INSTITUTO UNIVERSITARIO AERONÁUTICO

FACULTAD DE INGENIERÍA

INTEGRACIÓN DE GESTIÓN DE PRUEBAS A LA ARQUITECTURA DE INTEGRACIÓN CONTINUA DESARROLLADA PARA EL SOFTWARE CIENTÍFICO TÉCNICO

AGÜERO, Esteban Ulises – BIAGETTI, Alejandro Nicolás

Supervisado por:

Ing. MIRA, Natalia – Ing. PICCOLOTTO, Pablo

Dedicatoria

A nuestras familias, amigos y docentes que nos han brindado lo necesario para poder llevar a cabo este proyecto.

Título

Integración de gestión de pruebas a la arquitectura de integración continúa desarrollada para el software científico técnico

Firmas

Declaración de Derechos de Autor

Índice

Diccionario de Términos

A lo largo del desarrollo del presente Trabajo se utilizan términos que no pertenecen a la lengua española pero que sí son de común utilización en los entornos informáticos, es por ellos que continuación se realiza un listado de éstos seguido de su significado o referenciación. Muchos de éstos términos serán abordados en mayor profundidad a lo largo del texto.

Ad-hoc: Locución latina que se traduce literalmente "para esto". Se refiere a algo que es adecuado sólo para un determinado fin o en una determinada situación.

Black-Box: Caja negra. En el contexto de las pruebas de software se denomina así a un tipo de prueba, en dónde el foco principal se centra en las entradas y salidas del elemento estudiado.

Bug Tracking: Seguimiento de errores. Los sistemas de Bug Tracking son utilziados en el aseguramiento de la calidad de software.

Bug: Error que se produce en un sistema informático.

Build: Versión operativa de un producto software que incorpora un subconjunto de funciones que se incluyen en el producto final.

Dashboard: Interfaz o tablero desde dónde se pueden administrar distintas opciones del software.

Develoment manager: Gerente de Desarrollo. Es uno de los roles dentro de los equipo de desarrollo de software.

Fail: Disminutivo de Failure, asociado al significado de falla.

Framework: Entorno de trabajo formado por un conjunto estandarizado de conceptos, prácticas, criterios comunes para enfocar una problemática que sirve como referencia para enfrentar otros problemas de índole similar.

Funcionality: Funcionalidad.

Gray Box: Caja gris. Es uno de los tipos de pruebas utilizados en la ingeniería de software, en la que se combinan acciones de las pruebas de caja negra y de caja blanca.

GUI: Interfaz Gráfica de Usuario.

Happy paths: Caminos felices. Son pruebas en dónde se eligen las situaciones ideales, y se sabe que van a dar pocos problemas.

I/O: Input / Output, entrada y salida (de datos, por ejemplo)

ID: Identificador. Utilizado para identificar un elemento unequívocamente de entre otros elementos.

Input: Entrada de datos.

Integration Testing: Pruebas de Integración.

Log: Historial o registro de sucesos, acontecimientos, eventos o acciones que afectan a un proceso.

Middleware: Software o componente desarrollado que permite la integración de otros componentes o aplicaciones para favorecer la interacción entre ellos.

Open-source: Código abierto. Es el software distribuido y desarrollado libremente.

Output: Salida de datos.

Performance: Rendimiento

Plus: Añadido. Características que se añade a lo normal.

Project manager: Gerente de Producto. Es uno de los roles dentro de los equipo de desarrollo de software.

Quality assurance manager: Gerente de Aseguramiento de la Calidad. Es uno de los roles dentro de los equipo de desarrollo de software.

Regression Testing: Pruebas de Regresión.

Reliability: Confiabilidad.

Script: documentos que contiene un conjunto de intrucciones, escritas en algún lenguaje de programación, para ejecutar diversas funciones.

Software Quality Assurance (SQA): Aseguramiento de la Calidad de Software.

Supportability: Soporte.

Tag: Etiqueta asociada a un elemento.

Test case (TC): Caso de Prueba.

Test Leader | Test Manager | Test Coordinator: Líder, gerente y coordinador de pruebas. Son distintos roles que se peuden encontrar dentro de un equipo de desarrollo de software.

Test Plan (TP): Plan de Pruebas.

Test Run Environment (TE): Entorno de Pruebas.

Test Run (TR): Ejecucion de caso de prueba

Test: Prueba de Software.

Tester: Ingeniero de pruebas

Testware: término que proviene de la combinación de las palabras “test” y “software”. Es una aplicación diseñada para realizar y lelvar a cabo las pruebas de software.

Tracking: Seguimiento, rastreo.

Usabillity: Usabilidad.

Watarfall: Cascacada.

White-Box: Caja blanca. Pruebas basadas en un estudio y examen de los detalles procedimentales del código a evaluar, por lo que se hace necesario el conocimiento de la lógica del software.

Resumen

El presente Trabajo Final de Grado, llevado a cabo por estudiantes de la carrera Ingeniería en Informática dictada en el Instituto Universitario Aeronáutico, presenta una solución a la problemática de planificación de pruebas y su gestión durante el desarrollo de Software Científico-Técnico, mediante la integración de herramientas open-source, logrando así mejoras en el desempeño de los equipos de desarrollo y, en los productos resultantes.

Introducción

El proyecto profesional PIDDEF 42/11 titulado Metodología y Framework de Gestión de Líneas Base de Integración de Aplicabilidad en el Desarrollo de Software para el Proyecto UAV, es un proyecto destinado a ofrecer un modelo de referencia para automatizar aquellas tareas rutinarias que se realizan en el desarrollo de software con el fin de facilitar las tareas de desarrollo de software a los equipos de trabajo. A este modelo se le agrega una serie de herramientas adicionales tendientes a ofrecer un mejor seguimiento de los proyectos que se encuentran en la etapa de desarrollo.

