**Universidad de las Ciencias Informáticas**

**Facultad 3**

*Trabajo de diploma para optar por el título de Ingeniero en Ciencias Informáticas*

# Título: Herramienta para la generación automática de diseños de casos de prueba

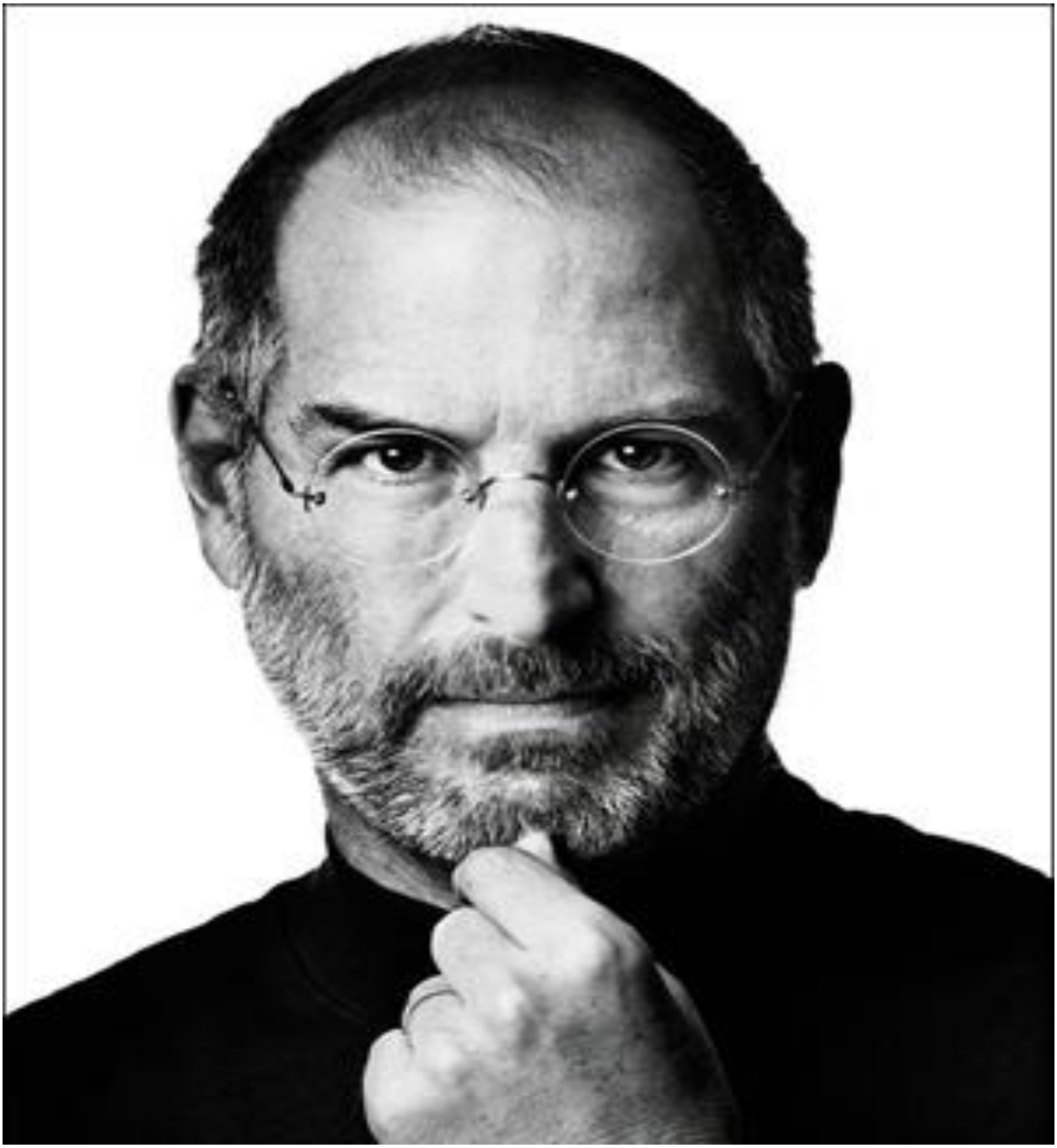
**Autor(es):**

**Yanetzi Jimeno Vázquez**

**Raúl M. Romero Rodríguez**

**Tutor: Raúl Velázquez Alvarez**

La Habana, 20 de junio de 2013



*"Ser el más rico del cementerio no es lo que más me importa... Acostarme por la noche y pensar que he hecho algo genial. Eso es lo que más me importa".*

*Steve Jobs.*

**DECLARACIÓN DE AUTORÍA**

Declaramos que somos los únicos autores de este trabajo y autorizamos a la Facultad 3 de la Universidad de las Ciencias Informáticas a hacer uso del mismo en su beneficio.

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

_____

Yanetzi Jimeno Vázquez

Autor

_____

Raúl M. Romero Rodríguez

Autor

_____

Raúl Velázquez Alvarez

Tutor

**AGRADECIMIENTOS**

A mi mamá que no ha descansado ni un momento para yo llegar a donde estoy, por estar siempre junto a mí y apoyarme en todas mis decisiones y quererme como soy, te quiero. A mi hermano Yoe ejemplo de perseverancia y de no rendirse nunca. A mi novio Leonel (mi titi) por aguantarme estos casi 2 años te amo mucho. A mi familia mi tía Lili, mi tío Raúl, mi tío Guillermo, a Viviana que aunque no es ya mi tía la sigo queriendo como tal, a mi primo Norbe. A la gente del barrio que han ayudado mucho a mi mamá: Dainerys, Sol, Maira, Yadelkis, Ailiaris, Daniela y todos los demás que sino no me alcanzan los agradecimientos. A los viejos y los nuevos amigos Yudith, Ivis, Reinier, Ismael, Neysi, a la gente de mi grupo que aunque no estuve con ellos desde el principio los quiero igual. A mi compañera de cuarto Leydis que estamos juntas desde primer año. A mi compañero de tesis Raúl que mejor no lo puedo pedir, a nuestro Tutor Raúl. Y por último y no menos importante a mi papá que ya no está entre nosotros, pero sé que ha estado conmigo desde el primer día en que se fue.

*Yanetzi*

A mi mamá y mi tía por ser las personas que siempre han estado presentes sin importar cuál sea la situación y pidiendo solo a cambio el título que hoy les doy. A mi tío Raulito por brindarme su ayuda siempre que la necesité. Al Ruso por ser el mejor amigo del mundo y siempre confiar en que se podía aun cuando ni yo mismo estaba seguro. A todas las personas que lucharon a mi lado para que este día fuera posible: Ricardo, Amilcar, Luis Angel, Buti, Bati, Duvergel, Adrian, Diogenes, Argelio, Julio, Nivaldo, César, y todos los que ya se han ido, que también los tengo siempre presentes. A todos mis amigos del Dota los cuales han hecho de mi tiempo libre un momento para recordar y me han enseñado mucho. A mi titi por ayudarme tanto y ser tan buena conmigo, lástima que no nos conocimos antes. A mi tutor Raúl segundo por nunca gustarle nada y así ayudar que el trabajo saliera lo mejor posible. A mi compañera de tesis por obligarme a ser cada día mejor.

*Raúl*

**DEDICATORIA**

Dedico este triunfo a mi mama, a mi papa, a mi hermano Yoe, a mi familia y a mi novio Leonel, a todos ellos por estar siempre conmigo.

*Yanetzi*

A mi mamá y a mi tía ya me siento bien conmigo mismo debido a que cumplí con ustedes. A mi abuela Ana Rosa que aunque ya no esté aquí sé que me vigila. A mi familia por siempre indicarme el camino correcto. Al ruso por siempre creer en mí. A mi tía Yanelis que ha demostrado ser una persona increíble.

*Raúl*

**RESUMEN**

El presente trabajo, titulado “Herramienta para la generación automática de Diseño de Casos de Prueba”, se realizó con el propósito de mejorar la elaboración de este artefacto de trabajo contribuyendo así al Aseguramiento de la Calidad de los proyectos de la facultad 3.

Para ello fue necesario realizar una fundamentación teórica en la que se abordan los diferentes conceptos emitidos por varios autores referentes a aspectos relacionados con Calidad de Software, Pruebas de Software, entre otros, que sustentan el hilo conductor del marco teórico referencial, se seleccionaron por otra parte el desarrollo de la herramienta estuvo guiado por las especificaciones que propone la metodología XP, obteniendo los artefactos de las diferentes fases de trabajo, como la exploración y planificación, diseño, implementación y pruebas.

Con el desarrollo de esta herramienta web y su incorporación en los proyectos se pretende minimizar la carga de trabajo que realizan los analistas de los equipos de desarrollo de software. Además se contará con una base de datos centralizada en todos los proyectos, donde se guardará toda la información referida a los casos de prueba así como los documentos de especificación de casos de uso y el documento de descripción de requisitos.

**Palabras Claves:**

Aseguramiento de la Calidad, Diseños de Casos de Prueba, Calidad de Software, Pruebas de Software

**ÍNDICE DE CONTENIDOS**

# INTRODUCCIÓN

El vertiginoso desarrollo actual de las tecnologías de la información y las comunicaciones, ha aumentado la necesidad de las compañías de destinar una mayor cantidad de sus presupuestos a las nuevas tecnologías aplicadas al desarrollo de su actividad. Todas estas innovaciones tecnológicas ameritan una exigente evaluación de la calidad de los productos.

La calidad es un factor muy importante a tener en cuenta durante el desarrollo de un producto software, dicho aspecto incentiva la preocupación de muchos productores de software para obtener un producto fiable, eficaz, seguro y funcionalmente operativo que cumpla con los requisitos del cliente.

No siempre se logra que el resultado final del producto cumpla los requisitos establecidos. Por tal motivo se le realizan pruebas al software durante el ciclo de desarrollo del mismo, lo que constituye un proceso aplicado al producto para corregir los errores que presenta.

Las pruebas de software son una técnica clásica para la verificación y validación del software; junto a otras alternativas de evaluación como revisiones, auditorías y métricas. Las pruebas de software no son más que la ejecución de un programa con el fin de detectar errores. Para ello, el diseño de las pruebas se basa en la creación de casos de prueba cuya ejecución permitirá observar posibles síntomas de defectos.

Como parte de los esfuerzos de Cuba por ascender a planos superiores en la rama de la informática, han surgido diferentes entidades dedicadas al desarrollo de software. Las mismas deben garantizar la entrega de sus productos con la mayor calidad posible y para ello es sumamente importante no olvidar detalles ni etapas en el desarrollo de una aplicación. Una de estas etapas es la de pruebas, fundamental para aumentar la calidad de los productos y con ellos las posibilidades de insertarse entre las naciones reconocidas como exportadoras de software.

En Cuba se encuentra la Universidad de las Ciencias Informáticas (UCI), institución que se dedica a la formación de profesionales Ingenieros en Ciencias Informáticas y a la producción de software.

La UCI poco a poco se ha ido insertando en la producción de software para esto cuenta con centros productivos destinados al desarrollo de productos, tanto para el mercado nacional como el internacional. Hoy en día en el Centro de Gobierno Electrónico (CEGEL) de la Facultad 3, se

encuentra el Grupo de Calidad destinado a la revisión y evaluación de los productos desarrollados en el mismo.

Actualmente los casos de prueba que se diseñan en los proyectos del Centro se realizan de forma manual, lo que hace que se dedique un tiempo considerable para realizar esta actividad. La mayoría de las veces, cuando estos artefactos son entregados para su liberación, durante el proceso de revisión se detectan inconsistencias entre la especificación del requisito o del caso de uso y el caso de prueba tales como:

- Ausencia de variables.
- La definición de las reglas o restricciones que son necesarias para realizar la prueba.
- Ausencia de secciones o escenarios de prueba que afectan la matriz de datos.

Normalmente el diseño de casos de prueba debe ser una actividad compartida entre el analista y el programador que son los principales actores que conocen el flujo de información dentro del sistema y cómo debería estar validado; pero por la carga de trabajo se suele asignar esta tarea al probador que en la mayoría de los casos no interviene directamente sobre los procesos de negocio que gestiona el sistema desconociendo muchos elementos importantes que deben probarse. De esta manera se insertan en el caso de prueba varias irregularidades convirtiéndose en no conformidades que afectan el desarrollo de las pruebas.

Teniendo en cuenta lo anteriormente expuesto se define como **problema a resolver** de la presente investigación ¿Cómo contribuir a la mejora del diseño de los casos de prueba en los proyectos del centro CEGEL de manera que se acelere la elaboración de este artefacto y que disminuyan las no conformidades detectadas en los mismos?

El **objeto de estudio** de la investigación se centra en el proceso de Pruebas funcionales y se concreta como **campo de acción**: Diseños de Casos de Prueba.

Se propone como **objetivo general** desarrollar una herramienta para la generación automática de Diseños de Casos de Prueba de manera que se acelere la elaboración de este artefacto y que disminuyan las no conformidades detectadas en los mismos contribuyendo así a la mejora del diseño de este producto de trabajo.

Para dar cumplimiento al objetivo general del trabajo de diploma en cuestión se identificaron los siguientes **objetivos específicos**:

- Elaborar el marco teórico de la investigación.

- Elaborar la propuesta de solución.
- Validar la solución propuesta.

Para dar cumplimiento a los objetivos específicos se trazaron las siguientes **tareas de la investigación**:

- Realización de un estudio del estado del arte de la temática en la UCI, Cuba y el mundo.
- Análisis de las metodologías de desarrollo, herramientas y tecnologías posibles a utilizar en el desarrollo de la solución.
- Levantamiento de requisitos.
- Planificación.
- Diseño y codificación del sistema.
- Realización de las pruebas de validación de la aplicación.

Para dar respuesta a las tareas planteadas con anterioridad se proponen Métodos Científicos que se clasifican en:

**Teóricos**:

- **Analítico–sintético:** la utilización de este método ha servido para analizar y comprender la teoría y documentación relacionada con el tema de investigación, permitiendo así, extraer los elementos más coherentes e importantes relaciones con el objeto de estudio.
- **Análisis histórico–lógico:** fue utilizado para realizar un estudio exhaustivo sobre el Diseño de Casos de Prueba, se escogió este método ya que posibilita un buen análisis y una mejor comprensión.
- **Modelación**: para el desarrollo del modelo de la base de datos del sistema por lo que se usará el método de modelación, mediante el cual se pueden crear abstracciones con el propósito de explicar la realidad.

**Empíricos**:

- **Entrevistas:** con el objetivo de adquirir nuevos conocimientos acerca de las herramienta de automatización de pruebas así como para conocer sobre el Diseño de Casos de Prueba.
- **Medición**: Este método se tiene en cuenta con las pruebas que se le realizan al software y las métricas.

Como **posible resultado** se espera obtener una herramienta para la generación automática de diseños de casos de prueba.

El presente Trabajo de Diploma posee la siguiente estructura:

**Capítulo 1: Fundamentación Teórica:** en este capítulo se abordan los aspectos teóricos y técnicos de la investigación. Propone un análisis de las metodologías, tecnologías y herramientas a utilizar durante el desarrollo de la solución.

**Capítulo 2: Planificación, Diseño y Desarrollo:** en este capítulo se determinan los requisitos funcionales y no funcionales que la herramienta a implementar debe cumplir, además se presenta la descripción del sistema. También se realiza la descripción del comportamiento del sistema a través de las historias de usuario, se establece un tiempo estimado en semanas para implementar cada historia de usuario, se elabora el plan de entregas así como las tarjetas CRC y las tareas de ingeniería para la implementación del sistema.

**Capítulo 3: Validación de la Propuesta de Solución:** en este capítulo se aplican diferentes métricas para validar los requisitos, el diseño y las funcionalidades del sistema y se realiza la verificación del cumplimiento de las variables planteadas en el problema.

# CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA

## 1.1. Introducción

En el presente capítulo se realiza el estudio de los contenidos teóricos que sustentan la investigación, se realiza el estudio de la problemática y se exploran soluciones previas en otras partes del mundo. Se realiza un análisis de las herramientas y tecnologías más adecuadas y brindar una panorámica de la solución. Estas herramientas y tecnologías han sido seleccionadas siguiendo la línea del Centro hacia la cual va dirigida la presente investigación, con el objetivo de reutilizar elementos y componentes que ya se encuentran en explotación en otros proyectos productivos.

## 1.2. Calidad de Software

El profesor Roger S. Pressman plantea que la calidad de software se refiere a la: "*Concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos con los estándares de desarrollo explícitamente documentados y con las características implícitas que se espera de todo software desarrollado profesionalmente*" (Pressman, 1998).

La ISO 8402 expresa que la calidad de software es "*El conjunto de características de una entidad que le confieren su aptitud para satisfacer las necesidades expresadas y las implícitas*" (ISO, 1994).

El término calidad ha ido evolucionando de forma continua a lo largo de su desarrollo. Muchos autores han dado su definición, la cual de forma general plantea que: "*es la totalidad de aspectos y características de un producto o servicio que se refieren a su capacidad para satisfacer necesidades dadas en la adecuación de sus objetivos*" (Control de Calidad en los Sistemas, junio 2000).

Enfocándose en el cliente, calidad del software podría ser el grado en que un cliente y/o usuario percibe que el producto de software satisface sus necesidades y expectativas; pero en la condición industrial del producto, calidad del software es la habilidad de un producto de software de satisfacer su especificación de requisitos (Lopéz, 2002).

Otra de las definiciones encontradas plantea que la calidad del software es el "*nivel de fiabilidad, robustez y eficiencia del software referido a todo su comportamiento en todo el período de vigencia*" (Sedna, 2006).

Es importante destacar que en estos conceptos los autores no tuvieron en cuenta el factor tiempo a la hora de desarrollar un producto de software, ya que siempre hay que tener presente los compromisos de entrega. Por lo tanto el primer compromiso que se debe trazar un equipo de desarrollo de software es producir un software de máxima calidad y en el tiempo establecido.

Todo proyecto tiene como objetivo producir software con la mayor calidad, que cumpla, y si es posible, supere las expectativas de los usuarios. La calidad del software ha pasado de una simple inspección y detección de errores a un cuidado total en su proceso de desarrollo y mantenimiento. El correcto funcionamiento de este, es fundamental para el óptimo comportamiento de los sistemas informáticos de las empresas. Un software de calidad es aquel que cumpla con los requisitos funcionales y de rendimiento, además de ser confiable y aceptable.

