RESUMEN – TRABAJO DE GRADO

AUTORES	**ANGIE LORENA BALLESTEROS CORONEL**
FACULTAD	**INGENIERÍAS**
PLAN DE ESTUDIOS	**INGENIERÍA DE SISTEMAS**
DIRECTOR	**BYRON CUESTA QUINTERO**
TÍTULO DE LA TESIS	**DISEÑO DE UN MODELO DE PRUEBAS BASADO EN LA METODOLOGÍA TEST DRIVEN DEVELOPMENT (TDD) PARA MEJORAR LAS PRACTICAS EN EL DESARROLLO DE LOS SISTEMAS DE INFORMACIÓN DE LA UNIVERSIDAD FRANCISCO DE PAULA SANTANDER OCAÑA.**

RESUMEN

(70 palabras aproximadamente)

ESTE DOCUMENTO PRESENTA UN MODELO DE PRUEBAS CUYO PROPÓSITO ES SERVIR DE GUÍA PARA MEJORAR LAS PRÁCTICAS DE DESARROLLO DE SOFTWARE EN LA DIVISIÓN DE SISTEMAS. ESTE MODELO CONSTA DE CUATRO FASES: PLANIFICACIÓN, DISEÑO, EJECUCIÓN Y EVALUACIÓN TENIENDO EN CUENTA LAS BUENAS PRÁCTICAS Y RECOMENDACIONES EN CUANTO A LAS PRUEBAS DE SOFTWARE ADEMÁS DE LA INTEGRACIÓN DE TEST DRIVEN DEVELOPMENT EN EL CICLO DE DESARROLLO *SCRUM*.

CARACTERÍSTICAS

PÁGINAS: 122	PLANOS:	ILUSTRACIONES: 31	CD-ROM: 1

DISEÑO DE UN MODELO DE PRUEBAS BASADO EN LA METODOLOGÍA *TEST DRIVEN DEVELOPMENT* (TDD) PARA MEJORAR LAS PRACTICAS EN EL DESARROLLO DE LOS SISTEMAS DE INFORMACIÓN DE LA UNIVERSIDAD FRANCISCO DE PAULA SANTANDER OCAÑA.

ANGIE LORENA BALLESTEROS CORONEL

Director

Ing. Byron Cuesta Quintero

Msc. Software libre

UNIVERSIDAD FRANCISCO DE PAULA SANTANDER OCAÑA

FACULTAD DE INGENIERÍAS

INGENIERÍA DE SISTEMAS

Ocaña, Colombia. **Agosto, 2018**

Tabla de Contenidos

Lista de tablas

Lista de Figuras

Capítulo 1 Diseño de un modelo de pruebas basado en la metodología *Test Driven Development* (TDD) para mejorar las prácticas en el desarrollo de los Sistemas de Información de la Universidad Francisco de Paula Santander Ocaña.

1.1 Planteamiento del problema.

Las empresas de desarrollo de software deben ajustar sus estándares de calidad además de reducir los costos en sus operaciones (Taha, 2006). De esta manera se plantean dos grandes desafíos: garantizar la calidad de los productos de software y mejorar su productividad durante el desarrollo. Retos enormes porque los cambios en el software introducen muchos errores, ocasionan que funcionalidades implementadas fallen en la etapa de mantenimiento debido a que las pruebas son realizadas al final del desarrollo del software donde es más difícil identificar los errores. (Fowler, 1999).

La Universidad Francisco de Paula Santander Ocaña (UFPSO) con el apoyo de la División de Sistemas, en aras de ofrecer un mejor servicio que le permita a través de los sistemas de información propios gestionar la información institucional de acuerdo a las necesidades actuales del mercado, ha definido una política en la cual se hace necesario implementar una nueva versión del software que incluye por un lado, la definición de una fase de migración para reemplazar las formas y reportes diseñados con *Developer Oracle* y por el otro implementar un desarrollo orientado a la web bajo el paradigma orientado a

objetos en *PHP* utilizando el patrón arquitectura Modelo Vista Controlador (MVC), donde todos los desarrollos son sometidos a una serie de pruebas que permitan identificar malos funcionamientos o errores (*bugs*). Entre las pruebas utilizadas se encuentran:

a) de caja blanca. "se centra en analizar el código fuente y la lógica interna del software" (Jenkins, 2008), su objetivo principal es probar la lógica del programa desde el punto de vista algorítmico ya que se analiza cada módulo individualmente, entre los diferentes casos de pruebas que se diseñan están:

- Cobertura de caminos (pruebas que hagan que se recorran todos los posibles caminos de ejecución)
- De Cobertura de Sentencias: Comprueba que todas las sentencias se ejecuten al menos una vez.
- De Cobertura de Condición: Ejecuta un caso de prueba a True y otro a False por cada condición, teniendo en cuenta que una decisión puede estar formada por varias condiciones.
- De Cobertura de Condición/Decisión: Se realizan las pruebas de cobertura de condición y las de decisión a la vez. (Tuya, 2007).

b) De caja negra siendo una técnica de pruebas de software en la cual la funcionalidad se verifica sin tomar en cuenta la estructura interna de código, detalles de implementación o escenarios de ejecución internos en el software (Pressman, 2010); "se centra en comparar las entradas con las salidas esperadas, sin detenerse en detalles de su desarrollo (código fuente)" (Jenkins, 2008). Se aplica realizando una prueba piloto con

los usuarios del sistema, revisando si las entradas son aceptadas de forma correcta, las salidas son las esperadas y mantenga la integridad de la información.

Los desarrolladores realizan pruebas evaluando el comportamiento del software pero se hace necesario implementar una guía formal de modelo de referencia o metodología que permita realizar un barrido sobre el código con el fin de minimizar la cantidad de errores que se presentan en el desarrollo de software y que afectan a la calidad del producto final además de la creación de casos de pruebas automatizadas para hacer seguimiento.

La adopción del modelo permitirá definir la trazabilidad a través de una práctica de control que ayudará a obtener el producto en el dominio de la solución lo más exacto y fiable posible a las necesidades expresadas por el cliente en el dominio del problema y la implementación de unidades de código que garanticen la calidad del mismo, evitando el riesgo de producir software que reporte continuos errores o fallas graves. (Gotel, 1994)

1.2 Formulación del problema.

¿Las prácticas de desarrollo de software se pueden fortalecer a través del diseño de un modelo de pruebas aplicado a los sistemas de información institucional?

1.3 Objetivos

1.3.1 Objetivo General. Diseñar de un modelo de pruebas basado en la metodología *Test Driven Development* (TDD) para mejorar las prácticas en el desarrollo de los Sistemas de Información de la Universidad Francisco de Paula Santander Ocaña.

1.3.2 Objetivo Especifico

- Revisar la literatura sobre las metodologías aplicadas a los procesos de pruebas de desarrollo ágil eligiendo la adecuada para el modelo.
- Construir la suite de casos de prueba para la elaboración del modelo.
- Implementar el modelo propuesto a la estandarización del proceso de pruebas de la División de Sistemas tomando como caso de estudio el Sistema de Información Académica (SIA).

1.4 Justificación

Las pruebas de software pueden definirse como una actividad donde un sistema o una parte de un sistema son ejecutado de forma controlada, es decir, bajo unas condiciones específicas y podemos observar su comportamiento. (Roger, 2002). Es necesario realizar pruebas unitarias, en pequeñas unidades de código, para probar el

funcionamiento de un módulo asegurando que cada uno actúe correctamente por separado y facilita localizar la causa de los errores. (Müller, 2003)

Las pruebas son el único instrumento que puede determinar la calidad de un producto software; asegura que un sistema software cumple con los requerimientos (Fontela, 2013). Para ello se propone utilizar los conceptos propuestos en la metodología TDD (*Test Driven Development*). TDD es una técnica de diseño e implementación de software incluida dentro de la metodología XP. Jurado (2010) define tres pilares fundamentales hallados en la metodología creada por Kent Beck,

- La implementación de las funciones justas que el cliente necesita y no más.
- La minimización del número de defectos que llegan al software en fase de producción.
- La producción de software modular, altamente reutilizable y preparado para el cambio.

La implementación de TDD aumenta la calidad del software desarrollado porque es construida desde el comienzo, no puede ser añadido en fases posteriores; puede referirse a cuán mantenible es el software, su estabilidad, velocidad, usabilidad, seguridad y número de fallas. (Beck, 2002).

El presente proyecto pretende realizar una modelo de pruebas para su adaptación en los sistemas de información institucional, enfocándose, en probar en pequeño (pruebas unitarias), para verificar el comportamiento de una sola unidad de código en un ambiente de trabajo paralelo al de producción.

Adicionalmente se debe contar con herramientas que permitan la automatización de las pruebas. Beck advierte al respecto que "una porción de código sin un programa que le pruebe automáticamente simplemente no existe" (Beck K. , 2002) En este proyecto se utilizará el *framework open source PHPUnit* orientado a pruebas para cualquier código *PHP*.

A la hora de realizar pruebas a los sistemas esta metodología proporcionará al equipo de desarrollo, procedimientos y herramientas, con la finalidad de estandarizar, aumentar la trazabilidad y mejorar el resultado de las pruebas, y por tanto la calidad del software, evitando correcciones en la fase de desarrollo, disminuyendo errores y evitando la pérdida de costos y tiempo para lograr un código limpio, altamente reutilizable.

Además, las pruebas realizadas se convierten en documentación, porque leer las pruebas ayuda a entender el software, lo que debe hacer el código.

1.5 Delimitaciones

1.5.1 Geográfica. Se desarrollará en la Universidad Francisco de Paula Santander Ocaña, en la División De Sistemas, dependencia que da soporte técnico y desarrolla los sistemas de información.

1.5.2 Temporal. El proyecto actual será realizado en un periodo de 6 meses a partir de la aprobación del mismo.

1.5.3 Conceptual. Se mencionarán conceptos *Test Driven Development* (TDD), *PHPUnit*, desarrollo ágil, Modelo Vista Controlador (MVC), *SCRUM*, software, ingeniería de software.

1.5.4 Operativa. El desarrollo del proyecto se realizará en la Universidad Francisco de Paula Santander Ocaña en la dependencia de División de sistemas.

Capítulo 2 Marcos de Referencia

2.1 Marco Histórico

2.1.1 Antecedentes históricos nacionales. Se han encontrado varios proyectos que documentan la metodología *Test-Driven Development (TDD)*.

Vaca (2014), explica las dificultades que se encuentran en la implementación de TDD dentro de los equipos de software en cuanto a incrementos de costos y tiempos de desarrollos en algunos estudios y se desprende del mismo que es una metodología muy utilizada en proyectos con ciclos de vida ágiles.

Villamizar (2015), buscando enriquecer la técnica TDD propone las historias de usuario con base en un formato único y completo que integre elementos de los casos de prueba mediante nuevas técnicas para la mejora de las pruebas funcionales integrales a través de la adición de pruebas de aceptación y la gestión de las historias de usuario mediante su con el lenguaje XML.

Moncada (2014), plantea obtener múltiples fuentes de información acerca del proceso de inclusión de pruebas unitarias en ambientes ágiles. El estudio se desarrolla por medio de una revisión sistemática de literatura, basándose en buenas prácticas referentes

al estudio de fuentes bibliográficas confiables, recogidas en múltiples dominios de actuación, se consolida la información disponible sobre la implementación de pruebas unitarias en entornos ágiles de desarrollo, encontrando una fuerte tendencia de adopción en lo ágil y una serie de técnicas especializadas para que el uso de pruebas unitarias en estos ambientes convirtiéndose en una estrategia de aseguramiento de la calidad en entornos con requerimientos incompletos o altamente cambiantes.

2.1.2 Antecedentes históricos internacionales. Un caso de estudio de TDD en la industria es el de Laurie Williams en *IBM* (Williams, 2003). Al aplicar TDD a un proyecto existente que iba a ser migrado a otra plataforma, encontraron que las pruebas escritas al desarrollar con TDD eran capaces de encontrar más errores que las pruebas escritas para el proyecto antes de la migración. La cantidad de defectos encontrados por línea de código se redujeron en un 40%.

Otro caso de estudio conocido es el de *Microsoft* llevado a cabo por Bhat y Nagappan en donde tomaron secciones de *Windows* y de *MSN* y los compararon con proyectos similares (Bhat, 2006). En el primer caso (*Windows*), encontraron que el proyecto sin TDD tenía 2.6 más defectos por 1000 líneas de código que el proyecto desarrollado con TDD. El tiempo total para el desarrollo fue entre 25 y 30% más en el caso del proyecto desarrollado con TDD. En el segundo caso (*MSN*) los resultados son

similares: el proyecto desarrollado sin TDD contenía 4.5 más defectos, pero el desarrollo con TDD tomó 15% más de tiempo.

Dieste (2015) afirma que los métodos ágiles y sus prácticas asociadas con TDD, son ampliamente utilizadas en la industria y han sido sometidas repetidamente a estudios empíricos, donde se han realizado diversos experimentos en empresas y academia acerca de TDD. En general, los experimentos no muestran un efecto positivo de TDD en la calidad del código o la productividad de los programadores, por lo tanto, este trabajo busca replicar el experimento UPM 2014 efectuado por N. Juristo y su equipo, para reproducir sus resultados y secundariamente, estudiar el efecto de la experiencia del desarrollador en la efectividad de TDD.

Goicochea (2015), nos dice que este trabajo de investigación brinda un enfoque general de la aplicación de la técnica *Test Driven Development* (TDD), o Desarrollo Guiado por Pruebas, dentro de la metodología tradicional con enfoque Cascada, y cómo su implicancia proporciona resultados favorables durante el proceso de implementación y en consecuencia la mejora de la calidad del producto. El experimento consistió en que dos (2) grupos apliquen la técnica de TDD dentro de la metodología Cascada en las primeras fases del modelo Cascada (Definición de requerimientos y Diseño del sistema) y los otros (2) grupos no la apliquen.

Aucancela (2011) afirma que este trabajo está orientado a la aplicación práctica de TDD en el desarrollo de un sistema web para reservar los laboratorios computacionales de la ESPE, siguiendo los lineamientos de la metodología de desarrollo de software AUP, la cual se basa en cortas iteraciones, desde el levantamiento de requisitos, análisis, diseño e implementación y en cuanto al proceso de pruebas se utilizó la herramienta JUNIT con la finalidad de verificar, manejar y ejecutar conjuntos de pruebas automatizadas.

2.2 Marco Conceptual

2.2.1 Metodologías Agiles. Las metodologías ágiles son "métodos de desarrollo de software en los que las necesidades y soluciones evolucionan a través de una colaboración estrecha entre equipos multidisciplinarios. Se caracterizan por enfatizar la comunicación frente a la documentación, por el desarrollo evolutivo y por su flexibilidad" (Pressman, 2010). Cumplen con el manifiesto ágil que es una serie de principios que se agrupan en 4 valores:

- Los individuos y su interacción, por encima de los procesos y las herramientas.
- El software que funciona, frente a la documentación exhaustiva.
- La colaboración con el cliente, por encima de la negociación contractual.
- La respuesta al cambio, por encima del seguimiento de un plan.

Entre las principales metodologías agiles se encuentran *Scrum* y *XP*.

2.2.1.1 Scrum. Este modelo fue identificado y definido por Nonaka e Takeuchi. Es una metodología de desarrollo ágil que se centra en el trabajo en equipo y la forma como permite la construcción de un producto desde piezas pequeñas, caracterizándose por la posibilidad de dar respuesta rápida a la retroalimentación recibida y el cambio constante para construir únicamente lo necesario. ***(Schwaber, 2017)***.

El marco de trabajo además de reglas cuenta con Roles claramente definidos:

- Dueño del Producto: Su función se encuentra orientada a determinar lo que hay que construir. Equipos de Desarrollo: Se encargan de construir lo que se necesita.
- *Scrum Masters*: Son los encargados de garantizar que el proceso ocurre en las mejores condiciones.

El funcionamiento de *Scrum* se identifican tres fases: planificación del sprint, seguimiento del sprint y revisión del sprint.

Planificación del *Sprint*

- Se define el *Product Backlog*, que consiste en una lista priorizada de requisitos del sistema.

- Planificación del sprint u iteración, es una jornada de trabajo previa al inicio de cualquier *sprint* y en la cual se determinan los objetivos y el trabajo que se debe cubrir. De esta reunión se obtiene una lista de tareas denominada *Sprint Backlog*.

Seguimiento del *Sprint*: Se llevan a cabo breves reuniones diarias, para revisar el avance de las tareas y el trabajo que está previsto para la jornada.

Revisión del *Sprint*: Una vez finalizado el *Sprint* se realiza un análisis y revisión del incremento generado. Se presentan los resultados finales.

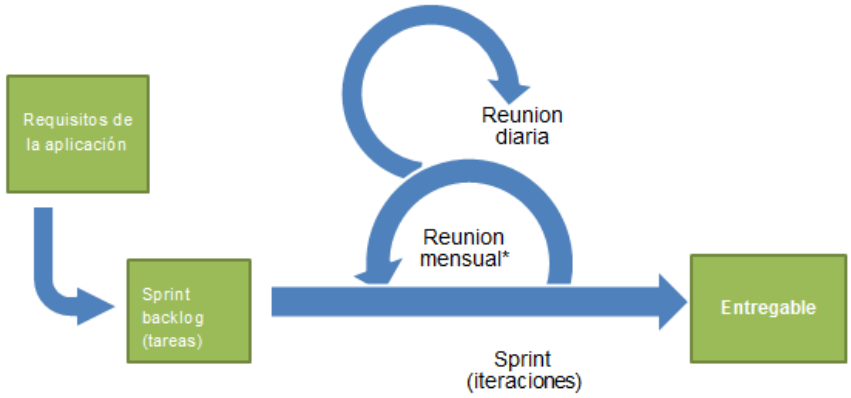

Figura 1 Scrum Ayala, M (2013) Scrum, Recuperado de https://es.wikipedia.org/wiki/Scrum

2.2.1.2 XP. Beck (2000) afirma:

> Es una metodología ágil basándose en realimentación continua entre el cliente y el equipo de desarrollo, comunicación fluida entre todos los participantes, simplicidad en las soluciones implementadas y coraje para enfrentar los cambios. *XP* se define como especialmente adecuada para

proyectos con requisitos imprecisos y muy cambiantes, y donde existe un alto riesgo técnico.