Este proyecto profesional se compone de un servidor con Ubuntu Server. En el cual corre el sistema de integración continua Cruise Control que se ocupa de automatizar el proceso de construcción (building) del proyecto. A su vez, se encuentra un sistema de control de versiones, Git, necesario para mantener un seguimiento (tracking) del proyecto, así como para proveer de las últimas modificaciones al sistema de integración continua. Esto conforma el núcleo de integración continua para líneas bases en un desarrollo de proyecto.

Este trabajo se centrará en la Gestión de Pruebas de la arquitectura de referencia, permitiendo mejorar el seguimiento y resolución de los incidentes que se generan en el momento de la integración de los distintos tests. Esto va a permitir mejorar la fiabilidad, confiabilidad y aseguramiento de la calidad de los productos críticos y la eficiencia de los recursos (tiempos de desarrollo, recursos humanos, equipamiento, etc.) empleados en los proyectos de desarrollo de software crítico, como así también servir de apoyo en los procedimientos de validación y verificación de dichos proyectos.

1.1 Situación Problemática

La Ingeniería de Software abarca variadas áreas de aplicación, las cuales poseen marcadas diferencias. Dentro de esas áreas podemos encontrar el desarrollo de software en ámbitos científicos (en el que nos centramos en éste proyecto) y, el que según la experiencia nos indica, presenta particularidades con respecto al campo comercial, que es con el que generalmente se está acostumbrado a trabajar en la Ingeniería del Software.

En el campo de desarrollo científico el dominio es muy específico y particular, y donde, sólo los científicos expertos en ese dominio poseen el conocimiento para llevar a cabo la solución al problema. Además, en este campo se presenta la inestabilidad de los equipos científicos, el desconocimiento de buenas prácticas para el desarrollo, falta de metodologías adaptadas a la producción de sistemas,

presencia de requerimientos emergentes a lo largo del desarrollo, base de datos acotadas, incertidumbre en outputs, y donde los mayores esfuerzos apuntan a cómo obtener resultados que demuestren estabilidad y precisión, teniendo en cuenta el potencial de operación, pero dejando de lado la reutilización, mantenimiento, documentación, trazabilidad, y demás herramientas y métodos que brinda la Ingeniería de Software.

Es por ello, que se considera sustancial realizar un estudio sobre este campo, a modo de marco contextual, para luego poder realizar una correcta elección de herramientas y de complementación de las mismas que permitan un verdadero acople y equilibrio con el entorno de desarrollo. El presente proyecto se centrará en la Gestión de Pruebas, y en herramientas que nos permitan lograr una mejor organización de pruebas, seguimiento de resultados de las mismas, y proveer medios para lograr trazabilidad entre incidentes y casos de pruebas.

1.2 Problema

El presente trabajo surge en el marco del grupo de Investigación y Desarrollo del Departamento de Informática del Instituto Universitario Aeronáutico, dónde se presenta la necesidad de integración de la Gestión de Pruebas a la Arquitectura de Desarrollo perteneciente al Proyecto PIDDEF 42/11, para mejorar el seguimiento y resolución de incidentes, lograr trazabilidad entre casos de pruebas e incidentes, documentar casos de pruebas, organizar dichos casos de pruebas, y servir de apoyo en los procedimientos de validación y verificación de proyectos

1.3 Objeto de Estudio

El objetivo de estudio que se abordará será la investigación de soluciones de incorporación de la Gestión de Pruebas a la arquitectura de desarrollo del Instituto, estudiando y evaluando distintas herramientas open-source, como así también desarrollando software que permitan la integración entre los distintos componentes de dicha arquitectura y brindando un conjunto de buenas prácticas y procesos que sirvan de referencia a los equipos de desarrollo de software científico-técnico y que, en su conjunto ayuden al aseguramiento de calidad del software desarrollado.

1.4 Objetivos

1.4.1 General

Integrar la Gestión de Pruebas al conjunto de componentes que forman parte del Sistema de Integración Continua de la Arquitectura de Desarrollo del Instituto Universitario Aeronáutico, mediante el desarrollo de middleware que interconecte las herramientas open-source que dan soporte a dicha gestión.

1.4.2 Específico

- Investigar el Entorno-Científico.
- Estudiar las posibles soluciones para la incorporación de la Gestión de Pruebas en la Arquitectura de Desarrollo del IUA.
- Investigar y estudiar las diferentes herramientas disponibles de Gestión de Pruebas open-source.
- Definir criterios de selección, estableciendo valores ponderados de acuerdo a las necesidades.
- Comparar herramientas de Gestión de Pruebas utilizando los criterios antes mencionados.
- Selección una herramienta de Gestión de Pruebas adecuada a las necesidades de la arquitectura existente.
- Integrar la herramienta elegida a la arquitectura existente.
- Desarrollar middleware para la interconexión de componentes.
- Generar referencias de buenas prácticas para los equipos de Desarrollo de Software Científico Técnico.
- Evaluar el funcionamiento de la herramienta en el entorno de pruebas.

1.5 Idea a Defender/Propuesta a Justificar/Solución a comprobar

Lo que se busca con el presente proyecto es brindar una solución viable para la organización, documentación, trazabilidad de los casos de usos, mediante la integración de la Gestión de Pruebas en la Arquitectura de Desarrollo del Instituto Universitario Aeronáutico, en el marco del proyecto "Metodología y Framework de Gestión de Líneas Base de Integración de aplicabilidad en el desarrollo de Software para el proyecto UAV", que es un proyecto destinado a ofrecer un modelo de referencia para automatizar aquellas tareas rutinarias que se realizan en el desarrollo de software con el fin de facilitar las tareas de desarrollo de software a los equipos de trabajo.

