Universidad de las Ciencias Informáticas

FACULTAD 6

Título: Implementación de un prototipo funcional para automatizar el proceso de pruebas de Caja Blanca del departamento Señales Digitales del Centro GEySED.

Trabajo de Diploma para optar por el título de

Ingeniero en Ciencias Informáticas

Autor: Javier Lambert Claramunt.

Tutora: Ing. Anay Iyenis Chapman Hernández.

La Habana, junio de 2012

DECLARACIÓN DE AUTORÍA

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

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

Javier Lambert Claramunt

_____

Firma del Autor

Ing. Anay Chapman Hernández

_____

Firma del Tutor

DATOS DE CONTACTO

Tutora: Anay Iyenis Chapman Hernández.

Institución: Universidad de las Ciencias Informáticas.

Dirección de la institución: Carretera a San Antonio de los Baños, Km. 2 ½, Reparto: Torrens, Municipio: Boyeros, Provincia: La Habana.

Correo electrónico: achapman@uci.cu.

Teléfono del trabajo: 837 2583

Cargo del trabajador: Profesor.

Título de la especialidad de graduado: Ingeniero en Ciencias Informáticas.

Año de graduación: 2007.

Institución donde se graduó: Universidad de las Ciencias Informáticas.

Dedicado…

A mi madre Ibis, por todo su amor brindado, por ser una mujer espectacular, por inculcarme ser mejor cada día y superarme siempre, por todo su esmero, cariño y dedicación le dedico esta tesis.

A mi padre Rodolfo, por todo su amor brindado, por ser mi ejemplo y mi guía, por todo su esfuerzo realizado, por hacer de mí un hombre de bien, por estar siempre presente le dedico esta tesis.

A mi familia, en especial a mis abuelas Nidia y Lucía, a mis tías María Elena, Anita, mis primos Ivet, Annie, Oscarín y Osmay, por su apoyo y cariño.

A los amigos, y a todas aquellas personas que han estado presentes y que son parte de mi vida, les dedico esta tesis.

AGRADECIMIENTOS

Agradezco…

A mis padres, mis grandes amigos. A ellos por su entrega total, por su amor y cariño, sus buenos consejos, por su guía y hacer de mí un hombre de bien, mi agradecimiento eterno por estar siempre en cada momento.

A mi familia, por siempre contar con su apoyo incondicional y demostrar que siempre están presentes en los momentos buenos y malos. Para mis abuelas, mis tías y tíos, mis primos, mi agradecimiento por todas sus demostraciones de afecto.

A la tutora Anay por su dedicación y ayuda.

Al tribunal por su ayuda, paciencia y esmero.

A los amigos que siempre han brindado su ayuda, Yasmina, Toto, Belkis, Antonio, Chachi, Aldo, Fela, Miriam, Aquino, Enrique, Leandro, Pedro, Douglas, Magalys, Gustavo, Tavito.

A los compañeros de la carrera, en especial a los que han compartido conmigo estos cinco años.

Para todos ellos mi agradecimiento.

RESUMEN

El objetivo fundamental del grupo de Calidad del Centro GEySED es lograr que los productos desarrollados en el mismo cuenten con una alta calidad. Para ello se realiza un proceso de revisiones y pruebas a los mismos con el objetivo de detectar deficiencias y posibles fallos que puedan tener. Entre los diferentes tipos de pruebas se encuentran las pruebas de caja blanca, que se identifican por incidir directamente en el código fuente de las aplicaciones, analizando su complejidad, eficiencia y calidad. Este tipo de pruebas tienden a ser complejas y trabajosas debido a las grandes cantidades de líneas de código que deben analizarse, por lo que resulta inconveniente realizar este proceso de forma manual, ya que trae consigo pérdida de tiempo productivo, a lo que se puede adicionar la posibilidad de errores humanos que pueden dañar el proceso de pruebas al software. Es por ello que surgió la necesidad en el Centro GEySED de automatizar este proceso con el fin de poder incorporarlas al proceso de pruebas que existe en el grupo de Calidad.

Durante la investigación realizada en este trabajo se logró desarrollar una herramienta capaz de realizar pruebas de caja blanca, específicamente la técnica del camino básico. Esta herramienta permitió que se iniciara el proceso de pruebas de caja blanca de manera automática, lo que trajo consigo analizar grandes cantidades de líneas de código en pocos segundos, ganando en tiempo y esfuerzo por parte de quien la utilice.

PALABRAS CLAVE

Calidad, Caja blanca, Camino básico, Pruebas de software.

ÍNDICE

INDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

Las Tecnologías de la Información y las Comunicaciones (TICs) han sido conceptualizadas como la integración y convergencia de la computación, las telecomunicaciones y las técnicas para el procesamiento de datos. Con el uso adecuado de la informática los países han visto la posibilidad de mejorar notablemente los mecanismos tradicionales de gestión y servicios, que se traduce en beneficios tangibles para sus habitantes. Principalmente la mayor aplicación de las computadoras es simplificar el trabajo de almacenamiento, contabilidad y análisis de datos. Aunque gracias al avance de la informática no se ha limitado solo a esto. En nuestros días estos artefactos han revolucionado la vida de las personas, ya que la mayoría utilizan la computadora como un bien indispensable en el desarrollo de sus tareas. Debido a esto bien se dice que ha surgido un nueva era, la de la informática y las comunicaciones (1).

Cuba no está ajena a este progreso que han tenido las TICs. Por tal motivo la máxima dirección del gobierno cubano, consciente de las ventajas y facilidades que brindan estas tecnologías para elevar el avance cultural de la sociedad en general, no solo prioriza y promueve su informatización, para lo cual dedica innumerables esfuerzos; sino que aboga por el desarrollo de soluciones informáticas encaminadas a solucionar diversos problemas.

La tecnología genera desarrollo y por ello surge por idea del comandante Fidel Castro Ruz la Universidad de las Ciencias Informáticas (UCI) como punta de lanza del proceso de informatización que se desarrolla en Cuba. La UCI, promueve mediante el modelo de formación estudio-trabajo-investigación, la capacitación y preparación de ingenieros altamente calificados e integrales. Para ello surgen en cada una de sus facultades varios centros especializados en diferentes esferas de la informática, con el objetivo de dar solución a los diversos problemas de la sociedad cubana. La facultad 6 cuenta con el centro de Geoinformática y Señales Digitales (GEySED). En el mismo existen dos departamentos: Geoinformática que se especializa en el desarrollo de aplicaciones vinculadas al campo de la geología, y Señales Digitales que trabaja en la captura y procesamiento de señales digitales de radio y televisión.

Desde la creación de la UCI hasta la actualidad se han obtenido notables resultados en la producción de software hacia diferentes esferas de la sociedad cubana y varios países del mundo. Además se ha podido constatar que se brindan productos de gran calidad a las empresas que han solicitado un servicio a la universidad, pues estas han mostrado su satisfacción por los mismos. Por la necesidad de establecer y controlar la calidad en estos productos, la universidad crea el Centro de Calidad para

Soluciones Informáticas (CALISOFT), que cuenta con varios grupos para el asesoramiento en cada una de las facultades. El grupo de Calidad del centro GEySED de la Facultad 6 se encarga de verificar y validar que los productos desarrollados en el centro presenten una óptima calidad.

Entre varios aspectos que dan validez a un software, se encuentra el estado óptimo del código fuente. Para ello es necesario que todos los desarrolladores del centro realicen su programación respetando los estándares de codificación establecidos, para su posterior documentación y comprobación. Para evaluar la eficiencia del software mientras se desarrolla, se aplican pruebas de caja blanca que inciden directamente sobre el código. Estas pruebas verifican el funcionamiento interno del software, así como detectan los posibles errores en los ciclos, decisiones, condiciones y todo lo relacionado a estructuras y variables.

En el grupo de Calidad del departamento Señales Digitales no se utilizan las pruebas de caja blanca. De realizarse estas pruebas se harían de forma manual, lo que implicaría gran cantidad de tiempo y personal calificado para ejecutarlas, incluyendo la posibilidad de que se cometan errores que puedan dañar el proceso de pruebas al software. Se considera que la no aplicación de estas pruebas en el departamento, conlleva a que no se controlen parámetros de calidad en el código fuente de las aplicaciones. Lo que traería consigo que no se puedan detectar posibles errores durante su implementación, ya que no se utilizan estas pruebas que son muy importantes durante todo el proceso de desarrollo del software.

Teniendo en cuenta la situación problemática planteada anteriormente se tomó como **problema a resolver:** ¿Cómo contribuir a agilizar el proceso de pruebas de caja blanca a los productos que se desarrollan en el departamento Señales Digitales del centro GEySED de la UCI?

Por tal motivo se toma como **objeto de estudio** de la presente investigación: el proceso de prueba de caja blanca. Enmarcado en el **campo de acción:** automatización del proceso de prueba de caja blanca.

Se define como **objetivo general:** Implementar un prototipo funcional que permita la automatización del proceso de pruebas de caja blanca que se aplican a los productos del departamento Señales Digitales del centro GEySED de la UCI.

Para lo cual se plantea la siguiente **idea a defender:** Si se implementa un prototipo funcional para la automatización de las pruebas de caja blanca en el departamento Señales Digitales, se podrá agilizar el proceso de pruebas al código fuente de los productos que se desarrollan en el departamento Señales Digitales del centro GEySED de la UCI.

Para dar cumplimiento al objetivo planteado se realizan las siguientes **tareas de investigación**:

1. Caracterización de las técnicas de caja blanca que serán implementadas.
2. Selección de las herramientas y tecnologías a utilizar para la implementación del sistema.
3. Realización de la documentación técnica correspondiente al Modelo de diseño de la herramienta.
4. Descripción de los principales algoritmos y clases a implementar.
5. Realización de la documentación técnica correspondiente al Modelo de Implementación de la herramienta.
6. Implementación del prototipo funcional.
7. Validación de la solución propuesta.

Una vez que se concluya el presente trabajo investigativo se pretenden alcanzar como **posibles resultados** los que se enuncian a continuación:

- Herramienta para la automatización del proceso de pruebas de Caja Blanca a los productos del departamento Señales Digitales.
- Documentación técnica del proceso ingenieril de la herramienta para la automatización del proceso de pruebas de Caja Blanca.

En el transcurso de la investigación se utilizaron diferentes **métodos de investigación científica,** los seleccionados fueron los siguientes:

Métodos Teóricos

- **Histórico - lógico:** se utilizó en la investigación enfocada al comportamiento histórico y actual de los procesos de pruebas de caja blanca, así como de las herramientas y tecnologías empleadas para el desarrollo de la aplicación.
- **Modelación:** permitió realizar la modelación de las pruebas de caja blanca que fueron aplicadas a los productos que se desarrollan en el departamento Señales Digitales.
- **Analítico sintético:** se empleó a partir del procesado de toda la información referente al proceso de pruebas de caja blanca, las cuales van a ser sintetizadas partiendo de las que son necesarias para el desarrollo de la aplicación orientado a las necesidades planteadas.

Métodos Empíricos

- **Entrevista:** permitió tener una comunicación con otras personas que sirvieron para apoyar y dar ideas en cuanto al sistema que se quiere construir, las cuales se pueden ver en el Anexo 1.

El contenido de este trabajo se distribuyó de la siguiente manera:

Capítulo 1. Fundamentación Teórica: este capítulo se compone de conceptos, definiciones, además de una caracterización de la técnica de caja blanca que fue implementada. Se realizó un estudio de las soluciones existentes para demostrar la necesidad de implementar una aplicación que automatice el proceso de pruebas de caja blanca.

Capítulo 2 Tecnologías y tendencias actuales: en este capítulo se realizó un estudio de las principales tendencias y tecnologías actuales que se utilizan en el mundo de la informática. Se ofrecen características de las tecnologías seleccionadas para el desarrollo de este trabajo, que están acorde con las necesidades y objetivos que se persiguen.

Capítulo 3 Análisis y Diseño del Sistema: en este capítulo se definieron los conceptos asociados al modelo de dominio del problema. También se especifican los requisitos funcionales y no funcionales del prototipo funcional. Se describen los patrones y estilos arquitectónicos utilizados, así como la descripción de los casos de usos de la aplicación.

Capítulo 4 Implementación y Pruebas: este capítulo está orientado a la implementación del prototipo funcional, con el objetivo de darle solución a los requisitos funcionales establecidos. En él se describe el modelo de implementación, que incluye los diagramas de despliegue y componente de la herramienta. Se realizó una descripción de los principales algoritmos utilizados en la implementación de la misma. También se probaron todas las funcionalidades del software para verificar su correcto funcionamiento.

CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA

Introducción

En este capítulo se analizaron diferentes conceptos de calidad que resultan importantes para la posterior comprensión de los temas abordados en el desarrollo de la tesis. Se realizó una caracterización de las técnicas de pruebas de caja blanca que serán implementadas, así como un análisis de las herramientas de pruebas de caja blanca existentes en el mundo.

1.1 Calidad de Software

Debido a la imposibilidad humana de trabajar y comunicarse de forma perfecta, el desarrollo de software debe ir acompañado de una actividad que garantice la calidad del software. Es por ello que a continuación se brindan conceptos asociados a la calidad de software, lo que permitió formalizar la definición que se tomará en lo que sigue.

1.1.1 Calidad de software

Según Roger Pressman[1] la calidad de software es la concordancia entre los requisitos funcionales y de rendimiento explícitamente establecidos con los estándares de desarrollo bien documentados y con las características implícitas que se espera de todo software desarrollado profesionalmente (2).
Genichi Taguchi[2] dice que la calidad de software son: "Las pérdidas que un producto o servicio infringe a la sociedad desde su producción hasta su consumo o uso. A menores pérdidas sociales, mayor calidad del producto o servicio" (3).
William Edwards Deming[3] define: "la calidad de software no es otra cosa más que una serie de cuestionamiento hacia una mejora continua".
Para la *IEEE* la calidad es el grado con el cual el software cumplirá con las expectativas del cliente. Se basa en combinación de atributos o características, expectativas de clientes y percepciones del usuario.

[1] Ingeniero de software americano, autor, consultor y presidente de R.S. Pressman & Associates.

[2] Ingeniero y estadístico japonés quien ha hecho grandes aportes sobre calidad.

[3] Estadístico norteamericano considerado como el padre de la calidad.

Tomando como referencia los conceptos mencionados anteriormente se puede afirmar que la calidad de software se basa en un conjunto de características presentes en un producto. Con el objetivo de satisfacer todos los requisitos planteados por el cliente, que está dada principalmente por la calidad de los procesos que sean capaces de cubrir esas características mediante tres principios fundamentales: planificación, control y mejora continua.

Una vez analizado el concepto de calidad de software se procede a definir aspectos importantes en la investigación. Entre los que se destacan: proceso de aseguramiento de la calidad, por sus siglas en inglés (PPQA) y proceso de prueba de software.

1.1.2 Proceso de aseguramiento de la calidad

El aseguramiento de la calidad es un conjunto de actividades planificadas y bien pensadas de soporte o protección que se desarrolla durante todo el proceso de software. Deben ser aplicadas de manera sistemática para aportar la confianza de que el producto de software satisfará los requisitos dados de calidad (4).

El aseguramiento de la calidad también se define como el esfuerzo total para plantear, organizar, dirigir y controlar la calidad de un sistema de producción con el objetivo de brindar al cliente mejores productos. Es un sistema, y como tal un conjunto organizado de procedimientos bien definidos y entrelazados armónicamente (4). La garantía de la calidad como también referencian algunos autores consiste en un conjunto de funciones de auditorías e información que evalúan la efectividad y complejidad de las actividades de control de la calidad. Su propósito es brindar al gestor los datos necesarios para que esté informado acerca de la calidad del producto, para su confianza y comprensión de que la calidad del producto esté cumpliendo sus expectativas (2).

Teniendo en cuenta los conceptos enunciados anteriormente se considera que el proceso de aseguramiento de la calidad es una actividad de protección que se concibe para cada aplicación mediante un plan de aseguramiento, el cual comprende métodos, herramientas, pruebas y control de la documentación así como los cambios realizados. Como se pudo constatar en este apartado, el proceso de aseguramiento de la calidad implica una serie de actividades con el objetivo de asegurar un software de calidad. Entre estas actividades se encuentra el proceso de pruebas de software, el cual se describe a continuación.

1.1.3 Proceso de pruebas de software

La mayoría de las personas involucradas en el proceso de desarrollo de un software tienden a enfocar el proceso únicamente desde el punto de vista constructivo, y se basan en generar más artefactos en

menos tiempo. Una vez terminado realizan una serie de pruebas y si no encuentran fallos pues caen en el error de pensar que todo lo que han hecho está bien. Las pruebas de software lejos de buscar que los artefactos están libres de fallos, resulta todo lo contrario, es el proceso de ejecutar un programa con la intención de encontrar fallos. Los errores pueden estar presentes desde el inicio del proceso de desarrollo por lo que resulta necesario ir comprobando que no se han cometido fallos durante la construcción del software. Las revisiones técnicas formales como un filtro previo a la prueba llegan a ser tan efectivas como las mismas pruebas para descubrir errores, lo que se traducen en una reducción de la cantidad de esfuerzo de prueba que se requiere para producir un software de calidad. Además de ahorrar tiempo y mejorar la calidad del producto, ya que posibilitan descubrir inconsistencias, omisiones y errores evidentes en el enfoque de la prueba que se desea realizar (2).

Las pruebas de software se aplican con diferentes objetivos en dependencia de los niveles de trabajo en que se encuentre el proceso de desarrollo, a continuación se muestra los niveles de pruebas existentes.

Niveles de prueba

- **Prueba de desarrollador:** esta prueba es diseñada e implementada por el propio equipo de desarrollo, comúnmente estas pruebas se han tenido en cuenta solo para las pruebas de unidad, pero se ha demostrado que en algunos casos se pueden ejecutar pruebas de integración.
- **Prueba independiente:** esta prueba es diseñada e implementada por alguna persona que esté ajena al equipo de desarrolladores. Se basa fundamentalmente en el aspecto diferente que le pueden dar estas personas mirando desde un entorno distinto al de los desarrolladores. Una vista de esta prueba es la propia prueba independiente de los *stakeholder*[4], ya que son basadas en las necesidades y preocupaciones de ellos mismos.
- **Prueba de unidad:** esta prueba se centra en el esfuerzo de verificación de la unidad más pequeña del diseño del software, dígase componente o módulo del software. Es diseñada e implementada por el propio equipo de desarrollo. En el ella se prueban importantes caminos de control para descubrir errores dentro de los límites del módulo. Esta prueba siempre está orientada a caja blanca (2).

