UNIVERSIDAD DE LAS CIENCIAS INFORMÁTICAS

Facultad 3

Trabajo de Diploma para optar por el título de Ingeniero en Ciencias Informáticas

Título: Procedimiento para la ejecución de las pruebas de unidad a los componentes de la Capa de presentación del sistema CEDRUX.

AUTOR:

Erick Córdova Dorta.

TUTOR:

Ing. Iliannis Pupo Leyva.

Ing. Tte David Martínez Alarcón

Ciudad Habana, Enero de 2011

Año 53 de la Revolución

"La calidad nunca es un accidente; siempre es el resultado de un esfuerzo de la inteligencia"

John Ruskin

Declaro ser el autor de la presente tesis y reconozco a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste firmamos la presente a los 22 días del mes de junio del año 2011.

_____ _____

Erick Cordova Dorta **Ing. Iliannis Pupo Leyva.**

_____

Ing. Tte David Martínez Alarcón.

Agradecimientos

Quisiera agradecer a las personas que hicieron posible que llegara hasta este momento tan especial, mi familia, mi papá, mi hermano, mi abuela, y todos los que me apoyaron y me dijeron desde niño que no me rindiera, que yo era el inteligente de la familia. Pero más que a nadie quiero agradecer a mi madre que es la única culpable de que hoy yo esté aquí, ella que nunca me dejó rendirme y siempre estaba diciéndome que si dios me había puesto una luz en el camino que la aprovechara, que no me fuera por lo más fácil que al final se verían los resultados, y tenía toda la razón.

Agradecer a mi novia que muchas veces que tuvo que leerse el documento revisando ortografía, y tantas me vio recitar lo que tenía que decir que fácilmente podría defender por mí.

A mis compañeros los viejos y los nuevos, que durante 5 largos años he molestado ayudándome cada vez que los necesite.

A mi tutora y mi cotutor que sin ellos no hubiera podido realizar este trabajo.

A todas las personas que me quieren y apoyan les agradezco por estar siempre hay.

Dedicatorias

A mi mamá que siempre me dijo que su único deseo era que este día llegara y si llegaba y ella no se encontraba en este mundo por favor que fuera a su tumba y le pusiera el título en ella, ya ves no tuvo que ser así.
A mi papá, mi hermano, mi abuela y toda mi familia que siempre creyeron en mí.
A mi novia que tanto apoyo me ha brindado en estos dos años, y en la realización de este trabajo.
A todos mis amigos que han estado siempre a mi lado apoyándome en estos 5 años.

Resumen

Las pruebas son el principal proceso en la elaboración de un producto para medir la calidad del mismo, por esto la realización correcta de las mismas es vital para que el software salga optimizado y listo para satisfacer las necesidades del cliente final.
En la Universidad de Ciencias Informáticas (UCI), específicamente en la facultad 3, se desarrolla dentro del Centro para la Informatización de Gestión de Entidades (CEIGE) un sistema de planificación de recursos empresariales denominado CEDRUX.
El presente trabajo se centra en la aplicación de un procedimiento para realizar pruebas de unidad a la capa de presentación de dicho sistema con el objetivo de lograr elevar los niveles de calidad y poder registrar la documentación con los resultados de cada una de las pruebas de unidad realizadas al producto. Para ello se utiliza una de las prácticas propuestas por la metodología Extreme Programe (XP), denominada Desarrollo guiado por pruebas o Test-driven development (TDD) para lograr una mejor organización en el trabajo y las librerías de pruebas del framework YUI insertadas dentro del framework Extjs para realizar las pruebas unitarias automáticamente.

Palabras Claves
Pruebas de unidad, procedimiento de pruebas, calidad de software.

Índice

Introducción

La realización de un sistema de software es un proceso que lleva consigo una serie de actividades en las cuales el fallo, debido a la propia naturaleza humana, se encuentra implícito. Estos errores pueden producirse desde el mismo comienzo del proceso; en el que los objetivos pueden ser tomados o especificados de forma errónea; hasta pasos posteriores de diseño e implementación. Debido a esta incapacidad de los humanos de poder desarrollar de forma perfecta, la creación de un software ha de ir de la mano de un conjunto de actividades que garanticen la calidad del mismo durante toda su fase de vida [3].

La calidad de los sistemas informáticos se ha convertido en uno de los principales objetivos estratégicos de las organizaciones tanto productoras como consumidoras de software, las cuales priorizando el aseguramiento de la calidad, asegurarán un mejor funcionamiento de estos.

La obtención de un software con calidad implica la utilización de metodologías o procedimientos estándares para el análisis, diseño, programación y prueba, que permitan uniformar la filosofía de trabajo y que a la vez eleven la productividad, tanto para la labor de desarrollo como para el control de la calidad del software.

Toda producción de software debe estar enfocada a los usuarios finales, y los procedimientos de calidad en los mismos deben tener como único fin, la satisfacción de estos usuarios. Un software que no cuenta con pruebas enfocadas en sus usuarios finales cierra su proyección de ventas en un gran porcentaje lo que implica grandes pérdidas de dinero, por lo que es preferible realizar una inversión en pruebas que garanticen la calidad a tiempo para poder lograr los resultados esperados.

Cuba se ha ido desarrollando la producción de software y actualmente la vinculación de todas sus ramas (económicas, políticas y sociales) con el mundo informático, son de gran importancia para el estado cubano. La producción de software no sólo trae beneficios desde el punto de vista del desarrollo de sistemas para el uso interno, sino también es una manera de introducirse en el mercado a escala mundial aprovechando su perspectiva económica. Este proceso requiere de una mejora constante del producto que garantice la calidad del mismo contando con varias entidades productoras de software dentro de ellas la Universidad de las Ciencias Informáticas (UCI).

Actualmente en la UCI, se manifiesta inmadurez en la producción del software, a razón de que la mayoría de los proyectos son creados con estudiantes inexpertos en desarrollo de aplicaciones. Son adiestrados en corto tiempo y adquieren

conocimientos básicos del contenido necesario para comenzar a desarrollar en una etapa inmediata. Con estas circunstancias, en muchos casos, pasa a un plano degradado el entrenamiento en técnicas de pruebas de software, la preparación en contenidos relacionados con buenas prácticas de programación, uso de patrones y otras cuestiones técnicas, que permiten escribir un código que sea factible en su prueba.

Esta situación también se ve dentro de la facultad 3 de la universidad y más específicamente en el Centro para la Informatización de Gestión de Entidades (CEIGE). El centro se encuentra desarrollando un sistema de Panificación de Recursos Empresariales, o ERP (por sus siglas en inglés, Enterprise Resource Planning) al que se le ha denominado CEDRUX, en el mismo, los desarrolladores emplean mucho tiempo concibiendo la depuración de sus códigos, de forma manual, muchas veces detectan errores que implican cambios de manera tardía, provocando así un retroceso en lo que se había empleado tiempo para implementar y demoras en los planes del desarrollo del proyecto.

El centro para velar por la calidad de sus proyectos crea el Grupo de Gestión de Calidad, en él se evalúa la calidad del sistema de ERP de igual forma que para los demás sistemas que se han desarrollado en la UCI, es decir, solo se le realizan pruebas a las funcionalidades del sistema y a las interfaces una vez terminado cada subsistema. La evaluación de las pruebas unitarias no se realiza a no ser de forma manual por los mismos desarrolladores.

Con una revisión sujeta a mecanismos intuitivos y criterios humanos existe la probabilidad mínima de calidad requerida del software.

A partir de esta situación surge el siguiente **problema:** La ejecución de las pruebas de unidad a los componentes de la capa de presentación del sistema CEDRUX se realizan de forma manual, lo que afecta la calidad del producto final.

Por lo cual, el **objeto de estudio** lo constituyen las pruebas de Caja Blanca y **el campo de acción** son las pruebas de unidad a la capa de presentación del sistema CEDRUX.

El **Objetivo** que se persigue con esta investigación es desarrollar un procedimiento para la ejecución de pruebas de unidad a los componentes de la capa de presentación del sistema CEDRUX que permita aumentar la calidad del producto y garantice que esté acorde con los requisitos del cliente.

Para el logro de este objetivo general se plantearon los siguientes **objetivos específicos:**

1. Revisar la documentación relacionada con las pruebas de unidad.
2. Revisar la documentación referente a los métodos de planificación, medición y control de pruebas.
3. Revisar la información existente sobre los métodos de documentación de pruebas.
4. Revisar la información existente sobre la capa de presentación del sistema CEDRUX.
5. Definir un procedimiento para la ejecución de pruebas de unidad a los componentes de la capa de presentación del sistema CEDRUX.
6. Definir una herramienta para la automatización de las pruebas de unidad en la capa de presentación del sistema CEDRUX.
7. Validar la solución propuesta.

Por lo que se proponen las siguientes tareas a realizar**:**

✓ Analizar los principales temas relacionados con la capa de presentación del sistema CEDRUX y los problemas en el código que afectan el buen desarrollo de esta.

✓ Analizar los principales temas relacionados con las pruebas de Caja Blanca que se realizan a la capa de presentación.

✓ Analizar los principales temas relacionados con las pruebas unitarias que se realizan a la capa de presentación.

✓ Proponer un procedimiento para las pruebas de Caja Blanca a los componentes de la capa de presentación del sistema CEDRUX.

✓ Proponer herramientas para la automatización de las pruebas de unidad.

✓ Validar el procedimiento.

Se parte de la siguiente **idea a defender:** con la creación de un procedimiento para la ejecución de las pruebas de unidad a los componentes de la capa de presentación del sistema CEDRUX, aumentará la calidad del producto final.

El presente trabajo está estructurado en 3 capítulos.

Capítulo 1: “Fundamentación teórica” comprende todo lo relacionado al objeto de estudio: conceptos importantes, relación directa entre las pruebas y la calidad de software, un detallado argumento acerca de las pruebas de unidad y un extracto de todo lo relacionado a las pruebas de Caja Blanca a la capa de presentación.

Capítulo 2: “Propuesta de procedimiento” es donde se obtienen la(s) herramienta(s) seleccionadas para automatizar las pruebas, así como la definición del procedimiento.

Capítulo 3: “Validación del procedimiento” es donde se aplica el procedimiento definido, aplicándolo, y obteniendo resultados.

Capítulo 1 Fundamentación teórica.

1.1 Introducción

Una de las problemáticas más grandes que se presentan en el desarrollo de sistemas de software es el aseguramiento de la calidad. La obtención de un software con calidad implica la utilización de metodologías o procedimientos estándares, para el análisis, diseño, programación y prueba del software que permitan uniformar la filosofía de trabajo, en aras de lograr una mayor confiabilidad y facilidad de prueba, a la vez que eleven la productividad. Para determinar dicho nivel de calidad se deben efectuar pruebas de software, estas son una vía óptima para la detección y prevención de errores durante todos los ciclos de desarrollo del mismo, permitiendo comprobar el grado de cumplimiento respecto a las especificaciones iniciales del sistema. La enorme cantidad de tiempo y esfuerzo que requiere la fase de pruebas hoy en día se calcula que representa más de la mitad del valor total del producto, pues requiere un tiempo similar al de la programación, y el coste de corregir los errores crece exponencialmente según avanza el proyecto.

En este capítulo se presenta el estado del arte de las pruebas de unidad, teniendo en cuenta los objetivos, son analizados algunos temas como: la relación que existe entre las pruebas y la calidad de un software, los objetivos y estrategias a seguir a la hora de aplicar las mismas, los tipos de pruebas que existen, métodos, niveles, herramientas que se utilizan para realizar pruebas, las metodologías posibles a usar y cómo se aplican.

1.2 Capa de presentación o Interfaz de usuario.

La creación de las interfaces de usuario ha sido un área del desarrollo de software que ha evolucionado dramáticamente a partir de la década de los setenta. La interfaz de usuario es el vínculo entre el usuario y el programa de computadora. Una interfaz es un conjunto de comandos o menús a través de los cuales el usuario se comunica con el programa [1].

Representa una de las partes más importantes de cualquier programa, determinado que tan fácilmente es posible que el programa haga lo que el usuario quiere hacer. Un programa muy poderoso con una interfaz pobremente elaborada tiene poco valor para un usuario no experto.

El éxito en la elaboración de una interfaz de usuario requiere gran dedicación, pues generalmente las interfaces son grandes, complejas y difíciles de implementar, depurar y modificar. Hoy en día las interfaces de manipulación directa (también llamadas interfaces gráficas de usuario, GUI por sus siglas en inglés) son

prácticamente universales. Las interfaces que utilizan ventanas, íconos y menús se han convertido en estándar en los materiales computacionales [2].

1.3 Calidad del software

Entre los problemas que afronta actualmente la esfera de la computación se encuentra la calidad del software. Desde la década del setenta, este tema es motivo de preocupación para especialistas, ingenieros, investigadores y comercializadores de software, debido a que los clientes se vuelven más selectivos y comienzan a rechazar productos poco fiables o que realmente no den respuesta a sus necesidades.

Existen diversas definiciones de Calidad del Software enunciadas por varias compañías entre ellas la Organización Internacional para la Estandarización (ISO), que proponen normas y estándares para llevar a cabo una correcta práctica que garantice la buena ejecución de los procesos, dentro de las cuales pueden citarse:

“La calidad del software es el grado con el que un sistema componente o proceso cumple los requerimientos especificados y las necesidades o expectativas del cliente o usuario” [4].

“El conjunto de características de una entidad que le confieren su aptitud para satisfacer las necesidades expresadas y las implícitas” [5].

Por lo que a esta investigación respecta, la calidad del software se define como:

Concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos, con los estándares de desarrollo explícitamente documentados, y con las características implícitas que se espera de todo software desarrollado profesionalmente [3].

Para obtener un software con calidad se requiere de la utilización de metodologías y procedimientos estándares para el desarrollo de los requisitos, el análisis, el diseño, la implementación y, finalmente, las pruebas del software, que son el elemento fundamental para el logro de la calidad de cualquier sistema o parte integrante de este. Las pruebas permiten nivelar la estrategia de trabajo en aras de lograr una mayor confiabilidad y facilidad de las soluciones.