Aunque en otras fuentes de información aparecen algunas variaciones y extensiones de roles *XP*, en este apartado se describirán los roles de acuerdo con la propuesta original de Beck.

- Programador: Escribe las pruebas unitarias y produce el código del sistema. Debe existir una comunicación y coordinación adecuada entre los programadores y otros miembros del equipo.
- Cliente: Escribe las historias de usuario y las pruebas funcionales para validar su implementación. Además, asigna la prioridad a las historias de usuario y decide cuáles se implementan en cada iteración centrándose en aportar mayor valor al negocio.
- Encargado de pruebas (*Tester*): Ayuda al cliente a escribir las pruebas funcionales. Ejecuta las pruebas regularmente, difunde los resultados en el equipo y es responsable de las herramientas de soporte para pruebas.
- Entrenador (Coach): Es necesario que conozca a fondo el proceso XP para proveer guías a los miembros del equipo de forma que se apliquen las prácticas XP y se siga el proceso correctamente.
- Gestor: Es el vínculo entre clientes y programadores, ayuda a que el equipo trabaje efectivamente creando las condiciones adecuadas. Su labor esencial es de coordinación.

2.2.2 Modelo Vista Controlador. Es un patrón de arquitectura de las aplicaciones software que separa la lógica de negocio de la interfaz de usuario descrito por primera vez en 1979 para *Smalltalk* (es un lenguaje reflexivo de programación, orientado a objetos y con tipado dinámico). (Sommervile, 2005). De esta forma, dividimos el sistema en tres capas que son descritas a continuación.

2.2.2.1 Modelo. El modelo representa la parte de la aplicación que implementa la lógica de negocio, siendo el responsable de la recuperación de datos, así como su procesamiento, validación, asociación y cualquier otra tarea relativa a la manipulación de dichos datos. (Pavón, 2008)

Los objetos del modelo pueden ser considerados como la primera capa de la interacción con la base de datos.

2.2.1.2 Vista. La vista o interfaz de usuario hace una representación visual de los datos. Es responsable del uso de la información de la cual dispone para producir cualquier interfaz, ofreciendo una amplia variedad de formatos en función de sus necesidades tales como videos, música, documentos y demás (Sommervile, 2005).

Por ejemplo, como la capa de modelo devuelve un conjunto de datos, la vista los usaría para hacer una página *HTML* que los contenga.

2.2.1.3 Controlador. (Pavón, 2008) Afirma: que la capa del controlador gestiona las peticiones de los usuarios. Es responsable de responder la información solicitada con la ayuda tanto del modelo como de la vista.

Los controladores pueden ser vistos como administradores cuidando de que todos los recursos necesarios para completar una tarea. Espera peticiones de los clientes, comprueba su validez de acuerdo a las normas de autenticación o autorización, delega la búsqueda de datos al modelo y selecciona el tipo de respuesta más adecuado según las preferencias del cliente.

2.2.1.4 Funcionamiento.

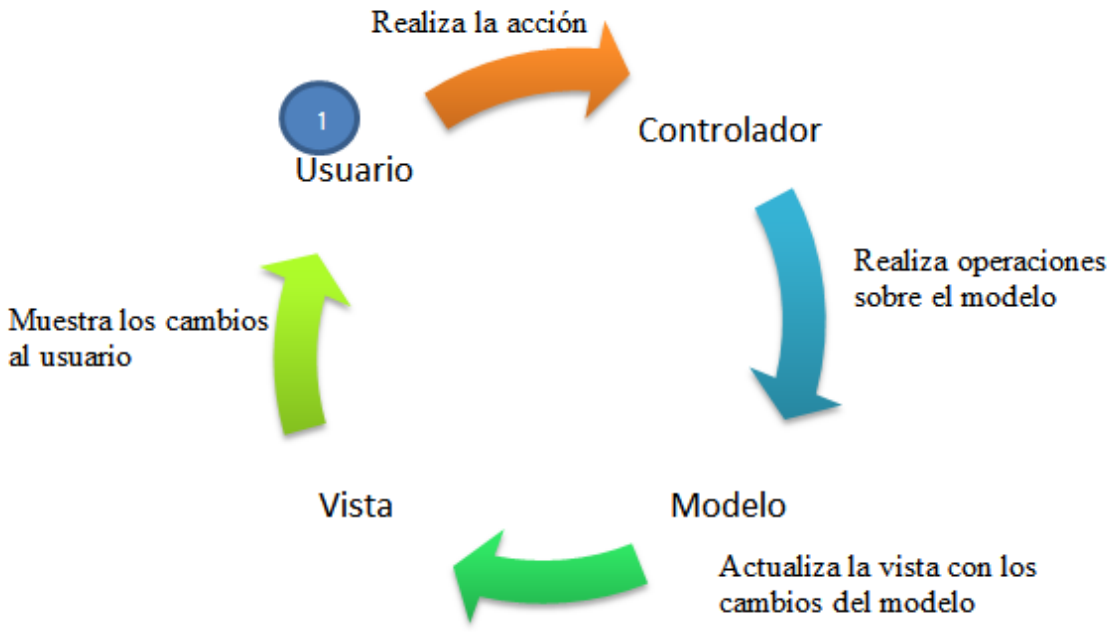

Figura 2 Modelo Vista Controlador

Fuente: Autor del proyecto

2.2.3 Aserciones. Una aserción se define como: "código que se usa durante el desarrollo -usualmente una rutina o una macro- que permite a un programa auto diagnosticarse al momento que se ejecuta" (McConnell, 1996).

Las aserciones son predicados que ponemos al final de las pruebas para comprobar si nuestro código está correcto, es como la pregunta que tratamos de responder y si la respondemos correctamente entonces pasamos la prueba

`assertTrue($x)`	Fail if $x is false
`assertFalse($x)`	Fail if $x is true
`assertNull($x)`	Fail if $x is set
`assertNotNull($x)`	Fail if $x not set
`assertIsA($x, $t)`	Fail if $x is not the class or type $t
`assertNotA($x, $t)`	Fail if $x is of the class or type $t
`assertEqual($x, $y)`	Fail if $x == $y is false
`assertNotEqual($x, $y)`	Fail if $x == $y is true
`assertWithinMargin($x, $y, $m)`	Fail if abs($x - $y) < $m is false
`assertOutsideMargin($x, $y, $m)`	Fail if abs($x - $y) < $m is true
`assertIdentical($x, $y)`	Fail if $x == $y is false or a type mismatch
`assertNotIdentical($x, $y)`	Fail if $x == $y is true and types match
`assertReference($x, $y)`	Fail unless $x and $y are the same variable
`assertClone($x, $y)`	Fail unless $x and $y are identical copies
`assertPattern($p, $x)`	Fail unless the regex $p matches $x
`assertNoPattern($p, $x)`	Fail if the regex $p matches $x
`expectError($x)`	Fail if matching error does not occour
`expectException($x)`	Fail if matching exception is not thrown

Figura 3 Assert Dante (2014) Assert, Recuperado de http://danteslab.blogspot.com.co

2.2.4 *Framework*: Se a realizará una comparación entre los *framework*s más utilizados para el desarrollo de pruebas unitarias. Actualmente existe multitud, pero vamos a centrarnos simplemente en unos pocos, estos están orientados a las pruebas unitarias, también sabemos que existen algunos otros para realizar pruebas de interfaz. Los *frameworks* que vamos a ver con más profundidad son *PHPUnit, JUnit, NUnit*.

2.2.4.1 JUnit. "es un conjunto de bibliotecas creadas por Erich Gamma y Kent Beck que son utilizadas en programación para hacer pruebas unitarias de aplicaciones. *Java".* (Beck K. , 2004).

JUnit es un conjunto de clases (*framework*) que permite realizar la ejecución de clases *Java* de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera.

Figura 4 JUnit Baskirt, O (2016) Junit, Recuperado de https://junit.org

2.2.4.2 PHPUnit. "Es un entorno para realizar pruebas unitarias en el lenguaje de programación *PHP*. *PHPUnit* es un *framework* de la familia *xUnit* de Kent Beck". (Bergmann, 2005).

Figura 5 PHPUnit. Software Testing (2013) PHPUnit, Recuperado de www.softwaretestingmagazine.com

PHPUnit se creó con idea de que cuanto antes se detecten los errores en el código antes podrán ser corregidos. Este conocido *framework* para PHP nos permite crear y

ejecutar juegos de *tests* unitarios de manera sencilla Como todos los *frameworks* de pruebas unitarias, *PHPUnit* utiliza *assertions* para verificar que el comportamiento de una unidad de código es el esperado.

2.2.4.3 NUnit. Es un *framework open source* de Pruebas de unidad para *Microsoft .NET* y *Mono*. Sirve al mismo propósito que *JUnit* realiza en el mundo *Java*, y es uno de muchos en la familia *xUnit*.

Figura 6 NUnit Poole, C (2002) NUnit, Recuperado de http://www.nunit.org/

2.3 Marco Teórico

2.3.1 ***Test Driven Development***. También llamado diseño basado en pruebas, "es un proceso de desarrollo de *software* que se basa en la idea de desarrollar pruebas, codificar y refactorizar el código construido" (Beck K. , 2002) su objetivo principal según Varela (2012):

1. Código limpio: El código debe cumplir con una buena legibilidad, debe estar ausente de duplicaciones, redundancias y mantener un alto nivel de mantenibilidad.

2. Código que obedece a una especificación. Todo el código de la aplicación se encuentra testeado, documentado y revisado.

3. Facilitar la resolución de problemas. Conseguir que los problemas complejos que surgen de la programación, o del diseño de sistemas software puedan ser simplificados por los desarrolladores de una forma predecible, evitando, de esta manera, estancamientos en su solución.

4. Flexibilizar el desarrollo. Este tipo de metodología permite dejar las opciones de desarrollo abiertas para estudiar diferentes caminos o soluciones a un mismo problema.

Las principales características de TDD son:

Pruebas unitarias. Una prueba unitaria, es cada una de las pruebas que verifican el buen funcionamiento de un módulo de código, así se asegura que cada mini versión de la aplicación final funciona correctamente por separado (Jurado, 2010).

En el caso de la metodología TDD, los *tests* deben ser automatizados, fáciles de repetir o modificar, y que permitan ser ejecutados continua y consecutivamente de una manera automática con el mínimo esfuerzo posible. La ejecución de un test debe producir un resultado inmediato de éxito, fallo o error.

Refactorización. Según Fowler (1999) es:

> El proceso de cambiar un sistema de software de manera tal que no cambia el comportamiento externo del código, sino que mejora su estructura interna. Es una manera disciplinada de ordenar el código para minimizar las posibilidades de introducir errores en el futuro. En esencia, cuando se refactoriza se está mejorando el diseño del código después de haber sido escrito.

Por tanto, el término refactorizar hace referencia a la acción de realizar modificaciones al código fuente asegurando que, tras esta transformación, no se produzca ningún cambio en el comportamiento final del código o de la aplicación, es lo que conocemos informalmente por limpiar el código, ni arregla errores ni añade funcionalidad.

Desarrollo iterativo e incremental. Es una de las prácticas recomendadas en las metodologías agiles como es el caso de la TDD. Está dirigida por las pruebas y la refactorización periódica (Varela, 2012).

Funcionamiento*:* El algoritmo de TDD según Jurado ***(2010)***:

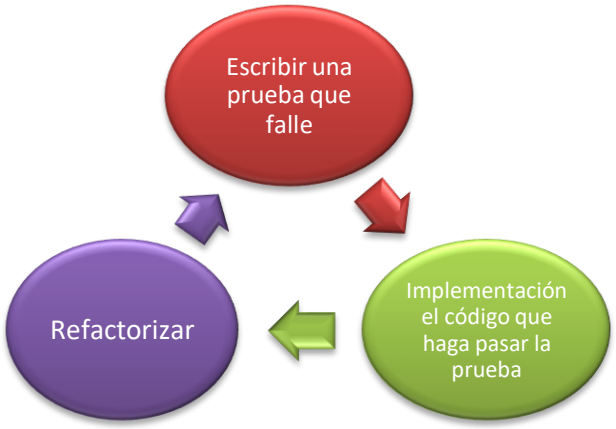

Figura 7 Funcionamiento TDD

Fuente: Tomado de (Jurado, 2010)

2.3.2 Pruebas. Las pruebas tienen por objeto asegurar la calidad del software, existen diferentes tipos de pruebas y a continuación se detallarán cada uno.

Las pruebas de caja blanca "se centra en analizar el código fuente y la lógica interna del software" (Jenkins, 2008), su objetivo principal es probar la lógica del programa desde el punto de vista algorítmico ya que se analiza cada módulo individualmente y las pruebas de caja negra las cuales "se centra en comparar las entradas con las salidas esperadas, sin detenerse en detalles de su desarrollo (código fuente)" (Jenkins, 2008).

Tabla 1
Tipos de prueba.

Tipo de pruebas	Descripción	objetivo
Pruebas de Unidad.	Normalmente son una combinación de Caja Blanca y Caja Negra ya que analizan la lógica del módulo usando métodos de caja blanca y completan las pruebas con métodos de caja negra para garantizar que cada módulo cumple con las especificaciones. (Varela, 2012)	Detectar errores en los datos, lógica y algoritmos
Pruebas de Integración	Las Pruebas de Integración son pruebas que comprueban el funcionamiento entre módulos, verificando que los componentes de la aplicación funcionan correctamente actuando en conjunto (Pressman, 2010). Existen dos estrategias o acercamientos, uno incremental y otro no incremental.	Detectar errores de interfaces y relaciones entre componentes
Pruebas de Sistema	Evalúan el sistema en su conjunto. Suele ser habitual que se prueba el sistema en un entorno similar al de producción.	Detectar fallas en el cubrimiento de los requerimientos funcionales
Las Pruebas de Carga	Someten el sistema a cargas de trabajo extremas determinando la capacidad de resistencia límite del programa, número de usuarios simultáneos, capacidad de cálculo máximo, número de conexiones, etc.	
Pruebas de Regresión	Las Pruebas de Regresión sólo son aplicables cuando existen versiones previas del sistema. Consisten en comprobar que las versiones anteriores, o el funcionamiento de las versiones anteriores, siguen estando soportadas por el sistema “ (Varela, 2012)	Asegurar que los cambios no introducen un comportamiento no deseado u errores adicionales
Pruebas de Aceptación	Evalúan que el sistema cumple con los requisitos del cliente, para ello se cuenta con su participación y por norma general la superación de este tipo de pruebas significa que el cliente acepta el sistema y éste queda validado” (Varela, 2012)	Detectar fallas en la implementación del sistema

Fuente: Autor del proyecto

2.4 Marco Legal

La calidad del producto software se encuentra determinada en la norma (IEC, 1998), que describe un modelo dividido en 2 partes: la calidad interna y externa, especificando 6 características como lo son funcionalidad, fiabilidad, usabilidad, eficiencia, mantenibilidad y portabilidad contenidas en las normas ISO/IEC 9126-2 para métricas externas y ISO/IEC 9126-3 para métricas internas de calidad.

A continuación se resume cada una de las características antes mencionadas:

- Funcionalidad: es la capacidad del producto software de proporcionar las funciones que satisfacen las necesidades implícitas o explicitas.
- Fiabilidad: conjunto de atributos relacionados con la capacidad del software de mantener su nivel de prestación bajo condiciones específicas.
- Usabilidad: es la capacidad de software de ser entendido, aprendido, usado y atractivo para el usuario.
- Eficiencia: se refiere a la capacidad de proveer un desempeño adecuado, acorde a la cantidad de recursos utilizados.
- Mantenibilidad: se relaciona con la facilidad de extender, modificar, corregir errores en un software.
- Portabilidad: es la capacidad de un software para ser transferido desde una plataforma a otra.

ISO 9000 es un conjunto de estándares internacionales para sistemas de calidad. Diseñado para la gestión y aseguramiento de la calidad, especifica los requisitos básicos para el desarrollo, producción, instalación y servicio a nivel de sistema y a nivel de producto.

- ISO 9001 es un estándar que describe el sistema de calidad utilizado para mantener el desarrollo de un producto que implique diseño.
- ISO 9000-3 es un documento específico que interpreta el ISO 9001 para el desarrollador de software.
- ISO 9004-2 proporciona las directrices para el servicio de facilidades del software como soporte de usuarios.

ISO/IEC 15504 también conocido como *Software Process Improvement Capability Determination*, abreviado *SPICE*, en español, (Determinación de la Capacidad de Mejora del Proceso de Software) es un modelo para la mejora, evaluación de los procesos de desarrollo, mantenimiento de sistemas de información y productos de software.

ISO/IEC 15504 es un estándar internacional emergente de evaluación y determinación de la capacidad y mejora continua de procesos de ingeniería del software, con la filosofía de desarrollar un conjunto de medidas de capacidad estructuradas para todos los procesos del ciclo de vida y para todos los participantes.

ISO/IEEE 29119 es un estándar para pruebas de software que puede ser utilizado en cualquier ciclo de vida., se compone de conceptos y definiciones, procesos de prueba, documentación, técnicas y pruebas dirigidas por palabra clave.