[4] Son considerados todas aquellas personas que están involucrados en la realización de un software determinado, que pueden afectar o ser afectados tanto de forma negativa o positiva durante el proceso de desarrollo del software.

- **Prueba de integración:** es una técnica sistemática para construir la arquitectura del software mientras, al mismo tiempo se aplican las pruebas para descubrir errores asociados a la interfaz (2). Comprueban que los componentes se unan de manera correcta a través de sus interfaces, además de verificar si cumplen con la funcionalidad establecida. Descubren errores en las especificaciones de las interfaces de los paquetes del modelo de implementación, es responsabilidad aplicarla por parte del equipo de desarrolladores y personal independiente.
- **Prueba de regresión:** comprueban que los cambios sobre un componente de un sistema (cada vez que se añade un módulo, el software cambia), no introducen un comportamiento no deseado o errores adicionales en otros componentes no modificados.
- **Prueba basada en hilos:** integra el conjunto de clases necesario para responder a una entrada o evento del sistema. Cada hilo se integra y prueba individualmente. Se aplica la prueba de regresión para asegurar que no ocurren efectos colaterales.
- **Prueba basada en uso**: comienza la construcción del sistema probando aquellas clases (llamadas independientes) que usan muy pocas de las clases servidor. Luego se comprueban la próxima capa de clases, llamadas clases dependientes, que usan las clases independientes. Esta secuencia de capas de clases dependientes continúa hasta construir el sistema por completo.
- **Prueba de sistema:** prueban a fondo el sistema, comprobando su funcionalidad e integridad globalmente, en un entorno lo más semejante posible al entorno final de producción.
- **Prueba de aceptación:** esta prueba se aplica antes del despliegue del sistema, verifican que el sistema cumple con todos los requisitos establecidos, además de permitir la aceptación o no de los usuarios del sistema.

Luego de analizar los distintos niveles de pruebas por los que puede ser sometido un software durante todo su ciclo de vida, resulta necesario conocer los distintos tipos de pruebas que se aplican para verificar y validar la máxima calidad del mismo.

1.2 Tipos de Pruebas

La prueba de software es un proceso que puede planearse y especificarse periódicamente. Se diseña el caso de prueba, se define una estrategia y se evalúan los resultados frente a las expectativas creadas (2).

Para probar la calidad de un producto determinado, se encuentran dos métodos fundamentales, el método de caja negra y el de caja blanca. Las pruebas de caja negra reciben su nombre debido a los

elementos que se prueban y las condiciones bajo las cuales se hacen las pruebas, se basan en los requerimientos funcionales del sistema, probando la entrada y salida de los datos así como la integridad de la información externa. A continuación se muestra una imagen para representar las pruebas de caja negra y caja blanca (2).

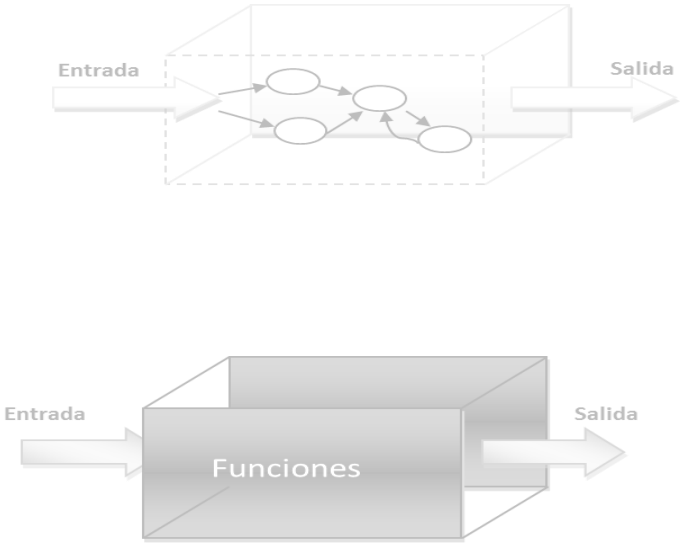


Figura 1. **Representación de las pruebas de Caja Blanca y Caja Negra.**

Las pruebas de Caja Blanca comprueban los caminos lógicos del software. Propone casos de prueba para recorrer conjuntos específicos de ciclos y condiciones presentes en el código fuente del software. Permite además examinar el estado del programa en varios puntos para determinar si el estado real coincide con lo esperado.

¿Por qué realizar pruebas de caja blanca?

- Los errores lógicos y las suposiciones incorrectas son inversamente proporcional a la probabilidad de que se ejecute un camino del programa.
- A veces se piensa que un camino lógico tiene pocas posibilidades de ejecutarse cuando puede hacerlo de forma normal (5).

Este método se abordará y profundizará en el desarrollo de este trabajo, pues constituye el objeto de estudio de la investigación.

1.2.1 Pruebas de Caja Blanca

Las pruebas de Caja Blanca o de Cristal como también se le conoce deben su nombre a que estas revisan la parte interna del software, específicamente el código fuente generado a diferencia de las pruebas de Caja Negra que se centran en la interfaz del programa. Estas pruebas basan su función en un examen minucioso de los detalles procedimentales, además de comprobar los caminos lógicos o

independientes del programa para diseñar los casos de pruebas que verifiquen todas las condiciones y/o bucles, con el objetivo de determinar si el estado real coincide con lo que se espera (6).
Mediante las pruebas de caja blanca, el ingeniero o el probador de ejecute la prueba puede generar casos de pruebas que garanticen que se:

- Ejerciten todas las decisiones lógicas en las vertientes verdadera y falsa.
- Ejecuten todos los bucles en sus límites operacionales.
- Ejerciten las estructuras internas de datos para asegurar su validez.
- Ejerciten por lo menos una vez todos los caminos independientes de cada módulo, programa o método.

Es por ello y a diferencia de los que muchos piensas, que se considera a las pruebas de Caja Blanca como una de las pruebas más importantes que se le pueden aplicar a un software. Logrando como resultado disminuir en gran porciento el número de errores existentes en los sistemas y por ende una mayor calidad (2).

1.2.2 Métodos de pruebas de Caja Blanca

Existen diferentes métodos que analizan diferentes partes del código y se complementan entre ellos para garantizar la máxima calidad posible. A continuación se definen varios de ellos, haciendo énfasis en la prueba del camino básico ya que es el método que se desea implementar.

- **Prueba de condición:** se encamina hacia la ejercitación de las condiciones, basándose en el principio de que si un conjunto de casos de pruebas es capaz de ejercitar todas las condiciones contenidas en un bloque de código. Este mismo conjunto servirá para encontrar más errores en el programa que no tengan que ver directamente con las condiciones.
- **Prueba de flujo de datos:** verifica la validez en el uso de las variables para manipular los datos de la aplicación, selecciona los casos de prueba atendiendo a las definiciones y los usos de las variables. Este procedimiento indica que se debe encontrar las sentencias donde se define cada variable y las sentencias donde se hace uso de la misma. Luego se encuentran las "cadenas de definición – uso" que representan el ciclo de vida de las variables y se diseñan casos de prueba que las ejecuten en su totalidad.
- **Prueba de bucles:** se centra en la validez de las estructuras cíclicas o bucles, dados que estos son la piedra angular de la mayoría de las aplicaciones de software. Su objetivo es probar el comportamiento de estas estructuras en sus valores límites de iteración, los bucles se clasifican en cuatro tipos: simples, anidados, concatenados y no estructurados; aunque la esencia es la misma, cada tipo se prueba de forma diferente.

- **Prueba del camino básico:** se basa en obtener una medida de la complejidad del diseño procedimental de un programa, medida dada por la complejidad ciclomática de Tom McCabe[5].

Al analizar los métodos de pruebas, resulta evidente la importancia de cada uno de ellos en la búsqueda de posibles fallos en el código fuente del software. Se enmarcan en probar la eficiencia del código analizado, dando como resultado aplicaciones con mejor calidad en su código fuente. Específicamente la prueba del camino básico se basa en el diagrama de flujo determinado por las estructuras de control de un determinado código. De dicho análisis se puede obtener una medida cuantitativa de la complejidad del código. Es por ello que a continuación se profundiza en este método evidenciando sus potencialidades como métrica de software, y su utilidad dentro de un entorno de prueba.

Técnica del camino básico

Para aplicar esta técnica es necesario seguir una serie de pasos que se describen a continuación (2):

- A partir del código fuente, se dibuja el grafo de flujo asociado.
- Se calcula la complejidad ciclomática del grafo.
- Se determina el conjunto de caminos independientes.
- Se diseñan los casos de prueba que garanticen la ejecución de cada camino.

El grafo de flujo se utiliza para hacer una representación lógica del flujo de control de un programa. Está formado por tres componentes esenciales que ayudan a su construcción y comprensión, estos componentes son: los nodos, que representan una o varias sentencias en secuencia. Las aristas, que son las líneas que unen a los nodos y representan el flujo de control, cada arista debe terminar en un nodo aunque este no represente ninguna sentencia procedimental. Por último las regiones que son las áreas delimitadas por las aristas y nodos, es importante destacar que la cantidad de regiones del grafo coincide con la complejidad ciclomática del mismo, así como la cantidad de caminos independientes del conjunto básico de un programa. Un camino independiente es cualquier camino del programa que introduce como mínimo un nuevo conjunto de sentencias o una nueva condición, este camino debe recorrer alguna arista que no haya sido visitada anteriormente. A Continuación se muestran las notaciones de los grafos para cada una de las instrucciones que se verificarán.

[5] Creador de la métrica de software, complejidad ciclomática.

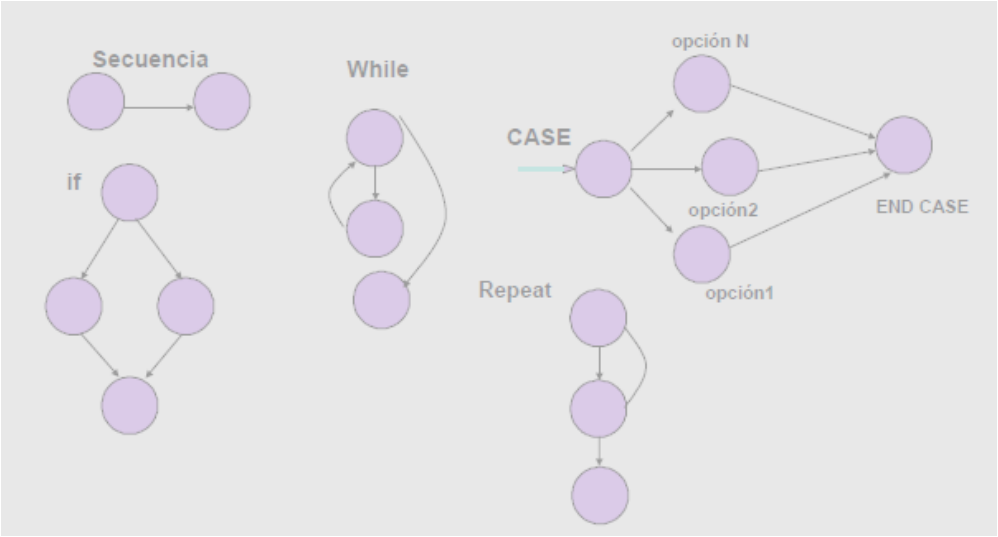


Figura 2. Representación de las instrucciones secuencia, if, while/for, do while y case.

La complejidad ciclomática es una métrica de software que brinda medir cuantitativamente la complejidad lógica de un programa. Este valor coincide con la cantidad máxima de casos de pruebas que se deben diseñar para asegurar que se ejecute cada camino del programa. Existen varias formas para calcular la complejidad ciclomática de un software:

- El número de regiones del grafo de flujo coincide con la complejidad ciclomática, V (G).
- V (G) = A - N + 2 donde A es la cantidad de aristas del grafo y N la cantidad de nodos.
- V (G) = P + 1 donde P es la cantidad de nodos predicados contenidos en el grafo.

Un nodo predicado es cuando en una condición aparecen uno o más operadores lógicos (AND, OR, XOR…), para representarlo se crea un nodo distinto por cada una de la condiciones simples.

Por último para diseñar los casos de prueba que garanticen la ejecución de cada camino, se preparan los datos de forma que las condiciones de los nodos estén bien definidas para verificar cada camino. Luego de confeccionar los casos de prueba se ejecutan cada uno de estos y se comparan los resultados con los esperados. Una vez terminados todos los casos de prueba se podrá asegurar que todas las sentencias del programa se han ejecutado por lo menos una vez.

Una vez definidos los aspectos fundamentales del objeto de estudio en cuestión se procede a hacer un análisis de las principales soluciones existentes a nivel mundial. Se pretende hacer una descripción detallada de los programas que se encuentren, llegando a conclusiones importantes para la continuidad del presente trabajo.

1.3 Soluciones existentes.

El uso de las tecnologías en el mundo cada día va en ascenso debido a las grandes ventajas que ofrecen estos medios en el desarrollo de aplicaciones que permiten automatizar los procesos que se originan en la vida cotidiana. Las pruebas de caja blanca como parte del proceso que se realiza para comprobar la calidad de un producto, no han escapado a esta tendencia global de automatización de los procesos. Sin dejar de mencionar que no por esto, estas aplicaciones han dejado de tener un alto grado de complejidad en su desarrollo, así como en su aplicación práctica, debido fundamentalmente a grandes cantidades de líneas de código que se deben analizar. Es por ello que este tipo de herramientas apenas se utiliza en ambientes industriales, sin embargo en el mundo han surgido diferentes aplicaciones de este tipo, con el objetivo de viabilizar y facilitar el proceso de pruebas de software.

En Cuba no existe una herramienta que sea capaz de automatizar los diferentes procesos de pruebas de caja blanca que existen. En entrevistas realizadas a directivos y trabajadores del Centro de Calidad para Soluciones Informáticas (CALISOFT), centro de referencia de calidad y órgano de certificación de software conocido y acreditado a nivel nacional, se evidencia que este tipo de pruebas apenas se utiliza en los diferentes productos que allí son analizados. Esto se debe en gran medida a la carencia de un software que pueda viabilizar este proceso, por lo cual cuando se solicita este tipo de prueba deben desarrollarlo manualmente invirtiendo gran cantidad de tiempo en su ejecución. En el departamento de Señales Digitales del Centro GEySED, este tipo de pruebas no se aplica, debido fundamentalmente a las razones que se expresan a continuación. Los desarrolladores de los productos en su mayoría no conoce la importancia de este tipo de pruebas mientras se desarrolla una aplicación, y el departamento de calidad no cuenta con una aplicación que viabilice este proceso. De hacer este tipo de pruebas de forma manual conllevaría gran cantidad de tiempo y personal calificado para ejecutarlas, producto a las grandes cantidades de líneas de código a analizar.

En la investigación realizada en búsqueda de soluciones informáticas fuera de Cuba que diera una posible solución al problema planteado, se encontraron algunas herramientas que realizan pruebas basadas en algunas métricas de calidad de software. Sin embargo estas herramientas no se enfocan en la técnica que se desea implementar como solución al problema planteado. Dentro de las pruebas de software, específicamente la técnica del camino básico se analizó la herramienta *Program Exploration* la cual presenta una serie de características que se describen a continuación.

Program Exploration

Se trata de una herramienta desarrollada por un equipo de *Microsoft*[6] *Research*, cuya última actualización data de febrero del 2010. Tiene la capacidad de explorar el código fuente de las aplicaciones, encontrar los grafos de caminos, seleccionar el subconjunto mínimo suficiente de caminos para probar todas las sentencias de código y, finalmente, genera las entradas representativas necesarias al programa para recorrer todos estos caminos (7).

Esta herramienta posee una licencia libre y no tiene precio, sin embargo se pudo comprobar que no es posible realizar una descarga efectiva de la aplicación, por lo que no se encuentra al alcance de la universidad y no es posible su utilización.

También existen dentro de los Entornos de Desarrollo Integrado (IDE)[7], algunas pruebas unitarias integradas con algún componente que tenga el IDE. Sin embargo el número de métricas que ofrecen estos componentes son muy reducidas, por lo que no se pueden realizar las pruebas del camino básico, técnica que se desea implementar como solución al objetivo planteado en este trabajo.

Como conclusión de este acápite se puede decir que existen diferentes herramientas que realizan pruebas de caja blanca, sin embargo el enfoque de estas herramientas no están dirigidas hacia la técnica del camino básico, que fue la escogida como solución al problema planteado, por lo que no es posible contar con ellas como posible solución al objetivo trazado en esta investigación. Sin embargo se pudo constatar la herramienta *Program Exploration*, de la compañía *Microsoft* que si enfoca su objetivo en la realización de la técnica del camino básico, sin embargo no se ha podido contar con esta herramienta ya que no aparece disponible para Cuba en ningún sitio.

Por lo expuesto anteriormente el autor de la tesis considera que surge la necesidad de crear una herramienta que contribuya a automatizar el proceso de pruebas de caja blanca a los productos desarrollados en el departamento Señales Digitales, específicamente la técnica del camino básico. Con ella se lograría analizar gran cantidad de código en poco tiempo. Se analizarían diferentes lenguajes de programación en ella y evitaría posibles errores humanos en el desarrollo del proceso de pruebas de caja blanca que se desea implementar en el grupo de Calidad del departamento Señales Digitales. Por ello resulta importante una herramienta de este tipo que constituiría el primer paso de avance hacia la automatización del proceso de pruebas de caja blanca en la UCI y el país.

[6] Empresa multinacional de origen estadounidense dedicada al sector de la informática.

[7] Es un programa que agrupa un conjunto de herramientas que viabilizan la labor de los programadores.