1.3.1 Control de la calidad

El control de la calidad son aquellas técnicas y actividades de carácter operativo, utilizadas para satisfacer los requisitos relativos a la calidad, centrados en dos objetivos fundamentales:

- ✓ Mantener bajo control un proceso.
- ✓ Eliminar las causas de los defectos en las diferentes fases del ciclo de vida [5].

El control de la calidad está regido por una serie de técnicas dinámicas y estáticas que incluyen métodos para la evaluación del software. El control de la calidad encierra un bucle de realimentación del proceso que creó el producto. La combinación de medición y realimentación permite afinar el proceso cuando los productos de trabajo creados fallan al cumplir sus especificaciones. Este enfoque ve el control de calidad como parte del proceso de fabricación [3].

En general son las actividades para evaluar la calidad de los productos desarrollados. Las actividades de control de calidad pueden ser manuales, completamente automáticas o una combinación de herramientas automáticas e interacción humana. Un concepto clave del control de calidad es que se hayan definido todos los productos y las especificaciones mensurables en las que se puedan comparar los resultados de cada proceso.

1.3.2 Garantía de la calidad

La Garantía de Calidad del Software, como algunos autores acostumbran llamarle, se define como una actividad que posibilita asegurar y proteger todo el proceso de ingeniería de software.

Es un conjunto de procedimientos, técnicas y herramientas, aplicados por profesionales, durante el ciclo de desarrollo de un producto, para asegurar que el producto satisface o excede los estándares o niveles de calidad preestablecidos [6].

Es la guía de los preceptos, de gestión y de las disciplinas de diseño para el espacio tecnológico y la aplicación de la ingeniería del software [4].

La garantía de calidad consiste en la auditoría y las funciones de información de la gestión. El objetivo de la garantía de calidad es proporcionar la gestión para informar los datos necesarios sobre la calidad del producto, por lo que se va adquiriendo una visión más profunda y segura de que la calidad del producto está cumpliendo sus objetivos.

1.4 Pruebas de software

En todo proceso de desarrollo de aplicaciones es indispensable la presencia de un proceso de Pruebas de Software que coexista y se integre con este primero para garantizar así el buen funcionamiento y la calidad del producto final. Para lograr lo antes expuesto se debe partir del concepto de que las mismas desempeñan un papel fundamental en esta disciplina.

“proceso de ejecutar un programa con el fin de encontrar errores”. [8]

El nombre “prueba”, además de la actividad de probar, se puede utilizar para designar “un conjunto de casos y procedimientos de prueba” [10].

“La prueba es el flujo de trabajo fundamental cuyo propósito general es comprobar el resultado de la implementación mediante las pruebas de cada construcción, incluyendo tanto construcciones internas como intermedias, así como las versiones finales del sistema que van a ser entregadas a terceras partes” [11]
Toda prueba de software desempeña un papel fundamental en el desarrollo de cualquier tipo de aplicación, pero si se estudia la mejor forma de hacerlo, siguiendo los pasos de acuerdo con los especialistas en el tema, se incrementan las posibilidades de que esta llegue a un feliz término y arroje resultados más cercanos a los esperados, permitiendo así, realizar a posteriori un mejor análisis de la situación.

Kaner, Falk y Nguyen sugieren los siguientes atributos de una buena prueba [3]:

- ✓ Una buena prueba no debe ser redundante. El tiempo y los recursos para las pruebas son limitados. No hay motivo para realizar una prueba que tiene el mismo propósito que otra. Todas las pruebas deberían tener un propósito diferente (incluso si es sutilmente diferente).
- ✓ Una buena prueba debería ser la mejor de la cosecha. En un grupo de pruebas que tienen propósitos similares, las limitaciones de tiempo y recursos pueden abogar por la ejecución de sólo un subconjunto de estas pruebas. En tales casos, se debería emplear la prueba que tenga la más alta probabilidad de descubrir una clase entera de errores.
- ✓ Una buena prueba no debería ser ni demasiado sencilla ni demasiado compleja. Aunque es posible a veces combinar una serie de pruebas en un caso de prueba, los posibles efectos secundarios de este enfoque pueden enmascarar errores. En general, cada prueba debería realizarse separadamente.

Durante todo el proceso de desarrollo de un software las pruebas son una parte fundamental del mismo, pues a partir de estas es posible controlar que los productos cumplan requisitos mínimos de operatividad además de garantizar la calidad de los mismos.

1.4.1 Objetivos de las pruebas de software

Estos son algunos de los objetivos que debe perseguir todo diseño y ejecución de Pruebas de Software [7]:

- ✓ Probar si el software hace lo que debe.

✓ Probar si el software hace lo que no debe, es decir, si provoca efectos secundarios adversos.
✓ Descubrir un error que aún no ha sido descubierto.
✓ Encontrar el mayor número de errores con la menor cantidad de tiempo y esfuerzo posible.
✓ Mostrar hasta qué punto las funciones del software operan de acuerdo con las especificaciones y requisitos del cliente.

1.4.2 Principios de las pruebas de software

✓ A todas las pruebas se les deberían poder hacer un seguimiento hasta los requisitos del cliente.
✓ Deberían planificarse mucho antes de que empiecen. La planificación de las pruebas pueden empezar tan pronto como esté completo el modelo de requisitos y se tenga consolidado el modelo de diseño [7].
✓ Deberían empezar por «lo pequeño» y progresar hacia «lo grande». Las primeras planeadas y ejecutadas se centran generalmente en módulos individuales del programa. A medida que se avanza en estas, el enfoque de las pruebas cambia en un intento de encontrar nuevos errores relacionados con la integración de estos módulos y finalmente con la interacción del sistema completo [7].
✓ No son posibles las pruebas exhaustivas. El número de permutaciones de caminos incluso para un programa de tamaño moderado es demasiado grande. Por lo cual, es imposible ejecutar todas las combinaciones de caminos durante las mismas. Sin embargo, es posible elegir y ejecutar una serie de caminos lógicos importantes que permitan probar adecuadamente el software.
✓ Para ser más eficaces, deberían ser realizadas por un equipo independiente. El ingeniero de software que creó el sistema no es el más indicado para realizar las pruebas debido a que consciente o inconscientemente puede omitir casos de prueba importantes que conlleven a descubrir nuevos errores. Por consiguiente, es recomendable organizar un grupo de trabajo independiente para las mismas que suministre una visión más objetiva del software [7].
✓ Se deberían inspeccionar a fondo los resultados de cada prueba.
✓ Examinar un programa para ver que haga lo que se espera, es solo la mitad de la batalla; la otra mitad es ver si hace lo que no se espera.
✓ No se debe hacer una planificación de los esfuerzos de prueba bajo la suposición de que no se encontrarán errores.

✓ La probabilidad de que existan más errores en una sección de programa es proporcional al número de estos que se hayan encontrado en esa sección.
✓ Son una tarea extremadamente creativa e intelectualmente desafiante [8].

1.4.3 Características de las pruebas de software

Algunas de las características que debe tener una «buena prueba» son:

✓ Tiene una alta probabilidad de encontrar un error. El ingeniero de software debe tener un alto nivel de entendimiento de la aplicación a construir para poder diseñar casos de prueba que encuentren el mayor número de defectos.
✓ No debe ser redundante. Uno de los objetivos de las pruebas es encontrar el mayor número de errores con la menor cantidad de tiempo y esfuerzo posible, por lo cual no se deben diseñar Casos de prueba que tengan el mismo propósito que otros, sino que se debe tratar de diseñar el menor número de Casos de prueba que permitan probar adecuadamente el software y optimizar los recursos [4].
✓ Debería ser la mejor de la cosecha. La limitación en tiempo y recursos puede impedir que se ejecuten todos los Casos de prueba de un grupo de pruebas similares por lo cual en estas situaciones se debería seleccionar la prueba que tenga la mayor probabilidad de descubrir errores [8].
✓ No debería ser ni demasiado sencilla ni demasiado compleja.

1.5 Técnicas de pruebas

Entre las técnicas más importantes para elaborar los casos de prueba de software se encuentran la prueba de Caja Negra y la prueba de Caja Blanca.

1.5.1 Técnica de Caja Negra.

Las pruebas se llevan a cabo sobre la interfaz del software, y es completamente indiferente el comportamiento interno y la estructura del programa, centrándose en los requisitos funcionales del software [13].
Los casos de prueba de Caja Negra pretenden demostrar que:
✓ Las funciones del software son operativas.
✓ La entrada se acepta de forma adecuada.
✓ Se produce una salida correcta.
✓ La integridad de la información externa se mantiene.

Se derivan conjuntos de condiciones de entrada que ejerciten completamente todos los requerimientos funcionales del programa. La prueba de Caja Negra intenta encontrar errores de las siguientes categorías:

- ✓ Funciones incorrectas o ausentes.
- ✓ Errores de interfaz.
- ✓ Errores en estructuras de datos o en accesos a bases de datos externas.
- ✓ Errores de rendimiento.
- ✓ Errores de inicialización y de terminación.

Los casos de prueba deben satisfacer los siguientes criterios:

- ✓ Reducir, en un coeficiente mayor que uno, el número de casos de prueba adicionales.
- ✓ Que digan algo sobre la presencia o ausencia de clases de errores.

A menudo los desarrolladores de software experimentados dicen que la prueba nunca termina, simplemente se transfiere del ingeniero de software al cliente. Cada vez que el cliente usa el programa lleva a cabo una prueba. Aplicando el diseño de casos de prueba, el ingeniero del software puede conseguir una prueba más completa, descubrir y corregir el mayor número de errores antes de que comiencen las pruebas del cliente.

1.5.2 Técnica de Caja Blanca

Esta técnica se basa en un minucioso examen de los detalles procedimentales del código a evaluar, por lo que es necesario conocer la lógica del programa.

Permite examinar la estructura interna del programa. Se diseñan casos de prueba para examinar la lógica del programa. Es un método de diseño de casos de prueba que usa la estructura de control del diseño procedimental para derivar casos de prueba que garanticen que [13]:

- ✓ Se ejercitan todos los caminos independientes de cada módulo.
- ✓ Se ejercitan todas las decisiones lógicas.
- ✓ Se ejecutan todos los bucles.
- ✓ Se ejecutan las estructuras de datos internas.

A este tipo de técnica se le conoce también como Técnica de Caja Transparente o de Cristal. Se centra en cómo diseñar los casos de prueba atendiendo al comportamiento interno y la estructura del programa. Se examina así la lógica interna del programa sin considerar los aspectos de rendimiento.

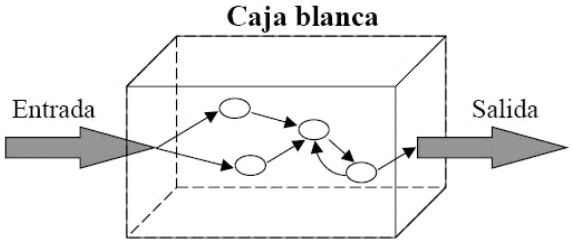

Figura 1: Prueba de Caja Blanca

El objetivo de la técnica es diseñar casos de prueba para que se ejecuten, al menos una vez, todas las sentencias del programa y todas las condiciones tanto en su vertiente verdadera como falsa.

1.6 Niveles de pruebas del Software

Una definición que se puede dar de las pruebas es la siguiente: “Una actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas, los resultados se observan y registran y se realiza una evaluación de algún aspecto” [8].

La prueba es un elemento crítico para la calidad del software. La importancia de los costos asociados a los errores promueve la definición y aplicación de un proceso de pruebas minuciosas y bien planificadas. Las pruebas permiten validar y verificar el software, entendiendo como validación del software el proceso que determina si el software satisface los requisitos y verificación como el proceso que determina si los productos de una fase satisfacen las condiciones de dicha fase. La prueba es aplicada para diferentes tipos de objetivos, en diferentes escenarios o niveles de trabajo. Se distinguen los siguientes niveles de pruebas [14]:

1.6.1 Prueba de desarrollador

Es la prueba diseñada e implementada por el equipo de desarrollo. Tradicionalmente estas pruebas han sido consideradas sólo para la prueba de unidad, aunque en la actualidad en algunos casos pueden ejecutar pruebas de integración. Se recomienda que estas pruebas cubran más que las pruebas de unidad.

1.6.2 Prueba independiente

Es la prueba que es diseñada e implementada por alguien independiente del grupo de desarrolladores. El objetivo de esta prueba es proporcionar una perspectiva diferente. Una vista de la prueba independiente es la prueba independiente de los stakeholders, que están basadas en las necesidades y preocupaciones de los stakeholders.

1.6.3 Prueba de unidad

Es la prueba fundamental en que se basará nuestra investigación. Está enfocada a los elementos testeables más pequeños del software. Es aplicable a componentes representados en el modelo de implementación para verificar que los flujos de control y de datos estén cubiertos y que ellos funcionen como se espera.

Antes de iniciar cualquier otra prueba es preciso probar el flujo de datos de la interfaz del componente. Si los datos no son introducidos correctamente, todas las demás pruebas no tienen sentido. El diseño de casos de prueba de una unidad comienza una vez que se ha desarrollado, revisado y verificado en su sintaxis el código a nivel fuente.

Las pruebas unitarias aseguran que un único componente de la aplicación produzca una salida correcta para una determinada entrada. Este tipo de pruebas validan la forma en la que las funciones y métodos trabajan en cada caso particular. Se encargan de un caso cada vez, lo que significa que un método puede necesitar varias pruebas unitarias si su funcionamiento varía en función del contexto.

Es importante tener claros los siguientes aspectos:

- ✓ Los datos de entrada son conocidos por el Tester o Diseñador de Pruebas, deben ser preparados con minuciosidad, ya que el resultado de las pruebas depende de estos.
- ✓ Se debe conocer qué componentes interactúan en cada caso de prueba.
- ✓ Se debe conocer de antemano qué resultados debe devolver el componente según los datos de entrada utilizados en la prueba.
- ✓ Finalmente se deben comparar los datos obtenidos en la prueba con los datos esperados, si son idénticos se puede decir que el módulo superó la prueba, pasando a la siguiente.

Estas pruebas no descubrirán todos los errores del código, sólo prueban que el segmento de código que se revisa esté lógicamente correcto. No detectarán errores de integración. Aisladas proporcionan algunas ventajas básicas:

Fomentan el cambio: Las pruebas unitarias facilitan que el programador cambie el código para mejorar su estructura (lo que se conoce como refactorización), puesto que permiten hacer pruebas sobre los cambios y así asegurarse de que los nuevos cambios no han introducido errores.