Capítulo 3 Diseño Metodológico

3.1 Tipo de Investigación.

De acuerdo con Hernández, Baptista & Fernández (2010), la investigación descriptiva permite especificar las características de la realidad observada para su posterior análisis. Para el caso concreto de la investigación, se buscó hacer una descripción de la situación actual del proceso de aseguramiento de calidad a través de un modelo de pruebas utilizando TDD para mejorar las prácticas en el desarrollo de los sistemas de información de la UFPSO.

3.2. Método de investigación.

De acuerdo con Bernal (2010) el enfoque cualitativo está enfocado en describir y comprender situaciones a partir de características determinantes de acuerdo a como sean percibidos para encontrar resultados confiables. La presente investigación presenta un enfoque cualitativo, que permitirá analizar y describir el proceso de control de calidad del desarrollo de software. Finalmente, se realizará la implementación de un modelo de pruebas automáticas que ayudaran a la optimización del control de la calidad.

3.3. Instrumentos de recolección de información

En la presente investigación se utilizaron instrumentos de recolección de información de los cuales a continuación se presenta una breve descripción e intención de cada uno ellos:

3.3.1 Análisis documental. La técnica de análisis documental es un procedimiento sistemático para revisar o evaluar documentos dentro de una perspectiva de estudio, que requiere que los datos sean interpretados y examinados a fin de obtener significado. (Corbin, 2008). El análisis documental estará enfocado en la revisión de literatura acerca del testing agile y principalmente en prácticas relacionadas con las siguientes metodologías de prueba: TDD, BDD y ATDD.

3.3.2 Entrevista semi-estructurada. Canales (2006), la define como "la comunicación interpersonal establecida entre el investigador y el sujeto de estudio, a fin de obtener respuestas verbales a los interrogantes planteados sobre el problema propuesto".

La entrevista estuvo enfocada en averiguar:

- El proceso de desarrollo de software: se busca conocer en detalle el proceso de desarrollo que se tiene actualmente en la empresa.

- La ejecución de casos de prueba: se busca identificar cuáles de los casos de prueba existentes eran relevantes para una posible automatización y de ellos tomar una muestra para una simulación a realizar en el marco de la investigación.

3.3.2.1 Población. La población objeto de estudio está conformada por el equipo de desarrollo de la División de Sistemas actualmente esta cuenta con 8 integrantes, los cuales tienen roles definidos de *BackEnd* y *FrontEnd.*

3.3.2.2 Selección de la Muestra. En este proyecto teniendo en cuenta los diferentes sistemas de información que han sido desarrollados por la división de sistemas, se tomará el Sistema de Información Académica SIA se implementará la metodología de TDD, se encuentra a cargo de 4 ingenieros y el coordinador del sistema que lideran la nueva versión del software.

3.3.3 Simulación. La simulación se define como un proceso cuyo objetivo es plantear la representación de un sistema real, con la única finalidad de entender el funcionamiento del sistema o la evaluación de nuevas estrategias (Johaness & Shanon, 1976). La simulación es relevante porque permitirá codificar y ejecutar los casos de prueba.

3.4 Actividades de elaboración del proyecto

3.4.1 Fundamentación Conceptual

- Revisión de literatura
- Lectura crítica y analítica
- Elaboración del marco histórico
- Elaboración del marco teórico y conceptual
- Elaboración del marco legal.

3.4.2 Actividades Operativas.

- Desarrollar un modelo de pruebas que permita fusionarse con la arquitectura MVC.
- Obtener fuentes de documentación con el fin de identificar casos de prueba.
- Describir el Caso de Pruebas.
- Escribir casos de pruebas para requisitos funcionales del *product Backlog*.
- Entender defecto reportado y validar la corrección del defecto
- Registrar los resultados obtenidos

- Realización de la documentación e implementación en los sistemas de información.
- Revisión del director
- Correcciones y ajustes
- Entrega de documento final
- Sustentación

Capítulo 4 Recursos

4.1 Recursos

4.1.1 Recursos Humanos. Investigador: Angie Lorena Ballesteros Coronel, estudiante de Ingeniería de Sistemas de la Universidad Francisco de Paula Santander Ocaña, autor del proyecto

Director del proyecto: Msc. Byron Cuesta Quintero

4.1.2 Recursos Institucionales

- Universidad Francisco de Paula Santander Ocaña
- Biblioteca UFPSO
- Biblioteca virtual ACM

4.1.3 Recursos Materiales

- Equipo de computación y comunicación
- Equipo de transporte
- Útiles de papelería
- Internet
- Equipo de oficina

4.1.4 Recursos Financieros

Los egresos ocasionados durante la realización del proyecto serán asumidos por el autor, causando conceptos específicos así:

Tabla 2
Ingresos y Egresos ocasionados en el proyecto

Presupuesto		
Descripción	Debe	Haber
Aportes del autor del proyecto		$ 4’640.000
Personal	$ 3.360.000	
Equipo	$ 700.000	
Papelería y fotocopias	$ 50.000	
Transporte	$ 300.000	
Digitación e impresión	$ 100.000	
Preparación informe final	$ 80.000	
Gastos varios	$ 50.000	
TOTAL	$ 4’640.000	$ 4’640.000

Fuente: Autor del proyecto

Tabla 3.
Detalle de gastos personales.

Detalle de Gasto de Personal				
NOMBRE	**VALOR HORA**	**HORAS/MES**	**# MESES**	**TOTAL**
Director	$ 20.000	12	4	$ 960.000
Desarrollador	$ 12.000	50	4	$ 2.400.000
TOTAL				$ 3.360.000

Fuente: Autor del proyecto

Capítulo 5 Resultados

Los resultados de la entrevista semiestructurada que se realizó al equipo de desarrollo de la División de Sistemas de la UFPSO (ver apéndice A) tuvo como objetivo conocer el actual proceso de las pruebas de software, se logró denotar que para el desarrollo de los proyectos utilizan la metodología *Scrum*, junto con la arquitectura modelo vista controlador, además en la etapa de *testing* no se están llevando a cabo casos de pruebas automatizados, solamente pruebas manuales de caja negra y caja blanca ejecutadas en la etapa de desarrollo antes de ser lanzado a producción.

Acerca de la documentación del proceso se guarda en un archivo compartido junto con los bugs o errores encontrados pero no se tiene la información completa para reproducir nuevamente cualquier escenario de error.

La mayoría de los desarrolladores manifiestan no tener conocimientos acerca del manejo de los framework de pruebas que ayuden a la automatización.

5.1 Análisis documental

A continuación se presenta el análisis documental acerca del *testing agile* exponiendo algunos de los principios, aspectos más importantes y las diferencias entre las metodologías: TDD, BDD, ATDD.

5.1.1 Testing agile. En el mundo del desarrollo de software, el término ágil generalmente se refiere a "cualquier enfoque de gestión de proyectos que se esfuerce por unir a los equipos en torno a los principios de colaboración, flexibilidad, simplicidad, transparencia y capacidad de respuesta durante todo el proceso de desarrollo de un nuevo programa o producto". Y las pruebas ágiles generalmente significan la práctica de probar software para detectar errores o problemas de rendimiento orientada a dichas metodologías de desarrollos ágil, como XP, AUP, Scrum entre otras, "incorporando, una serie de herramientas orientadas al diseño basado en comportamiento, automatización de las pruebas, integración continua, propiedad colectiva del código y métricas de calidad del software" (Beck K. A., 2000)

Los principios del *testing agile*:

- Las metodologías ágiles no ven al *testing* como una fase separada, sino como parte integral del desarrollo de software, dado que éstas son indispensables para la liberación exitosa de cada entregable.

- *Agile Testing* proporciona retroalimentación continua, permitiendo corregir el rumbo continuamente durante el desarrollo de software y haciendo avanzar más rápido el proyecto.

- Los analistas de negocio y desarrolladores de software también ejecutan pruebas, no sólo los *testers* como en métodos convencionales.

- Los equipos del área de negocio, los clientes están involucrados en cada iteración, como resultado, el tiempo de retroalimentación se reduce y el costo de correcciones también es menor.

- Los defectos en el código se corrigen en la misma iteración, por lo que se mantiene el código limpio.

Existe una gran cantidad de pruebas, descritas en los denominados cuadrantes del *agile testing* donde se esquematizan los tipos de pruebas y el grado de automatización, además ayudan al probador a comprender las técnicas del *testing* y el momento en que se deben aplicar.

A continuación una breve explicación de los cuadrantes del *Agile Testing*, originados por Brian Marick en un post sobre el tema y luego adaptados por Lista Crispin y Janet Gregory en el libro de *Agile testing* (Crispin & Gregory, 2008)

Q2 automatizadas y manuales Test funcionales simulaciones prototipos pruebas de historias	**Q3 Manuales** Test de escenarios pruebas de usabilidad pruebas de aceptación (alfa / beta)
Q1 Automatizadas Pruebas unitarias pruebas de componentes	**Q4 herramientas** Pruebas de desempeño y carga pruebas de seguridad

Figura 8 Cuadrantes agiles

Fuente: Autor del proyecto, adaptado de (Crispin & Gregory, 2008)

Cuadrante 1 Este cuadrante representa el test guiado por pruebas (TDD), como parte de las pruebas unitarias que verifican la funcionalidad de una pequeña parte del código y pruebas de componentes que verifiquen el comportamiento de una parte del sistema.

Cuadrante 2 Son pruebas llamadas *business-facing tests* que representan las pruebas de cliente es decir, lo que el cliente quiere, escrito en lenguaje de "negocio" y sirven para darles una visión del producto al equipo.

Prueba de ejemplos de posibles escenarios y flujos de trabajo, pruebas de la experiencia del usuario como prototipos, prueba de pareja.

Cuadrante 3 En este cuadrante, es una prueba integral para determinar si cumple con las expectativas del cliente. Se trata de emular lo más que se pueda el entorno en el cual se ejecutarán siendo realizadas por los clientes, por este motivo son pruebas manuales como: pruebas exploratorias, pruebas de usabilidad, pruebas exploratorias, pruebas colaborativas, prueba de aceptación del usuario.

Cuadrante 4 Se concentra en los requisitos no funcionales como el rendimiento, la seguridad, la estabilidad, etc. Se incluyen las pruebas relacionadas con el rendimiento, carga, seguridad, etc.

Algunas de las practicas relacionadas con el *Testing* Ágil son: TDD, BDD, ATDD, en el siguiente cuadro se expone las definiciones, objetivos, herramientas de software utilizadas y ciclo de vida de cada uno.

Tabla 4
Marco comparativo.

TDD	BDD	ATDD
Test Driven Development: Desarrollo dirigido por pruebas	Behaviour Driven Development: Desarrollo Guiado por el Comportamiento	*Aceptance Test Driven Develepment*: Desarrollo guiado por pruebas de aceptación.
Es de finales de los 90 principios del 2000, fue la primera técnica.	Surgida en 2002, como mejora de TDD.	Empezada a desarrollar en 2003 y surge para realizar una ampliación a TDD
Es una técnica que combina un enfoque de refactorización del lado de desarrollo con la técnica de probar primero en cuanto al *testing*.	Primero se desarrolla una prueba funcional o de historia de usuario automatizada, luego se ejecuta el desarrollo aplicando TDD hasta que la prueba es exitosa.	Son pruebas escritas a nivel de cliente, es decir, lo equivalente a una prueba de aceptación o test funcional
El objetivo de esta técnica es minimizar la creación del código implementado solo lo imprescindible, reduciendo errores, creando software con aceptación ante cambios	Lo que plantea es definir un lenguaje común para el negocio y para los técnicos, y utilizarlo como parte inicial del desarrollo y el *testing*	El objetivo de ATDD es especificar requisitos detallados y ejecutables para su solución en el momento justo
Las pruebas se escriben antes del propio código a probar, realizamos el código que pase la prueba y por último la refactorización para mantener la calidad del diseño, se cambia el diseño sin cambiar la funcionalidad.	Construiremos sentencias con las que vamos a describir las funcionalidades teniendo en cuenta estas palabras: Dado: los pasos necesarios para poner al sistema en el estado que se desea probar Cuando: La interacción del usuario que acciona la funcionalidad a testear. Entonces: observar los cambios en el sistema y ver si son los deseados.	Elegimos la historia a implementar, escribimos los criterios y pruebas de aceptación por ultimo realizamos la implementación de las funcionalidades mostrando el demo al cliente Las pruebas de aceptación dirigen el diseño/desarrollo del sistema
Herramienta: XUnit como: *PHPUnit*, JUnit NUnit, DBUnit	Cucumber, Jbehave, Jasmine,Kahlan y Behave.	Fit / FitNess, concordion

Fuente: Autor del proyecto

En los desarrollos de proyectos agiles puedes utilizar estas 3 metodologías de prueba: TDD, BDD y ATDD, no es necesario definir cuál metodología es mejor, sino más bien identificar claramente cuándo aplicar cada una, dado que los proyectos son cambiantes, es decir diferentes en muchos aspectos, entre ellos en tamaño, en complejidad, etc. Toda esa serie de elementos nos permitirán definir si las metodologías ágiles son aplicables o no en cada caso, o donde es realmente necesario aplicar las 3 técnicas, si bien hay puntos en que se pueden superponer por ejemplo, la creación de Criterios de aceptación en ATDD pueden reusarse en los escenarios de BDD, las tres herramientas sirven para gestionar de una forma eficiente durante el sprint.

En este documento se va solo a utilizar la metodología de *Test Driven Development* debido a que se ejecutaran pruebas unitarias a los diferentes componentes, además las prácticas de BDD y ATDD no son necesarias ya que el software a desarrollar es una nueva versión y se encuentran definidos los requisitos.

Para las pruebas automatizadas, utilizar estas mejores prácticas puede garantizar que las pruebas sean exitosas y obtenga el máximo retorno de la inversión

A continuación se expone los principios expuestos por (Sundaram) e *Intervise* realizando un cuadro comparativo para tomar las ventajas de las pruebas automatizadas.

Tabla 5

Marco comparativo.

	SMART BEAR	INTERVISE
Decidir qué casos de prueba automatizar	El beneficio de las pruebas automatizadas está relacionado con cuántas veces se puede repetir una prueba determinada. Puede obtener el mayor beneficio tenga en cuenta los siguientes aspectos: Casos de pruebas repetitivos que se ejecutan para compilaciones múltiples. Pruebas que tienden a causar un error humano. Pruebas que requieren múltiples conjuntos de datos.	Las pruebas para requisitos no funcionales a menudo deben ser automáticas para determinar qué configuraciones del sistema ofrecen un rendimiento óptimo. Las pruebas de requisitos funcionales, por otro lado, tienden a pasar o fallar por naturaleza: la aplicación bajo prueba (AUT) cualquiera da la respuesta esperada o no.
Pruebe temprano y pruebe a menudo	Cuanto antes se involucren los probadores en el ciclo de vida del proyecto, mejor, y cuanto más pruebas, más errores se pueden encontrar.	Comience por esas pruebas manuales que se ejecutan de manera repetitiva y que pueden causar errores automatice los procesos genéricos tanto como sea posible para que tenga tiempo de analizar en detalle el problema y los errores
Seleccione la herramienta de prueba automatizada.	Se debe elegir la que mejor se adapte a sus necesidades, por ejemplo el soporte de plataformas, tecnologías, lenguajes de programación, exportación de informes y demás.	La automatización de las pruebas requiere recursos de hardware y software que se deben planificar teniendo en cuenta la curva de aprendizaje. Elegir la mejor herramienta es fundamental para la construcción de pruebas robustas y mantenible
Divida sus esfuerzos de prueba automatizados	La creación de diferentes pruebas se basa en los niveles de habilidad de los ingenieros de calidad, es importante identificarlas junto con los conocimientos para dividir esfuerzos.	Formar un equipo de automatización de pruebas que conste de al menos un arquitecto de automatización. Las pruebas realizadas por un equipo son más efectivas para encontrar defectos.
Pruebas manuales	Las pruebas manuales no se deben dejar a un lado probar todos los escenarios manualmente de esta manera ayuda a identificar el alcance real de la automatización	La buena automatización de pruebas comienza con la creación de buenos casos de prueba manual. Escribir primero el caso de prueba en forma manual ayuda a identificar todos los requisitos previos y datos de prueba.

Fuente: Autor del proyecto

5.2 Guía del modelo de pruebas

El desarrollo de software requiere estándares que garanticen la calidad de los productos generados antes de ser implementados en una organización. Los principios y exigencias definidos para tal propósito se conocen como buenas prácticas.

La guía planteará un proceso de pruebas aunado con las buenas prácticas existentes para enriquecer el ciclo desarrollo de software de la División de Sistemas de la Universidad Francisco de Paula Santander Ocaña integrándolas en un único modelo formal y completo.

La sección 1 esbozará el nuevo ciclo aplicado en el desarrollo de software integrando la fase de pruebas. En la sección 2 mostrará las indicaciones necesarias para la adecuación del entorno dedicado para la fase de pruebas, preparación del hardware, software y librerías. La sección 3 planteará el plan de pruebas donde se especificará las pruebas a ser ejecutadas con la programación de su ejecución. Señalará los casos de pruebas, su evaluación, el reporte de errores encontrados siendo el informe final que deberá presentarse.

Ciclo dentro del proceso de desarrollo de software: Estas pruebas no se realizan de manera separada ni en una fase posterior al desarrollo, sino que se integran en él. Estas pruebas tienen que comenzar antes de la codificación inicial. Como siempre, las

pruebas y el código se realizan de forma que al final del sprint tenga el valor suficiente para subirlo a producción.