Conclusiones del capítulo

En este capítulo se expone la importancia de las pruebas que se realizan en cada una de las etapas de desarrollo del software, mediante un proceso de aseguramiento de la calidad bien pensado y planificado. Se enfatiza en la prueba de caja blanca, centrándose dentro de esta, en la técnica de camino básico, que fue seleccionada para realizar la aplicación que se propone. Esta técnica, permite obtener una medida certera de la complejidad lógica de un software y definir los caminos independientes que han de recorrerse para que la función sea ejecutada en todas sus variantes. Por lo que se conocerá la cantidad de casos de pruebas a diseñar para efectuar una buena prueba.

Se analizaron algunas de las herramientas cuyo objetivo es realizar pruebas al código fuente de las aplicaciones. Las cuales arrojaron como resultado que las herramientas que realizan la técnica del camino básico son muy escasas y no se encuentran disponibles. Por lo que no es posible contar con ellas para agilizar el proceso de pruebas de caja blanca en el grupo de calidad del centro GEySED.

CAPÍTULO 2: TECNOLOGÍAS Y TENDENCIAS ACTUALES.

Introducción

En este capítulo se describen las principales tendencias y tecnologías actuales en el mundo de la informática. Se muestran características de algunas herramientas, con el fin de realizar una comparación para mostrar las ventajas y desventajas de cada una de ellas. Se seleccionan aquellas que mejor se adecúan a los objetivos fundamentales de este trabajo.
A continuación se define qué metodología de desarrollo de software es la más idónea para la construcción del sistema que se pretende desarrollar.

2.1 Metodologías de desarrollo de software

Las metodologías de desarrollo de software surgen ante la necesidad de utilizar procedimientos, técnicas y herramientas para proporcionar una disciplina a todo el proceso de desarrollo de un sistema. Ofrece un conjunto de métodos para cada actividad que se realiza dentro de las distintas fases por la que atraviesa un proyecto, además de establecer estándares para toda la organización de los requerimientos de recolección, diseño, programación y pruebas. Existe gran variedad de metodologías para la creación de un software, separadas en dos grandes grupos:

- **Metodologías pesadas:** estas metodologías son orientadas al control de los procesos estableciendo rigurosamente las actividades a desarrollar, herramientas a utilizar y notaciones que se usarán.
- **Metodologías ligeras:** estas metodologías son orientadas a la interacción con el cliente y el desarrollo incremental del software, mostrando versiones parcialmente funcionales del software al cliente en intervalos de corto tiempo, para que pueda evaluar y sugerir cambios en el producto mientras se desarrolla (8).

A continuación se exponen las principales metodologías utilizadas en el centro GEySED para una comparación entre ellas y determinar cuál resultaría más factible de utilizar para el desarrollo de este trabajo.

2.1.1 RUP (*Rational Unified Process*)

En la actualidad resulta muy difícil omitir la utilización de metodologías para el desarrollo y creación de aplicaciones, por ende resulta esencial seguir metodologías que conlleven a una competitividad durante el proceso de desarrollo del software.

El proceso unificado de desarrollo de software (RUP) por sus siglas en inglés, constituye la metodología estándar más utilizada para el análisis, diseño, implementación y documentación de sistemas orientados a objetos. Proporciona una aproximación disciplinada a la asignación de tareas y responsabilidades. RUP actúa como modelo y puede ser adaptado y extendido (9).

Como una entre varias metodologías de desarrollo, RUP presenta una serie de características esenciales que lo definen:

- **Proceso dirigido por los Casos de Uso:** los casos de uso reflejan lo que los usuarios futuros necesitan y desean, lo cual se capta cuando se modela el negocio y se representa a través de los requerimientos. A partir de aquí los casos de uso guían el proceso de desarrollo ya que los modelos que se obtienen, como resultado de los diferentes flujos de trabajo, representan la realización de los casos de uso.
- **Proceso Centrado en la Arquitectura**: la arquitectura muestra la visión común del sistema completo en la que el equipo de proyecto y los usuarios deben estar de acuerdo, por lo que describe los elementos del modelo que son más importantes para su construcción, los cimientos del sistema que son necesarios como base para comprenderlo, desarrollarlo y producirlo económicamente.
- **Proceso Iterativo e Incremental:** propone que cada fase se desarrolle en iteraciones. Una iteración involucra actividades de todos los flujos de trabajo, aunque desarrolla fundamentalmente algunos más que otros. Es práctico dividir el trabajo en partes más pequeñas o mini proyectos. Cada mini proyecto es una iteración que resulta en un incremento. Las iteraciones hacen referencia a pasos en los flujos de trabajo, y los incrementos, al crecimiento del producto. Cada iteración se realiza de forma planificada es por eso que se dice que son mini proyectos.

RUP divide el desarrollo en 4 fases que definen su ciclo de vida, estas fases son:

- **Inicio:** se comprenden los requisitos y determinan el alcance y visón del proyecto.
- **Elaboración:** se estudia en profundidad el dominio del problema así como se define una arquitectura básica.
- **Construcción:** se documenta el sistema construido así como el manejo del mismo.
- **Transición:** se libera el producto y se entrega al cliente para su uso real.

RUP concentra las tareas o actividades en grupos lógicos definiéndose nueve flujos de trabajo principales, los seis primeros denominados flujos de ingeniería y los tres restantes flujos de apoyo (2).

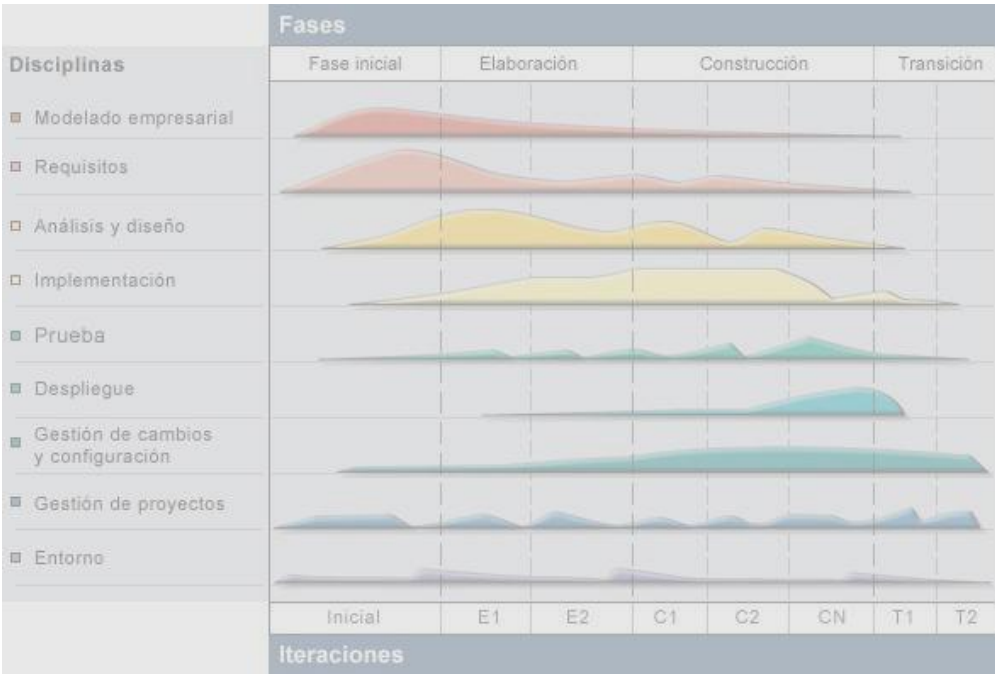


Figura 3. Disciplina, fases e iteraciones de RUP.

Resulta importante mencionar que RUP es una metodología que soporta el paradigma de programación orientado a objetos. Se puede aplicar a proyectos con disímiles características ya que es un marco de proceso configurable para satisfacer necesidades específicas. Además implementa las mejores prácticas de desarrollo de software y es altamente difundido y utilizado por los desarrolladores de software.

2.1.2 Programación Extrema.

Extreme Programming (XP) por sus siglas en inglés, es una metodología de desarrollo de software de tipo ágil, que promueve prácticas que son adaptativas en vez de predictivas, centradas en las personas o en los equipos, iterativas, orientadas hacia prestaciones y hacia la entrega, de comunicación intensiva y requieren que el negocio se involucre en forma directa (10).

La metodología XP se centra en el trabajo en equipo, contribuye a lograr una buena comunicación entre el cliente y el equipo de desarrollo, trayendo consigo un mejor clima en el área de trabajo; instruye a sus desarrolladores pues una de sus principales tareas es el aprendizaje de los mismos. XP se fundamenta de una retroalimentación continua entre el equipo de trabajo y el cliente (11).

Figura 4. Fases de la Metodología XP.

Las características esenciales de XP son las siguientes: historias de usuario (HU), roles, proceso y prácticas (12).

HU: es la técnica utilizada para especificar las necesidades del software. Se trata de tarjetas de papel en las cuales el cliente describe brevemente las características que el sistema debe poseer, sean funcionales o no funcionales. El tratamiento de las HU es muy dinámico y flexible. Cada HU es lo suficientemente comprensible y delimitada para que los programadores puedan implementarla en unas semanas.

Roles: los roles asociados a esta metodología de desarrollo son:

- Programador: el programador escribe las pruebas unitarias y produce el código del sistema.
- Cliente: escribe las HU y las pruebas funcionales para validar su implementación. Además, asigna la prioridad a las HU y decide cuáles se implementan en cada iteración centrándose en aportar mayor valor al negocio.
- Encargado de pruebas: ayuda al cliente a escribir las pruebas funcionales. Ejecuta las pruebas regularmente, difunde los resultados en el equipo y es responsable de las herramientas de soporte para pruebas.
- Encargado de seguimiento: proporciona realimentación al equipo. Verifica el grado de acierto entre las estimaciones realizadas y el tiempo real dedicado, con el objetivo de mejorar futuras estimaciones. Realiza el seguimiento del progreso de cada iteración.
- Entrenador: es responsable del proceso global. Debe proveer guías al equipo de forma que se apliquen las prácticas de XP y se siga el proceso correctamente.
- Consultor: es un miembro externo del equipo con un conocimiento específico en algún tema necesario para el proyecto.

- Gestor: es el vínculo entre clientes y programadores, ayuda a que el equipo trabaje efectivamente creando las condiciones adecuadas. Su labor esencial es la coordinación.

Proceso: el ciclo de desarrollo consiste en los siguientes pasos:

- El cliente define el valor de negocio a implementar.
- El programador estima el esfuerzo necesario para su implementación.
- El cliente selecciona qué construir, de acuerdo con sus prioridades y las restricciones de tiempo.
- El programador construye ese valor de negocio.
- Vuelve al principio.
- El ciclo de vida ideal de XP consiste de seis fases:
- Exploración.
- Planificación de entrega (Release).
- Iteraciones.
- Producción.
- Mantenimiento.
- Fin del Proyecto.

Por último las **Prácticas:** la principal suposición que se realiza en XP es la posibilidad de disminuir la mítica curva exponencial del costo del cambio a lo largo del proyecto, lo suficiente para que el diseño evolutivo funcione.

El objetivo de XP es muy simple: la satisfacción del cliente. Esta metodología trata de dar al cliente el software que él necesita y en el momento que lo necesita. Por tanto, se debe responder muy rápido a las necesidades del cliente, incluso cuando los cambios sean al final de ciclo de la programación. XP se encarga de potenciar al máximo el trabajo en grupo. Tanto los jefes de proyecto, los clientes y los desarrolladores, son parte del equipo y están involucrados en el desarrollo de software.

XP mejora un proyecto de software de cuatro formas esenciales: comunicación, simplicidad, retroalimentación y coraje. Una de las características de la programación extrema es que es imposible prever todo antes de comenzar a programar; es imposible o si lo fuera es demasiado costoso e innecesario, ya que muchas veces se gasta demasiado tiempo y recursos en cambiar la documentación de la planificación para que se parezca al código. Para evitar esto, XP intenta implementar una forma de trabajo donde se adapte fácilmente a las circunstancias (11).

2.1.3 Selección de una metodología de desarrollo

Se llegó a la conclusión de que la metodología de desarrollo RUP es la más conveniente para el desarrollo de este trabajo. Se selecciona esta metodología fundamentalmente por la extensa documentación que genera. Teniendo en cuenta que se propone construir un prototipo funcional, ya que se seleccionaron los casos de usos críticos propuestos por el analista. Resulta conveniente utilizar esta metodología ya que permite obtener una documentación extensa, que servirán de guía para aquellas personas que continuarán el desarrollo de la herramienta para posteriores mejoras a la versión construida, lo que posibilitará la continuidad de la investigación realizada.

2.2 Lenguaje Unificado de Modelado

UML es un lenguaje para especificar, construir, visualizar y documentar los artefactos (información que es utilizada o producida mediante un proceso de desarrollo de software) de un sistema de software orientado a objetos, que por su potencialidad se ha convertido en un estándar. Proporciona un vocabulario y una regla para permitir una comunicación. En este caso, este lenguaje se centra en la representación gráfica de un sistema (13).

Los objetivos de UML son muchos pero se pueden resumir en sus funciones:

- Visualizar: permite expresar de una forma gráfica un sistema de forma que otro lo puede entender.
- Especificar: permite especificar cuáles son las características de un sistema antes de su construcción.
- Construir: a partir de los modelos especificados se pueden construir los sistemas diseñados.
- Documentar: los propios elementos gráficos sirven como documentación del sistema desarrollado que pueden servir para su futura revisión.

Un modelo UML está compuesto por tres clases de bloques de construcción:

- Elementos: son abstracciones reales o ficticias (objetos, acciones).
- Relaciones: relacionan los elementos entre sí.
- Diagramas: son colecciones de elementos con sus relaciones.

UML resuelve de forma satisfactoria el viejo problema del desarrollo de software como es su modelado gráfico. Además ha llegado a una solución unificada basada en lo mejor que había hasta el momento, lo cual lo hace más excepcional.

Una vez seleccionado el lenguaje de modelado se escoge la herramienta que permita realizar la representación gráfica del sistema.

2.3 Herramienta Case

La Ingeniería de Software Asistida por Computadoras (CASE por sus siglas en inglés) son herramientas que permite la automatización de metodologías paso a paso para el desarrollo de software y sistemas. Su objetivo es reducir los niveles de trabajo repetitivos que los diseñadores necesitan hacer, también facilitan la creación de documentación estructurada y la coordinación de los esfuerzos de desarrollo del equipo de trabajo.

2.3.1 Visual Paradigm

Visual Paradigm, es una herramienta UML profesional fácil de usar. Soporta el ciclo de vida completo del desarrollo de software: análisis y diseño orientados a objetos, construcción, pruebas y despliegue. El software de modelado UML ayuda a una más rápida construcción de aplicaciones de calidad, mejores y a un menor coste. Permite dibujar todos los tipos de diagramas de clases, código inverso, generar código desde diagramas y generar documentación (14).
Dentro de las principales características de la herramienta están:

- Soporte de UML versión 2.0.
- Disponibilidad de integrarse en los principales IDEs.
- Soporta una gama de lenguajes en la Generación de Código e Ingeniería Inversa.
- Diagramas de Procesos de Negocio - Proceso, Decisión, Actor de negocio.
- Diagramas de flujo de datos.
- Generador de informes para generación de documentación.
- Distribución automática de diagramas.
- Importación y exportación de ficheros.
- Editor de figuras.

Visual Paradigm ofrece un entorno amigable para el usuario, permite crear fácilmente cualquier tipo de diagrama. Incluye gran variedad de estereotipos para la creación de diagramas de fácil entendimiento, los que organiza automáticamente.

2.3.2 Rational Rose

Rational Rose se ha convertido en una herramienta de modelado visual para el análisis y diseño de sistemas basados en objetos. Con el uso del lenguaje común de modelado UML, facilita el trabajo para el equipo de desarrollo y garantiza mayor eficiencia y calidad del trabajo. Permite generar código en diferentes lenguajes de programación partiendo de modelado UML. Además, hace posible efectuar la ingeniería inversa, es decir, obtener información del diseño de un software partiendo de su código.

Rational Rose permite mantener la consistencia de los modelos del sistema software, realiza chequeo de la sintaxis UML y genera documentos automáticamente (15).

2.3.3 Herramienta Seleccionada

Durante el estudio realizado se llegó a la conclusión de que ambas herramientas son muy similares, y poseen muchas ventajas para realizar el modelado del software. Sin embargo Rational Rose es una herramienta propietaria y no sería factible su utilización para el desarrollo de este trabajo. Esto iría en contra de las políticas por las que aboga la universidad y el departamento Señales Digitales, que son la de utilización de herramientas gratuitas, a fin con lograr una soberanía e independencia tecnológica.
Visual Paradigm en cambio es una herramienta con licencia libre, y además es multiplataforma. Tiene una interfaz amigable que permite diseñar todos los artefactos que genera de una forma rápida y con calidad. Es por ello que se considera como la mejor opción para el modelado de la herramienta que se desea desarrollar.
A continuación se escoge el lenguaje de programación a utilizar en la implementación de la herramienta.

2.4 Lenguajes de Programación

Los lenguajes de programación a través de los años han cobrado una importancia vital en el desarrollo de la informática como ciencia, cada vez más el mundo se rige por la automatización de los procesos que ocurren a diario en la vida cotidiana. Los lenguajes de programación han tenido un gran desarrollo desde sus inicios, comenzando por los lenguajes de bajo nivel que son aquellos que se asemejan más al lenguaje utilizado por la máquina, estos lenguajes son totalmente dependientes de esta ya que no pueden ser migrados ni se pueden ejecutar en otras máquinas.
Con el tiempo surgieron los lenguajes de alto nivel, los que son más usados hoy día por la mayoría de los programadores del mundo, estos propician un lenguaje más natural que el de la máquina. A continuación se describen los principales lenguajes de programación utilizados en el mundo y la UCI.

