UNIVERSIDAD DON BOSCO
FACULTAD DE INGENIERÍA

TRABAJO DE GRADUACIÓN
PARA OPTAR AL GRADO DE
INGENIERO EN CIENCIAS DE LA COMPUTACIÓN

PROTOTIPO DE EDITOR E INTÉRPRETE DE LENGUAJE DE MODELADO UNIFICADO (UML) PARA DIAGRAMAS DE CLASES Y GENERACIÓN DE CÓDIGO FUENTE PARA LA ENSEÑANZA Y APRENDIZAJE DE LA PROGRAMACIÓN ORIENTADA A OBJETOS

PRESENTADO POR:

JORGE MIGUEL ERAZO MELARA

ADIS REINALDO MELÉNDEZ ESCOBAR

ASESOR:
ING. JAIME ANTONIO ANAYA HERNÁNDEZ

JULIO 2008
EL SALVADOR, CENTROAMÉRICA

AUTORIDADES DE LA UNIVERSIDAD DON BOSCO

ING. FEDERICO MIGUEL HUGUET RIVERA
RECTOR

LIC. MARIO RAFAEL OLMOS
SECRETARIO GENERAL

ING. ERNESTO GODOFREDO GIRON
DECANO FACULTAD DE INGENIERIA

UNIVERSIDAD DON BOSCO
FACULTAD DE INGENIERÍA

PROTOTIPO DE EDITOR E INTÉRPRETE DE LENGUAJE DE MODELADO UNIFICADO (UML) PARA DIAGRAMAS DE CLASES Y GENERACIÓN DE CÓDIGO FUENTE PARA LA ENSEÑANZA Y APRENDIZAJE DE LA PROGRAMACIÓN ORIENTADA A OBJETOS

ING. JAIME ANTONIO ANAYA
ASESOR

ING. RAUL MARTINEZ RIVAS
LECTOR

Agradecimientos.

- A nuestro Señor Jesucristo por brindarnos la sabiduría, inteligencia y salud para esforzarnos en todo tiempo.
- A mi madre Mirna Cano por brindarme su apoyo incondicional.
- A mi hermano por su apoyo incondicional.
- A mi familia por su apoyo y consejos.
- A nuestro asesor y lector por su orientación, dedicación y paciencia para la realización del trabajo de graduación.
- A mi compañero Adis Meléndez por su esfuerzo.

Jorge Miguel Erazo Melara

- A Dios todo poderoso por darme la fuerza e iluminar mi camino para llevar a cabo el desarrollo de este trabajo de graduación y a la vez por colmar mi vida de bendiciones.
- A mis padres por su apoyo y consejos he llegado a finalizar este trabajo de graduación.
- A mis hermanas por el apoyo que siempre me brindaron.
- A mis familiares por estar pendientes del desarrollo de este trabajo de graduación.
- A mis amigos y amigas por su confianza y lealtad, especialmente a Jorge Erazo.
- A nuestro asesor y lector por el tiempo dedicado en el desarrollo de este trabajo de graduación.
- A todos aquellos que estuvieron al pendiente del desarrollo de este trabajo de graduación.

Adis Reinaldo Meléndez Escobar.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

INDICE DE TABLAS.

Introducción.

La programación orientada a objetos (en adelante la llamaremos POO) ha cambiado la visión para resolver los problemas computacionales y las metodologías de desarrollo de software. POO representa la realidad como objetos de software de diversas clases, que buscan un objetivo en común. Por ello, la correcta implementación de esta visión de programación se vuelve fundamental para los futuros profesionales en computación.

Esta investigación constituirá un esfuerzo para facilitar la enseñanza y aprendizaje de la POO. Por esta razón se realiza un estudio del Lenguaje de Modelado Unificado (UML), como herramienta para comprender este paradigma de programación y el Proceso Unificado (PU) como la metodología de desarrollo de software.

UML permite modelar los problemas a través de diversos diagramas, de los cuales se pondrán mayor énfasis en los diagramas de clases, estos muestran de manera estática las definiciones de clases y los componentes que la conforman, permitiendo que a partir de ellos se generen objetos.

El utilizar PU como metodología de desarrollo, se pretende la creación de una herramienta que permita la construcción de diagramas de clases, de manera fácil y practica, además de ver su implementación en un lenguaje orientado a objetos.

Este documento, proporciona información de las razones por las que se lleva a cabo la investigación y los problemas ha combatir, se establecen los objetivos, alcances, limitantes y delimitantes que conllevará la realización del proyecto. Se incluye una investigación con referencias históricas, conceptuales y referenciales, tanto de UML, PU y POO. Finalmente se establece la metodología de trabajo y quienes serán los beneficiados con el producto final del proyecto.

CAPITULO I
MARCO REFERENCIAL.

1.1 Antecedentes.

POO es una nueva forma de programar que permite representar los problemas de manera más cercana a la realidad, un programa se convierte entonces en un conjunto de objetos de una o varias clases que se relacionan entre si para resolver un problema.

El lenguaje unificado de modelado (UML, por sus siglas en ingles, Unified Modeling Lenguaje) nace con el propósito de crear un estándar de modelos y reglas prácticas, que permitan a todas las partes involucradas en el desarrollo de software la comprensión de lo que se desea hacer.

UML permite comprender un problema desde todas sus perspectivas, por medio de diversos elementos gráficos que se combinan para formar diagramas, para comprenderlos de manera correcta es útil categorizarlos por: estructura, comportamiento e interacción.

Entre los diagramas de estructuras de UML se encuentra el diagrama de clases, es uno de los más usados y mejor conocidos de los diagramas para POO, permiten la representación estática de las relaciones del software y además son la fuente primaria para la generación de código.

Los diagramas de clases incluyen la representación de sus componentes y permiten la definición de objetos, no objetos reales, más bien la descripción particular de cómo seria el objeto, que puede hacer y la manera de relacionarse con otros objetos.

Para sustentar esta descripción de los objetos los diagramas de clases incluyen atributos, operaciones, asociaciones, estereotipos, propiedades y herencia. Esta variedad de elementos proporcionan las herramientas para realizar modelos de software.

PU, como metodología de desarrollo, proporciona una nueva forma de trabajo para la creación de software. Utiliza UML con lenguaje de modelado, se centra en arquitectura y el resultado final es producto de la aplicación iterativa de esta metodología.

Tomando en cuenta lo anterior, una herramienta que integre la funcionalidad para modelar este pensamiento de objetos se vuelve necesaria, tanto para desarrolladores y programadores de software como estudiantes y educadores que deseen implementar esta visión y metodología de programación.

La Universidad Don Bosco, por ser una institución de educación superior, cuenta con una aplicación que utiliza UML, consiste en un editor para diagramas de casos de usos que facilita a los analistas y desarrolladores el proceso de diseño, lográndose los siguientes objetivos:

- Desarrollo de una de una herramienta que utiliza las características intrínsecas de cada estereotipo del diagrama de casos de usos.
- Desarrollo de un editor de diagramas de casos que muestre sus características y utilidad en la etapa de diseño.
- Construcción de un modelo de casos de usos que defina bien la funcionalidad y sus límites que presenta un diagrama completo de la etapa de requerimientos.
- Elaboración de diagramas de casos de usos con las especificaciones adecuadas para descomponer el proyecto en piezas funcionales.
- Generación de vistas en forma de diagramas y reportes.

1.2 Importancia de la investigación

Con la transición de la programación estructurada clásica a la programación orientada a objetos, para programadores o analistas experimentados, este cambio no representa un mayor reto, sin embargo, para futuros profesionales en estas áreas, esta visión de los problemas podría dar como resultado serias deficiencias y en consecuencia profesionales deficientes; esto hace necesario que se oriente una aplicación de software que facilite este proceso de cambio de paradigma, haciéndolo fácil de entender y explicar.

Aunque la universidad Don Bosco tenga un editor que soporta el lenguaje UML, solamente se enfoca a una parte de ella, específicamente a diagrama de casos de usos. Con éste diagrama (aunque sólo es uno de la variedad que posee UML) se puede ver la representación dinámica de los sistemas, es decir, como todos los "componentes" del problema se integran para alcanzar un objetivo común, lo cual puede resultar confuso para un programador principiante, ya que no tiene la visión de que constituye al "componente" del problema.

Con ésta investigación se pretende implementar un prototipo de editor de soporte al lenguaje UML enfocado a diagramas de clases, con el agregado que permitirá, además de ver la representación gráfica del problema, como será la implementación en un lenguaje de programación. Además se sentaran las bases para que en un futuro se desarrolle un editor que de soporte a todo el lenguaje UML.

1.3 Planteamiento del problema

En la actualidad, los profesionales en computación utilizan tecnología orientada a objetos. Hacen uso de diversas metodologías de desarrollo de software, como lo es UP, para realizar el análisis y diseño de sus respectivas aplicaciones buscando el ahorro de tiempo y dinero. Además utilizan lenguajes como UML para facilitar el modelado de problemas.

Es necesario entonces que se busque enseñar a implementar estas herramientas durante el proceso de formación profesional, ya que se llevaría de manera conjunta el aprendizaje del pensamiento orientado a objetos y UML para modelarlas.

Actualmente en nuestro país, ya se toman en cuenta las metodologías que incluyen UML, aunque no se tiene la visión que un buen diseño del sistema facilitaría la implementación de las demás etapas de desarrollo. Es decir, la funcionalidad del software dependería de la experiencia adquirida por el desarrollador y no de la aplicación correcta de las herramientas existentes. Es por ello que el producto de la investigación le brindará una herramienta a la Universidad Don Bosco, para que ésta sea utilizada en el proceso de formación, en el área de programación, análisis y diseño de sistemas e ingeniería de software. Utilizando UP como metodología alterna al desarrollo clásico en "cascada", se busca fomentar una buena cultura de construcción de sistemas.

Por esta razón se hace necesaria la creación de una herramienta propia de la Universidad Don Bosco, la cual esté disposición de la comunidad estudiantil y que le permita a futuros profesionales utilizar las herramientas UML, específicamente diagramas de clases, teniendo entonces una visión clara de lo que se pretende hacer y a partir del diseño generar su correspondiente código fuente. Pero esta herramienta perdería su razón de ser, si la institución no la utiliza paralelamente con las metodologías adecuadas en el proceso de la enseñanza y el aprendizaje.

La universidad Don Bosco introduce programación orientada a objetos en la materia Programación II de la carrera Ingeniería en Ciencias de la Computación, utilizando el lenguaje C++. Además imparte materias electivas de lenguaje entre las cuales están JAVA y Visual Basic .NET como lenguajes orientados a objetos. Entonces seria conveniente una herramienta que permita a los estudiantes, no solo entender el pensamiento a objetos sino además aplicarlo en UML.

Los problemas a abordarse son:

- Falta de cultura de utilización del lenguaje UML para modelar software.
- Falta una herramienta propia de la universidad Don Bosco para la enseñanza de POO.
- La adquisición de las herramientas para modelado poseen altos costos.
- Las herramientas existentes que generan código, tienen alto grado de complejidad para aplicarse en el proceso de enseñanza y aprendizaje.

1.4 Definición del tema

Desarrollo de un prototipo de editor e interprete de Lenguaje de Modelado unificado (UML) para diagramas de clases y generación de código fuente para la enseñanza y aprendizaje de la Programación Orientada a Objetos.

Se describe de la siguiente manera:

Prototipo: por que constituye el primer esfuerzo para la construcción de herramienta CASE propia de la Universidad Don Bosco.

Editor e intérprete: permitiendo crear, modificar, eliminar y relacionar los elementos gráficos que forman parte de UML.

Diagramas de clases: utilizando la simbología definida por UML para representar este tipo de diagramas.

Generación de código fuente: crear una plantilla de código en un lenguaje orientado a objetos.

Enseñanza y aprendizaje: facilidad de uso para estudiantes y profesores de la programación orientada a objetos.

1.5 Justificación del tema

La Universidad Don Bosco, institución encargada de la formación de futuros profesionales en ingeniería en ciencias de la computación, en su plan académico imparte materias en las cuales la programación orientada a objetos es un eje principal; pero debido al extenso contenido, el impartir estas materias se centra en la sintaxis del lenguaje de programación y no en las metodologías para aplicar esos lenguajes. Además la institución no cuenta con herramientas que faciliten el proceso de enseñanza-aprendizaje de la POO junto a técnicas de modelado.

La idea del editor surge por la necesidad que el estudiante no solamente aprenda la manera de resolver problemas utilizando un lenguaje de programación en específico sino que tenga el concepto general de cómo resolver problemas en cualquier lenguaje de programación orientado a objetos.

La investigación se enfoca en el lenguaje UML, como herramienta para el estudiante por su facilidad de implementación, interpretación y uso; permitiendo hacer uso de un lenguaje de modelado común, siendo independiente de la metodología de desarrollo que se utilice. Se centrará específicamente en la utilización de diagramas de clase debido a la facilidad de uso, además de representar la base para la generación de código.

El producto de esta investigación generará código fuente en un lenguaje de programación que soporte programación orientada a objetos a partir de los diagramas de clases. El lenguaje de programación seleccionado para la generación de código fuente es JAVA.

Algunas de las razones por la cual se seleccionó JAVA como uno de los lenguaje para generar la plantilla de código fuente es por ser un lenguaje de alto nivel, de propósito general y poseer las siguientes características[1]:

- Simple y familiar.
- Orientado a objetos.
- Independiente de la plataforma.
- Portable.
- Robusto.
- Seguro.

[1] Para ver detalladamente la razón por la cual se seleccionó JAVA vea anexos (Características de JAVA)

1.6 Objetivos

1.6.1 Objetivo General

Desarrollar un editor e interprete UML para diagramas de clases y generar su correspondiente plantilla código fuente, para facilitar la enseñanza y aprendizaje de Programación Orientada a Objetos

1.6.2 Objetivos Específicos

- Analizar herramientas de código libre que trabajen con lenguaje UML y que generen código fuente.
- Desarrollar una herramienta que contenga las características necesarias para la creación de diagramas de clases.
- Crear la estructura estática de las aplicaciones usando diagramas de clases.
- Generar una plantilla de código ejecutable en JAVA a partir del diseño de la estructura estática de las aplicaciones.
- Documentar la investigación de la herramienta propuesta en el manual del programador a fin de permitir la factibilidad de incorporación de módulos que soporten el resto de diagramas del lenguaje UML.

1.7 Alcances.

1. Permitir la construcción de diagramas estáticos de clases de manera fácil, coherente y amigable por medio del editor e intérprete de UML.

2. Optimizar el tiempo de desarrollo de programas por medio de la generación de código fuente ejecutable en el lenguaje de programación JAVA, a partir de los diagramas de clase.

3. Añadir las características básicas de diseño para diagramas de clases por ser una herramienta para la enseñanza y aprendizaje del lenguaje UML y programación orientada a objetos.

4. Almacenar en estructuras de datos (archivos) los elementos del diagrama, así como sus componentes y características, para su posterior modificación y reutilización.

5. Facilitar la visualización de los diagramas, permitiendo la exportación de diagramas a un archivo de imagen.

1.8 Limitaciones

- Debido a la extensión de la metodología UML, el editor se limitará a la utilización de diagramas de clases, por ser dichos diagramas la fuente primaria de generación de código.

- Creación, modificación y relaciones de clases se realizará directamente en el diagrama, permitiendo que el proceso de diseño sea rápido y fácil.

- El código generado será en lenguaje de programación JAVA, se generará una plantilla de código fuente, que permita su posterior modificación e implementación en sus respectivos compiladores (o IDE).

1.9 Delimitaciones

Se ha delimitado el sistema de la siguiente manera: “Prototipo de editor e interprete UML para diagramas de clases y generación de código en lenguaje JAVA”.

El editor poseerá las características básicas de diseño para diagramas de clases por ser una herramienta para la enseñanza y aprendizaje del método UML y la POO, las cuales serán las siguientes:

- Crear, editar y eliminar componentes clases
- Crear, editar y eliminar componentes notas
- Crear y eliminar componentes relaciones (asociación y herencia)
- Almacenar y abrir archivos propios del editor
- Generar plantilla de código fuente en JAVA del diseño realizado
- Exportar diagramas en formato Portable Network Graphic PNG.

El software estará orientado en primer lugar a la población estudiantil de la Universidad Don Bosco, especialmente aquellos que se encuentran cursando materias de programación orientada a objetos, tales como C++ y JAVA, así también a catedráticos que deseen una herramienta de apoyo para impartir su cátedra.

El lenguaje de programación para el desarrollo del producto es JAVA, el entorno de desarrollo integrado (IDE) es NetBeans 6.1, el sistema operativo de desarrollo será Windows XP Service Pack 2.

1.10 Proyección Social

Este editor beneficiará en primer lugar a los estudiantes, ya que contarán con una herramienta donde puedan aplicar los conocimientos teóricos de la programación orientada a objetos y además de generar código fuente de manera fácil.

Para docentes que imparten materias de POO, el editor se convertiría en un buen recurso didáctico, ya que permitiría la implementación de la metodología UML, facilitando la enseñanza y ahorrando tiempo de desarrollo de programas por su generador de código.

Este editor representaría un primer paso para el desarrollo de una herramienta con mayor funcionalidad, es decir, sentaría las bases para que futuros profesionales desarrollen un software que permita la implementación de toda la metodología UML.

Con base a lo anterior, la Universidad Don Bosco sería beneficiada, ya que poseería una herramienta propia de la institución, dedicada para enseñanza, evitando así la adquisición de un software comercial.

1.11 Marco Referencial

1.11.1 Referencias Históricas

Historia de Programación Orientada a Objetos (POO)

Con el tiempo las computadoras evolucionan y sus lenguajes propios o lenguajes de máquina cambian. Es por ello que surgió la necesidad de crear lenguajes intermedios, que cualquier programador pudiera aprender y que los programas no dependieran de la máquina en la que se iban a ejecutar. Así surgieron varios lenguajes de programación y los primeros compiladores.

Se crearon muchas formar de programar; pero uno de los defectos que la programación enfrentó, es que las variables globales podían ser utilizadas y modificar sus contenidos, desde cualquier parte del programa, por poseer esta indisciplina los programas tendían a ser inmanejables y no llega a haber un acuerdo entre todos los diferentes módulos que conforman los programas.

Este problema fue detectado en la década del 70, se propuso la norma de ocultar información como solución. La idea era encapsular cada una de las variables globales del programa en un módulo junto con sus operaciones asociadas, sólo mediante las cuales se podía tener acceso a estas variables. El resto de los módulos podrían acceder a las variables sólo de forma indirecta mediante las operaciones diseñadas a tal efecto.

El primer lenguaje que introdujo los conceptos de orientación a objetos fué SIMULA 67 creado en Noruega, por un grupo de investigadores, con el fin de realizar simulaciones discretas de sistemas reales. Se basaron en el lenguaje ALGOL 60 y lo extendieron con conceptos de objetos, clases, herencia, el polimorfismo por inclusión y procedimientos virtuales.

Siempre en la década del 70 fue desarrollado el LPOO llamado SMALLTALK en los laboratorios Xerox en Palo Alto, EE.UU. El hecho de ser creado en EE.UU., ayudó a que se introdujera a escala mundial el término de Orientación a Objetos y que cobrara importancia entre los diseñadores de lenguajes de programación.

En 1985, Bjarne Stroustrup extendió el lenguaje de programación C a C++, es decir C con conceptos de clases y objetos, cerca de esta fecha, en 1986, se creó desde sus bases el lenguaje EIFFEL por B. Meyer. Ambos manejan conceptos de objetos y herencia de clases. Estos lenguajes tuvieron importancia entre 1985 y hasta la primera mitad de los 90's.

En 1995 apareció JAVA, uno de los más recientes lenguajes OO, desarrollado por SUN Microsystems, que hereda conceptos de C++, pero los simplifica y evita la herencia múltiple.

Historia de UML

UML fue creado después de muchos intentos por unificar métodos para el proceso de desarrollo de software; pero en 1994 se da el primer acercamiento a UML cuando Grady Booch (precursor de Booch '93) y James Rumbaugh (precursor de OMT) se unen en una empresa común, Rational Rose Corporation y comienzan a unificar sus dos métodos (Booch y OMT).

En 1995 lanzaron la propuesta de su método integrado que fue la versión 0.8 con el nombre Método Unificado (*Unified Method*); fue en ese mismo año Ivar Jacobson se une a Racional para trabajar con Rumbaugh y Booch en el proceso de unificación.

En 1996 Jacobson, Booch y Rumbaugh concluyen su trabajo y lo nombran UML, es entonces donde el OMG decide convocar a otras compañías a participar con sus propuestas para mejorar el enfoque estándar que el UML pretendía. Fue así como en 1997 se creó el estándar de UML 1.0 gracias a las propuestas realizadas por las

empresas que participaron, el cual es adoptado por el OMG y organizaciones afines como el lenguaje de modelado estándar.

Entre las empresas que participaron con las propuestas se pueden mencionar IBM, Racional Software, Oracle, DEC, Hewlett-Pakard, Intellicorp, Microsoft, Texas Instrument entre otras.

Historia de las Herramientas CASE

Debido que el editor e intérprete a desarrollarse en este trabajo puede pertenecer a la categoría de las herramientas CASE[2] es importante conocer un poco acerca de la historia de dichas herramientas; la cual comienza a principios de los 70's con el procesador de palabras que se utilizaba para crear documentos con ello se centró la atención en herramientas de soporte de programas como traductores, compiladores, ensambladores y otras herramientas que auge en ese entonces.

El desarrollo de las herramientas CASE en las décadas de los 80's y 90's fijó su enfoque hacia la solución de problemas en los sistemas de desarrollo por lo que se propusieron elaborar productos acordes a las necesidades:

- Desarrollo Orientado a Objetos: estas herramientas ayudan a crear estructuras de código que fácilmente pueden ser reutilizables con la característica que fácilmente pueden ser utilizados en diferentes lenguajes de programación y plataformas.
- Herramientas de desarrollo visual: estas herramientas permiten construir interfaces de usuario, generar reportes y otras características de los sistemas de forma eficiente.

Según las investigaciones del INEI[3] las líneas de evolución de las herramientas case son:

- Herramientas para sistemas bajo la arquitectura cliente/servidor.

[2] CASE: Computer Aided Software Engeniering – Ingeniería de Software Asistida por Computadora
[3] INEI: Instituto Nacional de Estadística e Informática (Republica del Perú)

- CASE multiplataforma.
- CASE groupware (para trabajo en grupo).
- Desarrollo para sistemas orientados a objetos.
- Otras posibles líneas de evolución serán:
    - La utilización de tecnología multimedia.
    - Incorporación de técnicas de Inteligencia Artificial.
    - Sistemas de realidad virtual.

1.11.2 Referencia Conceptual

Programación Orientada a Objetos (POO)

La POO es una técnica o método para desarrollar programas, los cuales están organizados como colecciones de objetos. Los objetos se crean a partir de las clases. Una clase es un plano en el cual se definen las características (atributos) y comportamientos (operaciones) de los objetos.

Los objetos tienen grados de visibilidad para que sean utilizados por otros objetos para que estos no se referencien directamente. Entre los niveles de visibilidad se tienen:

- (-) Privado: es el más fuerte. Este nivel de visibilidad indica que será accedido desde dentro del objeto.
- (#) Los atributos/operaciones protegidos están visibles para las clases friends y para las clases derivadas de la original.
- (+) Los atributos/operaciones públicos son visibles a otras clases.

UML es un lenguaje de modelado gráfico que se utiliza para especificar, construir, visualizar y documentar los componentes de un sistema de software. Está creado para aplicarse en cualquier medio que necesite capturar requerimientos, decisiones y comportamientos del sistema en estudio.

UML captura las estructuras estáticas y dinámicas por medios de diagramas; donde la estructuras estáticas representan la forma lógica en como está relacionado un sistema con todos sus componentes; mientras que las estructuras dinámicas modelan el comportamiento de los objetos del sistema mientras interactúan entre si.

Una de las características de UML es que no es un método; sino más bien es un lenguaje de modelado, en el cual se ocupa de definir la notación gráfica y su significado, para que éste sea tomado como base para el diseño y no dependa de un proceso. En otras palabras UML es el encargado de orientar los pasos a seguir para elaborar el diseño.