Teniendo en cuenta que el desarrollo del software por parte de la División de Sistemas se realiza bajo la metodología ágil *Scrum*, se integraran las pruebas de software en cada sprint según la lista de requisitos priorizados en el *product backlog*. A continuación, se muestra el grafico correspondiente donde se selecciona una tarea del *Sprint Backlog* luego se crea una prueba unitaria, es decir código adicional que ejecuta un aspecto de una pieza de código producido, utilizando la herramienta como *PHPUnit*, se produce el código necesario para cumplir con la funcionalidad que debe tener y por último se refactoriza para simplificar el código, logrando obtener el código listo para implementar.

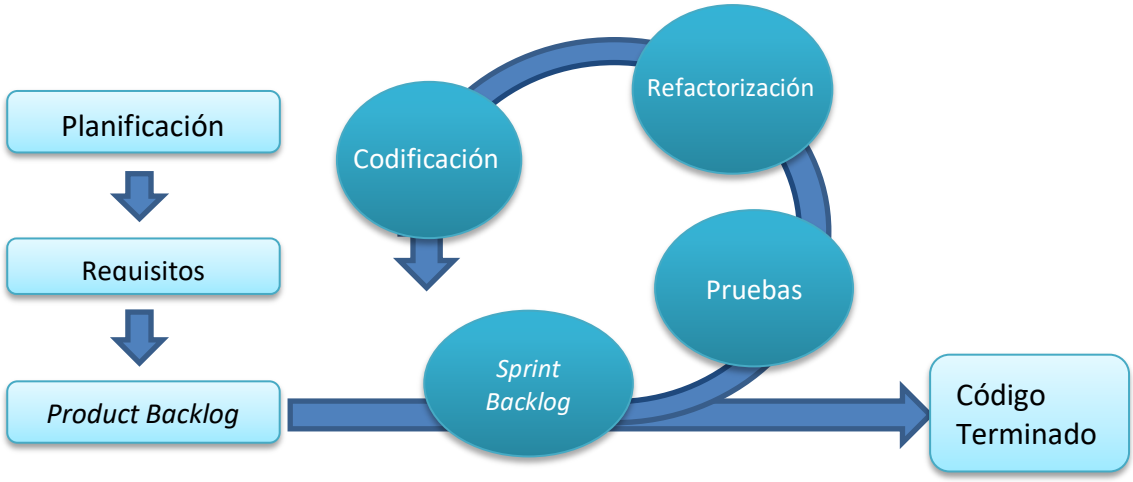

Figura 9 Ciclo de Desarrollo con TDD

Fuente: Autor del proyecto, adaptado de (Myklebust, 2015)

Fases del modelo de pruebas: En la Fig. 10 se expone las cuatro fases propuestas del modelo de pruebas: Planificación, Diseño, Ejecución y Evaluación con su respectiva definición.

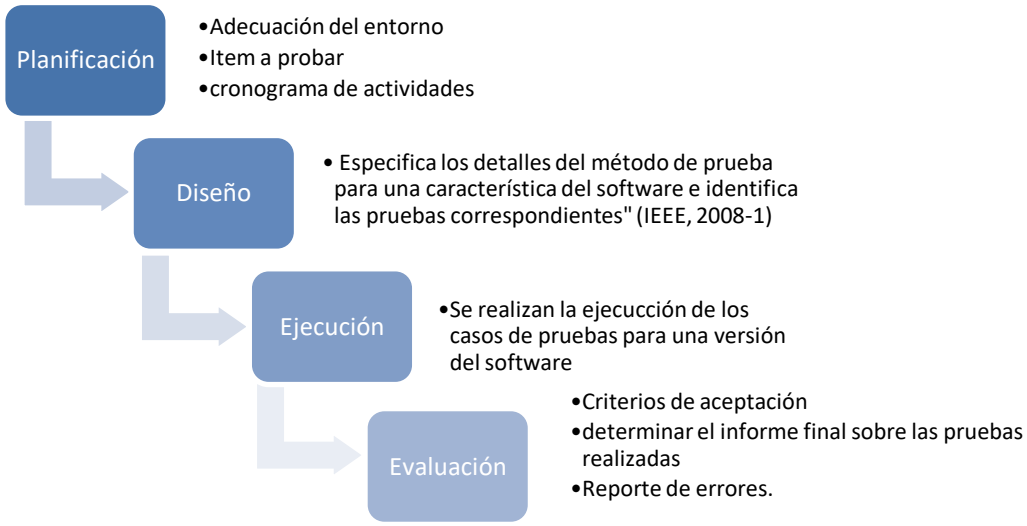

Figura 10 Fases del plan de pruebas

Fuente: Autor del proyecto

5.2.1 Conocimiento del sistema. Con el fin de testear las diferentes funcionalidades de un software, es necesario conocer el funcionamiento y la operatividad del sistema, tal cual el programador lo desarrollo, basado en los requerimientos y especificaciones para así encontrar errores y solucionarlos de manera ordenada y documentada.

5.2.2 Plan de pruebas. El plan "describe el alcance y enfoque de las actividades de pruebas previstas, e identifica las características a ser probadas" (Elaine, 2000), conformado por los diferentes casos de pruebas, determina los criterios de calidad para asegurar que los sistemas cumplen con los requisitos de funcionamiento, permitiendo detectar los errores, defectos o fallas que tiene el software, realizando las correcciones pertinentes, según sea el caso.

Las diferentes secciones del plan de pruebas, incluyen el alcance, estrategia de pruebas, tipos de pruebas de software a incluir, criterios de aceptación, requisitos de infraestructura, requisitos de personal, la planificación, el diseño y generación de casos de pruebas.

5.2.2.1 Planificación. Es un elemento esencial dentro del proceso de pruebas cuyo objetivo principal es llevar a cabo el control, la gestión y el seguimiento de cada uno de los casos de prueba.

Durante el desarrollo de la planificación, se identificarán y analizarán diferentes elementos que inciden en el desarrollo, alcance, la estrategia, roles involucrados, responsabilidades, la adecuación del entorno y tiempo de pruebas. Además, se deben tener en cuenta qué tipos de pruebas se realizarán por lo tanto es necesario seleccionar los ítems y características de la aplicación a probar incluyendo las funcionalidades implementadas y los aspectos asociados a los requerimientos no funcionales.

5.2.2.1.1 Adecuación del entorno. Es necesario llevar a cabo la preparación del entorno para ejecutar el conjunto de pruebas especificadas, que permitan a los desarrolladores reproducir la ejecución del sistema como si estuviera en un ambiente de producción.

"El entorno de pruebas debe contener el hardware, instrumentación, simuladores, herramientas software y otros elementos de soporte necesarios para realizar una prueba". (ISO, 1990)

El ambiente de pruebas debe incluir:

- Asignación de un servidor con las especificaciones técnicas requeridas.
- Creación de los perfiles de usuario necesarios para tener acceso a todas las instancias del aplicativo.
- Instalación del motor de la base de datos para realizar las pruebas.

Software para realización de las pruebas automatizadas.

PHPUnit. Los requerimientos para que el software funcione adecuadamente son: mínimo la versión *PHP* 5.6, preferiblemente utilizar la última versión de *PHP* y las extensiones "dom", "pcre", "reflection", "sp" y "json" las cuales están habilitadas por defecto.

Selenium: es un entorno de pruebas de software para aplicaciones basadas en la web, provee una herramienta de grabar/reproducir para crear pruebas a través de *Selenium IDE*, se instala con extensión en los navegadores (*Chrome*, *Firefox*). También se pueden ejecutar las pruebas, con *Selenium webdriver* utilizando *PHPUnit*.

5.2.2.1.2 Alcance de la prueba. Determina que funcionalidades del producto y/o software serán probadas durante el transcurso de la prueba. Este listado de funcionalidades a probar se extrae con base al impacto que podría ocasionar la falla de una funcionalidad y la probabilidad de que falle, producto de este análisis, permite determinar además del alcance detallado del proceso de pruebas, la prioridad con la que las funcionalidades deben probarse.

Tabla 6
Control de funcionalidades.

FUNCIONALIDAD	PRIORIDAD	VERSIÓN

Fuente: Autor del proyecto

Funcionalidades: listado de características a probar.

Prioridad: consiste en establecer un orden de ejecución y un nivel de importancia para los casos de prueba que permita alcanzar un determinado objetivo, logrando detectar fallos lo antes posible o conseguir un determinado nivel de cobertura de código cuanto antes. Los valores posibles son: alta, media y baja.

Versión de producto software: Identificador del software en la que se va a desarrollar las pruebas.

En la Tabla 7 se debe ingresar la información de los casos de prueba a ejecutar relacionados según la funcionalidad que se desea probar.

Tabla 7
Control de casos de prueba.

FUNCIONALIDAD	ID	CASO DE PRUEBA

Fuente: Autor del proyecto

Funcionalidades: listado de características a probar.

Id: identificador del caso de prueba.

Caso de prueba: nombre del caso de prueba que se va a ejecutar.

5.2.2.1.3 Roles y responsabilidades. Específica los roles involucrado y quién es el responsable de cada una de las tareas previstas en el plan. Para cada uno de ellos, se identifican sus responsabilidades dentro del equipo, así como las competencias que deberían poseer y el nivel de estudios requerido.

5.2.2.1.4 Tiempos. Se realiza un calendario de pruebas teniendo en cuenta: El responsable de la prueba y el caso de prueba que va a hacer ejecutado.

***5.2.2.2 Diseño de casos de prueba*.** Se establece toda la información necesaria para realizar una ejecución efectiva de las pruebas, teniendo en cuenta los detalles necesarios para organizar y ejecutar las pruebas; por ejemplo, cuales son las entradas requeridas, las salidas esperadas o el procedimiento a seguir para cada una de las pruebas. (IEEE/ISO, 2018)

5.2.2.2.1 Formato para los casos de prueba. Una de las estrategias para las pruebas es crear una plantilla para formalizar dicha ejecución, este formato se encuentra disponible en el Apéndice B donde se explica detallamente cada elemento que lo conforma.

***5.2.2.2.2 Reporte de errores*.** Informe detallado del error si es posible la ubicación de cada uno y su gravedad e impacto en el sistema además se brindan sugerencias de corrección para dichos errores. En el Apéndice C, se expone la plantilla para llevar seguimiento de errores encontrados.

5.2.2.3 Generación de casos de prueba. El propósito es realizar la ejecución de los casos de prueba para evaluar que el producto integrado y completo funciona correctamente. La ejecución de las pruebas debe llevarse a cabo de forma disciplinada, utilizando los *scripts* generados.

- Escribir los *scripts* para el caso de prueba

- Realizar el código que pase la prueba.
- Refactorización
- Verificar que cumplen con las condiciones de aprobación.

5.2.2.3.1 Escribir los scripts para los casos de prueba Para crear la clase de prueba, es necesario incluir la funcionalidad de carga automática de *PHPUnit*. A continuación, defina la clase de prueba que se extiende *PHPUnit_Framework_TestCase*.

- *ClassTestHereda* (la mayor parte del tiempo) de *PHPUnit\Framework\TestCase*. es una clase de prueba abstracta proporcionada por el marco *PHPUnit*
- Las pruebas son métodos públicos que se denominan *test**.

Durante las pruebas con base de datos es importante verificar el estado de la base de datos antes y después de que se haya ejecutado la prueba y limpiar la base de datos para cada nueva prueba.

A continuación, se encuentra un ejemplo utilizando el *framework PHPUnit*

```php
<?php
require 'model.php';
use PHPUnit\Framework\TestCase;
class testuser extends TestCase {
    function testUsuario() {
        $result = $this->user->getUsuario('1091671555', 'divisis*2018-1');
        $this->assertEquals('ES', $result);
        return $result;
    }
```

Figura 11 Ejemplo de test con PHPUnit

Fuente: Autor del proyecto

En el Apéndice D, puedes encontrar muestras de código que ayudan a entender mejor el funcionamiento de *PHPUnit*.

En el caso de escribir Script con Selenium IDE, véase el Apéndice E que contiene ejemplos acerca de utilización para la ejecución de pruebas automatizadas.

5.2.2.3.2 Realizar el código que pase la prueba. En esta fase se debe realizar el código más simple que haga que la prueba pase cada una de invocaciones a métodos de comprobación (Assert Methods).

PHPUnit provee una gran cantidad de métodos *assert*, cuyas referencias se pueden encontrar en el manual oficial. Algunas características comunes: para la mayoría de método *assert* existe su opuesto: *assertContains*() y *assertNotContains*(). Adicionalmente, a cada método *assert*, requieren el paso de dos parámetros obligatorios, generalmente guardan el siguiente orden: *metodoAssert*($valor_esperado, $valor_recibido) y un parámetro opcional, un mensaje personalizado para ser arrojado en caso de error.

5.2.2.3.3 Refactorización. Existe la llamada regla de los tres strikes (Fowler, 1999) La primera vez solo hazlo, la segunda vez que haces algo similar se realiza la duplicación y finalmente, cuando estás haciendo algo por tercera vez, comience a refactorizar. Otros momentos propicios para refactorizar son:

- Al momento de agregar funcionalidad. Es común refactorizar al momento de aplicar un cambio al software ya funcionando, ya que ayuda a comprender mejor el código y hace que sea más fácil agregar nuevas características.
- Al momento de resolver una falla. El reporte de una falla del software suele indicar que el código no estaba lo suficientemente claro como para evidenciar la misma.
- Durante la revisión de código, puede ser la última oportunidad para poner en orden el código antes de que esté disponible para el público, lo mejor es realizar dichas revisiones en un par con un autor para solucionar problemas rápidamente.

Síntomas que indican la necesidad de refactorizar (Piatini, 2003) analizan los síntomas que indican la necesidad de refactorizar, a los que Fowler llamaron *bad smells* (malos olores). se describen brevemente en el Apéndice F

5.2.2.3.4 Ejecución y evaluación de los casos de prueba: El objetivo principal de esta actividad es analizar el resultado obtenido de la ejecución de las pruebas. Se comparará el resultado obtenido durante la ejecución con el resultado esperado para esto es necesario ejecutar la prueba mediante el *Framework* de pruebas *PHPUnit* y determinar si se observaron fallas.

Los resultados de las pruebas de *PHPUnit* se muestran en dos de las ventanas del IDE, Resultados de prueba y Salida. La ventana Resultados de la prueba tiene un panel gráfico y un panel corto de texto. La ventana de salida ofrece una versión textual más detallada de la salida.

En la figura 12, se muestra la ventana de resultados cuenta con los siguientes botones.

Ejecutar los test 

Muestra los test que han sido pasados 

Muestra los test fallidos 

Muestra los test pasados pero con errores 

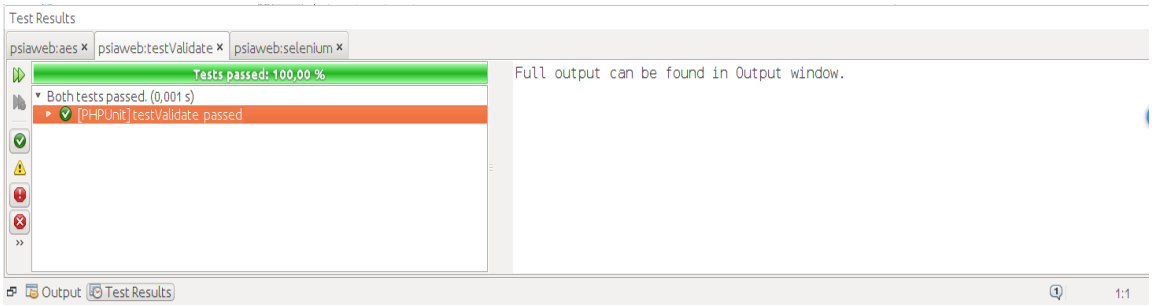

Figura 12 Ventana de resultados PHPUnit
Fuente: Autor del proyecto

En la ventana de salida mostrada en la figura 13, se puede visualizar el resultado detallado del script, donde se especifica cuantos test han sido ejecutados, cuantas aserciones y en caso de que una aserción no se cumpla incluye enlaces a la línea de la clase de prueba que falló.

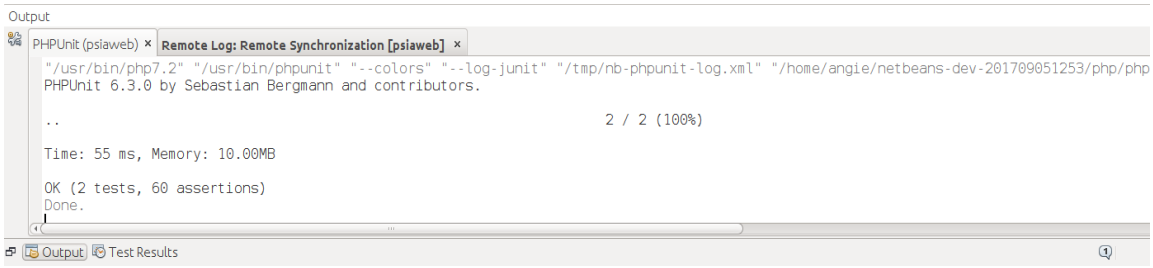

Figura 13 Ventana de Salida PHPUnit

Fuente: Autor del proyecto

En el caso de ejecutar las pruebas con *Selenium*, la ventana de resultados, se encuentra en la parte inferior, se muestran los resultados automáticamente en la ejecución: los mensajes de error, mensajes de información y el progreso. En los mensajes de información proporciona información sobre la ejecución del paso de prueba actual, los mensajes de advertencia brindan información que se encuentra en situaciones especiales y por último los mensajes de error se generan cuando *Selenium IDE* encontró un error. Estos mensajes también se pueden generar si no se cumple una condición especificada por los comandos *Verify* o *Assert*, indicando el tipo de error y una captura de pantalla en el momento que ocurrió el error.