Simplifica la integración: Puesto que permiten llegar a la fase de integración con un alto grado de seguridad de que el código está funcionando correctamente.

Documenta el código: Las propias pruebas son documentación del código puesto que ahí se puede ver cómo utilizarlo.

Separación de la interfaz y la implementación: Dado que la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otro.

Se da más seguridad al programador: Normalmente, la persona que ha programado un módulo no es la misma que la que tiene que corregir sus errores. Esto crea una sensación de inseguridad al programador, ya que a la hora de corregir un error no tiene la certeza de que va a afectar a otros módulos que desconoce. Las pruebas unitarias aseguran que una corrección no repercuta en otros módulos, permitiendo al programador centrarse en la solución del error y no en la repercusión que puede tener esa corrección.

Las pruebas funcionales se hacen más sencillas: Pues la mayoría de los aspectos individuales de cada unidad ya están probados a través de las pruebas unitarias. De este modo, las pruebas funcionales deben centrarse sólo en verificar la correcta cooperación de las distintas unidades y en los funcionamientos generales del programa.

La prueba de unidad centra el proceso de verificación en la menor unidad del diseño del software, el componente software o módulo.

Se prueba la interfaz del módulo para asegurar que la información fluye de forma adecuada hacia y desde la unidad de programa que está siendo probado. Se examinan las estructuras de datos locales para asegurar que los datos que se mantienen temporalmente conservan su integridad durante todos los pasos de ejecución del algoritmo. Se prueban las condiciones límite para asegurar que el módulo funciona correctamente en los límites establecidos como restricciones de procesamiento.

Se ejercitan todos los caminos independientes (caminos básicos) de la estructura de control con el fin de asegurar que todas las sentencias del módulo se ejecutan por lo menos una vez. Finalmente, se prueban todos los caminos de manejo de errores.

1.6.4 Prueba de integración

Es ejecutada para asegurar que los componentes en el modelo de implementación operen correctamente cuando son combinados para ejecutar un caso de uso. Se

prueba un paquete o un conjunto de paquetes del modelo de implementación. Descubren errores en las especificaciones de las interfaces de los paquetes. Esta prueba debe ser responsabilidad de desarrolladores y de los stakeholders, sin solaparse las pruebas. En la prueba de integración el foco de atención es el diseño y la construcción de la arquitectura del software.

Hay dos formas de integración:

✓ Integración no incremental: Se combinan todos los módulos por anticipado y se prueba todo el programa en conjunto.

✓ Integración incremental: El programa se construye y se prueba en pequeños segmentos.

1.6.5 Prueba de sistema

Son las pruebas que se hacen cuando el software está funcionando como un todo. Es la actividad de prueba dirigida a verificar el programa final, después que todos los componentes de software y hardware han sido integrados. En un ciclo iterativo estas pruebas ocurren más tempranamente, tan pronto como subconjuntos bien formados de comportamiento de casos de uso son implementados.

1.6.6 Prueba de aceptación

Prueba de aceptación del usuario es la prueba final antes del despliegue del sistema. Su objetivo es verificar que el software está listo y que puede ser usado por usuarios finales para ejecutar aquellas funciones y tareas para las cuales el software fue construido.

1.7 Metodología de desarrollo.

Con los avances que ha alcanzado la informática a nivel mundial, fundamentalmente en la evolución tecnológica se ha hecho necesario desarrollar metodologías para asegurar la calidad de los productos de software y obtener un mejoramiento continuo de todos los procesos relacionados con el desarrollo de software. No existe una metodología universal que garantice el éxito de cualquier proyecto de desarrollo de software. Sin embargo cada proyecto debe tener una metodología definida para su desarrollo.

1.7.1 Proceso Unificado de Desarrollo de Software (RUP)

El Proceso Unificado Racional es un proceso de desarrollo de software, el cual constituye la metodología estándar más utilizada para el análisis, implementación y documentación de sistemas orientados a objetos. El ciclo de vida RUP es una

implementación del desarrollo en espiral y organiza las tareas en fases e iteraciones. En su modelación se definen como elementos: Trabajadores (quién): Define el comportamiento y responsabilidades (rol) de un individuo, grupo de individuos, sistema automatizado o máquina, que trabajan en conjunto como un equipo. Ellos realizan las actividades y son propietarios de elementos. Actividades (cómo): Tarea que tiene un propósito claro, es realizada por un trabajador y manipula elementos. Artefactos (qué): Productos tangibles del proyecto que son producidos, modificados y usados por las actividades. Pueden ser modelos, elementos dentro del modelo, código fuente y ejecutables. Flujo de Actividades (cuándo): Secuencia de actividades realizadas por trabajadores y que produce un resultado de valor observable.

1.7.1.1 Flujo de trabajo de pruebas en la metodología RUP

RUP divide el proceso en cuatro fases, dentro de las cuales se realizan varias iteraciones en número variable según el proyecto y en las que se hace un mayor o menor hincapié en las distintas actividades.

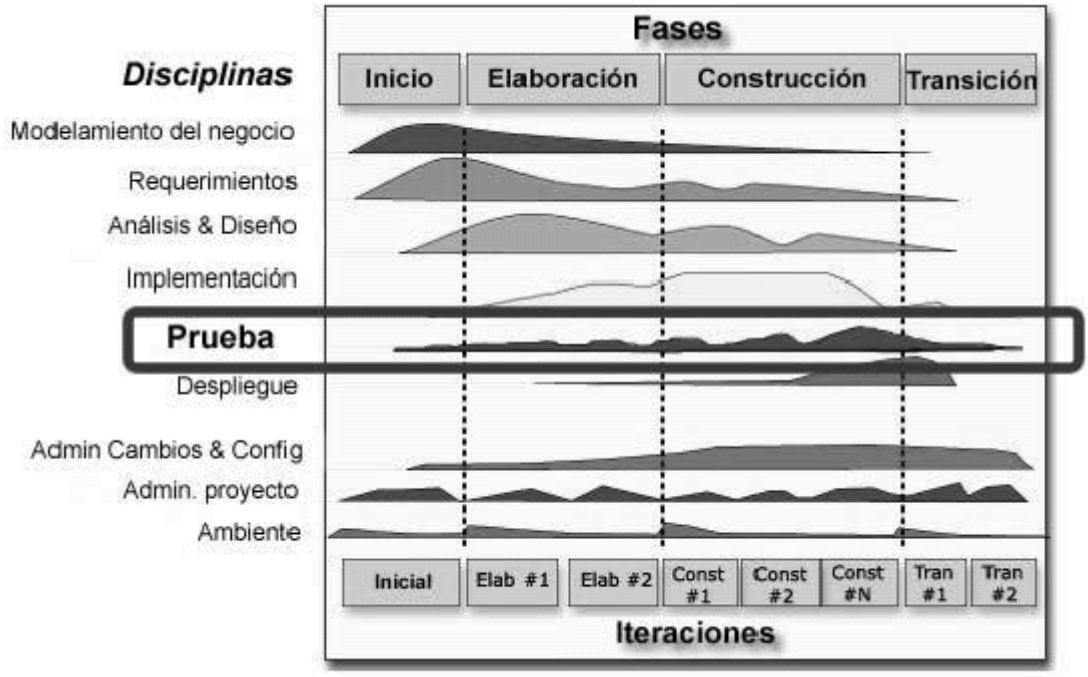

Figura 2: Flujo de Trabajo de RUP

El flujo de trabajo de prueba es el encargado de evaluar la calidad del producto que se está desarrollando, pero no para aceptar o rechazar el producto al final del proceso de desarrollo, sino que debe ir integrado en todo el ciclo de vida.

1.7.1.2 Ventajas

- ✓ Mitigación temprana de posibles riesgos altos.
- ✓ Progreso visible en las etapas tempranas.
- ✓ El conocimiento adquirido en una iteración puede aplicarse de iteración a iteración.
- ✓ Los usuarios están involucrados continuamente.

1.7.1.3 Desventajas

- ✓ Por el grado de complejidad puede no resultar muy adecuado.
- ✓ EL RUP es generalmente mal aplicado en el estilo cascada.
- ✓ Requiere conocimientos del proceso y de UML.

1.7.2 Extreme Programe (XP)

Es una metodología ágil centrada en potenciar las relaciones interpersonales como clave para el éxito del desarrollo de software, promoviendo el trabajo en equipo, preocupándose por el aprendizaje de los desarrolladores, y propiciando un buen clima de trabajo. XP se basa 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.

1.7.2.1 Proceso XP

El ciclo de desarrollo consiste (a grandes rasgos) en los siguientes pasos:

1. El cliente define el valor de negocio a implementar.
2. El programador estima el esfuerzo necesario para su implementación.
3. El cliente selecciona qué construir, de acuerdo con sus prioridades y las restricciones de tiempo.
4. El programador construye ese valor de negocio.
5. Vuelve al paso 1.

En todas las iteraciones de este ciclo tanto el cliente como el programador aprenden. No se debe presionar al programador a realizar más trabajo que el estimado, ya que se perderá calidad en el software o no se cumplirán los plazos. De la misma forma el cliente tiene la obligación de manejar el ámbito de entrega del producto, para asegurarse que el sistema tenga el mayor valor de negocio posible con cada iteración.

El ciclo de vida ideal de XP consiste de seis fases:

Exploración, Planificación de la Entrega (Release), Iteraciones, Producción, Mantenimiento y Muerte del Proyecto.

1.7.2.2 **Desarrollo guiado por pruebas**, o **Test-driven development (TDD)**

Es una práctica dentro de la metodología XP que involucra otras dos prácticas: Escribir las pruebas primero (Test First Development) y Refactorización (Refactoring). En primer lugar se escribe una prueba y se verifica que las pruebas fallen, luego se implementa el código que haga que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito. El propósito del desarrollo guiado por pruebas es lograr un código limpio que funcione, la idea es que los requisitos sean traducidos a pruebas, de este modo, cuando las pruebas pasen se garantizará que los requisitos se hayan implementado correctamente.

Cada prueba será suficientemente pequeña como para que permita determinar unívocamente si el código probado pasa o no la prueba que esta le impone. El diseño se ve favorecido ya que se evita el indeseado “sobre diseño” de las aplicaciones y se logran interfaces más claras y un código más cohesivo.

1.7.2.2.1 Características

Se evita escribir código innecesario, si pasa una prueba aunque sepamos que es incorrecto nos da una idea de que tenemos que modificar nuestra lista de requisitos agregando uno nuevo.

La generación de pruebas para cada funcionalidad hace que se confíe en el código escrito; permite hacer modificaciones profundas del código, pues si se logra hacer pasar todas las pruebas se tendrá un código que funcione correctamente.

Los programadores que utilizan el desarrollo guiado por pruebas en un proyecto virgen encuentran que en raras ocasiones tienen la necesidad de utilizar el depurador o debugger.

1.7.2.2.2 Ventajas

A pesar de los elevados requisitos iniciales de aplicar esta metodología, el desarrollo guiado por pruebas (TDD) puede proporcionar un gran valor añadido en la creación de software, produciendo aplicaciones de más calidad y en menos tiempo. Ofrece más que una simple validación del cumplimiento de los requisitos, puede guiar el diseño de un programa. Se centra en primer lugar en los casos de prueba, por lo tanto, al programador solo le importa la interfaz y no la implementación.

El poder del TDD radica en la capacidad de avanzar en pequeños pasos cuando se necesita. Permite centrarse en la tarea actual y la primera meta es a menudo hacer que la prueba pase. Inicialmente no se consideran los casos excepcionales y el manejo de errores. Estos, se implementan después de que se haya alcanzado la funcionalidad principal. Otra ventaja es que, cuando es utilizada correctamente, se

asegura de que todo el código escrito esté cubierto por una prueba. Esto puede dar un mayor nivel de confianza en el código.

1.7.2.2.3 Limitaciones

El desarrollo guiado por pruebas requiere que las pruebas puedan automatizarse. Esto resulta complejo en los siguientes dominios:

Objetos distribuidos, aunque los objetos simulados (*MockObjects*) pueden ayudar.

Bases de datos. Hacer pruebas de código que trabaja con base de datos es complejo porque requiere poner en la base de datos unos datos conocidos antes de hacer las pruebas y verificar que el contenido de la base de datos es el esperado después de la prueba. Todo esto hace que la prueba sea costosa de codificar, aparte de tener disponible una base de datos que se pueda modificar libremente.

1.7.3 Metodología a desarrollar

Vistas estas dos metodologías las cuales tienen definidas etapas de pruebas durante todo el desarrollo de software se hace una comparación entre las mismas y se decide guiar el proceso siguiendo lo planteado por TDD ya que es un proceso ágil, que guía el desarrollo del producto guiándose por las pruebas, poniéndole gran importancia a las mismas, dando cumplimiento a los objetivos planteados en la investigación. No se toma RUP como metodología ya que a pesar de que las pruebas se encuentran implícitas en toda la fase de desarrollo del producto, las mismas se realizan al final de cada una de las etapas; por lo que si se detecta algún error es posible que tengamos que virar al principio de la etapa, provocando con esto pérdidas de tiempo y recursos.

1.8 Herramientas para realizar pruebas unitarias.

Una herramienta ofrece los servicios necesarios para dar soporte a una tarea determinada. Para la realización de pruebas unitarias existen herramientas y entornos de desarrollo (frameworks) que facilitan su creación en multitud de lenguajes de programación.

Algunas de estas son [15]:

JUnit: Entorno de pruebas para Java creado por Erich Gamma y Kent Beck. Se encuentra basado en Sunit creado originalmente para realizar pruebas unitarias para el lenguaje Smalltalk. Es un framework de pruebas unitarias en Java, permite la automatización del código de los casos de prueba Java.

Con Junit se logra ejecutar pruebas automatizadas fácilmente. Contiene mecanismos para recolectar resultados de manera estructurada, produce varios tipos de reportes a

partir de los resultados obtenidos en la ejecución de las pruebas, permite que existan relaciones entre las pruebas y que unas reutilicen código de otras. Además, soporta la jerarquía entre las pruebas estableciendo la prioridad y el orden de unas con otras.