UML ayuda a los desarrolladores de software para comunicarse con un mismo lenguaje de modelado siendo independiente de la metodología que utilicen en el proceso de desarrollo.

Vistas y Diagramas de UML 2.0

Área	Vista	Diagrama	Elementos
Estructural	Vista Estática	Diagrama de Clases	Clase, asociación, generalización, dependencia, realización, interfaz.
	Vista de Casos de Uso	Diagramas de Casos de Uso	Caso de Uso, Actor, asociación, extensión, generalización.
	Vista de Implementación	Diagramas de Componentes	Componente, interfaz, dependencia, realización.
	Vista de Despliegue	Diagramas de Despliegue	Nodo, componente, dependencia, localización.
Dinámica	Vista de Estados de máquina	Diagramas de Estados	Estado, evento, transición, acción.
	Vista de actividad	Diagramas de Actividad	Estado, actividad, transición, determinación, división, unión.
	Vista de interacción	Diagramas de Secuencia	Interacción, objeto, mensaje, activación.
		Diagramas de Comunicación	Colaboración, interacción, rol de colaboración, mensaje.
Administración o Gestión de modelo	Vista de Gestión de modelo	Diagrama de Clases	Paquete, subsistema, modelo.
Extensión de UML	Todas	Todos	Restricción, estereotipo, valores, etiquetados.

Tabla 1.1: Vistas y Diagramas de UML 2.0

DIAGRAMAS DE CLASES

La vista de los diagramas de clases visualiza la forma en como interactúan las clases que conforman el sistema. Un diagrama de clases presenta las clases del sistema con sus relaciones estructurales y de herencia. La creación de las clases incluye definiciones para atributos y operaciones. El modelo de casos de uso aporta información para establecer las clases, objetos, atributos y operaciones.
Cada clase se representa con un rectángulo el cual está divido en tres segmentos, el primero es para el nombre de la clase, el segundo para definir los atributos de la clase y el tercero para definir las operaciones.

Las clases se relacionan con otras entre estas relaciones se tiene:

- Asociación. Es una relación estructural que describe la conexión que existe entre dos objetos, esta conexión puede ser unidireccional o bidireccional; además puede existir una multiplicidad que indica el número de instancias de una clase se relacionan con una instancia de otra clase.
- Dependencia. Indica la relación de uso en la cual una instancia de una clase cumple un papel activo y la otra instancia de la clase un papel pasivo.
- Herencia (Generalización/Especialización). Indica la relación una superclase y una subclase.

Los diagramas de clase comúnmente están formados por:

- Clases
- Atributos
- Operaciones
- Asociaciones
- Generalizaciones
- Agregaciones
- Composiciones
- Notas y restricciones.

1.11.3 Referencia Experimental

En la actualidad existen muchos tipos de editores para el diseño de los diagramas que se pueden modelar en UML, algunos de estos editores son tan complejos como costosos económicamente hablando, otros editores se pueden adquirir de forma gratuita y también es posible adquirir su codificación por ser herramientas open source[4]. Algunas de las herramientas utilizadas para el diseño en UML organizadas por orden del precio[5].

Herramientas UML

Compañía	Software	Versión	Plataforma	Costo
Tigres	ArgoUML	0.24	Java VM	Gratis
BOUML	BOUML	2,5,5	Windows, Unix	Gratis
BlueJ	BlueJ	2,2,0	Java VM	Gratis
Oracle	Jdeveloper	9	Java VM	Gratis
Object Plant	Object Plant	3,32	MacOS	$ 35,00
Visual Object Modelers	Visual UML Standard Edition	3,1	Windows	$ 495,00
Microsoft Visio	Visio 2002 Professional	2002	Windows	$ 499,00
MagicDraw	MagicDraw UML Enterprise	10	Java VM	$ 1.599,00
IBM-Rational	Rose XDE Developer for Visual Studio	2004	Windows	$ 2.495,00
IBM-Rational	Rose XDE Developer for Java	2004	Windows	$ 2.495,00
Microsoft	Visual Studio .NET Enterprise Architect	1	Windows	$ 2.500,00
IBM-Rational	Rose Technical Developer	2004	Windows	$ 5.995,00
Borland	Together Architect	2006	Java VM	$11.500,00

Tabla 1.2: Herramientas UML

[4] Open Source: Es el termino que se le atribuye al software desarrollado y distribuido libremente; el cual está a disposición de su modificación por cualquier persona que necesite editarlo.

[5] La información se encuentra disponible en http://www.objectsbydesign.com/tools/umltools_byPrice.html

Los productos detallados en la tabla anterior se tomaron al azar o por el nombre de la compañía que los desarrolla y por lo que se observa una de las grandes desventajas de algunas de las herramientas son los altos costos de adquisición.

Para Universidad Don Bosco sería demasiado costoso adquirir una de las mejores herramientas, además de ese problema, existe un alto grado de complejidad en el uso de dichas herramientas por lo que sería una mala inversión, ya que los estudiantes no aprovecharían todo el potencial que poseen dichas herramientas.

1.12 Metodología del Proyecto

Para determinar los requerimientos del editor se hará uso de las siguientes metodologías para tratamientos de datos:

Entrevistas: estas se llevaran a cabo en la Universidad Don Bosco, con profesionales en computación para determinar el grado de utilización de las metodologías UML, haciendo énfasis en diagramas de clases. Además de determinar el uso de herramientas comerciales para la enseñanza de POO.

Encuestas: con alumnos de la Universidad Don Bosco, para medir sus conocimientos en POO así como las técnicas de implementación, específicamente UML.

Pruebas de software: a través del uso de software de libre distribución de la misma funcionalidad del que se desea implementar, para tener en claro lo que un editor puede hacer y como generara el código fuente.

En cuanto al diseño y construcción del editor se utilizara:

Lenguaje UML: implementación de las etapas del desarrollo de software utilizado del proceso unificado de desarrollo (UP).

NetBeans 6.1: la herramienta de desarrollo idónea para la realización del editor contando con las librerías adecuadas para la creación de interfases de usuario y la funcionalidad esperada.

Almacenamiento de Datos: a través de la utilización de archivos permitiendo almacenar el diseño y estructura de los componentes.

La metodología para desarrollar el sistema será el Proceso Unificado de Desarrollo de software o simplemente Proceso Unificado (UP). Debido a que provee la disciplina necesaria para la asignación de tareas y responsabilidades a la hora de desarrollar software, que garantice la generación de un software de calidad.

Entre las características de UP:

- Esta dirigido por casos de usos.
- Esta centrado en Arquitectura.
- Es iterativo e incremental.

Por estar dirigido a casos de usos, durante el proceso permitirá determinar lo que debe hacer el sistema, comenzando por especificar los requisitos. Además servirá de guía durante todo el proceso de desarrollo ya que establece las bases para el diseño, implementación y pruebas del sistema.

Por estar centrado en arquitectura, permitirá describir el sistema desde diversas perspectivas, permitiendo así la comprensión general de lo que debe hacer el sistema.

Por ser un proceso iterativo e incremental, permitirá dividir el sistema en mini-sistemas, conocidos como iteraciones, los cuales al irse realizando significaran un avance en el desarrollo del proyecto.

Por todo lo anterior, este proceso permitirá el desarrollo de un prototipo de editor y generador de código de calidad con las funcionalidades necesaria para el usuario.

La investigación se realizara utilizando los cinco flujos fundamentales de trabajo para el Proceso Unificado:

- Requerimientos.
- Análisis.

- Diseño.
- Implementación.
- Pruebas.

Además de las cuatro fases del Proceso Unificado:

- Inicio: se define la descripción general del problema al que se le pretende dar solución, así como de la funcionalidad, tiempo y costos del proyecto.
- Elaboración: consiste en analizar las diversas perspectivas del sistema, además de la planificación de las actividades y recursos necesarios.
- Construcción: se comienza la construcción del producto con base a lo establecido en las fases anteriores.
- Transición: se considera como una etapa de pruebas donde los usuarios verifican el sistema detectando los posibles errores críticos del producto a crear.

Cada fase se descompone en conjunto de iteraciones, se realizaran un máximo de pruebas. La figura 1 muestra como se relaciona las fases de UP con los flujos de trabajos.

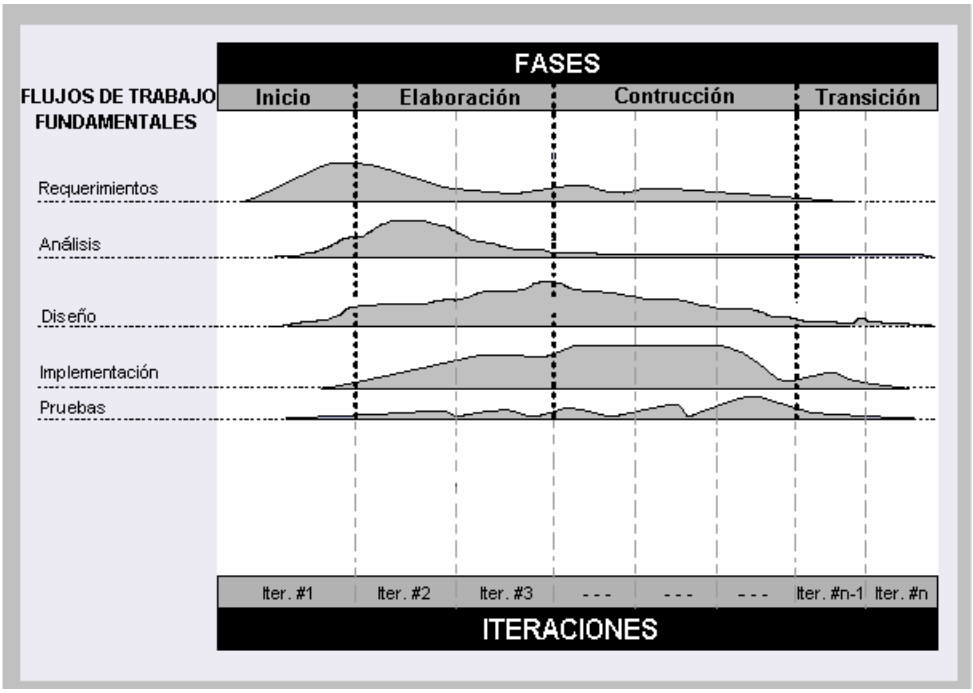


Figura 1.1: Proceso Unificado.[6]

Entre los roles que se asignaran:

- Administrador del sistema: será el encargado de la dirección de la investigación durante el proceso de desarrollo, además de supervisar todas las actividades y toma de decisiones.
- Analista de software: será el encargado de las decisiones técnicas en cuanto a diseño del sistema.
- Programadores: serán los encargados de llevar a cabo la implementación del software y de realizar las pruebas (iteraciones).
- Usuarios: serán los encargados de probar el sistema, verificando el cumplimiento de lo requerido por el sistema.

Con estos roles se busca definir cual será la manera de trabajar al momento de construir la aplicación final.

[6] Figura editada del original Racional Unified Process.

CAPITULO II.

ANALISIS DE RESULTADO Y DIANOSTICO.

2.1 Análisis Estadístico.

El proceso unificado cuenta con una fase llamada incepción, que consiste en realizar la suficiente investigación para determinar la factibilidad del proyecto. La información requerida en esta etapa se obtiene a través de un estudio y análisis estadístico de la población de estudiantes de la Universidad Don Bosco, por ser ellos los beneficiados con el producto final.

2.1.1 Tipo de Investigación.

Los objetivos que se pretenden alcanzar con la investigación son los siguientes:

- Definir la muestra de población.
- Medir el conocimiento de UML y metodologías de desarrollo de software en la población de estudio.
- Conocer el grado de uso de herramientas CASE en la población.
- Verificar la factibilidad del estudio.

Se utiliza la técnica de observación de campo, entrevistas y encuestas.

2.1.2 Población y Muestra.

El universo de población para la investigación son los estudiantes de ingeniería en Ciencias de la Computación de la Universidad Don Bosco, centrados en aquellos que cumplen el requisito de encontrarse cursando arriba del tercer año de su carrera profesional, por ser estos alumnos los que cuentan con mas experiencia académica en programación y desarrollo de software.

Para el cálculo de muestra de población se utilizo la siguiente formula:

$$n = Z_{\alpha}^{2} \frac{N \cdot p \cdot q}{i^{2}(N-1) + Z_{\alpha}^{2} \cdot p \cdot q}$$

Figura 2.1 Ecuación de cálculo de población.

Donde:

n Tamaño de muestra encuestada.

N Tamaño de la población, número total de estudiantes arriba del tercer año de ingeniería. (400 alumnos)

Z Valor correspondiente a la distribución de Gauss 1,96.

p Factor de éxito esperado del parámetro a evaluar. (50%)

q Factor de fracaso. (50%)

i Error que se prevé cometer. (10%)

$$n = \frac{1.96^2 * 400 * 0.5 * 0.5}{0.1^2(400-1) + 1.96^2 * 0.5 * 0.5}$$

Figura 2.2 Valores utilizados en ecuación.

Con base a lo anterior, la población muestra es de **n = 77** alumnos de la ingeniería en ciencias de la computación.

2.1.3 Técnicas y Herramientas de Investigación

Las técnicas de investigación utilizadas para el desarrollo de la investigación serán:

1. Técnica documental: con la que se pretende obtener todo el respaldo teórico y documental sobre UML, UP e información técnica para la implementación del software. Como fuente primaria están libros, recursos en línea, libros electrónicos y así como documentación directa de los creadores de UML.
2. Técnica de campo: contacto directo con estudiantes y profesores, por medio de encuestas y entrevistas.

2.1.4 Presentación y Análisis de Resultados

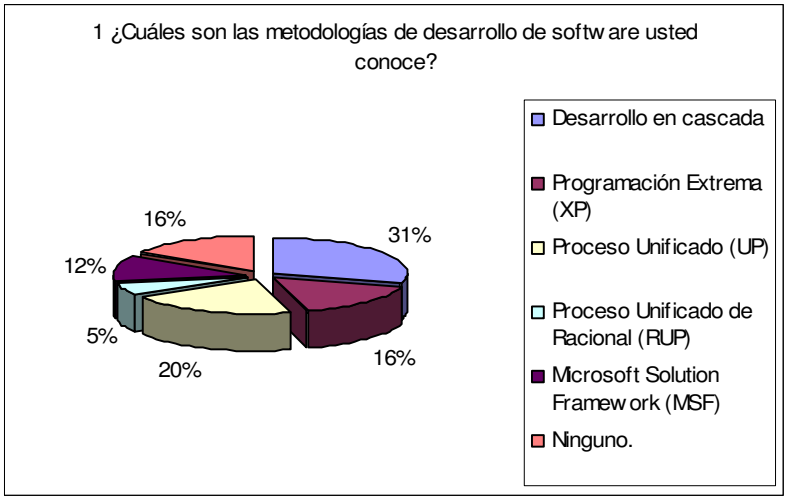
Objetivo General.

Medir el grado de conocimiento y aplicación de los estudiantes de ingeniería en Ciencias de la Computación de las diversas metodologías de desarrollo de software orientados a objetos y lenguaje de modelado (UML).

Resultados de las encuestas.

Pregunta 1.

Objetivo: Identificar cuales metodologías de desarrollo de software son conocidas por la población universitaria.

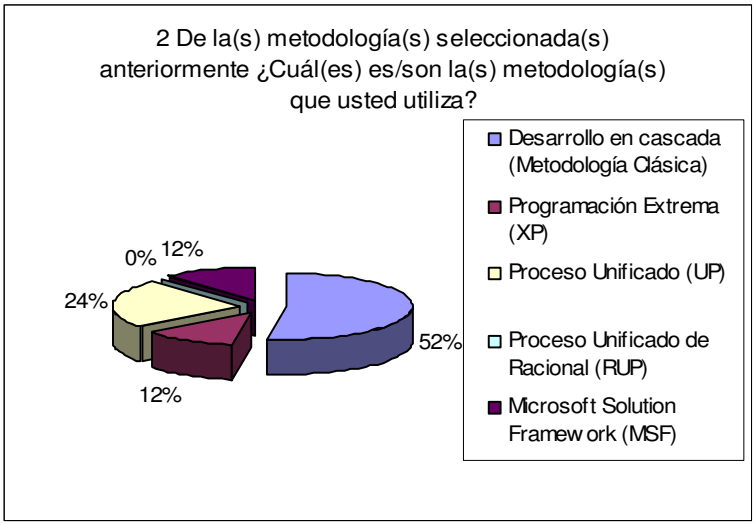


Conclusiones.

- Es notorio el conocimiento de la metodología en cascada por ser utilizada en la enseñanza del desarrollo de software por parte de la Universidad Don Bosco.
- Aunque el desarrollo en cascada es predominante, el proceso unificado UP es conocido por buena parte de la población, a pesar de ser enseñado como metodología alternativa.

Pregunta 2:

Objetivo: Conocer cuales son las metodologías de desarrollo que la población aplica.

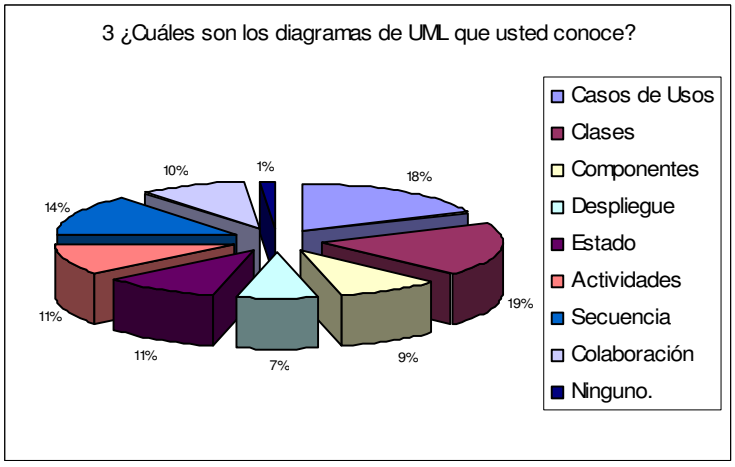


Conclusiones.

- La metodología en cascada predomina por ser aplicada durante el proceso de enseñanza.
- En segundo lugar se ubica el proceso unificado, lo cual indica que los estudiantes al conocer de metodologías de desarrollo de software optan como alternativa a UP.

Pregunta 3.

Objetivo: medir el conocimiento del lenguaje UML a través de identificar sus diagramas.

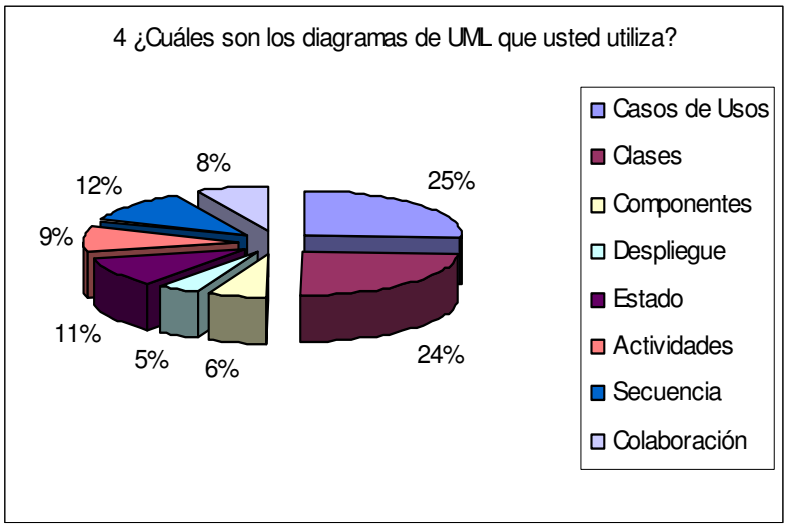


Conclusiones.

- La población estudiantil conoce el lenguaje UML, gran parte de sus diagramas son identificados.
- Los diagramas de casos de uso y clases son los mas conocidos entre la población.

Pregunta 4.

Objetivo: Identificar si los diagramas UML son aplicados por la población.

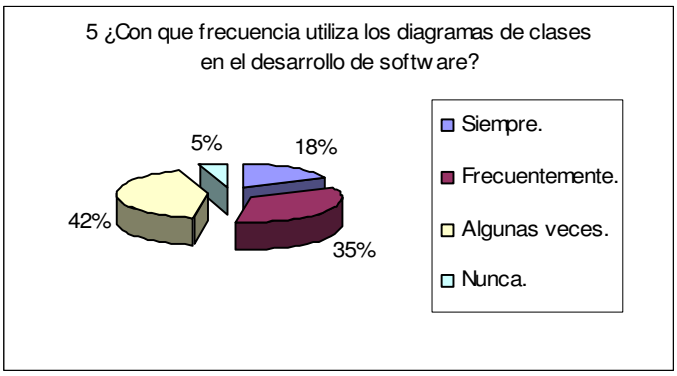


Conclusiones:

- Los diagramas UML son en gran manera aplicados por la población. Los diagramas de casos de usos y clases predominan con respecto al resto, esto indica que los estudiantes poseen una base conceptual para modelar requerimientos y representar sistemas de manera estática.

Pregunta 5.

Objetivo: medir la frecuencia con que se utilizan los diagramas UML.

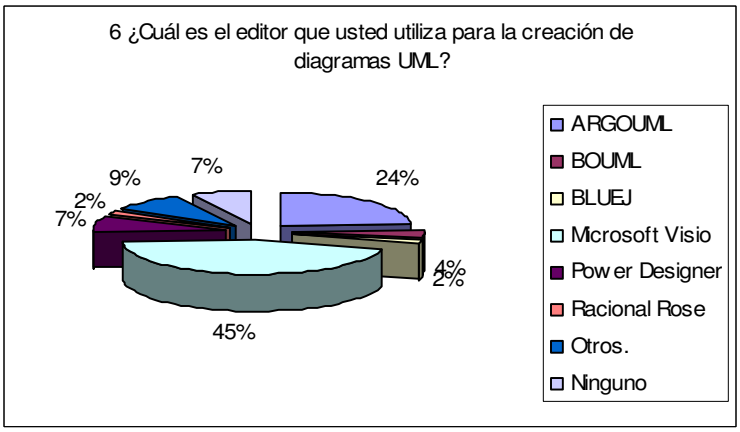


Conclusión.

- La utilización de los diagramas UML se están convirtiendo en una práctica común para los estudiantes en el proceso de desarrollo de software.

Pregunta 6.

Objetivo: medir el grado de utilización de software para modelado de sistemas que utilizan lenguaje UML.



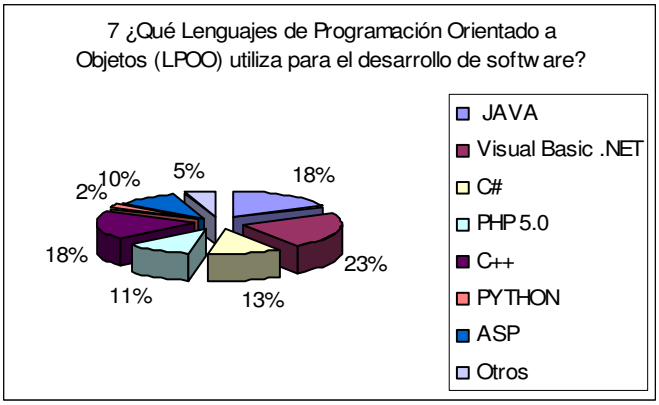
Conclusiones

- Los estudiantes utilizan herramientas de software para modelar sus sistemas, sin embargo hay una gran tendencia ha utilizar software pagado. Lo que indica

que hay desconocimiento de herramientas de software libre que soportan el lenguaje UML.

Pregunta 7.

Objetivo: conocer los lenguajes de programación que los estudiantes utilizan para el desarrollo de sistemas.



Conclusiones.

- Los estudiante optan por paquetes comerciales como los es Visual Studio.Net (Basic.Net, ASP, C#) dejando a un lado a lenguajes de software libre como java, php y pitón.