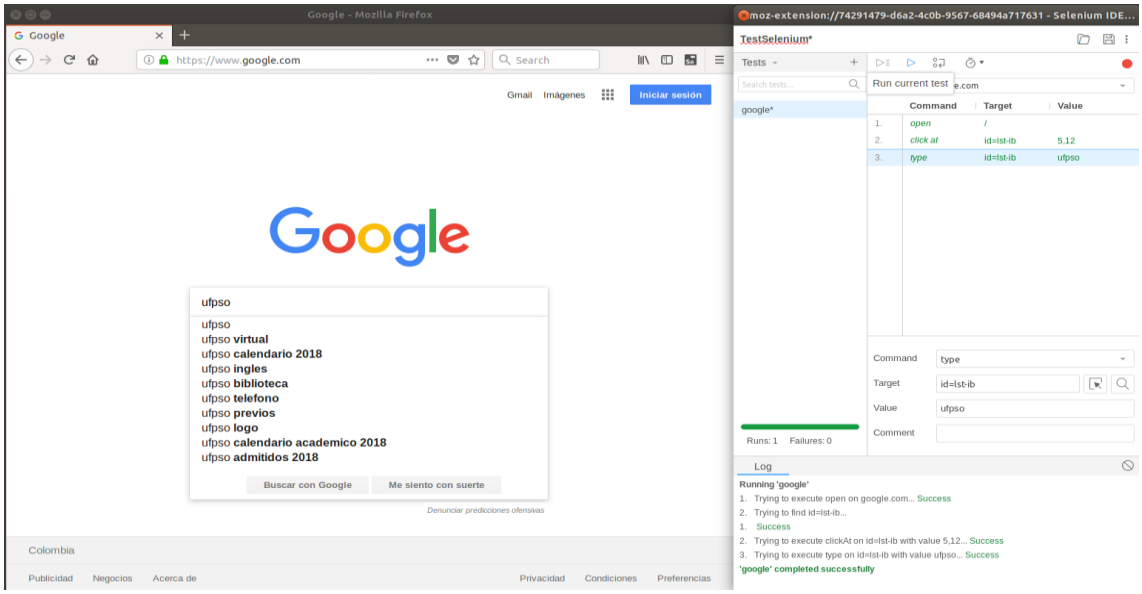

Figura 14 Ventana de resultados de Selenium

Fuente: Autor del proyecto

Cuando se detecte un fallo en el sistema, el mismo debe ser documentado y registrado en el reporte de errores. Una vez que el defecto haya sido corregido, es necesario realizar un re-test que indique o permita confirmar que los defectos fueron solucionados de manera exitosa

Esta información es útil para la depuración de casos de prueba.

5.2.2.4 Evaluar las pruebas realizadas. El objetivo principal de esta actividad es comunicar las conclusiones obtenidas del análisis y medición del proceso sirven para adoptar medidas correctivas, comprobar que el proceso se está aplicando conforme a lo esperado o aprovechar una posible oportunidad que se presente para su mejora.

Criterios de Salida: entre las partes involucradas en el proceso, se define de manera formal, bajo qué condiciones se puede considerar que una actividad de pruebas fue finalizada. Algunos ejemplos de criterios de salida que pueden ser utilizados son: porcentaje de la cobertura de las pruebas esperado, cierto porcentaje de casos exitosos, cobertura de todos los componentes, porcentaje de defectos corregidos, entre otros.

Control de los casos de prueba. Este formato permite realizar un seguimiento de la cantidad de los casos de pruebas y revisar su estado

Tabla 8.
Evaluación de casos de prueba.

ID	NOMBRE	ESTADO	OBSERVACIONES

Fuente: Autor del proyecto

5.3 Implementación

La división de Sistemas es la dependencia encargada de administrar los sistemas de información de la Universidad Francisco de Paula Santander Ocaña, actualmente se encuentra en fase de desarrollo de software como es el caso del Sistema de Información Académica SIA.

A los fines de comprobar el desempeño de la metodología propuesta y tomando como caso de estudio el Sistema de Información Académica SIA, se realizaran las respectivas pruebas de software planificadas en el Plan de pruebas para la mejora de las actividades relacionadas con el proceso de pruebas software

5.3.1 Conocimiento del sistema. La División de Sistemas se enfoca en el desarrollo orientado a la web, para conseguirlo hace uso de tecnologías de software del lado del servidor y del cliente que involucran una combinación de procesos de base de datos que se encuentra instalada en Oracle con el uso de un navegador web, utilizando como lenguaje de programación del lado del servidor, *PHP* bajo el paradigma orientado a objetos, del lado del cliente para brindar diseño e interactividad *HTML, CSS, JavaScript, Jquery y Ajax* que mantiene la comunicación asíncrona con el servidor en segundo plano. De esta forma es posible realizar cambios sobre las páginas sin necesidad de recargarlas, mejorando la interactividad, velocidad y usabilidad en las aplicaciones.

Además el patrón arquitectura Modelo Vista Controlador (MVC) para separar los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos logrando reutilización de código, definir los roles de *Backend* y *Frontend*, separar el código, facilidad en el desarrollo y mantenimiento.

5.3.2 Plan de pruebas específica los procesos de test y de verificación soportando los siguientes objetivos

- Definir los ítems y funcionalidades que serán probados.
- Describir, en términos generales, el enfoque de pruebas a ser usado en ejecución de las mismas.
- Diseñar y ejecutar los casos de pruebas para cada una de las interacciones a probar.
- Identificar defectos y fallas para presentar conclusiones acerca del proceso de pruebas.

5.3.2.1 Glosario

Error: Discrepancia entre el valor calculado y el valor teórico o esperado, con responsabilidad del desarrollador.

Defecto software: Desviación en el valor esperado por una cierta característica. Defecto de calidad.

Fallo: Consecuencia de un error o un defecto software.

5.3.2.2 Planificación

Alcance: En este plan de pruebas para el Sistema de Información Académica SIA, se va a probar el módulo de autenticación del sistema y cambio de contraseña, identificando un set de casos de prueba para cada módulo que verifiquen la funcionalidad del sistema a desarrollar.

Figura 16 Modulo autenticación

Fuente: Autor del proyecto

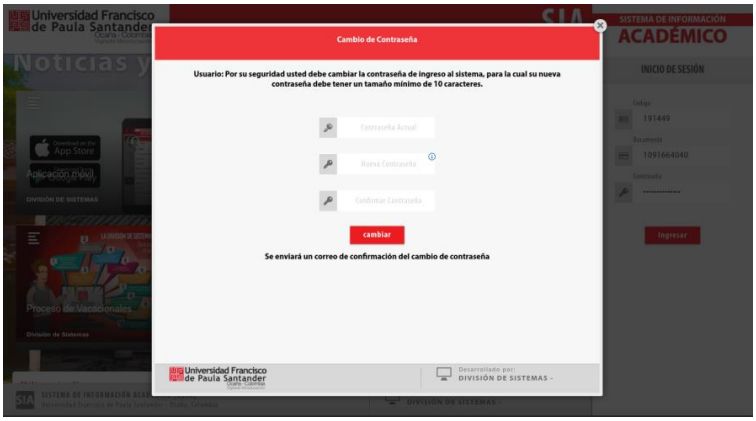

Figura 15 Modulo de cambio de contraseña

Fuente: Autor del proyecto

Los ítems a probar son los siguientes:

Tabla 9

Funcionalidades a probar.

FUNCIONALIDAD	PRIORIDAD	VERSIÓN	
Validación de datos	ALTA	3,1	
Cifrado de datos	MEDIA	3,1	
Autenticación de usuarios LDAP	ALTA	3,1	
Validación de credenciales al SIA	ALTA	3,1	
Cambio de contraseña	MEDIA	3,1	

Fuente: Autor del proyecto

En la tabla 10 se detallan los casos de pruebas que van a ser ejecutados según la funcionalidad a testear.

Tabla 10
Control de casos de prueba.

FUNCIONALIDAD	ID	CASO DE PRUEBA
Validación de datos	CP01	Validate_correct
Validación de datos	CP02	Validate
Validación de datos	CP03	Analizador
Cifrado de datos	CP04	obtener_sal
Cifrado de datos	CP05	Encriptación
Cifrado de datos	CP06	Desencriptación
Autenticación de usuarios LDAP	CP07	Autenticación de usuarios LDAP
Validación de credenciales al SIA	CP08	Validación de credenciales al SIA
Validación de credenciales al SIA	CP09	Login
Validación de credenciales al SIA	CP10	Límite_intentos
Validación de credenciales al SIA	CP11	Datos_Erróneos
Validación de credenciales al SIA	CP12	Usuario_Autenticado
Cambio de contraseña	CP13	Cambio de contraseña
Cambio de contraseña	CP14	No_pass_validate
Cambio de contraseña	CP15	nueva_diff_confirmacion
Cambio de contraseña	CP16	actual_erronea
Cambio de contraseña	CP17	Cambio_exitoso

Fuente: Autor del proyecto

Estrategia de Pruebas: En esta sección se identificarán los distintos tipos de pruebas que serán efectuadas en el módulo autenticación y cambio de contraseña, junto con las técnicas y criterios aplicables a cada uno, y se establecerá la estrategia general de ejecución de pruebas que permita verificar el correcto funcionamiento.

Teniendo en cuenta los ítems a probar se realizarán pruebas unitarias y pruebas funcionales, más conocidas como pruebas de caja negra, las cuales permiten detectar funcionamiento incorrecto o incompleto, errores de interfaz, errores accesos estructuras de datos externas, problemas de rendimiento, errores de inicio y terminación. Su criterio se basa en las interfaces y las especificaciones de los módulos.

Rol y responsabilidades: En este apartado se describen los roles que se han identificado para llevar a cabo las actividades y tareas incluidas en los procesos definidos.

Frontend: Es el encargado de diseñar la interfaz creando la representación visual a utilizar en la aplicación

Tener conocimiento en: *HTML, CSS, JavaScript* y *Selenium IDE*

Backend: Es el encargado de desarrollar el código es decir la parte lógica de la aplicación. Tener conocimiento en: *PHP, Oracle, PHPUnit*

Administrador. El Líder de un proyecto de pruebas tiene conocimiento y experiencia en el diseño, ejecución y reporte de pruebas sobre un producto de software encargándose de diseñar y planificar el plan de pruebas,

Tester: Es el encargado de crear y ejecutar los casos de prueba, asegurando que éstos se ejecutan con los datos requeridos y siguiendo los procedimientos previamente

definidos, así como de analizar e interpretar resultado de las pruebas sobre un producto de software. Poseer conocimiento del sistema o aplicación a probar, además de tener conocimiento de las herramientas de pruebas automáticas como: *PHPUnit* y *Selenium IDE*.

Cronograma de actividades: A continuación se detalla la siguiente tabla que contiene la planificación referida al plan de pruebas

Tabla 11
Cronograma del plan de pruebas.

Actividad	**Responsable**	**Comienzo**	**Término**	**Duración**
Planeación	Angie B.	06/11/2017	29/12/2017	40 días
Diseño de casos de prueba	Angie B.	05/02/2018	06/02/2018	1 día
Especificación de casos de prueba	Angie B.	07/02/2018	23/02/2018	13 días
Ejecución de Pruebas unitarias	Angie B.	26/02/2018	30/03/2018	30 días
Ejecución de Pruebas funcionales	Angie B.	02/04/2018	20/04/2018	15 días
Evaluación	Angie B.	23/04/2018	27/04/2018	4 días
Reporte de errores	Angie B.	23/04/2018	27/04/2018	4 días
Informe de Reportes de Pruebas	Angie B.	30/04/2018	04/05/2018	5 días

Fuente: Autor del proyecto

Adecuación del entorno: Las pruebas se desarrollarán en un computador todo en uno marca HP *ProOne* 400 con las siguientes especificaciones:

- Sistema Operativo: Ubuntu 17.10 64 bits
- Procesador: *Intel core* i7-4765T
- Disco duro: 1 Tb
- Memoria RAM: 8gb

Base de datos: Oracle 11g que reside en el servidor de pruebas.

Instalación de software: Para la automatización de los casos de pruebas, es necesario la utilización de *framework* como *PHPUnit* y *Selenium*.

***PHPUnit*:** Todos los pasos para la realizar la instalación del *framework*, se puede encontrar en el Apéndice G, teniendo en cuenta los requisitos mínimos para que se ejecute normalmente.

***Selenium*:** Todos los pasos para la realizar la instalación del *framework*, se puede encontrar en el Apéndice H, teniendo en cuenta los requisitos mínimos para que se ejecute normalmente.

5.3.2.3 Especificación de casos de prueba. En esta sección se presentan los casos de pruebas generales para ser ejecutados en la herramienta de *PHPUnit* y *Selenium*. Cada uno de los siguientes cuadros está asociado a un caso de prueba y se evalúa el resultado obtenido.

CP01 validate_correct			
Función a probar	validate_correct		
Descripción	La función validate_correct, busca comparar la cantidad de campos enviados con la cantidad de campos que cumplieron con los requisitos de aprobación para determinar si pasó o no el validate (validaciones de usuario)		
Tipo de prueba	Test Unitario		
Precondiciones			
Perfil de usuario	Alumno		
Valores	**Resultado esperado**	**Resultado obtenido**	**Código error**
Campos vacíos	El sistema devuelve un mensaje “Digite los campos”	pass	
$parametros = array("name" => "documento", "value" => "1091671404")	campos = correcto	pass	
$parametros = array("name" => "documento", "value" => "Divisis")	[“msg”=>”datos erroneos”]	pass	

Figura 17 CP01 validate_correct

Fuente: Autor del proyecto

CP02 validate			
Función a probar	Validate		
Descripción	Por cada campo que cumpla con los requisitos de aprobación retorna un uno (1) en caso contrario devuelve un array con los requisitos. (validaciones de usuario)		
Tipo de prueba	Test Unitario		
Precondiciones			
Perfil de usuario	Alumno		
Valores	**Resultado esperado**	**Resultado obtenido**	**Código error**
Campos vacíos	El sistema devuelve un mensaje “Digite los campos”	pass	
$parametros = ["documento"=> "1091671404"]	correcto=1	pass	
["document"=> "1091671404"]	Campos incorrectos	El key del arreglo no existe	val01
["document"=> "cedula"]	[msg=>”Datos erróneos”]	pass	

Figura 18 CP02 validate

Fuente: Autor del proyecto

CP03 analizador				
Función a probar		analizador		
Descripción		El analizador, se encarga de revisar si cada campo cumple con los requisitos (tamaño, condición)		
Tipo de prueba		Test Unitario		
Precondiciones				
Perfil de usuario		Alumno		
Condición	**Valores**	**Resultado esperado**	**Resultado obtenido**	**Cód. error**
Campos vacíos		El sistema devuelve un mensaje “Digite los campos”	Pass	
Código, máximo 6 números (V)	código=’035693’	$correcto=1	Pass	
Código, máximo 6 números	código=’0356937’	El máximo 6 caracteres	Pass	
Estado, 1 carácter (V)	estado=’A’	$correcto=1	Pass	
estado, 1 carácter	estado=’’	El mínimo 1 caracteres	Pass	
estado, 1 carácter	estado=’ABC’	El mínimo 1 caracteres	Pass	
Materia, máx. 7 y mín. 6 caracteres (V)	materia=’191110’	$correcto=1	Pass	
Materia, máximo 7 y mínimo 6 caracteres (V)	materia=’191110A’	$correcto=1	Pass	
Materia, máximo 7 y	materia=’19’	El mínimo 6 caracteres	Pass	

mínimo 6 caracteres				
continua				
Materia, máximo 7 y mínimo 6 caracteres	materia='191110AB'	El máximo 7 caracteres	El mensaje no corresponde	Val 02
Nombre, solo acepta letras. (V)	Nombre= 'Angie'	$correcto=1	Pass	
Nombre, solo acepta letras.	Nombre= '34569A#'	Sólo letras (A-Za-z)	Pass	
Teléfono solo números (V)	teléfono='5695889'	$correcto=1	Pass	
Teléfono solo números	teléfono='569-5889'	Solo números	Pass	
Teléfono solo números	teléfono='telefono5889'	Solo números	Pass	
Empresa, acepta letras ó números (V)	empresa='envíos 447'	correcto=1	Pass	
Empresa, acepta letras ó números (V)	empresa='ufpso'	Letras ó números correcto=1	Pass	
Empresa, acepta letras ó números	empresa='giro 360°'	Letras (A-Za-z) o números(0-9)	Pass	
Correo, debe contener identificador de usuario seguido del signo @ y por ultimo el dominio (V)	correo='alballesterosc @ufpso.edu.co"	$correcto=1	Pass	
Correo, debe contener identificador de usuario seguido del signo @ y por último el dominio	correo='alballesterosc @"	correo	Pass	
Correo, debe contener identificador de usuario seguido del signo @ y por último el dominio	correo='alballesterosc.ufpso.edu.co"	correo	Pass	
Correo, debe contener identificador de usuario seguido del signo @ y por último el dominio	correo='alballesterosc"	correo	Pass	
	dirección='calle 14 N° 14-49'	Letras(A-Za-z). Números (0-9). Símbolos(#.-)	Pass	
Nota, números flotantes separados por (.)	Nota=0.35	Números (0-9). Símbolos(.)	Pass	
Nota, números flotantes separados por (.)	Nota =0,35	Números (0-9). Símbolos(.)	Pass	
Nota, números flotantes separados por (.)	Nota =35ª		Pass	
Formato para fechas, dia, mes y año	Fecha	DD-MM-AA o DD/MM/AA	Pass	