1.6 Delimitación del Proyecto

En este proyecto solo se describirá a manera descriptiva el entorno científico-técnico en el que se desenvuelve la Arquitectura planteada. Y, la implementación final de la integración de la Gestión de Pruebas en los servidores reales de la institución se llevará a cabo por personal que posea la autorización correspondiente para realizar dicha tarea.

1.7 Aporte Teórico – Práctico

Nuestro proyecto, mediante la investigación y desarrollo, aportará: el análisis y estudio de la integración de la Gestión de Pruebas en la arquitectura de Integración Continua, el desarrollo de middleware que permita la conexión de los distintos componentes de la Arquitectura de Desarrollo, una mayor documentación, organización y trazabilidad de las pruebas que se llevarán a cabo, brindando también un conjunto de buenas prácticas referenciales para los equipos de desarrollo, teniendo en cuenta que lo antes mencionado, son conceptos que generalmente no se aplican con frecuencia en los entornos de desarrollo científico-técnico, y que, mediante su implementación se puede lograr dar soporte a la validación y verificación del software desarrollado, sumando valor a la calidad del software desarrollado en la institución.

La integración de una herramienta de gestión de pruebas tiene el objetivo de facilitar las tareas de validación y verificación que se realizan obligatoriamente en el desarrollo de software científico técnico, por ser este software de características críticas. Una arquitectura que automatice el proceso de integración de componentes plantea mejoras en la forma de realizar el desarrollo de software ya que propone un esquema donde se integra continuamente, por otra parte también ofrece trazabilidad de los componentes que se desarrollan posibilitando tener un registro de las modificaciones que han ido ocurriendo. Además esta arquitectura se continúa ampliando hacia la gestión de pruebas permitiendo hacer un seguimiento de los casos de pruebas planteados como así también seguir una incidencia ocurrida y verificar su estado de avance. Los beneficios obtenidos con la aceptación de esta arquitectura son muy positivos y el desafío de este proyecto es lograr su uso y aceptación promoviendo la importancia de la validación y verificación del software que se desarrolla en los proyectos de sistemas críticos en el ámbito de Defensa.

1.8 Factibilidad

1.8.1 Factibilidad Técnica

El equipo que llevará a cabo este proyecto cuenta con los conocimientos adecuados para poder concretarlo, los cuales han sido impartidos durante el cursado de la carrera. No obstante, para todos aquellos conceptos de los cuales no se tenga conocimiento, se considera necesaria la investigación en distintos recursos disponibles y que pueden brindarlo, como ser libros, páginas web, blogs, etc. como así también mediante la consulta con docentes expertos en la materia, pertenecientes a la institución.

1.8.2 Factibilidad Operativa

El equipo cuenta con los recursos tecnológicos para poder llevar a cabo éste proyecto, parte de ellos suministrados por responsables de la Arquitectura de Desarrollo de la Institución.

1.8.3 Factibilidad Económica

Económicamente el proyecto es factible debido a que la mayoría de las fuentes de consultas son online y gratuitas, y las herramientas que se consultan se encuentran bajo licencia open-source.

1.9 Métodos de Investigación

Para llevar a cabo éste proyecto se usarán básicamente dos Métodos de Investigación: Empírico y Lógico. Con el primero nos basaremos en la experiencia, experimentación, investigación y percepción, debido a que nos indica qué es lo que existe y cuáles son sus características, pero no nos aclara si algo deba ser necesariamente así y no de otra forma. Mientras que con el segundo (método lógico), nos basaremos en la utilización del pensamiento y en sus funciones de deducción, análisis y síntesis para la resolución de problemas.

1.10 Enfoque Metodológico

La Metodología adoptada para llevar a cabo el proyecto responderá a un enfoque iterativo e incremental, en el cual se irán generando entregables, donde por medio de reuniones con los responsables de la Arquitectura de Desarrollo y el equipo de tesistas y tutores se evaluarán los mismos, y en base a ello se continuará con el proyecto. Cabe mencionar que se hará uso de herramientas que faciliten la organización del proyecto, ellas son Microsoft Project y Trello.

1.11 Planificación del Proyecto

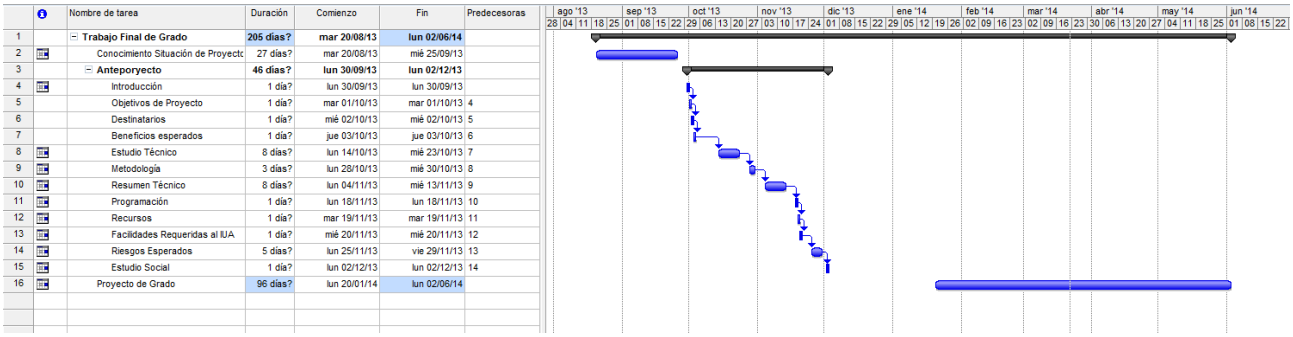
1.11.1 Etapas y Actividades

- **Conocimiento de Situación de Proyecto:** etapa en la que se toma conocimiento de la necesidad de integración de la Gestión de Pruebas en el entorno de desarrollo científico-técnico del Proyecto PIDDEF 42/11.