2.2 Herramientas CASE.

El propósito de analizar herramientas CASE es clarificar que funciones o características se pretenden implementar en la aplicación a realizar, que elementos estas herramientas incluyen para facilitar el desarrollo de software. Se busca herramientas que soportan la notación UML, específicamente el área de diagramas de clases y generadores de código, por ser similares a lo que se pretende construir.

Estas herramientas CASE fueron seleccionada con base a las encuesta, por ser herramientas conocidas por los estudiantes de la Universidad Don Bosco. Se analizan las siguientes herramientas:

- Power Designer 12.
- Racional Rose 2002.
- Netbeans UML.
- BlueJ.
- Bouml.
- ArgoUml.

Las herramientas como Power Designer y Racional Rose se utilizo versiones demo, debido a que su obtención con toda la funcionalidad implicaría un gran costo y son exclusivas de plataformas Windows. Las restantes son de distribución libre, multiplataformas y se adquieren es sus respectivos sitios web.

2.2.1 Ventajas de las herramientas CASE.

Entre los beneficios de usar herramientas CASE en el desarrollo de software:

- Facilidad de verificar y mantener de la consistencia de la información.
- Facilita establecer estándares en los procesos de desarrollo y documentación.
- Facilita el mantenimiento y actualización de la documentación.
- Permite la aplicación de diversas metodologías de desarrollo.
- Facilita la aplicación de técnicas de reutilización e ingeniería.

- Permite la gestión y planificación de proyectos.
- Generación de código.
- Permite incrementar la productividad al hacer optimización de recursos.

2.2.2 Desventajas de las herramientas CASE.

Entre los inconvenientes del uso de herramientas CASE tenemos:

- Adquisición de herramienta con amplia funcionalidad implica un gran costo económico.
- Son herramientas con un alto nivel de complejidad, que puede requerir de una gran inversión de tiempo de aprendizaje para su manejo.

2.2.3 Criterios de evaluación de herramientas CASE.

Los criterios y ponderaciones seleccionados para el análisis de las herramientas CASE son los siguientes:

- Facilidad de uso (30%): considerando la interfaz de usuario, apariencia, área de trabajo, además la facilidad para crear diagramas de clases.
- Lenguaje UML (20%): comparando los elementos notacionales de la herramienta con el estándar UML 2.0.
- Generador de código (20%): se pretende verificar si cuentan con generador de código ejecutable y su facilidad de creación.
- Soporte de usuario (30%): si esta herramienta proporciona la ayuda necesaria al usuario para utilizarla.

2.2.4 Herramientas.

Power Designer 12 (Plataformas Microsoft)

Facilidad de Uso (28% de 30%)

Posee una interfaz muy agradable y fácil de usar, se elige el tipo de diagrama a crear, luego en la interfaz se cargan los componentes necesarios del diagrama seleccionado. La etapa de creación es muy amigable, la modificación a los

diagramas no se realiza directamente en el grafo; sino por medio de ventanas de configuración. Es una herramienta de diseño muy poderosa.

Lenguaje de UML (20 de 20%)
Soporta los diagramas de UML en su mayoría, se trabajan con los componentes necesarios de cada diagrama, los cuales están posicionados en una barra de herramientas.

Generación de Código (17% de 20%)

- ✓ La generación de código es muy fácil de usar, no se requiere de mucha configuración para realizar este proceso.
- ✓ El plantilla de código generado es muy fácil de entender.
- ✓ La generación de la plantilla de código es en el LP (Lenguaje de Programación) Java.

Soporte Usuario (30% de 30%)

- ✓ La ayuda está disponible en cualquier momento, esta ayuda se encuentra por cada pantalla de configuración
- ✓ El soporte disponible offline es fácil de entender.
- ✓ Cuenta con soporte en línea, mediante sitios Web.

Racional Rose 2002 (Plataformas Microsoft)
Facilidad de Uso (25% de 30%)
Este software posee cierto grado de complejidad al utilizarlo, por la saturación de opciones con las que cuenta, por ser una herramienta muy poderosa es posible que un usuario inexperto se pierda cuando realice la edición de un diagrama.

Lenguaje de UML (20% de 20%)
Posee la mayor parte del estándar UML, el único inconveniente es que no es posible utilizar todos los componentes de UML de manera gráfica; pero, si se pueden utilizar por medio de ventanas de configuración a medida se avanza en el diseño de los diagramas.

Generación de Código (20% de 20%)
Cuenta con poderosos generadores de código a los lenguajes de programación C++, Ada 83, Ada 95, Java entre otros. El inconveniente es la configuración un tanto complicada que se debe seguir, cuando se generará el código.

Soporte Usuario (25% de 30%)
Cuenta con soporte offline, aunque un tanto compleja por la saturación de temas de ayuda. El nivel de abstracción de la información es un tanto compleja; por esta razón un usuario inexperto no podría extraer todo el potencial del soporte.

UML de Netbeans (Multiplataforma)
Facilidad de Uso (25% de 30%)
Posee una facilidad de uso un tanto escondida, por no ser como cualquier editor normal (al hacer doble clic sobre el lugar especifico se debe editar) para poder editar es necesario conocer la combinación de teclas indicadas, por ejemplo, para poder editar o crear los atributos de diagrama de clases es necesario presionar Alt+Mayúscula+A, también para las operaciones; pero por lo demás es fácil de usar y posea una interfaz agradable, similar a la empleada en power designer.

Lenguaje de UML (20% de 20%)
UML lo soporta en su mayoría, se apega a los diagramas definidos del lenguaje. Los diagramas que soporta son: diagramas de actividad, diagramas de clases, diagramas de colaboración, diagramas de componentes, diagramas de despliegue, diagramas de secuencia, diagramas de estado y diagramas de casos de uso. Los diagramas se pueden editar gráficamente y mediante venta.

Generación de Código (20% de 20%)
La generación de la plantilla de código es solamente en el LP Java; pero no por ello pierde fortaleza. Al contrario al ser solo en un LP es muy fácil generar la plantilla de código, esto se hace en cualquier momento del desarrollo, para ello la clase

generada puede insertarse en un proyecto, el cual es indicado por el usuario al momento de la generación de código.

Soporte Usuario (25% de 30%)
Cuenta con ayuda offline (en ingles), es una ayuda fácil de utilizar por estar distribuida en una estructura de árbol, por lo tanto es muy fácil encontrar la solución a los problemas que surgen por el uso de la interfaz. Además cuanta con ayuda online donde se encuentran tutoriales a diferentes niveles de usuarios.

BlueJ. (Multiplataforma)
Facilidad de Uso. (30% de 30%)
La interfaz es sencilla y con amplio espacio de trabajo. Posee pocos botones, la selección de elementos gráficos se hace a través de un menú que contiene sus nombres. La conexión entre elementos es fácil y rápida. Posee un visor de objetos y un evaluador de expresiones en Java.

Lenguaje UML. (5% de 20%)
Solamente genera diagramas de clases y las respectivas instancias que el usuario desee. Es visible el nombre de la clase y su representación, los atributos y métodos no son mostrados. Permite la realización de relaciones y herencias, además de representar paquetes.

Generación de Código. (20% de 20%)

Posee una opción llamada “Compile” (Compilar) con el cual una ves colocado el elemento grafico en el área de trabajo permite generar en segundo plano su código fuente en Java. Cuenta con editor de código, es aquí donde se establecen los atributos y métodos que la clase utilizara.

Soporte Usuario. (25% de 30%)
Es posible acceder a manuales y documentación del sistema desde la interfase grafica. Una ves se “compilan” las modificaciones a una clase, las inicializaciones y la

manipulación de objetos se puede realizar a través de asistentes desde la interfaz grafica. El usuario debe poseer un conocimiento del lenguaje Java para realizar las modificaciones de las clases de lo contrario se limitara a dibujar clases y objetos sin una funcionalidad especifica.

Bouml. (Multiplataforma)

Facilidad de Uso. (20% de 30%)

La interfase grafica es visualmente simple, no muy atractiva al usuario. En un inicio cuenta con pocas opciones para la construcción de diagramas. El área de trabajo es amplia pero no es visible hasta que se ha definido lo que se desea construir. Los elementos gráficos son de tamaño fijo estos crecen a medida se modifican y son accesibles a través de un navegador (browser). Además al colocar un grafico permite agregar texto sin alterar la configuración de diseño.

Lenguaje UML. (15% de 20%)

Soporto diagramas de clases y paquetes, diagramas de objetos, vista de componentes y distribución, diagramas de casos de usos. Una vez seleccionado el tipo de diagrama se habilitan múltiples opciones propias de cada diagrama permitiendo realizar diseño mas especializados.

Generación de Código. (20% de 20%)

Genera código fuente para los lenguajes C++ y Java, además genera información en formato HTML, XML 1.2, XML 2.1. Se necesitan realizar varias configuraciones para acceder al código fuente lo cual dificulta su revisión.

Soporte Usuario. (10% de 30%)

No cuenta con acceso a documentación, lo cual vuelve difícil el trabajo para el usuario ya que cuenta con múltiples opciones configurables. El proceso de diseño se realiza por medio de asistentes.

ARGOUML (Multiplataforma)

Facilidad de Uso. (30% de 30%)

Interfaz visualmente agradable, área de trabajo no muy espaciosa pero ajustable. Posee un navegador de objetos y dos sub-secciones en el área de trabajo: una para priorizar objetos y la otra con información de configuración, ayuda y propiedades. Todos lo elementos son visibles desde el inicio de la aplicación. Los componentes gráficos son fáciles de editar, ajustar en tamaño y modificar.

Lenguaje UML. (20% de 20%)

Permite generar diagramas de clases, casos, estados, actividades, colaboración, instalación y secuencias. Incluye opciones propias de cada diagrama para construirlos con más detalle.

Generación de Código. (20% de 20%)

Genera código fuente para el lenguaje java puede ser para todas las clases del diagrama o las seleccionadas por el usuario, incluye todas las especificaciones del usuario.

Soporte Usuario. (30% de 30%)

Cuenta con un excelente soporte de usuario, cuenta con una opción “Critiques” (Criticas) que brinda indicaciones de la construcción de los elementos del diagrama así como de envió de correo electrónico a expertos.

CAPITULO III

LENGUAJE DE MODELADO UNIFICADO UML.

3.1 Conceptos Básicos.

El Lenguaje de Modelado Unificado (UML) es un lenguaje estándar para software y desarrollo de sistemas. Este lenguaje permite modelar un sistema desde las perspectivas lógica, física, procedimental y de desarrollo. Para hacerlo hace uso de diversos diagramas que se encargan de modelar la complejidad que conlleva el desarrollo de software o de sistemas.

A continuación se explican los elementos notacionales que se ocuparon para el desarrollo del prototipo de editor J-UML, tanto para el desarrollo de la investigación y los utilizados por la aplicación de software para la creación de diagramas.

Antes de profundizar en los diferentes diagramas, es necesario hablar de dos elementos notacionales utilizados para modelar: notas y estereotipos.

Notas.

Las notas permiten agregar información adicional que no es capturada en los diagramas. La notas son ayudas que facilitan la lectura y comprensión de los diagramas, estos pueden encontrarse aislados o conectados a otro elemento. La notación utilizada se muestra en la figura 3.1.

Figura 3.1 Representación de notas en UML.

Estereotipos.

Permiten modificar el significado de un elemento describiendo el rol que este desempeña en el modelo. Además puede ser aplicado a la mayoría de elementos de la notación UML. Para representar los estereotipos se utiliza un doble ángulo y el

nombre, de esta manera <<nombre de estereotipo>>. UML no limita el número de estereotipos asociados a un elemento.

3.2 Diagrama de Casos de Uso.

Los diagramas de casos de uso son el punto de partida en el desarrollo de sistemas orientado a objetos. Un caso de uso es una situación en la cual un *modelo* debe satisfacer uno o más requerimientos del usuario. Además permiten describir partes funcionales del sistema de una perspectiva "externa" al modelo.

Los casos de uso deben especificar lo que el *modelo* debe hacer, para ello ocupan los siguientes elementos notacionales:

Actor.

Un actor es representado por un "hombre de palo" o por medios de un rectángulo estereotipado, etiquetado con su respectivo nombre como se muestra en la figura 3.2.

Figura 3.2. Representación del actor usuario.

Es importante resaltar que el actor no forma parte del sistema sino que interactúa con el. El actor no necesariamente son personas, se pueden pensar en ellos como cajas negras, no interesa como funcionan sino como interactúan con el sistema.

Caso de Uso.

Definidos los actores se deben representar las situaciones o acciones donde son involucrados. La notación UML de los casos de usos se muestra en la figura 3.3.

Figura 3.3. Representación caso uso.

Líneas de Comunicación.

Este elemento permite identificar en que casos de uso participa un actor, no existe un límite numérico de cuantos actores por caso de uso pueden participar. Para hacer esta comunicación es conectar una línea entre el actor y el caso de uso en que participa como se muestra en la figura 3.4.

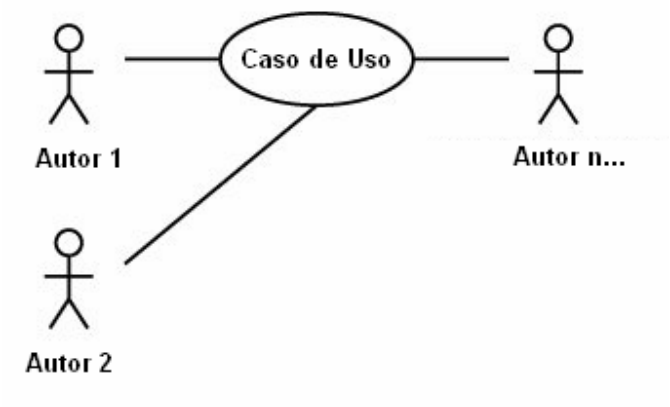


Figura 3.4. Comunicación Actor-Caso de uso.

Relaciones entre casos de uso.

Las relaciones permiten la relación entre casos de uso de manera estructural. Entre las relaciones tenemos:

Asociación.

Es la relación que consiste en comunicar un actor y un caso de uso, es similar a las líneas de comunicación con excepción que agregan navegabilidad al sistema al dibujar una flecha. Figura 3.5.

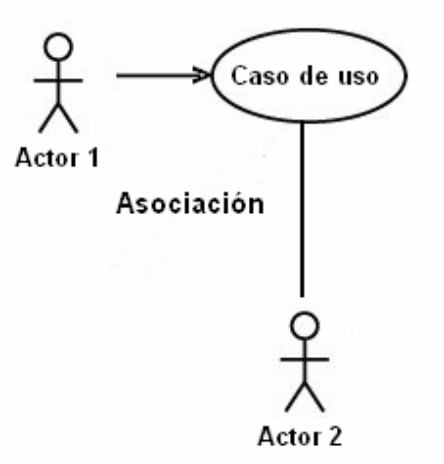


Figura 3.5 Relación de asociación entre casos de uso.

Inclusión.

La relación de inclusión, en ingles <<include>>, se utiliza cuando existen comportamientos similares entre uno o más casos de uso, permitiendo eliminar las descripciones de acciones repetitivas.

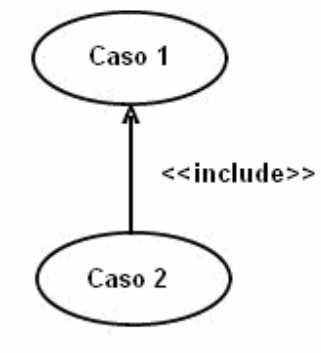


Figura 3.6 Relación de inclusión.

Generalización.

Es la relación que ocurre entre un caso con un comportamiento general y otro con acciones más específicas. Un caso de uso particular hereda de un caso de uso general sus acciones agregando las suyas propias.

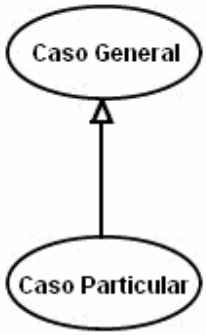


Figura 3.7 Relación de generalización.

Extensión.

Es la relación que ocurre cuando un caso de uso tiene un comportamiento similar a otro, pero este último contiene mayor funcionalidad.

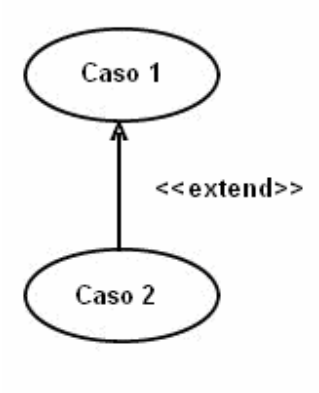


Figura 3.8 Relación de extensión.

3.3 Diagrama de clases.

Las clases son el corazón de la programación orientada a objetos, estas permiten describir atributos y operaciones de objetos creados a partir de ellas. Los atributos de una clase representan las características relevantes que describen al objeto mientras que las operaciones los comportamientos que realiza.

La representación de las clases en UML consiste en un rectángulo dividido en tres secciones. La parte superior contiene el nombre de la clase, la sección media contiene los atributos y la parte inferior contiene las operaciones.

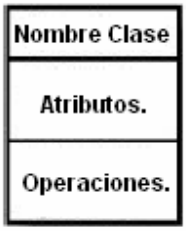


Figura 3.9 Representación de una clase.

Los atributos y operaciones son partes opcionales, el no mostrarlos no le quita información a la clase simplemente facilita la lectura.

Visibilidad.

La visibilidad permite limitar el acceso del mundo exterior a los atributos y operaciones de las clases. La tabla 1.5 muestra la notación UML para la visibilidad.

Tabla 1.3 Tabla de visibilidad UML.

Notación.	Nombre.	Descripción.
+	**Publico.**	Es posible acceder clase desde cualquier parte del sistema.
#	**Protegido.**	Sólo pueden acceder a la clase otras clases que hereden de ella.
~	**Paquete.**	Sólo pueden acceder a la clase la que pertenezcan al mismo paquete*****.
-	**Privado.**	La información de la clase es inaccesible al resto sistema.

*****Un paquete es una colección de clases.

Sintaxis UML para atributos y operaciones.

Para colocar los atributos en una clase se utiliza la siguiente sintaxis:

visibilidad nombre: tipo = valor_inicial, donde:

- Visibilidad: el tipo de accesibilidad al exterior de la clase.
- Nombre: identificador del atributo.
- Tipo: tipo de dato que será el atributo.

- Valor_inicial: dato con que se inicializara.

Para colocar una operación en una clase se utiliza la siguiente sintaxis:
visibilidad nombre (parámetros): tipo_valor_retorno, donde:

- Visibilidad: tipo de accesibilidad al exterior de la clase.
- Nombre: identificador de la operación.
- Parámetros: son los tipos de datos que toma de entrada.
- Tipo_valor_retorno: tipo de valor de retorno.

Relaciones entre clases.

Las clases pueden interactuar entre si por medio de las siguientes relaciones:

Dependencia.

Es la relación entre dos clases en la cual una clase necesita conocer de otra clase para usar objetos de ella. Su notación UML se muestra en la figura 3.10.

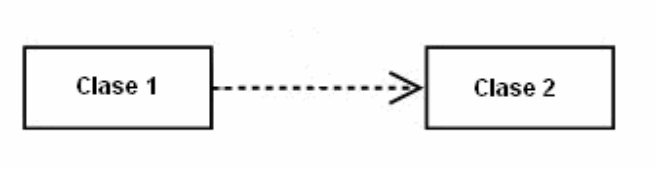


Figura 3.10. Dependencia de clases.

Asociación.

En esta relación una clase contiene referencias a uno o más objetos de otra clase en forma de atributos. Al número de objetos que participan en la asociación de dos clases se le conoce como multiplicidad. Su notación UML se muestra en la figura 3.11.

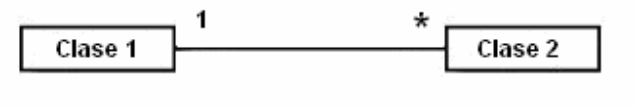


Figura 3.11. Asociación entre clases con relación de uno a muchos.

Agregación.

Es un tipo especial de asociación, el la cual se indica el grado de pertenencia de una clase con respecto a otra permitiendo que los objetos puedan ser utilizados por otra clase. Su notación UML se muestra en la figura 3.12.

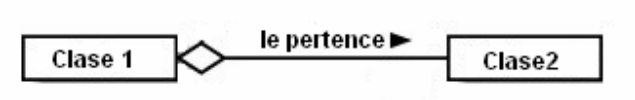


Figura 3.12. Relación de agregación.

Composición.

Esta es una relación similar a la agregación, pero se encarga de modelar las partes internas que constituye a una clase. Su notación UML se muestra en la figura 3.13.

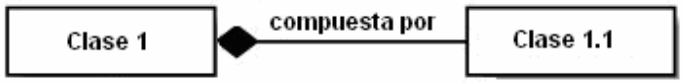


Figura 3.13. Composición de clases.

Generalización.

Conocida por herencia, permite modelar una subclase que hereda atributos y operaciones de su superclase, permitiendo además a la subclase implementar los suyos propio. Su notación UML se muestra en la figura 3.14.

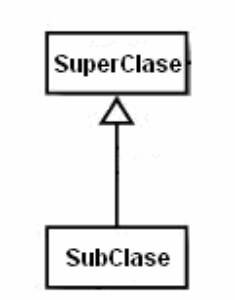


Figura 3.14 Generalización de clases.

Interfaces.

Las interfaces se consideran un tipo avanzado de clase, estas consisten en una colección de operaciones en las cuales su implementación dependerá de otra clase, subsistema u otro componente. La notación UML permite representarla a través de una clase estereotipada o de un círculo. Como se muestra en la figura 3.15.

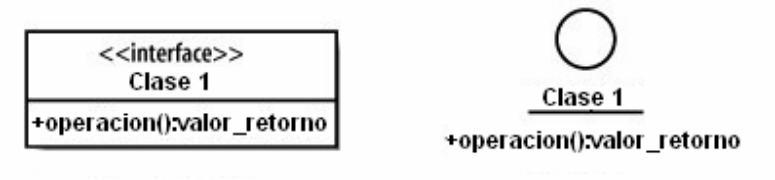


Figura 3.15. Notación UML interfaces.

3.4 Diagrama de paquetes.

Los paquetes son elementos notacionales que permiten representar primordialmente un grupo de clases, sin embargo puede ser utilizado para organizar casi cualquier elemento de la notación UML (diagrama de casos de uso, diagrama de iteración incluso otros paquetes). Su representación se muestra en la figura 3.16.

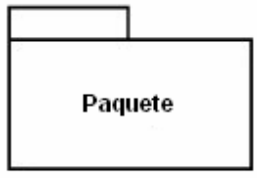


Figura 3.16. Notación UML Paquetes.

Dependencia de Paquetes.

Este tipo de relación es similar a las dependencias entre clases, si un paquete ocupa elementos que pertenezcan a otro, entonces se dice que hay dependencia de paquetes. La Figura 3.17 muestra la notación de dependencia.

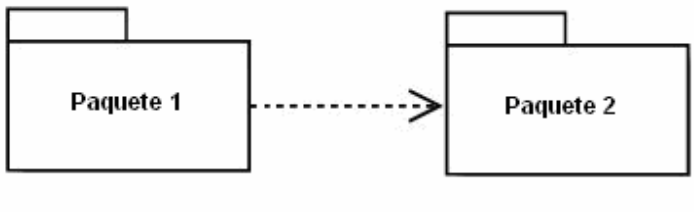


Figura 3.17. Dependencia de paquetes.

3.5 Diagrama de interacción.

Los casos de uso describen lo que el sistemas es capas de hacer, mientras que las clases se encargan de la descripción de su estructura interna. Pero por si mismos no pueden modelar aun como el sistema realizara su funcionamiento, para ello se utilizan los diagramas de interacción, los cuales pueden ser de tres tipos: secuencia, colaboración y coordinación.

3.5.1 Diagrama de secuencias.

Estos diagramas se encargan de describir las interacciones que se llevan a cabo al implementar un caso de uso y el orden en que son ejecutadas.

Para construir este tipo de diagramas hacemos uso de los siguientes elementos notacionales:

Participantes.