2.4.1 Lenguaje C#

Este lenguaje surge como una versión avanzada de C y de C++ y se ha diseñado especialmente para el entorno .NET, es un lenguaje orientado a objetos empleado por programadores de todo el mundo, el cual marca un paso muy importante en la evolución de los lenguajes de programación. C# amplía las capacidades de C, C++, Visual Basic y Java, mezcla la potencia de C, las capacidades de orientación a objetos de C++ y la interfaz gráfica de Visual Basic, además de que los programas se compilan a

bytecode(16). C# también proporciona la capacidad de generar componentes de sistema duraderos en virtud de las siguientes características:

- Gran robustez, gracias a la recolección de elementos no utilizados (liberación de memoria) y a la seguridad en el tratamiento de tipos.
- Seguridad implementada por medio de mecanismos de confianza intrínsecos del código.
- Plena compatibilidad con conceptos de metadatos extensibles (17).

Además, es posible interaccionar con otros lenguajes, entre plataformas distintas, y con datos heredados, en virtud de las siguientes características:

- Plena interoperabilidad por medio de los servicios de .NET Framework con un acceso limitado basado en bibliotecas.
- Compatibilidad con XML para interacción con componentes basados en tecnología Web.
- Capacidad de control de versiones para facilitar la administración y la implementación (17).

2.4.2 Lenguaje C++

C++ es un lenguaje de programación, diseñado a mediados de los años 1980, por Bjarne Stroustrup, como extensión del lenguaje de programación C. Actualmente existe un estándar, denominado ISO C++, al que se han adherido la mayoría de los fabricantes de compiladores más modernos. Existen también algunos intérpretes como ROOT (17). Las principales características del C++ son: el soporte para programación orientada a objetos, el soporte de plantillas o programación genérica (*templates*), la portabilidad, brevedad, programación modular, compatibilidad con C y velocidad. Se puede decir que C++ es un lenguaje que abarca tres paradigmas de la programación: la programación estructurada, la programación genérica y la programación orientada a objetos (18). Además, se trata de un lenguaje de programación estandarizado (ISO/IEC 14882:1998), ampliamente difundido, y con una biblioteca estándar C++, que lo ha convertido en un lenguaje universal, de propósito general, y ampliamente utilizado tanto en el ámbito profesional como en el educativo. Además, posee una serie de propiedades difíciles de encontrar en otros lenguajes de alto nivel entre las cuales se destacan la posibilidad de redefinir los operadores (sobrecarga de operadores) y la identificación de tipos en tiempo de ejecución (RTTI). C++ está considerado por muchos como el lenguaje más potente, debido a que permite trabajar tanto a alto como a bajo nivel, sin embargo, es a su vez uno de los que trae menos automatismos (obliga a hacerlo casi todo manualmente al igual que C) lo que dificulta mucho su aprendizaje .

2.4.3 Lenguaje Java

Este lenguaje es desarrollado por la compañía *Sun Microsystems* a principios de los años 90. Está basado en el lenguaje C++ eliminando algunos de sus errores, su propósito fue obtener un producto pequeño; que permita su ejecución en uno o varios sistemas operativos tanto a nivel de código fuente como a nivel de código binario (19).

Java presenta compiladores que generan archivos de código binario especial llamado *bytecode* que es un código máquina de bajo nivel, el cual puede ser viable e interpretado como lenguaje por un microprocesador.

Todas las aplicaciones desarrolladas en Java deben ser ejecutadas en la máquina virtual de Java, Java Virtual Machine (JVM) por sus siglas en inglés, ejecutable en varios sistemas operativos, cuya función es interpretar y ejecutar instrucciones expresadas en código bytecode pre compiladas inicialmente. JVM provee un conjunto de instrucciones para las diferentes tareas entre las que se encuentran:

- Manejo de excepciones.
- Gestión de Pilas.
- Carga y almacenamiento.
- Liberación de recursos usados.
- Conversiones de tipos y operaciones aritméticas.
- Creación y manipulación de objetos.
- Transferencia de control de programa.

Java es un lenguaje cuyas principales características demuestran que es un lenguaje simple, orientado a objetos y distribuido ya que brinda una colección de clases para su uso en aplicaciones de red. Es un poco más robusto al simplificar la gestión de memoria, para ello proporciona numerosas comprobaciones en compilación y tiempo de ejecución. Es un lenguaje seguro ya que la máquina virtual, al ejecutar el código java, realiza comprobaciones de seguridad, además el propio lenguaje carece de características inseguras, como por ejemplo los punteros. Java soporta sincronización de múltiples hilos de ejecución y proporciona mecanismos de carga dinámica de clases en tiempo de ejecución.

Ha sido un lenguaje exitoso que se ha desarrollado rápidamente debido a que grandes cantidades de empresas han colaborado y dado su aporte para enriquecerlo, siendo más potente en el desarrollo de aplicaciones web. En el 2007 Sun Microsystems liberó la mayoría de las tecnologías Java bajo la licencia GNU GPL (en inglés *General Public License*), de modo que es un lenguaje gratuito que distribuye su producto base, el J2SE (*Java 2 Standard Edition* por sus siglas en inglés), que incluye

herramientas para generar programas Java con depurador, compilador y herramienta para la documentación de código (20).

2.4.4 Selección de un lenguaje de programación.

Se llegó a la conclusión de que para el desarrollo de este trabajo es conveniente la utilización del lenguaje C#. Este lenguaje simplifica en gran medida las labores de implementación. Esto se debe que a diferencia del C++, este realiza una recolección automática de basura, además elimina el uso obligatorio de punteros, no existen variables ni funciones globales porque todo pertenece a una clase.
Respecto a Java permite el uso de punteros cuando realmente se necesita, así como acceder a bibliotecas que no se ejecutan sobre la máquina virtual. También se usan indizadores que permiten acceder a cualquier objeto como si se tratase de un arreglo.
En el siguiente apartado, una vez definido el lenguaje de programación a utilizar, se escoge el entorno de desarrollo utilizado en la implementación de la herramienta.

2.5 Entornos de desarrollo

Un Entorno de Desarrollo Integrado, Integrated Development Environment (IDE), por sus siglas en inglés, es un programa que agrupa un conjunto de herramientas que viabilizan el trabajo de los programadores. Los componentes esenciales que los definen son:

- Editor de texto (código).
- Compilador.
- Intérprete.
- Depurador.
- Herramientas de automatización (completamiento de código y navegación rápida dentro del código).
- Sistema de control de versiones.
- Diseñador de interfaces gráficas.
- Agregación de herramientas externas.
- Soporte para varios lenguajes de programación y plataformas de desarrollo.

No todos los entornos poseen estas propiedades ni las implementan al mismo nivel, pero esas son las más usuales. A continuación se describen los entornos de desarrollo para generar aplicaciones en lenguaje C#.

2.5.1 Microsoft Visual Studio

Es un IDE desarrollado por la Microsoft Corporation. Entre sus ventajas está que puede soportar varios lenguajes de programación como C#, C++, Visual Basic.NET y ASP.NET.

Este entorno permite desarrollar aplicaciones tanto web, como de escritorio en cualquier entorno que soporte la plataforma .NET. Las aplicaciones pueden intercomunicarse entre diferentes computadoras, páginas web y dispositivos móviles (21).

Visual Studio .NET es uno de los puntos de referencia para la productividad de los programadores. Con un único entorno de programación (IDE) integrado para todos los lenguajes, las organizaciones de programación pueden aprovechar las ventajas de un cuadro de herramientas, un depurador y una ventana de tareas comunes, reduciendo enormemente la curva de aprendizaje del programador y garantizado que siempre puedan elegir el lenguaje más apropiado para sus tareas y conocimientos.

Con la función para completar instrucciones de *IntelliSense* y la comprobación automática de errores de sintaxis, Visual Studio .NET informa a los programadores cuando el código es incorrecto y proporciona el dominio inmediato de las jerarquías de clases y las API. Con el Explorador de soluciones, los programadores pueden reutilizar fácilmente código a través de diferentes proyectos e incluso, generar soluciones multilenguaje que satisfagan con mayor eficacia sus necesidades empresariales.

Una de las mejores características que se agregaron a Visual Studio es la capacidad de especificar el Framework sobre el cuál se desea compilar. En las cajas de diálogo *Advanced Compiler Settings* (VB) y *Advanced Build Settings* (C#), ahora existe un nuevo campo denominado Target Framework que permite seleccionar lo siguiente (22):

- .NET Framework 2.0
- .NET Framework 3.0
- .NET Framework 3.5

Como desventaja presenta que es un IDE propietario, y que sus aplicaciones están encaminadas hacia el entorno de Windows.

2.5.2 SharpDevelop

Es un IDE que se encuentra disponible solo para entornos del sistema operativo Windows. Su ventaja radica en que se ofrece de forma gratuita bajo una licencia de código abierto. Soporta el desarrollo de

aplicaciones escritas en C#, Visual Basic.NET y BOO[8]. A continuación se presentan las principales características que presenta como entorno de desarrollo (23):

- Completamiento de código.
- Depurador incorporado.
- Plantillas de proyectos.
- Diseñador de formularios.
- Tiene herramientas para ir a definición, encontrar referencias y renombrado.
- Integración con herramientas externas.
- Posee analizadores para ensamblado.

Una de las ventajas que tiene SharpDevelop es que sus archivos de proyectos son compatibles con Visual Studio Express y Visual Studio 2005. Es posible trabajar con proyectos indistintamente en uno u otro ambiente sin cambiar el formato de archivos de proyecto y de código. Con ello brinda una opción para aquellos desarrolladores que no tienen la posibilidad de adquirir el Visual Studio.

2.5.3 MonoDevelop

Es un entorno de desarrollo libre y gratuito desarrollado por Novell y los impulsores de Mono con el objetivo de adaptar las funcionalidades y ventajas de SharpDevelop. Permite desarrollar aplicaciones tanto web como de escritorio en Linux relativamente rápido. Además permite portar el código de aplicaciones .NET creadas en Visual Studio a Linux, y mantiene un solo código base para todas las plataformas de forma sencilla (24). Es usado por los desarrolladores de Mono en algunas distribuciones de Linux y de Mac OS.

Sus principales características son (25):

- Soporta los lenguajes C#, Visual Basic .NET, C/C++, Java y BOO[9].
- Manejo de clases.
- Completamiento de código.
- Funciones de navegación por el código.
- Diseñador de interfaz gráfica para *GTK#*[10].

[8] Lenguaje de programación orientado a objetos de tipos estáticos muy parecido a Phyton e integrable con Microsoft.NET y Mono.

[9] Lenguaje de programación orientado a objetos de tipos estáticos muy parecido a Phyton e integrable con Microsoft .NET y Mono.

- Control de versiones integrada con soporte para *Subversion*.
- Pruebas unitarias integradas con el componente *NUnit*.
- Posibilidad de crear proyectos ASP.NET.
- Editor y explorador de bases de datos integrado.
- Integración con la herramienta Monodoc para la generación de documentación sobre las clases.
- Compatible con Visual Studio.
- Control de empaquetamiento de código.
- Herramientas de líneas de comando para compilar y administrar proyectos.
- Agregación de herramientas externas para su enriquecimiento.
- Incluye ayuda de la aplicación para el usuario.

Como desventaja se puede mencionar que no dispone de funciones de depuración como la de establecer puntos de ruptura, o ejecutar paso a paso. Por ello es que en ocasiones puede ser difícil encontrar un fallo específico dentro del código. Actualmente ya existe una versión de MonoDevelop para Windows (26).

2.5.4 Selección de un entorno de desarrollo

Luego de analizar los distintos entornos de desarrollo expuestos anteriormente, se llegó a la conclusión que para el desarrollo de este trabajo la mejor opción es utilizar MonoDevelop. Esto se debe a que Visual Studio es considerado uno de los mejores IDE, según los principales programadores. Pero posee el inconveniente de que es una herramienta propietaria, por lo que va en contra de las políticas de la Universidad y el departamento Señales Digitales, que abogan por alcanzar la soberanía tecnológica y la utilización de software gratuitos. Por su parte SharpDevelop tiene el inconveniente de que aunque es una herramienta libre, sus aplicaciones están hechas para un entorno de Windows, lo que limita la posibilidad de crear una herramienta que se emplee en varias plataformas.
A continuación se escoge el *framework* de desarrollo utilizado por el entorno de desarrollo para la implementación de la herramienta.

[10] Protegido por la licencia GNU LGPL es un conjunto de bibliotecas y rutinas para desarrollar interfaces gráficas principalmente para sistemas operativos de la familia Linux aunque se puede utilizar otros.

2.6 Framework de desarrollo

Un framework es una estructura conceptual y tecnológica de soporte definida, normalmente con artefactos o módulos de software concretos, en base a la cual otro proyecto de software puede ser organizado y desarrollado. También se puede definir como un conjunto de bibliotecas orientadas a la reutilización a gran escala de componentes de software para el desarrollo rápido de aplicaciones (27).
Para el desarrollo de este trabajo se utilizó como framework Mono, ya que es de código abierto desarrollado para Linux, como alternativa del framework .NET. Entre los componentes básicos de Mono se encuentra el compilador de C#, la biblioteca de clases y una máquina virtual, que trae integrado la recolección de basura y el entorno de desarrollo MonoDevelop, el cual se seleccionó para la implementación de la herramienta.

Conclusiones del capítulo

En este capítulo se realizó un análisis de las principales herramientas y tecnologías a utilizar para la construcción de la herramienta. Se identificaron las ventajas y desventajas de cada una de ellas lo cual posibilitó seleccionar las mejores opciones para la construcción de la aplicación. Además se precisó que cada una de estas tecnologías y herramientas cumplieran con las normas y políticas de desarrollo de software, que se utilizan en la Universidad de las Ciencias Informáticas. Con el objetivo de lograr la soberanía e independencia tecnológica.
Para ello se definió como metodología de desarrollo RUP, la cual utiliza como lenguaje de modelado UML versión 2.0. Para la representación del diseño de la aplicación, se usó como herramienta de modelado Visual Paradigm para UML versión 8.0. Para una mayor rapidez y facilidad a la hora de implementar la herramienta se utilizó C# como lenguaje de programación y MonoDevelop 2.4 como entorno de desarrollo. El cual será integrado al framework de desarrollo Mono 3.5.

CAPÍTULO 3: ANÁLISIS Y DISEÑO DEL SISTEMA.

Introducción

En este capítulo se realiza la descripción y el diseño del prototipo funcional que se presenta en este trabajo. Se elabora el Modelo de Dominio para una mejor comprensión del entorno donde se ubica el prototipo funcional. Para ello se describen una serie de conceptos que son agrupados en dicho modelo. Se especifican los casos de usos del sistema. Se muestra el diagrama de Casos de Uso del Sistema, los diagramas de clases del análisis y los diagramas de colaboración y secuencia de cada caso de uso.

3.1 Modelo de dominio

El modelo de dominio muestra las clases conceptuales más significativas en el dominio del problema. Está considerado como un subconjunto del modelo de negocio, ya que se centra en una parte del negocio, la relacionada con el ámbito del proyecto.

Durante el desarrollo del prototipo funcional no se pudo contactar procesos bien definidos en el entorno del negocio. Esto provocó que se hiciera más difícil determinar los elementos más importantes del prototipo y sus conexiones. Por ello se hizo necesario un modelado del dominio, donde se pueden identificar entidades, objetos y eventos involucrados en ese entorno. A continuación se muestra una descripción de los conceptos fundamentales asociados al dominio del problema.

3.1.1 Descripción de los conceptos fundamentales

Las **pruebas de caja blanca** se centran en el código fuente de las aplicaciones, donde se controlan determinados parámetros de calidad presentes en el mismo.

Se denomina **fichero** a un conjunto de bits almacenado en un dispositivo periférico. Facilitan una manera de organizar los recursos usados para almacenar permanentemente datos de un sistema informático.

Un **código** en el ámbito informático se conoce por un texto desarrollado en un lenguaje de programación que debe ser compilado o interpretado para poder ejecutarse en un ordenador.

Se denomina **estructura de código** a la porción de texto escrito generalmente por una persona, se utiliza como base para generar otro código que será interpretado o ejecutado por un ordenador.

El **grafo de flujo** es una representación de un programa usado para el diseño de casos de prueba estructurales. Se compone de nodos y aristas.

Los **nodos** representan ninguna, una o varias sentencias de decisión o bifurcación como también se le conoce.

Las **aristas** son líneas que unen los nodos.

La **complejidad ciclomática** es la medición cuantitativa de un programa. Este valor coincide con la cantidad máxima de casos de pruebas que se deben diseñar para asegurar que se ejecute cada camino del programa.

Los **caminos independientes** son cualquier camino del programa que introduce como mínimo un nuevo conjunto de sentencias o una nueva condición, este camino debe recorrer alguna arista que no haya sido visitada anteriormente.

Una vez definidos los conceptos anteriormente expuestos, estos se agrupan en el siguiente modelo de dominio para un mayor entendimiento del dominio del problema.

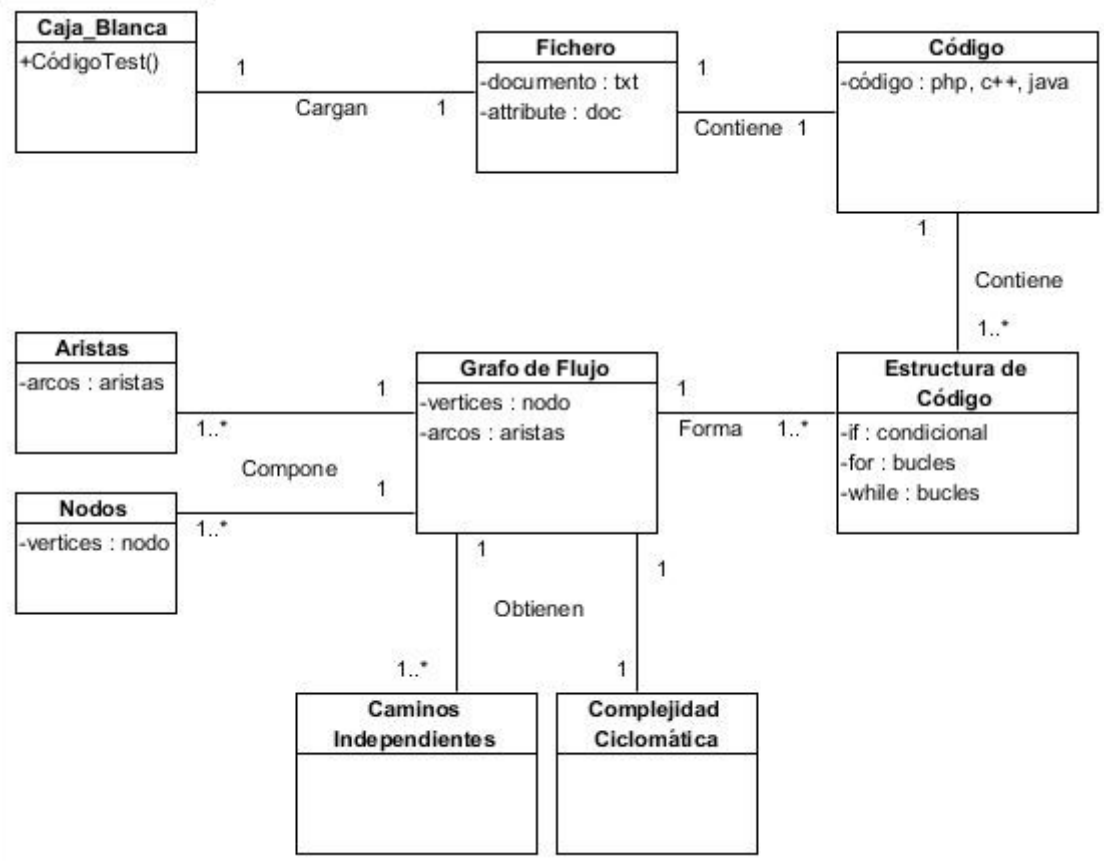


Figura 5. Diagrama de modelo de dominio.