- **Anteproyecto:** etapa en la que se llevará a cabo la realización del Anteproyecto, donde se expone la posible solución a llevar a cabo, y el análisis de distintas dimensiones.

- **Proyecto Grado:** en esta etapa se procederá al desarrollo de la solución, investigación, desarrollo y documentación del Proyecto en general.

1.11.2 Diagrama de Gantt



	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1	**Trabajo Final de Grado**	**205 días?**	**mar 20/08/13**	**lun 02/06/14**	
2	Conocimiento Situación de Proyecto	27 días?	mar 20/08/13	mié 25/09/13	
3	**Anteporyecto**	**46 días?**	**lun 30/09/13**	**lun 02/12/13**	
4	Introducción	1 día?	lun 30/09/13	lun 30/09/13	
5	Objetivos de Proyecto	1 día?	mar 01/10/13	mar 01/10/13	4
6	Destinatarios	1 día?	mié 02/10/13	mié 02/10/13	5
7	Beneficios esperados	1 día?	jue 03/10/13	jue 03/10/13	6
8	Estudio Técnico	8 días?	lun 14/10/13	mié 23/10/13	7
9	Metodología	3 días?	lun 28/10/13	mié 30/10/13	8
10	Resumen Técnico	8 días?	lun 04/11/13	mié 13/11/13	9
11	Programación	1 día?	lun 18/11/13	lun 18/11/13	10
12	Recursos	1 día?	mar 19/11/13	mar 19/11/13	11
13	Facilidades Requeridas al IUA	1 día?	mié 20/11/13	mié 20/11/13	12
14	Riesgos Esperados	5 días?	lun 25/11/13	vie 29/11/13	13
15	Estudio Social	1 día?	lun 02/12/13	lun 02/12/13	14
16	Proyecto de Grado	96 días?	lun 20/01/14	lun 02/06/14	

1.12 Organización del Trabajo Final de Grado

El presente Proyecto se dividirá en los siguientes capítulos:

- **Marco Contextual:** Explica el marco y entorno en el que se aplica el proyecto. Además se presenta la Arquitectura de Desarrollo PIDDEF 42/11.

- **Marco Teórico:** En este capítulo se explica qué es la Gestión de Pruebas, cuáles son sus componentes. Y en base a ello, determinar que herramienta puede ser integrada en la Arquitectura de Desarrollo de la Institución.

- **Concreción del Modelo y Marco Práctico:** Se llevará a cabo la implementación de la herramienta elegida, y el desarrollo de Middleware que permita la integración de la misma con los distintos componentes que forman parte de la Arquitectura de Desarrollo.

- **Conclusiones:** Se analiza los resultados obtenidos, y en base a ello se establecerán recomendaciones a tener en cuenta al momento de integrar y hacer uso de la Gestión de Pruebas.

Marco Contextual

En este capítulo se explicará el entorno en el que se desarrolló éste proyecto, un entorno de desarrollo científico-técnico, el cual posee características propias, y en el cual se lleva a cabo la Arquitectura de Desarrollo del Proyecto PIDDEF 42/11.

Entorno Científico Técnico

La Ingeniería de Software abarca variadas áreas de aplicación, y, entre ellas existe marcadas diferencias. Dentro de esas áreas podemos encontrar el desarrollo de software en ámbitos científicos (en el que nos centramos) y, el que según la experiencia nos indica, presenta particularidades con respecto al campo comercial, que es con el que generalmente se está acostumbrado a trabajar en la Ingeniería del Software.

En el campo de desarrollo científico el dominio es muy específico y particular, y donde, sólo los científicos expertos en ese dominio poseen el conocimiento para llevar a cabo la solución al problema. Otra de las particularidades de este campo es la inestabilidad de los equipos científicos, el desconocimiento de buenas prácticas para el desarrollo, falta de metodologías adaptadas a la producción de sistemas, presencia de requerimientos emergentes a lo largo del desarrollo, base de datos acotadas, incertidumbre en outputs, entre otras características.

Y más allá de que han sido las aplicaciones científicas uno de los tipos de software más antiguos, por lo que se podría suponer que el avance de dichas aplicaciones ha ido de la mano con la implementación de técnicas desarrollo de software, estudios indican que dicha suposición dista de la realidad, donde las técnicas y herramientas que ofrece la Ingeniería de Software, no son consideradas y tenidas en cuenta al momento del desarrollo de proyectos científicos-técnicos. (1)

Es por ello, que se considera sustancial conocer el entorno en el que se desenvuelve la Arquitectura, para que en base a ello se pueda realizar una elección de herramientas adecuadas, que permitan un verdadero acople y equilibrio entre los distintos componentes intervinientes.

Integración Continua

La Integración continua es una práctica de desarrollo donde los miembros de un grupo de desarrollo integran sus trabajos con frecuencia, por lo menos una vez al día. Cada integración es realizada de forma automática con el fin de detectar los errores de integración lo antes posible. Según Martín Fowler,

muchos equipos de desarrollo han encontrado que este enfoque reduce significativamente los problemas de integración y permite que los equipos desarrollen software cohesivo más rápido.

El principal beneficio de la integración continua es la reducción del riesgo. Se puede predecir el tiempo de integración, puesto que es algo que se realiza de forma continua.

También permite reducir la aparición de bugs, puesto que la realización constante de pruebas permite su pronta detección y corrección antes de que entren en producción.