Los diagramas de secuencia están constituidos por una colección de participantes que son partes del sistema que interactúan entre si durante una secuencia. Estos se encuentran ordenados horizontalmente evitando sobreponer un participante en otro, como se muestra en la figura 3.18.

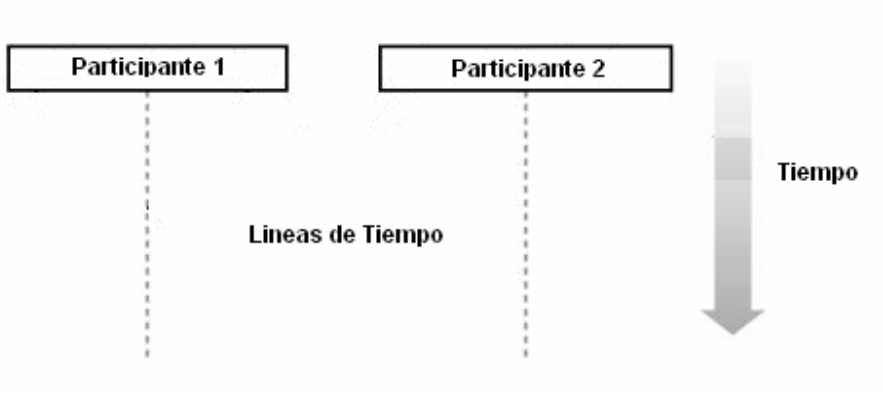


Figura 3.18. Participantes en un diagrama de secuencias.

Las líneas cortadas verticales son conocidas como el tiempo de vida del participante, es un indicador de cuando el participante es creado o destruido.

Tiempo.

Los diagramas de secuencias toman en cuenta el orden en que suceden las acciones, es por esta razón que el tiempo es un elemento importante. Este comienza a la altura donde se inicio a construir el diagrama y se extiende hasta el final.

Eventos.

Cualquier punto en el diagrama en donde ocurra una interacción es conocido como evento. Estos eventos ocurren a través de mensajes entre los participantes, como muestra la figura 3.19.

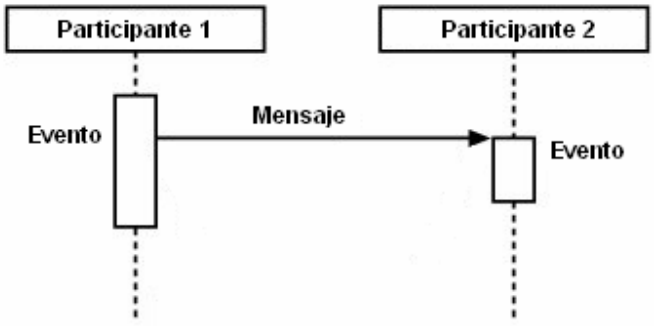


Figura 3.19. Eventos y mensajes entre participantes.

Mensajes.

Los mensajes entre eventos se clasifican de la siguiente manera:

- Asíncrono: cuando un participante envía mensajes sin esperar respuesta para enviar otros.
- Síncrono: un participante envía mensaje esperando una respuesta para continuar enviando.
- Retorno: es una notación opcional que permite devolver mensajes.

La figura 3.20 muestra la notación para los diferentes tipos de mensajes:

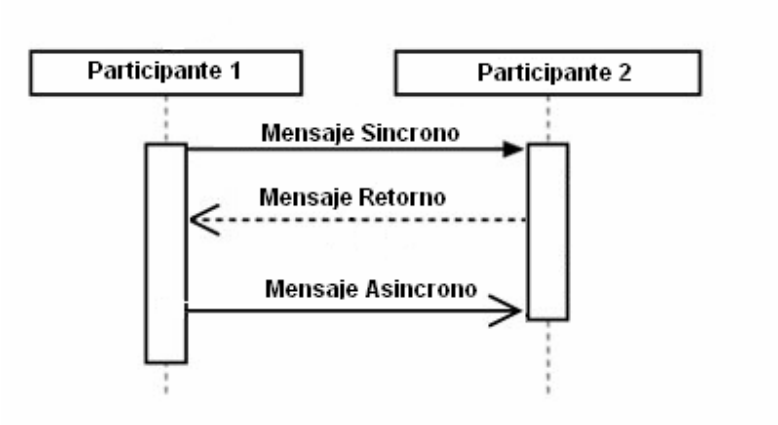


Figura 3.20 Tipos de mensajes entre participantes.

CAPITULO IV
PROCESO UNIFICADO

4.1 Definición de proceso unificado

El proceso unificado (UP) es un proceso de desarrollo de software, el cual Ivar Jacobson (unos de sus creadores) lo define como "el conjunto de actividades necesarias para transformar los requisitos de un usuario en un sistema de software". Además de la definición es bueno considerar que el UP es un proceso de desarrollo de software configurable que se adapta a las necesidades a través de los proyectos variados en tamaños y complejidad.

Los aspectos que distinguen al UP de otros procesos son:

- Dirigido por Casos de Uso
- Centrado en Arquitectura
- Iterativo e Incremental

Como está dirigido por casos de uso, el proceso se centra en las funcionalidades del sistema, por lo tanto se comienzan especificando los requisitos. Los casos de uso también guían el proceso de desarrollo (diseño, implementación, y prueba). Basándose en los casos de uso los desarrolladores crean una serie de modelos de diseño e implementación que llevan a cabo los casos de uso. De este modo los casos de uso no solo inician el proceso de desarrollo sino que le proporcionan un hilo por el cual desplazarse, se avanza a través de una serie de flujos de trabajo que parten de los casos de uso.

La arquitectura de un sistema de software, se describe mediante diferentes vistas del sistema en construcción. Es por ello que UP describe al sistema desde varios puntos de vistas, es decir, la forma en como ven al sistema las personas que están involucradas.

Los casos de uso y la arquitectura están profundamente relacionados. Los casos de uso deben encajar en la arquitectura, y a su vez la arquitectura debe permitir el desarrollo de todos los casos de uso requeridos, actualmente y a futuro.

Una iteración es una secuencia de actividades dentro de un plan establecido, con unos criterios claros de evaluación, que se organiza con el propósito de entregar parte de la funcionalidad del producto. Con ello el proyecto se entrega en pequeños mini proyectos, los cuales representan un incremento en el desarrollo del proyecto; por lo tanto se consideran como una serie de pasos para llegar al objetivo.

Beneficios del enfoque iterativo

- La iteración controlada reduce el riesgo a los costes de un solo incremento.
- Reduce el riesgo de retrasos en el calendario atacando los riesgos más importantes primero.
- Acelera el desarrollo. Los trabajadores trabajan de manera más eficiente al obtener resultados a corto plazo.
- Tiene un enfoque más realista al reconocer que los requisitos no pueden definirse completamente al principio.

4.2 La vida del proceso unificado

El Proceso Unificado se repite a lo largo de una serie de fases que constituyen la vida del sistema. Estas fases son

- Inicio (Inception)
- Elaboración (Elaboration)
- Construcción (Constrution)
- Transición (Transition)

Cada fase se divide en iteraciones, estas iteraciones se repiten a largo de una serie de ciclos, al finalizar un ciclo se obtiene una versión del software. En una iteración se pueden llevar a cabo los 5 fundamentales flujos de trabajo del proceso unificado, estos flujos son:

- Requisitos (Requirements)
- Análisis (Analysis)
- Diseño (Desing)
- Implementación (Implentation)

- Pruebas (Test)

4.2.1 Fase de Inicio

En esta fase de inicio se desarrolla una descripción del producto final (J-UML), y se presenta el análisis del proyecto. Esta fase se responderá las siguientes preguntas:

- ¿Cuáles son las funciones más importantes para el usuario?
- ¿Cómo podría ser la mejor arquitectura del sistema?
- ¿Cuál es el plan del proyecto y cuanto costará desarrollar el producto?

En esta fase se identifican y priorizan los riesgos mas importantes. El objetivo es ayudar al equipo de proyecto a decidir cuales son los verdaderos objetivos del proyecto. Las iteraciones exploran diferentes soluciones posibles, y diferentes arquitecturas posibles.

Puede que todo el trabajo físico realizado en esta fase sea descartado. Lo único que normalmente sobrevive a la fase de inicio es el incremento del conocimiento en el equipo.

4.2.2 Fase de Elaboración

Durante la fase de elaboración se especifican en detalle la mayoría de los casos de uso del producto y se diseña la arquitectura. El resultado de esta fase es la línea base de la arquitectura.

En esta fase se construyen típicamente los siguientes artefactos:

- El cuerpo básico del software en la forma de un prototipo arquitectural.
- Casos de prueba
- La mayoría de los casos de uso (80%) que describen la funcionalidad del sistema.
- Un plan detallado para las siguientes iteraciones.

La fase de elaboración finaliza con el hito de la arquitectura del Ciclo de Vida.

Este hito se alcanza cuando el equipo de desarrollo y los stakeholders (grupo interesado en desarrollar el sistema) llegan a un acuerdo sobre:

- Los casos de uso que describen la funcionalidad del sistema.
- La línea base de la arquitectura
- Los mayores riesgos han sido mitigados
- El plan del proyecto

Al alcanzar este hito debe poder responderse a preguntas como:

- ¿Se ha creado una línea base de la arquitectura? ¿Es adaptable y robusta? ¿Puede evolucionar?
- ¿Se han identificado y mitigado los riesgos más graves?
- ¿Se ha desarrollado un plan del proyecto hasta el nivel necesario para respaldar una agenda, costes, y calidad realistas?
- ¿Proporciona el proyecto, una adecuada recuperación de la inversión?
- ¿Se ha obtenido la aprobación de los inversores?

4.2.3 Fase de Construcción

En esta fase se realiza la construcción del software, a esta altura ya se debe tener el diseño de la arquitectura que se desarrolló en la fase de elaboración, la cual se toma como línea base la cual crece hasta la finalización de la construcción del producto.

Los artefactos producidos durante esta fase son:

- El sistema software
- Los casos de prueba
- Los manuales de usuario

La fase de construcción finaliza con el hito de Capacidad Operativa Inicial. Este hito se alcanza cuando el equipo de desarrollo y los stakeholders llegan a un acuerdo sobre:

- El software es estable
- El software provee alguna funcionalidad de valor para el usuario
- Todos los procesos están listos para migrar a la fase de transición

4.2.4 Fase de Transición

La fase de transición cubre el período durante el cual el producto se convierte en la versión beta, esta fase puede considerarse como una fase de prueba en la que un número determinado de usuarios realizan pruebas sobre el producto (versión beta) e informan de posibles errores o defectos de éste.

Las iteraciones en esta fase continúan agregando características al software. Sin embargo las características se agregan a un sistema que el usuario se encuentra utilizando activamente.

Los artefactos construidos en esta fase son los mismos que en la fase de construcción, el equipo se encuentra ocupado fundamentalmente en corregir y extender la funcionalidad del sistema desarrollado en la fase anterior, esta finaliza con el hito de Lanzamiento del Producto,.

Este hito se cumple cuando el equipo de desarrollo y los stakeholders llegan a un acuerdo sobre:

- Se han alcanzado los objetivos pactados en la fase de Inicio.
- El cliente está satisfecho.

4.3 Elementos del proceso unificado

Como se conoce el PU no es simplemente un proceso; sino más bien un marco de trabajo en el cual se definen adecuadamente *quién* esta haciendo *qué*, *cuándo* lo hace y *cómo* se debe hacer para llagar a la meta final que se traduce en la finalización del producto. En el PU estas interrogantes son representadas mediante estos cuatro elementos:

- ¿Quién? - Trabajadores
- ¿Cómo? - Actividades
- ¿Qué? - Artefactos
- ¿Cuándo? - Flujos de trabajo

4.3.1 Trabajadores

Un rol del trabajador define el comportamiento y responsabilidades de un individuo, o de un grupo de individuos trabajando juntos como un equipo. Una persona puede desempeñar diversos roles, así como un mismo rol puede ser representado por varias personas.

Las responsabilidades de un rol son tanto el llevar a cabo un conjunto de actividades como el ser el dueño de un conjunto de artefactos; según la metodología RUP (Rational Unified Process) que es la versión propietaria de UP, hoy en día de IBM define grupos de roles, agrupados por participación en actividades relacionadas. Estos grupos son

Analistas:

- Analista de procesos de negocio.
- Diseñador del negocio.
- Analista de sistema.
- Especificador de requisitos.

Desarrolladores:

- Arquitecto de software.
- Diseñador

- Diseñador de interfaz de usuario
- Diseñador de cápsulas.
- Diseñador de base de datos.
- Implementador.
- Integrador.

Gestores:

- Jefe de proyecto
- Jefe de control de cambios.
- Jefe de configuración.
- Jefe de pruebas
- Jefe de despliegue
- Ingeniero de procesos
- Revisor de gestión del proyecto
- Gestor de pruebas.

Apoyo:

- Documentador técnico
- Administrador de sistema
- Especialista en herramientas
- Desarrollador de cursos
- Artista gráfico

Especialista en pruebas:

- Especialista en Pruebas (*tester*)
- Analista de pruebas
- Diseñador de pruebas

Otros roles:

- *Stakeholders*.
- Revisor

- Coordinación de revisiones
- Revisor técnico
- Cualquier rol

Actividades.

Una actividad en concreto es una unidad de trabajo que una persona que desempeñe un rol puede ser solicitado a que realice. Las actividades tienen un objetivo concreto, normalmente expresado en términos de crear o actualizar algún producto, esto se debe a que una actividad posee los límites bien definidos.

Artefactos

Un producto o artefacto es una parte de información que es producida, modificada o usada durante el proceso de desarrollo de software. Los productos son los resultados tangibles del proyecto, los elementos que se van creando y usando hasta obtener el producto final; un artefacto puede ser cualquiera de los elementos siguientes:

- Un documento, como el documento de la arquitectura del software.
- Un modelo, como el modelo de Casos de Uso o el modelo de diseño.
- Un elemento del modelo, un elemento que pertenece a un modelo como una clase, un Caso de Uso o un subsistema.

Flujos de Trabajo.

Con los roles, actividades y artefactos no se define un proceso, necesitamos contar con una secuencia de actividades realizadas por los diferentes roles, así como la relación entre los mismos. Un flujo de trabajo es una relación de actividades que producen resultados observables.

4.4 Características del proceso unificado

Dirigido a casos de usos.

Los casos de usos permiten capturar y unificar los requerimientos del cliente. No todos los casos de usos son buenos, es necesario seleccionar aquellos que se ajustan en gran medida a las necesidades del cliente y con ello crear un modelo de casos de uso. Este modelo permitirá llegar a un acuerdo entre clientes y trabajadores de cómo utilizar el sistema durante los diferentes flujos de trabajo.

Centrado en arquitectura.

Con el modelo de casos de uso creado, se define la arquitectura que servirá de guía durante el proceso de desarrollo. Como arquitectura entenderemos al conjunto de decisiones y selección de elementos estructurales que componen un sistema: interfaces, comportamiento, colaboraciones y composición. Los casos de uso y la arquitectura van están sumamente relacionados. A medida que los casos de uso adquieren más especificaciones o funcionalidad se va descubriendo más arquitectura, que puede expresarse en términos de subsistemas, clases y componentes.

Proceso iterativo e incremental.

Cada esfuerzo realizado durante el proceso de creación de software se le conoce como miniproyectos. Cada uno de ellos se le considera una iteración o incremento. Este permite llevar un control o planificación durante el desarrollo, obteniendo resultados a corto plazo y una visión mas realista del producto final del cliente.

CAPITULO V

DESARROLLO DEL PROTOTIPO J-UML.

5.1 Requisitos del sistema.

El objetivo de los capítulos anteriores es brindar las bases teóricas para la comprensión de UML y UP. En este capitulo se desarrolla el prototipo de editor J-UML, su nombre se deriva de la combinación de J por java, por ser el lenguaje de programación y generación de código, y UML por ser el lenguaje de modelado.

El propósito de los requisitos es obtener una descripción correcta de lo que el sistema debe o no hacer. Por lo tanto debe haber un acuerdo entre usuarios y desarrolladores de lo que se pretende construir. La fase de inicio de UP se encarga de proveer este espacio para que los responsables del sistema realicen las actividades necesarias para generar los primeros artefactos de software.

Para el desarrollo del prototipo J-UML, la etapa de inicio lo constituye el capítulo 1 de ésta investigación, fue en ésta sección que se estableció los acuerdos entre usuarios y desarrolladores del sistema, acordando objetivos, alcances y limitaciones. El artefacto resultante de la etapa de inicio es el anteproyecto de trabajo de graduación y es considerado como primera iteración del sistema.

Para encontrar los requerimientos de usuario fue necesario tomar como base el capitulo 2, al encuestar y entrevistar usuarios obtuvo una referencia del nivel de conocimientos sobre metodologías y herramientas para el desarrollo de software. Con la evaluación de software fue posible encontrar las características necesarias para implementar J-UML.

5.1.1 Encontrar actores y casos de usos.

Los casos de uso son los diagramas que permitirán modelar los requerimientos, en primer lugar se establece lo que el sistema debe hacer, tomando como base el análisis de herramientas y los acuerdos entre usuarios y desarrolladores.

El prototipo J-UML permitirá realizar diagramas de clases UML, basada en la notación mostrada en el capitulo 3. El usuario podrá realizar lo siguiente:

- Colocar y eliminar dentro de un área de diseño los componentes UML, contenidos en una barra para la creación de diagramas, los cuales serán: clases, notas y relaciones de conexión a notas, asociación, generalización, agregación y dependencia.
- En un componente de clases, agregar atributos y operaciones directamente en el área de diseño.
- Crear, guardar y abrir un archivo que contendrá el diagrama.
- Generar un plantilla de código en JAVA que genere la representación estática.
- Exportar el diagrama a formato de archivo png (Portable Network Graphics).

Solamente es visible un único autor que es el usuario del programa. A continuación se muestran lo casos de usos identificados para el prototipo J-UML:

5.1.2 Requerimientos del sistema.

1. Crear Componente de Diagrama.
   1.1 Crear Componente de Clase.
   1.2 Crear Componente de Notas.
2. Crear Componente de Relación.
   2.1 Crear relación de Asociación.
   2.2. Crear relación de Generalización.
   2.3. Crear relación de Composición.
   2.4. Crear relación de Agregación.
3. Guardar Archivo Como.
4. Abrir Archivo.
5. Crear Nuevo Archivo.
6. Agregar atributos y/u operaciones a clases.

7. Editar información de nota.
8. Guardar diagrama en imagen.
9. Guardar Cambios.
10. Generar Código.

5.1.3 Diagramas de casos de uso para creación de diagramas.

La figura 4.1 muestra los casos de uso utilizados para modelar los requerimientos de creación de diagramas en J-UML. Solo es visible un autor que es el usuario de la aplicación.

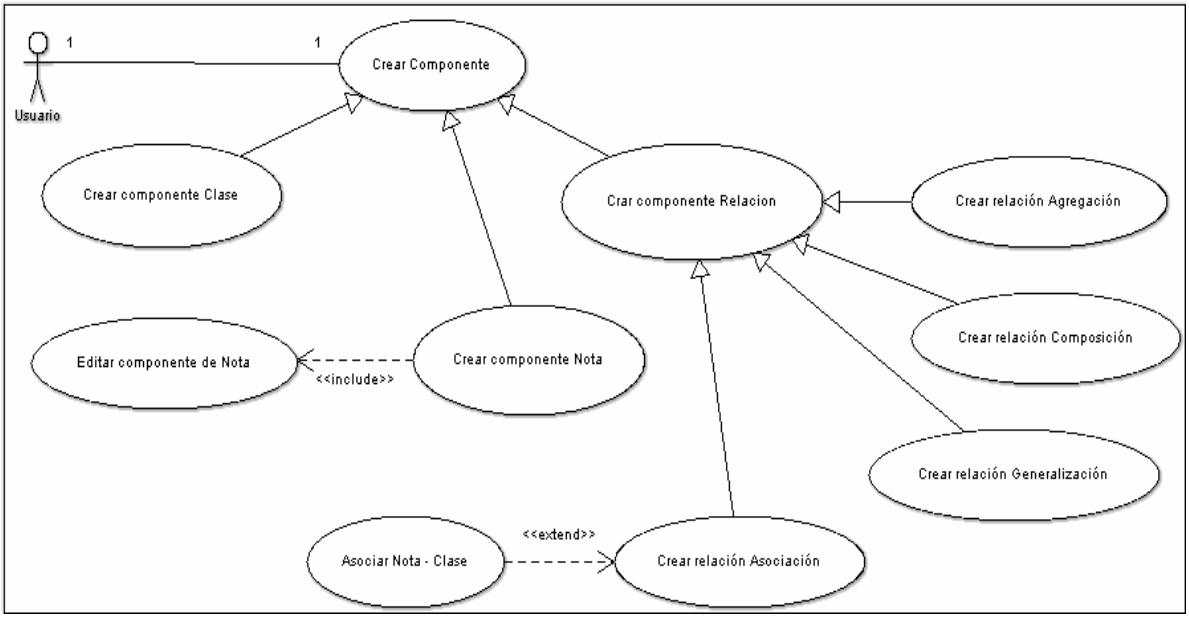


Figura 4.1 Diagrama de casos de uso para sección de diagramas.

5.1.4 Diagrama de casos de uso para la sección de editor.

La figura 4.2 muestra los casos de uso para los requerimientos del editor, todo los diagramas de casos de uso pertenecen al prototipo J-UML para efectos de claridad se han separado.

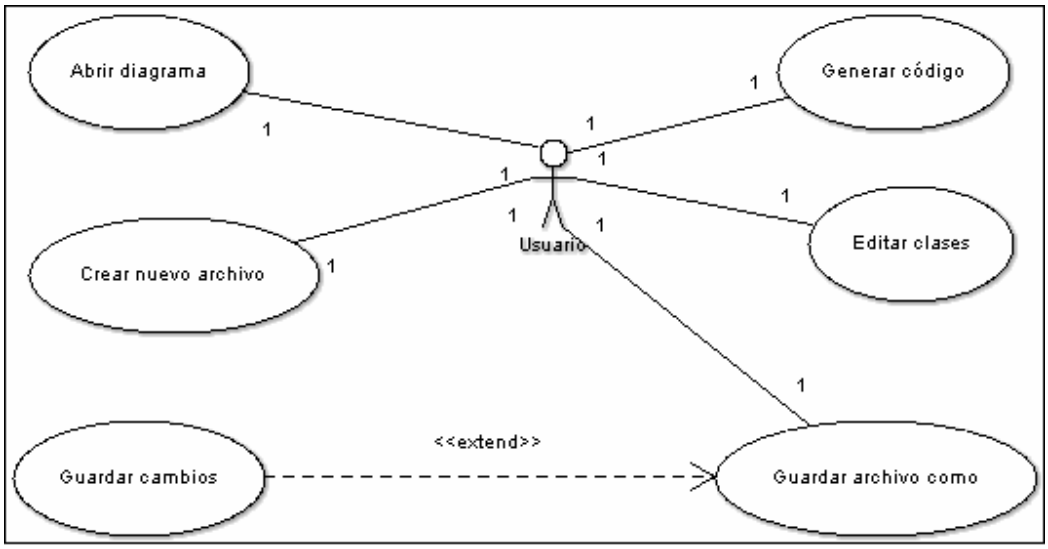


Figura 4.2 Casos de Uso para la parte de editor.

A los diagramas anteriores se les denomina modelo de casos de uso, los cuales hacen posibles la comunicación clara y sencilla entre el usuario y los desarrolladores. Sin embargo esto no es suficiente para comenzar la construcción de la arquitectura, por lo tanto es necesario detallar los casos de usos para que exista un único significado para cada diagrama.

5.1.5 Detalle de casos de uso.

La tabla 1.6 muestra como se detallan los casos de uso para el prototipo J-UML. Es posible identificar en la tabla aspectos como importantes como: actor, condiciones, objetivos y la secuencia de acciones entre el actor y su caso de estudio.

JUML Caso-Uso 01. Creación de componente de diagrama de clases.