3.2 Especificación de los requisitos del prototipo funcional

La captura de los requisitos de un software es uno de los procesos críticos en la ingeniería de software. Por ello los requisitos del sistema se tratan como uno de los aspectos más importantes a tener en cuenta en la construcción de un software. Ellos constituyen la base, el fundamento y el argumento para desarrollar el modelado del sistema.

3.2.1 Requisitos funcionales

Los requerimientos funcionales de un sistema describen lo que este debe hacer, son condiciones o capacidades que el sistema debe cumplir.

A continuación se presentan los requisitos que debe cumplir el sistema.

RF 1_Cargar Código: permite seleccionar desde un archivo un código ya escrito, para una vez cargado, aplicar la técnica del camino básico.

RF 2_Guardar Código: permite guardar todos los cambios realizados a cualquier código analizado.

RF 3_Nuevo Código: permite escribir un nuevo código directamente en la aplicación.

RF 4_Generar Grafo de Flujo: una vez seleccionado el código que se desea analizar, la herramienta permite generar un grafo de flujo, para lo cual crea un objeto de tipo grafo que contiene toda la información referente al código. Este proceso no muestra ningún resultado, solo se crea un objeto.

RF 5_ Mostrar Grafo de Flujo: una vez generado el grafo de flujo, la herramienta permite mostrar dicho grafo para tener una idea de cómo quedó conformado el mismo.

RF 6_ Calcular Complejidad Ciclomática: una vez generado el grafo de flujo, la herramienta permite calcular la complejidad ciclomática, para lo cual procede a contar la cantidad de nodos y aristas, para mostrar en un cuadro de texto el valor de la misma.

RF 7_ Mostrar Caminos Independientes: la herramienta permite mostrar los caminos independientes del grafo de flujo, para ello selecciona aquellos caminos los cuales contenga al menos una arista que no haya sido recorrida anteriormente.

3.2.2 Requisitos no funcionales

Los requisitos no funcionales de un software se refieren a las propiedades emergentes del sistema cómo son: la fiabilidad, seguridad, el tiempo de respuesta, los requisitos de hardware, así como la representación de datos que se utiliza en las interfaces del software. Los requisitos no funcionales

definidos para la creación del prototipo funcional para realizar pruebas de caja blanca se muestran a continuación.

RNF 1 Usabilidad

- La interfaz debe ser sencilla y amigable de manera que potencie la comodidad del usuario para su trabajo además de que las opciones más usadas presentarán vías rápidas y cómodas de invocarse.
- Debe ser operada por usuarios que deseen probar su código fuente, sin necesidad de tener grandes conocimientos acerca de las pruebas de caja blanca, para ello la herramienta cuenta con una ayuda para guiar a quien la utilice.

RNF 2 Rendimiento

- La herramienta debe poseer la capacidad de analizar grandes cantidades de líneas de código en tres segundos, para ello se realizaron varias pruebas para ver el tiempo de respuesta de la herramienta.

RNF 3 Disponibilidad

- La herramienta debe estar disponible las veinticuatro horas del día, los siete días de la semana.

RNF 4 Restricciones de diseño

- El lenguaje utilizado para la implementación del prototipo funcional Prueba del Camino Básico será C#.
- La herramienta será implementada en el IDE *Mono Develop*.
- Se utilizará como herramienta CASE Visual Paradigm para el modelado de la herramienta.

RNF Software

- En los ordenadores donde se encuentre desplegada la herramienta deberá estar instalado como Sistema Operativo Ubuntu 10.04.
- Framework Mono 3.5

RNF Hardware

- Procesador de 600 MHz o superior.
- 128 MB de memoria RAM.
- Monitor VGA o superior.
- Espacio en disco duro de 30 MB.

Una vez realizado el modelo de dominio y determinados lo requerimientos del prototipo funcional es posible modelar y proponer la herramienta que se desea desarrollar.

3.3 Definición de los Casos de Uso del Sistema

Los casos de uso se utilizan para obtener información de cómo debe trabajar el sistema, o sea son descripciones de las funcionalidades del sistema. Independiente de la implementación, describen bajo la forma de acciones y relaciones, el comportamiento de un sistema desde la perspectiva del usuario o cliente.

3.3.1 Actores del sistema

Se le llama actor a toda entidad externa al sistema que guarda una determinada relación con este y que le exige una funcionalidad. Esto incluye a los operadores humanos, pero también pueden ser sistemas externos, así como entidades abstractas como el tiempo.

Actor	Descripción
Probador	Representa al usuario que hará uso de la herramienta y tendrá la oportunidad de interactuar con todas las funcionalidades de la aplicación.

Tabla 1. Descripción de actores del sistema.

3.3.2 Diagrama de Caso de Uso del Sistema.

El diagrama de casos de uso del sistema sirve para especificar la comunicación y el comportamiento de un sistema, mediante su interacción con los usuarios y/u otros sistemas. A continuación se muestra en la figura 7 el diagrama de caso de uso del sistema perteneciente al desarrollo de este trabajo.

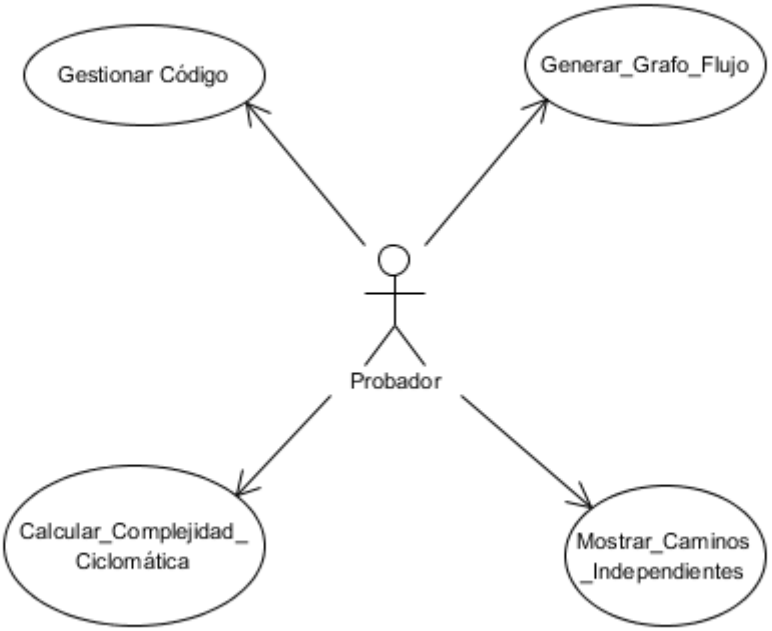


Figura 6. Diagrama de casos de uso del sistema.

3.3.3 Descripción de los Casos de uso del sistema

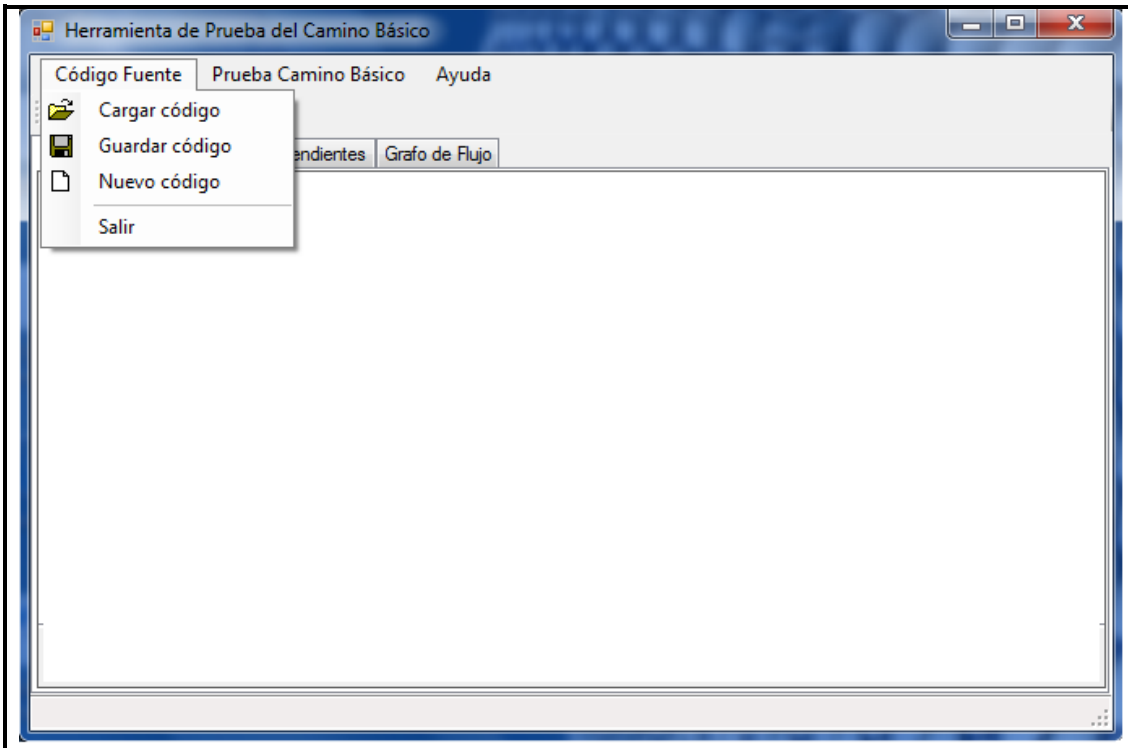
Un caso de uso es una técnica para la captura de requisitos potenciales de un nuevo sistema o una actualización de software. Cada caso de uso proporciona uno o más escenarios que indican cómo debe actuar el sistema con el usuario, para lograr un objetivo específico. A continuación se describen los casos de uso Gestionar Código y Generar Grafo de Flujo. El resto de las descripciones de los casos de uso, se pueden observar en el Anexo 3.

CU Gestionar Código

Caso de Uso:	Gestionar Código.
Actores:	Probador
Resumen:	El caso de uso inicia cuando el probador desea cargar, guardar o escribir un nuevo código en la herramienta, para aplicar la técnica del camino básico. El caso de uso finaliza, cuando el usuario carga, guarda o escribe un nuevo código.

Precondiciones:	
Referencias	RF-1 (RF-1.1, RF-1.2, RF-1.3).
Prioridad	Crítico
Flujo Normal de Eventos	
Acción del Actor	**Respuesta del Sistema**
1. El caso de uso inicia cuando el probador selecciona en el menú la opción “Código Fuente”.	1.1 La herramienta despliega un menú con las opciones “Cargar código”, “Guardar código” y “Nuevo código”.
2. El probador selecciona una de las siguientes opciones: Cargar código. Guardar código. Nuevo código.	2.1 Si selecciona la opción Cargar código, ver sección “Cargar código”. Si selecciona la opción Guardar código, ver sección “Guardar código”. Si selecciona la opción Nuevo código, ver sección “Nuevo código”.
Sección “Cargar código”	
	2.2 La herramienta muestra una ventana con la opción para cargar un código.
3. El probador selecciona el código a cargar, y presiona el botón “Aceptar”.	3.1 Se carga el código seleccionado, y finaliza el caso de uso.
Flujo Alterno	
Acción del Actor	**Respuesta del Sistema**
	3.1 El probador presiona el botón cancelar y la herramienta no carga el código, finalizando así el caso de uso.
Sección “Guardar código”	
	2.2 La herramienta muestra una ventana con la opción para guardar un código.
3. El probador selecciona la ubicación para guardar el código, y presiona el botón	3.1 Se guarda el código y finaliza el caso de uso.

“Aceptar”.	
Flujo Alterno	
Acción del Actor	**Respuesta del Sistema**
	3.1 El probador presiona el botón cancelar y la herramienta no guarda el código, finalizando así el caso de uso.
Sección “Nuevo código”	
	2.2 La herramienta si existe un código escrito o cargado anteriormente, muestra una ventana con la opción de guardar dicho código.
3. El probador guarda el código anterior.	3.1 La herramienta guarda el código anterior y limpia el cuadro de texto para que se escriba el nuevo código, finalizando así el caso de uso.
Flujo Alterno	
Acción del Actor	**Respuesta del Sistema**
3. El usuario no desea guardar el código anterior	3.1 La herramienta no guarda el código anterior, y limpia el cuadro de texto para que se escriba el nuevo código, y finaliza el caso de uso.
Prototipo de Interfaz	



Poscondiciones	Se carga, se guarda o se escribe un nuevo código.

Tabla 2. Descripción del Caso de Uso Gestionar Código.

CU Generar grafo de flujo.

Caso de Uso:	Generar grafo de flujo.
Actores:	Probador.
Resumen:	El caso de uso se inicializa cuando el probador desea generar el grafo de flujo a partir del código fuente. El caso de uso termina cuando se ha generado el grafo de flujo a partir del código.
Precondiciones:	Se debe haber realizado el CU Gestionar Código.
Referencias	RF-2
Prioridad	Crítico.
Flujo Normal de Eventos	
Acción del Actor	**Respuesta del Sistema**
1. El caso de uso inicia cuando el probador selecciona en el menú la opción	1.1 La herramienta despliega un menú con las opciones “Generar Grafo de

"Prueba del Camino Básico"	Flujo", "Calcular Complejidad Ciclomática" y "Mostrar Grafo de Flujo".
2. El probador selecciona una de las siguientes opciones: Generar Grafo de Flujo. Mostrar Grafo de Flujo.	2.1 Si selecciona la opción Generar Grafo de Flujo, ver sección "Generar Grafo de Flujo". Si selecciona la opción Mostrar Grafo de Flujo, ver sección "Mostrar Grafo de Flujo".
Sección "Generar Grafo de Flujo"	
	2.2 La herramienta analiza el fragmento de código cargado y genera los nodos y las aristas necesarias para conformar el grafo de flujo. El caso de uso finaliza una vez generado el grafo.
Flujo Alterno	
Acción del Actor	**Respuesta del Sistema**
	2.2 La herramienta muestra un mensaje indicando que debe haber escrito un código y que este cumpla con la condición que los bloques de código deben ir entre llaves, y finaliza el caso de uso.
Sección "Mostrar Grafo de Flujo"	
	2.2 La herramienta muestra en una ventana, cómo quedó conformado el grafo asociado al código fuente analizado, y finaliza el caso de uso.
Flujo Alterno	
	2.2 La herramienta muestra un mensaje indicando que para mostrar el grafo de flujo se debe haber ejecutado la sección

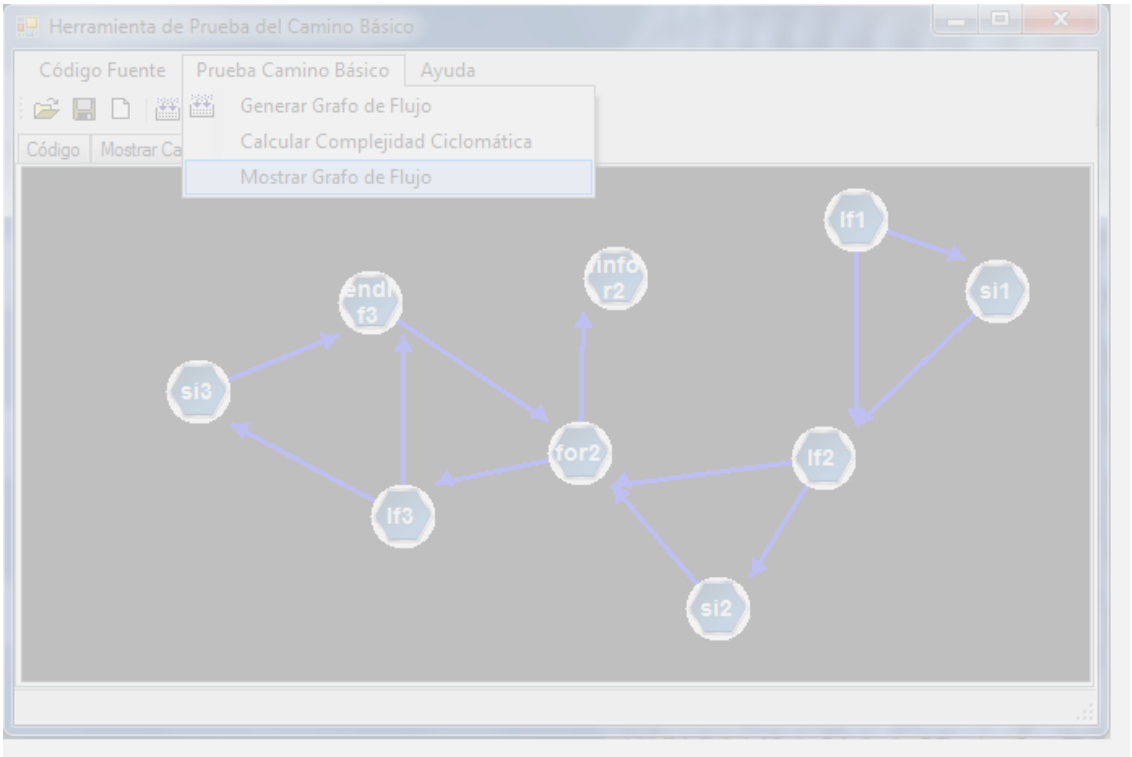
	"Generar Grafo de Flujo", y finaliza el caso de uso.
Prototipo de Interfaz 	
Poscondiciones	Se genera y se muestra el grafo de flujo asociado al código fuente analizado.