Figura 19 CP02 validate

Fuente: Autor del proyecto

CP04 obtener_sal	
Función a probar	Encriptación del Login
Descripción	Crea una cadena de caracteres que será usada como sal para cifrar los datos.
Tipo de prueba	Test Unitario
Resultado esperado	Generar la SAL
Resultado obtenido	Pass
Precondiciones	
Perfil de usuario	Alumno
Código de error	

Figura 20 CP04 obtener_sal

Fuente: Autor del proyecto

CP05 encriptación	
Función a probar	Encriptación del Login
Descripción	Realización de la encriptación de los campos (código, documento, contraseña) del módulo Autenticación.
Tipo de prueba	Test Unitario
Resultado esperado	Datos encriptados
Resultado obtenido	Pass
Precondiciones	Haber generado la SAL
Perfil de usuario	Alumno
Código de error	

Figura 21 CP05 encriptación

Fuente: Autor del proyecto

CP06 Desencriptación	
Función a probar	Desencriptación del Login
Descripción	Realización de la desencriptación de los campos (código, documento, contraseña) del módulo Autenticación.
Tipo de prueba	Test Unitario
Resultado esperado	Datos desencriptados
Resultado obtenido	Pass
Precondiciones	
Perfil de usuario	Alumno
Código de error	RE03

Figura 22 CP06 Desencriptación

Fuente: Autor del proyecto

CP07 UsuarioLdap				
Función a probar	UsuarioLdap			
Descripción	Revisar si el usuario se encuentra registrado en el sistema de autenticación única LDAP.			
Tipo de prueba	Test Unitario			
Precondiciones	El alumno debe estar ingresado en la tabla Datos_Personales			
Perfil de usuario	Alumno			
Valores	**Resultado esperado**		**Resultado obtenido**	**Código de error**
Campos vacíos	El sistema devuelve un mensaje “Digite los campos”		pass	
Documento = contraseña	Cambio de contraseña, variable $result=CA		pass	
Campos erróneos: Documento=’10916568999’ contraseña=’Angie12354’	Campos erróneos, variable $result=CE		pass	
Campos erróneos: Documento=’1091656999’ contraseña=’Angie12354’	Campos erróneos, variable $result=CE		pass	
Campos validos: Documento=’1091656999’ contraseña=’Angie*5698+’	$result=datos del usuario registrados en el ldap	pass		

Figura 23 CP07 UsuarioLdap

Fuente: Autor del proyecto

CP08 UsuarioBD			
Función a probar	UsuarioBD		
Descripción			
Tipo de prueba	Test Unitario		
Precondiciones	El alumno debe estar autenticado en el LDAP		
Perfil de usuario	Alumno		
Valores	**Resultado esperado**	**Resultado obtenido**	**Cód error**
Campos vacíos	El sistema devuelve un mensaje “Digite los campos”	pass	
Documento=’109165999’ codigo=’’	variable $result=””	pass	
Campos erróneos: Documento=’109165999’ codigo=’191919’	variable $result=””	pass	
Campos validos: Documento=’1091656999’ codigo=’190619’	Usuario valido.	pass	

Figura 24 CP08 UsuarioBD

Fuente: Autor del proyecto

CP09 Inicio de sesión.			
Función a probar	Login		
Descripción	Autenticación de los usuarios al sistema		
Tipo de prueba	Test Unitario		
Precondiciones	El alumno debe estar ingresado en la tabla Datos_Personales		
Perfil de usuario	Alumno		
Valores	**Resultado esperado**	**Resultado obtenido**	**Código de error**
Campos vacíos	El sistema devuelve un mensaje "Digite los campos"	Pass	
Campos erróneos	El sistema devuelve un mensaje ""	Pass	
Usuario no registrado en LPDA	Mensaje: Campos incorrectos	Pass	
Usuario no activo en Base de datos	Mensaje usuario no activo	Pass	
Campos válidos	El sistema devuelve un mensaje "Usuario Autenticado"	Pass	

Figura 25 CP09 Inicio de sesión

Fuente: Autor del proyecto

CP10 limite_logeo				
Función a probar		limite_logeo		
Descripción		Controla la cantidad de intentos de ingreso que un usuario puede realizar en un determinado intervalo de tiempo		
Tipo de prueba		Test Funcional		
Perfil de usuario		Alumno		
Pasos:	**Valores**	**Resultado esperado**	**Resultado obtenido**	**Cód. error**
Digitar los campos solicitados y presionar el botón ingresar	Codigo='191975' documento='109166804' password='Divisis*2018'	El sistema devuelve un mensaje "Error de Autenticación" y el botón se deshabilita por 5seg	pass	
Digitar los campos solicitados y presionar el botón ingresar	Codigo='191975' documento='109166804' password='Divisis*2018'	El sistema devuelve un mensaje "Error de Autenticación" y el botón se deshabilita por 10seg	pass	
Digitar los campos solicitados y presionar el botón ingresar	Codigo='191975' documento='1091668488' password='Divisis*2018'	El sistema devuelve un mensaje "Error de Autenticación" y el botón se deshabilita por 15seg	pass	
Digitar los campos solicitados y presionar el botón ingresar	Codigo='19165' documento='109166804' password='Divisis*2018'	El sistema devuelve un mensaje "Ha superado el límite de intentos"	pass	

Figura 26 CP10 limite_logeo

Fuente: Autor del proyecto

CP11 datos_erroneos				
Función a probar		Autenticacion		
Descripción		Error de autenticación		
Tipo de prueba		Test Funcional		
Precondiciones				
Perfil de usuario		Alumno		
Pasos:	**Valores**	**Resultado esperado**	**Resultado obtenido**	**Cód. error**
Digitar los campos solicitados y presionar el botón ingresar	Código=’19’ documento=’109166804’ password=’Divisis*2018’	El sistema devuelve un mensaje “Datos Erróneos” y coloca la clase error al código	pass	
Digitar los campos solicitados y presionar el botón ingresar	Codigo=’191975’ documento=’1’ password=’Divisis*2018’	El sistema devuelve un mensaje “Datos Erróneos” y coloca la clase error el campo documento	pass	
Digitar los campos solicitados y presionar el botón ingresar	Codigo=’191975’ documento=’10916680488’ password=’Divisis*2018’	El sistema devuelve un mensaje “Datos Erróneos” y coloca la clase error el campo *password*	pass	
Digitar los campos solicitados y presionar el botón ingresar	Codigo=’codigo’ documento=’cedula’ password=’password’	El sistema devuelve un mensaje “Datos Erróneos” y coloca la clase error el campo *password*	pass	

Figura 27 CP11 datos_erroneos

Fuente: Autor del proyecto

CP12 Usuario_autenticado				
Función a probar		Autenticacion		
Descripción		Usuario autenticado		
Tipo de prueba		Test Funcional		
Precondiciones				
Perfil de usuario		Alumno		
Pasos:	**Valores**	**Resultado esperado**	**Resultado obtenido**	**Cód. error**
Digitar los campos solicitados y presionar el botón ingresar	Código=’191919’ documento=’1091668048’ *password*=’Divisis*2018’	El sistema devuelve un mensaje “Usuario Autenticado” y redirige a la página principal.	pass	

Figura 28 CP12 Usuario_autenticado

Fuente: Autor del proyecto

CP13 inicio_cambio_pass				
Función a probar		Autenticación		
Descripción		Cambio de contraseña		
Tipo de prueba		Test Funcional		
Precondiciones				
Perfil de usuario		Alumno		
Pasos:	**Valores**	**Resultado esperado**	**Resultado obtenido**	**Código error**
Digitar los campos solicitados y presionar el botón ingresar	Código=’191919’ documento=’1091668048’ *password*=’Divisis*2018’	El sistema devuelve un mensaje “Cambio de contraseña” y muestra la flotante para realizar dicho cambio.	pass	

Figura 29 CP13 Cambio de contraseña

Fuente: Autor del proyecto

CP14 no_pass_validate				
Función a probar		Autenticacion		
Descripción		Cambio de contraseña, los datos ingresados no cumplen los requisitos (no_pass_validate)		
Tipo de prueba		Test Funcional		
Precondiciones				
Perfil de usuario		Alumno		
Pasos:	**Valores**	**Resultado esperado**	**Resultado obtenido**	**Código error**
Digitar los campos solicitados y presionar el botón cambiar	antigua=’1’ nueva=’1’ confirmacion=’1	El sistema devuelve un mensaje “La contraseña nueva debe contener mayúscula, minúscula, números y mínimo 10 caracteres” y muestra la flotante para realizar dicho cambio.	pass	

Figura 30 CP14 no_pass_validate

Fuente: Autor del proyecto

CP15 nueva_diff_confirmacion				
Función a probar		Autenticación		
Descripción		Cambio de contraseña, la nueva contraseña y la confirmación son diferentes		
Tipo de prueba		Test Funcional		
Precondiciones				
Perfil de usuario		Alumno		
Pasos:	**Valores**	**Resultado esperado**	**Resultado obtenido**	**Código error**
Digitar los campos solicitados y presionar el botón cambiar	antigua=’10916658’ nueva=’Divisis*2018’ conf_new=’Divisis/2018’	El sistema devuelve un mensaje “La nueva contraseña y la confirmación deben ser iguales”, los input de nueva y confirmación se le agrega la clase error.	pass	

Figura 31 CP15 Contraseñas diferentes

Fuente: Autor del proyecto

CP16 pass_actual_erronea				
Función a probar		Autenticación		
Descripción		Cambio de contraseña, la contraseña actual no corresponde		
Tipo de prueba		Test Funcional		
Precondiciones				
Perfil de usuario		Alumno		
Pasos:	**Valores**	**Resultado esperado**	**Resultado obtenido**	**Código de error**
Digitar los campos solicitados y presionar el botón cambiar	antigua=’109166589’ nueva=’Divisis*2018’ conf_new=’Divisis*2018’	El sistema devuelve un mensaje “Contraseña actual errónea”, al input de la contraseña actual se le agrega la clase error.	pass	

Figura 32 CP16 Contraseña actual errónea.

Fuente: Autor del proyecto

CP17 Cambio_exitoso				
Función a probar		Autenticacion		
Descripción		Cambio de contraseña exitoso		
Tipo de prueba		Test Funcional		
Precondiciones				
Perfil de usuario		Alumno		
Pasos:	**Valores**	**Resultado esperado**	**Resultado obtenido**	**Código error**
Digitar los campos solicitados y presionar el botón cambiar	antigua=’109168Divisis’ nueva=’Divis*2018’ conf_new =’Divis*2018’	El sistema devuelve un mensaje “Cambio de contraseña exitoso”, y se recarga la página	pass	

Figura 33 CP17 Cambio_exitoso

Fuente: Autor del proyecto

5.3.2.4 Reporte de errores

Código	RE01
Descripción	Debido a que la clave del arreglo no existe genera un error, “Error al afirmar que una matriz tiene la clave 'document'.”
Gravedad	Medio
Archivo adjunto	

Figura 34 RE01.

Fuente: Autor del proyecto

Código	RE02
Descripción	No se tiene en cuenta si el tamaño de la variable es mayor o menor a la exigida por lo tanto el mensaje no corresponde con la información ingresada. Mensaje esperado: Máx. 5 caracteres, Mensaje obtenido: Mín. 5 caracteres,
Gravedad	Bajo
Archivo adjunto	

Figura 35 RE02.

Fuente: Autor del proyecto

Código	RE03
Descripción	La generación del SAL estaba contenida entre comillas, y al momento de la desencriptacion no concordaba con la que había sido generada.
Gravedad	Medio
Archivo adjunto	

Figura 36 RE03

Fuente: Autor del proyecto

5.3.2.5 Evaluación. El plan de pruebas de software diseñado para evaluar las funcionalidades del módulo de autenticación y cambio de contraseña del Sistema de información Académica, donde se abarcaron las etapas: alcance de las pruebas, cronograma de actividades, planificación y diseño de los casos de pruebas, ejecución de los mismos y por último se realizó la evaluación donde se aprobará el plan de pruebas con un 100% de las pruebas ejecutadas pero con un 90% de aceptación. Esto quiere decir el 90% de las pruebas deben ser exitosas y sin errores. El restante 10% pueden existir errores medios o bajos, pero no graves. En caso no cumplir con el nivel exigido, el proyecto se rechaza completo y se deberá reformular un nuevo plan de pruebas, corrigiendo los errores encontrados.

Relación de los casos de prueba con su respectivo criterio de aceptación.

Tabla 12 *Casos de pruebas*

ID	NOMBRE	CRITERIO	OBSERVACIONES
CP01	Validate_correct	Aprobado	
CP02	Validate	Aprobado	Reporte de errores RE01
CP03	Analizador	Aprobado	Reporte de errores RE02
CP04	obtener_sal	Aprobado	
CP05	Encriptación	Aprobado	
CP06	Desencriptación	Aprobado	Reporte de errores RE03
CP07	Autenticación de usuarios LDAP	Aprobado	
CP08	Validación de credenciales al SIA	Aprobado	
CP09	Login	Aprobado	
CP10	Límite_intentos	Aprobado	
CP11	Datos_Erróneos	Aprobado	
CP12	Usuario_Autenticado	Aprobado	
CP13	Cambio de contraseña	Aprobado	
CP14	No_pass_validate	Aprobado	
CP15	nueva_diff_confirmacion	Aprobado	
CP16	actual_erronea	Aprobado	
CP17	Cambio_exitoso	Aprobado	

Fuente: Autor del proyecto

En el proceso de pruebas se ejecutaron el 100% de los casos de pruebas planteados y se evidenció como se observa en la tabla 12, los casos de pruebas fueron aprobados en su totalidad y algunos poseen observaciones acerca de los errores encontrados y están documentados en el reporte de errores, los cuales fueron solucionados inmediatamente.

Esta información será almacenada y utilizada en pruebas futuras, además, permitirá realizar una mejora continua.

CONCLUSIONES

La metodología *Test Driven Development* TDD requiere que los desarrolladores piensen en el software en términos de unidades pequeñas que pueden escribirse y probarse de manera independiente para posteriormente integrarse. Además la refactorización continua de nuestro código es una de las fases de TDD más importante, lo que conduce a un código modularizado, extensible, legible y mantenible.

El uso de *Scrum* y TDD brindó al proyecto una manera eficiente de desarrollarlo, al ser las pruebas lo que se diseñe e implemente primero otorga la información necesaria sobre lo que su código debe de hacer, siendo importante tener una buena comprensión del sistema o aplicación que se quiere construir desde el principio, teniendo claro los requisitos y especificaciones tan pronto como sea posible, lo que genera una mayor eficiencia y confianza en la codificación.

TDD se trata de una curva de aprendizaje que requiere habilidad, madurez y tiempo, sobre todo cuando los desarrolladores están acostumbrados a dejar de ultimo la fase de pruebas, por lo tanto al comienzo los proyectos tienden a demorarse un poco más pero el esfuerzo es importante para la prevención de defectos.

Para fomentar la adopción de TDD se requieren herramientas de soporte para generar casos de prueba como es la utilización de *PHPUnit* y *Selenium*, haciendo que sean más fáciles de ejecutar cuando sea necesario.

En cuanto a la aplicación del modelo de pruebas al SIA, se puede limitar la mayoría de los defectos a las primeras fases del desarrollo de software, mediante el diseño continuo de pruebas y el aseguramiento de calidad, (Beizer, 2003) dice que el acto de diseñar pruebas es de las técnicas más efectivas de prevención de defectos. Este factor tiene un gran efecto positivo en el presupuesto del proyecto; hay menos defectos y son menos costosos de arreglar.

El plan de pruebas aporta la documentación detallada que servirá para la reproducción de los casos de pruebas en cualquier momento del desarrollo.

RECOMENDACIÓN

En este proyecto nos enfocamos en las pruebas unitarias teniendo en cuenta los principios de TDD, pero no es el único tipo de prueba; hay muchos otros que evalúan diferentes aspectos de nuestras aplicaciones y no deben omitirse debido a que agregan valor y se debe investigar cómo implementarlos en su práctica de desarrollo de software.

También se hace necesario contar con una guía para la fase de pruebas de los requisitos no funcionales permitan validar aspectos como: usabilidad, confiabilidad, rendimiento y portabilidad; disminuyendo los riesgos de despliegue a producción.

En cuanto a las pruebas de integración se debe contar con una base de datos de pruebas para no afectarla

REFERENCIAS

Aucancela, C. P. (2011). *Desarrollo dirigido por test (TDD) utilizando el framework JUnit en un sistema web de asignación de aulas de los laboratorios generales de computación de la Espe, aplicando la metodología agile unified process (AUP).* Sangolqui: Escuela Politecnica del Ejercito.

Beck, K. (2002). *Test Driven Development: By Example.* AddisonWesley Professional.

Beck, K. (2004). *JUnit Pocket Guide.* O´Reilly.

Beck, K. A. (2000). *Una explicación de la programación extrema. Aceptar el cambio.*

Beizer, B. (2003). *Software Testing Techniques.* Dreamtech.

Bergmann, S. (2005). *PHPUnit Pocket Guide* . O´Reilly.

Bernal, C. A. (2010). *Metodología de la investigacion.* Bogota, Colombia: Pearson.

Bhat, T. y. (2006). *Evaluating the efficacy of test-driven development:*. New York, NY, USA: En Proceedings of the 2006 ACM/IEEE.

Canales C, M. (2006). Metodología de la investigacion social. Santiago: LOM ediciones.

Contributors, S. (2004). *Seleniumhq.org.* Obtenido de https://www.seleniumhq.org/about/contributors.jsp

Corbin, J. &. (2008). *Basics of qualitative research: techniques and procedures for developing grounded.* Thounsand Oaks: Sage.