Objetivo: El alumno agrega un elemento notacional al diagrama de clases.
Pre-Condición: El editor debe estar iniciado o pulsar sobre el ítem 'Nuevo'.

Flujo de sucesos:

Flujo principal.

1. El alumno pulsa sobre el ítem del elemento notacional que desea agregar.
2. El alumno hace clic sobre el área de diseño y coloca el elemento notacional.
3. JUML genera el elemento notacional seleccionado.
4. El alumno reacomoda el elemento notacional donde desee dentro del área de diseño.

Post-Condición: elemento notacional agregado en el diagrama de clases.

Tabla 1.4 Detalle de caso de uso para crear componentes del diagrama.

Nota: El detalle del resto de requerimientos se encuentra en el archivo Excel Detalle de casos de uso que se anexara con la copia digital.

5.1.6 Esquema de diseño editor.

J-UML es un prototipo de editor, por lo que requiere de una interfaz de modo que se acople a los requerimientos del usuario. La figura 4.3 muestra el esquema de la ventana principal de editor en donde el usuario podrá interactuar directamente con el software.

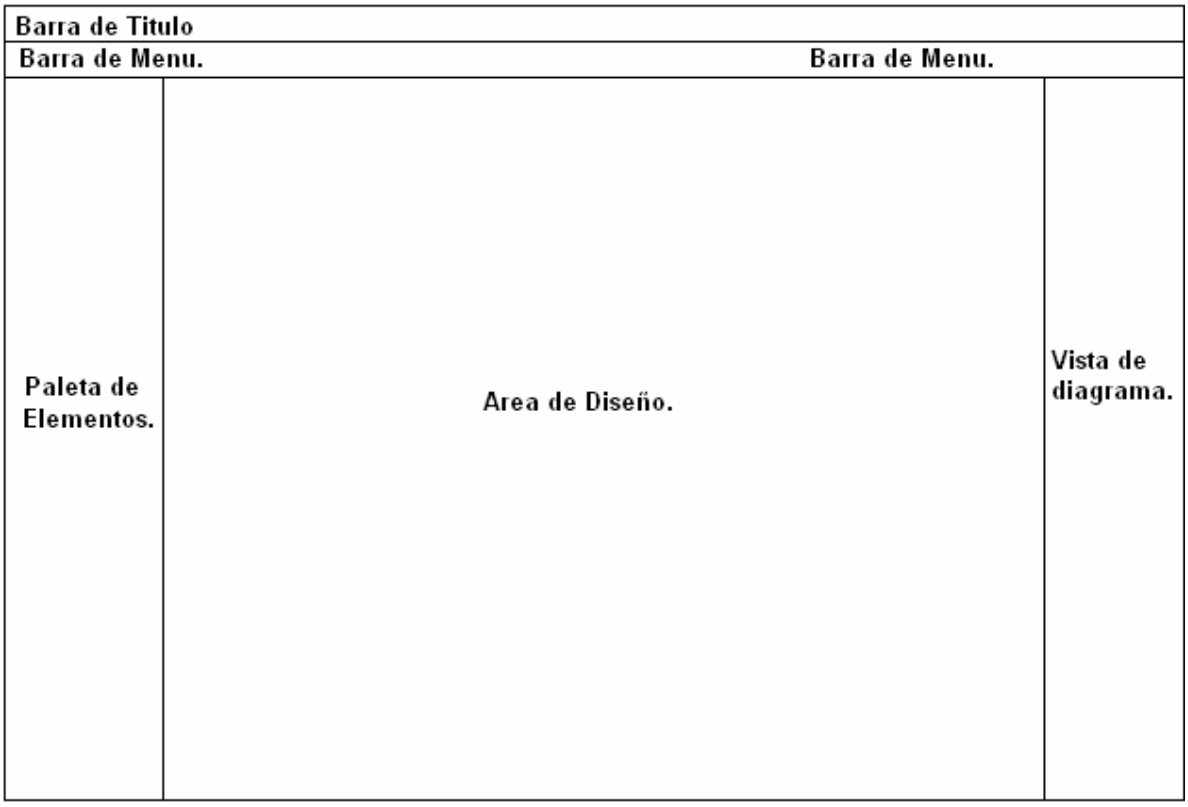


Figura 4.3 Esquema de Ventana Principal.

La ventana principal se divide en las siguientes secciones:

- Barra de Titulo: contiene el nombre de la aplicación y las funciones de maximizar – restaurar, minimizar y cerrar.
- Barra de Menú: contendrá las acciones necesarias de un editor: abrir, guardar, guardar como y generar código.
- Paleta de elementos: contendrá la notación UML que el usuario utilizara para la creación de diagramas: clase, notas y relaciones.
- Área de diseño: es el espacio donde el usuario colocara los elementos visuales para la creación de los diagramas.
- Vista Diagrama: es el área donde estará una vista aérea de todo el diagrama.

5.2 Análisis.

En esta etapa se refinan y especifican los requisitos del usuario, a través de la realización de los casos de usos que permita la construcción de la arquitectura de todo el sistema. La arquitectura de J-UML proporciona soporte a los requerimientos de usuario para diagramas de clases; además proporcionara el marco de trabajo para integrar más funcionalidad.

En esta etapa se obtienen los siguientes artefactos:

- Modelo de análisis.
- Clases de análisis.

5.2.1 Análisis de Arquitectura.

Para el diseño y construcción de la arquitectura es necesario abstraer un conjunto de objetos conceptuales que trabajan dentro del sistema y agruparlos en un modelo llamado modelo de análisis; el cual esta estructurado por clases que contendrán la funcionalidad interna del sistema.

5.2.2 Análisis de casos de uso.

La actividad de analizar casos de usos permite identificar las clases de análisis que representan un nivel de funcionalidad dentro del sistema, debido a que describen el comportamiento de un caso de uso; se hará uso de los diagramas de secuencias para ver la interacción entre ellas.

5.2.3 Identificación de clases.

Para esta actividad fue necesario analizar el flujo principal de acciones que se presentan en los casos de usos, identificando los sustantivos o pronombres en las descripciones y posteriormente renombrarlas como clases y agruparlos en subsistemas.

Las clases, comúnmente, suelen coincidir con tres los tres estereotipos básicos:

- Interfaz (boundary): clases que modelan la interacción entre actores y sistema.

- Control (control): que se encargan del manejo de operaciones de coordinación y objetos.
- Entidad (entity): sirven para modelar información duradera del sistema.

Para identificar y nombrar las clases se considero agregar la letra J como identificador de nombre de clase para hacer referencia a J-UML.

5.2.4 Identificación de relaciones.

Identificadas las clases que constituyen el sistema, es necesario establecer las relaciones que existen entre sus objetos correspondientes. Para ello se utiliza los diagramas de interacción.

Para esta actividad es necesario usar los diagramas de comunicación (en UML version 1.4 son los diagramas de colaboración), sin embargo, los diagramas de secuencia y comunicación muestran información similar por lo tanto se opto por utilizar diagramas de secuencia.

Las consideraciones que se tomaron son las siguientes:

- Se toman como base los casos de uso.
- Se sigue pasos a paso el flujo principal de acciones poniendo énfasis en los verbos, para identificar posibles mensajes entre objetos.

5.2.5 Análisis de clases.

Esta actividad consiste en identificar responsabilidades, atributos y relaciones entre clases.

5.2.5.1 Identificación de responsabilidades y atributos.

Las responsabilidades son las órdenes que puede solicitar un objeto para que realicen una acción, por lo tanto estas surgen a partir de los mensajes enviados entre objetos de las clases identificadas; además las responsabilidades se documentan en las clases. Para el caso se opto por nombrar lo más claro posible los nombres de la función.

Un atributo es también una responsabilidad, pero se encuentra limitado a contener información propia de la clase en sí.

5.2.5.2 Identificación de relaciones.

Asociaciones.

Para identificar una asociación se tomo en cuenta:

- Un objeto A es parte física o lógica de un objeto B.
- Un objeto A esta física o lógicamente contenido en B.
- Un objeto A es compuesto por B.

Para la agregación y composición se aplica las mismas consideraciones por ser un tipo mas especializado de relación.

Generalización.

Para este tipo de relación solo se identificaron clases con comportamientos similares enfocándose en las más obvias.

5.2.6 Diagrama de secuencias J-UML.

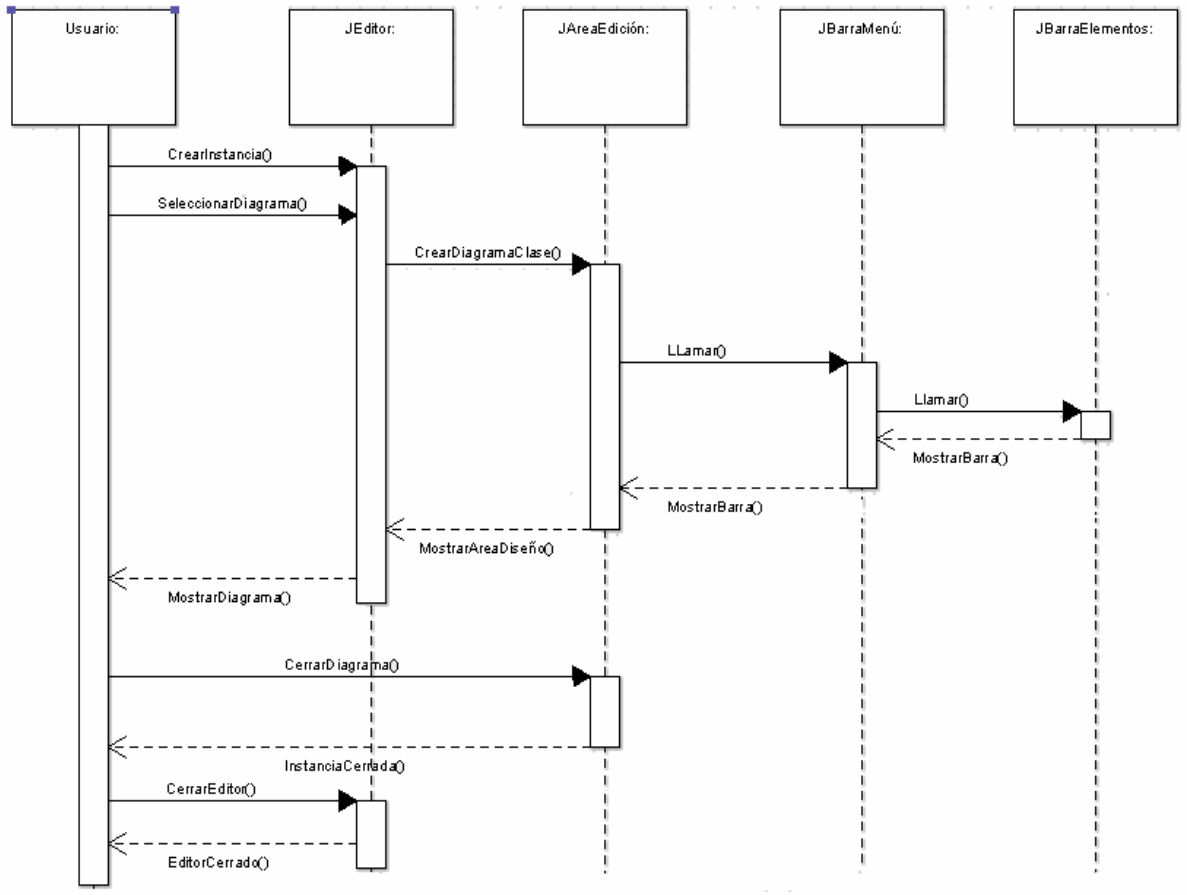


Figura 4.4. Diagrama de Secuencias de J-UML.

La figura 4.4 es el diagrama de secuencias general del J-UML, este diagrama muestra la interacción entre las diferentes clases identificadas en la etapa de análisis. Este proporciona el comportamiento básico de cómo el usuario interactúa con J-UML y sus componentes.

5.2.7 Diagrama de clases.

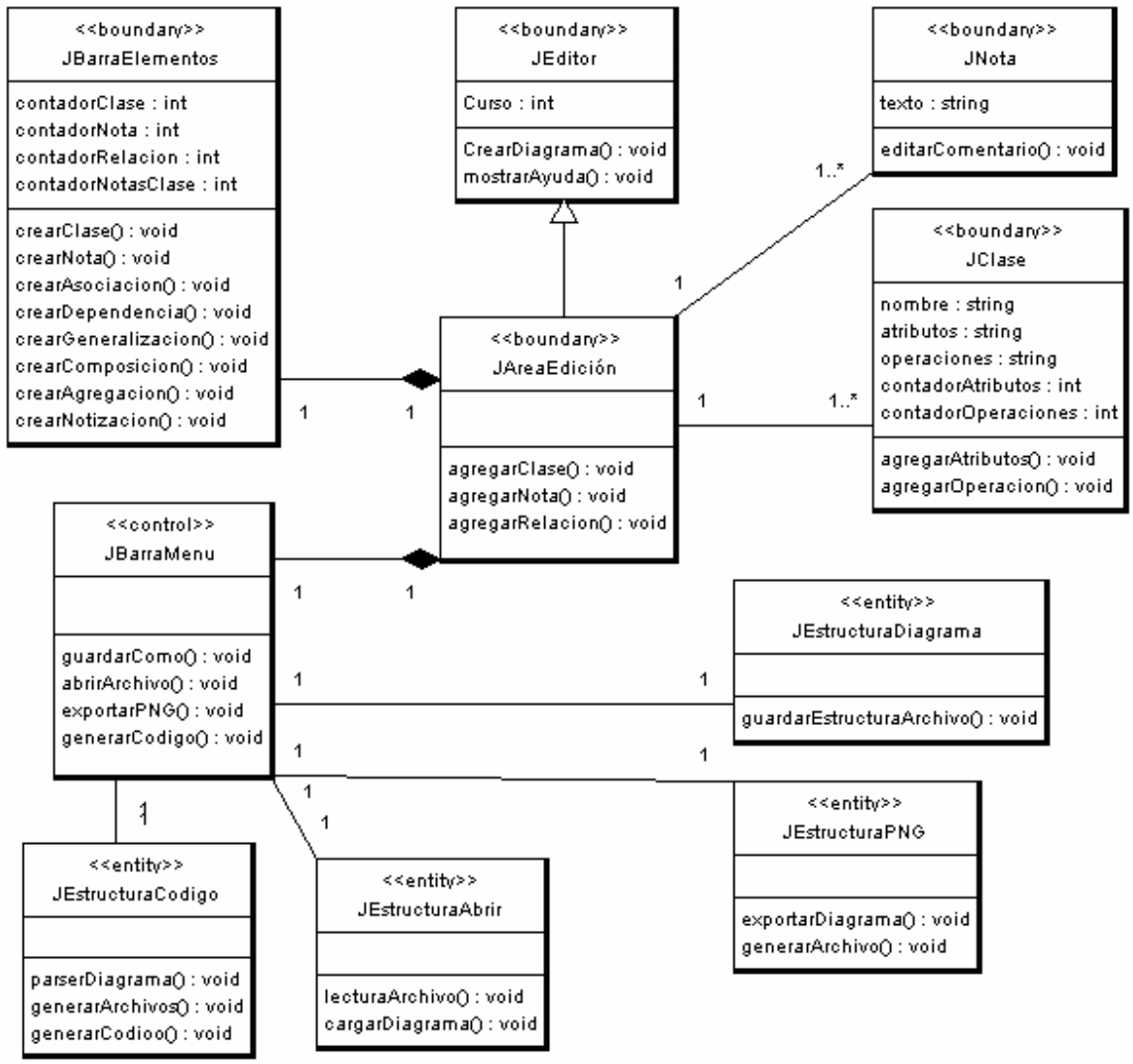


Figura 4.5 Diagrama de Clases de J-UML.

La figura 4.5 muestra el modelo de análisis de J-UML, contiene las clases y relaciones identificadas en la parte de análisis; éste diagrama es la guía para dirigir la etapa de diseño, por ser el modelo visual entendible para analistas y programadores de cómo la aplicación estará constituida.

5.3 Diseño.

Con el modelo de análisis y el diagrama de secuencias, como artefactos surgidos en la etapa de análisis, es posible obtener la visión general de la implementación del sistema. En esta etapa se desarrolla el diseño del editor, tomando de base los diagramas generados en la etapa previa.

Para la etapa de diseño se tomaron dos consideraciones:

- Para la creación del editor, se utiliza el diagrama de clases.
- Se utilizara la plataforma Netbeans para dar soporte a la funcionalidad del sistema.

5.3.1 Diseño del Editor.

En J-UML, el editor consiste en un marco de trabajo en donde se ejecutaran módulos con diferentes funcionalidades. El primer modulo lo constituyen los diagramas de clases y generador de código desarrollados en el presente trabajo. Este marco de trabajo proporciona la arquitectura para agregar más módulos que permitan implementar la amplia gama de diagramas UML y convertir a J-UML en un editor completo. La clase JEditor es la que se tomo como referencia para diseñar esta sección. La función del objeto de esta clase es proporcionar el marco de trabajo para la creación de diagramas de clase UML.

La plataforma Netbeans provee un marco de trabajo ajustado a los requerimientos de esta clase. Esto lo hace a través de las llamadas aplicaciones de cliente rico (rich client application), estas consisten en construcción de aplicaciones de escritorio en base a una arquitectura modular. Esta sección se desarrolla ampliamente en el Apéndice A.

5.3.2 Diseño del área de edición.

El área de edición en J-UML será un panel de fondo blanco que permita colocar los elementos del diagrama que el usuario crea conveniente; será encargada de

contener las herramientas con los elementos notacionales para la construcción diagramas, así como las funciones de almacenamiento de estos diagramas. La clase JAreaDiseño se tomara como guía de diseño. Las secciones para las herramientas y opciones de almacenamiento son:

- Barra de Menú: basada en la clase JBarraMenu, se encargara de realizar las acciones para generar código fuente, almacenar o recuperar diagramas en forma de archivo o imagen.
- Barra de Elementos: basada en la clase JBarraElementos, es la encargada del manejo de la notación especial para relacionar diagramas y atributos de control.

En las aplicaciones de cliente rico en netbeans, es posible crear dentro de un modulo un componente de ventana conocido como TopComponent. Este es un panel en java que sirve como área de trabajo para la construcción de interfaz de usuarios. Provee de las funciones básicas para manejo de ventanas y creación de una instancia única del área de diseño para J-UML.

5.3.3 Diseño de elemento notacional de clases.

Para este diseño se tomo como base la clase JClase. Una clase en J-UML estará dividida en tres secciones que representan el nombre de la clase, atributos y comportamientos que se desean modelar. La agregación o eliminación de atributos y comportamientos se realizara directamente en el diagrama; además de asignación de los permisos de acceso. Se provee la libertad para mover las clases en el área de diseño, para que el usuario lo coloque donde considere conveniente.

Netbeans, a partir de la versión 6.0 de su IDE, incluye en su plataforma la librería Visual 2.0 (Visual Library 2.0). La librería fue diseñada para dar soporte para la creación de diagramas orientados a grafos y propósitos generales de modelado. La unidad básica de modelado es conocida como Wigdet. Un widget es un modelo visual de representación de grafos con propósitos generales; es posible usarlos como contenedor de otros widgets, cadena de caracteres, imágenes y para representación de relaciones entre widget.

Por lo tanto en J-UML, los diagramas de clases que se modelen, en realidad serán un conjunto de widget ajustados para representar los componentes de clase, nota y las relaciones entre ellos.

5.3.4 Diseño de elemento notacional de nota.

Para este diseño se sigue el mismo esquema de las clases, con la excepción que solamente contendrá texto que sirva de documentación de las clases. Se toma como base la clase JNota.

5.3.5 Diseño del elemento notacional de relación.

Para el diseño de estas notaciones es necesario tener dos componentes creados, ya sea de clases o notas, para indicar que hay algún tipo de relación entre esas dos entidades.

Para representarlo es necesario usar un tipo especial de widget llamados widget de conexión (ConectionWidgets), los cuales requieren dos instancias previas de otros widget, además proveen movilidad en el área de diseño. Para una referencia mas detallas de los tipos de widget hágase referencia al apéndice A.

5.3.6 Diseño de las clases entidad (entity).

Para este tipo de clase no fue requerida una librería dedicadas para su desarrollo, la razón es que son clases estereotipadas para representar algún tipo de almacenamiento de datos, por lo tanto solo es necesario crear las funciones con la lógica de implementación requerida. Esta etapa se vera con más detalle en la etapa de implementación.

5.4 Implementación.

En esta sección especificaremos con más detalle la etapa de diseño, se usara un lenguaje mas especifico de lo que se requirió para la construcción del editor J-UML.

5.4.1 Implementación del editor.

Para la creación del editor se utilizo el marco de trabajo de netbeans, debido a su arquitectura modular es posible reutilizar la plataforma netbeans de tal manera que pueda ser ajustada a las necesidades del editor. Para ello es necesario crear en netbeans un contenedor de módulos (module suite), este permite crear un marco de trabajo en donde se añaden o remueven módulos con parte de la funcionalidad del editor J-UML. Además provee de una organización de paquetes para que el desarrollador almacene los archivos de java necesarios para el funcionamiento del editor.

Aunque J-UML esta diseñado para dar soporte a diagramas de clases, gracias a la arquitectura de netbeans, el nivel de organización de módulos y paquetes, es posible agregar otros modulo que contemplen el resto de diagrama del lenguaje UML, sin la necesidad de crear una aplicación independiente. Para mas detalle ver la Apéndice A.

5.4.2 Estructura de paquetes de J-UML.

La estructura de paquetes necesaria para la creación del J-UML es la siguiente:

- org.udb.juml: contiene archivos xml importantes para la reutilización de plataforma netbeans, estos son generados automáticamente por el Netbeans IDE.
- org.udb.juml.Pics: contiene la simbología grafica de la aplicación.
- org.udb.juml.connections: contiene la clase general para controlar la creación de los widget de conexión.
- org.udb.juml.drawing: contiene las clases necesarias para ajustar los widget de manera visual para que sean acordes a las necesidades del editor.

- Org.udb.juml.drawing.tools: contiene las clases para adaptar los widget en componentes de diagrama, basado en el lenguaje UML.
- org.udb.juml.editor: contiene las clases para la construcción de la interfaz grafica del editor. La clase más importante de este paquete es jumlTopComponent una instancia de TopComponent, consiste en un panel de java que permite la construcción de la interfaz grafica.
- org.udb.juml.file: contiene las clases necesaria para el almacenamiento en archivos de los diagramas creados en J-UML.
- org.udb.juml.generator: es la encargada de la generación de código en archivos con extensión .juml.
- org.udb.juml.tools: contiene las clases que dan soporte ha toda la funcionalidad de J-UML.

Todos estos paquetes están almacenados en los paquetes de origen (Source Package) del modulo juml. Esta misma organización se encuentra almacenada en disco duro.

5.4.3 Implementación del área de diseño.

Para el área de diseño fue necesaria la creación de una subclase llamada jumlTopComponent que se deriva de la clase TopComponent. Este consiste en un panel de java con las capacidades para crear una ventana de Windows y adaptarla acorde a los requisitos del usuario. Además en esta clase se agregan la barra de menú y paleta de elementos.

El área de diseño es la zona central donde se construirán los diagramas de clases. Esta área es un widget especial conocido como Scene (Escena). Este actúa como contenedor de todos los widget que se coloque dentro de el.

Construir un diagrama de clases en J-UML es realmente crear un árbol de widgets los cuales tendrán como padre común al widget Scene. A medida se agregan más elementos al diagrama, mayor cantidad de hijos va adquiriendo en la escena. Para J-

UML la escena es una clase llamada UDBScene que es una subclase de la clase Scene.

UDBScene puede contener tantos hijos como la memoria los permita, pero estos deben ser colocados en un tipo especial de widget conocido como LayerWidget. Los LayerWidgets permiten crear capas para proporcionar algún nivel de acceso. Los tres tipos capas que se deben crear son:

- mainLayer: es la capa principal en donde se colocaran los widgets.
- interractionLayer: es la capa que proporciona movilidad a los widgets dentro del área de diseño (escena).
- connectionLayer: es la capa que permite crear las conexiones entre widgets.