Ya que se dispone en todo momento de ejecutables del proyecto, esto permite la rápida adopción por parte de los usuarios de las nuevas características añadidas al proyecto. Esto también permite que los usuarios valoren estos cambios y sugieran cambios nuevos de forma rápida. (2)

Proyecto PIDDEF 42/11

La aplicación del presente proyecto se integra al Proyecto PIDDEF 42/11 titulado “Metodologías y Framework de Gestión de Líneas Base de Integración de Aplicabilidad en el Desarrollo de Software para el Proyecto UAV” llevado a cabo en el Instituto Universitario Aeronáutico.

Dicho proyecto cuenta con un framework para el desarrollo de software científico-técnico, con la misión de automatizar parte de los procesos de desarrollo de software a través de la utilización de herramientas open-source y de ésta manera lograr de establecer una solución necesaria para éste ámbito de desarrollo.

El núcleo principal del marco de referencia se compone de un sistema de control de versión, un motor de integración continua, ambos interconectados. Y a esta arquitectura se le complementará con un sistema de Gestión de Pruebas, el cual nos permita mejorar la organización de la documentación de las pruebas, mantener un seguimientos de los resultados de las mismas, y proveer los medias para llevar la trazabilidad entre los incidentes y los respectivos casos de pruebas.

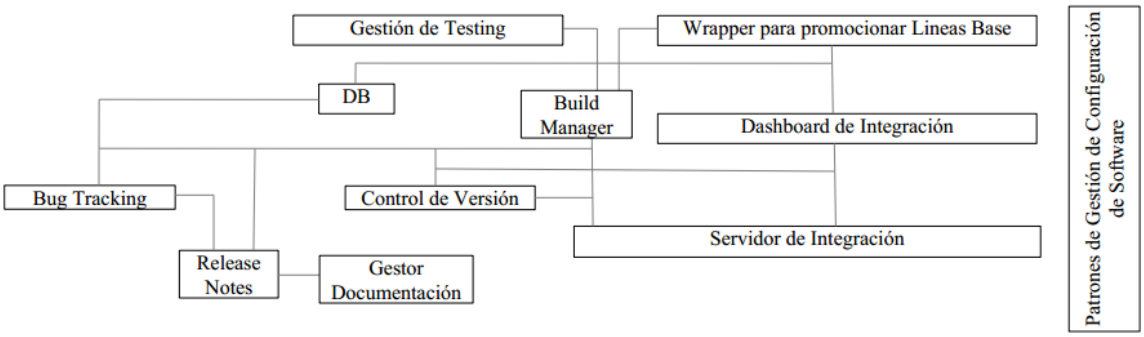
Arquitectura PIDDEF 42/11

- Sistema Operativo (Ubuntu Server)
- Servidor de Control de Versiones (Git)

- Servidor de Integración (Cruise Control)
- Motor de Base de Datos (MySQL)
- Sistema de Gestión de Incidencias (Bugzilla)

Se muestran a continuación los distintos componentes que forman parte de la Arquitectura, y cómo éstos se encuentran interconectados. (1)

Arquitectura



Marco Teórico

1. Calidad

1.1 ¿A qué llamamos calidad?

Es una pregunta que surge frecuentemente y para la cual existen varias respuestas y un difícil consenso para llegar a una unificación del concepto, por lo que podríamos decir que es un concepto que depende de la subjetividad de quien lo desarrolle.

Se podría decir que es una característica que nos permite comparar distintas cosas, virtudes, características, del mismo tipo. Se define, o se calcula, o se le asigna un valor en base a un conjunto de propiedades que podrán ser ponderadas de distintas formas. Lo complejo reside en que cada persona y para cada situación seguramente la importancia de cada propiedad será distinta desde su enfoque. Y también, quizás, la subjetividad sea dependiente del tiempo, porque algo que hoy personalmente considero es de calidad, dentro de un tiempo ya no lo sea. (3)

Jerry Weinberg, nos dice que "…la calidad es valor para una persona", a lo que Cem Kaner añadió "la calidad es valor para una persona a la que le interesa".

Enumeramos a continuación una serie de definiciones acerca de qué representa la calidad.

1. Característica o atributo de una cosa. Y de esta forma definir que la calidad de los productos puede medirse como una comparación de sus características y atributos. Una de las formas de realizar una medida de calidad es mediante la observación de las diferencias ocurridas en la producción de productos iguales, minimizando las diferencias entre las distintas unidades producidas.

2. Calidad suele significar el conjunto de las cualidades. Se encuentra referido a que un cierto producto posee las cualidades que lo determinan como tal.

3. Calidad es la aplicación de principios y técnicas estadísticas en todas las fases de producción.

4. Propiedad o conjunto de propiedades inherentes a una persona o cosa que permiten diferenciarla con respecto a las restantes de su especie, como de mejor o peor calidad.

1.2 Calidad en el Software

Para la definición de qué es la calidad en el software, primeramente, haremos referencia a la definición que enuncia ISO sobre calidad para luego mencionar un conjunto de estándares familiarizados con la ingeniería de software provenientes de ISO 9000. (4)

ISO 9000 define a la calidad como: "Grado en el que un conjunto de características inherentes a un objeto (producto, servicio, proceso, persona, organización, sistema o recurso) cumple con los requisitos".

Estándares relacionados a la calidad en el Software:

- ISO/IEC 9126-1: Ingeniería de software-calidad de producto-modelos de calidad.
- ISO/IECTR 9126-4: Ingeniería de software-calidad de producto-calidad en métricas de uso.
- ISO 9241-11: Guías de usabilidad
- Especificaciones: ISO 20282: Usabilidad en productos. Interfaz e interacción.
- ISO/IEC TR9126-2: Ingeniería de software-calidad de producto-métrica externas
- Especificaciones: ISO 9241: Requisitos ergonómicos para trabajo en oficinas y terminales de trabajo.