DbUnit: Es una extensión de Junit para proyectos que utilizan bases de datos. Su principal objetivo es colocar la base de datos en un estado conocido para la ejecución de las pruebas.
Ayuda a evitar muchos problemas que usualmente ocurren cuando una prueba corrompe la base de datos y subsiguientes pruebas puedan fallar o empeorar el daño. DbUnit tiene la habilidad de exportar e importar los datos de la base de datos hacia y desde ficheros XML y puede verificar si la información contenida en la base de datos coincide con conjuntos de valores esperados.

SimpleTest: Entorno de pruebas para aplicaciones realizadas en PHP. Es similar a Junit / PHPUnit. Compatible con los objetos simulados, se puede utilizar para automatizar las pruebas de regresión de aplicaciones web con un cliente de secuencias de comandos HTTP que puede analizar las páginas HTML y simular cosas como hacer clic en los enlaces y la presentación de formularios [16].

PHPUnit: Framework para realizar pruebas unitarias en PHP. Es un miembro de la familia xUnit de los marcos de las pruebas, proporciona un marco que hace que la escritura de las pruebas sea fácil y tiene la funcionalidad de ejecutar fácilmente las pruebas y analizar sus resultados. Es la herramienta incluida dentro de la suite de pruebas del centro, para la realización de pruebas de Caja Blanca.

Zend framework [17]**:** Framework de código abierto, orientado a objeto, donde cada componente tiene una baja dependencia respecto a otros componentes, lo cual permite a los desarrolladores utilizar los componentes por separado. Entre estas herramientas podemos encontrar:

- ✓ Componentes modelo-vista-controlador (MVC).
- ✓ Abstracción a Base de Datos.
- ✓ WebServices.

Facilidades que aporta el uso de este framework:

- ✓ Simplicidad en el uso.
- ✓ Códigos más estables y con menos probabilidad de error.
- ✓ Simplicidad en el mantenimiento del código.

Este framework insiste fundamentalmente en la calidad del código, a través de pruebas unitarias, utilizando PHPUnit.

YUI[18]**:** Framework de pruebas de soluciones basadas en el navegador JavaScript. Usando YUI prueba, se puede agregar fácilmente las pruebas unitarias a sus soluciones de JavaScript. Aunque no es un puerto directo de cualquier marco xUnit específica, YUITest emplea algunas características de nUnit y Junit.

Características:

- ✓ Rápida creación de casos de prueba a través de la sintaxis simple.
- ✓ Avanzada detección de fallos en los métodos que generan errores.
- ✓ Agrupación de casos de prueba relacionadas con bancos de pruebas.
- ✓ Simulacro de objetos para la escritura de pruebas sin dependencias externas.
- ✓ Pruebas asíncronas de eventos de prueba y de comunicación Ajax.
- ✓ DOM de simulación de eventos en todos los navegadores de la categoría A (a través de eventos).

Después de vista las características de algunas de las principales herramientas para desarrollar pruebas de unidad, centraremos la investigación sobre el framework Yui, más específicamente en su librería de prueba. Yui fue una de las bases para el desarrollo de Extjs y su compatibilidad con el mismo es excelente. Extjs es el framework que se utiliza en el proyecto de CEDRUX, para la programación de la Interfaz de usuario.

Conclusiones del capítulo.

En el presente capítulo se realizó un análisis del estado actual de las pruebas del software y de los distintos tipos y niveles de pruebas que existen, enfocándose principalmente en las pruebas de unidad. Se hizo además una breve descripción de la metodología XP y el uso de una de sus prácticas; el desarrollo guiado por pruebas; el cual guiará el proceso de pruebas en nuestra investigación. Finalmente se analizaron distintas herramientas que se utilizan para realizar las pruebas de unidad, entre ellas las que se encuentran presente en la suite de pruebas propuesta para el centro, haciendo énfasis en el framework YUI; el cual se propone se incorpore a la suite de pruebas; pues forma parte del escenario de pruebas previsto por Extjs que es el framework para el desarrollo de la interfaz de usuario que se utiliza en el proyecto CEDRUX

Capítulo 2 Propuesta de procedimiento

Introducción

Las pruebas de unidad tienen un impacto importante en el código y la calidad del producto final, por esta razón en el presente capítulo se ofrece una propuesta de solución para garantizar la realización de este tipo de pruebas. Se elabora un procedimiento para ser aplicado en el Centro de Informatización de la Gestión de Entidades (CEIGE). Se define el alcance del procedimiento con el objetivo de mostrar la magnitud que tendrá el mismo especificando qué y cómo se va a probar, también los objetivos que se persiguen, las actividades propuestas a desarrollar para lograr esos objetivos, así como una descripción de las mismas. Se utiliza el procedimiento descrito por TDD y se apoya en la Plantilla de diseño de casos de pruebas y el framework para pruebas de unidad YUI.

2.1 Procedimientos para pruebas de unidad.

Un procedimiento es la acción de proceder o el método de ejecutar algunas cosas. Se trata de una serie común de pasos definidos, que permiten realizar un trabajo de forma correcta. En la universidad existe una propuesta de procedimientos para realizar pruebas de unidad, la cual fue concebida para el desarrollo del proyecto Aduana y la realización de pruebas unitarias en el mismo a través del freamwork Symfony y las herramientas que trae asociadas el mismo, se basa en la utilización de la metodología Proceso Unificado de Desarrollo de Software (RUP).

Debido principalmente a que este procedimiento está concebido para trabajarse con las herramientas de desarrollo que propone el framwork Symfony; no resulta de utilidad su puesta en práctica en nuestro sistema ya que el mismo está realizado con el framwork Extjs

El hecho de que este procedimiento propuesto por el proyecto Aduana este desarrollado bajo la metodología RUP, provoca que no se cumplan los objetivos de la presente investigación. A pesar de que RUP propone un proceso de prueba en cada una de sus fases de desarrollo, las mismas se realizan al final de las mismas por lo que un error en una de las pruebas provocaría tener que empezar de nuevo desde la última prueba realizada o sea rehacer una fase que se tenía por terminada. Por eso es que el procedimiento que se propone en esta investigación está basado en la práctica de la metodología XP; TDD; que como ventaja tiene que el desarrollo de un software esté dirigido por el proceso de pruebas , brindándole un peso importante a esta actividad.

2.2 Principales problemas para realizar las pruebas de unidad en el proyecto CEDRUX.

Pruebas manuales.

En el proyecto CEDRUX en caso de que se efectúen las pruebas; porque no es una normativa; las pruebas unitarias se realizan de manera manual por parte de los programadores del centro, lo cual es un trabajo muy tedioso y a la larga dificulta el desarrollo posterior de estas pruebas. Esto se debe principalmente a la falta de conocimiento respecto al tema en cuestión.

Con cambios constantes en los requisitos las pruebas, principalmente las unitarias, deben poder ser repetibles, reutilizables e independientes, las cuales ayudan a producir códigos más seguros.

Reutilizables o Repetibles: No se deben crear pruebas que sólo puedan ser ejecutadas una sola vez. Las pruebas deben poder ser ejecutadas varias veces.

Independientes: La ejecución de una prueba no debe afectar a la ejecución de otra, es decir, estas deben ser independientes de otras pruebas para que ellas se ejecuten solas.

Automatizables: La automatización es una de las formas de mantener la calidad de las pruebas, generando cada vez más grandes y complejas cantidades de código con menos esfuerzo.

Por estas razones se describirá un procedimiento para la realización de estas pruebas de forma automática.

Pruebas automáticas

Cualquier programador con experiencia en el desarrollo de aplicaciones web conoce de sobra el esfuerzo que supone probar correctamente la aplicación. Crear casos de prueba, ejecutarlos y analizar sus resultados es una tarea tediosa. Además, es habitual que los requisitos de la aplicación varíen constantemente, con el consiguiente aumento del número de versiones de la aplicación y la refactorización continua del código, la realización de las pruebas de forma automática facilitan todo el trabajo.

Este es el motivo por el que la automatización de pruebas es una recomendación útil para crear un entorno de desarrollo satisfactorio. Los conjuntos de casos de prueba garantizan que la aplicación hace lo que se supone que debe hacer.

Incluso cuando el código interno de la aplicación cambia constantemente, las pruebas automatizadas permiten garantizar que los cambios no introducen incompatibilidades en el funcionamiento de la aplicación. Este tipo de pruebas obligan a los programadores a crear pruebas en un formato estandarizado y muy rígido que pueda ser procesado por un framework de pruebas.

En ocasiones, las pruebas automatizadas pueden reemplazar la documentación técnica de la aplicación, ya que ilustran de forma clara el funcionamiento de esta.

2.3 Pruebas unitarias en Extjs.

Extjs es una biblioteca de Java Script para el desarrollo de aplicaciones web interactivas usando tecnologías como AJAX, DHTML y DOM. Fue desarrollada por Sencha. Originalmente construida como una extensión de la biblioteca YUI, en la actualidad puede usarse como extensión para la biblioteca jQuery y Prototype. Desde la versión 1.1 puede ejecutarse como una aplicación independiente.

No es hasta la versión 3.2 del producto que los desarrolladores de Extjs se dan cuenta de la importancia de incluir en su librería un módulo de prueba para las pruebas unitarias basadas en el freamwork de YUI Test.

En el Centro de Informatización de la Gestión de Entidades la versión de Extjs que se utiliza es la 2.0 la cual no tiene implementada todavía la suit de pruebas, por lo que para la realización de las mismas se le incluirá las librerías de pruebas que propone YUI Test.

En caso de que el proyecto migre a una versión superior del framework Extjs el procedimiento continúa siendo válido, porque cumple con las características idóneas para desarrollarse en cualquier versión del mismo.

2.4 Procedimiento para realizar pruebas de unidad.

La construcción de un sistema de software tiene como objetivo satisfacer una necesidad planteada por el usuario. Para asegurar que se han alcanzado los niveles de calidad acordados es necesario evaluar el producto de software a medida que se va construyendo. Por lo tanto se hace necesario llevar a cabo, en paralelo al proceso de desarrollo, un proceso de evaluación o comprobación de los distintos productos o modelos que se van generando.

Basándonos en el diseño guiado por pruebas (TDD) se cumplirá este tipo de requisito, ya que esta es una herramienta de diseño que convierte al programador en un “oficial de primera”. Es la respuesta a las grandes preguntas de:
¿Cómo lo hago?, ¿Por dónde empiezo?, ¿Cómo sé qué es lo que hay que implementar y lo que no?, ¿Cómo escribir un código que se pueda modificar sin romper funcionalidad existente?
No se trata de escribir pruebas por escribir sino de diseñar adecuadamente según los requisitos.

Descripción

El procedimiento para realizar pruebas de unidad definirá de forma detallada los pasos para llevar a cabo estas pruebas. Analizará en detalle cada una de las fases que forma este procedimiento.

Alcance

Este procedimiento está dirigido a realizar las pruebas de unidad. Probar las funciones individuales o métodos: se probarán las entradas y las salidas y se comprobará que los valores obtenidos son los esperados.

Objetivos

Las pruebas unitarias desarrolladas en este procedimiento tienen como objetivo aislar cada parte del programa y mostrar que las partes individuales son correctas. Proporcionan un contrato escrito que el trozo de código debe satisfacer. Son fragmentos de unidades estructurales del programa encargados de una tarea en específico. El objetivo principal sería producir las piezas de código de la manera más eficiente y eficaz posible generando pruebas de unidad para las mismas que aseguren su correcto comportamiento.

2.4.1 Fases del procedimiento

El presente procedimiento de pruebas de unidad se divide en las siguientes fases:

1. **Escribir la especificación del requisito (el ejemplo, el test).**
2. **Implementar el código.**
3. **Refactorizar para eliminar duplicidad y hacer mejoras.**

Figura 3: Fases del Procedimiento

2.4.2 Descripción textual del procedimiento.

Roles	Entradas	Actividades	Descripción	Salidas
Analista.		1. Identificar requisito.	El analista identifica los requisitos según la descripción dada por el cliente.	Plantilla de Especificación de requisitos.
Programador.	Plantilla de Especificación de requisitos.	2. Tomar requisito.	Se revisa la Plantilla de Especificación de requisitos. Y se toma uno para desarrollar.	
		3. Diseñar la prueba.	Se diseña la prueba que dé solución al requisito. Si surgen nuevos requisitos se pasa al flujo alterno 2.a.	Plantilla diseño caso de prueba unidad.
		4. Escribir la prueba.	Se escribe el código mínimo necesario que dé solución al requisito.	

		5. Realizar prueba unitaria.	Se somete el código a la prueba en la herramienta Yui Test.	Reporte de la prueba.
		6. Revisión de los resultados del reporte.	Se revisa el resultado del reporte. Si el resultado no es el esperado se pasa al flujo alterno 5.a.	
		7. Refactorizar.	Se revisa el código para eliminar la duplicidad y que cumpla con los estándares de diseño del centro.	
Flujos Alternos.				
Analista.	2. a Nuevos requisitos.		Se insertan en la Plantilla de Requisitos para tomarlos nuevamente después. Se continúa con el flujo de actividades.	Plantilla de Requisitos.
Programador.	5. a Fallo en las pruebas.		Se vuelve a la actividad 3.	

Tabla 1: Flujo de actividades.