5.4.4 Implementación componente de clases.

Una clase en J-UML son instancias de la clase UMLClassWidget, que es subclase de Widget. Su estructura consiste lo siguiente:

- LabelWidget: un widget de que contendrá una cadena de caracteres con el nombre de la clase.
- Dos widgets que contendrán labelWidget para representar los atributos y comportamientos de las clases a modelar.
- WidgetActions son widgets especiales que permiten implementar funciones de edición de clases tales como agregar, eliminar o editar atributos y comportamientos.

Para identificar el grado de acceso a variables y métodos se utiliza una simbología de colores; estos colores no son estándar de UML, es simplemente una contribución como programadores de la aplicación para facilitar su identificación.
Los colores son los siguientes:

- Rojo: acceso private.
- Amarillo: acceso protegido.
- Verde: acceso publico.

5.4.5 Implementación de componentes de notas.

Las notas son instancias de la clase UMLNoteWidget. Tanto las clases y notas son contenidos en la capa mainLayer (capa principal).

5.4.6 Implementación de los componentes de conexiones.

Las conexiones son instancias de la clase UDBConnectionWidget, para su colocación en los diagramas se requirió:

- Un connectionLayer es un widget que sirve de capa para colocar el widget.
- Una interactionLayer es un widget que permite crear una relacion de clase-clase o clase-nota.

Las relaciones entre clases a las que J-UML soportara son:

- Dependencia.
- Asociación.
- Generalización.
- Composición.
- Agregación.

Las relaciones entre notas y clases se utiliza la relación de notizacion.

5.4.7 Implementación de clases entidad.

Funciones de guardar y guardar como.

Para poder almacenar los archivos se utilizo archivos xml. El formato fue diseñado para que sea capas de identificar todo los elementos notacionales colocados en el área de diseño. Las clases encargadas para esta funcionalidad son:

- FindClassNote.java.
- UDBSceneDataSerializer.java.

Formato de guardado en xml.

Estructura encabezado y final de archivo.

En el encabezado es necesario colocar la etiqueta que identifique la versión y el codificado, se agregan las etiquetas <Scene> y </Scene> como delimitadores de cuerpo de trabajo.

Se incluyen dentro de la definición de Scene las etiquetas:

- classIDcounter: contador de clases creadas.
- connectIDcounter: contador de conecciones.
- noteIDcounter: contador de notas.
- Ids: identificador general.

Estructura de nota.

Para este elemento se ocupa la etiqueta <note *definición de nota* />; las etiquetas que se utilizan son:

- id: identificador de instancia a la que pertenece.
- idNote : identificador de nota.
- text: contenido textual de la nota.
- x, y : coordenada de ubicaron en el área de trabajo.

Estructura de clases.

Las clases utilizan las etiquetas <class> y </class>; las etiquetas que se utilizan son:

- id: identificador de instancias a la que pertenece.
- idClass: identificador de clase.
- name: nombre de la clase.
- numPropiedades: numero de propiedades que posee
- numMetodos: numero de métodos que posee.
- <property />: especifican una propiedad de la clase, puede haber tantas como el usuario desee.

- <metodo: />: especifican un método de la clase, puede haber tantas como el usuario desee.

Estructura de conexiones.

Las conexiones ocupan la etiqueta <connect />; las etiquetas que se utilizan son:

- id: identificador de instancia a la que pertenece.
- idConnection: identificador de conexión.
- source: identificador de elemento origen.
- target: identificador de elemento destino.

```
<?xml version="1.0" encoding="UTF-8"?>
 //Identificador de inicio de escena.
<Scene classIDcounter="" connectIDcounter="" ids="" noteIDcounter="" version="">
     //Identificador de notas.
    <note id="" idNote="" text="" x="" y=""/>
     //Identificador de clases.
    <class id="" idClass="" name="" numMetodos="" numPropiedades="" x="" y="">
        <property acces="" id="" name="" tipo=""/>
        <metodo acces="" id="" name="" retorno=""/>
    </class>
    .
    .//Se colocaran el resto de clase contenidas en el diagrama.
    .
    //Identificadores de conecciones.
    <connect id="" idConnection="" source="" target=""/>
    .
    .//Se colocaran el resto de conexiones contenidas en el diagrama.
    .
//Identificador de inicio de escena.
</Scene>
```

Figura 4.6 Esquema xml para para guardar diagramas.

Los archivos que contengan diagramas generados en J-UML tienen extensión .juml. El proceso de carga de un diagrama al editor es el proceso inverso al guardado, un archivo extensión .juml tiene una estructura interna como la que se presenta en la figura 4.6.

5.4.8 Funciones de generación de código.

J-UML posee la capacidad de generar código en java en archivos en texto plano. Para lograrlo se analizan los elementos del diagrama y se seleccionan aquellos que

permitan la generación de código. Las clases encargadas de realizar estas acciones son:

- Archivo JavaCode.java: esta clase se encarga de contener palabras reservadas y operadores del lenguaje java, que se irán agregando conforme se analizan los diagramas.
- Archivo GeneradorCode.java: se encarga de interpretar los diagramas y convertirlos en archivos extensión .java.

El proceso que se sigue para generar el código es el siguiente:

1. Agregar comentarios de cabecera al archivo.
2. Crear implementación de clase con herencia si existiera.
3. Crear propiedades generadas de las conexiones.
4. Crear propiedades de clases.
   4.1 Especificar acceso de propiedades.
   4.2 Especificar tipo de dato.
   4.3 Especificar el nombre.
5. Crear métodos de clases.
   5.1 Especificar el acceso del método.
   5.2 Especificar el tipo de dato de retorno.
   5.3 Especificar nombre del método.
   5.4 Especificar los parámetros de entrada si lo hubiera.
   5.5 Especificar el cuerpo.
6. Cerrar la definición de la clase.
7. Generar el archivo extensión .java.

5. 5 Pruebas.

En esta sección se muestran la funcionalidad de J-UML, para ello se necesita un modelo de prueba.

5.5.1 Modelo de prueba.

Como modelo de prueba se considero crear un ejemplo donde se implemente un diagrama en J-UML. Esta prueba consiste en mostrar los diferentes tipos de componentes notacionales que el editor es capas de contener, sus relaciones y generación de código.

5.5.2 Prueba de interfaz grafica.

En la figura 4.7 se muestra la ventana principal de J-UML, con la cual el usuario interactúa directamente con el sistema.

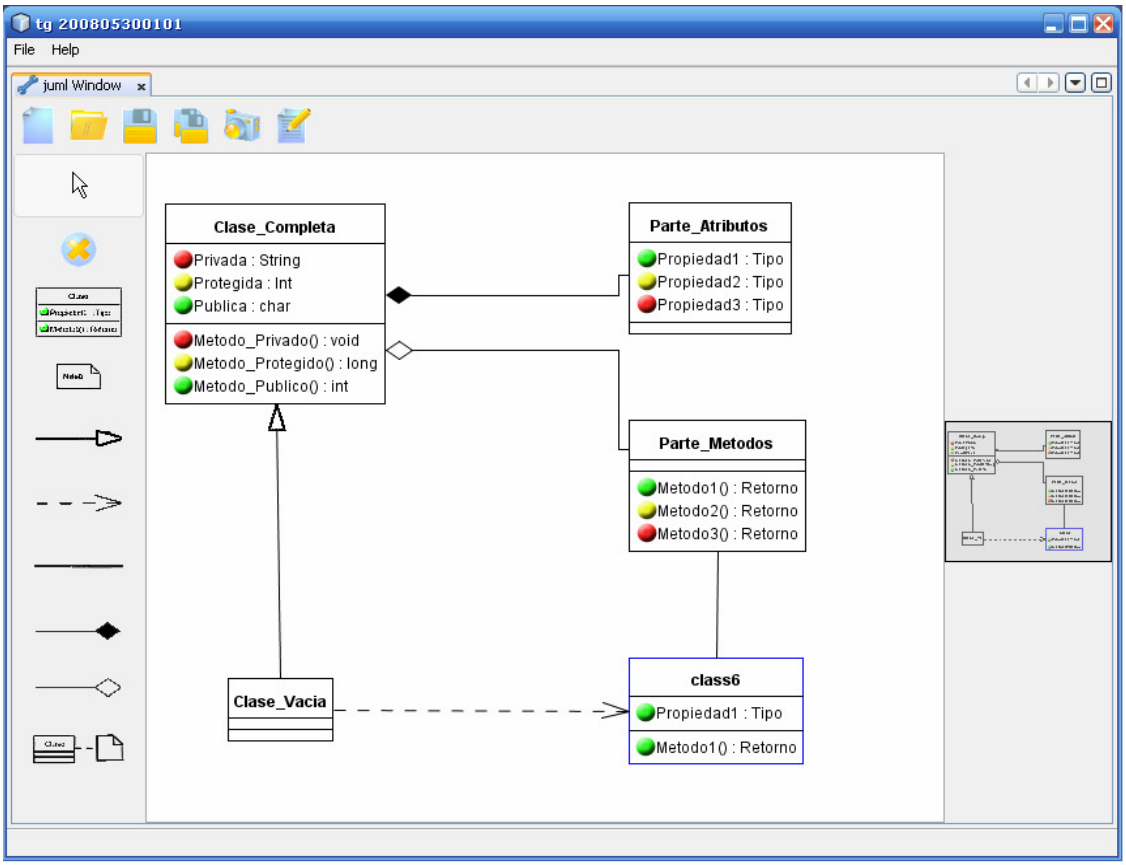


Figura 4.7 Ventana principal de diagramas de clases.

Además en la figura 4.7 se observa un diagrama de clases conteniendo los posibles casos de componentes de clases. Estos son los siguientes:

- Clases con atributos y comportamientos.
- Clases con solo atributos.
- Clases con solo comportamientos.
- Clases sin atributos y comportamientos.
- Clases sin especificiones del usuario.
- Todos lo tipos de relaciones entre clases soportadas por J-UML.
- Componente de Nota.

5.5.3 Prueba de exportar diagramas.

La figura 4.8 muestra como se observa un diagrama exportado por J-UML.

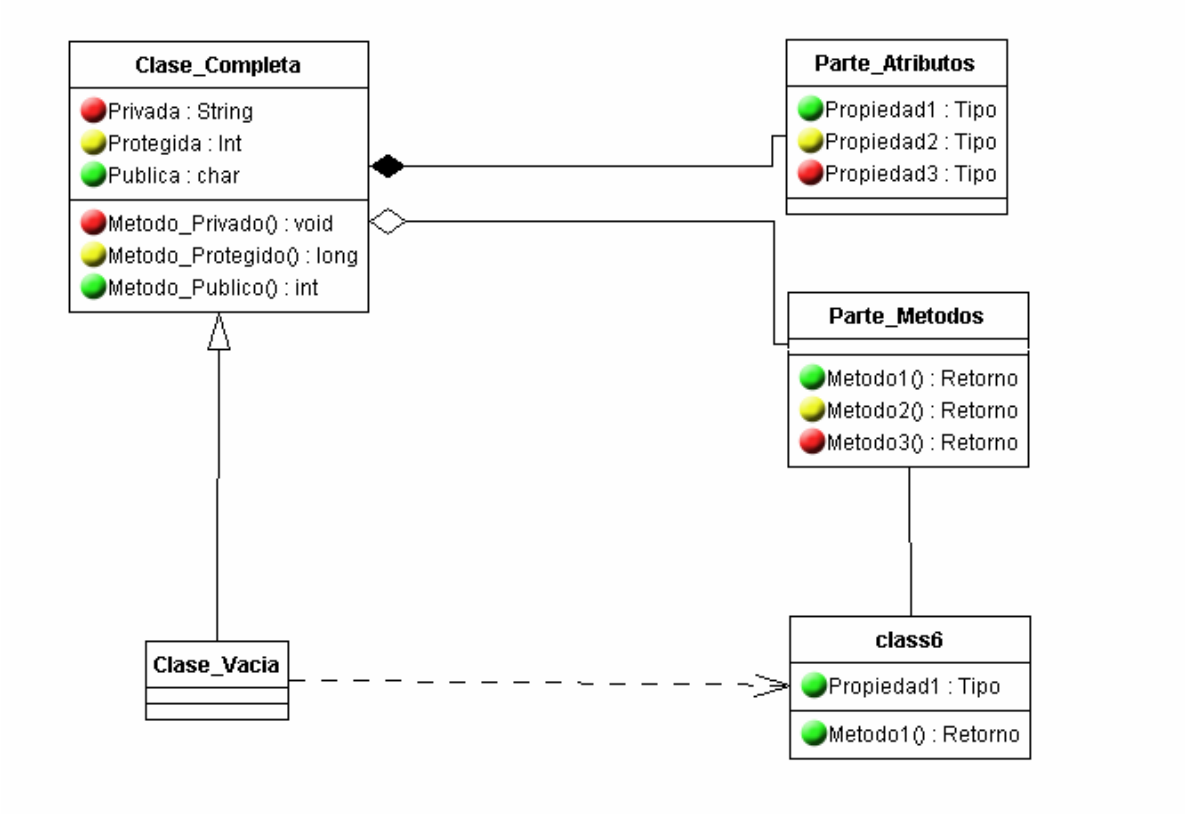


Figura 4.8 Diagrama en formato png generado por J-UML.

5.5.4 Prueba de Generación de Código.

J-UML hace la generación de código al crear archivos con extensión java, estos se almacenan en el directorio especificado por el usuario. La figura 4.9 muestra la estructura de generación de código para una clase en J-UML.

```
    /*******************************************************/
/*  ******************************************************/
/*  **     UNIVERSIDAD DON BOSCO - J-UML Editor 0.7   ***/
/*  **     Autores :                                  ***/
/*  **     Jorge Miguel Erazo Melara                  ***/
/*  **     Adis Reinaldo Meléndez Escobar             ***/
/*  ******************************************************/
/*  ******************************************************/

/*****************************************************************************/
/**Código generado por la herramienta J-UML Editor 0.7                      **/
/**Trabajo de graduación para optar al grado de Ing. en CC. de la Computación**/
/*****************************************************************************/
/*class  = Clase Completa*/
class Clase_Completa {
    Parte_Metodos parte_metodos_0;
    Clase_Vacia clase_vacia_1;
    Parte_Atributos parte_atributos_2;
    private String Privada;
    protected Int Protegida;
    public char Publica;

    private void Metodo_Privado () {
        /*Agrege su código aquí*/
    }

    protected long Metodo_Protegido () {
        /*Agrege su código aquí*/
    }

    public int Metodo_Publico () {
        /*Agrege su código aquí*/
    }
}
```

Figura 4.9 Estructura de generación de código.

CONCLUSIONES.

El prototipo de editor J-UML es el primer esfuerzo para desarrollar una herramienta CASE para la Universidad Don Bosco; con el propósito de proporcionar a maestros y alumnos de Ingeniería en Ciencias de la Computación de una herramienta de desarrollo de software.

El proceso de desarrollo de J-UML se facilito y agilizo por el uso del lenguaje UML y el Proceso Unificado, debido ha que es posible obtener una marco de desarrollo en poco tiempo y entendible tanto a usuarios, programadores y desarrolladores.

J-UML es capas de modelar diagrama de clases de manera rápida y sencilla, además permite la generación de código fuente ejecutable en lenguaje JAVA. Sin embargo no es una herramienta completa, por esta razón el editor se encuentra a nivel prototipo.

La herramienta provee un entorno grafico desarrollado en lenguaje JAVA bajo el marco de trabajo de Netbeans, debido a su filosofía modular de trabajo y sus librerías de modelado de grafos. Esto significa que la herramienta es capas de ampliar sus capacidades de modelado al incluir módulos que den soporte a otros diagramas de UML. Por otra parte, J-UML es multiplataforma, permitiendo ser utilizado en sistemas operativos Windows, Linux o que soporten la maquina virtual de JAVA.

Con este trabajo se deja la documentación necesaria a programadores y desarrolladores para realizar los siguientes puntos:

- Convertir el prototipo en una herramienta CASE formal para el desarrollo de software.
- Dar soporte a todos los diagramas de UML y elementos notacionales.
- Ampliar la gama de lenguajes de programación para generar código fuente.
- Desarrollo de software basado en metodología de desarrollo rápido como lo es el Proceso Unificado.

APENDICE A.

MANUAL DEL PROGRAMADOR.
NETBEANS 6.0.

Introducción.

En este apartado se explican la terminología y metodología de Netbeans para la creación de software. Este apartado puede ser utilizado como manual para orientar al programador.

NetBeans.

Es mejor conocido como un ambiente integrado de desarrollo de aplicaciones en JAVA, que cuenta con arquitectura modular y extensible marco de trabajo.

Plataforma Netbeans.

Se le conoce como Plataforma NetBeans al marco de trabajo (framework) para aplicaciones de cliente rico (rich client), con el propósito de crear software que se *escriba una vez y funcione en cualquier sistema operativo*. Se le conoce como aplicaciones de cliente rico (rich client application) a un a pieza de software donde una buena porción de las características del sistema trabaja en el sistema local del usuario. Es el término de NetBeans como sinónimo de aplicaciones de escritorio.

Entre los beneficios que provee Netbeans en su plataforma estan:

- NetBeans es gratis y su código es libre de reutilizar.
- NetBeans es framework maduro con gran cantidad de características para el desarrollo de aplicaciones.
- NetBeans se basa en el pensamiento *"escrito una vez, funciona donde sea"*, lo cual permite desarrollar aplicaciones independiente del sistema operativo del usuario.
- NetBeans es una tecnología basada en estándares y fuente abierta (open source).
- NetBeans cuenta con una gran comunidad de desarrolladores.

Para el desarrollo de aplicaciones NetBeans provee las siguientes características:

- Un sistema Windows que simplifica la manipulación de múltiples componentes en una misma área de trabajo.
- Un sistema Action que facilita la instalación y desinstalación de menús, barra de herramientas.
- Un mecanismo de actualización dinámica de los elementos de las aplicaciones.
- Una arquitectura extensible para las aplicaciones al usar técnicas de programación modular.

La arquitectura modular de netbeans convierte a las aplicaciones en un conjunto de pequeño, separados y asilados módulos, conteniendo su respectiva funcionalidad.

Estos pueden ser de tres tipos:

- Interfase de Usuarios. (End-Users Interface).
- Librería Simple.
- Modulo de Librerías.

Este sistema de módulos es un contenedor en tiempo de ejecución que asegura la integridad de estas piezas. La utilización de módulos lo realiza por medio de las llamadas dependencias de modulo. El propósito de la modularidad es acercar que la aplicación se convierta en interacción entre sistemas, en vez de partes de subsistemas.

Sistema de Windows de NetBeans.

J-UML es una aplicación de escritorio desarrollado en plataforma NetBeans. Para ello fue necesario utilizar Windows System API (Sistema de ventanas), cuyas clases están contenidas en el paquete `org.openide.windows`.

NetBeans cuenta con una interfase GUI, como componente de Windows System, la cual proporciona al desarrollador todo el conjunto de herramientas para el desarrollo de aplicaciones de escritorio utilizando la librería Swing. Con base en lo anterior, el

desarrollo del prototipo J-UML consistió en agregar los componentes swing (botones, menus, texto, etc) al contenedor GUI, el Windows System se encarga de la generacion del codigo necesario, permitiendo enfocar nuestro esfuerzo en la funcionalidad del sistema.

Clases en Windows System API.

Las clases que NetBeans proporciona en su Windows System API son:

- TopComponent: es un JPanel (Panel en Java) con algunos metodos adicionales que permiten el manejo e iteración con Windows System.
- Mode: es una clase de acoplamiento (Docking mode), permite a los componentes acoplarse en las ventanas.
- Window Manager: se encarga de manejar los estados de la aplicación.

Proceso de creación del modulo J-UML en NetBeans.

El prototipo J-UML es capaz de crear diagramas de clases, sin embargo estos no son los únicos diagramas que contempla UML. Por lo tanto J-UML representara el primer modulo de un sistema mas completo que abarque toda la metodología de UML, bajo la arquitectura modular de NetBeans.

Por lo tanto, es necesario crear un tipo especial de modulo capaz de contener otros módulos. En NetBeans esto es posible por medio del *Module Suite,* el cual es un contenedor de módulos que representan piezas funcionales del sistema.

1. Abrimos **NetBeans IDE 6.01** y seleccionamos **File → New Project**

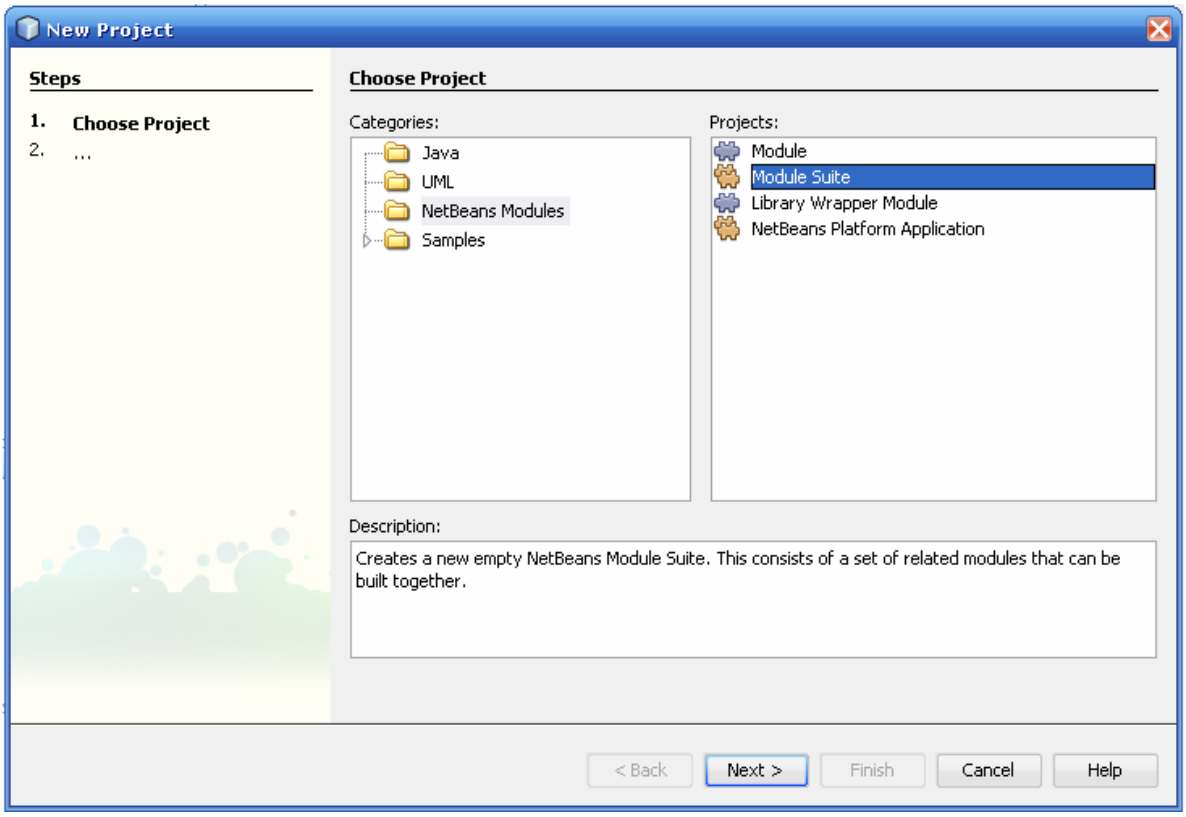


Figura A1

2. En la figura A1 seleccionamos en Categories → **NetBeans Modules**, en Projects → **Module Suite**.

3. Luego nos aparece la ventana de la Fig. A2 en donde especificamos el nombre del proyecto, así como su localización y ubicación en disco. Y presionamos Finish.

4. La Fig. A3 muestra lo que genera al completar el paso anterior. Para agregar un modulo, simplemente hacemos clic derecho en **Modules** y seleccionamos **Add New**.