Y cómo se mencionó anteriormente, a la hora de definir qué es la calidad, y que ésta representaba un concepto dependiente también de la subjetividad de la persona, se presentan a continuación una serie de distintos puntos de vista de la calidad de software.

Roger Pressman, desde el punto de vista del cumplimiento de los requerimientos define a la calidad de software como *"…El cumplimiento de los requerimientos funcionales y de performance explícitamente definidos, de los estándares de desarrollo explícitamente documentados y de las características implícitas esperadas del desarrollo de software profesional…"*

Watts Humphrey, desde el punto de viste del cliente/usuario nos dice *"…El foco principal de cualquier definición de calidad de software debería ser las necesidades del cliente. Crosby al igual que Pressman define la calidad como conformidad con los requerimientos. Mientras uno puede discutir la diferencia entre requerimientos, necesidades y deseos, la definición de calidad debe considerar la perspectiva de los usuarios. Entonces las preguntas claves son ¿Quiénes son los usuarios? ¿Qué es importante para ellos?*

¿Cómo sus prioridades se relacionan con la manera en que se construye, empaqueta y se da soporte al producto? ..."

Al Davis, define a la calidad de software *como "… La calidad no se trata de tener cero defectos o una mejora medible de la producción de defectos, no se trata de tener los requerimientos documentados. No es más ni menos que satisfacer las necesidades del cliente (por más que las necesidades estén o no correctamente documentadas)"*

El glosario de la IEEE para la ingeniería de software define la calidad de software como *"El grado con el cual un sistema, componente o proceso cumple con los requerimientos y con las necesidades y expectativas del usuario"*

Independientemente de las distintas perspectivas y conceptos desarrolladas, para que cada una de ellas tenga sentido debe ser "medible". Para poder controlar la calidad de software es necesario establecer una serie de parámetros, indicadores y criterios de medición, ya que, como plantea Tom De Marco: *"…no se puede controlar lo que no se puede medir..."*

1.2.1 Atributos de la Calidad de Software

Para la identificación de costos y beneficios del software se definieron los atributos de la calidad. El objetivo es dividir el software en atributos que puedan ser medidos y cuantificados (en términos de costo/beneficio).

Se han definido varios modelos para la clasificación de los atributos. Uno de ellos es el Modelos FURPS+, proveniente de *F*uncionality (Funcionalidad) - *U*sability (Usabilidad) – *R*eliability (Confiabilidad) – *P*erfomance (Prestación) - *S*upportability (Soporte), desarrollado por Robert Grady y Deborah Caswell. (5) (6)

1.2.1.1 Modelo FURPS+

SIGLA	TIPO DE REQUERIMIENTO		DESCRIPCIÓN
F	Functionality – Funcionalidad	Funcional	Características, capacidades y algunas aspectos de seguridad
U	Usability - Usabilidad	Facilidad de Uso	Factores Humanos (interacción), ayuda, documentación
R	Reliability - Confiabilidad	Fiabilidad	Frecuencia de fallos, capacidad de recuperación de un fallo y grado

			de previsión
P	Performance - Prestación	Rendimiento	Tiempos de respuesta, productividad, precisión, disponibilidad, uso de los recursos
S	Supportability – Soporte	Soporte	Adaptabilidad, facilidad de mantenimiento, internacionalización, facilidad de configuración
+	Plus	Implementación	Limitación de recursos, lenguajes y herramientas, hardware
		Interfaz	Restricciones impuesta para interacción con sistemas externos
		Operaciones	Gestión del sistema, pautas administrativas, puesta en marcha
		Empaquetamiento	Forma de distribución
		Legales	Licencia, derechos de autor, etc.

1.2.2 Metodologías - Conformidad

La obtención de un verdadero software con calidad implica la utilización de metodologías o procedimientos estándares para el análisis, diseño, programación y prueba del software que permita unificar la filosofía de trabajo, con el objetivo una mayor confiabilidad, mantenibilidad y facilidad de prueba, a la vez que eleven la productividad, tanto para el desarrollo como para el control de la calidad del software. (7)

Cuando no se cumplen los estándares o procesos de la organización o del proyecto se dice que estamos frente a una no conformidad. Lo esperable es la ausencia de no conformidades (conformidad).

El costo de conformidad (calidad) no es despreciable pero representa una porción menor que el costo del costo de no conformidad.

Crosby describe al costo de no conformidad como aquel en el que se incurre porque el producto o servicio no desarrolló de forma apropiada la primera vez.

Costo de No conformidad (Fallos internos + Fallos externos) + Costo de Conformidad (Prevención + Evaluación) = Costo de la Calidad

La calidad del software claramente puede medirse una vez elaborado el producto, pero esto puede resultar muy costoso si se detectan errores y problemas derivados en las primeras etapas del desarrollo, ya sea en las fases requerimiento o diseño.

Pressman diseña un cuadro con el costo relativo de corregir errores en distintas etapas del ciclo de vida (basado en metodologías tradicionales)

Fase	Requisitos	Diseño	Código	Prueba (desarrollo)	Prueba (sistema)	Explotación, producción
Multiplicador de Coste	1	3-6	10	15-40	30-70	40-1000

Podemos concluir que es mucho menos costoso corregir los problemas en sus fases iniciales que esperar hasta que problema se manifieste a través del usuario final.