Se describen a continuación algunas de las características que crean a un software de calidad.

- Mantenibilidad: el software debe ser diseñado de tal manera, que permita ajustarlo a los cambios en los requisitos del cliente. Esta característica es crucial, debido al inevitable cambio del contexto en el que se desempeña un software.
- Confiabilidad: incluye varias características como la seguridad, control de fallos, entre otras.
- Eficiencia: se refiere al uso eficiente de los recursos que necesita un sistema para su funcionamiento.
- Usabilidad: el software debe ser utilizado sin un gran esfuerzo por los usuarios para los que fue diseñado y documentado (Coral, y otros, 2010).

Reuniendo de forma conjunta las características anteriormente expuestas y para formar un criterio que sirva como fundamento para el trabajo a realizar, los autores plantean, que la calidad de software es el conjunto de requisitos que necesariamente deben estar presentes en un software de forma documentada a través de estándares de desarrollo, donde mediante su seguimiento y la realización de revisiones periódicas se asegure que cuente con mantenibilidad, confiabilidad, eficiencia y usabilidad logrando de este modo fomentar la satisfacción del cliente. Una de las actividades para garantizar la Calidad de Software es el Aseguramiento de la Calidad.

**Aseguramiento de la Calidad**

El aseguramiento de la calidad es un conjunto de actividades planificadas, sistemáticas y necesarias para aportar la confianza de que el producto de software satisface los requisitos de calidad (Neuland Agüero, 2008). Según la norma ISO 9000-3 el propósito del aseguramiento de la calidad es proporcionar una adecuada seguridad de que los productos de software y los procesos en el ciclo de vida del proyecto están conformes con sus requisitos específicos y se ajustan a sus planes establecidos.

El aseguramiento de la calidad aborda principalmente (Coral, y otros, 2010):

- Un enfoque de gestión de la calidad.
- Métricas del software.
- Verificación y validación a lo largo del ciclo de vida del software, incluyendo pruebas, procesos de revisión y auditorías.
- Gestión de configuración del software.
- El control de la documentación del software.
- Un procedimiento que asegure los ajustes a los estándares en el proceso de desarrollo de software siempre que sea posible.

El aseguramiento de la calidad, como actividad de soporte en el proceso de desarrollo, comprende procedimientos para la aplicación efectiva de métodos y herramientas, revisiones y estrategias de prueba, control de cambios, aseguramiento de ajuste a los estándares entre otras. El Instituto de Ingeniería del Software (SEI) recomienda la aplicación de las siguientes actividades:

- Establecimiento del Plan de Aseguramiento de la Calidad para un proyecto.
- Participación en el desarrollo de la descripción del proceso de software.
- Revisión de las actividades de ingeniería del software.
- Auditorías de los procesos de software designadas para verificar el ajuste con los definidos como parte del proceso de software.
- Registrar lo que no se ajuste a los requisitos e informar a los superiores.
- Coordinar el control de cambio.

Básicamente, el aseguramiento de la calidad del software consiste en la revisión de los productos y su documentación relacionada para verificar su cobertura, corrección, confiabilidad y facilidad de mantenimiento, incluyendo la garantía de que un sistema cumpla las especificaciones y los

requisitos para su uso y desempeño deseado. Una de las actividades que forma parte del aseguramiento de la calidad son las pruebas de software.

**Pruebas de Software**

Cuando se habla de Aseguramiento de la Calidad es importante mencionar las Pruebas de Software, estas juegan un papel fundamental durante el ciclo de desarrollo del mismo ya que son el elemento crítico para la garantía de la calidad. Permiten la verificación y validación del software que fue construido. Asegura que el software implementa correctamente una funcionalidad en específico y se comprueba que el producto compensa las exigencias del cliente.

La IEEE plantea que una prueba es: *"Una actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas, los resultados se observan, se registran y se realiza una evaluación de algún aspecto".*

Según Pressman *"La prueba de software es un elemento crítico para la garantía de la calidad del software y representa una revisión de las especificaciones, el diseño y la codificación".*

Para lograr que este proceso sea satisfactorio se tienen que trazar objetivos fundamentales de la eficacia requerida. Estos objetivos son:

- Contribuir a la calidad del producto desarrollado.
- Descubrir errores de un programa no detectados hasta entonces.
- Manejar los resultados sistemáticamente, de forma que los defectos importantes puedan ser arreglados.

Se puede decir que la prueba no es una actividad sencilla, no es una fase del proyecto en que se asegura la calidad, sino que debe ocurrir durante todo el ciclo de vida, se puede probar la funcionalidad de los primeros prototipos; probar la estabilidad, cobertura y rendimiento de la arquitectura; probar el producto final. Es un proceso que se enfoca sobre la lógica interna del software y las funciones externas, no puede asegurar la ausencia de defectos, sólo puede demostrar que existen defectos en el software.

Todas las pruebas no son aplicables a todas las ramas de construcción del software. Existen diferentes tipos de pruebas identificadas por el alcance que tienen y por los resultados que deben arrojar al ponerse en aplicación, uno de estaos tipos de pruebas son las Pruebas de Funcionalidad.

**Pruebas de Funcionalidad**

Una vez que se ha desarrollado una aplicación, los desarrolladores necesitan verificar que está libre de anomalías y que se ha logrado el objetivo de su diseño. Las pruebas de funcionalidad determinan la extensión en la que la aplicación satisface los requisitos funcionales esperados. Este proceso simula varios escenarios para confirmar que todos los resultados satisfacen las expectativas establecidas.

Además de estos objetivos, las pruebas de funcionalidad se trazan metas específicas:

- Verificar el procesamiento, recuperación e implementación adecuada de las reglas del negocio.
- Verificar la apropiada aceptación de datos.

Para la implementación y desarrollo de estas pruebas se siguen los métodos de caja negra donde se ejecuta cada caso de uso, flujo de caso de uso, o función, usando datos válidos e inválidos, para verificar lo siguiente:

- Que se aplique apropiadamente cada regla de negocio.
- Que los resultados esperados ocurran cuando se usen datos válidos.
- Que sean desplegados los mensajes apropiados de error y precaución cuando se usan datos inválidos (Myers, 2004).

Además de los tipos de pruebas existen métodos que tienen como objetivo diseñar pruebas que descubran diferentes tipos de errores con menor tiempo y esfuerzo. Dado que la presente investigación está encaminada al tipo de prueba de Funcionalidad el método de prueba que se seleccionó fue el de Caja Negra el cual se describe a continuación.

**Método de Caja Negra**

Se refiere a las pruebas que se llevan a cabo sobre la interfaz del software, por lo que los casos de prueba pretenden demostrar que las funciones del software son operativas, que la entrada se acepta de forma adecuada, que se produce una salida correcta y que la integridad de la información externa se mantiene. Esta examina algunos aspectos del modelo fundamentalmente del sistema sin tener mucho en cuenta la estructura interna del software. Se obtienen conjuntos de condiciones de entrada que ejerciten todos los requisitos funcionales del mismo.

A continuación se mencionan algunas de las de técnicas para el diseño de casos de prueba que complementan este método.

**Técnicas de diseño de casos de prueba**

El diseño de casos de prueba se centra en un conjunto de técnicas que satisfacen los objetivos globales de la prueba. Las técnicas más comunes son:

> **Análisis de los valores límites:** la experiencia muestra que los casos de prueba que exploran las condiciones límites producen mejor resultado que aquellos que no lo hacen. Las condiciones límites son aquellas que se hallan en los márgenes de la clase de equivalencia, tanto de entrada como de salida. Por ello, se ha desarrollado el análisis de valores límites como técnica de prueba. Esta técnica lleva a elegir los casos de prueba que ejerciten los valores límites. Por lo tanto, esta técnica complementa la de partición de equivalencia de manera que en lugar de seleccionar cualquier caso de prueba de las clases válidas e inválidas, se eligen los casos de prueba en los extremos y en lugar de centrarse sólo en el dominio de entrada, los casos de prueba se diseñan también considerando el dominio de salida (Roger S., 2005).

Las pautas para desarrollar casos de prueba con esta técnica son:

- Si una condición de entrada especifica un rango de valores, se diseñarán casos de prueba para los dos límites del rango y otros dos casos para situaciones justo por debajo y por encima de los extremos.
- Si una condición de entrada especifica un número de valores, se diseñan dos casos de prueba para los valores mínimo y máximo, además de otros dos casos de prueba para valores justo por encima del máximo y justo por debajo del mínimo.
- Aplicar las reglas anteriores a los datos de salida. Si la entrada o salida de un programa es un conjunto ordenado, habrá que prestar atención a los elementos primero y último del conjunto (Roger S., 2005).

**Particiones de equivalencia**: Pressman presenta la partición equivalente dentro del Método de Prueba de Caja Negra que divide el campo de entrada de un programa en clases de datos de los que se pueden derivar casos de prueba. Un caso de prueba ideal descubre de forma inmediata una clase de errores que, de otro modo, requeriría la ejecución de muchos casos antes de detectar el error genérico. La partición equivalente se dirige a la definición de casos de prueba que descubran clases de errores, reduciendo así el número total de casos de prueba que hay que desarrollar (Roger S., 2005).

El diseño de casos de prueba para la partición equivalente se basa en una evaluación de las clases de equivalencia para una condición de entrada. Una clase de equivalencia representa un conjunto de estados válidos o inválidos para condiciones de entrada. Una condición de entrada es un valor numérico específico, un rango de valores, un conjunto de valores relacionados o una condición booleana (sí o no). Las clases de equivalencia se pueden definir de acuerdo a las siguientes directrices (Roger S., 2005):

- Si una condición de entrada especifica un rango, se define una clase de equivalencia válida y dos inválidas.
- Si una condición de entrada requiere un valor específico, se define una clase de equivalencia válida y dos inválidas.
- Si una condición de entrada especifica un miembro de un conjunto, se define una clase de equivalencia válida y una inválida.
- Si una condición de entrada es booleana, se define una clase válida y una inválida.

Aplicando esas directrices para la derivación de clases de equivalencia, se pueden desarrollar y ejecutar casos de prueba para cada elemento de datos del dominio de entrada. Los casos de prueba se seleccionan de forma que se ejercite el mayor número de atributos de cada clase de equivalencia a la vez (Roger S., 2005). El objetivo es minimizar el número de casos de prueba, así cada caso de prueba debe considerar tantas condiciones de entrada como sea posible. No obstante, es necesario realizar con cierto cuidado los casos de prueba de manera que no se enmascaren faltas. Así, para crear los casos de prueba a partir de las clases de equivalencia se deben seguir los siguientes pasos:

- Asignar a cada clase de equivalencia un número único.
- Hasta que todas las clases de equivalencia hayan sido cubiertas por los casos de prueba, se escribirá un caso que cubra tantas clases válidas no incorporadas como sea posible.
- Hasta que todas las clases de equivalencia no válidas hayan sido cubiertas por casos de prueba, escribir un caso de prueba para cubrir una única clase no válida no cubierta.

La técnica de particiones equivalentes para el diseño de casos de prueba es la que se emplea actualmente en la universidad por esa esta razón fue la que se utilizó para el desarrollo de este trabajo.

**Diseño de Casos de Prueba**

Los casos de prueba son un conjunto de condiciones que determinarán si los requisitos establecidos por los usuarios finales de una aplicación son satisfactorios, con el objetivo de detectar la mayor cantidad de defectos posibles en el software y satisfacer las necesidades del cliente.

El propósito de un caso de prueba es especificar una forma de probar el sistema, incluyendo las entradas con las que se prueba, los resultados esperados y las condiciones bajo las que ha de probarse. Los casos de prueba son un producto de desarrollo de software, que ayudan a validar y verificar las expectativas de los stakeholders[1]. Los requisitos son la fuente principal para obtener los casos de prueba pero no son el único medio y muchas veces son insuficientes para proporcionar una base completa que permita desarrollar las pruebas. Por lo que es necesario considerar otros elementos como riesgos, restricciones, tecnologías, cambios y fallos. Normalmente, un caso de prueba se deriva de un caso de uso en el modelo de casos de uso. Con estos casos de prueba se validan los requisitos funcionales del sistema (Jacobson, 2000).

Los siguientes son casos de prueba comunes:

- Un caso de prueba que especifica cómo probar un caso de uso o un escenario específico de un caso de uso. Un caso de prueba incluye la verificación del resultado de la interacción entre los actores y el sistema, que se satisfacen las precondiciones y pos condiciones especificadas por el caso de uso y que se sigue la secuencia de acciones especificadas por el caso de uso.
- Un caso de prueba que especifica cómo probar una realización de caso de uso o un escenario específico de la realización. Un Caso de Prueba de este tipo puede incluir la verificación de la interacción entre los componentes que implementan dicho caso de uso.

Los casos de prueba escritos, incluyen una descripción de la funcionalidad que se probará, la cual es seleccionada ya sea de los requisitos o de los casos de uso.

**Diseño de Casos de Prueba basados en Casos de Usos**

Un caso de uso es un fragmento de funcionalidad del sistema que proporciona al usuario un resultado importante. Los Casos de Uso representan los requisitos funcionales. Sin embargo, no

[1]Desde el punto de vista del desarrollo de sistemas es aquella persona o entidad que está interesada en la realización de un proyecto o tarea, auspiciando el mismo ya sea mediante su poder de decisión y/o de financiamiento o a través de su propio esfuerzo.

son solo una herramienta para especificar los requisitos de un sistema, también guían su diseño, implementación, y prueba; guían el proceso de desarrollo (Rumbaugh, 1999).

Los casos de prueba basados en casos de uso reflejan la trazabilidad de los mismos (Funcionalidad), ya que muestran una secuencia ordenada de eventos, al describir flujos básicos, flujos alternos, precondiciones y post-condiciones, estos se expresan a través de los diferentes escenarios que posee el Caso de Prueba (Actas de Talleres de Ingeniería del Software y Bases de Datos, Vol. 1, 2007).

Se debería diseñar un caso de prueba para cada caso de uso. Los escenarios de un caso de uso se identifican describiendo los distintos caminos del flujo básico y flujo alternativo del caso de uso. Existe otra manera por la cual realizar los diseños de casos de prueba como se explica a continuación.

**Diseño de Casos de Prueba basados en Requisitos**

Los requisitos son de gran importancia para el desarrollo de software ya que determinan lo que se pretende construir, es decir, las capacidades y/o cualidades que el futuro sistema debe cumplir para satisfacer las necesidades de los clientes. De forma general los requisitos de software especifican lo que el sistema debe hacer, pero sin expresar cómo debe hacerlo. Expresan lo que el cliente desea obtener y constituyen una especificación de lo que debería ser implementado.

Según Roger S. Pressman, "*los requisitos comprenden necesidades de información y control, funcionalidad del producto y comportamiento, rendimiento general del producto, diseño, restricciones de la interfaz y otras necesidades especiales"* (Pressman, 2005).

Los Casos de Prueba basados en Requisitos se utilizan para determinar si el requisito de una aplicación es parcial o completamente satisfactorio.

## 1.3 Herramienta de automatización de Pruebas funcionales

Hoy en día han surgido tendencias en virtud de demostrar la importancia de las pruebas de software, así como proponer mejoras a las mismas y de esta forma, incrementar la calidad del proceso de desarrollo. Entre ellas se encuentran vincular el proceso de pruebas a lo largo del ciclo de vida de desarrollo de software y la automatización de las mismas (Susana, 2008).

Después de consultar la bibliografía no se encontraron herramientas capaces de crear diseños de casos de prueba, lo más cercano a lo que se desea implementar son las herramientas que automatizan pruebas funcionales, por lo que se decide profundizar en este tema.

La automatización de las pruebas es la parte del proceso de desarrollo, en la que el software de automatización es utilizado para controlar la ejecución de pruebas, comparación de los resultados, la preparación de precondiciones y la realización de informes (Sonia, y otros, 2010). En el grupo de avanzada entre las herramientas de automatización de Pruebas funcionales se encuentra Selenium, es la única herramienta que en estos momentos se planea su implantación en la UCI, se encuentra siendo estudiada por el personal de CALISOFT[2].Por lo que se enfoca en ella el estudio realizado.

**Selenium**

Conjunto de herramientas para automatizar pruebas sobre aplicaciones web. Además, funciona en varios sistemas operativos como Windows, Linux y Macintosh. Es un software de código abierto, liberado bajo la licencia Apache 2.0 y puede ser descargado y utilizado sin costo alguno (SeleniumHQ, 2013).

Selenium realiza pruebas de regresión, las cuales se basan en verificar el estado de las no conformidades encontradas en iteraciones anteriores a una misma funcionalidad o cuando una prueba debe ser repetida en múltiples ocasiones. Por lo que siempre existirá la necesidad de diseñar casos de prueba para ser aplicados en una primera iteración. Para que así la herramienta pueda grabar las pruebas que va a automatizar.

Selenium mejora considerablemente el tiempo en que se realizan las Pruebas funcionales a una aplicación web pero no resuelve los problemas detectados en el diseño de los casos de prueba en el centro CEGEL.

Para el desarrollo de la propuesta de solución se hizo necesario el estudio de las metodologías de desarrollo de software así como las herramientas de modelado ya que son una guía para elaboración de la misma.

[2] CALISOFT: Centro Nacional de Calidad de Software.

## 1.4. Metodologías de Desarrollo de Software y herramientas de modelado

Para obtener un software con la calidad requerida, es necesario lograr un correcto ciclo de desarrollo de software, que permita organizar y efectuar satisfactoriamente los procesos que intervienen en la construcción del mismo. La correcta realización de estos procesos implica la utilización de metodologías de desarrollo que guíen la forma en que se aplica la Ingeniería de Software elevando la productividad del ciclo de desarrollo en aras de lograr un producto confiable y eficiente. A continuación se valoran algunas metodologías.

Existen dos grandes grupos en los cuales se dividen las metodologías: tradicionales o pesadas y los procesos ágiles.