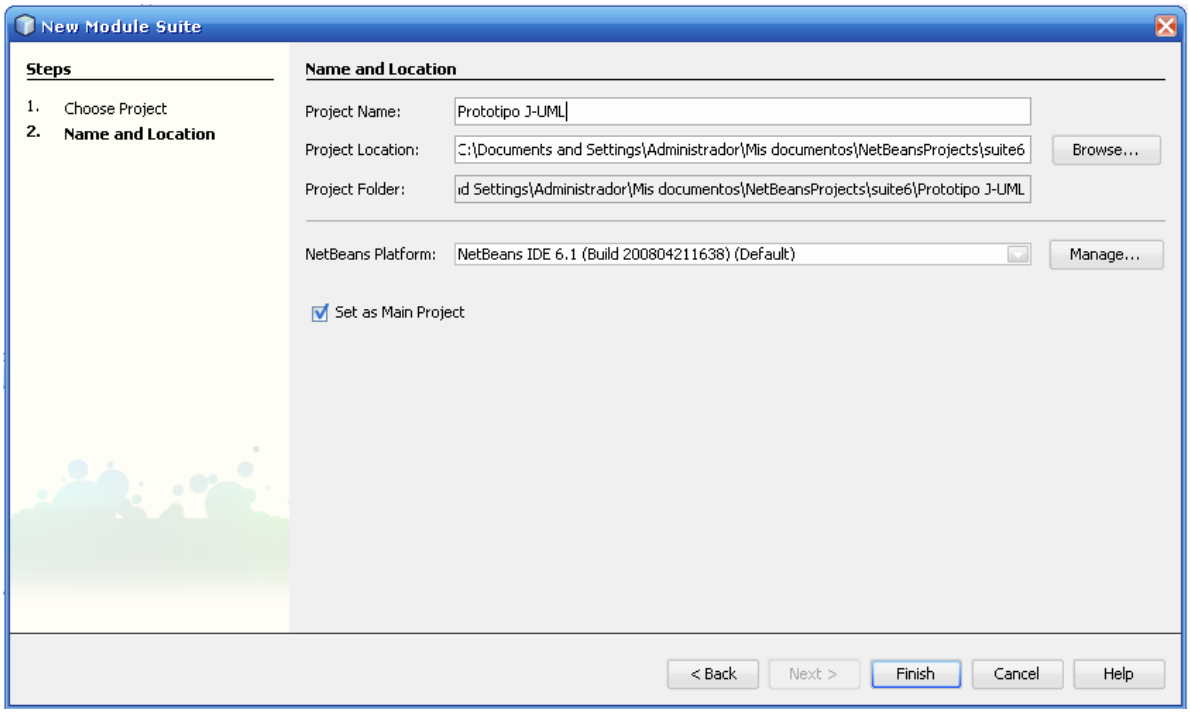


Figura A2

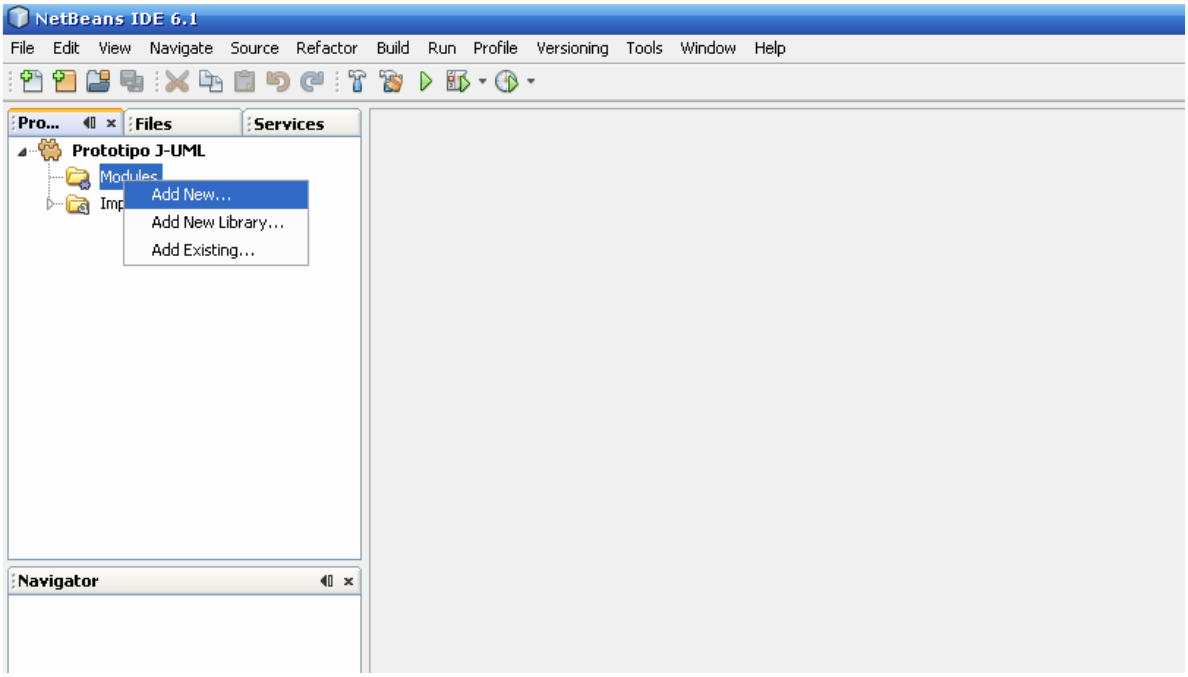


Figura. A3

5. La Figura A4 terminado el paso 4 se nos muestra una ventana donde agregamos el nombre del modulo, asi como su localizacion y ubicacion en disco.

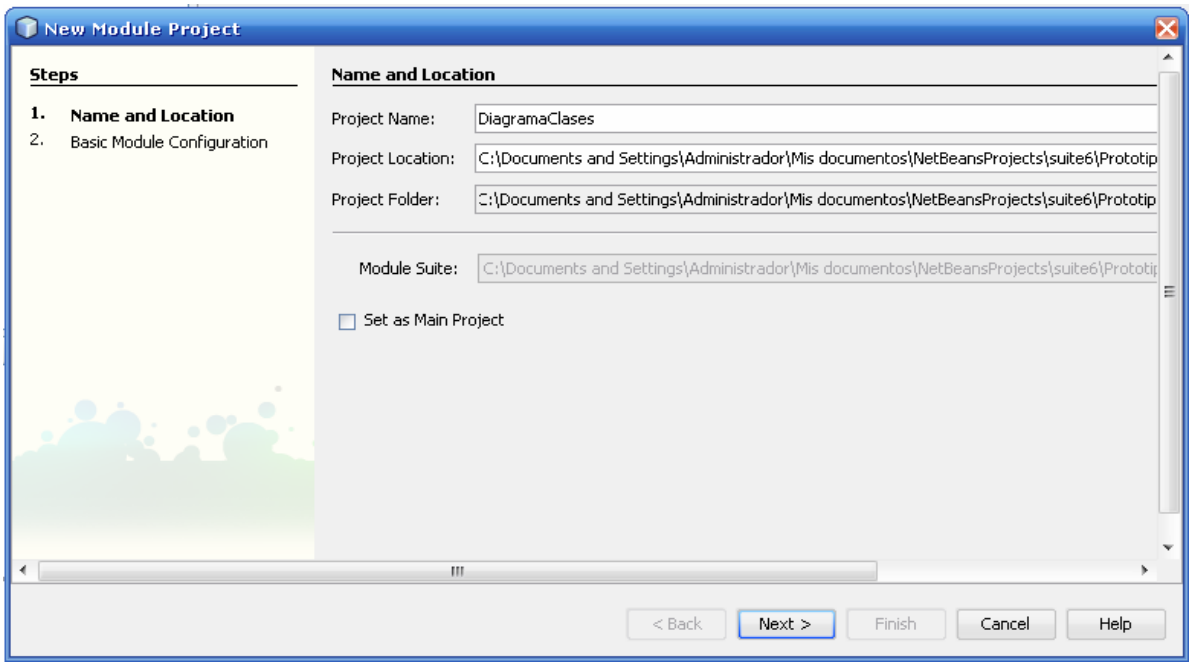


Figura A4

6. Por ultimo se especifica la configuración básica del modulo, que consiste en crear la paquete en java y archivos de configuración propios de Netbeans. Como se muestra en la figura A5.

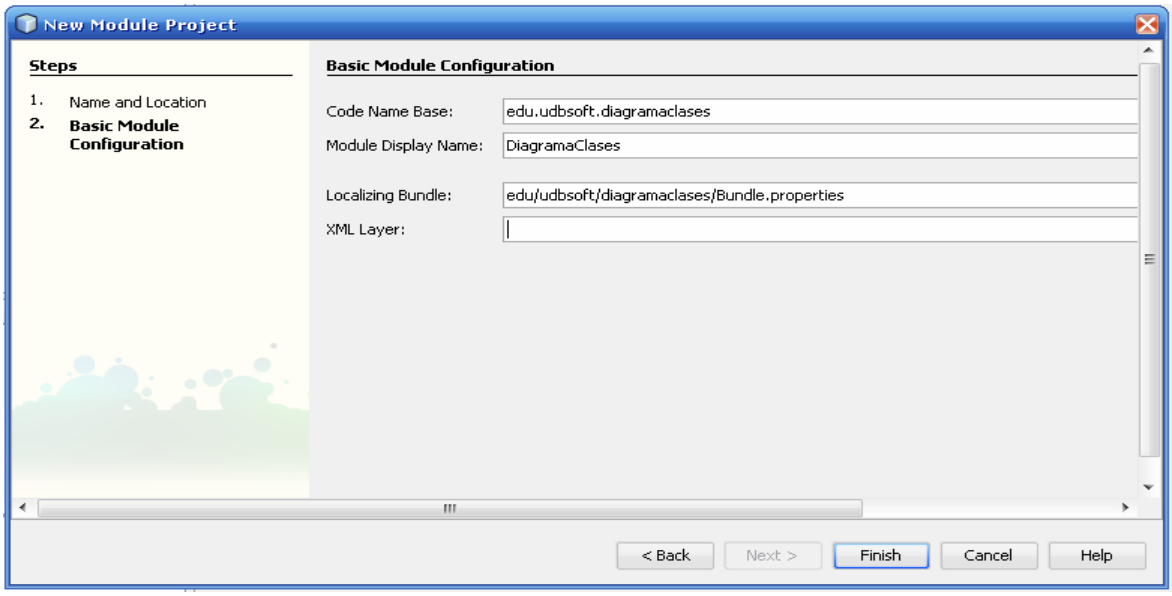


Figura A5

Terminado el procedimiento anterior se tiene la arquitectura base para la creación del J-UML (así como cualquier otra aplicación modular) bajo el entorno de NetBeans.

Usando la clase TopComponent de J-UML.

Creada la arquitectura de todo el sistema, es necesario crear una ventana principal, la cual se encarga de almacenar todos los elementos que contara el usuario para crear los diagramas en el J-UML.

Creado el modulo, aparecerá una ventana como se muestra en la Fig. A6

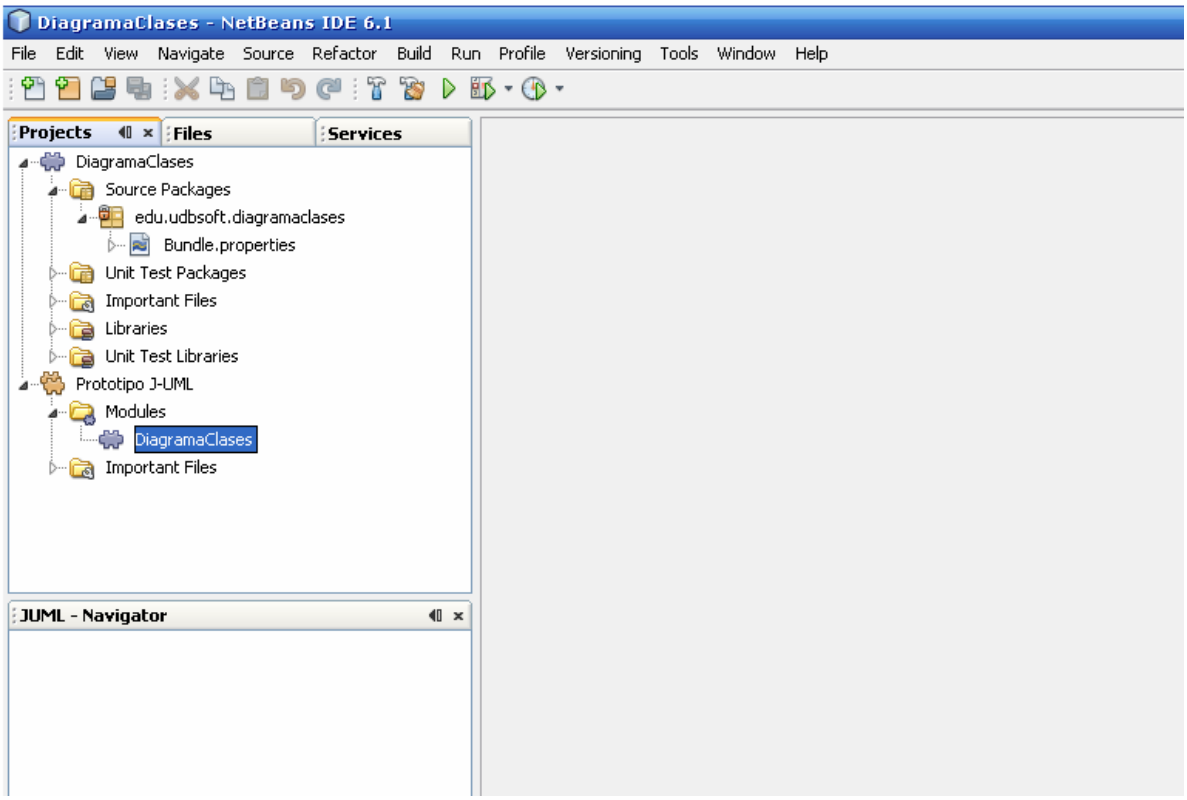


Fig. A6

En la pestaña **Projects** se puede observar que se encuentran creados el modulo suite (contenedor de modulos) y un modulo llamado DiagramaClases. Este último contiene una estructura interna divida en:

- Source Package: contiene todos los archivos con extensión java para implementar la funcionalidad del sistema.
- Unit Test Package: permite la utilización de paquetes de prueba.

- Important Files: contiene archivos xml generados por el IDE para el manejo de funciones especiales.
- Libraries: muestra las librerías que el IDE ocupa para su funcionalidad.
- Unit Test Libraries: espacio para prueba de librerías.

Dentro de Source Package esta creado el paquete llamado `edu.udbsoft.diagramadeclase`, el nombre depende del desarrollador, sin embargo la sintaxis que NetBeans recomienda es separado por puntos de la siguiente manera:

`TipoDeOrganización.NombreDeOrganizacion.NombreDeProyecto.`

Para crear la ventana principal se hace click derecho en el paquete, como se muestra en la figura A7.

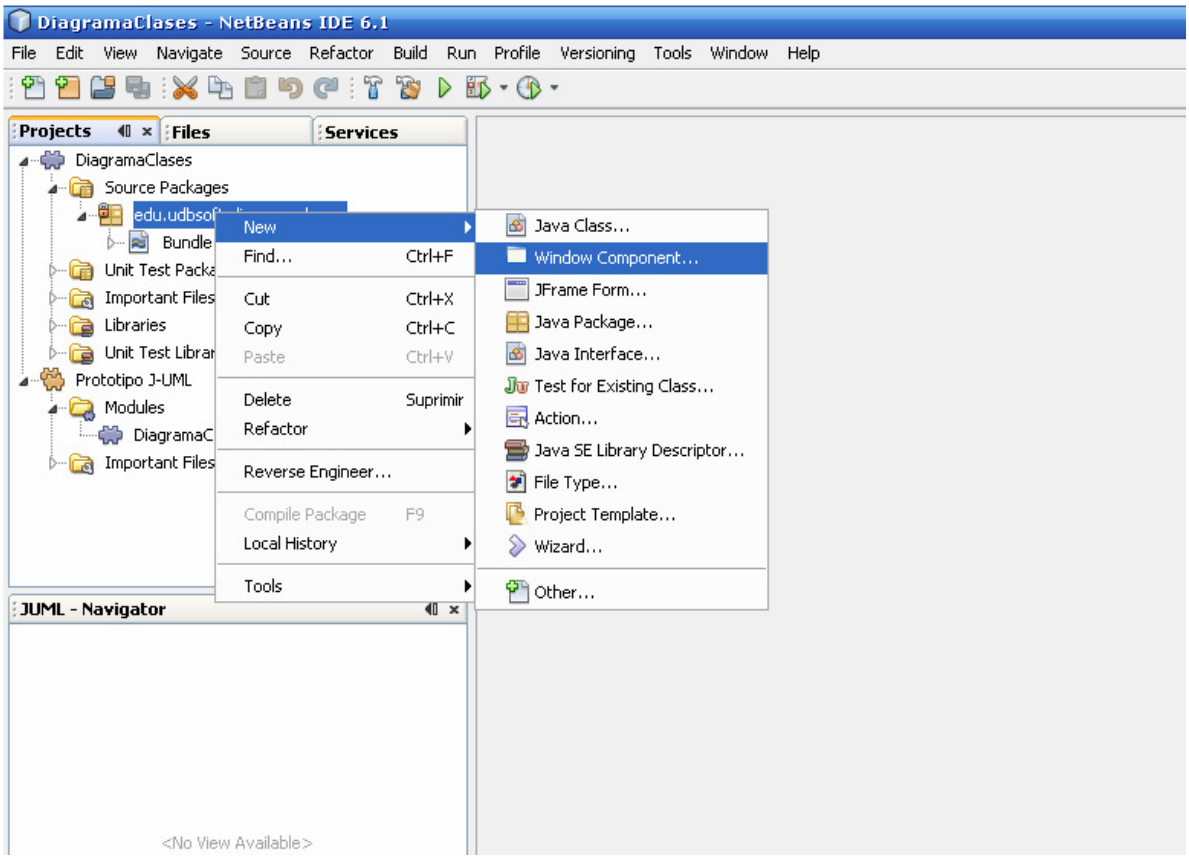


Figura A7.

Seleccionamos Windows Component. (Componente de Ventana).

Windows Component es una subclase de TopComponent que nos permite editar y agregar elementos que confomaran la ventana principal del prototipo J-UML. Luego se selecciona el tipo de acoplamiento (utilizando la clase Mode), el acoplamieto significa en que área del IDE de netbeans se visualizara la aplicación. Como se observa en la figura A8.

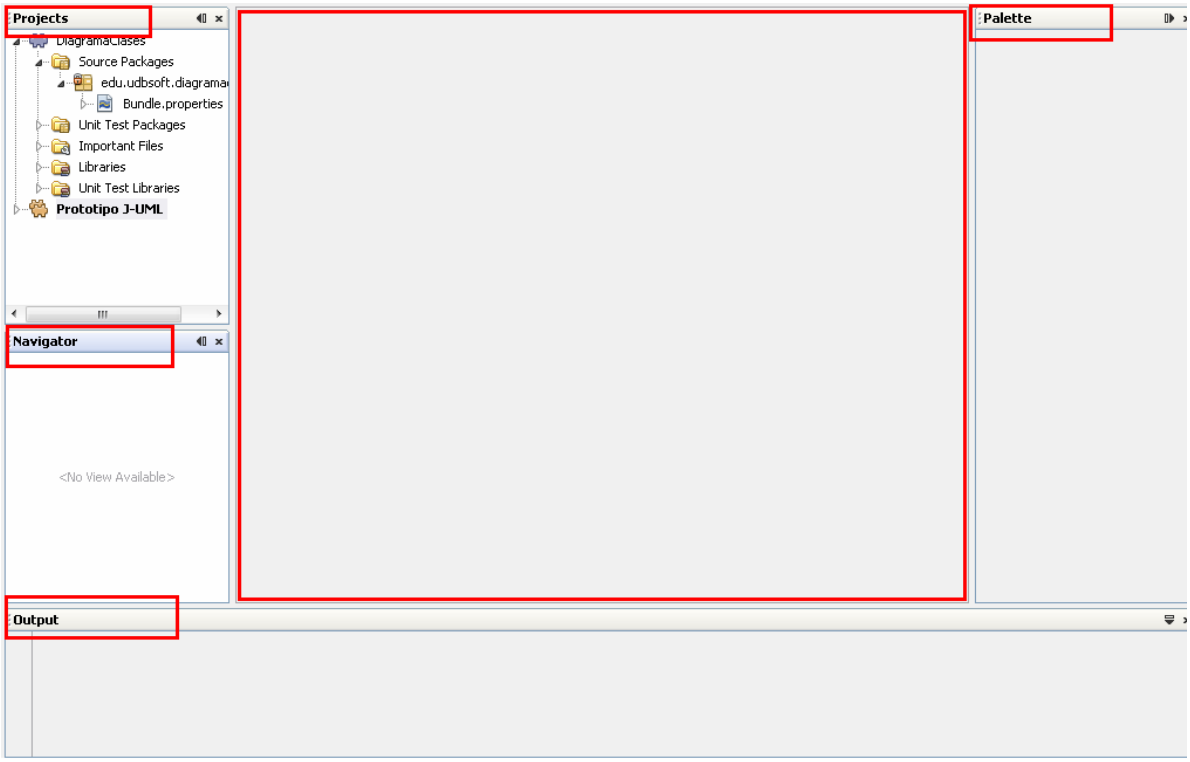


Fig A8. Las áreas marcadas en rojo son las diferentes áreas de visualización. La parte central es la de editor, donde J-UML de visualizara.

Por último se le asigna un prefijo de nombre a la aplicación, es sólo un identificador que se concatena con los archivos que generara de manera automática el IDE. El nombre que ocuparemos para esta demostración será Demo.

Cuando se realiza el procedimiento de crear un Windows Component, dentro del Source Package, en el paquete `edu.udbsoft.diagramasclases` el IDE creara los siguientes archivos:

- Bundle.properties: es una archivo xml con información de nombre y descripción del modulo.
- DemoAction.java: clase encargada de manejar las llamadas (o acciones) de parte del usuario.
- DemoTopComponent.form: este archivo nos proporciona un formulario para trabajar con el diseño de la ventana principal, bastara con solo seleccionar de manera visual los elementos (botones,paneles, menus, etc) y colocarlos dentro del formulario.
- DemoTopComponent.java: este archivo contiene el codigo fuente de la de la subclase TopComponent.
- DemoTopComponentSettings.xml: continene información que permite la persistencias de los datos en la aplicación.
- DemoTopComponentWstcref.xml: la información de este archivo permite enlazar la información de DemoTopComponenentSetting con una particular definición de mode.
- layer.xml: este es un archivo de configuración que se encarga de instalar el (o los) modulo(s).

Con base a todo lo anterior descrito, estos fueron los pasos que se necesitaron para la creación de la arquitectura básica de J-UML bajo el entorno NetBeans. El IDE ha proporcionado de manera automática la organización de clase y paquetes del software, así como las de los archivos y fólderes en el disco duro. Provee de herramientas para el diseño de aplicaciones de escritorio basada en Swing, generando de manera automática el código necesario. Además de permitir la reutilización de la arquitectura y funcionalidades propias de NetBeans.

NetBeans: Visual Library 2.0.

NetBeans cuenta con librerías para crear grafos de visualización. Esto lo realizaba a través de la librería Graph Library 1.0. Actualmente NetBeans IDE en sus versiones recientes esta introduciendo la librería Visual Library 2.0, que es una mejora de la Graph Library, con la que pretende dar soporte al modelado para propósitos generales.

Esta librería hace el modelado construyendo y modificando un árbol de elementos visuales llamados Widgets. Donde la raíz del árbol es un widget especializado llamado Scene, el cual se encargara de contener todos los elementos. El estilo de programación es similar al de Swing.

Elementos básicos para la creación de Diagramas de Clases.

Todos estos elementos se encuentran en el paquete:

org.netbeans.api.visual.widget

Widget: es el componente visual básico de la librería, contiene información de localización, límites, tamaños, capas, bordes, etc.