1.2.3 Principios básicos del concepto de calidad

- La calidad debe ser una preocupación durante todo el ciclo de vida del software
- Sólo se alcanza con la contribución de todas las personas involucradas
- Debe ser planificada y gestionada con eficacia
- Dirigir esfuerzos a prevención de defectos
- Reforzar los sistemas de detección y eliminación de defectos durante las primeras fases
- La calidad es un parámetros importante del proyecto al mismo nivel que los plazos de entrega, costo y productividad
- Es esencial la participación de la dirección, que ha de proporcionar la calidad.

1.3 Testing

Dijkstra enunciaba allá por 1970, *"…La prueba de software puede ser usada para mostrar la presencia de bugs, pero nunca su ausencia…"*

El testing es una actividad desarrollada para evaluar la calidad del producto, y para mejorarlo al identificar defectos y problemas. El testing de software consiste en la verificación dinámica del comportamiento de un programa sobre un conjunto finito de casos de prueba, apropiadamente seleccionados a partir del dominio de ejecución que usualmente es infinito, en relación con el comportamiento esperado. (8)

Al mencionar que es dinámico se hace referencia a que el programa se verifica poniéndolo en ejecución de la forma más parecida posible a como se ejecutará cuando esté en producción (en contra posición de la técnica estática que se encarga de analizar definiciones, documentación y código).

Por su parte la IEEE nos deja la siguiente definición: *"Es el proceso de evaluar un sistema o componente de un sistema de forma manual o automática para verificar que satisface los requisitos esperados, o para identificar diferencias entre los resultados esperados y los reales"*

Básicamente el testing nos aporta:

- Calidad durante todo el proceso
- Disminución de costos
- Reducción de riesgos
- Optimización de recursos
- Seguimiento de estándares

"Aumentando, administrando y monitoreando la calidad de entregables"

A partir de la información que se puede obtener de los procesos de prueba se pueden tomar decisiones, desde cuándo es el momento oportuno para la liberación de un producto hasta la mejora de diferentes áreas dentro de una organización. En definitiva el testing puede ser considerado un agente de cambio, que involucra actividades cognitivas, imaginación y percepción, donde lo importante es la interpretación de la información obtenida para que todos los actores puedan actuar en forma oportuna donde sea necesario.

1.4 Verificación y Validación

La verificación y validación es el nombre que se le da a los procesos de comprobación y análisis que aseguran que el software que se desarrolla está acorde a su especificación y cumple con las necesidades de los clientes.

La verificación y la validación no representan la misma definición, por ello se menciona a continuación lo que constituye para distintos referentes de la ingeniería de software éstos conceptos. (9)

Para Sommerville:

- Verificación: Busca comprobar que el sistema cumple con los requerimientos especificados (funcionales y no funcionales). ¿El software está de acuerdo con su especificación?
- Validación: Busca comprobar que el software hace lo que el usuario espera. ¿El software cumple las expectativas del cliente?

Para Boehm:

- Verificación: ¿Estamos construyendo el producto correctamente?
- Validación: ¿Estamos construyendo el producto concreto?

Para Ghezzi:

- Verificación: Todas las actividades que son llevadas a cabo para averiguar si el software cumple con sus objetivos.

La IEEE std1012,

- Verificación: El proceso de evaluación de software para determinar si los productos de una determinada fase de desarrollo cumplen las condiciones impuestas en el inicio de esa fase.
- Validación: El proceso de evaluación de software durante o al final del proceso de desarrollo para determinar su cumple los requisitos especificados.

Podríamos decir que el objetivo de la verificación es comprobar si el software es fiel a las especificaciones y requerimientos funcionales y no funcionales detallados. Mientras que el objetivo de la validación, es el aseguramiento que el software construido cumple las expectativas del cliente.

1.4.1 Técnicas de Verificación y Validación

Dentro del proceso unificado de verificación y validación podríamos encapsular las técnicas en dos grupos:

1. Inspecciones del software: técnicas que tienen por objetivo el análisis y comprobación de las representaciones del sistema como los documentos de requerimientos, diagramas de diseño y código fuente del sistema. También se tienen en cuenta los documentos que complementan al

sistema. Forman parte de pruebas estáticas, ya que no es necesario de que el programa se ejecute.

2. Pruebas de Software: técnicas que tienen por objetivo contrastar las respuestas de una implementación del software a seres de datos de prueba y examinar las respuestas del software y comportamiento operacional, para comprobar que se desempeñe conforme a lo requerido. Forman parte de pruebas dinámicas.

1.5 Fallos, Defectos y Errores

1.5.1 Fallos

Un fallo ocurre cuando algo deja de funcionar, cuando debería de hacerlo o como debería de hacerlo. Por ejemplo: Un infarto cardíaco es un “fallo”.

Los fallos están referidos directamente al incumplimiento de las especificaciones tanto explícitas como implícitas. (10)

1.5.2 Defectos

Un defecto es la causa de un fallo. Representa algo en el software que: está, pero no debe; no está, pero debe; No está como debe estar. Por ejemplo: la acumulación de colesterol en el sistema circulatorio es un “defecto”.

La presencia de defecto apunta de forma directa a la validez de los procedimientos de inspección y prueba.

1.5.3 Errores

Un error es la acción que ha provocado la introducción de un defecto en el producto. Por ejemplo: no mantener una dieta sana y equilibrada es un “error”.

1.6 Aseguramiento de la Calidad

Recurriendo un poco a la historia de la Ingeniería de Software nos encontramos que el primer software en hacer uso de la verificación y validación fue el programa del Misil Atlas, que fue utilizado para lanzar satélites y sondas espaciales, a finales de 1950. Desde ese entonces se ha recolectado mucha y variada información que indica que los proyectos a los cuales se les aplica verificación y validación se realizan o ejecutan de mejor manera que los proyectos sin ellas.