Tabla 3. Descripción del Caso de Uso Generar Grafo de Flujo.

Una vez realizado la descripción de los casos de uso, se pasa a la fase del análisis y diseño de la de la herramienta.

3.4 Modelo del Análisis

El modelo del análisis se emplea con el objetivo de representar la estructura global del sistema, sirviendo como una abstracción del Modelo de Diseño. Este artefacto puede ser opcional, pero también tiene la propiedad de ser temporal, su utilidad consiste en que permite un acercamiento visual del sistema (28).

3.4.1 Diagrama de clases del análisis

El diagrama de clases del análisis es un tipo de diagrama estático que describe la estructura de un sistema mostrando sus clases, atributos y relaciones entre ellos. Está compuesto por clases y relaciones. Estas clases del análisis se centran en los requerimientos funcionales, representan conceptos y relaciones del dominio.

Como metodología de desarrollo RUP propone una serie de clasificaciones para estas clases, entre las que resaltan la:

- **Clase interfaz (CI):** modela la interacción entre el sistema y los actores.
- **Clase controladora (CC):** coordina la relación de uno o algunos pocos casos de uso coordinando las actividades de los objetos que implementan la funcionalidad del caso de uso.
- **Clase entidad (CE):** modela información que posee larga vida y es a menudo consistente.

Visual Paradigm representa los tipos de clases mencionados anteriormente de la siguiente forma:

Figura 7. Representación de los tipos de clases del análisis.

A continuación se muestran los diagramas de clases del análisis para cada caso de uso:

Figura 8. Diagrama de clase de análisis del caso de uso Gestionar Código.

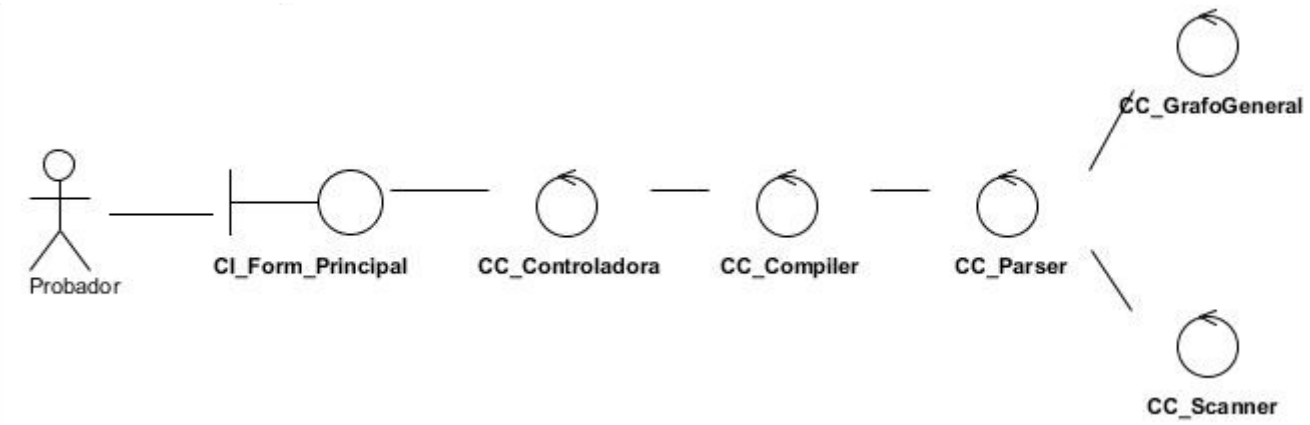


Figura 9. Diagrama de clase de análisis del caso de uso Generar Grafo de Flujo.

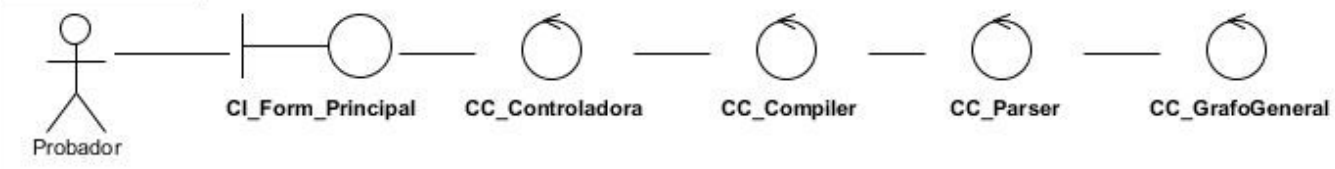


Figura 10: Diagrama de clase de análisis del caso de uso Calcular Complejidad Ciclomática.

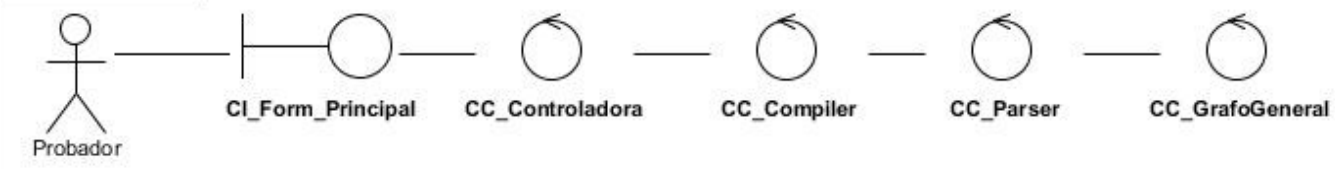


Figura 11. Diagrama de clase de análisis del caso de uso Mostrar Caminos Independientes.

3.5 Arquitectura del Software

Siguiendo la definición que ofrece la *IEEE*, la Arquitectura del Software de un programa es la estructura o las estructuras de un sistema, que incluyen los componentes del software, las propiedades visibles externamente de esos componentes y las relaciones entre ellos.

Durante el desarrollo de un software es frecuente encontrar distintos puntos de vista entre las personas que componen el equipo de trabajo en la concepción y construcción del software. La arquitectura es el instrumento designado para agrupar las diferencias que puedan existir en el equipo de trabajo de una manera sencilla.

A continuación se escogen los patrones utilizados en el diseño de la herramienta.

3.5.1 Patrones

Muchas son las definiciones que brinda la bibliografía de que es un patrón de diseño, sin embargo la mayoría converge en la definición brindada por Christopher Alexander[11], la cual se tomó como punto de partida en este trabajo. El plantea que:

- Cada patrón describe un problema que ocurre una y otra vez en un entorno, para luego describir el núcleo de la solución a ese problema, de tal manera que esa solución puede usarse cuántas veces se quiera en el futuro, sin haberlo hecho de la misma dos veces.

Una vez expuesto el criterio de Christopher Alexander se considera que un patrón de diseño se define de la siguiente manera:

- Un patrón describe un problema que ocurre con frecuencia, y posteriormente describe la solución y aplicación del mismo, el cual puede ser usado en el mismo contexto varias veces sin repetirse la misma solución.

3.5.2 Patrones de Arquitectura

La arquitectura de software básicamente indica la estructura, funcionamiento e interacción entre las partes del software. Para la construcción del prototipo funcional propuesto se decidió utilizar el patrón arquitectura en capas. Este patrón descompone una aplicación en un conjunto de capas independientes y ordenadas jerárquicamente. Cada nivel o capa usa los servicios de la capa inmediatamente inferior y ofrece servicios a la inmediatamente superior. Permite estructurar aplicaciones que se pueden descomponer en grupos de subtareas, donde cada grupo está en un determinado nivel de abstracción.

Para ello se definió la utilización de dos capas, que tienen como objetivo fundamental separar la lógica del negocio con la lógica del diseño. Dentro del modelo en capas se utilizan solo dos, porque en la herramienta no es necesario almacenar ni acceder a datos previamente definidos.

Entre sus principales ventajas se encuentra que admite una amplia reutilización del código generado, permitiendo el fraccionamiento de problemas complejos y facilitando la localización de errores. A continuación se muestran las capas:

- Capa de presentación: esta capa representa la interfaz con que interactuará el usuario.

[11] Destacado arquitecto reconocido internacionalmente por sus numerosos aportes en la Teoría de Patrones.

- Capa lógica del negocio: en esta capa se concentra el código implementado, donde se responde a las solicitudes del usuario, permitiendo automatizar los procesos de negocio que lleva a cabo el usuario.

3.5.3 Patrones de diseño

Un patrón de diseño describe una estructura de diseño que resuelve un problema de diseño en particular, dentro de un contexto específico, y en medio de fuerzas que pueden tener un impacto en la manera que se aplica y utiliza el patrón.

Para asignar la responsabilidad a un objeto determinado se sugiere el uso de los patrones de diseño ***GRASP*** (Patrones para asignar responsabilidades). Entre los diferentes patrones utilizados en la herramienta destacan los siguientes (29):

- Experto: este patrón define quien asume la responsabilidad en el caso general, la clase que cuenta con la información necesaria para cumplir una tarea debe ser la responsable de ejecutar la misma, proporcionando que los objetos aprovechen su propia información para cumplir con sus funcionalidades. En este caso este patrón se identificó en la clase Grafo General, ya que contiene toda la información necesaria del grafo de flujo.
- Creador: este patrón se ocupa de la creación de instancias en las clases, su trabajo consiste en hallar un creador el cual se conectará con el objeto generado. Este patrón se identificó en la clase *Parser.*
- Alta Cohesión: este patrón se encarga de que las clases del diseño cumplan con las tareas que tienen definida, aplicando atributos y métodos de manera sencilla para implementar dichas tareas.
- Bajo Acoplamiento: este patrón se encarga de que las clases del diseño colaboren unas con otras, siempre y cuando esta colaboración se mantenga en un mínimo aceptable, reduciendo el impacto de posibles cambios en la herramienta.
- Controlador: este patrón se identificó en la clase Controladora que define quien se encarga de atender cualquier eventualidad en la herramienta. Esta se diseña de forma tal que no posean demasiada responsabilidad, transmitiendo esta responsabilidad hacia otras clases, mientras la clase Controladora coordina la actividad en cuestión.

Los patrones ***GOF*** se clasifican según su propósito en creacionales, estructurales y de composición, mientras que respecto a su ámbito se clasifican en clases y objetos.

A continuación se muestran los patrones *GOF* utilizados en la herramienta:

- *Singleton* (Instancia única): Garantiza la existencia de una única instancia para una clase, en este caso se puede identificar este patrón en la clase Controladora.
- *Facade* (Fachada): Provee de una interfaz unificada simple para acceder a las diferentes peticiones del usuario mediante una clase que delega la responsabilidad de responder a la petición realizada.
- *Interpreter* (Intérprete): Dado un lenguaje, define las herramientas necesarias para interpretarlo, y representarlo en otro lenguaje, se identificó en la clase *Parser*.

3.6 Diseño

El diseño facilita una mejor comprensión del prototipo funcional que se implementó. En él se recogen los diagramas correspondientes que ayudaron a la programación del prototipo. Se considera la parte fundamental en el proceso de construcción de un sistema que se pretende crear, pues se muestra como una abstracción de cómo quedó el producto final.

3.6.1 Diagrama de clases del diseño

A través de los diagramas de clases del diseño es posible obtener una imagen de cómo quedará la implementación del prototipo funcional. Estos diagramas especifican la relación entre los distintos elementos lo que conlleva a que se haga más fácil la implementación del mismo. A continuación se muestran los diagramas de clases del diseño correspondientes a cada Caso de Uso.

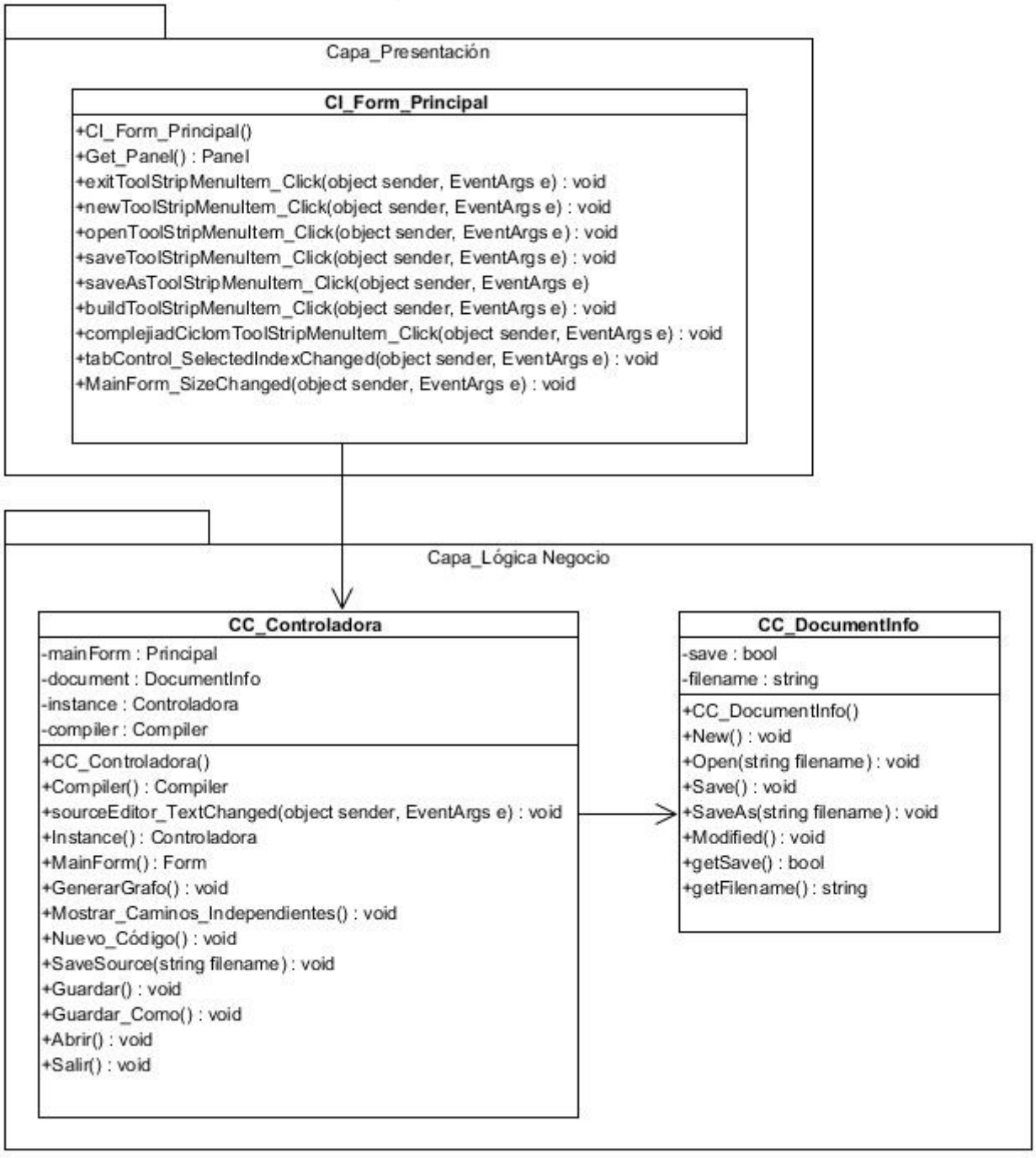


Figura 12. Diagrama de clases del diseño para el caso de uso Gestionar Código.

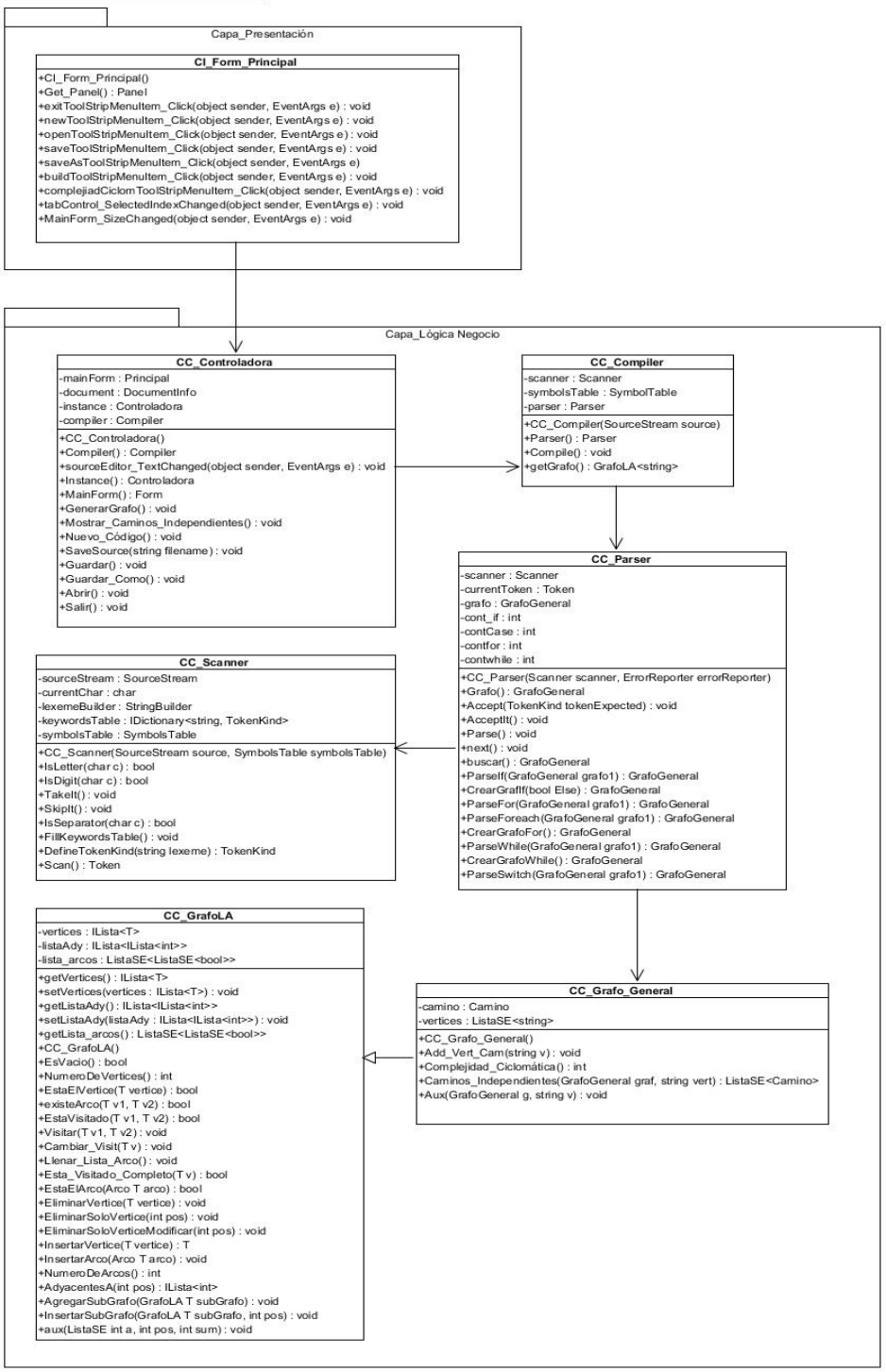


Figura 13. Diagrama de clases del diseño para el caso de uso Generar Grafo de Flujo.