2.4.3 Descripción gráfica del procedimiento

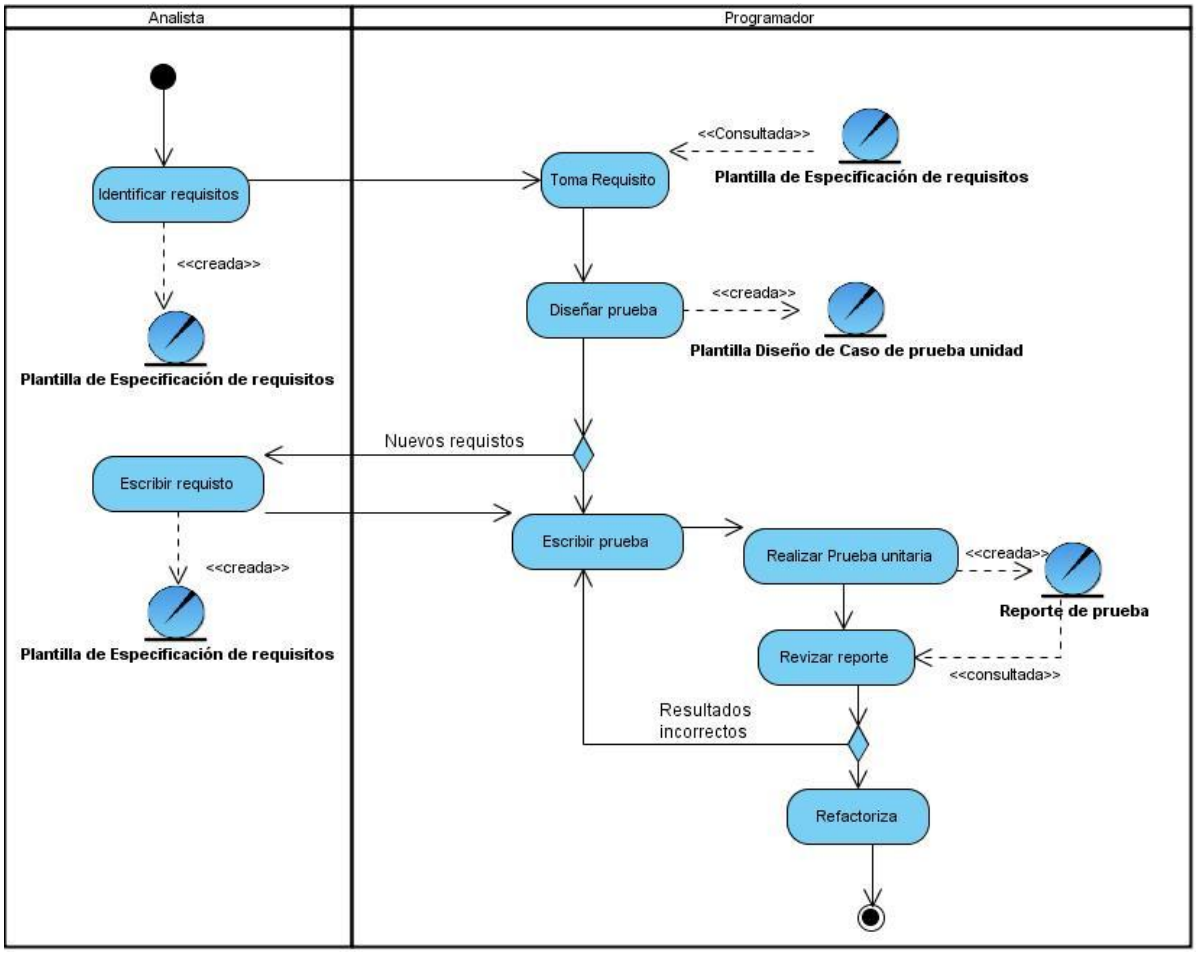

Figura 4: Diagrama de actividades.

Para un mayor entendimiento del proceso, los siguientes epígrafes se dedican a cada fase, describiendo en cada una las actividades a realizar y las salidas que se generarán.

2.4.3.1 Escribir la especificación del requisito.

En esta fase se planificarán las pruebas unitarias que se le realizarán a la aplicación. Recogiéndose todo en la plantilla Diseño de casos de prueba unidad, en la misma se documentan una serie de elementos que aseguran una mejor calidad del producto.

Se toman los test de aceptaciones iniciales, definiéndose el código mínimo necesario para darle solución a estos, sin preocuparnos por los nuevos requisitos que puedan salir a la luz una vez que se haya empezado a escribir el código.

Estos nuevos casos de usos que van surgiendo mientras se escriben los test o la lógica del negocio y que no estaban contemplados inicialmente, se escriben en un lugar aparte donde se puedan retomar más tarde he incluirlos en la Plantilla de requisitos. Esto no debe distraer lo que estamos haciendo ni tampoco influenciar; hay que centrarse en una sola cosa a la vez para llevarla a cabo como es debido.
Se toma uno de los test de aceptación y se hace una lista de elementos que hacen falta para llevarlo a cabo.
Pudiera surgir la incógnita:
¿Acaso no es posible escribir una especificación antes de implementarla?

Por eso, un test no es inicialmente un test sino un ejemplo o especificación. La palabra especificación podría tener la connotación de que es inamovible, algo preestablecido y fijo, pero no es así, un test se puede modificar. Para poder escribirlo, se tiene que pensar primero en cómo se quiere que sea la API del SUT, es decir, hay que trazar antes de implementar. Pero sólo una parte pequeña, un comportamiento del SUT bien definido y sólo uno. Hacer el esfuerzo por imaginar cómo sería el código del SUT si ya estuviera implementado y cómo se comprobaría que, efectivamente, hace lo que le debe hacer. La diferencia con los que dictan normalmente una prueba es que no se diseñan todas las especificaciones antes de implementar cada una, sino que van una a una siguiendo los tres pasos del algoritmo TDD. El hecho de tener que usar una funcionalidad antes de haberla escrito le da un giro de 180 grados al código resultante. Cuidando de diseñar lo que nos sea más cómodo, más claro, siempre que cumpla con el requisito objetivo.

Documentación de las pruebas.

Como en el sistema CEDRUX las pruebas de unidad no se realizan, no existe documentación que gestione las mismas, para resolver este problema se crea la Plantilla diseño de casos de prueba unidad (Ver anexo 1.). Esta plantilla definida dentro de los estándares que plantea el centro CEIGE, cuenta con 3 partes principales:

1. **Descripción:** Se describe la prueba que se va a realizar.
2. **Tabla de Funcionalidades:** Se nombran de las funcionalidades que se van a probar del módulo o requisito escogido, así como una pequeña descripción de éstas.

Funcionalidades a probar	Descripción
[1: Nombre del requisito]	[Breve descripción de la acción que se realiza en el requisito]

Tabla 2: Tabla de funcionalidades.

3. **Tabla de Casos de Pruebas:** Se detallan los casos de prueba para las funcionalidades descritas en la tabla anterior. Se recogen los resultados obtenidos después de aplicar los métodos que ofrece Yui para comprobar el correcto funcionamiento de las funcionalidades a probar.

Esta tabla está compuesta por una serie de campos, el primero (Funcionalidad), recoge el nombre de la funcionalidad que será comprobada, el segundo (Método utilizado) es el método de la clase Yuitest utilizado, el tercer campo (Recibe) contiene los parámetros que recibe la funcionalidad al ser comprobada, el cuarto(Resultado esperado del método) es el resultado que se espera que devuelva el método, en el quinto(Resultado esperado de la prueba) se escribe el resultado que se espera devuelva la prueba luego de ser ejecutada y la última columna(Resultado de la prueba) es el valor resultado real que obtuvo finalmente dicha prueba. El resultado es satisfactorio si coinciden los resultados reales de las pruebas con los esperados, en caso de no coincidir es detectado un error.

Funcionalidad		Método Utilizado	Recibe	Resultado esperado del método	Resultado esperado de la prueba	Resultado de la prueba
[1]	[Nombre de la funcionalidad].	[Nombre del método de prueba empleado]	[Variable(s) que recibe el método].	[Resultado que se espera de él método].	[Resultado que se espera de la	[Resultado real que da la prueba].

Tabla 3: Tabla de casos de pruebas.

2.4.3.2 Implementar el código.

Teniendo el ejemplo escrito, se implementa el código mínimo necesario para que el test pase.

Típicamente, el mínimo código es el de menor número de caracteres, mínimo quiere decir el que menos tiempo llevó escribirlo. No importa que el código parezca tedioso o chapucero. En este paso, la máxima es no implementar nada más que lo estrictamente obligatorio para cumplir la especificación actual.

No se trata de hacerlo sin pensar, sino concentrarse para ser eficientes. Parece fácil pero, al principio, no lo es; siempre se escribe más código del que hace falta. Siempre vendrán a la mente dudas sobre el comportamiento del SUT ante distintas entradas, es decir, los distintos flujos condicionales que pueden entrar en juego. Sólo la atención ayudará a contener el impulso y a anotar las preguntas que han surgido en un lugar al

margen para convertirlas en especificaciones que se retomarán después, en iteraciones consecutivas.

2.4.3.2.1 Yui test.

Para este paso se utilizará el framework de pruebas completas para las aplicaciones de Java Script y Web (YUI Test). YUI puede utilizar la sintaxis simple Java Script para escribir pruebas unitarias que se pueden ejecutar en los navegadores web o en la línea de comandos, así como las pruebas funcionales que se ejecutan en navegadores web. Herramientas adicionales que se acumulan en la parte superior de la funcionalidad principal de la biblioteca de Java Script que permiten la integración con los sistemas de construcción.

Para poder realizar las pruebas unitarias en la versión de Extjs que utiliza el sistema CEDRUX a través de la herramienta de ejecución de pruebas , usando el framework de pruebas Yui Test 3.1 debemos seguir los siguientes pasos:

1. Descargar la librería para pruebas de unidad que se encuentra en el repositorio de calidad del centro.

2. Crear un directorio con el nombre del caso a probar dentro del sistema de archivo Test , que tiene el ejecutor de pruebas

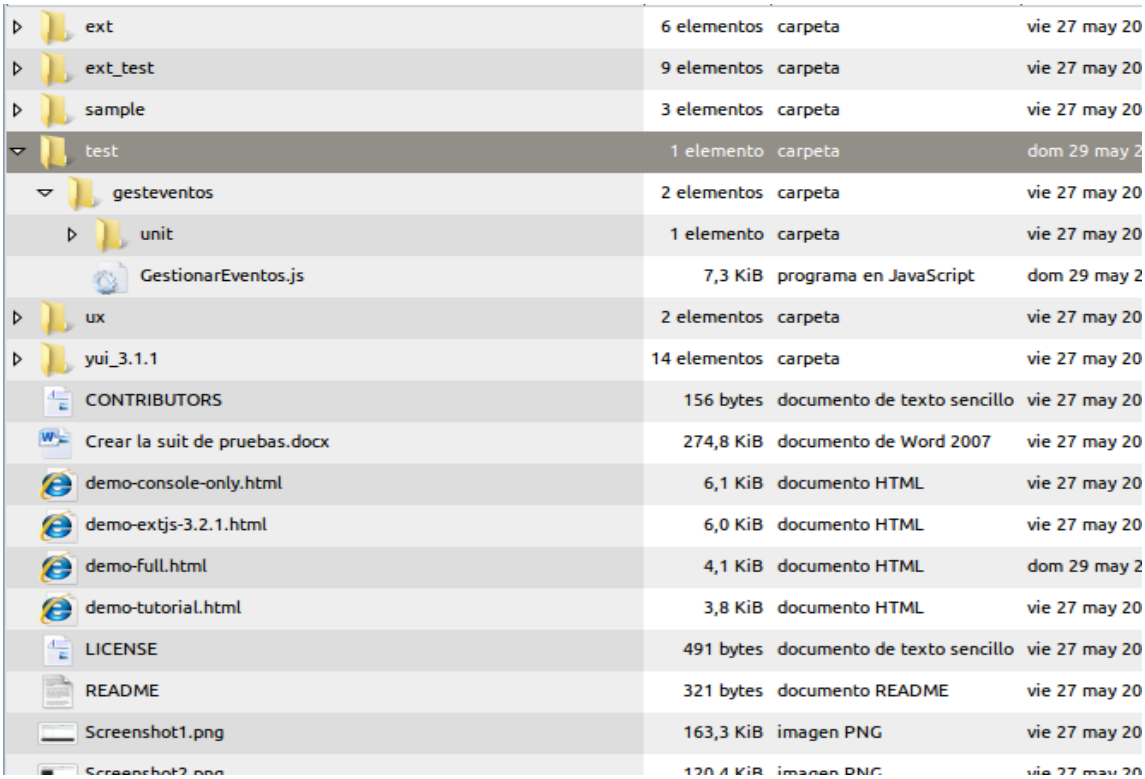

Figura 5: Crear la carpeta de pruebas.

3. Copiar el código del caso a probar dentro de esta carpeta y crear una carpeta llamada unit con el código del caso de prueba (se recomienda nombrar el fichero con el mismo nombre del caso a probar antecedido por la letra T , que indica que es el fichero de prueba)

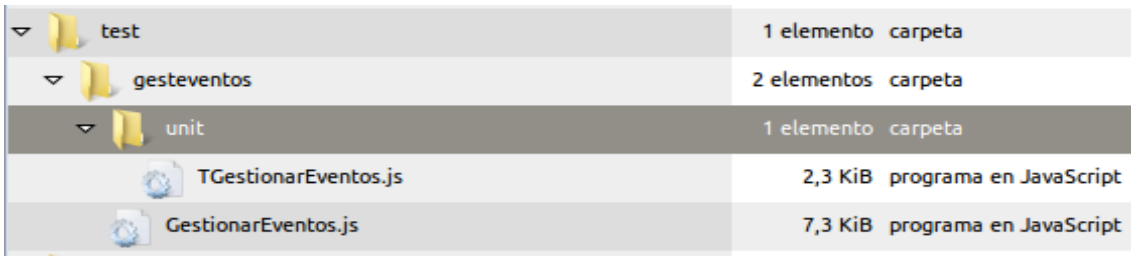

Figura 6: Crear código del caso de prueba.

4. Incluir el directorio dentro del bloque pruebas unitarias del fichero html

```
<!--Bloque de pruebas unitarias-->
    <script type="text/javascript" src="test/gesteventos/unit/TGestionarEventos.js"></script>
<!--Fin del  Bloque de pruebas unitarias-->
```

Figura 7: Directorios.

5. Incluir el caso de uso en el bloque de casos de usos

```
<!--    Bloque de casos de uso-->
<script type="text/javascript" src="test/gesteventos/GestionarEventos.js"></script>
<!--    Fin Bloque de casos de uso-->
```

Figura 8: Casos de usos.

En los dos anteriores pasos (3 y 4) , se incluyeron en la herramienta los ficheros encargados de asegurar las pruebas unitarias .

2.4.3.2.2 Métodos de pruebas

Yui Test dispone de un gran número de clases y métodos para las pruebas unitarias, en la siguiente tabla se muestra un ejemplo de los más importantes.