Crispin, L., & Gregory, J. (2008). *Agile Testing.* Estados Unidos.

Dieste, O. (2015). Efectividad del Test-Driven Development: Un Experimento Replicado. *Latinoamericana de Ingeniería de Software*.

Elaine, J. a. (2000). *Experience with performance testing of software systems: Issues, approach and case study. IEEE transactions on software engineering.*

Fowler, M. (1999). *Refactoring : improving the design of existing code.* . Boston: Addison-Wesley Professional.

Goicochea, A. (2015). *Una evaluación experimental para comparar la calidad de un software aplicando o no tdd dentro del modelo cascada.* San Miguel: Pontificia Universidad Católica del Perú.

Gotel, O. a. (1994). *An analysis of the requirements traceability problem.* Colorado Springs.

Hernández, R., Baptista, P., & Fernández , C. (2010). *Metodología de la investigación.* México D.F: McGraw-Hill.

IEC, I. /. (1998). *9126: Quality in use metrics.*

IEEE/ISO. (2018). *Estándar para documentación de pruebas de software.*

ISO, A. (1990). *IEEE Standard Glossary of Software Engineering Terminology* .

Jenkins, N. (2008). *A Software Testing Primer: An Introduction to Software Testing.* USA: Creative Commons.

Johaness, & Shanon. (1976). *Systems Simulation: The Art and Science.* Obtenido de http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4309432

Jurado, C. (2010). *Diseño Ágil con TDD.* www.iExpertos.com.

McConnell, S. (1996). *Rapid development.*

Moncada, J. N. (2014). *Proceso de Pruebas Unitarias Bajo Entornos de Desarrollo Ágiles: Un Estudio Sistemático.* Cali, Colombia: Universidad ICESI.

Myklebust, T. &. (2015). *mportant considerations when applying other models than the Waterfall/V-model when developing software according to IEC 61508.*

Nonaka, I. T. (1986). The New New Product Development Game.

Pavón, J. (2008). *Patron Modelo Vista Controlador.* Madrid.

Piatini, M. &. (2003). *Calidad en el desarrollo y mantenimiento del software.* RA-MA.

Pressman, R. (2010). *Ingenieria de software, un enfoque practico.* McGraw-Hill.

Schwaber, K. y. (21 de 01 de 2017). *Scrum Guides.* Obtenido de La Guía de Scrum: http://www.scrumguides.org/

Serna, E. (2013). *Prueba funcional del software: un proceso de verificación constante.* Medellín Colombia.: Fondo Editorial ITM.

Sommervile, I. (2005). *Ingenieria del Software.* Madrid: Pearson Educacion S.A.

Sundaram, A. (s.f.). *Smartbear.* Obtenido de https://smartbear.com

Tuya, J. R. (2007). *Técnicas cuantitativas para la gestión en la ingeniería del software.* Oleiros, La Coruña: Netbiblo.

Vaca, P. M. (2014). *Estudio de Test-Driven Development en el proceso de desarrollo.* Cordoba, Colombia: Workshop de Investigadores en Ciencias de la Computación.

Varela, L. (2012). *INTRODUCCION AL SOFTWARE TESTING.*

Villamizar, K. T. (2015). Mejora de historias de usuario y casos de prueba de metodologías ágiles con base en TDD. *Cuaderno Activa,*.

Wheeler, D. A. (2011). *The most important software innovations.*

Williams, L. M. (2003). *Test-driven development as defect reduction practice. En Proceedings of the 14th International Symposium.* Washington, DC, USA.

Apéndice

Apéndice A. Entrevista semiestructurada.

Objetivo: Conocer el proceso actual de las pruebas de software.

1. ¿Cuál es la metodología empleada en el desarrollo de software?

2. ¿En sus proyectos de desarrollo en que momento realiza las pruebas?

3. ¿Puede describir las actividades de prueba?

4. ¿Los casos de pruebas manuales pueden ser automatizados?

5. ¿Cree usted que la automatización de pruebas puede mejorar la eficacia de la prueba manuales?

6. ¿Domina alguna herramienta de automatización de prueba?

Apéndice B. Plantilla para casos de pruebas.

A continuación, se expone la plantilla con cada uno de los ítems con base a los propuestos por (IEEE/ISO, 2018)

Identificador y título.			
Función a probar			
Descripción			
Tipo de prueba			
Precondiciones			
Perfil de usuario			
Valores	**Resultado esperado**	**Resultado obtenido**	**Código de error**

Figura 37 Formato para casos de prueba.
Fuente: Autor del proyecto

Identificador y título. Este identificador debe ser único para cada caso de prueba comenzando por las siglas CP seguido del número consecutivo, además un título breve y simple de manera que resulte claro que se quiere probar.

Función a probar: Describe el título de la característica o funcionalidad que se está probando.

Descripción: Escribe un resumen del caso de prueba que proporciona una visión general de lo que hace el caso, Esto debe permitir que alguien sin conocimiento previo tenga una comprensión clara de lo que se desea hacer.

Tipo de prueba: No todos los productos de software requieren la aplicación de todos los tipos de pruebas que existen. Es estrictamente necesario determinar cuáles son aplicables al proyecto en evaluación.

Los posibles tipos de prueba a aplicar son: pruebas unitarias, pruebas de integración, pruebas de stress, pruebas de rendimiento, pruebas de carga, pruebas funcionales, pruebas de usabilidad, pruebas de regresión, entre otros.

Precondiciones. Conjunto de situaciones previas que se deben cumplir para que pueda iniciar y ejecutar un caso prueba.

Criterios de entrada. Identificar los insumos necesarios para cada caso de prueba a partir de ellas se extraen las clases de equivalencia tanto para datos validos como no válidos.

Según Serna (2013) las consideraciones a tener en cuenta son:

- Por cada rango de valores, se especifica una clase válida y dos no válidas. Ej.: Si la entrada requiere un rango entre 1-12, una clase valida corresponde a un dígito

entre ese rango definido y las clases no validas son: cuando es menor que 1, y otra cuando es mayor a 12

- Si se especifica un número de valores, se creará una clase válida y dos no válidas. Ej.: En el programa del triángulo, define un caso con tres valores (1, 3,3), otro con dos o menos valores (1,3) y otro con cuatro o más valores (1, 2, 3,4).
- Si se trata de una entrada booleana, existen 2 clases: verdadero y falso.
- Si se requiere un valor existen en un conjunto, aparecen 2 clases de equivalencia: pertenece o no pertenece al conjunto. Ej.: el tipo de un vehículo pueden ser [moto, carro, camión], selecciona uno valido [moto], y otro invalido [tren].
- Si se especifica un conjunto de valores admitidos, y el programa trata de forma distinta cada uno de ellos, se crea una clase válida por cada valor, y una no válida. Ej.: Tres tipos de inmuebles (válidas) apartamentos, fincas, locales comerciales (No válida) lote

Permite identificar otros aspectos más específicos que se le deben probar a cada funcionalidad de la aplicación:

- Campos obligatorios: se debe validar que en los formularios sean ingresados todos aquellos campos que sean necesarios y si los dejan en blanco, mostrar el mensaje de alerta.

- Campos opcionales: verificar si es posible que el usuario deje el campo en blanco

- Formato de los datos: Tamaño permitido en los campos y Tipos de datos permitidos

Resultado esperado. Datos de salida que se espera que produzca un caso de prueba

Resultado obtenido. Datos de salida que se producen al ejecutar la prueba. Si el resultado esperado es igual al obtenido, entonces el caso de prueba se considera exitoso y coloca la palabra “pass”, en caso contrario cuando el resultado no cumple con lo definido para este ítem colocamos “no pass”.

Código del error. Sí se generó un error se coloca la referencia del reporte de errores es decir el identificador único.

Apéndice C. Formato para el reporte de errores.

Para el tratamiento de errores,

Código	
Descripción	
Gravedad	
Archivo adjunto	

Figura 38 Formato para reporte de errores.
Fuente: Autor del proyecto

Código: Es muy importante para poder referirse al defecto en los informes, siendo un identificador único comenzado por las siglas "RE" seguido del número consecutivo.

Descripción: es un resumen de alto nivel del defecto y la falla observada. Debe ser lo más destacado del defecto, ya que esto es lo que los desarrolladores o los revisores ven por primera vez en el informe de errores.

Gravedad: Los niveles de gravedad no están estandarizados, pero se suelen utilizar los siguientes:

- Alto: Error que afecta seriamente a la funcionalidad de la aplicación (por ejemplo, una función no funciona en otro navegador o SO).

- Medio: Impacto de nivel medio (por ejemplo, algunos los mensajes de la aplicación son incorrectos).
- Bajo: Error de bajo impacto (por ejemplo, problemas estéticos en la interfaz de usuario o errores tipográficos).

Adjuntos: Cualquier evidencia de la falla debe ser capturada y enviada junto con el informe de defectos. Esta es una explicación visual de la descripción del defecto y ayuda al revisor y al desarrollador a comprender mejor el defecto.

Apéndice D. Script de pruebas utilizando *PhpUnit*

Objetos simulados (*Mocks*): Simulan el comportamiento de los objetos que van a ser probados al momento de realizar las pruebas. La instancia a ser probada llamará a los *Mock Objects* para ejecutar el método que se vaya a probar realizando la prueba correspondiente.

```php
<?php
use PHPUnit\Framework\TestCase;
class testuserLdap extends TestCase {
    private $array;
    function testAutenticar() {
        $user = $this->getMockBuilder("Model")
                ->setMethods(array('User'))
                ->getMock();
        $user->expects($this->once())
                ->method('User')
                ->with($this->array)
                ->willReturn(3);
        $this->assertEquals(3, $user->User($this->array));
    }
}
```

Figura 39 Objetos simulados o mocks

Fuente: Autor del proyecto

Como invocar métodos privados o protegidos. Primero se debe realizar la instancia a la clase, luego se accede al método protegido a través de la función *getPrivateMethod*, la que recibe por parámetros el nombre de la clase y el nombre del método.

$method2 = $this->getPrivateMethod(nombre_clase, nombre_metodo);

Por ultimo llamar a invokeArgs que invoca a la función y se pasan los argumentos.

$result = $method2->invokeArgs($this->instancia, argumentos);

En la Figura 40 se muestra un ejemplo, donde *Model* es el nombre de la clase y *user* es el método al cual deseamos acceder.

```php
<?php

require 'model.php';

use PHPUnit\Framework\TestCase;

class testuser extends TestCase {

    private $array;
    private $user;

    public function getPrivateMethod($className, $methodName) {
        $reflector = new ReflectionClass($className);
        $method = $reflector->getMethod($methodName);
        $method->setAccessible(true);

        return $method;
    }

    function testUsuario() {
        $this->user = new Model();
        $method2 = $this->getPrivateMethod('Model', 'user');
        $result = $method2->invokeArgs($this->user, $this->array);
    }
```

Figura 40 Acceder a métodos protegidos

Fuente: Autor del proyecto

***Dataprovider*:** proveedor de datos, es un *array* de *arrays* que contiene los datos que constituyen los parámetros para el método de prueba. Para usar el método proveedor de datos se especifica con la anotación *@dataProvider*.

```php
<?php

require 'model.php';

use PHPUnit\Framework\TestCase;

class testuser extends TestCase {

    private $array;
    private $user;

    function __construct() {
        $this->user = new Model();
    }

    /**
     * @dataProvider dataProvider
     */
    function testUsuario($doc, $pass, $exp) {
        $result = $this->user->getUsuario($doc, $pass);
        $this->assertEquals($exp, $result);
    }

    public function dataProvider() {
        return array(
            array('109168888', 'pdsad89*', 'CE'),
            array('109168888', 'ufpso_2018*', 'ES'),
            array('1091656899', 'divisis*2018-1', 'CA')
        );
    }
```

Figura 41 DataProvider o Proveedor de datos

Fuente: Autor del proyecto

Para el ejemplo anterior visualizado en la figura 41, el *testUsuario* fue ejecutado 3 veces, una por cada *array* del proveedor de datos.

Dependencias (*depends*): Usando la anotación *@depends* para expresar dependencias entre los métodos de prueba, permiten la devolución de una instancia y la pasan como argumentos a los métodos dependientes.

```php
<?php

require 'model.php';
use PHPUnit\Framework\TestCase;

class testuser extends TestCase {

    private $array;
    private $user;

    function __construct() {
        $this->user = new Model();
    }

    public function dataProvider() {
        return array(
            array('109168888', 'pdsad89*', 'CE'),
            array('109168888', 'ufpso_2018*', 'ES'),
            array('1091656899', 'divisis*2018-1', 'CA')
        );
    }
    /**
     * @dataProvider dataProvider
     */
    function testUsuario($doc, $pass, $exp) {
        $result = $this->user->getUsuario($doc, $pass);
        $this->assertEquals($exp, $result);
        return $result;
    }
    /**
     * @depends testUsuario
     */
    function testAutenticar($result) {
        $exp = ["CE" => "Datos erroneos", "ES" => "Usuario Autenticado", "CA" => "Cambio de contraseña"];
        $reg = $this->user->autenticar($result);
        $this->assertContains($exp[$result], $reg);
    }
```

Figura 42 Dependencias o Depends

Fuente: Autor del proyecto

En el ejemplo de la figura 42, el método de prueba *TestAutenticar* depende del resultado del *TestUsuario* para ejecutarse ya que recibe como parámetros lo que retorna el test dependiente.

Apéndice E. Script de pruebas utilizando Selenium

En el caso de escribir test con *Selenium* IDE, vamos abrir la extensión del navegador, dando doble clic sobre el icono de *Selenium* localizado en la parte superior derecha.

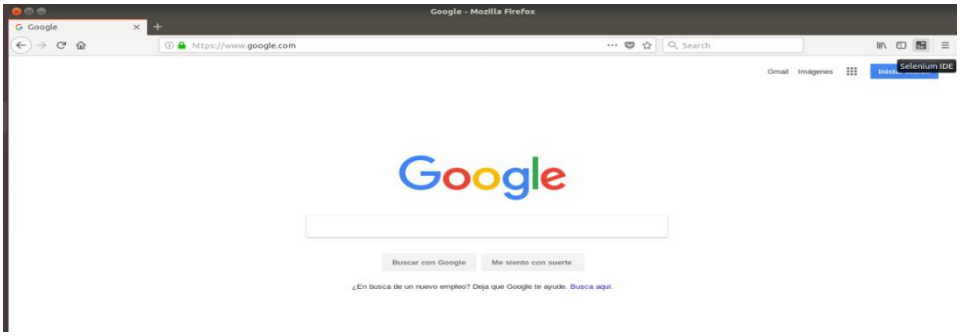

Figura 43 Extensión en el navegador de Selenium IDE

Fuente: Autor del proyecto

Una vez abierta la interfaz de *Selenium* IDE, podemos comenzar a grabar nuestros test.

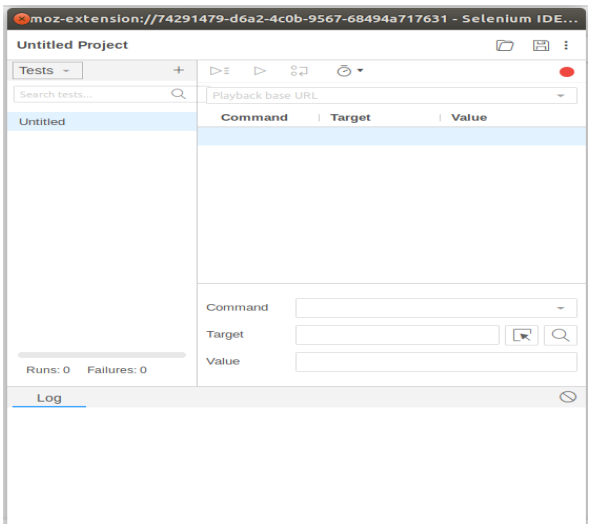

Figura 44 Interfaz de Selenium IDE

Fuente: Autor del proyecto

Para agregar un nuevo test es necesario presionar el botón + y en la ventana flotante colocar el nombre del test.

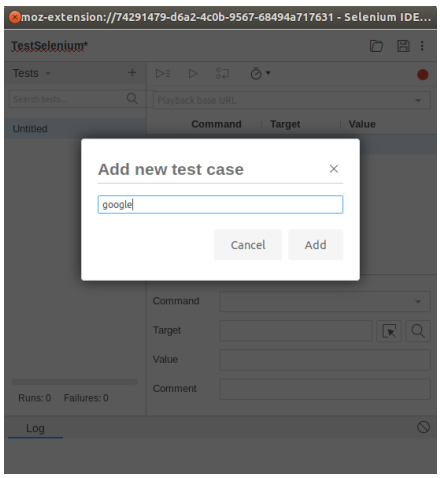

Figura 45 Añadir un test con Selenium IDE

Fuente: Autor del proyecto

Presionando el botón rojo, ubicado en la parte derecha, comienza a grabar todos los eventos realizados en el navegador como por ejemplo abrir la página google.com, colocar el cuadro de búsqueda UFPSO y luego volver a presionar el botón rojo.