Actualmente persisten varias metodologías Orientadas a Objeto basadas en UML entre las que se encuentran Proceso Unificado del Software (RUP) y XP (Programación Extrema). A continuación se estará viendo una breve descripción de ambas metodologías de desarrollo de software.

**Programación Extrema**

La Programación Extrema (XP por sus siglas del inglés Extreme Programming) es una metodología ágil de desarrollo de software que se basa en la simplicidad, la comunicación y la retroalimentación o reutilización del código desarrollado. La metodología XP tiene dos objetivos esenciales, la satisfacción del cliente tratando de darle el software que necesita y en el momento que lo requiere; por otra parte XP se propone potenciar al máximo el trabajo en grupo identificando a cada miembro del equipo de desarrollo con el producto entregable al cliente (Beck, 1999).
La misma propone actividades básicas para el desarrollo de un buen software las cuales se describen a continuación:

- Planificación Incremental: Los requisitos se registran en tarjetas de historias y las historias a incluir en una entrega se determinan según el tiempo disponible y su prioridad relativa.

Los desarrolladores dividen estas historias en tareas de desarrollo:

- Entregas Pequeñas: El mismo conjunto útil de funcionalidad que proporcione valor de negocio se desarrolla primero. Las entregas del sistema son frecuentes e incrementalmente añaden funcionalidad a la primera entrega.

- Diseño Sencillo: Solo se lleva a cabo el diseño necesario para cumplir los requisitos actuales.
- Desarrollo previamente probado: Se utiliza un sistema de pruebas de unidad automatizado para escribir pruebas para nuevas funcionalidades antes de que estas se implementen.
- Refactorización: Se espera que todos los desarrolladores refactoricen el código continuamente tan pronto como encuentren posibles mejoras en el código.

Esto conserva el código sencillo y mantenible.

- Programación en Parejas: Los desarrolladores trabajan en parejas, verificando cada uno el trabajo del otro proporcionando la ayuda necesaria para hacer siempre un buen trabajo.
- Propiedad Colectiva: Las parejas de desarrolladores trabajan en todas las áreas del sistema, de modo que no desarrollen islas de conocimiento y todos los desarrolladores posean todo el código. Cualquiera puede cambiar cualquier cosa.
- Integración Continua: En cuanto acaba el trabajo en una tarea, se integra en el sistema entero.

Después de la integración, se deben pasar al sistema todas las pruebas de unidad.

- Ritmo Sostenible: No se consideran aceptables grandes cantidades de horas extras, ya que a menudo el efecto que tienen es que se reduce la cantidad del código y la productividad a medio plazo.
- Cliente Presente: Debe estar disponible al equipo de la XP un representante de los usuarios finales del sistema (el cliente) a tiempo completo. En un proceso de la programación externa, el cliente es miembro del equipo de desarrollo y es responsable de formular al equipo los requisitos del sistema para su implementación.

El ciclo de desarrollo de XP incluye cuatro fases:

- Planificación del proyecto

**Historias de usuario:** el primer paso de cualquier proyecto que siga la metodología XP es definir las historias de usuario con el cliente. Las historias de usuario tienen la misma finalidad que los casos de uso pero con algunas diferencias: Constan de 3 ó 4 líneas escritas por el cliente en un lenguaje no técnico sin hacer mucho hincapié en los detalles; no se debe hablar ni de posibles algoritmos para su implementación ni de diseños de base de datos adecuados. Son usadas para estimar tiempos de desarrollo de la parte de la aplicación que describen. También se utilizan en la fase de pruebas, para verificar si el programa cumple con lo que especifica la historia de usuario. Cuando llega la hora de implementar una historia de usuario, el cliente y los desarrolladores se reúnen para concretar y detallar lo que tiene que hacer dicha historia. El tiempo de desarrollo ideal para una historia de usuario es entre 1 y 3 semanas.

**Release Planning:** después de tener ya definidas las historias de usuario es necesario crear un plan de entrega, en inglés "Release plan", donde se indiquen las historias de usuario que se crearán para cada versión del programa y las fechas en las que se publicarán estas versiones. Un plan de entrega es una planificación donde los desarrolladores y clientes establecen los tiempos de implementación ideales de las historias de usuario, la prioridad con la que serán implementadas y las historias que serán implementadas en cada versión del programa. Después de un "Release plan" tienen que estar claros estos cuatro factores: los objetivos que se deben cumplir (que son principalmente las historias que se deben desarrollar en cada versión), el tiempo que tardarán en desarrollarse y publicarse las versiones del programa, el número de personas que trabajarán en el desarrollo y cómo se evaluará la calidad del trabajo realizado.

**Iteraciones:** todo proyecto que siga la metodología XP se ha de dividir en iteraciones. Al comienzo de cada iteración los clientes deben seleccionar las historias de usuario definidas en el "Release planning" que serán implementadas. También se seleccionan las historias de usuario que no pasaron el test de aceptación que se realizó al terminar la iteración anterior. Estas historias de usuario son divididas en tareas de entre 1 y 3 días de duración que se asignarán a los programadores.

**La Velocidad del Proyecto:** es una medida que representa la rapidez con la que se desarrolla el proyecto; estimarla es muy sencillo, basta con contar el número de historias de usuario que se pueden implementar en una iteración; de esta forma, se sabrá el cupo de historias que se pueden

desarrollar en las distintas iteraciones. Usando la velocidad del proyecto controlaremos que todas las tareas se puedan desarrollar en el tiempo del que dispone la iteración. Es conveniente reevaluar esta medida cada 3 ó 4 iteraciones y si se aprecia que no es adecuada hay que negociar con el cliente un nuevo "Release Plan".

**Programación en Parejas:** La metodología XP aconseja la programación en parejas pues incrementa la productividad y la calidad del software desarrollado. El trabajo en pareja involucra a dos programadores trabajando en el mismo equipo; mientras uno codifica haciendo hincapié en la calidad de la función o método que está implementando, el otro analiza si ese método o función es adecuado y está bien diseñado. De esta forma se consigue un código y diseño con gran calidad.

**Reuniones Diarias:** Es necesario que los desarrolladores se reúnan diariamente y expongan sus problemas, soluciones e ideas de forma conjunta. Las reuniones tienen que ser fluidas y todo el mundo tiene que tener voz y voto.

- Diseño

**Diseños Simples:** la metodología XP sugiere que hay que conseguir diseños simples y sencillos. Hay que procurar hacerlo todo lo menos complicado posible para conseguir un diseño fácilmente entendible y desarrollable que a la larga costará menos tiempo y esfuerzo desarrollar.

**Glosarios de Términos:** Usar glosarios de términos y una correcta especificación de los nombres de métodos y clases ayudará a comprender el diseño y facilitará sus posteriores ampliaciones y la reutilización del código.

**Riesgos:** Si surgen problemas potenciales durante el diseño, XP sugiere utilizar una pareja de desarrolladores para que investiguen y reduzcan al máximo el riesgo que supone ese problema.

**Funcionabilidad extra:** Nunca se debe añadir funcionalidad extra al programa aunque se piense que en un futuro será utilizada. Sólo el 10% de la misma es utilizada, lo que implica que el desarrollo de funcionalidad extra es un desperdicio de tiempo y recursos.

**Refactorizar:** es mejorar y modificar la estructura y codificación de códigos ya creados sin alterar su funcionalidad. Supone revisar de nuevo estos códigos para procurar optimizar su funcionamiento. Es muy común rehusar códigos ya creados que contienen funcionalidades que no serán usadas y diseños obsoletos.

- Codificación

Como ya se dijo en la introducción, el cliente es una parte más del equipo de desarrollo; su presencia es indispensable en las distintas fases de XP. A la hora de codificar una historia de usuario su presencia es aún más necesaria. No olvidemos que los clientes son los que crean las historias de usuario y negocian los tiempos en los que serán implementadas. Antes del desarrollo de cada historia de usuario el cliente debe especificar detalladamente lo que ésta hará y también tendrá que estar presente cuando se realicen los test que verifiquen que la historia implementada cumple la funcionalidad especificada. La codificación debe hacerse ateniendo a estándares de codificación ya creados. Programar bajo estándares mantiene el código consistente y facilita su comprensión y escalabilidad.

- Pruebas

Uno de los pilares de la metodología XP es el uso de test [3]para comprobar el funcionamiento de los códigos que vayamos implementando. El uso de los test en XP es el siguiente:

- Se deben crear las aplicaciones que realizarán los test con un entorno de desarrollo específico para test.
- Hay que someter a tests las distintas clases del sistema omitiendo los métodos más triviales.
- Se deben crear los test que pasarán los códigos antes de implementarlos; en el apartado anterior se explicó la importancia de crear antes los test que el código.
- Un punto importante es crear test que no tengan ninguna dependencia del código que en un futuro evaluará.
- Como se comentó anteriormente los distintos test se deben subir al repositorio de código acompañados del código que verifican.
- Test de aceptación. Los test mencionados anteriormente sirven para evaluar las distintas tareas en las que ha sido dividida una historia de usuario.
- Al ser las distintas funcionalidades de la aplicación no demasiado extensas, no se harán test que analicen partes de las mismas, sino que las pruebas se realizarán para las funcionalidades generales que debe cumplir el programa especificado en la descripción de requisitos.(ver Anexo #1)

[3] Test o Tests: Prueba o Pruebas.

En XP se comienza en pequeño y se añaden funcionalidades con la retroalimentación continua y no antes de que sean necesarias; el manejo de cambios es importante en el proceso; el costo del cambio no depende de la fase o etapa y el cliente o el usuario se convierte en miembro del equipo (Beck, 1999).

**Proceso Unificado del Software**

El Proceso Unificado del Software (RUP por sus siglas en inglés), es una metodología de desarrollo de software, pesada y orientada a objetos, es un marco de trabajo genérico que puede ser especializado para una gran variedad de software para distintas áreas de aplicación, diferentes tipos de organizaciones, diferentes niveles de aptitud y diferentes tamaños de proyectos (Ivar Jacobson, 2004).

RUP está basado en componentes interconectados a través de interfaces y utiliza UML como lenguaje de modelado de procesos, es dirigido por casos de uso, centrado en la arquitectura, es iterativo e incremental (Ivar Jacobson, 2004).

- Dirigido por casos de uso, porque los casos de uso reflejan lo que los usuarios futuros desean y necesitan, lo cual se capta cuando se modela el negocio y se representa a través de los requisitos, a partir de ahí, los casos de uso guían el proceso de desarrollo.
- Centrado en la arquitectura, porque la arquitectura muestra una visión común del sistema completo, en la que el equipo del proyecto y los usuarios deben estar de acuerdo, por lo que describe los elementos del modelo que son más importantes producirlos económicamente.
- Iterativo e incremental, porque RUP propone que cada fase se desarrolle en iteraciones, y cada iteración tiene que proponerse un incremento en el proceso de desarrollo del software.

Luego de un estudio sobre estas metodologías de desarrollo, analizando las características de cada una, se determina que la metodología de software a utilizar es XP. Esta metodología se ajusta a las necesidades de la solución a implementar: el tamaño del equipo de desarrollo, en este caso de dos persona; necesidad de versiones funcionales de la solución a corto plazo; dificultad para adoptar una metodología robusta a causa de la cantidad de documentación generada para un equipo de desarrollo tan pequeño.

**Lenguaje Unificado de Modelado**

El Lenguaje Unificado de Modelado (UML) es un lenguaje para el desarrollo de software orientado a objetos, su propósito es visualizar, especificar, construir y documentar proyectos de software. Ayuda al usuario a entender la realidad de la tecnología y la posibilidad de que reflexione antes de invertir y gastar grandes cantidades en proyectos que no estén seguros en su desarrollo, reduciendo el coste y el tiempo empleado en la construcción de las piezas que constituirán el modelo; posee una gran cantidad de propiedades que han sido las que han contribuido a hacer de UML el estándar de la industria que es en realidad, algunas de estas propiedades son (Larman, 2004):

- Modela estructuras complejas.
- Las estructuras más significativas que soportan tienen su fundamento en las tecnologías orientadas a objetos, tales como objetos, clase, componentes y nodos.
- Utiliza operaciones abstractas como guía para variaciones futuras, incrementando variables si es necesario.

### 1.5. Herramientas de Ingeniería de Software Asistida por Ordenador

Se puede definir a las Herramientas CASE (*Computer Aided Software Engineering* y en su traducción al español significa Ingeniería de Software Asistida por Computación) como un conjunto de programas y ayudas que dan asistencia a los analistas, ingenieros de software y desarrolladores, durante todos los pasos del ciclo de vida de desarrollo de un software.

CASE se define también como:

- Conjunto de métodos, utilidades y técnicas que facilitan la automatización del ciclo de vida del desarrollo de sistemas de información, completamente o en alguna de sus fases.
- La sigla genérica para una serie de programas y una filosofía de desarrollo de software que ayuda a automatizar el ciclo de vida de desarrollo de los sistemas.
- Una innovación en la organización, un concepto avanzado en la evolución de tecnología con un potencial efecto profundo en la organización. Se puede ver al CASE como la unión de las herramientas automáticas de software y las metodologías de desarrollo de software formales.

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

- Verificar el uso de todos los elementos en el sistema diseñado.

- Automatizar el dibujo de diagramas.
- Ayudar en la documentación del sistema.
- Ayudar en la creación de relaciones en la Base de Datos.
- Generar estructuras de código (2010).

Actualmente existen Herramientas CASE tales como:

- Visual Paradigm for UML.
- Rational Rose.
- Enterprise Architect.
- Umbrello.
- ArgoUML.

**Rational Rose**

El Rational Rose es una herramienta CASE desarrollada por Rational Corporation, basada en UML. Esta herramienta propone la utilización de cuatro tipos de modelos para realizar un diseño del sistema, utilizando una vista estática y otra dinámica de los modelos del sistema (lógico y físico). Permite crear y refinar vistas creando de esta forma un modelo completo que representa el dominio del problema y el sistema de software. Permite crear los diferentes diagramas que se generan en el proceso de ingeniería durante el desarrollo del software. Presenta un gran número de estereotipos que permiten el proceso de modelación del software. Esta herramienta es capaz de generar el código fuente de las clases definidas en el flujo de trabajo de diseño. Proporciona mecanismos para realizar ingeniería directa e inversa, posibilita la construcción de un modelo de casos de usos, identifica los objetos y representa cómo interactúan con los diagramas de secuencia y colaboración, así como otras operaciones. Además Rational Rose organiza sus diagramas en vistas: la vista de casos de uso, la vista lógica, la vista de componentes y la vista de despliegue. El uso de estas vistas facilita la organización del trabajo para una mejor comprensión del mismo (Cámara, 2005).

Rational Rose es una potente herramienta que se integra bien con RUP y se ajusta en procesos de negocio complejos y por ende proyectos grandes, sin embargo el sistema requerido como se describe en secciones anteriores no presenta procesos de negocios complejos, sumándole que es una herramienta propietaria por lo que atenta contra la política de migración de la universidades por eso que no se selecciona como herramienta CASE para el desarrollo de los artefactos de la presente investigación.

**Visual Paradigm for UML**

Visual Paradigm for UML es una herramienta CASE profesional que soporta la última versión de UML 2.4 así como el ciclo de vida completo del desarrollo de software, análisis y diseño orientados a objetos, construcción, pruebas y despliegue, exportación desde Rational Rose, exportación e importación XML, generación de informes y edición de figuras. Visual Paradigm tiene disponible distintas versiones: Enterprise, Professional, Standard, Modeler, Personal y Community que es gratuita (Paradigm, 1995).

Visual Paradigm ofrece además (Paradigm, 1995):

- Diseño centrado en casos de uso y enfocado al negocio que genera un software de mayor calidad.
- Uso de un lenguaje estándar común a todo el equipo de desarrollo que facilita la comunicación.
- Capacidades de ingeniería directa e inversa.
- Modelo y código que permanece sincronizado en todo el ciclo de desarrollo.
- Disponibilidad de múltiples versiones, para cada necesidad.
- Disponibilidad de integrarse en los principales IDE.
- Disponibilidad en múltiples plataformas.

Visual Paradigm desde los inicios del proyecto productivo fue definida como herramienta CASE a utilizar en el proceso de desarrollo del proyecto debido a que es una herramienta multiplataforma que se adecua con las políticas migratorias de la universidad hacia software libre y gratuito, por otra parte brinda un conjunto de programas y ayudas que dan asistencia a los analistas, ingenieros de software y desarrolladores generando porciones de código automáticos en correspondencia con el lenguaje utilizado para el desarrollo del sistema, brindando la capacidad de gestionar el modelado de las entidades de la base de datos así como plasmar las relaciones entre ellas.

## 1.6. Herramientas y Tecnologías para el desarrollo

Se decidió trabajar con tecnología web, por las ventajas que proporciona en cuanto a soporte, disponibilidad y eficiencia, pues facilita la actualización y mantenimiento de aplicaciones web sin distribuir e instalar software a miles de usuarios potenciales, y además, la estructura de almacenamiento y centralización de la información, también puede ser usado con solo

conectarse a Internet desde cualquier navegador, siempre y cuando se tenga acceso a este. A continuación se justifican las tecnologías actuales para el desarrollo de software enmarcado en las principales metodologías y herramientas, plataformas de desarrollo y métricas. En el proceso de selección se priorizaron las herramientas y tecnologías de código abierto o pertenecientes al software libre que contribuyan el desarrollo de aplicaciones web. Teniendo en cuenta lo anterior, se determinó:

### 1.6.1. Servidor de aplicaciones

Tipo de servidor que permite el procesamiento de datos de una aplicación. Las principales ventajas de la tecnología de los servidores de aplicación son la centralización y la disminución de la complejidad del desarrollo de aplicaciones, dado que las aplicaciones no necesitan ser programadas; en su lugar, estas son ensambladas desde bloques provistos por el propio servidor (Herrara, 2009).

**Apache 2.2**

Apache es un servidor web de software libre, cuyo objetivo es servir o suministrar páginas web (en general, hipertextos) a los clientes web o navegadores que las solicitan, funciona sobre cualquier plataforma, permite que otros ordenadores vean la Web mediante un navegador. Es una solución altamente configurable y extensible a través de módulos, se integra perfectamente con varias tecnologías, lenguajes, plataformas y bases de datos (2008).

