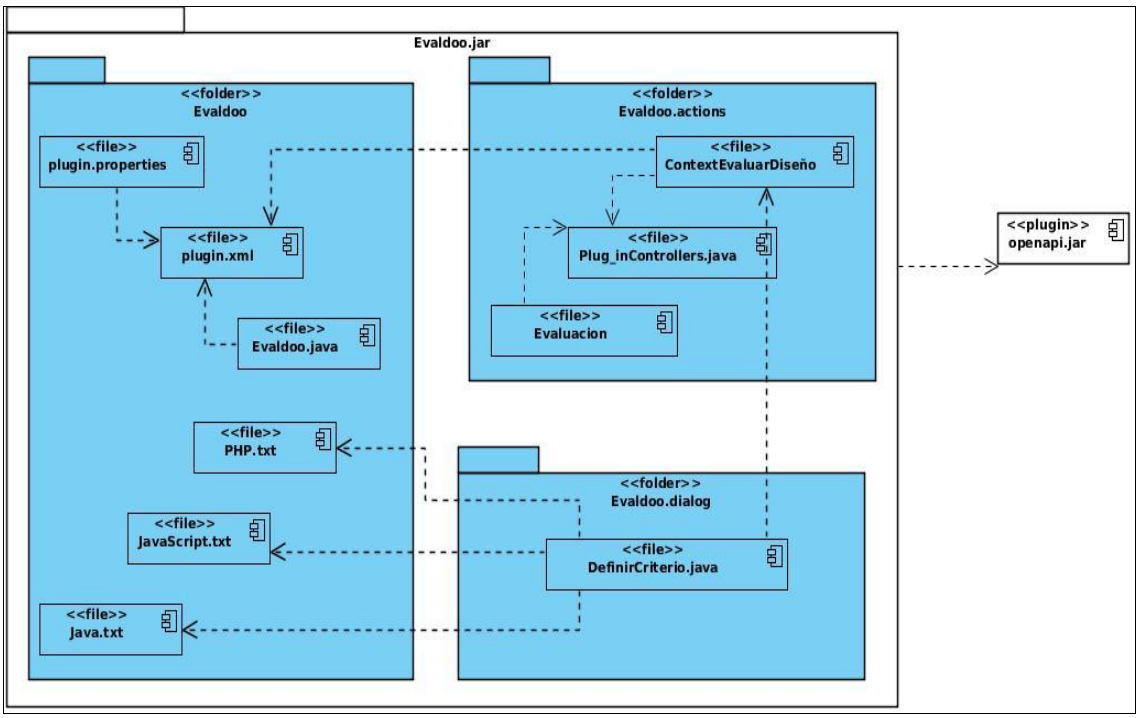


Figura 18: Diagrama de componentes para el CU Definir Criterio.

Evaldoo
Evaluador de Diseño Orientado a Objeto

Figura 19: Logo de la aplicación.

1 GLOSARIO DE TERMINOS

EVALDOO: Evaluador de diseño Orientado a Objeto.

Componente: Un componente es una clase de uso específico, que puede ser configurada o utilizada de forma visual desde el entorno de desarrollo.

Casos de uso: Especificación de las secuencias de acciones, incluyendo secuencias variantes y secuencias de errores, que pueden ser efectuadas por un sistema, subsistema o clase por interacción con autores externos.

Diagrama: Representación gráfica en la que se muestran las relaciones entre las diferentes partes de un conjunto o sistema.

Herramientas: Es un objeto elaborado a fin de facilitar la realización de una tarea mecánica que requiere de una aplicación correcta de energía.

Tecnología: Tecnología es el conjunto de conocimientos técnicos, ordenados científicamente, que permiten construir objetos y máquinas para adaptar el medio y satisfacer las necesidades de las personas.

Metodología: Se refiere a los métodos de investigación que se siguen para alcanzar una gama de objetivos.

Software: Es un término genérico que designa al conjunto de programas de distinto tipo (sistema operativo y aplicaciones diversas) que hacen posible operar con el ordenador.

Stakeholder: Cualquier persona o entidad que es afectada por las actividades de una organización.

UCI: Universidad de las Ciencias Informáticas creada en la ciudad de la Habana, Cuba, año 2002. **UML:** Lenguaje Unificado de Modelado (Unified Modeling Language). Notación gráfica utilizada para describir sistemas de software.

API: Interfaz de Programación de Aplicaciones, por sus siglas en inglés. Conjunto de funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción. **PDF:** Formato de Documento Portátil, por sus siglas en inglés. Es un formato de almacenamiento de documentos.

Requisitos: una condición o capacidad para un usuario para resolver un problema o alcanzar un objetivo.

RF: Requerimientos Funcionales.

RNF: Requerimientos No Funcionales.

DATEC: Centro de Tecnologías de Gestión de Datos

Proceso: se define como un conjunto de actividades enlazadas entre sí que partiendo de una o más entradas las transforman y genera un resultado.

Proyecto: esfuerzo temporal que se lleva a cabo para crear un producto o un servicio con un resultado único.

RAM: Abreviatura del Inglés Random Access Memory. Es la memoria desde donde el procesador recibe las instrucciones y guarda los resultados. Es el área de trabajo para la mayor parte del software de un computador.

Artefactos: productos tangibles del proyecto que son producidos, modificados y usados por las actividades. Pueden ser modelos, elementos dentro del modelo, código fuente y ejecutables.

IDE: Entorno Integrado de Desarrollo, es un programa compuesto por un conjunto de herramientas para un programador. Puede dedicarse exclusivamente para un lenguaje de programación o bien para varios.

1