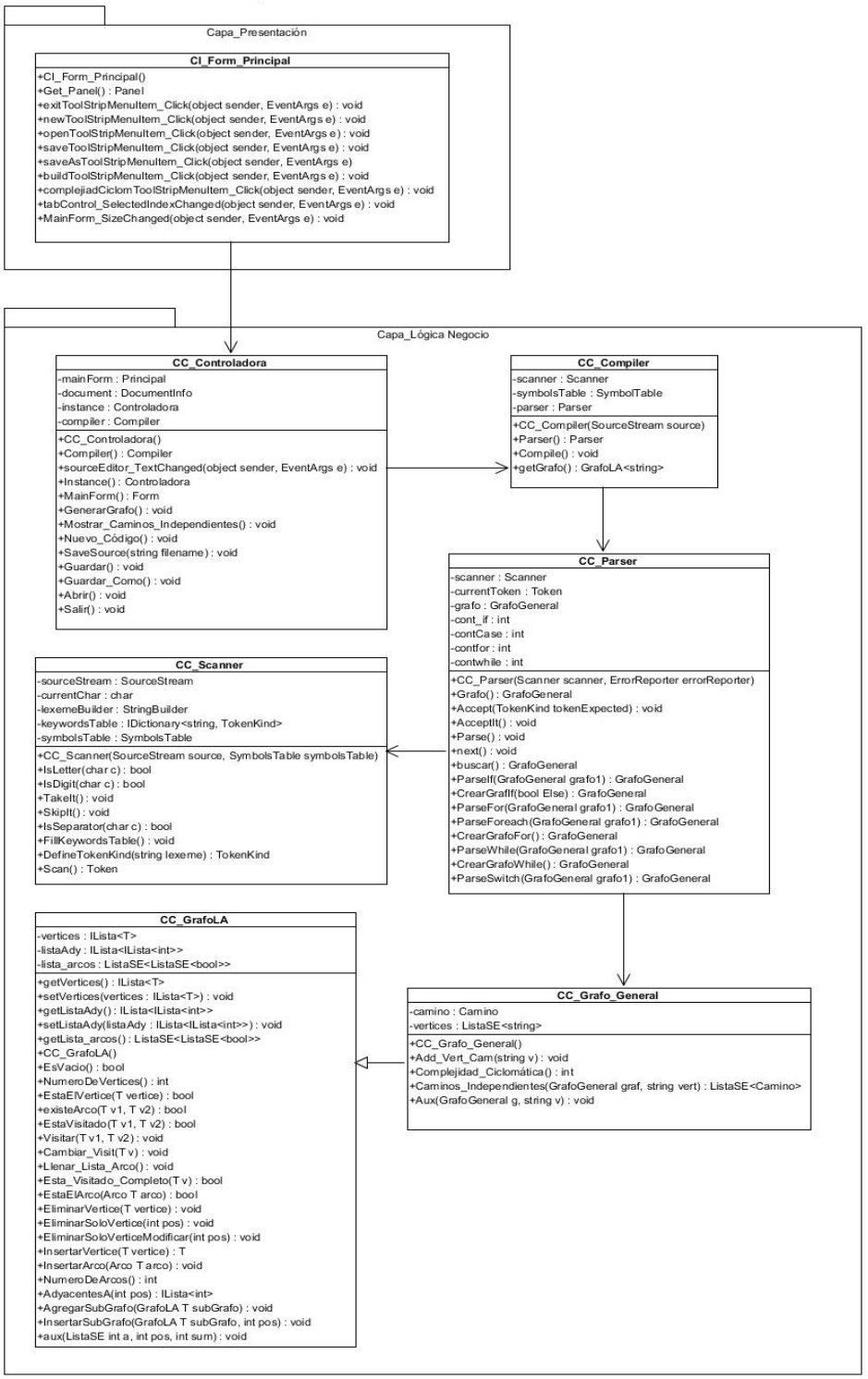


Figura 14. Diagrama de clases del diseño para el caso de uso Calcular Complejidad Ciclomática.

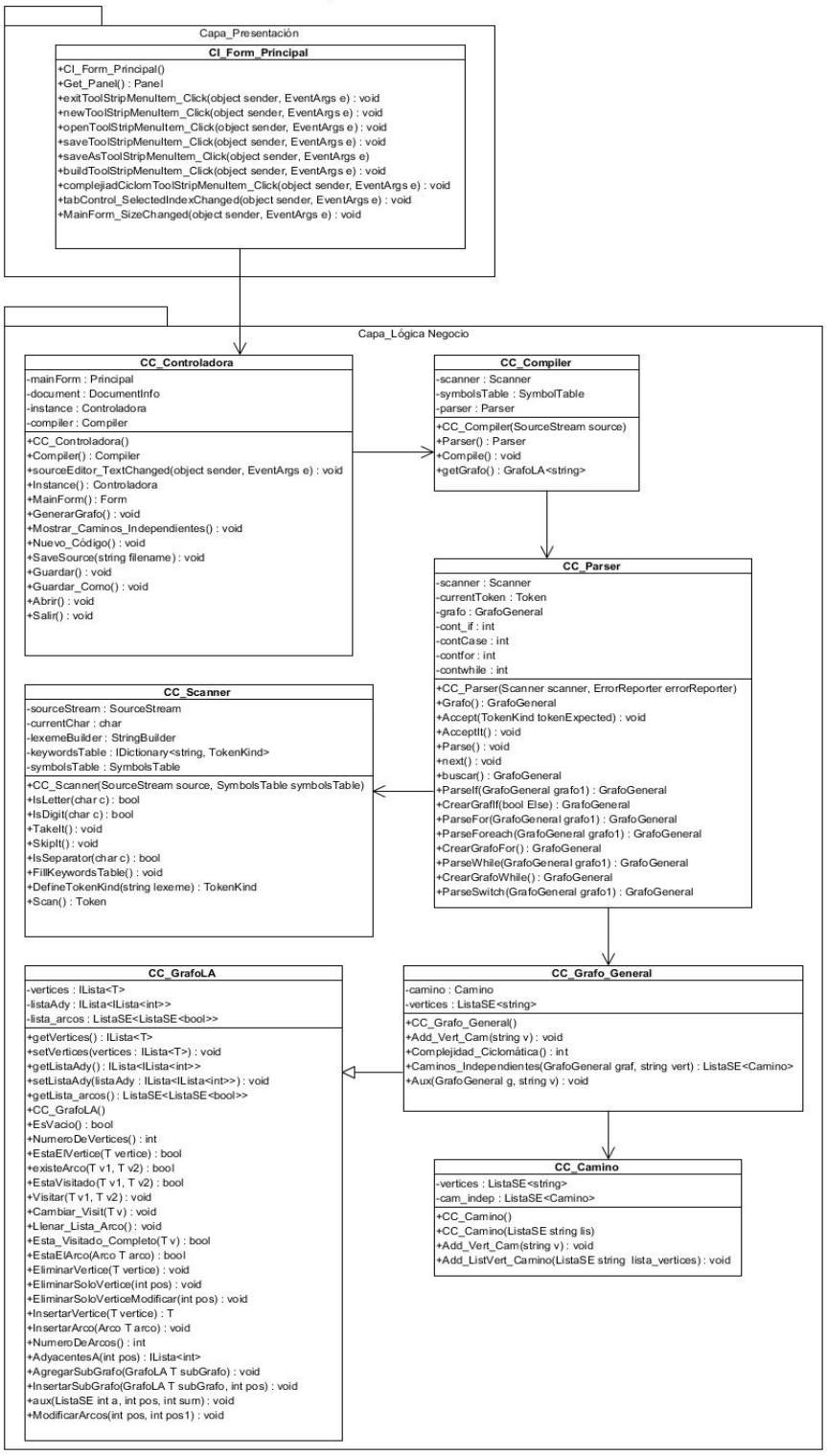


Figura 15. Diagrama de clases del diseño para el caso de uso Mostrar Caminos Independientes.

Una vez mostrados los diagramas de clases del diseño para cada uno de los casos de uso, se muestran a continuación una descripción general de cada una de las clases presentes en el diseño.

Nombre de la clase	**Tipo de clase**	**Descripción**
Form_Principal	Interfaz	Se encarga de conectar la capa de presentación y la capa de lógica del negocio.
Controladora	Controladora	Se encarga de controlar todas las acciones realizadas por el usuario, distribuyendo en dependencia de la funcionalidad, las obligaciones al resto de las clases del negocio.
Compiler	Controladora	Se encarga de capturar el código fuente insertado en la clase interfaz, para luego invocar al método *Parser* de la clase del mismo nombre.
Parser	Controladora	Se encarga de generar el proceso más importante de la herramienta, pues en ella se conforma el grafo de flujo, y se accede a los métodos de la clase Grafo General.
Scanner	Controladora	Se encarga de brindar cada caracter del código para que la clase *Parser* ejecute todas las acciones pertinentes con el caracter.
Grafo General	Controladora	Se encarga de ejecutar los métodos Calcular Complejidad Ciclomática, y Caminos Independientes, que son invocados una vez generado el grafo de flujo.
Grafo LA	Controladora	Esta clase contiene los atributos del objeto grafo, como la lista de adyacencia, número de vértices y arcos, y una serie de métodos importantes que modifican los valores del grafo de flujo.
Camino	Controladora	Esta clase contiene la lista de caminos independientes presentes en el grafo de flujo.

Tabla 4. Descripción de las clases del diseño.

Conclusiones del capítulo

Los conceptos definidos en este capítulo fueron relacionados mediante el modelo de dominio y se mostró de forma general cómo se desarrolla este proceso. Se brindó una clara definición del actor definido para hacer uso de la herramienta, así como los requisitos que debe cumplir la aplicación, para los cuales se definieron los Casos de Uso, Gestionar Código, Generar Grafo de Flujo, Calcular Complejidad Ciclomática y Mostrar Caminos Independientes. Se escogió la arquitectura y patrones utilizados, así como una descripción de las principales clases del diseño, que permitieron una mejor comprensión de la estructura de la herramienta desarrollada.

CAPÍTULO 4: IMPLEMENTACIÓN Y PRUEBA.

Introducción

En este capítulo se realiza el flujo de trabajo de implementación, en los que se analiza los principales artefactos como el Modelo de Implementación que incluye los diagramas de despliegue y componentes. Se describieron los estilos de codificación utilizados, así como una descripción de los principales algoritmos presentes en la implementación de la herramienta. Además se realizaron las pruebas una vez concluido la implementación de la herramienta.

4.1 Modelo de Implementación

El modelo de implementación representa la composición física de la implementación en términos de subsistema de implementación y elementos de implementación. Describe cómo los componentes del diseño se implementan en componentes. Este artefacto se considera el más significativo del flujo de trabajo de implementación, debido a la gran importancia que tiene para los desarrolladores comprender el funcionamiento del sistema desde el punto de vista de componentes y sus relaciones.

4.1.1 Implementación

La implementación es la fase más esperada dentro del proceso de desarrollo de un software. En ella se hacen realidad todas las ideas y artefactos modelados por el equipo de desarrollo durante todas las fases previas. En esta fase existen una serie de artefactos que tienen como elementos de entrada para su confección, los desarrollados en la fase anterior de análisis y diseño. Su objetivo es ir conformando el proyecto como un sistema completo y lograr un acabado correspondiente a los requerimientos previamente definidos.

4.2 Diagrama de despliegue

El diagrama de despliegue es un modelo de objetos que describe la distribución física del prototipo funcional en términos de cómo se distribuye la funcionalidad entre los nodos de cómputo. Se utiliza como entrada fundamental en las actividades de diseño e implementación debido a que la distribución del prototipo funcional tiene una influencia principal en su diseño. A continuación se muestra el diagrama de despliegue de la herramienta.

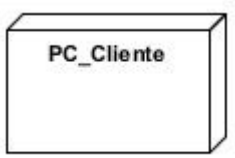


Figura 16. Diagrama de despliegue.

La PC donde se desplegará la herramienta debe poseer las siguientes características:

- Sistema Operativo Ubuntu 10.04
- Framework Mono 3.5
- Procesador de 600 MHz o superior.
- 128 MB de memoria RAM.
- Monitor VGA o superior.
- Espacio en disco duro de 30 MB.

4.3 Diagrama de Componentes

El diagrama de componentes muestra las organizaciones y dependencias lógicas entre componentes de software. Habitualmente contienen componentes, interfaces y relaciones entre ellos como todos los diagramas. También puede contener paquetes utilizados para agrupar elementos del modelo.

Este diagrama describe cómo se organizan los componentes de acuerdo a los mecanismos de estructuración y modularización disponibles en el entorno de implementación y en el lenguaje de programación utilizado, y cómo dependen los componentes unos de otros.

A continuación se muestra el diagrama de componente correspondiente a la herramienta Prueba del Camino Básico.

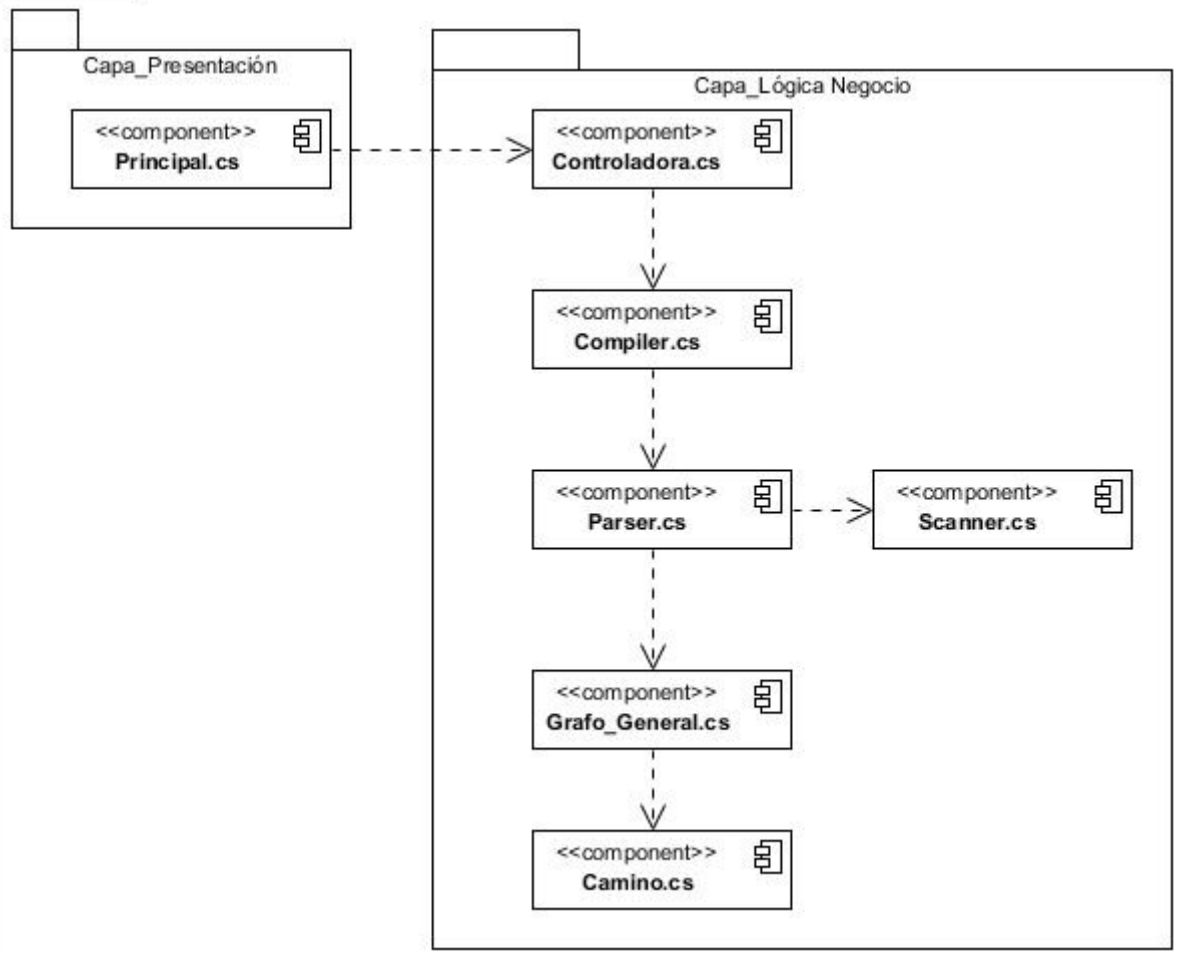


Figura 17. Diagrama de componente.