### 1.6.2. Lenguajes de programación

Dependiendo del tipo de programa que se desee realizar hay diferentes lenguajes en los que se puede programar: Java, C++, Visual C# .NET, PHP, HTML entre otros. Es importante tener en cuenta sus características para elegir el correcto. El estudio comparativo del lenguaje de programación estuvo condicionado por el conocimiento del equipo de desarrollo, teniendo en cuenta la premura del cliente referente al uso del sistema. Es por esto que a pesar de existir diversos lenguajes, con características importantes, se centra el estudio en PHP, con el objetivo de ganar en tiempo, aprovechar la facilidad de aprendizaje, las novedades que aporta y la necesidad de crear una aplicación web.

**PHP 5.0**

PHP es un lenguaje interpretado de propósitos generales ampliamente utilizado, especialmente para el desarrollo web y puede ser embebido en páginas HTM. Dentro de sus principales

características están: rapidez, facilidad de aprendizaje, soporte multiplataforma tanto de diversos sistemas operativos, como servidores HTTP y de bases de datos. Además posee capacidad de expandir su potencial utilizando la enorme cantidad de módulos llamados extensiones. Todas las funciones del sistema están explicadas y ejemplificadas en un único archivo de ayuda. Es libre, por lo que se presenta como una alternativa de fácil acceso para todos. Permite las técnicas de Programación Orientada a Objetos, tiene una biblioteca nativa de funciones sumamente amplia e incluida, no requiere definición de tipos de variables y tiene manejo de excepciones (PHP, 2001).

Si bien PHP no obliga a quien lo usa a seguir una determinada metodología a la hora de programar aun estando dirigido a alguna en particular, el programador puede aplicar en su trabajo cualquier técnica de programación y/o desarrollo que le permita escribir código ordenado, estructurado y manejable. Un ejemplo de esto son los desarrollos que en PHP se han hecho del patrón Modelo Vista Controlador (MVC), que permiten separar el tratamiento y acceso a los datos, la lógica de control y la interfaz de usuario en tres componentes independientes (Sklar, 2004).

**Javascript**

Javascript es un lenguaje de programación web, se utiliza actualmente para desarrollo de sitios web, es un lenguaje de programación interpretado, por lo que no es necesario compilar los programas para ejecutarlo, al usarlo se puede crear páginas web dinámicas, menús desplegables, efectos visuales sencillos, manipular datos y crear aplicaciones web, utilizando poca memoria y manteniendo un tiempo de descarga rápido para páginas web. Los programas escritos con Javascript se pueden probar directamente en cualquier navegador sin necesidad de procesos intermedios (Javascript, 2013).

**HTML**

HTML (Lenguaje de Marcas HiperTextuales) es un lenguaje de marcación diseñado para estructurar textos y presentarlos en forma de hipertexto, que es el formato estándar de las páginas web. Con el auge de Internet y el uso de navegadores como Internet Explorer, Opera, Firefox o Netscape, el HTML se ha convertido en uno de los formatos más populares que existen para la construcción de documentos y también de los más fáciles de aprender.

Establece una serie de nuevos elementos y atributos que reflejan el uso típico de los sitios web modernos. Posee etiquetas para manejar grandes conjuntos de datos permitiendo así generar

tablas dinámicas que pueden filtrar, ordenar y ocultar contenido. Propone mejoras en los formularios y nuevos tipos de datos que facilitan validar el contenido (Castillo Cantón, 2011).

**CSS3**

Las hojas de estilo en cascada (Cascading Style Sheets) hacen referencia a un lenguaje usado para describir la presentación semántica de un documento escrito en lenguaje de marcas. Su aplicación más común es dar estilo a páginas web escritas en lenguaje HTML y XHTML. CSS tiene una sintaxis muy sencilla, usa palabras claves para especificar los nombres de sus selectores, propiedades y atributos (Alvarez, 2013).

Después de seleccionados los lenguajes de programación a utilizar tanto del lado del servidor como del lado del cliente se prosigue a identificar los marcos de trabajo que satisfacen estos lenguajes.

### 1.6.3. Marco de Trabajo

Son una estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, que pueden servir de base para la organización y desarrollo de un software.

**Symfony 1.4.18**

Un proyecto en el marco de trabajo de Symfony 1.4.18 cuenta con una jerarquía de directorios bien definida ya que se organiza de forma lógica en aplicaciones las cuales se ejecutan en modo independiente respecto a otras aplicaciones del mismo proyecto, cada aplicación está compuesta por uno o más módulos que almacenan las acciones que representan cada una de las operaciones que se pueden realizar en un módulo determinado.

Symfony 1.4.18 está desarrollado completamente en PHP 5.3, ha sido probado en numerosos proyectos reales y se utiliza en sitios web de comercio electrónico de primer nivel, es compatible con la mayoría de gestores de bases de datos, como MySQL, PostgreSQL, Oracle y Microsoft SQL Server. Se puede ejecutar tanto en plataformas *nix (Unix, Linux, entre otras.) como en plataformas Windows (Synfony, la guia definitiva, 2008).

Symfony 1.4.18 incluye como novedades destacadas:

- Rendimiento: Mejora en el rendimiento de todos sus componentes. Las mejoras más notables se producen en el sistema de enrutamiento y en una nueva tarea llamada Project-optimize que optimiza el rendimiento de los proyectos en producción.
- Emails: Cuenta con un mecanismo sencillo, potente e integrado para el envío de emails a través de la librería SwiftMailer 4, proyecto incorporado a la gran familia de Symfony.
- Formularios: Sus formularios son tan fáciles de extender e intuitivos de utilizar ya que incluyen eventos para redefinir fácilmente su comportamiento y se han añadido varios métodos para facilitar su creación y uso.
- Depuración: la barra de depuración web incluye dos nuevos paneles dedicados a la vista y al envío de emails. El panel de la vista muestra información sobre las plantillas y elementos parciales utilizados para crear la página que visualiza el navegador y también incluye el listado completo de variables disponibles en la parte de la vista.
- Eficiencia: gracias al nuevo cargador automático de clases, ya no será necesario limpiar la caché cada vez que se añade una nueva clase. Las pruebas unitarias y funcionales también son más eficientes porque permiten volver a ejecutar solamente las pruebas que produjeron un error la última vez.

Ventajas del marco de trabajo Symfony 1.4.18 (Synfony, la guia definitiva, 2008):

- Permite la internacionalización para la traducción del texto de la interfaz, los datos y el contenido de localización.
- La presentación usa templates y layouts que pueden ser construidos por diseñadores de HTML que no posean conocimientos del marco de trabajo.
- Los formularios soportan la validación automática, lo cual asegura mejor calidad de los datos en las base de datos y una mejor experiencia para el usuario.
- El manejo de caché reduce el uso de banda ancha y la carga del servidor.
- La facilidad de soportar autenticación y credenciales facilita la creación de áreas restringidas y manejo de seguridad de los usuarios.
- El enrutamiento y las URL inteligentes hacen amigable las direcciones de las páginas de la aplicación.
- Las listas son más amigables, ya que permite la paginación, clasificación y filtraje automáticos.
- Los plugins proveen un alto nivel de extensibilidad

El marco de trabajo Symfony 1.4.18 hace aportes significativos al sistema informático propuesto pues permite importar código reutilizable a través del uso de los plugins para el manejo de funcionalidades vinculadas al sistema permitiendo manipular la autenticación de los usuarios gestionando el acceso que posee el usuario a la información; constituyendo la herramienta para el desarrollo del sistema llevado a cabo por dicho proyecto ya que viene a fin con las prestaciones solicitadas y el tipo de sistema a desarrollar.

El marco de trabajo seleccionado por la dirección del proyecto es Symfony 1.4.18 debido a que está diseñado para optimizar el desarrollo de las aplicaciones web, algunas de sus características son: separa la lógica de negocio, la lógica de servidor y la presentación de la aplicación web, proporciona varias herramientas y clases encaminadas a reducir el tiempo de desarrollo de una aplicación web compleja y automatiza las tareas más comunes, permitiendo al desarrollador dedicarse por completo a los aspectos específicos de cada aplicación.

**Doctrine**

Doctrine es un ORM que posee una poderosa capa de abstracción de BD. Una de sus características es la opción de escribir consultas de BD en un objeto apropiado orientado al dialecto SQL (Structured Query Language) y que se le denomina Lenguaje de Consulta de Doctrine o DQL del inglés Doctrine Query Language, inspirado por el Lenguaje de Consulta de Hibernate (HQL por sus siglas en inglés). Este proporciona a los desarrolladores una poderosa alternativa al SQL que mantiene la flexibilidad sin requerir duplicación de código innecesario (Doctrine, 2007).

**ExtJS**

ExtJS es neutral al lenguaje que se use en el servidor. Siempre que el resultado se envíe a la página en el formato adecuado, ExtJS no se preocupa de lo que pase en el servidor. Hay docenas de widgets[4] a escoger en ExtJS, incluyendo composiciones automáticas de páginas, pestañas, menús, barras de herramientas, diálogos, vistas en árbol. Proporciona un selector de nodos DOM extremadamente poderoso llamado DomQuery (puede usarse como una librería independiente, pero en el contexto de ExtJS se usará para seleccionar elementos para interactuar con ellos a

[4]Es un componente gráfico con el cual el usuario interactúa; usualmente presentado en archivos o ficheros pequeños, como por ejemplo, una ventana, una barra de tareas o una caja de texto.

través de la interfaz Element, contiene mucho de los métodos y propiedades de DOM que se necesitará proporcionando una interfaz conveniente, unificada y multinavegador) (ExtJS, 2008).

### 1.6.4. Entorno de Desarrollo Integrado (IDE)

Un Entorno de Desarrollo Integrado (IDE siglas de Integrated Development Environment en inglés), consiste en un programa informático que ha sido instalado en la máquina del desarrollador y cuyo objetivo es la elaboración de aplicaciones, es decir, es un entorno de programación que ha sido empaquetado como "Software", que va a contener un editor de código, un compilador, un depurador y un constructor de interfaz gráfica, que van a facilitar las diferentes tareas necesarias en el desarrollo y mantenimiento de cualquier tipo de aplicación (2010).

**Eclipse**

Este es un IDE de código abierto multiplataforma para desarrollar lo que el proyecto llama "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores. Eclipse fue desarrollado originalmente por IBM [5]como el sucesor de su familia de herramientas para Visual Age. Este IDE es ahora desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios. Eclipse es un IDE, puesto que provee herramientas para administrar áreas de trabajo, construir, lanzar y depurar aplicaciones, así como compartir artefactos con un equipo y versionar el código fuente. Es abierto, debido a que su diseño le permite ser extendido fácilmente por terceras partes. Ha sido utilizado para construir ambientes para un amplio rango de temas como el desarrollo en lenguaje de programación Java, Servicios Web, certámenes de programación de juegos, entre otros (Arthorne, y otros, 2004).

**Netbeans**

NetBeans es una herramienta para programadores pensada para escribir, compilar, depurar y ejecutar programas. Está escrito en Java, pero puede servir para otros lenguajes de programación. Existe además un número importante de módulos para extenderlo. El IDE NetBeans es un producto libre y gratuito sin restricciones de uso.

Ventajas:

[5]International Business Machines

- Auto-completado y documentación de funciones PHP: Rápido acceso a la documentación de PHP, y si se necesita más información se provee el link directo a la función.
- Auto-completado de código propio: Esto es una consecuencia del punto anterior, al documentar código con el formato esperado, estos serán mostrados.
- Atajos de teclado muy útiles: Permite a través de varios comandos la ejecución de numerosas funcionalidades y existen muchas más como integración con Xdebug, soporte para Symfony, Zend Framework, Smarty, entre otros.

NetBeans es el ambiente de desarrollo a utilizar para el desarrollo de la aplicación. Aunque ambos, Eclipse y NetBeans, ofrecen similares recursos para el desarrollo de aplicaciones de web, como es el caso que se presenta, NetBeans es la mejor opción debido a que proporciona otras características muy útiles como: gestor de ventanas, API[6]de acciones y API para la creación de diálogos y wizards[7].

### 1.6.5. Sistema Gestor de Base de Datos

En la actualidad los Sistemas de Gestión de Base de Datos (SGBD) sirven de interfaz entre la base de datos, el usuario y las aplicaciones que la utilizan, teniendo como propósito manejar de manera clara, sencilla y ordenada un conjunto de datos que posteriormente se convertirán en información relevante para una organización.

A continuación se exponen las características del SGBD seleccionado por el equipo de desarrollo así como los argumentos de su selección.

**PostgreSQL**

PostgreSQL es un servidor de base de datos, liberado bajo la licencia BSD[8]. Como muchos otros proyectos open source[9], el desarrollo de PostgreSQL no es manejado por una sola compañía sino que es dirigido por una comunidad de desarrolladores y organizaciones comerciales las cuales trabajan en su desarrollo, dicha comunidad denominada el PGDG (PostgreSQL Global Development Group). Fue el pionero en muchos de los conceptos existentes en el sistema objeto-relacional actual, incluido más tarde en otros sistemas de gestión comerciales. PostgreSQL es

[6]Interfaz de programación de aplicaciones.
[7]instalador independiente con asistente o guía (en inglés wizard, comúnmente usados en Windows).
[8]Licencia de Distribución de Software Berkeley.
[9] Código abierto.

un sistema objeto-relacional, ya que incluye características de la orientación a objetos (PostgreSQL-es, 2012).

**Características de PostgreSQL:**

- Implementación del estándar SQL92/SQL99.
- Soporta distintos tipos de datos, además del soporte para los tipos base, también soporta datos de tipo fecha, monetarios, elementos gráficos, datos sobre redes y cadenas de bits. También permite la creación de tipos propios.
- Incorpora una estructura de datos array10.
- Incorpora funciones de diversa índole: manejo de fechas, geométricas, orientadas a operaciones con redes, entre otras.
- Permite la declaración de funciones propias, así como la definición de disparadores.
- Soporta el uso de índices, reglas y vistas.
- Incluye herencia entre tablas, por lo que a este gestor de bases de datos se le incluye entre los gestores objeto-relacionales.
- Permite la gestión de diferentes usuarios, como también los permisos asignados a cada uno de ellos.

PostgreSQL es un gestor de bases de datos disponible en casi cualquier Unix (34 plataformas en la última versión estable), y una versión nativa de Windows, usando una estrategia de almacenamiento de filas llamada MVCC[11] para conseguir una mejor respuesta en ambientes de grandes volúmenes, teniendo un mantenimiento y ajuste mucho menor que los productos de los proveedores comerciales. Conservando todas las características, estabilidad y rendimiento dando al traste con las necesidades que plantea el proyecto productivo, dado por las características potenciales del gestor.

## 1.7. Conclusiones parciales

En este capítulo se trató lo referente a la calidad de software lo cual creó la base teórica de la investigación y se realizó un estudio de las herramientas, lenguajes y tecnologías más importantes en la actualidad. Después de tener todos estos elementos se arriba a las siguientes conclusiones:

[10]Tipo de dato el cual almacena una secuencia de objetos definida por el desarrollador.
[11]Multi-Versión para el control de concurrencia.

- El estudio de las Pruebas Funcionales definió que la generación de los diseños de casos de prueba se basará en el método de prueba de Caja Negra, a través de la utilización de la técnica de Partición equivalente.
- El estudio de sistemas existentes en el área del conocimiento de esta investigación reveló que las herramientas encontradas han sido diseñadas para la realización de pruebas automatizadas, específicamente para ejecutar pruebas de regresión a través de casos de prueba previamente elaborados. De esta manera sigue manifestándose la necesidad de la existencia de un equipo dedicado al diseño de los mismos, trayendo consigo demora e introducción de errores durante el desarrollo de esta actividad.
- Se define XP como metodología de desarrollo a seguir debido a la vinculación del cliente en el grupo de desarrollo y las facilidades que brinda una metodología ágil a un proyecto de corta duración.
- Se precisa Symfony como marco de trabajo para el desarrollo del sistema ya que presenta como principal ventaja la gran reutilización de código lo cual reducirá el tiempo de implementación.

# CAPÍTULO 2: PLANIFICACIÓN, DISEÑO Y DESARROLLO

## 2.1. Introducción del capítulo

En este capítulo se dará solución al problema planteado, analizando para ello el desarrollo del software a través del análisis y descripción de las tres primeras fases de la metodología XP: Planificación, Diseño, e Implementación, generándose los artefactos correspondientes para de esta forma obtener como resultado un producto con la calidad requerida por el usuario final.

## 2.2. Descripción de la solución

El sistema propuesto se basa en la creación de una herramienta para generar el artefacto Diseños de Casos de Prueba, además de generar los documentos de Especificación de Casos de Uso y Descripción de Requisitos; facilitando así el trabajo a los analistas en los proyectos del Centro y mejorar la elaboración de este artefacto.

## 2.3. Planificación

La metodología de desarrollo de software XP muestra la planificación como un permanente diálogo entre el usuario final y el equipo de desarrollo; definiéndose lo que el software tiene que resolver para que genere algún valor, la parte del software que debe hacerse primero, lo que se necesita hacer para saber si el negocio requiere o no de la informatización a través de un software, fechas de entrega de las primeras versiones del software, la organización del trabajo y el equipo de desarrollo y por último se detalla dentro de una versión del producto los problemas que deben ser resueltos con carácter urgente.

En esta fase el cliente establece la prioridad de cada historia de usuario y los programadores realizan una estimación del esfuerzo necesario de cada una de ellas. Se toman acuerdos sobre el contenido de la primera entrega y se determina un cronograma en conjunto con el cliente. Esta fase dura unos pocos días, identificándose además, el número y tamaño de las iteraciones, al igual que se plantean ajustes necesarios a la metodología según las características del proyecto.

**Historias de Usuario**

Las Historias de Usuario o HU son descritas por los propios clientes, tal y como ven ellos las necesidades del sistema. Por tanto constituyen descripciones cortas y escritas en el lenguaje del usuario, sin terminología técnica, las mismas conducen el proceso de creación de las pruebas de aceptación las cuales se utilizarán para verificar que las HU han sido implementadas correctamente, la principal diferencia es el nivel de detalle, el cual debe ser el mínimo posible que permita hacerse una ligera idea de cuánto costará implementar el sistema, siendo los desarrolladores los que hacen una estimación de cuánto tiempo les llevará implementar cada HU. Estas proporcionan los detalles sobre la estimación del riesgo y cuánto tiempo necesita la implementación de la misma (Angela, y otros, 2004).