Metodos	Descripción
areEqual (expected , actual , message)	Afirma que un valor es igual a otro.
areNotEqual (unexpected , actual , message)	Afirma que el valor no es igual a otro.
Fail (message)	Fuerza que se produzca un error en la aplicación.

isArray (actual , message)	Afirma que un valor es un arreglo.
isBoolean (actual , message)	Afirma que un valor es un valor booleano.
isFalse (actual , message)	Afirma que un valor es falso.
isFunction (actual , message)	Afirma que un valor es una función.
isInstanceOf (expected , actual , message)	Afirma que un valor es una instancia de un objeto en particular.
isNaN (actual , message)	Afirma que el valor no es un número
isNotNaN (actual , message)	Afirma que el valor no es el valor especial NaN.
isNotNull (actual , message)	Afirma que el valor no es nulo.
isNotUndefined (actual , message)	Afirma que un valor no está definido.
isNull (actual , message)	Afirma que un valor es nulo.
isNumber (actual , message)	Afirma que un valor es un número.
isObject (actual , message)	Afirma que un valor es un objeto.
isString (actual , message)	Afirma que un valor es una cadena.
isTrue (actual , message)	Afirma que un valor es cierto.
isTypeOf (expectedType , actualValue , message)	Afirma que un valor es de un tipo determinado.
isUndefined (actual , message)	Afirma que un valor no está definido.

Tabla 4: Principales Métodos de YUI Test

2.4.3.3 Refactorizar para eliminar duplicidad y hacer mejoras.

El verbo refactorizar no existe como tal en la Real Academia Española pero, tras discutirlo en la red, resulta la mejor traducción del término refactoring.

Refactorizar no significa reescribir el código; reescribir es más general que refactorizar; refactorizar es modificar el diseño sin alterar su comportamiento. En este tercer paso del algoritmo TDD, se rastrea el código (también el del test) en busca de líneas duplicadas y eliminarlas refactorizando. Además, se revisa que el código cumpla con los principios de diseño definidos por el centro, en caso contrario, se refactoriza para que así sea.

Si no existe código duplicado, entonces se ha conseguido uno de más calidad que el que presentaba duplicidad. Más allá de la duplicidad, durante la refactorización se permite darle una vuelta de tuerca al código para hacerlo más claro y fácil de mantener. Eso ya depende del conocimiento y la experiencia de cada uno.

Cualquier cambio en los adentros del código, que mantenga su API pública, es una refactorización. La clave de una buena refactorización es hacerlo en pasitos muy pequeños. Se hace un cambio, se ejecutan todos los tests y si todo sigue funcionando, se hace otro pequeño cambio. Cuando se refactoriza, se piensa en global, se ve desde la perspectiva general, pero se actúa en local. Es el momento de detectar malos olores y eliminarlos.

La tarea de buscar y eliminar código duplicado después de haber completado los dos pasos anteriores, es la que más tiende a olvidarse. Es común entrar en la dinámica de escribir el test, luego el SUT, y así sucesivamente olvidando la refactorización.

Conclusiones

En este capítulo se realizó la descripción del procedimiento de pruebas de unidad diseñado. A lo largo del procedimiento se verificaron y se les da respuesta a todos los aspectos que se definieron en un principio como objetivos para desarrollar un proceso correcto y eficaz de pruebas de unidad. Se espera que en los resultados de su aplicación se demuestre su fiabilidad, partiendo de que no existe una iniciativa precedente a la que se expone en este trabajo para la aplicación de pruebas de unidad.

Capítulo 3: Aplicación del procedimiento y evaluación de los resultados.

Introducción

El objetivo del siguiente capítulo es validar y poner en práctica el procedimiento descrito anteriormente, exponiéndose los resultados obtenidos al aplicar el mismo sobre el Subsistema de notificaciones de alertas y avisos en el marco de trabajo Sauxe. Se realizarán las actividades y se documentarán las pruebas llenando la Plantilla de diseño casos de prueba unidad. Se implementarán automáticamente, con el objetivo de llegar a conclusiones respecto a los resultados.

3.1 Descripción del subsistema

El Subsistema de notificaciones de alertas y avisos en el marco de trabajo Sauxe, es una herramienta que permitirá notificar de forma distribuida a diferentes usuarios ya sea por correo por jabber o teléfono de que ha ocurrido un evento en el sistema. Para llevar a cabo esta tarea el mismo se integra con un servidor de ejecución asincrónica de tareas denominado Gearman y con un servidor de Telecomunicaciones Platel.

Sus componentes son: Configuración de Eventos y Avisos, Ejecución de tareas asincrónicas y el Sistemas de telecomunicaciones Platel.

3.2 Requisitos a probar

Se probará dentro del requisito Gestionar eventos, los requisitos funcionales, Inicialización e Insertar eventos.

3.2.1 Diseño de la pruebas.

Se elaboran los casos pruebas para los requisitos que se van a probar apoyándose en la Plantilla de Diseño de casos de prueba unidad.

3.2.1.1 Requisito Inicialización.

Descripción

Prueba de unidad al requisito funcional “Inicialización” el cual se encarga de inicializar la configuración de los componentes implicados en la realización del caso de uso.

Funcionalidades.

Funcionalidades a probar	Descripción
Crear Interfaz	Se valida que se cree la interfaz.
Gestionar Columnas	Se gestiona que se creen las ventanas.

Casos de pruebas

Funcionalidad	Método Utilizado	Recibe	Resultado esperado del método	Resultado esperado de la prueba	Resultado de la prueba
testCrearInterfazGestionar	isTrue	Instancias de las clases necesarias para crear la interfaz	True	Passed	Passed
testInizializarColumnas	areEqual	El número de columnas	2	Passed	Passed

Requisito Insertar evento.

Descripción

Prueba de unidad al requisito funcional Insertar Evento.

Funcionalidades.

Funcionalidades a probar	Descripción
Mostrar ventana	Muestra la ventana de agregación de eventos.

Llenar formulario	Llena el formulario de la ventana de datos.
Aceptar insercion	Acepta la inserción de los datos.
Validacion	Valida que los datos insertados sean correctos.
Notificación	Notifica al usuario que la inserción de los datos se ha llevado de forma satisfactoria

Casos de pruebas

Funcionalidad	Método Utilizado	Recibe	Resultado esperado del método	Resultado esperado de la prueba	Resultado de la prueba
testMostrarInsertarEventos	isNotUndefined	Instancia del objeto ventana para editar eventos	True	Passed	Passed
testValidarFormulario	isTrue	Boolean	True	Passed	Passed
testEnviarDatos	isTrue	Boolean	True	Passed	Passed

3.2.2 Escribir las pruebas

El programador es el encargado de escribir las pruebas que se le realizarán al software definiendo las entradas y salidas esperadas para que el framework pruebe si son correctas o no.

Para escribir las pruebas se debe primero configurar de los elementos de entrada:

1. Creación de la Suit de pruebas

```
var suite  = Ext.test.session.getSuite('App.GestionarEventos'),assert = Y.Assert;
```

Figura 9: Creación de la suit de pruebas.

2. Inicialización de los componentes a probar

```
function buildListarEventos(config) {
    return new gestionarEventos.ListarEventos();
};
function buildEditarEventos(config) {
    return new gestionarEventos.EditarEvents();
};
```

Figura 10: Inicialización de los componentes.

3.2.2.1. Creación del caso de prueba “Inicialización”.

```
suite.add(new Y.Test.Case({
    name: 'Inicialización de componentes',

    testCrearInterfazGestionar: function() {
        var gridlistar = buildListarEventos();
        var wineditar = buildEditarEventos();
        assert.isTrue((gridlistar instanceof Ext.grid.GridPanel &&
        | wineditar instanceof gestionarEventos.EditarEvents));
    },
    testInizializarColumnas: function() {
        var grid = buildListarEventos();
        var colModel = grid.colModel;
        assert.areEqual(2, colModel.getColumnCount());;
    }
}));
```

Figura 11: Creación del caso de prueba “Inicialización”.

3.2.2.2. Creación del caso de prueba “Insertar evento”.

```
    suite.add(new Y.Test.Case({
        name: 'insertarEventos',

        testMostrarInsertarEventos: function() {
            var listar = buildListarEventos();
            listar.MostrarEventos();
            assert.isNotUndefined (listar.winEditarEvents ,"Se ha creado la ventana para editar");
            listar.winEditarEvents.destroy();
        },

        testValidarFormulario: function() {
           var editar = buildEditarEventos();
           assert.isTrue(!editar.form.getForm().isValid());
        },
        testEnviarDatos: function() {
            var editar = buildEditarEventos();
            var values;
            var bool= true;
            var data ={
                eventName:"Evento 1",
                descripcion:"Esto es un evento de prueba",
                ejecucion:"Antes de:",
                dir_action:"Crear proyecto"
            };
            var contador = 0;
            var elements= editar.form.getForm();
            elements = elements.items.items;
            for(var property in data){
                elements [contador].setValue(data[property]);
                contador++;
            }
            editar.sendValues();
            assert.isTrue(bool);
        }
    }));
})();
```

Figura 12: Creación del caso de prueba “Insertar evento”.

3.2.3 Ejecución de las pruebas

El programador ejecutará las pruebas de forma automática, apoyándose de la herramienta descrita en epígrafes anteriores. Para la ejecución de las mismas se realizarán los siguientes pasos.

1. Se abre el fichero ejecutor de pruebas unitarias.html , con el navegador y se presiona el boton Start

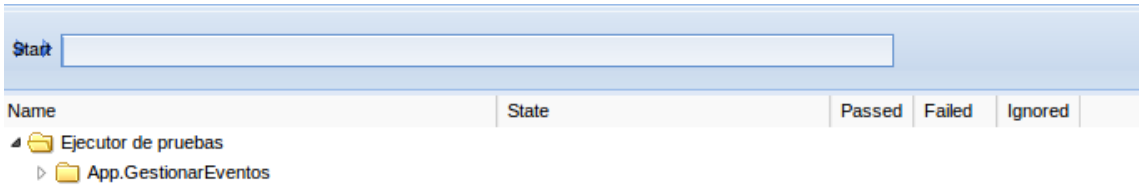

Figura 13: Ejecutar la prueba.

La herramienta muestra una barra de progreso , el área de información de casos de prueba y la consola de notificación de los resultados al usuario .

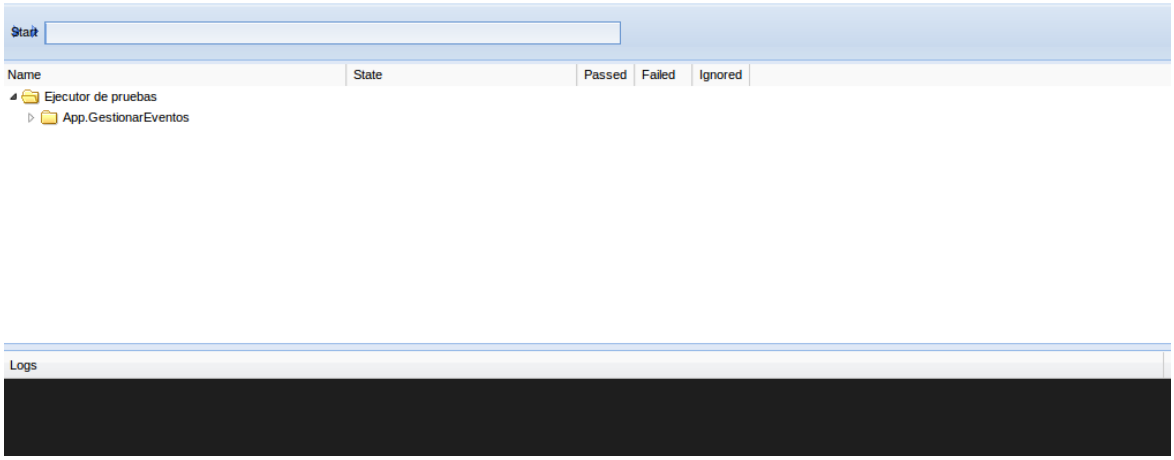

Figura 14: Por ciento de ejecución de la prueba.

Después de llegada la barra de ejecuición al 100% se muestran los resultados obtenidos para los diferentes casos de pruebas como se muestran a continuación.

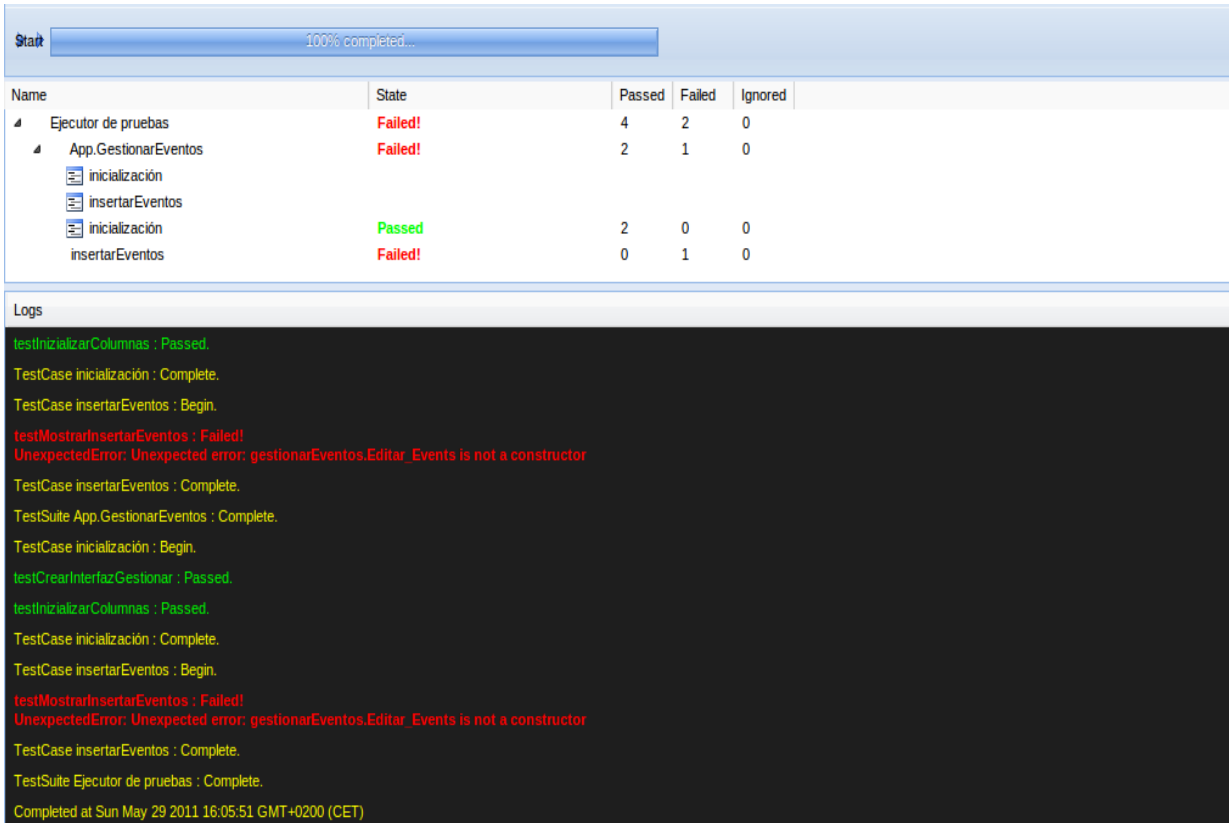

Figura 15: Fallo en la prueba.