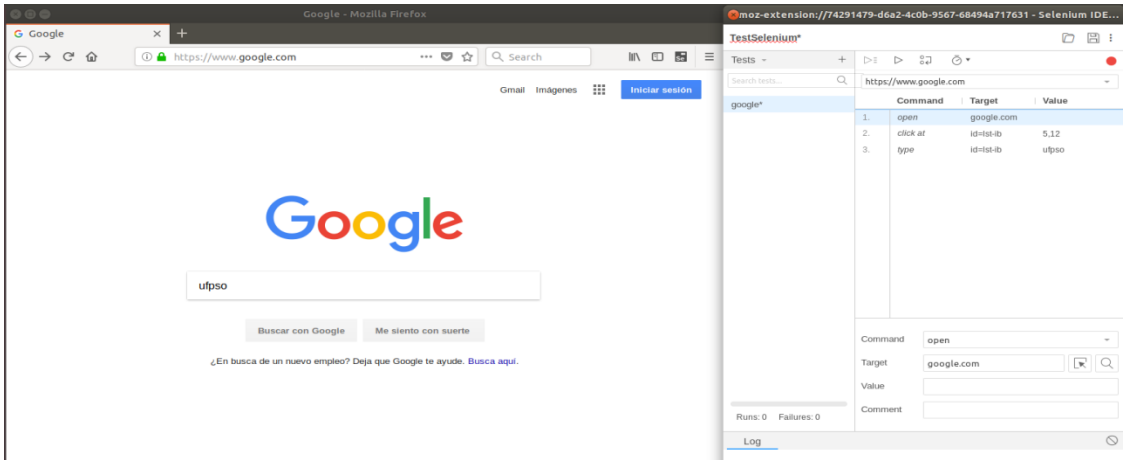

Figura 46 Grabar test con Selenium IDE

Fuente: Autor del proyecto

Como podemos visualizar en la figura 46, en la parte derecha se encuentran todos los comandos que han sido grabados cada vez que se efectuó una operación en el navegador.

Apéndice F. Refactorización.

Explicación de los síntomas de refactorización con algunos ejemplos donde se busca facilitar el aprendizaje y obtener un código más limpio y reutilizable.

Métodos de composición

- Extraer Método: cuanto más largo es el método más difícil es de entender y además puede tener fragmentos que se pueda agrupar, para llevar a cabo la refactorización es necesario crear un nuevo método, Copia el fragmento de código relevante. Elimine el fragmento de su ubicación anterior y en su lugar realice una llamada para el nuevo método.

  Entregando un código más legible, menor duplicidad, aísla el código independiente

Tabla 13
Extraer método.

Problema	Solución

function printOwing() {
  $this->printBanner();
 	
  //print details
  print("name: " . $this->name);
  print("amount " . $this->getOutstanding());
}
	function printOwing() {
	  $this->printBanner();
	  $this->printDetails($this->getOutstanding());
	}
	
	function printDetails ($outstanding) {
	  print("name: " . $this->name);
	  print("amount " . $outstanding);}

Fuente: Autor del proyecto

- Extraer Variable: La razón principal para extraer variables es hacer que una expresión compleja sea más comprensible dividiéndola en sus partes intermedias, entregando un código más legible. Lo que debes realizar es: Inserta una nueva línea antes de la expresión relevante y declara allí una nueva variable. Asigna parte de la expresión compleja a esta variable.

  Reemplace esa parte de la expresión con la nueva variable en todas las partes donde se estaba utilizando

Tabla 14
Extraer variable.

Problema	Solución

if(($base * $altura/2) < 10){
return $base * $altura/2;
}
	$areaTriangulo=$base * $altura/2;
	if($areaTriangulo < 10){
	return $areaTriangulo;
	}

Fuente: Autor del proyecto

- Variable temporal: si se está almacenando algún valor simple y que no se utiliza en ninguna otra parte, se puede omitir la declaración de la variable y utilizar solo la expresión que tiene asignada.

Tabla 15
Variable temporal.

Problema	Solución
$mensaje= “Registro exitoso” return $mensaje;	Return “Registro exitoso”;

Fuente: Autor del proyecto

Mover características entre objetos

- Mover método: Cuando un método se usa más en otra clase que en su propia clase debe mover un método a una clase que contiene la mayoría de los datos utilizados para reducir o eliminar la dependencia de la clase para esto es necesario crear un nuevo método en la clase que usa más el método, luego mueva el código del método anterior a allí. Convierta el código del método original en una referencia al nuevo método en la otra clase o elimínelo por completo.

- Mover campo: cuando un campo se usa más en otra clase que en su propia clase por regla general se debe mover al mismo lugar que los métodos que lo usan o de lo contrario, donde se encuentran la mayoría de estos métodos.

- Extraer clase: una clase tiene más de una funcionalidad y ya no es tan clara de entender puesto que tiene muchas responsabilidades y se han ido agregando nuevos métodos y campos, antes de comenzar a refactorizar se decide como quiere dividir dichas responsabilidades, cree una nueva clase para contener las funcionalidades relevantes y por ultimo una relación unidireccional entre las dos clases, esto nos ayudará a mantener el cumplimiento del Principio de Responsabilidad Individual.

Organizando datos

- Datos duplicados observados: Desea tener múltiples vistas de interfaz para los mismos datos, se debe tener separados la lógica de la GUI dividiendo la responsabilidad entre estas, logrando así que puedan existir los roles de *Frontend* y *Backend*.

- Reemplazar número mágico con constante simbólica: si existe un valor con algún significado es necesario declararla para poder entender más fácil el funcionamiento, además si ese valor llegará a cambiar con el tiempo solo es necesario modificar la constante y no entrar a modificar cada lugar donde se esté llamando.

Tabla 16
Reemplazar número mágico con constante simbólica

Problema	Solución

function Potencial($masa, $altura) {
  return $masa * $altura * 9.81;
}
	define("gravedad", 9.81);
	function Potencial($masa, $ altura) {
	  return $masa * $altura * gravedad;
	}

Fuente: Autor del proyecto

Simplificando Expresiones Condicionales

- Consolidar Expresión Condicional: Tiene múltiples condicionales que conducen al mismo resultado. Se busca extraer el condicional a un método

separado para mayor claridad, y se pueden unir utilizando AND para condicionales anidados y OR para condicionales consecutivos.

Tabla 17

Consolidar Expresión Condicional

Problema	Solución

function disabilityAmount() {
if ($this->seniority < 2) return 0;
if ($this->monthsDisabled > 12) return 0;
if ($this->isPartTime) return 0;
}
	function disabilityAmount() {
	if ($this->isNotEligableForDisability()) return 0;
	}

Fuente: Autor del proyecto

- Consolidar Fragmentos Duplicados dentro de Condicionales: El código idéntico se encuentra dentro de todas las ramas de un condicional, se puede refactorizar moviendo el código fuera del condicional de la siguiente manera: Si está al principio de las ramas condicionales, mueva el código a un lugar antes del condicional y si el código se ejecuta al final de las ramas, colóquelo después del condicional.

Tabla 18

Fragmentos duplicados dentro de condicionales

Problema	Solución

if (descuento()) {
$total = $precio * 0.95;
detalleFactura();
}
else {
$total = $precio * 0.98;
detalleFactura();
}
	if (descuento ()){
	$total = $precio * 0.95;
	}else{
	$total = $precio * 0.98;
	}
	detalleFactura();

Fuente: Autor del proyecto

- Reemplazar condicional anidado: cuando hay un gran grupo de condicionales anidados se hace más complejo verificar el flujo de ejecución.

Tabla 19

Reemplazar condicional anidado

Problema	Solución

public function obtenerPago() {
  if ($this->Pensionado)
  $result = $this->PagoPensionado();
  else {
  if ($this->OPS)
  $result = $this->PagoOPS();
  else {
  if ($this->Retirado)
  $result = $this->PagoRetirado();
  else
  $result = $this->PagoNormal();
  }
  }
  return $result;
}
	public function obtenerPago() {
	  if ($this-> Pensionado){
	  return $this-> PagoPensionado();
	}
	  if ($this->OPS){
	  return $this-> PagoOPS ();
	}
	  if ($this-> Retirado){
	  return $this->PagoRetirado ();
	}
	
	  return $this->PagoNormal();
	}

Fuente: Autor del proyecto

Apéndice G. Instalación de *PHPUnit*

Para realizar la instalación se hace a través de los siguientes comandos desde la terminal:

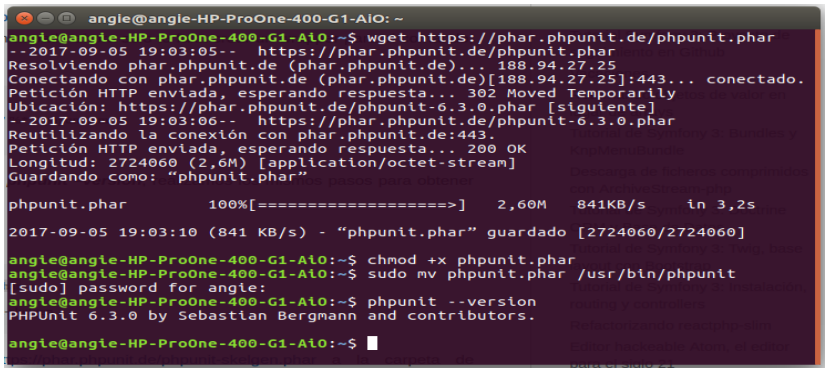

Figura 47 Instalación de PHPUnit

Fuente: Autor del proyecto

Obtenga *PHPUnit-skelgen.phar* ejecutando los comandos a continuación, es necesario instalarlo ya que estamos trabajando con *NetBeans IDE*.

Figura 48 Instalación de PHPUnit skeletor

Fuente: Autor del proyecto.

Configuración de *PHPUnit* con la *IDE* de *NetBeans*: Ingresamos en el menú de herramientas de la parte superior de *NetBeans* en la opción "*Tools*" y luego escogemos en la lista desplegable "*options*". Se muestra una interfaz como la visualizada en la Figura 10, ingresando a la opción con el logo de *PHP* => *Framework* & *Tools* y por último en *PHPUnit*.

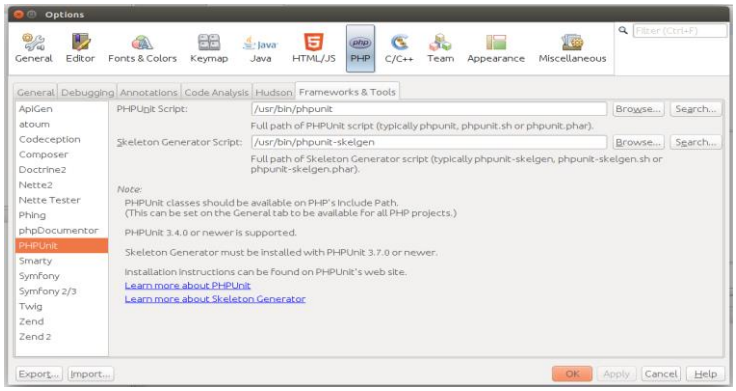

Figura 49 Configuración de NetBeans

Fuente: Autor del proyecto

Ahí escogeremos las rutas donde se encuentra instalado *PHPUnit* y *PHPUnit-skeletor*, luego presionamos el botón *apply*

Apéndice H. Instalación de *Selenium*

Ingresamos en la página oficial de *Selenium* (Contributors, 2004) y descarga la última versión y luego ejecutamos el siguiente comando para realizar la instalación desde la terminal.

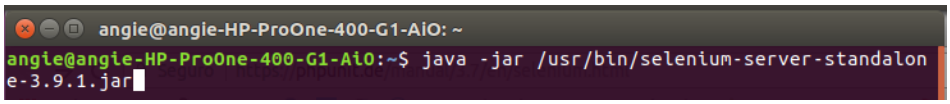

Figura 50 Instalación de Selenium
Fuente: Autor del proyecto

Editamos el archivo de configuración .bashrc

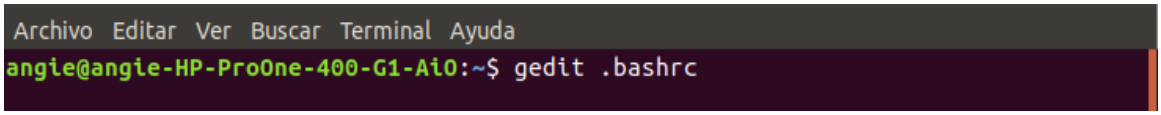

Figura 51 Editar .bashrc
Fuente: Autor del proyecto

Colocamos el alias de Selenium en el archivo de configuración.

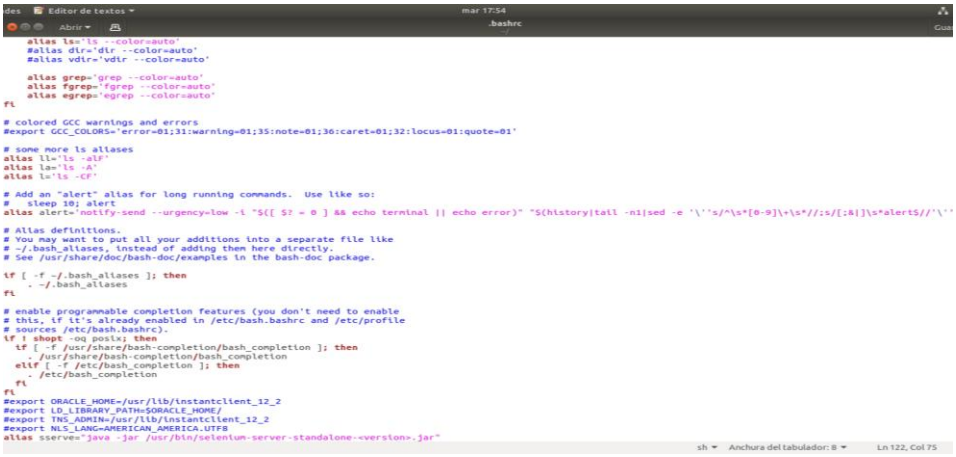

Figura 52 Configurando el alias de Selenium

Fuente: Autor de proyecto

Instalamos la librería de *Selenium* para *PHPUnit*

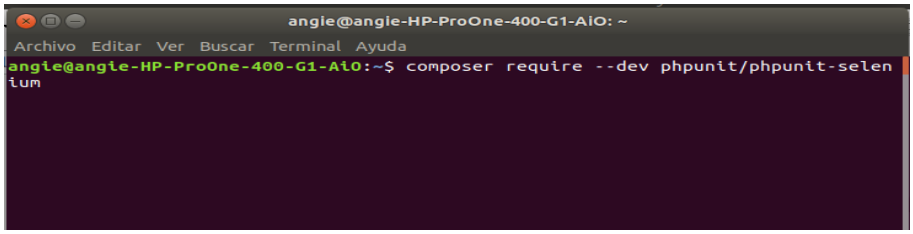

Figura 53 Selenium y PHPUnit

Fuente: Autor de proyecto

Después de realizada la instalación del software *PHPUnit* y *Selenium*, ingresamos en *NetBeans* para configurar la carpeta donde se va a guardar los test, para esto ingresamos en configuración del proyecto y en el menú de la parte izquierda seleccionamos el ítem de "*Testing*", escogemos el folder que hemos preparado para las pruebas de *PHPUnit*

En el caso de Selenium ingresamos en configuración del proyecto y en el menú de la parte izquierda seleccionamos el ítem de “Selenium Testing”, luego escogemos la carpeta y presionamos el botón “ok” para guardar las configuraciones.

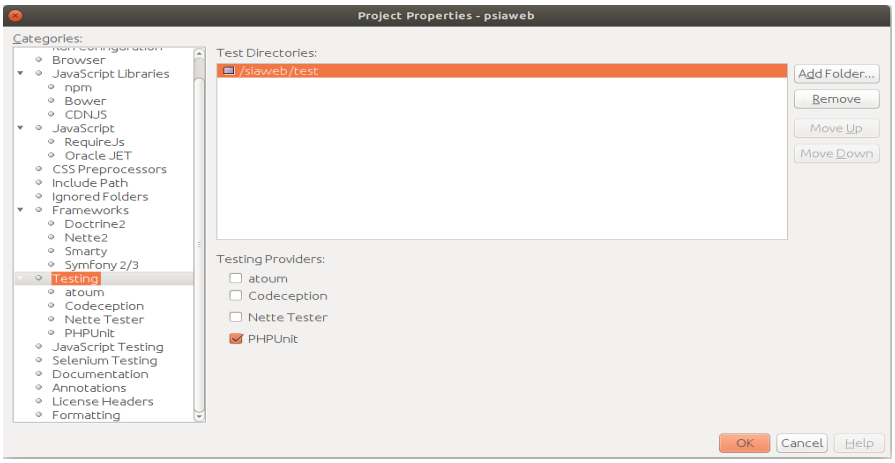

Figura 54 Carpeta de PHPUnit

Fuente: Autor del proyecto

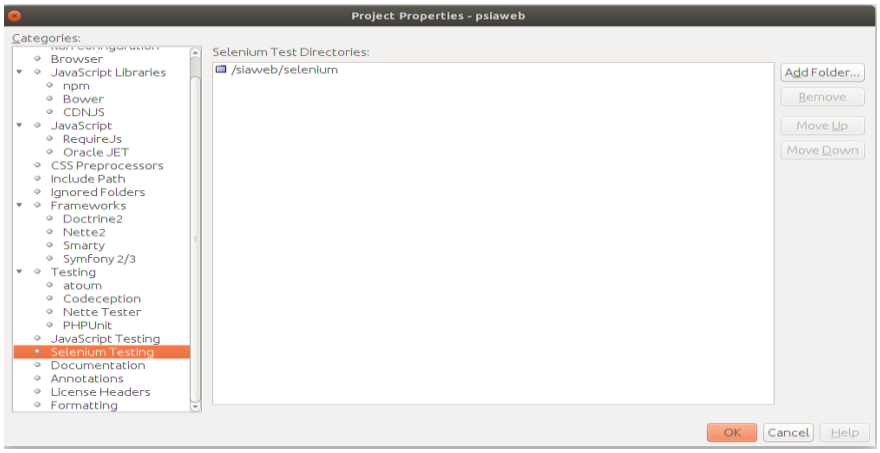

Figura 55 Carpeta de Selenium.

Fuente: Autor del proyecto