Las Historias de Usuario deben ser:

- Valoradas por los clientes o usuarios: Los intereses de los clientes y de los usuarios no siempre coinciden, pero en todo caso, cada historia debe ser importante para alguno de ellos más que para el desarrollador.
- Estimables: Un resultado de la discusión de una Historia de Usuario es la estimación del tiempo que tomará completarla. Esto permite estimar el tiempo total del proyecto.
- Pequeñas: Las historias muy largas son difíciles de estimar e imponen restricciones sobre la planificación de un desarrollo iterativo. Generalmente se recomienda la consolidación de historias muy cortas en una sola historia.
- Verificables: Las HU cubren requisitos funcionales, por lo que generalmente son verificables. Cuando sea posible, la verificación debe automatizarse, de manera que pueda ser verificada en cada entrega del proyecto.

Durante la fase de exploración se identificaron 10 HU representando cada una las funcionalidades del sistema, a continuación se muestran algunas debido a su importancia en el desarrollo de la aplicación:

Tabla 1. HU. Gestionar documento de Especificación de casos de uso.

| **Historia de Usuario** | |
|---|---|
| **Número:** 2. | **Nombre de la Historia de Usuario:** Gestionar Documento de Especificación de Casos de Uso. |

| | |
|---|---|
| **Cantidad de modificaciones a la Historia de Usuario:** 0 | |
| **Usuario:** Analista | **Iteración asignada:** 1 |
| **Prioridad en negocio:** Alta | **Puntos estimados:** 16 días |
| **Riesgo en desarrollo:** Alto | **Puntos reales:** 16 días |
| **Descripción:** El sistema debe permitir al usuario la posibilidad de crear, modificar y eliminar una Especificación de Caso de Uso. | |
| **Observaciones:** | |
| **Prototipo de interfaces:** Ver anexo #1, anexo #2, anexo #3 | |

Tabla 2. HU. Gestionar documento de Descripción de requisitos.

| **Historia de Usuario** | |
|---|---|
| **Número:** 17. | **Nombre de la Historia de Usuario:** Gestionar Documento de Descripción de Requisitos. |
| **Cantidad de modificaciones a la Historia de Usuario:** 0 | |
| **Usuario:** Analista | **Iteración asignada:** 2 |
| **Prioridad en negocio:** Alta | **Puntos estimados:** 16 días |
| **Riesgo en desarrollo:** Alto | **Puntos reales:** 16 días |
| **Descripción:** El sistema debe permitir al usuario adicionar, modificar o eliminar una Descripción de Requisitos. | |
| **Observaciones:** | |
| **Prototipo de interfaces:** Ver anexo #4, anexo #5, anexo #6 | |

Tabla 3. HU. Gestionar Diseño de casos de prueba basado en caso de uso.

| **Historia de Usuario** | |
|---|---|
| **Número:** 6. | **Nombre de la Historia de Usuario:** Generar Diseño de Casos de Prueba basado en Casos de Uso. |
| **Cantidad de modificaciones a la Historia de Usuario:** 1 | |
| **Usuario:** Analista | **Iteración asignada:** 1 |
| **Prioridad en negocio:** Alta | **Puntos estimados:** 5 días |
| **Riesgo en desarrollo:** Alto | **Puntos reales:** 5 días |
| **Descripción:** El sistema deber permitir al usuario la opción de generar un Diseño de Casos de Prueba. | |
| **Observaciones:** | |
| **Prototipo de interfaces:** Ver anexo #7 | |

### Requisitos

Una vez realizadas por el cliente las distintas HU se procede a documentar las necesidades del mismo capturando los requisitos identificados a partir de las HU. Los requisitos son la base para un desarrollo exitoso así como para una plena conformidad con el entregable final. A continuación se muestra el listado de requisitos definidos para el sistema informático propuesto por la investigación.

### Requisitos funcionales

RF1- Adicionar Usuario

RF2- Modificar Usuario

RF3- Activar Usuario

RF3.1- Desactivar Usuario

RF4- Autenticar Usuario

RF5- Adicionar Documento de Especificación de Casos de Uso

RF5.1- Adicionar Encabezado de Especificación

RF5.2- Adicionar Introducción de Especificación

RF5.3- Adicionar Control de Cambios de Especificación

RF5.4- Adicionar Paquete de Análisis de Especificación

RF5.5- Adicionar Caso de Uso

RF5.5.1- Adicionar Evento

RF5.5.2- Adicionar Flujo Básico

RF5.5.3- Adicionar Flujo Alterno

RF5.5.4- Adicionar Prototipo de Interfaz de Caso de Uso

RF6- Modificar Documento de Especificación de Casos de Uso

RF6.1- Modificar Encabezado de Especificación

RF6.2- Modificar Introducción de Especificación

RF6.3- Modificar Control de Cambios de Especificación

RF6.4- Modificar Paquete de Análisis de Especificación

RF6.5- Modificar Caso de Uso

RF6.5.1- Modificar Evento

RF6.5.2- Modificar Flujo Básico

RF6.5.3- Modificar Flujo Alterno

RF6.5.4- Modificar Prototipo de Interfaz de Caso de Uso

RF7- Eliminar Documento de Especificación de Casos de Uso

RF8- Aprobar Documento de Especificación de Casos de Uso

RF10- Generar Documento de Especificación de Casos de Uso

RF11- Adicionar Documento de Descripción de Requisitos

RF11.1- Adicionar Encabezado de Descripción

RF11.2- Adicionar Paquete de Análisis de Descripción

RF11.3- Adicionar Introducción de Descripción

RF11.4- Adicionar Control de Cambios de Descripción

RF11.5- Adicionar Requisito

RF11.5.1- Adicionar Evento

RF11.5.2- Adicionar Flujo Básico

RF11.5.3- Adicionar Flujo Alterno

RF11.5.4 - Adicionar Prototipo de Interfaz de Requisito

RF12- Modificar Documento de Descripción de Requisitos

RF12.1- Modificar Encabezado de Descripción

RF12.2- Modificar Introducción de Descripción

RF12.3- Modificar Control de Cambios de Descripción

RF12.4- Modificar Paquete de Análisis de Descripción

RF12.5- Modificar Requisito

RF12.5.1- Modificar Evento

RF12.5.2- Modificar Flujo Básico

RF12.5.3- Modificar Flujo Alterno

RF12.5.4- Modificar Prototipo de Interfaz de Requisito

RF13- Eliminar Documento de Descripción de Requisitos

RF14- Aprobar Documento

RF15- Buscar Documento

RF16- Generar Documento de Descripción de Requisitos

RF17- Generar Diseño de Casos de Prueba basado en Requisitos

RF18- Generar Diseño de Casos de Prueba basado en Casos de Uso

**Requisitos no funcionales**

Los requisitos no funcionales (RNF) definen propiedades del sistema informático que se desea como producto final. Estos requisitos son normalmente a los que debe apuntar la arquitectura y si estos no son cumplidos, el sistema informático propuesto puede no funcionar o el cliente simplemente no aceptar el producto (Bikha, 2008).

**Apariencia o interfaz externa:**

- El sistema tiene que ofrecer una interfaz amigable, fácil de operar.
- El sistema tiene que mantener la línea de diseño establecida para la institución que mantiene la uniformidad y representatividad de la misma.
- Las interfaces tienen que ostentar un diseño sencillo, con pocas entradas, permitiendo un balance adecuado entre funcionalidad y simplicidad de tal manera que no se haga difícil para los usuarios la utilización del sistema.

**Usabilidad:**

- El software tendrá siempre la posibilidad de ayuda disponible para cualquier tipo de usuario, lo que le permitirá un avance considerable en la explotación de la aplicación en todas sus funcionalidades.
- El tiempo de entrenamiento requerido para que usuarios normales y avanzados sean productivos operando el sistema es de 7 días.

**Seguridad:**

- Para realizar una operación determinada el usuario deberá estar registrado.
- El sistema podrá ser usado en cualquier momento por todos los usuarios autorizados.

**Soporte:**

- Tiene que brindar soporte para grandes volúmenes de datos y velocidad de procesamiento.
- El sistema tiene que ser multiplataforma.

**Hardware:**

- Memoria RAM 1GB o superior.
- Disco duro de 20 GB o superior.
- Procesador Intel® a 1 GHz. de velocidad de procesamiento o superior.
- Tarjeta de red Ethernet.

**Software:**

- Multiplataforma, Internet Explorer, Mozilla Firefox.

**Estimación de esfuerzo**

Se realizó la estimación de esfuerzo para medir cuánto costará implementar cada una de las HU identificadas, para el buen desarrollo del sistema propuesto, llegando a los resultados que se muestran en el Anexo #2.

**Plan de iteraciones**

Este plan incluye varias iteraciones sobre el sistema antes de ser entregado. Los elementos que deben tenerse en cuenta durante su elaboración son: HU no abordadas, velocidad del proyecto, pruebas de aceptación no superadas y tareas no terminadas en la iteración. Este plan define cuáles HU serán implementadas en cada iteración. Tomando como base cada una de las HU y el esfuerzo que se requiere para el desarrollo de estas, se procede a fragmentar el trabajo en dos iteraciones; obteniendo un trabajo incremental, donde la comunicación entre el equipo de desarrollo y el cliente es de vital importancia.

A continuación se describen cada una de las iteraciones propuestas donde la duración total de iteraciones en días se obtiene a partir del esfuerzo en días estimado por el desarrollador para implementar cada HU:

Iteración # 1: Por problemas con la documentación necesaria para la implementación de la aplicación se decidió por los desarrolladores realizar la rama referente a Casos de Uso previendo que habían elementos comunes que después se podrían reutilizar en la rama de requisitos contribuyendo así a un avance considerable en la segunda iteración, resultado que las HU a desarrollar son las siguientes:

HU1- Autenticar usuario.
HU2- Gestionar Documento de Especificación de Casos de uso.
HU3- Generar Documento de Especificación de Casos de Uso.
HU4- Generar Diseño de Casos de Prueba basado en Casos de Uso.
HU5- Aprobar Documento.
HU6- Buscar Documento.
HU7- Gestionar Encabezado de Especificación.
HU8- Gestionar Introducción de Especificación.

HU9- Gestionar Control de Cambios de Especificación.

HU10- Gestionar Paquete de Análisis de Especificación.

HU11- Gestionar Caso de Uso.

HU12- Gestionar Evento de Caso de Uso.

HU13- Gestionar Flujo Básico de Caso de Uso.

HU14- Gestionar Flujo Alterno de Caso de Uso.

HU15- Gestionar Prototipo de Interfaz de Caso de Uso.

Iteración # 2: Se tuvieron en cuenta aquellas funcionalidades del sistema pertenecientes a la rama de Requisitos. Por otra parte en esta iteración se corrigen los errores encontrados en la iteración anterior, obteniéndose una nueva versión del sistema. Las HU de esta iteración a desarrollar son:

HU16- Gestionar Documento de Descripción de Requisitos.

HU17- Generar Documento de Descripción de Requisitos.

HU18- Generar Diseño de Casos de Prueba basado en Requisitos.

HU19- Gestionar Encabezado de Descripción de Requisitos.

HU20- Gestionar Introducción de Descripción de Requisitos.

HU21- Gestionar Control de Cambios de Descripción de Requisitos.

HU22- Gestionar Requisito.

HU23- Gestionar Evento de Requisito.

HU24- Gestionar Flujo Básico de Requisito.

HU25- Gestionar Flujo Alterno de Requisito.

HU26- Gestionar Prototipo de Interfaz de Requisito.

**Plan de duración de las iteraciones**

Este plan tiene como objetivo reflejar las HU que se desarrollarán en cada iteración, la duración de cada una, así como el orden en que serán implementadas en dichas iteraciones. En este caso se crea un solo plan de iteraciones debido a que existe un único equipo de desarrolladores.

Tabla 4. Plan de Iteraciones

| Iteración | Descripción de la iteración | Duración total |
|---|---|---|

| | | |
|---|---|---|
| 1 | Se tuvieron en cuenta las HU pertenecientes al módulo de Casos de Uso y las funcionalidades comunes a los dos módulos. | 7 semanas |
| 2 | Se tuvieron en cuenta las HU pertenecientes al módulo de Requisitos. Por otra parte en esta iteración se corregirán los errores encontrados en la iteración anterior, obteniéndose una nueva versión del sistema. | 4 semanas |

**Plan de entrega**

El plan de entregas es un cronograma con las HU que serán implementadas en cada iteración, es decir, un cronograma con las entregas de partes funcionales del software. A continuación se hace una propuesta de la fecha aproximada en que se harán versiones al sistema al finalizar cada iteración en la fase de implementación:

Ya elaboradas por el cliente las distintas HU y confeccionado por los desarrolladores el plan de iteración se procede a la confección del plan de entrega con la intención de que los mismos obtengan una estimación real, fijándose un periodo de tiempo sobre el cual se debe de implementar cada HU y definiéndose el grado de dificultad de la misma. Se presenta el plan de entrega elaborado por el equipo de desarrollo donde se reflejan las fechas de entrega para las primeras versiones de las HU. Ver Anexo #3.

## 2.4. Diseño

La metodología de desarrollo de software XP sugiere que se realicen diseños simples y sencillos, para de esta forma lograr que sean de fácil entendimiento en la fase de implementación, lo que le costará menos tiempo al desarrollador llevar a cabo esta tarea Para el diseño de las aplicaciones, la metodología XP no requiere la presentación del sistema mediante diagramas de clases utilizando notación UML, en su lugar se usan otras técnicas como las tarjetas CRC (Contenido, Responsabilidad y Colaboración). No obstante el uso de estos diagramas, puede aplicarse siempre y cuando influyan en el mejoramiento de la comunicación, no sea un peso su mantenimiento, no sean extensos y se enfoquen en la información importante. El uso de las tarjetas CRC permiten al programador centrarse y apreciar el desarrollo orientado a objetos olvidándose de los malos hábitos de la programación procedural clásica.

**Descripción de la Arquitectura**

La arquitectura de la aplicación está basada en el patrón arquitectónico MVC [12]el cual separa la lógica de negocio (el modelo) y la presentación (la vista) por lo que se consigue un mantenimiento más sencillo de las aplicaciones. El controlador se encarga de aislar al modelo y a la vista de los detalles del protocolo utilizado para las peticiones (HTTP, consola de comandos, email.). El modelo se encarga de la abstracción de la lógica relacionada con los datos, haciendo que la vista y las acciones sean independientes de, por ejemplo, el tipo de gestor de bases de datos utilizado por la aplicación. A continuación se muestra una breve descripción de cada una de las capas de la arquitectura definida.

[12] MVC: Modelo Vista Controlador

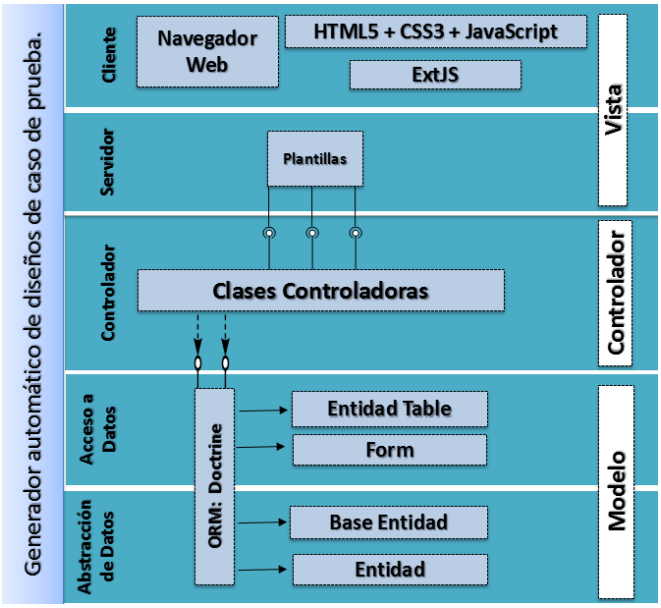



Figura 1. Esquema de la arquitectura del sistema.

**Vista**

Esta capa es la que ve el usuario, presenta el sistema al usuario, le comunica la información y captura la información del usuario en un mínimo de proceso. Esta capa se comunica únicamente con la capa del controlador. También es conocida como interfaz gráfica y debe tener la característica de ser amigable (entendible y fácil de usar) para el usuario. Esta capa contiene los componentes que implementan y muestran la interfaz de usuario (UI), además administra la interacción con el usuario y ejecuta la lógica asociada con la validación de los datos. Los componentes de UI proporcionan una forma para que los usuarios interactúen con la aplicación, mostrando los datos en el formato adecuado. Asimismo, obtienen y validan los datos entrados por el usuario.

La capa Vista es una sola, pero atendiendo a su distribución física se puede ver dividida en dos: del lado del cliente y del lado del servidor. Del lado del cliente la capa de la Vista realizará la interacción con el usuario mediante componentes de entrada y salida de información y componentes de control de procesos de interfaz de usuario. Estos componentes de control de UI serán escritos en Javascript y manipulados a través del framework ExtJS. Además realizará la lógica asociada a la validación de los datos proporcionados por el usuario.

Del lado del servidor en la capa Vista se encuentran las plantillas de la aplicación las cuales van a estar distribuidas de la siguiente forma: una plantilla indexSuccess por cada módulo y una plantilla global o Layout por cada app[13].

**Controlador**

En esta capa se encuentran las clases controladoras o clases Action de la aplicación las cuales se encargan de procesar las interacciones del usuario y realiza los cambios apropiados en el modelo o en la vista. En ella también se halla el Controlador Frontal, este es un componente que sólo tiene código relativo al MVC, por lo que no es necesario crear uno, ya que Symfony lo genera de forma automática.

**Modelo**

Esta capa se encuentra dividida en dos, la capa de Acceso a Datos la cual se encarga de la gestión de la información y la capa de Abstracción de Datos en la cual se hallan las clases entidades y el mapeo real de la base de datos. Dichas capas se generan automáticamente, en función de la estructura de datos de la aplicación. El ORM se encarga de crear el esqueleto o estructura básica de las clases y genera automáticamente todo el código necesario.