Como se puede observar el caso de prueba inicialización pasó, mientras el de insertar eventos falló, por lo que el programador deberá reescribir la prueba, basándose en los errores mostrados en la consola.

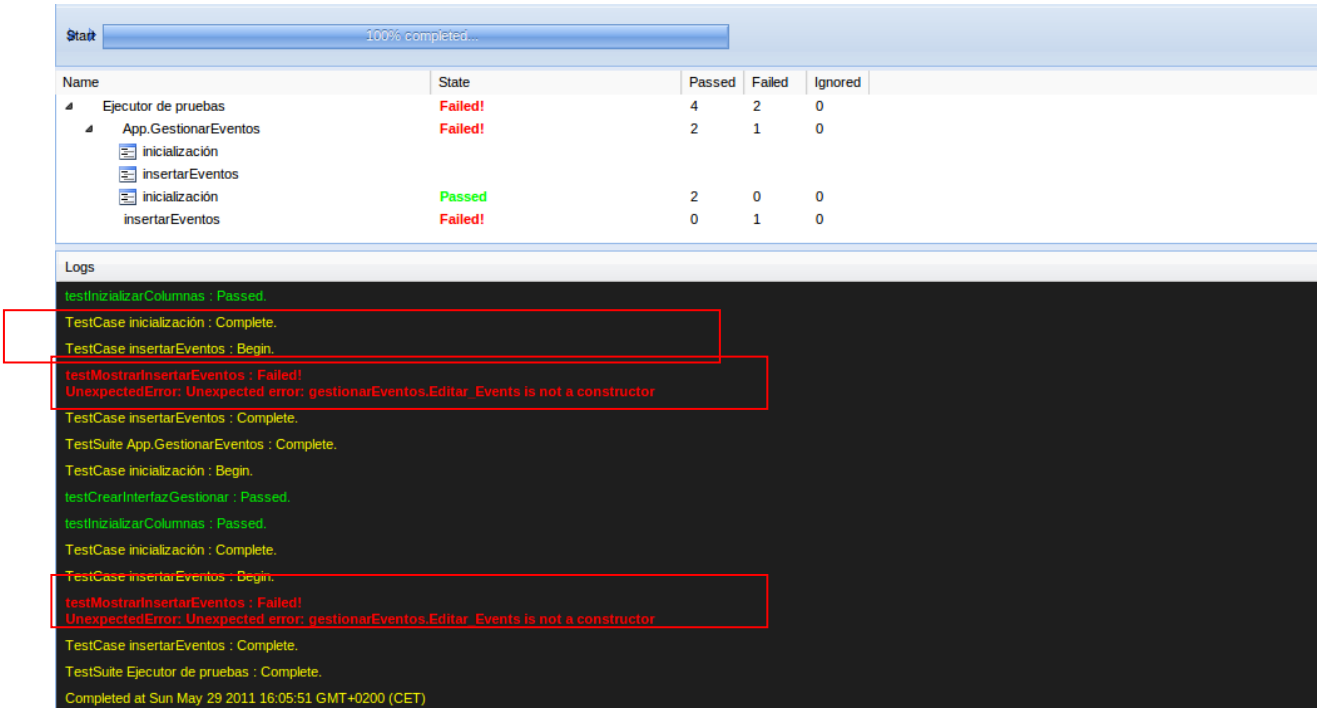

Figura 16: Errores detectados en la prueba.

En este caso los errrores provocados son:

1. La clase gestionarEceventos.Editar_Eventos no existe.
2. La ventana nunca se llegó a mostrar, por lo tanto el formulario no puede enviar los datos.

Después de reescrito el código y arreglado los errores se vuelve a ejecutar la prueba y el resultado obtenido es el esperado.

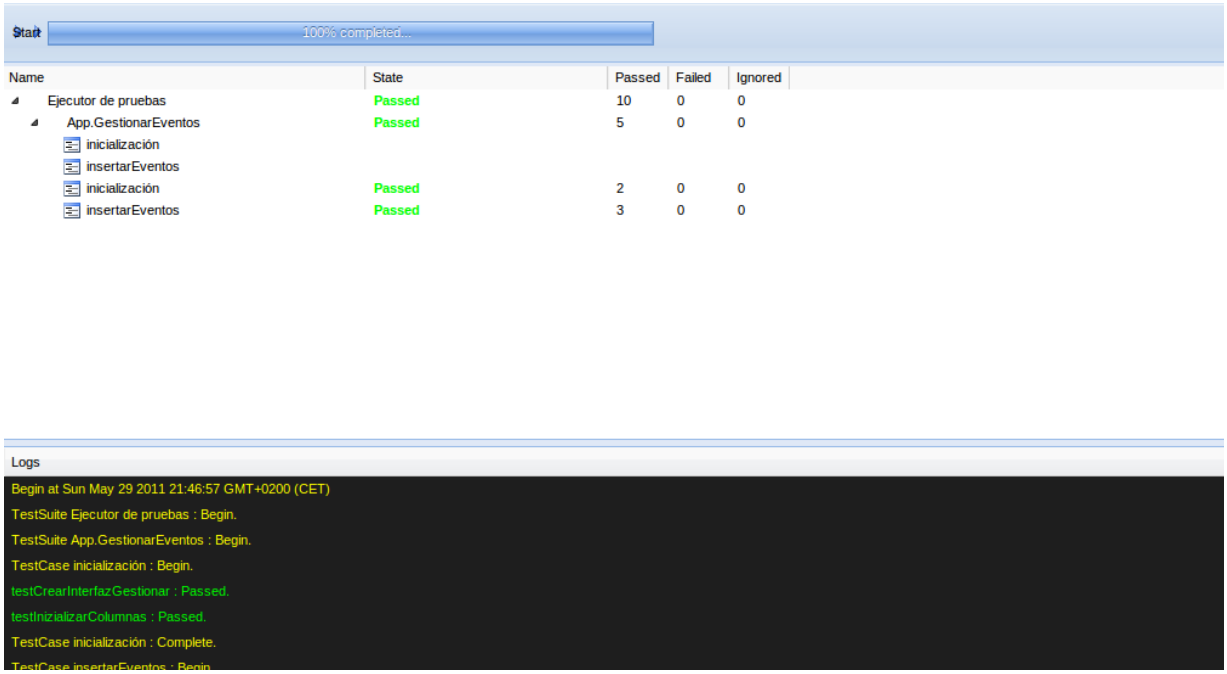

Figura 17: Éxito en la prueba.

3.2.4 Refactorización.

En este caso no fue necesario aplicar la refactorización, ya que los códigos implementados eran claros, no tenían repeticiones ni ciclos y cumplían con los estándares definidos por el centro.

3.3 Resultados Obtenidos.

Al aplicar el procedimiento automático para pruebas de unidad sobre el Subsistema de notificaciones de alertas y avisos en el marco de trabajo Sauxe, se obtuvo una gran satisfacción por el equipo de desarrollo del mismo, ya que fueron capaces de detectar errores en el código que de forma manual no hubieran sido capaces de ver. Además de que se agilizó el desarrollo del producto, sólo se implementó el código necesario para darle solución a los requisitos analizados.

Conclusiones.

En este capítulo se aplicó el procedimiento desarrollado en el capítulo 2, con el objetivo de demostrar su validez y de que sea utilizado para realizar pruebas de unidad en el sistema CEDRUX. Se probó la importancia del mismo para el desarrollo de un software ayudando a que el proceso tenga la calidad requerida y no existan problemas en una etapa avanzada del producto que provoque que haya que empezar el desarrollo desde cero.

Conclusiones generales

Durante el desarrollo de este trabajo se le dio cumplimiento a los objetivos propuestos al inicio del mismo. Se desarrolló y aplicó un procedimiento para pruebas de unidad de acuerdo con las características del sistema CEDRUX, lo que permite su aplicación en cualquiera de los subsistemas que forman parte del mismo

Se elaboró una Plantilla de casos de prueba unidad, se evaluaron los resultados de las pruebas comparando los resultados esperados y los reales, se logró desarrollar una adecuada documentación de todo el procedimiento de pruebas utilizado, el cual servirá de apoyo para futuros subsistemas que se quieran probar.

Partiendo de la generación de casos de prueba, la propuesta de solución que se presenta brinda la posibilidad de documentar las pruebas unitarias realizadas al software.

Al crearse el procedimiento de pruebas de forma automática reduce las inclusiones de errores y aumenta los niveles de calidad que garantizan un software más seguro. Los programadores tienen que esforzarse menos y desarrollan mejor a través de un proceso que integra continuamente el proceso de desarrollo con el de pruebas.

La integración continua es una práctica que ayuda a los equipos a mejorar la calidad en el ciclo de desarrollo. Reduce riesgos al integrar diariamente y genera software que puede ser desplegado en cualquier momento y lugar. Además, aporta mayor visibilidad sobre el estado del proyecto. Dado los beneficios que proporciona, vale la pena sacrificar algo de tiempo para que forme parte de nuestra caja de herramientas.

Recomendaciones

Los objetivos generales de este trabajo fueron alcanzados, pero durante su desarrollo han surgido ideas que serían recomendables tener en cuenta para su futuro perfeccionamiento:

✓ Utilizar en otros proyectos con características similares al sistema CEDRUX el nuevo procedimiento propuesto para las pruebas de unidad.

✓ Brindar cursos sobre pruebas unitarias a estudiantes y profesores de la Universidad de las Ciencias Informáticas, para que así puedan aplicar mejor el procedimiento.

✓ Continuar documentando el proceso de automatización de las pruebas unitarias.

✓ El estudio de los diferentes tipos de pruebas de unidad que existen y que no son desarrolladas en este procedimiento para su posterior aplicación.

✓ Continuar la investigación sobre el resultado obtenido y proponerlo como base referencial o material auxiliar en el desarrollo de futuros trabajos.

✓ Se recomienda que el resultado propuesto en la investigación sea implantado como uso cotidiano y sea introducido en todos los sistemas de gestión de la universidad.

Referencias Bibliográficas

1. Myers, B.A. “User interface software technology”. ACM Computing surveys. 1996

2. Shneiderman, B. “*Designing the user interface”*. Addison Wesley, Reading, Massachusetts. 1998

3. Pressman, R.S. “Ingeniería del Software. Un enfoque práctico”. S.l.: l Mc Graw Hill, Quinta ed. 2002.

4. (1990), IEEE. Computer Dictionary.Computer Society.

5. ISO, (2000). Sistema de gestion de la calidad.Requisitos. 2000

6. 1990, IEEE. IEEE Standard Glossary of Software Engineering Terminology.28 de septiembre de 1990. p. http://www.idi.ntnu.no/grupper/su/publ/ese/iee.

7. Piattini, Mario G. “Análisis y diseño de aplicaciones informáticas de gestión, una perspectiva de ingeniería del software”. 2004.

8. 8. **Myers, Glenford.** “The Art of Software Testing”.s.l. : John Wiley & Sons. ISBN 0-471-04328-1, segunda edición 2004.

9. Miguel-Angel.“Estándar ISO 9126 del IEEE y la Mantenibilidad”.Descripción del concepto de Mantenibilidad según el estándar ISO 9126 del IEEE. http://cnx.org/content/m17461/latest/.

10. López, J., “*Oracle. Fundamentos para el desarrollo de aplicaciones Web”*. MP Edisiones S.A ed, Buenos Aires-Argentina.

11. Jacobson, I., “Proceso unificado del desarrollo del software”. 2003.

12. UCI, D.C.I.d.S. “*Flujo de trabajo de pruebas”*. Asignatura: Ingeniería de Software II Curso: 2008-2009.

13. T. MacCbe. “A software complexity measure”. IEEE Trans. Software Engineering, 2:308-320, 1976.

14. Boris Beizer. “Software Testing Techniques”. Van Nostrand Reinhold, second edition, 1990.

15. Callón, J. “Pruebas unitarias en integración continua”. [cited; Available from: http://javier.callon.org/pruebas-unitarias-en-integracion-continua.

16. Rodríguez, E. Útiles herramientas para desarrolladores de PHP. 2009 [cited; Available from: http://www.tecnovi.net/utiles-herramientas-para-desarrolladores-de-php/.

17. M, I.A. *Mejora de procesos*. 2007 [cited; Available from: http://www.sg.com.mx/sg07/presentaciones/Mejora%20de%20procesos/SG07.P02.Scrum.pdf.

18. Yui Library. *Yui Library.* [En línea] Yahoo.com. [Citado el: 15 de 11 de 2010.] http://developer.yahoo.com/yui/.

Bibliografía.