Scene: es un widget que representa la raíz de una estructura jerárquica de widgets. Este puede ser extendido como: **GraphScene**, **ObjectScene** y **GraphPinScene**. Estas extensiones solo agregan funcionalidad adicional a las aplicaciones para modelado orientado a objetos o grafos.

Existen dos tipos de widget básicos para propósitos generales:

- **LabelWidget**: representa una línea de texto.
- **ImageWidget**: representa una imagen.

Layouts: cada widget posee una capa que permite colocar otros widget como **LabelWiget**, **ImageWidget**, etc. A través de la clase **LayoutFactory** es posible crear una alineación ordenada de los elementos.

ConnectionWidget: este es un elemento especial de widget, permite representar una conexión entre origen y destino. Posee la propiedad para configurar la forma de la flecha tanto de origen como destino.

Actions: o **ActionFactories** permiten la interacción dinámica entre los widget y el usuario. Permiten agregar, borrar y editar los widgets.

Para la construcción de diagramas es necesario básicamente conocer estos conceptos, existen una gran cantidad de funciones e interfaces que pueden ser vistas en la documentación de Visual Library.

APENDICE B

DETALLE DE CASOS DE USO.

Listado de casos de usos reales.

JUML Caso-Uso 01. Creación de componente de diagrama de clases.
Objetivo: El alumno agrega un elemento notacional al diagrama de clases.
Pre-Condición: El editor debe estar iniciado o pulsar sobre el ítem 'Nuevo'.

Flujo de sucesos:
Flujo principal.

1. El alumno pulsa sobre el ítem del elemento notacional que desea agregar.
2. El alumno hace clic sobre el área de diseño y coloca el elemento notacional.
3. JUML genera el elemento notacional seleccionado.
4. El alumno reacomoda el elemento notacional donde desee dentro del área de diseño.

Post-Condición: elemento notacional agregado en el diagrama de clases.

JUML Caso-Uso 02. Creación de componente notacional de clase.
Objetivo: Colocar un elemento notacional de clase en el área de diseño.
Pre-Condición: El editor debe estar iniciado o pulsar sobre el ítem 'Nuevo'.

Flujo de sucesos:
Flujo principal:

1. El alumno pulsa sobre el ítem que representa las clases de la barra de elementos.
2. El alumno coloca la clase haciendo clic sobre el área de diseño.
3. JUML genera la representación de clase en el área de diseño.
4. El alumno acomoda la clase dentro del área de diseño.

Post-Condición: nueva clase es creada en el diagrama.
Caso de uso base: **JUML Caso-Uso 01**.

JUML Caso-Uso 03. Creación de componente notacional de nota.
Objetivo: colocar un elemento notacional de nota en el área de diseño.
Pre-Condición: El editor debe estar iniciado o pulsar sobre el ítem 'Nuevo'.

Flujo de sucesos:
Flujo principal:

1. El alumno pulsa sobre el ítem que representa las notas en la barra de elementos.
2. El alumno coloca la nota pulsando en el área de diseño.
3. JUML genera la representación de nota en el área de diseño.
4. El alumno acomoda la nota dentro del área de diseño.

Flujo alternativo:
En el paso 3. El alumno hace doble clic sobre la nota para agregar información.

Post-Condición: nueva nota es creada en el diagrama.
Caso de uso base: **JUML Caso-Uso 01**.

JUML Caso-Uso 04. Creación de componente notacional de relación.
Objetivo: colocar un elemento notacional de relación entre clases, en el área de diseño.
Pre-Condición: se requiere de dos elementos de clase creados.

Flujo de sucesos:
Flujo principal:

1. El alumno pulsa sobre el ítem que representa la relación en la barra de elementos.
2. El alumno selecciona la clase origen, mantiene pulsado el clic izquierdo hasta seleccionar la clase destino.
3. JUML crea la relación, reconociendo el origen y destino.

Post-Condición: nueva relación entre clases es creada en el diagrama.
Caso de uso base: **JUML Caso-Uso 01**

JUML Caso-Uso 05. Creación de componente notacional de asociación.
Objetivo: colocar un elemento notacional de asociación entre clases, en el área de diseño.
Pre-Condición: se requiere de dos elementos de clase creados.

Flujo de sucesos:
Flujo principal:

1. El alumno pulsa sobre el ítem que representa la asociación en la barra de elementos.
2. El alumno selecciona la clase origen, mantiene pulsado el clic izquierdo del Mouse hasta seleccionar la clase destino.
3. JUML crea la relación de asociación, reconociendo el origen y destino.

Extensión:

1. El alumno asocia una clase con una nota.
2. El alumno pulsa la clase de origen, mantiene pulsado el puntero y lo arrastra a un elemento de nota destino.
3. JUML crea la relación de notizacion, reconociendo la clase origen y nota destino.

Post-Condición: nueva relación entre clases es creada en el diagrama.
Caso de uso base: **JUML Caso-Uso 04.**

JUML Caso-Uso 06. Creación de componente notacional de generalización.
Objetivo: colocar un elemento notacional de generalización entre clases, en el área de diseño.
Pre-Condición: se requiere de dos elementos de clase creados.

Flujo de sucesos:

Flujo principal:

1. El alumno pulsa sobre el ítem que representa la generalización en la barra de elementos.
2. El alumno selecciona la clase general, mantiene pulsado el clic izquierdo del Mouse hasta seleccionar la subclase.
3. JUML crea la relación de generalización, reconociendo la clase general y su subclase.

Post-Condición: nueva relación de generalización entre clases es creada en el diagrama.

Caso de uso base: **JUML Caso-Uso 04.**

JUML Caso-Uso 07. Creación de componente notacional de composición.

Objetivo: colocar un elemento notacional de composición entre clases, en el área de diseño.

Pre-Condición: se requiere de dos elementos de clase creados.

Flujo de sucesos:

Flujo principal:

1. El alumno pulsa sobre el ítem que representa la composición en la barra de elementos.
2. El alumno selecciona la clase origen, mantiene pulsado el clic izquierdo del Mouse hasta seleccionar la clase.
3. JUML crea la relación de composición, reconociendo la clase origen y destino.

Post-Condición: nueva relación de composición entre clases es creada en el diagrama.

Caso de uso base: **JUML Caso-Uso 04.**

JUML Caso-Uso 08. Creación de componente notacional de agregación.
Objetivo: colocar un elemento notacional de agregación entre clases, en el área de diseño.
Pre-Condición: se requiere de dos elementos de clase creados.

Flujo de sucesos:
Flujo principal:

1. El alumno pulsa sobre el ítem que representa la agregación en la barra de elementos.
2. El alumno selecciona la clase origen, mantiene pulsado el puntero y lo arrastra hasta seleccionar la clase destino.
3. JUML crea la relación de agregación, reconociendo la clase origen y destino.

Post-Condición: nueva relación de agregación entre clases es creada en el diagrama.
Caso de uso base: **JUML Caso-Uso 04.**

JUML Caso-Uso 09. Creación de componente notacional de dependencia.
Objetivo: colocar un elemento notacional de dependencia entre clases, en el área de diseño.
Pre-Condición: se requiere de dos elementos de clase creados.

Flujo de sucesos:
Flujo principal:

1. El alumno pulsa sobre el ítem que representa la dependencia en la barra de elementos.
2. El alumno selecciona la clase origen, mantiene pulsado el puntero y lo arrastra hasta seleccionar la clase destino.
3. JUML crea la relación de dependencia, reconociendo la clase origen y destino.

Post-Condición: nueva relación de dependencia entre clases es creada en el diagrama.
Caso de uso base: **JUML Caso-Uso 04**.

JUML Caso-Uso 10. Guardar un diagrama en archivos.
Objetivo: guardar un diagrama en un archivo con extensión juml.
Pre-Condición: se requiere de un diagrama en el área de diseño.

Flujo de sucesos:
Flujo principal:

1. El alumno pulsa el ítem 'Guardar como' en la barra de menú.
2. JUML genera un cuadro para que el usuario especifique el directorio donde desea guardar.
3. El alumno digita el nombre con que desea guardar el diagrama.
4. JUML genera el archivo en el directorio especificado por el alumno con extensión juml.

Post-Condición: un archivo con extensión juml es guardado en un directorio.

Extensión:

1. Guardar en un archivo previamente abierto.
2. El alumno pulsa sobre el ítem 'Guardar' en la barra de menú.
3. JUML guarda los cambios en el mismo archivo con que el alumno trabaja.

JUML Caso-Uso 11. Abrir un diagrama almacenada en archivos.
Objetivo: abrir desde un directorio un diagrama de clases.
Pre-Condición: se requiere de un área de diseño activa.

Flujo de sucesos:
Flujo principal:

1. El alumno pulsa el ítem 'Abrir' en la barra de menú.
2. JUML genera un cuadro de dialogo para que el usuario indique el directorio donde se encuentra el diagrama.
3. El alumno selecciona el archivo donde se encuentra el diagrama.

4. JUML carga en el área de diseño el diagrama.

Post-Condición: un archivo con extensión juml es colocado en el área de diseño.

JUML Caso-Uso 12. Crear un archivo nuevo.
Objetivo: abrir desde un directorio un diagrama de clases.
Pre-Condición: se requiere de un área de diseño activa.

Flujo de sucesos:
Flujo principal:

1. El alumno pulsa el ítem 'Nuevo' en la barra de menú.
2. JUML genera un mensaje indicándole al alumno que limpiara el área de diseño.
3. El alumno selecciona la opción 'Si'.
4. JUML limpia el área de diseño.

Post-Condición: el área de diseño queda si elementos y lista para construcción de uno nuevo.

JUML Caso-Uso 13. Edición de componentes de clase.
Objetivo: editar un componente de clase desde el área de diseño.
Pre-Condición: se requiere de un componente de clase creado.

Flujo de sucesos:
Flujo principal:

1. El alumno pulsa sobre la sección de nombre de clase que desea editar.
2. JUML almacena el nombre de la clase digitada por el alumno.
3. El alumno pulsa clic derecho del Mouse sobre el área de propiedades para agregar o remover atributos.
4. JUML registra y muestra todas las propiedades creadas por el alumno.
5. El alumno pulsa clic derecho del Mouse sobre el área de métodos para agregar o remover funciones.

Post-Condición: un elemento de clase con su nombre, atributos y propiedades definidas.

JUML Caso-Uso 14. Generación de código fuente.
Objetivo: generar archivos con extensión java, que tomen como base el diagrama creado en el área de diseño para su modificación manual.
Pre-Condición: se requiere de un diagrama de clases creado.

Flujo de sucesos:
Flujo principal:

1. El alumno pulsa sobre el ítem 'Generar Código'
2. JUML almacena el nombre de la clase digitada por el alumno.
3. El alumno pulsa sobre el área de propiedades para agregar o remover atributos.
4. JUML registra y muestra todas las propiedades creadas por el alumno.
5. El alumno pulsa sobre el área de métodos para agregar o remover funciones.

Post-Condición: un elemento de clase con su nombre, atributos y propiedades definidas.

JUML Caso-Uso 15. Crear una imagen png conteniendo un diagrama de clases.
Objetivo: almacenar un diagrama de clase en una imagen formato png.
Pre-Condición: se requiere de un área de diseño activa y un diagrama creado.
Flujo principal:

1. El alumno pulsa el ítem 'Generar Imagen' en la barra de menú.
2. JUML genera un cuadro para que el usuario especifique el directorio donde desea guardar.
3. El alumno digita el nombre con que desea guardar el diagrama.
4. JUML genera una imagen en el directoria conteniendo un diagrama.

Post-Condición: un diagrama es almacenado como imagen.

Glosario.

A

Analista.

Es aquel individuo que ejerce las tareas de análisis de los sistemas informáticos, con el fin de automatizarlos.

Asistente.

Programa de ayuda encargado de guiar paso a paso al usuario para realizar una tarea específica dentro de una aplicación.

C

Código Fuente.

Un conjunto de líneas que conforman un bloque de texto, escrito según las reglas sintácticas de algún lenguaje de programación destinado a ser legible por humanos.

Código Abierto.

Del inglés open source, es el término con el que se conoce al software distribuido y desarrollado libremente.

E

Editor UML.

Programa de software que permite utilizar lenguaje UML para realizar su funcionalidad.

I

IDE.

Entorno de desarrollo integrado o en inglés Integrated Development Environment ('IDE') es un programa compuesto por un conjunto de herramientas para un programador.

M

Manual Programador.

Documento que contiene la información necesaria para entender la funcionalidad interna del software.

P

Programador.

Individuo que escribe programas utilizando un lenguaje de computación, también conocido por desarrolladores de software.

Paradigma.

Modelo o patrón en cualquier disciplina científica u otro contexto que involucre conocimiento científico.

Programación Estructurada.

Una forma de escribir programas de computadoras de forma clara, para ello utiliza únicamente tres estructuras: secuencial, selectiva e iterativa.

Programación Orientada a Objetos. (POO)

Paradigma de programación que define los programas en términos de "clases de objetos", objetos que son entidades que combinan estado, comportamiento e identidad.

Plantilla.

Es un medio o un instrumento que permite guiar, portar o construir un diseño o esquema predefinido.

R

RUP

Proceso de Ingeniería del Software que proporciona un enfoque disciplinado para asignar tareas y responsabilidades en las organizaciones de desarrollo de software.

S

Sistema.

Combinación de hardware y software necesarios para cumplir un objetivo.

Software.

Los programas de computadoras, procedimientos, además la documentación y asociados que forman parte de un sistema.

Software Comercial.

Es el software, libre o no, que es comercializado, es decir, que las compañías que lo producen, cobran dinero por el producto, su distribución o soporte.

U

UML.

UML (Lenguaje Unificado de Modelado) es un lenguaje gráfico para visualizar, especificar, construir y documentar los componentes de un sistema software. UML permite tanto la especificación conceptual de un sistema como la especificación de elementos concretos, como pueden ser las clases o un diseño de base de datos.

Fuentes de Informacion.

A. Bibliografía

HAMILTON, Kim. Learning UML 2.0. 2º Edicion. Editorial O'Reilly. Estados Unidos. 2006.

MYATT, Adam. Pro Netbeans IDE 6.0 Rich Client Application. Editorial Apress. Estados Unidos. 2008.

BOUDREAU, Tim. TULACH, Joroslab. WIELENGA, Geertjan. Rich Client Programming. Sun Microsystem. 2007.

HORTON, Ivor. Beginning Java 2 JDK 5. Editorial Wiley Publishing. Estados Unidos. 2005.

THE UNIFIED MODELING LANGUAGE REFERENCE MANUAL.

B. Sitios Web

- www.wikipedia.com Enciclopedia Libre.
- www.monografias.com Sitio Web con tutoriales y monografías.
- www.rational.com Sitio Web de racional
- www.java.com Sitio Web de JAVA
- www.inei.gob.pe Sitio Web del Instituto Nacional de Estadística el informática (Republica del Perú).
- http://ieee.udistrital.edu.co Historia de C++

Cronograma de Actividades

No.	Objetivo	Actividad	2007 Septiembre S1	S2	S3	S4	Octubre S1	S2	S3	S4	Noviembre S1	S2	S3	S4	Diciembre S1	S2	S3	S4	2008 Enero S1	S2	S3	S4	Febrero S1	S2	S3	S4	Marzo S1	S2	S3	S4	Abril S1	S2	S3	S4	Mayo S1	S2	S3	S4	Responsable
1	Analizar herramientas de codigo libre que trabajen con lenguaje UML y generen codigo fuente.	Investigacion Previa																																					Jorge Erazo
2		Recoleccion de Datos.																																					Adis Melendez
3		Realizacion Encuestas																																					Adis Melendez
4		Realizacion Entevistas																																					Adis Melendez
5		Analisis de Software																																					Jorge Erazo
6		Analisis de Resultados																																					Ambos
7	Desarrollar una que herramienta que contenga las caracteristicas necesarias para la creacion de diagramas de clases.	Construccion del Sistema																																					
8		Construccion de Casos de Uso																																					Jorge Erazo
9		Analisis de Requerimientos																																					Adis Melendez
10		Construccion de Arquitectura																																					Ambos
11		Diseño del Sistema																																					Ambos
12		Diseño de Arquitectura																																					Ambos
13		Identificacion y Diseño de Casos de Uso																																					Ambos
14		Implementacion del Sistema																																					
15		Construccion de Interface																																					Jorge Erazo
16		Implementacion de Arquitectura																																					Adis Melendez
17		Prueba del Sistema (Primera Iteracion)																																					
18		Depuracion de errores.																																					Ambos
19	Crear una estructura estatica de las aplicaciones usando diagramas de clases.	Prueba del Sistema (Segunda Iteracion)																																					
20		Depuracion de Investigacion.																																					Ambos
21		Primera Defensa																																					
22		Correcciones de Primera Defensa																																					Ambos
23	Generar una plantilla de codigo ejecutable en JAVA a partir de la estructura estatica de las aplicaciones.	Desarrollo de generador de codigo																																					Ambos
24		Prueba del Sistema (Tercera Iteracion)																																					
25	Documentar la investigación de la herramienta propuesta en el manual del programador a fin de permitir la factibilidad de incorporación de módulos que soporten el resto de diagramas	Depuraciones Finales																																					
26		Pruebas Finales																																					
27		Segunda Defensa																																					
28		Correciones de Segunda Defensa																																					Ambos
29		Entrega Final																																					
30		Asesorias tutor y asesor.																																					

Cronograma De Actividades

Anexos

CARACTERÍSTICAS DE JAVA

Las características principales que ofrece Java respecto a cualquier otro lenguaje de programación, son:

Simple

Java ofrece toda la funcionalidad de un lenguaje potente, pero sin las características menos usadas y más confusas de éstos. C++ es un lenguaje que adolece de falta de seguridad, pero C y C++ son lenguajes más difundidos, por ello Java se diseñó para ser parecido a C++ y así facilitar un rápido y fácil aprendizaje.

Java elimina muchas de las características de otros lenguajes como C++, para mantener reducidas las especificaciones del lenguaje y añadir características muy útiles como el garbage collector (reciclador de memoria dinámica). No es necesario preocuparse de liberar memoria, el reciclador se encarga de ello y como es un thread de baja prioridad, cuando entra en acción, permite liberar bloques de memoria muy grandes, lo que reduce la fragmentación de la memoria.

Java reduce en un 50% los errores más comunes de programación con lenguajes como C y C++ al eliminar muchas de las características de éstos, entre las que destacan:

- Aritmética de punteros
- No existen referencias
- Registros (struct)
- Definición de tipos (typedef)
- Macros (#define)
- Necesidad de liberar memoria (free)

Aunque, en realidad, lo que hace es eliminar las palabras reservadas (struct, typedef), ya que las clases son algo parecido.

Orientado a objetos

Java implementa la tecnología básica de C++ con algunas mejoras y elimina algunas cosas para mantener el objetivo de la simplicidad del lenguaje. Java trabaja con sus datos como objetos y con interfaces a esos objetos. Soporta las tres características propias del paradigma de la orientación a objetos: encapsulación, herencia y polimorfismo. Las plantillas de objetos son llamadas, como en C++, clases y sus copias, instancias. Estas instancias, como en C++, necesitan ser construidas y destruidas en espacios de memoria.

Java incorpora funcionalidades inexistentes en C++ como por ejemplo, la resolución dinámica de métodos. Esta característica deriva del lenguaje Objective C, propietario del sistema operativo Next. En C++ se suele trabajar con librerías dinámicas (DLLs) que obligan a recompilar la aplicación cuando se retocan las funciones que se encuentran en su interior. Este inconveniente es resuelto por Java mediante una interfaz específica llamada RTTI (RunTime Type Identification) que define la interacción entre objetos excluyendo variables de instancias o implementación de métodos. Las clases en Java tienen una representación en el runtime que permite a los programadores interrogar por el tipo de clase y enlazar dinámicamente la clase con el resultado de la búsqueda.

Distribuido

Java se ha construido con extensas capacidades de interconexión TCP/IP. Existen librerías de rutinas para acceder e interactuar con protocolos como http y ftp. Esto permite a los programadores acceder a la información a través de la red con tanta facilidad como a los ficheros locales.

La verdad es que Java en sí no es distribuido, sino que proporciona las librerías y herramientas para que los programas puedan ser distribuidos, es decir, que se corran en varias máquinas, interactuando.

Robusto

Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución. La comprobación de tipos en Java ayuda a detectar errores, lo antes posible, en el ciclo de desarrollo. Java obliga a la declaración explícita de métodos, reduciendo así las posibilidades de error. Maneja la memoria para eliminar las preocupaciones por parte del programador de la liberación o corrupción de memoria.

También implementa los arrays auténticos, en vez de listas enlazadas de punteros, con comprobación de límites, para evitar la posibilidad de sobrescribir o corromper memoria resultado de punteros que señalan a zonas equivocadas. Estas características reducen drásticamente el tiempo de desarrollo de aplicaciones en Java.

Además, para asegurar el funcionamiento de la aplicación, realiza una verificación de los byte-codes, que son el resultado de la compilación de un programa Java. Es un código de máquina virtual que es interpretado por el intérprete Java. No es el código máquina directamente entendible por el hardware, pero ya ha pasado todas las fases del compilador: análisis de instrucciones, orden de operadores, etc., y ya tiene generada la pila de ejecución de órdenes.

Java proporciona:

- Comprobación de punteros
- Comprobación de límites de arrays
- Excepciones
- Verificación de byte-codes

Arquitectura neutral

Para establecer Java como parte integral de la red, el compilador Java compila su código a un fichero objeto de formato independiente de la arquitectura de la máquina en que se ejecutará. Cualquier máquina que tenga el sistema de ejecución (run-time) puede ejecutar ese código objeto, sin importar en modo alguno la máquina en que ha sido generado.

El código fuente Java se "compila" a un código de bytes de alto nivel independiente de la máquina. Este código (byte-codes) está diseñado para ejecutarse en una máquina hipotética que es implementada por un sistema run-time, que sí es dependiente de la máquina.

En una representación en que tuviésemos que indicar todos los elementos que forman parte de la arquitectura de Java sobre una plataforma genérica, obtendríamos una figura como la siguiente:

Seguro

La seguridad en Java tiene dos facetas. En el lenguaje, características como los punteros o el casting implícito que hacen los compiladores de C y C++ se eliminan para prevenir el acceso ilegal a la memoria. Cuando se usa Java para crear un navegador, se combinan las características del lenguaje con protecciones de sentido común aplicadas al propio navegador.

El lenguaje C, por ejemplo, tiene lagunas de seguridad importantes, como son los errores de alineación. Los programadores de C utilizan punteros en conjunción con operaciones aritméticas. Esto le permite al programador que un puntero referencie a un lugar conocido de la memoria y pueda sumar (o restar) algún valor, para referirse a otro lugar de la memoria. Si otros programadores conocen nuestras estructuras de datos pueden extraer información confidencial de nuestro sistema. Con un lenguaje como C, se pueden tomar números enteros aleatorios y convertirlos en punteros para luego acceder a la memoria.

Portable

Más allá de la portabilidad básica por ser de arquitectura independiente, Java implementa otros estándares de portabilidad para facilitar el desarrollo. Los enteros son siempre enteros y además, enteros de 32 bits en complemento a 2. Además, Java construye sus interfaces de usuario a través de un sistema abstracto de ventanas de forma que las ventanas puedan ser implantadas en entornos Unix, Pc o Mac.

Interpretado

El intérprete Java (sistema run-time) puede ejecutar directamente el código objeto. Enlazar (linkar) un programa, normalmente, consume menos recursos que compilarlo, por lo que los desarrolladores con Java pasarán más tiempo desarrollando y menos esperando por el ordenador. No obstante, el compilador actual del JDK es bastante lento. Por ahora, que todavía no hay compiladores específicos de Java para las diversas plataformas, Java es más lento que otros lenguajes de programación, como C++, ya que debe ser interpretado y no ejecutado como sucede en cualquier programa tradicional.

Aprendizaje

Algo que no es cierto es que es necesario aprender C++ antes de aprender Java.