La abstracción de la base de datos es completamente transparente para el programador, se realiza de forma nativa mediante PDO (PHP Data Objects). Así, si se cambia el sistema gestor de bases de datos en cualquier momento, no se debe reescribir ni una línea de código, ya que tan sólo es necesario modificar un parámetro en un archivo de configuración.

**Patrones de diseño**

Los patrones de diseño son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces.

Un patrón de diseño resulta ser una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias (Patrones de Diseño, 2009).

[13]App: backend (parte de la aplicación dedicada a la administración) o frontend (parte de la aplicación dedicada a los usuarios).

**Patrones GOF**

Los patrones de diseño Pandilla de los Cuatro del inglés (Gang of Four) se clasifican en 3 grandes categorías basadas en su propósito: creacionales, estructurales y de comportamiento.

**Singleton**

En las acciones se usan los métodos ->getRequest(), ->getUser(), esto se debe a que, en la acción, el método getContext(), guarda una referencia a todos los objetos del núcleo de Symfony, estos métodos pueden ser accedidos desde la vista y desde el controlador, solo varía la forma de llamarlos (2008).

**Command**

Este patrón se evidencia cuando un objeto o sistema puede recibir varias peticiones o comandos. Reducir la responsabilidad del receptor en el manejo de los comandos, aumenta la facilidad con que pueden agregarse otros comandos y ofrece las bases para registrar los mismos, para formar colas de espera con ellos y para cancelarlos.

Este patrón se pone de manifiesto en el método dispatch() de la clase sfWebFrontController, que es la encargada de determinar cuál módulo y acción usar en dependencia de la petición del usuario.

**Decorador**

En este método de la clase abstracta sfView, padre de todas las vistas, tienen cada una un decorador para permitir añadir funcionalidades a las vistas dinámicamente. Este patrón se observa en el archivo denominado layout.php que contiene el Layout (plantilla global) de todas las páginas del registro. El mismo almacena el código HTML que es común a todas las páginas del registro, para no tener que repetirlo en cada página, por lo que el Layout decora la plantilla (Tutorial Jobbet, 2009). Ver ejemplo en Anexo #7.

**Patrones GRASP**

Los Patrones generales de software para asignar responsabilidades del inglés (Responsibility Assignment Software Patterns) describen los principios fundamentales de la asignación de responsabilidades a objetos, expresados en forma de patrones (Larman, 1999). Existe un gran número de patrones que se sitúan dentro de este grupo. A continuación se señalan.

**Experto**

Asigna una responsabilidad al experto en información: la clase que cuenta con la información necesaria para cumplir la responsabilidad.

Este es uno de los patrones que se implementa al utilizar las clases controladoras o clases Action de la aplicación para la gestión de formularios en el sistema. Ver Anexo #8.

**Alta cohesión**

Asigna una responsabilidad de modo que la cohesión siga siendo alta. La cohesión es una medida de cuan relacionadas y enfocadas están las responsabilidades de una clase. Una alta cohesión caracteriza a las clases con responsabilidades estrechamente relacionadas que no realicen un trabajo enorme (Larman, 1999).

Una clase con baja cohesión hace muchas cosas no afines o un trabajo excesivo. No conviene este tipo de clases pues presentan los siguientes problemas:

Son difíciles de comprender, reutilizar y conservar.

Las clases con baja cohesión a menudo representan un alto grado de abstracción o han asumido responsabilidades que deberían haber delegado a otros objetos.

Una de las características principales del framework Symfony es la organización del trabajo en el mismo en cuanto a la estructura del proyecto, lo cual permite crear y trabajar con clases con una alta cohesión. Por ejemplo, la clase *Actions* contiene varias funcionalidades estrechamente relacionadas entre ellas, teniendo un sentido común y un propósito único, siendo estas las encargadas de controlar las acciones de las plantillas. Esto hace posible que el software sea flexible a cambios sustanciales con efecto mínimo.

**Bajo acoplamiento**

Asigna una responsabilidad para mantener bajo acoplamiento. El acoplamiento es una medida de la fuerza con que una clase está conectada a otras clases, las conoce y/o recurre a ellas. Una clase con bajo (o débil) acoplamiento no depende de muchas otras.

Este patrón está evidenciado en el framework Symfony ya que dentro de la capa modelo las clases de abstracción de datos son las más reutilizables y no tienen asociaciones con las clases de la capa vista ni con el controlador. Ver ejemplo en Anexo #9.

**Controlador**

Es un patrón que sirve como intermediario entre una determinada interfaz y el algoritmo que la implementa, de tal forma que es la que recibe los datos del usuario y la que los envía a las distintas clases según el método llamado. Sugiere que la lógica de negocios debe estar separada de la capa de presentación, esto para aumentar la reutilización de código y a la vez tener un mayor control. Se recomienda dividir los eventos del sistema en el mayor número de controladores para poder aumentar la cohesión y disminuir el acoplamiento. Un ejemplo de ellos son las clases controladoras empleadas para la implementación del sistema.

**Tarjetas CRC (Cargo o Clase, Responsabilidad y Colaboración)**

Las tarjetas CRC permiten desprenderse del método basado en procedimientos y trabajar con una metodología basada en objetos, así el programador se centra y comienza a apreciar el desarrollo orientado a objetos olvidándose de los malos hábitos de la programación clásica. Las tarjetas CRC representan objetos; la clase a la que pertenece el objeto se escribe en la parte de arriba de la tarjeta, a modo de título, en una columna a la izquierda se escriben las responsabilidades u objetivos que debe cumplir el objeto y a la derecha, las clases que colaboran con cada responsabilidad (Noble, 2004).

A continuación se muestran algunas de las tarjetas CRC correspondientes a las clases que desde el punto de vista del negocio tienen gran significación para el sistema:

Tabla 5.Clase: DocumentoTable

| **Responsabilidad** | **Clases Relacionadas** |
|---|---|
| Procesar un documento dado su identificador | • Documento<br>• Introducción<br>• ElaboradoPor<br>• ControlCambio<br>• PaqueteAnálisis |
| Obtener un documento dado su identificador | • Documento |
| Obtener la versión de un documento dado su identificador | • Documento<br>• Control de Cambio |

Tabla 6.Clase: AutenticacionActions

| Responsabilidad | Clases Relacionadas |
|---|---|
| Acceder al sistema | • sfWebRequest |
| Salir del sistema | • sfWebRequest |

**Modelo de Datos**

La creación del modelo de datos constituye prioridad en la abstracción para la creación de la base de datos siendo posible esto a través de un grupo de herramientas las cuales describen los datos y las relaciones que existen entre ellos. Haciendo uso del diagrama entidad relación de la herramienta CASE Visual Paradigm para UML se realizó el siguiente modelo de datos:

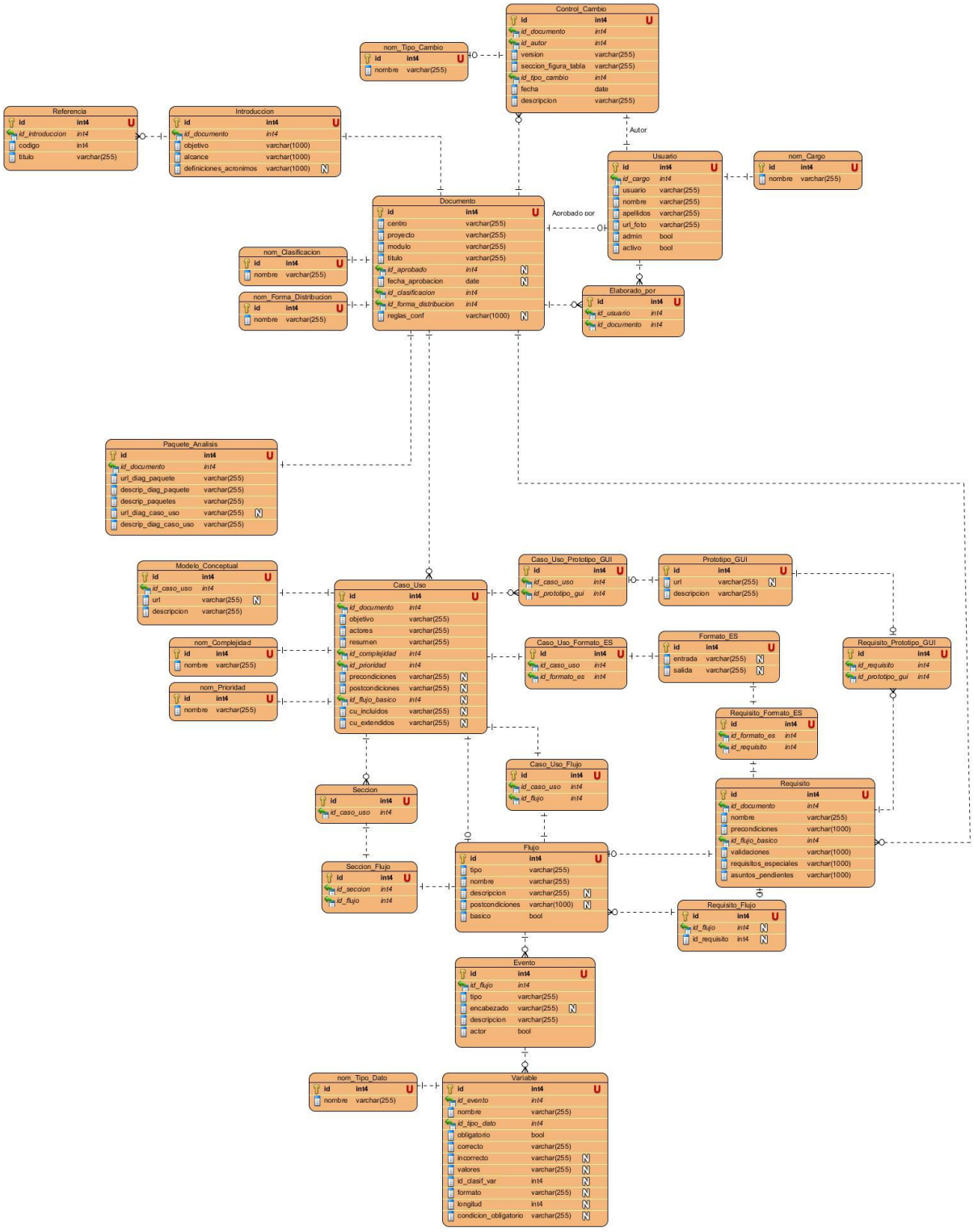



Figura 2. Modelo de la Base de Datos

El modelo de datos del sistema cuenta con 30 tablas o entidades necesarias para la gestión de los documentos de tipo especificación de casos de uso o descripción de requisitos. Dicho modelo describe las entidades Documento, Control_Cambio, Introducción, Paquete_Análisis como las entidades necesarias para la creación de un documento cualquiera sea su tipo.

Para la edición de la rama de casos de uso define las clases Caso_Uso, Modelo_Conceptual, Flujo, Sección, Caso_Uso_Prototipo_GUI, Caso_Uso_Formato_ES, para la edición de un documento de Requisitos se definen las clases Flujo, Requisito_Prototipo_GUI, Requisito_Formato_ES. La gestión de usuarios y relaciones de los mismos con la documentación necesita de las entidades Usuario y las relaciones Autor, Elaborado_por y Aprobado_por.

## 2.5. Desarrollo

La implementación del sistema es la parte más importante dentro del avance del proyecto en la metodología XP. Esta propone una serie de prácticas que sirven para el desarrollo exitoso del mismo, tales como el progreso de las iteraciones según el Plan de iteraciones, un diseño simple y poco redundante del código con las funcionalidades necesarias estrictamente en el presente y las pruebas continuas, donde los programadores escriben las pruebas por cada unidad de código.

La implementación de forma iterativa de XP es una ventaja ya que al finalizar cada una se obtiene una versión del sistema, superior a la anterior, donde se obtiene un producto funcional de la misma. En dicha funcionalidad se aplican pruebas al sistema las cuales pueden servir para retroalimentación del equipo de trabajo y cómo va encaminado el trabajo.

**Tareas de programación por historia de usuarios**

Las tareas de programación se definen con el objetivo de desglosar cada HU en tareas que serán desarrolladas por los programadores proporcionándoles una guía para un mejor desarrollo y cumplimiento de cada una de ellas.

Las tareas de programación son actividades que los programadores conocen que el sistema debe hacer. La mayoría de las tareas de programación se derivan directamente de las HU (Beck, 1999). Ver Anexo #4.

**Desarrollo en pareja**

En la metodología XP se propone un desarrollo en pareja. Esta estrategia o estilo de programación tiene muchas ventajas, entre las que se encuentra que los errores son detectados conforme son introducidos en el código; por lo que la cantidad de errores del sistema informático final suelen ser menos, los diseños son mejores y el tamaño del código menor, los problemas de la implementación se resuelven en menos tiempo, se posibilita la transferencia de conocimientos entre los miembros del equipo; pues cada uno entiende las diferentes partes del sistema y por último los programadores o desarrolladores disfrutan más su trabajo (Noble, 2004).

## 2.6. Conclusiones parciales

En este capítulo se abordó lo referente a la etapa de Planificación donde se definieron las HU, requisitos funcionales y no funcionales permitiendo así definir las propiedades y bases para el desarrollo del sistema a implementar. Se realizaron los artefactos Estimación de esfuerzo, Plan de iteraciones, Plan de duración de las iteraciones y Plan de entrega para una mayor organización en el trabajo de los programadores y la realización de un cronograma para una mayor planificación del tiempo de trabajo y compromiso con el cliente.

Se precisó la arquitectura y los patrones de diseño a utilizar para brindar un mayor entendimiento, organización y claridad para la implementación de la solución, lo cual permite que futuras mejoras en la programación sean menos engorrosas. Además se crean las tarjetas CRC para asignar responsabilidades a las clases y definir relaciones de dependencia entre las mismas.

Dentro de la etapa de desarrollo se describieron los estándares de codificación empleados en la propuesta de solución, para así llevar a cabo con éxito la propiedad colectiva del código, lo que asegura que todo el equipo se sienta cómodo con el código escrito por cualquiera de sus miembros. Además se puntualizaron las tareas de la programación para desglosar las HU lo cual permite a los programadores una guía para el desarrollo y cumplimiento de cada una de ellas.

# CAPÍTULO 3: VALIDACIÓN DE LA PROPUESTA DE SOLUCIÓN

## 3.1. Introducción

Uno de los pilares de la Programación Extrema es el proceso de pruebas. XP anima a probar constantemente tanto como sea posible. Esto permite aumentar la calidad de los sistemas reduciendo el número de errores no detectados y disminuyendo el tiempo transcurrido entre la aparición de un error y su detección. También permite aumentar la seguridad de evitar efectos colaterales no deseados a la hora de realizar modificaciones y refactorizaciones dividiendo dichas pruebas en dos grupos: pruebas unitarias, encargadas de verificar el código y diseñada por los programadores, y pruebas funcionales destinadas a evaluar si al final de una iteración se consiguió la funcionalidad requerida diseñadas por el cliente final (Noble, 2004).

En el capítulo se hace uso de diferentes métricas para validar tanto los requisitos establecidos por el cliente como el diseño planteado para el desarrollo del sistema informático. Para la verificación del sistema se realizan pruebas de caja blanca para comprobar que las salidas del código propuesto son las correctas y pruebas de caja negra para examinar las funcionalidades a nivel de interfaces en el sistema y se realiza una validación de las variables que se plantearon en el problema a resolver.

## 3.2. Métricas

Las métricas del software son una medida cuantitativa de evaluar la calidad de los atributos internos de un sistema. Se emplean con el objetivo de llevar el control de la calidad del producto que se está desarrollando, evaluar la efectividad del proceso y mejorar la calidad del trabajo. Las métricas proporcionan los conocimientos necesarios para crear modelos efectivos de análisis y diseño, un código sólido y pruebas exhaustivas (Roger S., 2005).

### 3.2.1. Métricas para validar requisitos

**Estabilidad de los requisitos:** “El objetivo de esta métrica es medir cuan estables son los requisitos para asegurar su adecuación antes de pasar a la próxima disciplina”.

Se considera que los requisitos son estables cuando no existen adiciones o supresiones en ellos que impliquen modificaciones en las funcionalidades principales de la aplicación. La estabilidad de los requisitos se calcula como: (Maite Sánchez Fornaris, 2010)

$$ETR = \left[ \frac{RT - RM}{RT} \right] * 100$$

Figura 3. Calcular estabilidad de los requisitos

**Donde**:

**RT:** Total de requisitos definidos.

**RM:** Cantidad de requisitos modificados.

Esta métrica ofrece valores entre 0 y 100. El mejor valor de ETR es el más cercano a 100 porque mostrará que no se están realizando cambios sobre los requisitos, son estables y, por tanto, es confiable trabajar el análisis y diseño sobre ellos.

**Especificidad de los requisitos:** El objetivo de esta métrica es cuantificar la especificidad o falta de ambigüedad en la definición de los requisitos. Para calcular esta métrica deben contarse los requisitos que tuvieron igual interpretación por los revisores y compararlos con el total de requisitos definidos. La especificidad de los requisitos se calcula como: (Maite Sánchez Fornaris, 2010)

$$Q_1 = \frac{n_{ui}}{n_r}$$

Figura 4. Calcular Especificidad de los requisitos

**Donde**:

**Q1:** Especificidad de los requisitos.

**Nr:** Total de requisitos definidos.

**Nui:** Total de requisitos para los que los revisores tuvieron interpretaciones idénticas.

El valor de esta métrica debe estar siempre entre 0 y 1. Mientras más cerca de 1 esté el valor de ER mayor será la consistencia de la interpretación de los revisores para cada requisito y menor será la ambigüedad en la especificación de los requisitos.