Biblioteca (Universidad de las Ciencias Informáticas). [En línea] Disponible en: http://biblioteca.uci.cu/

Benchmark – BestPractices.[En línea] 2009 Disponible en: http://www.softwaremetrics.com/benchmark.htm

Benchmarking Software and Discussion. [En línea] 2009 Disponible en: http://www.overclock.net/benchmarking-software-discussion/

IEEE Standard for Software Reviews – IEEE Std 1028-199. [En línea] [Citado el: 8 de marzo de 2010.] Disponible en: http://pesona.mmu.edu.my/~wruslan/SE2/Readings/detail/Reading-6.pdf

La Calidad en general y los Modelos de Calidad. Modelos de Gestion de la Calidad del Software . [En línea] [Citado el: 16 de noviembre de 2009.] Disponible en: http://modelosdegestiondelacalidad.blogspot.com/

Plataforma Moddle UCI. [En línea] 2009 Disponible en: http://teleformacion.uci.cu/

Página oficial de las pruebas TPC. [En línea] 2009 Disponible en: http://www.tpc.org/

ANSI/IEEE Standard 829-1983 for Software Test Documentation. (1983).

Aplicada, ISCA- Ingeniería de Software y Calidad. [http://www.isca.com.ve/productos.htm]. (s.f.). Productos IBM Rational.

B. Boehm, C. A. Software Cost Estimation with COCOMO II. Prentice-Hall. 2000.

Binder, R. Testing Object-Oriented Systems: Models, Patterns and Tools. Addison-Wesley. 2000.

Carr, C. Team Leader's Problem Solver. Prentice-Hall. 2000.

Cay Horstmann, G. C. Core Java 2. Volume 1: Fundamentals. Prentice-Hall. 1999.

Collofello, J. S. Assessing the software process maturity of software engineering courses. 1994.

Cortés, O. H. Aplicación práctica del diseño de pruebas de software a nivel de programación. 2004.

David Sykes, J. M. A Practical Guide to Testing Object-Oriented Software. Addison-Wesley. 2001.

Erich Gamma, R. H. Elements of Reusable Object-Oriented SoftwareAddison-Wesley. 1995.

colaboradores., Carlos Blé Jurado y. [En línea] Enero de 2010. [Citado el: 26 de febrero de 2011.]http://www.dirigidoportests.com/wpcontent/uploads/2010/02/disenoAgilConTdd_ebook.pdf.

Yahoo! Developer Network . *Yahoo! Developer Network .* [En línea] Yahoo, 2011. [Citado el: 26 de Febrero de 2011.] http://developer.yahoo.com/yui/3/test/.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

Green, R. (17 de diciembre 1999). Lista de características de Java que tienden a conducir a errores de programación. *Java Gotchas* .

http://www.adrformacion.com/cursos/calidad/calidad.html . (revisado, 25/4/2010).

http://www.giro.infor.uva.es/Publications/2004/MLC04/JENUI2004.pdf. (revisado, 17/Febrero/2010.).

http://www.ldc.usb.ve/~teruel/ci4713/clases2001/planPruebas.html. (revisado, 16/3/2010.).

Humphrey, W. (2000). *Introduction to the Team Software ProcessSM.* Addison-Wesley.

Humphrey, W. (1989). *Managing the Software Process.* Addison-Wesley . Humphrey, W. S. (2001).

Introducción al proceso de software personal. . Addison Wesley. Ian, S. *Ingeniería de Software.* Séptima Edición, Pearson Education . *ISO 9000:2000 [2000].*

Sistema de Gestión de la calidad. Principios Fundamentales y Vocabulario. Ginebra, Suiza: Secretaria General ISO, Traducción certificada.

Ivar Jacobson, G. B. (2000). *El proceso unificado de desarrollo de software.* Addison Wesley.

Jornada sobre Testeo de Software. [http://web.iti.upv.es/~squac/JTS/JTS2005/contenido.html] . (2005).

Lemus, Y. P. (Ciudad de la Habana : s.n., 2007). *SIMETSE – Sistema de Metricas para evaluar el Software Educativo.*

Glosario de términos

UCI: Universidad de las Ciencias Informáticas

ERP: Conjunto de sistemas de información gerencial que permite la integración de ciertas operaciones de una empresa, especialmente las que tienen que ver con la producción, la logística, el inventario, los envíos y la contabilidad.

Metodología: Conjunto de métodos por los cuales se regirá una investigación científica.

ISO: Organización Internacional para la Estandarización, que regula una serie de normas para fabricación, comercio y comunicación, en todas las ramas industriales.

Framework: Estructura de soporte definida en la cual otro proyecto de software puede ser organizado y desarrollado. Puede incluir soporte de programas, bibliotecas y un lenguaje de scripting entre otros software para ayudar a desarrollar y unir los diferentes componentes de un proyecto.

Artefacto: Productos tangibles del proyecto que son producidos, modificados y usados. Pueden ser modelos, elementos dentro del modelo, documentos, gráficos, código fuente y ejecutables.

Casos de prueba: Conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular, por ejemplo, ejercitar un camino concreto de un programa o verificar el cumplimiento de un determinado requisito. También se puede referir a la documentación en la que se describen las entradas, condiciones y salidas de un caso de prueba.

Fase: Son los pasos en que se descomponen las metodologías. Cada fase puede o no estar subordinada a otra fase, pudiendo existir entre ellas relaciones de dependencia.

Falla: Error en el funcionamiento que emite el sistema.

Prueba: Prueba de software. Ejecución de un sistema bajo condiciones específicas, se observan y se analizan los resultados realizándose una evaluación de los mismos.

Procedimiento: Forma específica de llevar a cabo una actividad. En muchos casos los procedimientos se expresan en documentos que contienen el objeto y el campo de aplicación de una actividad; que debe hacerse y quien debe hacerlo; cuando, donde y como se debe llevar a cabo; que materiales, equipos y documentos deben utilizarse; y como debe controlarse y registrarse.

Producto: Es cualquier cosa que puede ser ofrecida al mercado para su compra, para su utilización o para su consideración. Es cualquier bien, servicio o idea capaz de motivar y satisfacer a un comprador.

Proyecto: Elemento organizativo a través del cual se gestiona el desarrollo del software, el resultado de un proyecto es una versión del producto.

Refactorizar: Modificar la estructura interna de un software con el objeto de que sea más fácil de entender y de modificar a futuro, tal que el comportamiento observable del software al ejecutarse no se vea afectado.

API: Interfaz de programación de aplicaciones (siglas en ingles).

SUT: Subject Under Test. Es el objeto que nos ocupa, el que estamos diseñando a través de ejemplos

Stakeholders: Es un término en inglés para referirse a «quienes pueden afectar o son afectados por las actividades de una empresa».

Anexos

Anexo 1 Plantilla Diseño de casos de prueba unidad v 1.0.

CENTRO PARA LA INFORMATIZACIÓN DE LA GESTIÓN DE ENTIDADES

<<Nombre del subproyecto o área funcional>>

DISEÑO DE CASOS DE PRUEBA DE UNIDAD

<<Especificaciones adicionales: Nombre de artefacto, versión, fecha de actualización>>

Referencia: <<ERP-N-00-i001>>
Actualizado: <<dd/mm/aaaa>>

Control del documento

Título:

Versión:

	Nombre	Cargo
Elaborado por	<<Rango. Nombre y Apellidos>>	<<Cargo>>
Revisado por	<<Rango. Nombre y Apellidos>>	<<Cargo>>

Aprobado por	<<Rango. Nombre y Apellidos>>	**Firma**	
Cargo	<<Cargo>>	**Fecha**	<<dd/mm/aaaa>>

Reglas de confidencialidad

Clasificación: <<Confidencial, Uso Interno, Publicación, otras>>

Forma de distribución: <<PDF Digital>>

Este documento contiene información propietaria del **CENTRO DE SOLUCIONES DE GESTIÓN** y/o **<<CLIENTE>>,** y es emitido confidencialmente para un propósito específico.

El que recibe el documento asume la custodia y control, comprometiéndose a no reproducir, divulgar, difundir o de cualquier manera hacer de conocimientos público su contenido, excepto para cumplir el propósito para el cual se ha generado.

Las reglas son aplicables a las X páginas de este documento.

Control de cambios

Versión	Lugar*	Tipo**	Fecha	Autor	Descripción
0.1			<<dd/mm/aaaa>>	<<Rango. Nombre y Apellidos>>	Creación del documento.

* Sección del documento, Tabla, Figura.
** **A** Alta; **B** Baja; **M** Modificación

Índice de contenidos

Descripción.

[Se describe la prueba que se va a realizar].

Funcionalidades.

[En esta tabla se recogen las funcionalidades del módulo que se va a probar].

Funcionalidades a probar	Descripción
[1: Nombre del requisito]	[Breve descripción de la acción que se realiza en el requisito]

Casos de pruebas

[En esta tabla se recogen los Casos de Prueba de Unidad diseñados para el módulo a probar:

Funcionalidad	Método Utilizado	Recibe	Resultado esperado del método	Resultado esperado de la prueba	Resultado de la prueba
[1] [Nombre de la funcionalidad].	[Nombre del método de prueba empleado]	[Variable(s) que recibe el método].	[Resultado que se espera de él método].	[Resultado que se espera de la prueba].	[Resultado real que da la prueba].

<<Nombre del Subproyecto o Área funcional>>

<<NOMBRE DEL CONTRATO O CENTRO>>

Referencia: <<ERP-N-00-i001>>
Actualizado: <<dd/mm/aaaa>>

Anexo 2 Plantilla de Especificación de requisitos 5.0.

CENTRO DE SOLUCIONES DE GESTIÓN

<<Nombre del subsistema>>

Referencia: <<ERP-N-00-i001>>
Actualizado: <<dd/mm/aaaa>>

Anexos

ESPECIFICACIÓN DE REQUISITOS

<< Nombre de la agrupación de requisitos, versión, fecha de actualización>>

Control del documento

Título: Especificación de Requisitos

Versión:

	Nombre	Cargo	
Elaborado por	<<Rango. Nombre y Apellidos>>	<<Rol>>	
Revisado por	<<Rango. Nombre y Apellidos>>	<<Rol>>	
Aprobado por	<<Rango. Nombre y Apellidos>>	**Firma**	
Cargo	<<Cargo>>	**Fecha**	<<dd/mm/aaaa>>

Reglas de confidencialidad

Clasificación: Entregable.

Forma de distribución: PDF Digital

Este documento contiene información propietaria del **CENTRO DE SOLUCIONES DE GESTIÓN** y/o **CLIENTE,** y es emitido confidencialmente para un propósito específico.

El que recibe el documento asume la custodia y control, comprometiéndose a no reproducir, divulgar, difundir o de cualquier manera hacer de conocimientos público su contenido, excepto para cumplir el propósito para el cual se ha generado.

Las reglas son aplicables a las 72 páginas de este documento.

Control de cambios

Versión	Lugar*	Tipo**	Fecha	Autor	Descripción
1.1			<<dd/mm/aaaa>>	<<Rango. Nombre y Apellidos>>	Creación del documento.

* Sección del documento, Tabla, Figura.

** **A** Alta; **B** Baja; **M** Modificación

Índice de contenidos

1 Introducción

1.1 Objetivo

El objetivo de este documento es describir los requisitos relacionados con la <<Nombre de la agrupación de requisitos>>.

1.2 Alcance

Los requisitos que se describen tienen validez en la fase 1 del proyecto ERP.

1.3 Definiciones y acrónimos

- <<**Plantilla:** Documento de ejemplo que sustenta un formato y que describe los lineamientos para la elaboración de documentos similares>>.

1.4 Referencias

<<**IEEE. 1991.** *IEEE Standard Glossary of Software Engineering Terminology.* Spring 1991 Edition. 1991. IEEE Standard 610.12–1990.>>

2 Especificación de requisito <<Nombre del requisito>>

2.1 Descripción textual del requisito

Precondiciones	<<La precondición declara de qué debe estar seguro el sistema antes de autorizar el inicio del caso de uso. Esto no debe ser chequeado de nuevo mientras dura el caso de uso. Usualmente tener una precondición indica que otro caso de uso está o se ha ejecutado. Una precondición debe escribirse como una afirmación simple en pasado continuo. Por ejemplo: El cliente ha sido validado. (COCKBURN '00). >>
Flujo de eventos	
Flujo básico <<Nombre del flujo básico>>	
1	Seleccionar Activos Fijos
2	
3	
4	
5	
Pos-condiciones	
1	
2	
Flujos alternativos	
Flujo alternativo <<Nº Evento>>.<<letra iniciando por la a>> <Condición que dio lugar a la extensión>	
1	Seleccionar módulo.
2	
Pos-condiciones	
1	N/A
Validaciones	
1	Se validan los datos según lo establecido en el Modelo conceptual <<Referencia al modelo conceptual en cuestión>>.
Relaciones	**Requisitos** Listado de los requisitos incluidos por el requisito.

	Incluidos	<<Evento en el que se incluye>>: <<Nombre del requisito>>, en la agrupación <<Nombre de la agrupación de requisitos>>. Ejemplo: Seleccionar Activos Fijos: Adicionar activos fijos, en la agrupación Gestionar activos fijos.
	Extensiones	Requisitos que extienden a este requisito. <<Evento en el que se extiende>>: <<Nombre del requisito>>, en la agrupación <<Nombre de la agrupación de requisitos>>. Ejemplo: Seleccionar módulo: Adicionar activos fijos, en la agrupación Gestionar activos fijos.
Conceptos	**<<Nombre del concepto>>**	Atributos del concepto que se utilizan en el requisito, tanto internamente como para mostrarlos al usuario. Deben separarse en dos grupos: Visibles en la interfaz: <<Nombre del atributo 1>> Utilizados internamente: <<Nombre del atributo 2>>
Requisitos especiales	Son los requisitos no funcionales específicos para el requisito.	
Asuntos pendientes	Posibles mejoras al requisito.	

2.2 Prototipo elemental de interfaz gráfica de usuario

N/A

2.3 Formatos de entrada/salida

Opcional. En esta sección se especificarán los formatos de entrada y salida de la información, según corresponda, si existe alguno establecido que serán generados por el requisito o que se deben reproducir por el caso de uso exactamente como son en el negocio. Por ejemplo, si en la actualidad se utiliza alguna plantilla para introducir determinados datos y se quiere que la interfaz del sistema sea igual, esta se inserta aquí.

2.3.1 Entradas

<<Imagen que muestra el formato de entrada>>

2.3.2 Salidas

<<Referencia al documento Descripción de Salida del sistema correspondiente>>

<<Nombre del subsistema>>

CENTRO DE SOLUCIONES DE GESTIÓN