4.4 Estándares de Codificación

Un estándar de codificación completo comprende todos los aspectos de la generación de código. Usar técnica de codificación sólida y realizar buenas prácticas de programación con vistas a generar un código de alta calidad, es de gran importancia para la calidad del software y para obtener un buen rendimiento.

4.4.1 Estilo de codificación utilizado

Para la implementación de la herramienta propuesta se definieron algunos estilos de codificación para lograr un mejor entendimiento para aquellas personas que deseen realizar posteriores mejoras a la versión realizada.

- Todos los nombres de las variables declaradas se escriben con letra minúscula, si poseen dos o más palabras se separan mediante un guión bajo.
- Los nombres de los métodos comienzan con letra mayúscula y seguidamente el resto en minúscula, si existe dos o más palabras en el nombre del método se separan mediante un guión bajo.
- Los bloques de código siempre deben estar encerrado entre llaves, incluso si solo constan de una línea de código.

A continuación se describen algunos de los principales algoritmos implementados en la herramienta.

4.5 Descripción de los principales algoritmos implementados

Nombre de la clase	Método	Descripción
Parser	Buscar ()	Este método encuentra dentro del código introducido aquellas palabras reservadas, que a partir de ellas se conforma el grafo de flujo. Una vez encontrada una palabra definida pues llama al método que construye el grafo para dicha secuencia. Este método devuelve el grafo de flujo asociado al código analizado.
Grafo General	Complejidad Ciclomática ()	Este método una vez conformado el grafo de flujo, calcula la complejidad ciclomática del mismo, a partir de los nodos y aristas que conforman a este.
Grafo General	Caminos Independientes (grafo, vértice)	Este método recibe como parámetro el grafo de flujo y el primer vértice de este. Luego mediante un algoritmo recursivo determina los caminos independientes del grafo.

Tabla 5. Algoritmos implementados.

4.6 Pruebas

La fase de Prueba es una de las más costosas durante el ciclo de vida del software. En un orden bien pensado y planificado, deben realizarse las pruebas a todos los artefactos generados durante el tiempo que demore la construcción del software. En ello se incluye la especificación de requisitos, los casos de uso, diagramas de diversos tipos, y el código fuente de la aplicación.

Pruebas de Caja Negra

Para la aplicación de pruebas a la herramienta se seleccionó la prueba de caja negra. Estas pruebas también denominadas pruebas de comportamiento, se concentran en los requisitos funcionales del software. Esto permite a quien la aplique derivar conjuntos de condiciones de entrada que ejercitan por completo todos los requisitos funcionales del programa. En esta prueba se pueden aplicar diferentes métodos entre los que se encuentran (2):

- Método de gráficos de pruebas: esta prueba empieza al crear una gráfica de objetos importantes y sus relaciones y luego idea una serie de pruebas que cubran la gráfica de tal manera que se ejercite cada objeto y relación y que se descubran errores.
- Análisis de valores límites (AVL): esta técnica lleva a una selección de casos que prueba los valores al límite. En lugar de concentrarse exclusivamente en las condiciones de entrada, el AVL también deriva casos de prueba del dominio de salida.
- Prueba de tabla ortogonal: este método resulta útil sobre todo en encontrar errores asociados con las *fallas de región (una categoría de error asociada con los defectos de la lógica en un componente de software)*.

En la investigación de este trabajo se utilizó específicamente el método de Partición Equivalente para la aplicación de las pruebas funcionales sobre la interfaz de la herramienta desarrollada. Este método divide el dominio de entrada de un programa en clases de datos a partir de las cuales pueden derivarse casos de prueba. Este método de prueba se esfuerza por definir un caso de prueba que descubra una serie de errores, reduciendo así el número de total de casos de prueba que deben desarrollarse (2).

Para ello se diseñaron casos de prueba a través de la descripción de los casos de uso, que tienen como objetivo demostrar que las funcionalidades son operativas, que los datos de entrada se aceptan de forma adecuada y que se produce una salida correcta, garantizando la integridad de la información que se procesa.

A continuación se muestran los diseños de casos de pruebas a los Casos de Uso Generar Grafo de Flujo, Calcular Complejidad Ciclomática y Mostrar Caminos Independientes.

DCP Generar Grafo de Flujo.

ID del escenario	Escenario	Variable 1 Código	Respuesta del Sistema	Resultado de la Prueba

EC 1.1	Generar Grafo de Flujo con éxito.	V/
		Public bool Prueba(int cond)
		{
		If (cond > 3) { return true}
		Else { return false}
		}
			La herramienta genera el grafo de flujo asociado al código fuente analizado.	Satisfactoria.

EC 1.2	Generar Grafo de Flujo falla.	I/ “”	La herramienta muestra un mensaje indicando que se debe escribir o cargar un código o que los bloques de código	Satisfactoria.

Tabla 6. DCP Generar Grafo de Flujo.

DCP Calcular Complejidad Ciclomática.

ID del escenario	Escenario	Variable 1 Código	Respuesta del Sistema	Resultado de la Prueba

EC 1.1	Calcular Complejidad Ciclomática.	V/
		Public bool Prueba(int cond)
		{
		If (cond > 3) { return true}
		Else { return false}
		}
			La herramienta muestra un mensaje con el valor de la complejidad ciclomática del código.	Satisfactoria.

Tabla 7. DCP Calcular Complejidad Ciclomática.

DCP Mostar Caminos Independientes.

ID del escenario	Escenario	Variable 1 Código	Respuesta del Sistema	Resultado de la Prueba

EC 1.1	Mostrar caminos Independientes	V/
		Public bool Prueba(int cond)
		{
		If (cond > 3) { return true}
		Else { return false}
		}
			La herramienta muestra un listado con los caminos independientes del grafo de flujo.	Satisfactoria.

Tabla 8. DCP Mostrar caminos independientes.

El diseño de caso de prueba del Caso de Uso Gestionar Código se puede observar en el Anexo 2. Durante la fase de prueba se encontraron algunas no conformidades que fueron corregidas en una segunda iteración de pruebas. En este proceso se eliminaron dichas no conformidades por lo que se demostró que la herramienta construida está preparada para ser usada en el departamento Señales Digitales del Centro GEySED, como parte del proceso de pruebas funcionales que se aplican en el departamento.

Conclusiones del capítulo

En este capítulo se analizó la forma en que estará distribuida la aplicación a través de los diagramas de despliegue y componentes. Se pudo comprobar que utilizando un estilo de codificación se obtuvo un código más entendible y organizado. Además se describieron los principales algoritmos utilizados en la implementación de la herramienta. También se realizaron pruebas funcionales a la herramienta para probar su correcto funcionamiento a través de las pruebas de caja negra, las cuales resultaron satisfactorias.

CONCLUSIONES

Una vez terminado el desarrollo de la investigación es posible arribar a las siguientes conclusiones:

- El proceso de desarrollo de software realizado permitió generar todos los artefactos y documentación correspondientes al mismo, lo que sienta las bases para el desarrollo del producto final que desea desplegar la dirección del grupo de Calidad del departamento Señales Digitales.
- Como resultado de la investigación se logró solucionar el problema científico que dio origen a la misma, pues se cumplió con el objetivo principal trazado, al implementarse el prototipo funcional de la herramienta Prueba del Camino Básico, de forma tal que se pueda ejecutar el proceso de pruebas de caja blanca en el departamento Señales Digitales.
- Se demostró la importancia del prototipo funcional Prueba del Camino Básico como una herramienta automática de pruebas de caja blanca, ya que ella ofrece una prueba más confiable a quien la utilice, y la posibilidad de realizarlas con mayor frecuencia y menos esfuerzo para ser ejecutadas que las pruebas manuales.

RECOMENDACIONES

Como resultado del desarrollo de la investigación y conclusiones de esta investigación, se expresan a continuación las siguientes recomendaciones.

- Estimular y promover el continuo desarrollo de la herramienta, agregándole nuevas funcionalidades y refinando las ya existentes para posteriores versiones. Las funcionalidades que se recomiendan agregar son, desarrollar como un complemento de la técnica del camino básico, los métodos de pruebas de condiciones y ciclos, para lograr una prueba más profunda y efectiva. Mejorar en la herramienta la opción mostrar el grafo de flujo asociado al código, para que el grafo se muestre de forma organizada automáticamente.
- Continuar el estudio del tema abordado, ampliando el uso de los elementos teóricos que sustentan la investigación, aplicándolos en el departamento Señales Digitales del centro GEySED.

BIBLIOGRAFÍA Y REFERENCIAS BIBLIOGRÁFICAS

1. *http://riie.com.pe.* [En línea] [Citado el: 20 de enero de 2012.] http://riie.com.pe/?a=40262.

2. **Pressman, Roger S.** *Ingeniería del software, un enfoque práctico.* s.l. : Félix Varela, 2005.

3. mgar.net. *Calidad.* [En línea] Mgar.net. [Citado el: 5 de noviembre de 2011.] http://mgar.net/soc/isointro.htm.

4. **Neulan Aguero, Dennis.** http://calisoft.uci.cu. [En línea] 2009. [Citado el: 5 de noviembre de 2011.] http://calisoft.uci.cu/tmp/documentos/articulos/articulo_sqa.pdf.

5. **Letelier, Patricio.** Pruebas del Software. 2009.

6. *Material de caja blanca y caja negra.* [Documento PDF]

7. [En línea] 3 de diciembre de 2011. http://geeks.ms/blogs/mllopis/archive/2008/07/07/pex-herramienta-automatica-para-la-realizacion-de-pruebas-unitarias-en-net.aspx.

8. **Carillo Pérez, Isaías, Rodrigo, Pérez y Martín, Rodríguez.** *Metodología de Desarrollo del Software.* 2008.

9. **Rumbaugh, James.** *Ayuda Extendida del Rational Unificado Process.* 2003.

10. **Beck, Ken.** *Extreme Programming Explained.* s.l. : Pearson Education, 1999.

11. **Beck.** *Planeando en Programación Extrema.* 2000.

12. **Joskowicz, José.** [En línea] 2002. http://iie.fing.edu.uy/~josej/docs/XP%20-%20Jose%20Joskowicz.pdf.

13. **Orallo, Enrique.** *El lenguaje Unificado de Modelado (UML).*

14. **Salazar Ramírez, José Ramón.** [En línea] noviembre de 2009. [Citado el: 3 de diciembre de 2011.]

15. **Booch, Jacobson.** *El Proceso unificado de desarrollo de Software.* Habana : Félix Varela, 2004.

16. **Geetanjali, Arora, Balasubramaniam, Aiaswamy y Nitin, Pandey.** *Programación en C#.* s.l. : Félix Varela, 2002.

17. [En línea] [Citado el: 1 de diciembre de 2011.] http://msdn.microsoft.com/es-es/library.

18. **Mueller, John.** *Visual C++.Net.* 2002.

19. **Mitchell, Will David.** *Java sin errores.* Madrid : s.n., 2001.

20. **Zukowsky, John.** *Java.* Madrid : s.n., 2003.

21. *Microsoft.* [En línea] [Citado el: 2 de diciembre de 2011.] http://msdn.microsoft.com/en-gb/vstudio.

22. [En línea] http://mredison.wordpress.com/2007/12/02/caractersticas-de-visual-studio-2008/.

23. *icsharpcode.* [En línea] [Citado el: 2 de marzo de 2012.] http://www.icsharpcode.net/.

24. *scribd.* [En línea] [Citado el: 2 de marzo de 2012.] http://es.scribd.com/doc/25957895/Net-framework-General-Overview.

25. [En línea] [Citado el: 3 de marzo de 2012.] http://monodevelop.com/.

26. *dcases.* [En línea] [Citado el: 21 de enero de 2012.] http://www.dcases.com/61/programacion-en-c-compiladores-y-entornos-de-desarrollo/.

27. *ecured.* [En línea] [Citado el: 5 de marzo de 2012.] http://www.ecured.cu/index.php/Framework.

28. Ingeniería de Software 2. Conferencia #1. Continuación del FT Análisis y Diseño. Modelo de Diseño. 2011.

29. **Larman, Craig.** *UML y Patrones.*

ANEXOS

Anexo 1 Entrevista.

Entrevistado: Delvis Hecheverría Pérez, Tayché Capote García.

Entrevistador: Javier Lambert Claramunt.

Preguntas

1. ¿Se realizan pruebas de caja blanca a los productos que llegan al centro de Calisoft?
2. ¿Conocen de alguna herramienta que realice la técnica del camino básico?

Respuestas

1. En Calisoft por la etapa en la que entramos a realizar pruebas a los proyectos, que es cuando ya el producto está listo para ser entregado al cliente, no hacemos pruebas de caja blanca que son las que se realizan al código. Estas pruebas es posible que la realicen algunos proyectos de la UCI, pero no lo hacemos en el Departamento de Pruebas Software de Calisoft debido a que los artefactos ya llegan cuando están finalizados y listos para ser entregados al cliente. Realizar este tipo de pruebas en esta etapa con el rigor que lleva podría complejizar mucho la liberación y realmente sería muy trabajoso por la cantidad de líneas de código.
2. Conocemos algunas herramientas y los métodos generales, a partir de los mismos elementos que se imparten en Ingeniería de Software. Estas herramientas que algunas vienen integradas en algunos componentes de los mismos entornos de desarrollo, se basan fundamentalmente en el cálculo de la complejidad ciclomática, y otras métricas respecto al código cómo cantidad de métodos y atributos por clases, densidad de código, entre otras. Sin embargo no conocemos de alguna herramienta que realice la técnica del camino básico

Anexo 2 Diseño de casos de prueba.

DCP Cargar Código

ID del escenario	Escenario	Respuesta del Sistema	Resultado de la Prueba
EC 1.1	Cargar código con éxito.	La herramienta carga el archivo que contiene el código fuente y lo muestra en el campo código.	Satisfactoria.
EC 1.2	Cargar código falla.	La herramienta no puede cargar el archivo que contiene el código fuente.	Satisfactoria.

Tabla 9 SC 1. Cargar código.

DCP Guardar Código

ID del escenario	Escenario	Respuesta del Sistema	Resultado de la Prueba
EC 1.1	Guardar código con éxito.	La herramienta guarda el archivo que contiene el código fuente.	Satisfactoria.
EC 1.2	Guardar código con falla.	La herramienta no guarda el archivo que contiene el código fuente.	

Tabla 10 SC 2. Guardar código.

DCP Nuevo Código

ID del escenario	Escenario	Respuesta del Sistema	Resultado de la Prueba
EC 1.1	Nuevo código con éxito.	Se muestra el código escrito en el campo código.	Satisfactoria.

Tabla 11 SC 3. Nuevo código

Anexo 3 Descripción de los Casos de Uso.

CU Calcular Complejidad Ciclomática

Caso de Uso:	Calcular Complejidad Ciclomática.
Actores:	Probador.
Resumen:	El caso de uso se inicia cuando el probador desea calcular el valor de la complejidad ciclomática del código escrito. El caso de uso finaliza una vez que se muestra en un cuadro de diálogo el valor de la complejidad ciclomática.
Precondiciones:	Se debe haber realizado el CU Gestionar Código y el CU Generar Grafo de Flujo.
Referencias	RF-3
Prioridad	Crítico.
Flujo Normal de Eventos	
Acción del Actor	**Respuesta del Sistema**
1. El caso de uso inicia cuando el probador selecciona en el menú la opción “Prueba del Camino Básico”	1.1 La herramienta despliega un menú con las opciones “Generar Grafo de Flujo”, y “Calcular Complejidad Ciclomática”.
2. El probador selecciona la opción: Generar Grafo de Flujo.	2.1 La herramienta muestra en un cuadro de diálogo el valor de la

	complejidad ciclomática calculada para el código analizado, y se finaliza el caso de uso.
Poscondiciones	Se muestra un cuadro de diálogo con el valor de la complejidad ciclomática para el código analizado.

Tabla 12. Descripción del Caso de Uso Calcular Complejidad Ciclomática.

CU Mostrar Caminos Independientes

Caso de Uso:	Mostrar caminos Independientes.
Actores:	Probador.
Resumen:	El caso de uso se inicia cuando el probador, una vez generado el grafo de flujo desea visualizar los caminos independientes asociados a dicho grafo. El caso de uso termina cuando se muestran en la interfaz los caminos independientes.
Precondiciones:	Se debe haber realizado el CU Gestionar Código.
Referencias	RF-4.
Prioridad	Crítico.
Flujo Normal de Eventos	
Acción del Actor	**Respuesta del Sistema**
1. El caso de uso inicia cuando el usuario selecciona la opción “Mostrar caminos independientes”	1.1 Se muestra una ventana con el listado de los caminos independientes que permite visualizar los detalles del mismo y finaliza el caso de uso.
Poscondiciones	Se visualiza un listado de todos los caminos independientes calculados.

Tabla 13. Descripción del Caso de Uso Mostrar Caminos Independientes.

GLOSARIO

- **Calidad de software**: es un conjunto de características presentes en un software, que satisfacen las necesidades del cliente.
- **Caja blanca**: es un tipo de prueba que se basa en el diseño de casos de prueba a partir del código fuente de las aplicaciones.
- **Caja negra**: es un tipo de prueba que permite obtener un conjunto de condiciones de entrada que ejerciten completamente todos los requisitos funcionales del programa.
- **Camino básico**: es una técnica de prueba de caja blanca que permite obtener una medida de la complejidad lógica de un diseño.
- **Complejidad ciclomática**: es una métrica de software que proporciona una medición cuantitativa de la complejidad lógica de un programa.
- ***Framework***: estructura de soporte definida, que está constituido por programas, bibliotecas y un lenguaje interpretado que facilitarán el desarrollo de un proyecto informático.
- **IDE**: Entorno de Desarrollo Integrado. Es un programa compuesto por un conjunto de herramientas para facilitar la interacción del programador con el lenguaje en cuestión.
- **Pruebas de software**: se encargan de verificar el comportamiento de un programa en un conjunto finito de casos de pruebas.
- ***Stakeholder***: son considerados todas aquellas personas que están involucradas en la realización de un software determinado, que pueden afectar o ser afectados tanto de forma negativa o positiva durante el proceso de desarrollo del software.
- **TIC**: Tecnologías de la Información y las Comunicaciones. Son todos los servicios, software y hardware que interconectados contribuirán a mejorar las condiciones de vida de las personas.