**Grado de validación de los requisitos:** Los requisitos deben ser posibles de validar. La validación de los requisitos se realiza en consenso del equipo de desarrollo al contrastar lo que desea el cliente con la posibilidad real de implementarlo. El grado de validación de los requisitos

mide la corrección en la definición de los requisitos. Este valor se calcula como: (Maite Sánchez Fornaris, 2010)

$$Q_3 = \frac{n_c}{(n_c + n_{nv})}$$

Figura 5. Calcular grado de validación de los requisitos

**Donde:**

**Q3:** Grado de validación de requisitos.

**Nc:** Total de requisitos validados correctamente.

**Nnv:** Total de requisitos no validados.

El resultado de esta métrica está siempre entre 0 y 1. El valor óptimo, es el más cercano a 1 e indica un alto nivel de corrección en la definición de los requisitos.

Al aplicar estas métricas a los requisitos del sistema arrojó como resultado que el 94.43% de los requisitos se comportó de manera estable o no sufrieron muchos cambios por lo que es confiable trabajar con ellos en una próxima fase. Después de aplicada la métrica especificidad de los requisitos y grado de validación el índice obtenido es de 0.94 lo cual muestra que la ambigüedad existente en los requisitos definidos es muy baja y la corrección de la definición en los mismos es alta.

### 3.2.2. Métricas para validar el diseño

**Tamaño Operacional de Clase (TOC)**

Consiste en medir el tamaño general de una clase tomando el valor de la cantidad de operaciones. Si el resultado obtenido indica valores grandes, significa que la clase posee un alto grado de responsabilidad, debido a esto se reducirá la reutilización, se hará mucho más difícil la implementación y la realización de pruebas de dicha clase. Por tanto mientras menor sea el valor para el TOC se hará mucho más fácil la reutilización de dicha clase dentro del sistema.

Tabla 7. Relación de números de procedimiento por clase

| Clase | Cantidad de Procedimientos |
|---|---|
| Backend::autenticacionActions | 2 |
| Backend::principalActions | 10 |

| | |
|---|---|
| Frontend::autenticacionActions | 3 |
| Frontend::principalActions | 6 |
| Frontend::editorActions | 21 |
| CasoUsoTable | 1 |
| CasousoFlujoTable | 1 |
| ControlCambioTable | 2 |
| DocumentoTable | 5 |
| ElaboradoPorTable | 2 |
| EspecificacioncuTable | 1 |
| EventoTable | 1 |
| FlujoTable | 1 |
| FormatoEsTable | 1 |
| IntroduccionTable | 2 |
| ModeloConceptualTable | 1 |
| NomCargoTable | 1 |
| NomClasificacionTable | 1 |
| NomComplejidadTable | 1 |
| NomFormaDistribucionTable | 1 |

**Resultado de la aplicación de la métrica Tamaño Operacional de Clase (TOC)**

La métrica TOC fue aplicada a una muestra de 20 clases escogidas aleatoriamente, la Figura 6 muestra el porciento de la cantidad de procedimientos presentes en las clases agrupadas en intervalos definidos, resultando 17 clases entre 1-5 procedimientos, 2 clases entre 6-10, 1 clase con más de 21 procedimientos.

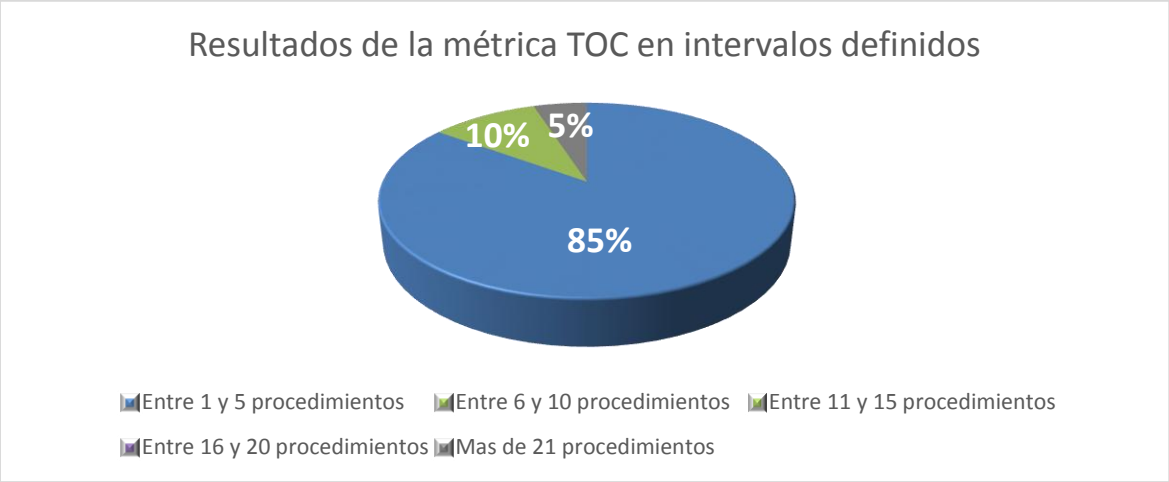



Figura 6. Representación en porciento de los resultados obtenidos, agrupados en intervalos definidos tras aplicar la métrica TOC.

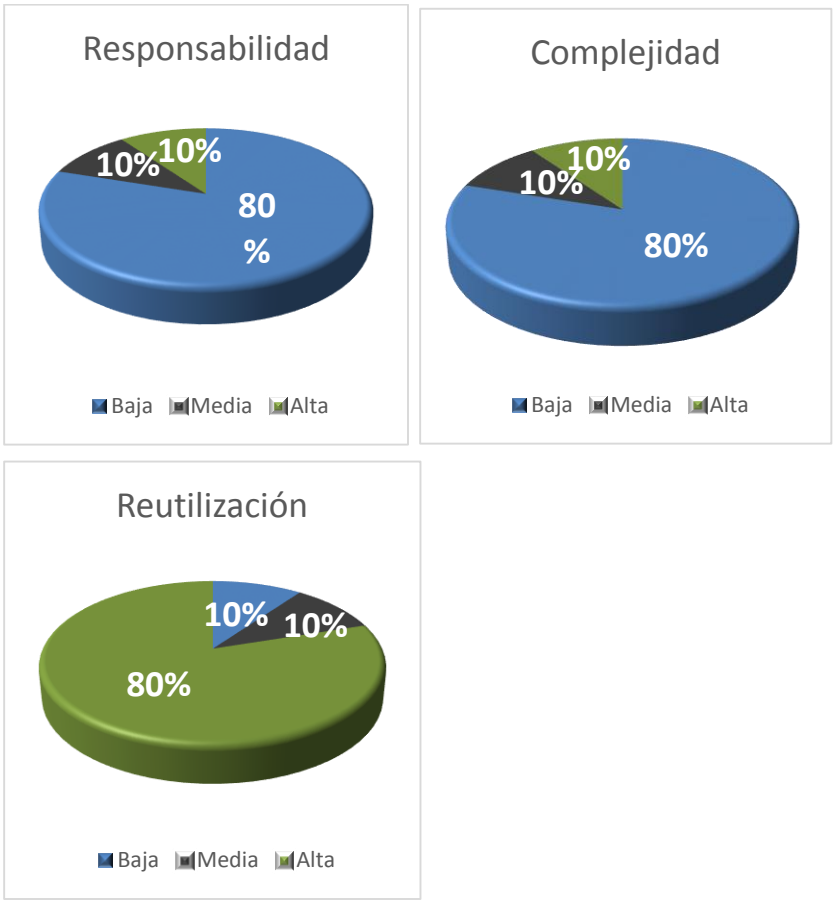



Figura 7. Representación de los resultados de la evaluación de la métrica TOC.

**Relaciones entre Clases (RC)**

Está dado por el número de relaciones de uso de una clase con otra y evalúa los siguientes atributos de calidad Acoplamiento, Complejidad de Mantenimiento, Reutilización y Cantidad de Pruebas:

Tabla 8. Relaciones de uso entre clases.

| Clase | Cantidad de Relaciones de Uso |
|---|---|
| autenticacionActions | 2 |
| principalActions | 2 |
| autenticacionActions | 2 |
| principalActions | 5 |
| editorActions | 7 |
| CasoUsoTable | 3 |
| CasousoFlujoTable | 2 |
| ControlCambioTable | 3 |
| DocumentoTable | 8 |
| ElaboradoPorTable | 3 |
| EspecificacioncuTable | 2 |

**Resultado de la aplicación de la métrica Relación entre Clases (RC)**

La métrica de diseño RC fue aplicado a una muestra de 11 clases seleccionadas aleatoriamente, la Figura 5 ilustra la cantidad de dependencias y el porciento que representan de la muestra seleccionada. Las figuras 6, 7, 8 y 9 muestran los resultados que arrojó la aplicación de la métrica en los atributos de calidad acoplamiento, complejidad de mantenimiento, reutilización y cantidad de pruebas.

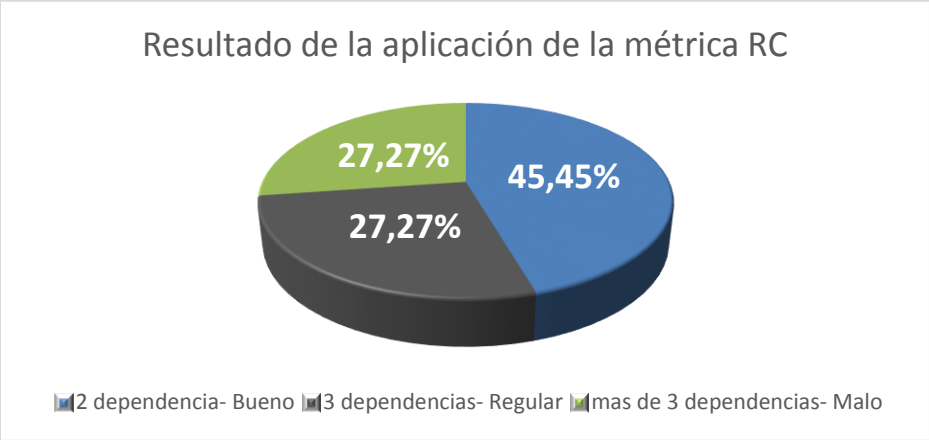



Figura 8. Representación en porciento de la aplicación de la métrica RC en intervalos definidos.

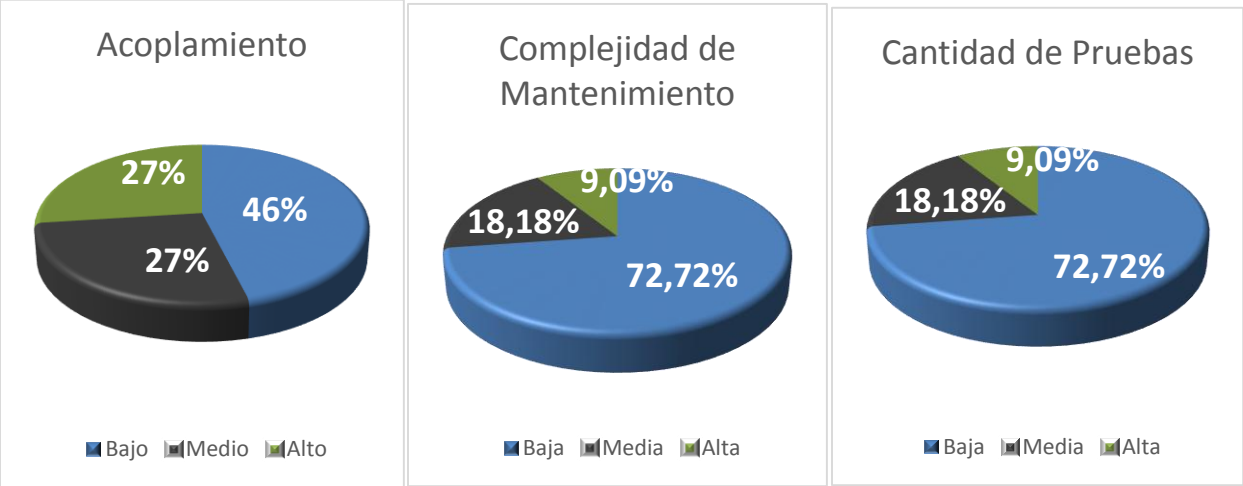



Figura 9. Representación de los resultados de la evaluación de la métrica RC.


Figura 10. Representación de los resultados de la evaluación de la métrica RC en el atributo Reutilización.

### 3.3. Verificación del Sistema

Para la verificación del sistema se realizaron pruebas de unidad con el objetivo de comprobar el correcto funcionamiento del software, a nivel de interfaz y de codificación mediante los métodos de prueba de caja negra y de caja blanca respectivamente.

**Pruebas de caja negra**

Para verificar las funcionalidades del sistema propuesto se realizaron pruebas funcionales por el método de caja negra mediante la técnica de particiones de equivalencia, a continuación un ejemplo que por su importancia en la seguridad del sistema se decide mostrar.

Caso de Prueba correspondiente a la HU Autenticar Usuario el cual permite acceder al sistema:

Tabla 9. Caso de Prueba Autenticar Usuario

| Escenario | Descripción | Nombre de | Contraseña | Respuesta del sistema | Flujo central |
|---|---|---|---|---|---|
| | | V | V | | |
| EC 1.1 Autenticar usuario correctamente. | Al insertar los datos requeridos el usuario se loguea correctamente | yjimeno | admin*9 | El sistema permite al usuario loguearse en el mismo. | Página principal del sistema |
| EC 1.2: Campos Incorrectos. | Cuando se insertan datos incorrectos en uno de los campos, el usuario no logra loguearse en el sistema. | I (No válido) | NA | El sistema muestra un mensaje de aviso, informando que el usuario o la contraseña introducida no son encontrados. | Página principal del sistema |
| | | NA | I (No válido) | El sistema muestra un mensaje de aviso, informando que el usuario o la contraseña introducida no son encontrados | Página principal del sistema |
| EC 1.3: Campos Incompletos. | Cuando quedan datos incorrectos en uno de los campos, el usuario no logra loguearse en el sistema. | I (Vacío) | NA | El sistema muestra un mensaje de aviso, informando que el usuario o la contraseña introducida no son encontrados. | Página principal del sistema |
| | | NA | I (Vacío) | El sistema muestra un mensaje de aviso, informando que el usuario o la contraseña introducida no son encontrados. | Página principal del sistema |

**Análisis de los resultados:**

Se confeccionaron 30 casos de prueba para la misma cantidad de HU existentes en el sistema informático lo cual arrojó como resultado la existencia de NC [14]en 12 de las HU implementadas por lo cual se hizo necesario una segunda iteración de pruebas a la aplicación para ver el estado de las no conformidades, encontrándose que el 100% de las deficiencias encontradas habían sido resueltas.

Tabla 10. Resultado de aplicar el método de caja negra

| Iteración | Satisfactorias | No Satisfactoria | Total |
|---|---|---|---|
| 1 | 18 | 12 | 30 |
| 2 | 30 | 0 | 30 |

**Pruebas de caja blanca**

Las pruebas de caja blanca se basan en el conocimiento de la lógica interna del código del sistema, contemplan los distintos caminos que se pueden generar teniendo en cuenta las estructuras condicionales o los distintos estados del mismo. Para poner en práctica este tipo de pruebas se procede a utilizar la técnica del camino básico.

**Técnica del camino básico**

La prueba del camino básico es una técnica de prueba unitaria que permite al equipo de desarrollo obtener una medida de la complejidad lógica del código implementado como resultado del sistema propuesto en la investigación; usando dicha medida como guía para la definición de un conjunto de caminos independientes[15]de ejecución, lo que garantiza que durante la prueba se ejecuten por lo menos una vez cada sentencia del programa (Pressman, 1997).

[14] NC: No Conformidades o errores encontrados en una revisión.

[15]Camino independiente se entiende aquel que introduce un nuevo conjunto de sentencias o una nueva condición. En términos del grafo, por una arista que no haya sido recorrida antes.

**Complejidad Ciclomática**

La complejidad ciclomática está basada en la teoría de grafos y da una métrica del software extremadamente útil, calculándose de tres formas:

- El número de regiones del grafo de flujo coincide con la complejidad ciclomática.
- La complejidad ciclomática, V (G), de un grafo de flujo G se define como V (G)=A-N+2, donde A es el número de aristas del grafo de flujo y N es el número de nodos del mismo.
- La complejidad ciclomática, V (G), de un grafo de flujo G también se define como V (G) P+ 1, donde P es el número de nodos predicado contenidos en el grafo de flujo G.

En la Figura 11 se puede apreciar el código correspondiente al método Eliminar perteneciente a la clase PrincipalAction, se escogió este método debido a la relevancia que posee en el sistema propuesto. A continuación se procede a realizar el cálculo de la complejidad ciclomática

- Confeccionar el grafo de flujo correspondiente.
- Calcular la complejidad ciclomática asociada.
- Extracción de los caminos independientes según el valor de la complejidad ciclomática.
- Realizar los casos de pruebas.
- Analizar los resultados obtenidos en las pruebas.

```
public function executeEliminar(sfWebRequest $request) {
    $entidad = $request->getParameter('entidad');//1
    $id = $request->getParameter('id');//1
    if ($entidad == 'Usuario') {//2
        $user = UsuarioTable::obtenerPorId($id);//3
        if ($user[0]['usuario'] == $this->getUser()->getAttribute('usuario')) {//4
            $this->renderText('No se puede eliminar el mismo usuario. Está logueado.');//5
            return sfView::NONE;//5
        }
    }
    $obj = Doctrine::getTable($entidad)->find($id);//6
    $formC = $entidad . 'Form';//6
    $form = new $formC($obj);//6
    $val = $form->getObject()->delete();//6
    if ($val) {//7
        $this->renderText('OK');//8
    } else {
        $this->renderText('No se pudo eliminar.');//9
    }
    return sfView::NONE;//10
}//11
```

Figura 11. Código fuente que genera la funcionalidad eliminar usuario

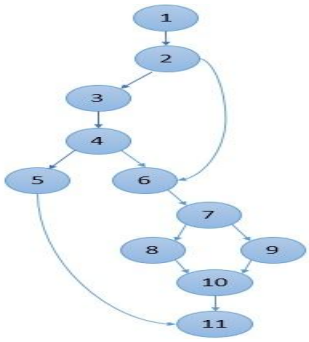



Figura 12. Grafo de flujo

**Cálculo de la Complejidad Ciclomática:**

El número de regiones del grafo de flujo coincide con la complejidad ciclomática.
El número de regiones del grafo del flujo definido es 4 lo que deviene que se obtenga el siguiente resultado:

V (G) = 4
V (G)=A-N+2=13-11+2=4
V (G)=P+1=3+1=4
Una vez calculada la complejidad ciclomática por cada una de sus variantes se define como límite superior cuatro pruebas a realizar para que todo el código se encuentre probado, no siendo complejo el código analizado por el equipo de desarrollo debido que el resultado arrojado por la métrica pertenece al intervalo entre 1-10 indicando qué el código perteneciente al sistema informático propuesto en la investigación es simple y sin mucho riesgo.

**Ta**bla 11. Caminos independientes establecidos

| Números | Caminos Básicos |
|---|---|
| 1 | 1-2-6-7-8-10-11 |
| 2 | 1-2-3-4-5-11 |
| 3 | 1-2-3-4-6-7-8-10-11 |
| 4 | 1-2-3-4-6-7-9-10-11 |

Cada camino es un caso de prueba, de forma tal que los datos introducidos provoquen que se visiten las sentencias vinculadas a cada nodo del camino como se refleja a continuación.

**Caso de Prueba para el camino básico #1**

Entrada: $entidad = `CasoUso´, $id = 12

Condiciones de ejecución: $val = true

Resultados esperados: El sistema elimina el caso de uso seleccionado.

Resultados obtenidos: Satisfactorio

**Caso de Prueba para el camino básico #2**

Entrada: $entidad = `Usuario´, $id = `12´

Condiciones de ejecución:

$user[0]['usuario'] = ¨rmromero¨

$this->getUser()->getAttribute('usuario') = ¨rmromero¨

Resultados esperados: El sistema muestra un mensaje. No se puede eliminar el mismo usuario. Está logueado.

Resultados obtenidos: Satisfactorio

**Caso de Prueba para el camino básico #3**

Entrada: $entidad = `Usuario´, $id = `12´

Condiciones de ejecución:

$user[0]['usuario'] = ¨rmromero¨

$this->getUser()->getAttribute('usuario') = ¨yjimeno¨

$val = true

Resultados esperados: El sistema elimina el usuario seleccionado.

Resultados obtenidos: Satisfactorio

**Caso de Prueba para el camino básico #4**

Entrada: $entidad = `Usuario´, $id = `12´

Condiciones de ejecución:

$user[0]['usuario'] = ¨rmromero¨

$this->getUser()->getAttribute('usuario') = ¨yjimeno¨

$val = false

Resultados esperados: El sistema muestra el mensaje. No se pudo eliminar.

Resultados obtenidos: Satisfactorio

**Análisis de los resultados**

Para validar qué el código implementado, se realizaron 20 pruebas a las funcionalidades del sistema informático propuesto como parte de la investigación. De las pruebas unitarias realizadas 4 estuvieron vinculadas a la funcionalidad Eliminar descrita con anterioridad, donde de los 20 casos de pruebas realizados 18 resultaron satisfactorios representando un 90%; verificando la estabilidad de la lógica aplicada en el código probado, perteneciente al sistema informático propuesto en la investigación, mientras que las 2 restantes resultaron fallidas representando un 10%. En una segunda iteración de pruebas al sistema de un total de 20 pruebas se obtuvo como resultado 20 pruebas satisfactorias representando un 100% y 0 pruebas fallidas lo que significa un 0% quedando probado más del 80% del código del sistema informático propuesto en la investigación.

A continuación se muestran los resultados obtenidos para el desarrollo de las iteraciones correspondientes a las pruebas unitarias:

Tabla 12. Iteraciones de pruebas de caja blanca

| Iteración | Satisfactoria | Insatisfactoria | Total |
|---|---|---|---|
| 1 | 18 | 2 | 20 |
| 2 | 20 | 0 | 20 |

**Pruebas de Aceptación**

Las pruebas de aceptación son destinadas a evaluar si al final de una iteración se consiguió la funcionalidad requerida por el cliente final. Estas pruebas aseguran el comportamiento del sistema y especifican los aspectos a probar cuando una historia de usuario ha sido correctamente implementada.

A continuación se muestra el caso de prueba de aceptación para una Historia de Usuario de prioridad alta en el negocio, el resto se puede encontrar en el expediente de proyecto. El Caso de Prueba correspondiente a la HU Gestionar Usuario el cual permite al administrador controlar el flujo y permisos de los usuarios en la aplicación:

Tabla 13. Caso de prueba de aceptación Gestionar usuario

| **Caso de prueba de aceptación** | |
|---|---|
| **Código:** HU1_P1 | **Historia de Usuario:** 1 |
| **Nombre:** Gestionar Usuario. | |
| **Descripción:** El administrador debe ser capaz de adicionar, editar o eliminar un usuario del sistema. | |
| **Condiciones de Ejecución:**<br>El usuario debe tener permisos de administración para realizar esta operación.<br>El administrador debe conocer el usuario UCI de la persona que desea gestionar en el sistema. | |
| **Resultados esperados:** El sistema adiciona, edita o elimina el usuario que se desee. | |
| **Evaluación de la prueba:** Prueba satisfactoria. | |

Para validar que la salida emitida por el sistema informático concordara con el resultado esperado por el cliente se diseñaron 11 pruebas funcionales en conjunto cliente-desarrolladores de las cuales 5 salidas coincidieron con los resultados esperados representando un 45%. Fue notable el poco tiempo empleado por el sistema en la devolución del resultado, por otra parte 6 pruebas resultaron fallidas representando un 55%, mientras que en una segunda iteración para las pruebas se arrojó resultados satisfactorios pues para el desarrollo de 11 pruebas funcionales se obtuvieron un total de 11 pruebas exitosas para un 100% de pruebas satisfactorias, mientras que 0 pruebas resultaron fallidas representando un 0% de las pruebas realizadas.

A continuación se muestra cómo quedan reflejados los resultados por cada una de las iteraciones de pruebas funcionales realizadas al sistema:

Tabla 14. Iteraciones de Pruebas Funcionales

| Iteración | Satisfactoria | Insatisfactoria | Total |
|---|---|---|---|
| 1 | 5 | 6 | 11 |
| 2 | 8 | 3 | 11 |

## 3.4. Validación de las variables

Con el desarrollo del sistema GenDCP, se facilita el Diseño de los Casos de Pruebas en los proyectos del Centro CEGEL ya que disminuye el tiempo que se tarda en realizar esta actividad, además de disminuir las No Conformidades (NC) que se detectan en los mismos, esto se evidencia a través de una prueba piloto realizada donde se tomó como referencia los Diseños Casos de Pruebas del módulo del subsistema Penal del proyecto TPC ya revisados por el Grupo de Calidad de la Facultad con sus respectivas NC y los Diseños de Casos de Pruebas de ese mismo módulo generados con la aplicación arrojando los siguientes resultados:

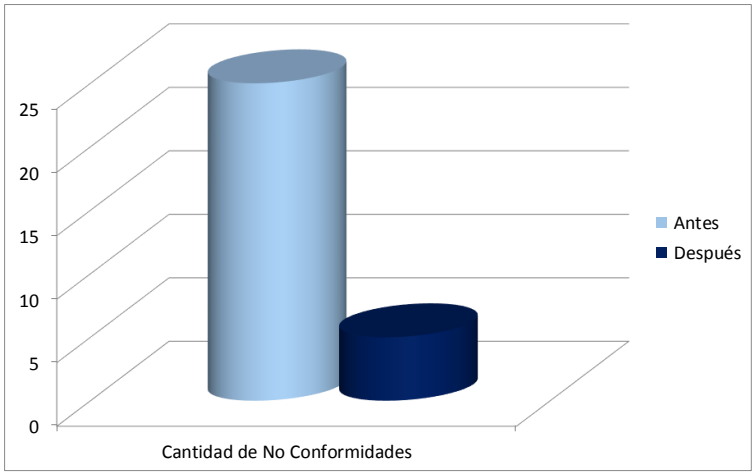



Figura 13. Cantidad de NC

La gráfica anterior muestra la cantidad de NC detectadas en los Diseños de Casos de Prueba, el color azul claro muestra la cantidad de NC detectadas antes de ser desarrollada la aplicación y el color azul las NC detectadas luego de generar los Casos de Pruebas con la aplicación, llegando a la conclusión de que se redujo considerablemente el número de NC detectadas.

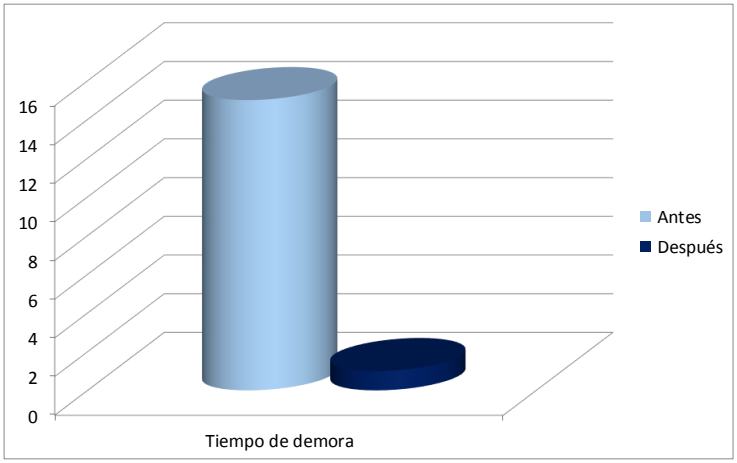



Figura 14. Grafo de flujo

La gráfica anterior muestra el tiempo de demora en elaborar los Diseños de Casos de Prueba, el color azul claro muestra el tiempo de demora antes de ser desarrollada la aplicación y el color azul el tiempo de demora luego de generar los Casos de Pruebas con la aplicación, llegando a la conclusión de que se redujo considerablemente el tiempo de demora.

## 3.5. Conclusiones parciales

A lo largo del capítulo se analizaron y construyeron los elementos imprescindibles que forman parte de la validación concluyendo lo siguiente:

- En los valores obtenidos en las 2 iteraciones de pruebas funcionales se comprobó que las funcionalidades diseñadas por el cliente se encontraban correctas.
- Por otra parte al aplicar la métrica TOC se obtuvieron valores medios para los atributos de calidad responsabilidad, complejidad y reutilización.
- La aplicación de la métrica Relaciones entre Clases mostró un bajo acoplamiento entre las clases debido a la baja dependencia que existe entre las mismas además de revelar que la complejidad para realizar mantenimiento al código de la aplicación es baja y que la cantidad de pruebas a realizar para que el diseño de las clases sea óptimo será baja.
- Por último, se validaron las variables que forman parte del problema de la investigación, demostrando que con el sistema desarrollado se cumplen los objetivos planteados.

# CONCLUSIONES GENERALES

Una vez concluido el desarrollo del presente trabajo se puede llegar a las siguientes conclusiones:

- El estudio del estado del arte de la temática investigada permitió seleccionar las herramientas, tecnologías y la metodología adecuadas para guiar el ciclo de vida del producto.
- El uso de la metodología XP para guiar el proceso y los artefactos generados en cada una de sus fases propició una mayor comunicación entre el equipo de desarrollo y el cliente, logrando obtener un mejor funcionamiento del sistema y que éste responda a todas las necesidades del cliente.
- Se desarrolló una aplicación web que permite generar Diseños de Casos de Pruebas facilitando el trabajo a los analistas de los proyectos del Centro.
- Con el desarrollo del sistema se mejora el Diseño de Casos de Prueba en los proyectos del Centro en cuanto a tiempo de demora en realizar esta actividad y la disminución de la cantidad de NC detectadas en los mismos.
- El empleo de métricas y pruebas de calidad aplicadas al sistema arrojaron resultados satisfactorios, demostrando que fue diseñado e implementado correctamente.

# RECOMENDACIONES

- Integrar al proceso de pruebas de las aplicaciones de la universidad el sistema desarrollado.
- Se propone la realización de pruebas funcionales mediante la herramienta Selenium apoyada en el sistema desarrollado en este trabajo de diploma.
- Implementar funcionalidades que contribuyan a la mejora de la aplicación.

# BIBLIOGRAFÍA

*Actas de Talleres de Ingeniería del Software y Bases de Datos, Vol. 1.* **Edumilis, Méndez, María, Pérez y Luis E., Mendoza. 2007.** Caracas, Venezuela : s.n., 2007. Aplicación de un Método para Especificar Casos de Prueba de Software en la Administración Publica. págs. 47-48.

**Alvarez, Miquel Angel. 2013.** *Manual de Css 3.* 2013.

**Angela, Martín Noble y James, Robert Biddle. 2004.** *The XP Customer Role in Practice: Three of Studies Agiles.* 2004.

*Automatización y Gestión de las Pruebas Funcionales.* **Ignacio Esmite, Mauricio Farías, Nicolás Farías, Beatriz Pérez. 2007.** 2007, Centro de Ensayos de Software (CES), Universidad de la República, págs. 9-10.

**Beck, K. 1999.** *Extreme Programming Explained.* s.l. : Embrace Change. s.l., 1999.

**Bikha, Avinash. 2008.** *Requirements Management-Defining the Project Scope and Developing Use Cases.* 2008.

**Camacho, Arlene Lugo. 2013.** *Herramientas Automatizadas para Pruebas Funcionales.* 2013.

**Castillo Cantón, Alejandro. 2011.** *Manual de HTML5 en español.* 2011.

*Control de Calidad en los Sistemas.* **Informática, Departamento de Control de la Calidad y Auditoría. junio 2000.** junio 2000.

**Coral, Calero Muñoz y Mario Gerardo, Piattini Velthuis, María Ángeles, Moraga de la Rubia. 2010.** *Calidad del Producto y Proceso software.* s.l. : Ra-Ma, 2010.

**Deporte, Observatorio Tecnológico-Ministerio de Educación. Cultura y. 2008.** INTEF. *Instituto Nacional de Tecnologías educativas y Formación del Profesorado.* [En línea] 4 de 2008. www.recursostic.educacion.es.

**Doctrine. 2007.** Doctrine. *Doctrine.* [En línea] 2007. http://www.doctrine-project.org/.

**2010.** ecured. *ecured.* [En línea] 14 de diciembre de 2010. http://www.ecured.cu/index.php/Herramientas_CASE.

**ExtJS. 2008.** EXT JS. *EXT JS.* [En línea] 2008. www.extjs.com.

**González, Espinosa Susana. 2008.** *Estrategia para la aplicación de Pruebas de Caja Blanca y Caja Negra al proyecto Registros y Notarías.* Cuba, Ciudad de la Habana, Boyeros : s.n., 2008.

**Herrara, Érica y Uribe. 2009.** *Servidor de Aplicaciones.* 2009.

**ISO. 1994.** *Gestión de Calidad y garantía de calidad.Vocabulario.* 1994.

**Ivar Jacobson, Grady Booch, James Tumbaugh. 2004.** *El Proceso Unificado de Desarrollo de Software.* s.l. : Addison- Wesley, 2004.

**Javascript, Introducción a. 2013.** librosweb. *librosweb.* [En línea] 2013. www.librosweb.es.

**Larman, Craig. 1999.** Introducción al análisis y diseño orientado a objetos. *UML y Patrones.* 1999, págs. 189-424.

**Larman, Craig. 2004.** *UML y Patrones.* 2004. págs. 3-4.

**Lopéz, C. 2002.** *Calidad.* 2002.

**Maite Sánchez Fornaris, Dayanis Alcantara Rabí. 2010.** *Propuesta de una guía de métricas para evaluar el desarrollo de los Sistemas de Información Geográfica.* 2010.

**Myers, G. 2004.** *The Art of Software Testing.* s.l. : John Wiley & Sons Inc., 2004.

**Neuland Agüero, Dennis. 2008.** *Áreas del aseguramiento de la calidad.* La Habana : La Habana: s.n, 2008.

**Noble, Angela Martin y de Robert Biddle, James. 2004.** *The XP Customer Role in Practice: Three Studies Agile.* 2004.

**Paradigm, Visual. 1995.** Visual Paradigm. *Visual Paradigm.* [En línea] 1995. www.visual-paradigm.com.

*Patrones de Diseño.* **Universidad Autónoma de Madrid. 2009.** 2009, págs. 5-10.

**PHP. 2001.** php. *php.* [En línea] 2001. www.php.net.

**PostgreSQL-es. 2012.** PostgreSQL-es. *PostgreSQL-es.* [En línea] 2012. http://www.postgresql.org.es/.

**Potencier, Fabien. 2008.** Synfony, la guia definitiva. *Synfony, la guia definitiva.* [En línea] 2008. [Citado el: 2013 de marzo de 6.] www.librosweb.es.

**Pressman, R. S. 1998.** *Ingeniería de software.Un enfoque práctico.Tercera Edición.* 1998.

**Pressman, Roger. 2002.** *Ingeniería de software.Un enfoque práctico.Tercera Edición.* 2002.

**Pressman, Roger S. 1997.** *Ingenieria de Software.* Madrid : s.n., 1997.

**Roger S., Pressman. 2005.** *Ingeniería del Software Un enfoque práctico.* s.l. : Sexta edición, 2005.

**Sedna, P. 2006.** *La Revolución Industrial.* 2006.

**SeleniumHQ. 2013.** [En línea] 10 de febrero de 2013. www.docs.seleniumhq.org/.

**Sklar, David. 2004.** *Learning PHP 5.* 2004.

**Sonia, Jaramilla Valbuena, Sergio Augusto, Cardona Torres y Leonardo Alonso, Hernández Rodríguez. 2010.** *Programación Orientada a Objetos.* Armenia, Quindío : Elizcom, 2010.

**Susana, González Espinosa. 2008.** *Estrategia para la aplicación de Pruebas de Caja Blanca y Caja Negra al proyecto Registros y Notarías.* Cuba, Ciudad de la Habana, Boyeros : s.n., 2008.

*Synfony, la guia definitiva.* **Fabien Potencier, François Zaninotto. 2008.** 2008, págs. 16-17.

*Tutorial Jobbet.* **Potencier, Fabien. 2009.** 2009, Practical Symfony, págs. 45-46.