Lo que hoy llamamos aseguramiento de la calidad de software evoluciona directamente de la verificación y validación, donde muchas de las tareas que se asocia con el SQA (Software Quality Assurance) son originarias de la verificación y validación. (11)

1.6.1 Definiciones

Al igual que ocurre cuando se define qué es la calidad y otros conceptos donde juega la subjetividad y el punto de vista de quién lo expone, se presentan a continuación una serie de definiciones acerca del aseguramiento de la calidad.

Daniel Galin (punto de vista sistemático):

“Un conjunto, sistemático y planificado, de acciones necesarias para proveer la evidencia adecuada de que el proceso de desarrollo o mantenimiento de un sistema de software cumple los requerimientos técnicos funcionales tan bien como los requerimientos gerenciales para cumplir la planificación y operar dentro del presupuesto confinado.”

El Software Engineering Institute (punto de vista de la visibilidad):

“El aseguramiento de la calidad de software provee claro control del proceso que está siendo usado por el proyecto y del producto que se está construyendo”

Don Reifer (punto de vista del aseguramiento):

"El aseguramiento de la calidad del software es el sistema de métodos y procedimientos y usados para asegurar que el producto de software alcanza sus requerimientos. El sistema involucra la planificación, estimación y monitoreo de las actividades de desarrollo realizadas por otros"

Schulmeyer – McManus (punto de vista de la capacidad):

"Las actividades sistemáticas que proveen evidencia de la capacidad o disponibilidad de uso del producto software total"

IEEE:

"Una guía planificada y sistemática de todas las acciones necesarias para proveer la evidencia adecuada de que un producto cumple los requerimientos técnicos establecidos. Un conjunto de actividades diseñadas para evaluar el proceso por el cual un producto es desarrollado o construido"

1.6.2 Roles de SQA

1. Asegurar que el desarrollo sigue el proceso establecido, auditando los productos del trabajo para identificar deficiencias, determinando el cumplimiento del plan de desarrollo del proyecto y del proceso de desarrollo de software y juzgando el proceso y no el producto.

2. Representar al cliente, identificando la funcionalidad que al cliente le gustaría encontrar, ayudando a la organización a sensibilizarse con las necesidades del cliente, y actuando como un cliente de prueba para obtener una alta satisfacción del cliente.

3. Recabar información acerca de los aspectos esenciales del producto y proceso, y en base a dicha información ayudar en su mejora.

4. Revisar que lo se ha hecho en base a los objetivos sea cumplido para que la gerencia pueda realizar una correcta toma de decisión de negocio.

5. Participar en la definición de planes, procesos, estándares y procedimientos para asegurar que se ajustan a las necesidades del proyecto y que pueden ser usados para realizar evaluaciones de calidad y cumplir los requerimientos del proyecto y las políticas de la organización.

2. Técnicas de Testing

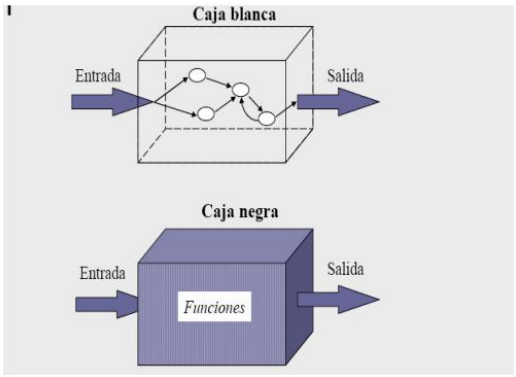
El presente trabajo forma parte de un modelo de integración que permite el aseguramiento de la calidad del software que es desarrollado en la Institución, y gracias a la Gestión de Pruebas se puede llevar a cabo el testeo de diferentes componentes desarrollados e integrados. Es por ello, que a continuación se presentan un conjunto de técnicas aplicables a la hora de realizar el proceso de pruebas. Se comienza por la explicación de técnicas básicas como lo son White-Box y Black-Box, y luego se mencionan un conjunto de técnicas que se adecuan a distintas necesidades del desarrollo, y en particular al desarrollo científico. Además, cómo estamos hablando de automatización, se detalla la diferencia entre pruebas automatizadas y manuales. En nuestro caso se automatizas las pruebas, pero se deja en mención qué técnicas es conveniente automatizar y cuales realizar de forma manual. A modo de recomendación, y para pruebas funcionales repetitivas se aconseja realizar pruebas automáticas, pero sin dejar de lado el valor agregado de tener presente el rol de tester humano que aporta su experiencia en la materia.

A continuación, veremos algunas técnicas básicas y generales del testing como lo son las pruebas Black Box, White Box y Gray Box. También se dará una breve introducción a las pruebas automatizadas y manuales; Testing estático y dinámico.

-***White Box (Estructural):*** Pruebas basadas en un estudio y examen de los detalles procedimentales del código a evaluar, por lo que se hace necesario el conocimiento de la lógica del software.

-***Black Box (Funcional):*** Pruebas teniendo en cuenta las entradas y salidas del software, donde no es necesario conocer la lógica del programa, únicamente la funcionalidad que debe realizar.

A continuación se muestra una esquematización de lo que representan las pruebas White Box y Black Box. (12)



2.1 Pruebas White Box – Estructurales

Este método se centra en cómo diseñar los casos de pruebas teniendo en cuenta el "comportamiento interno y la estructura del programa", examinando lo lógica interna del programa sin considerar los aspectos de rendimiento. (13)

El objetivo de la técnica es el diseño de casos de pruebas para ejecuten al menos una vez todas las sentencias del programa, y todas la condiciones tanto en su flujo verdadero como falso.

Al ser prácticamente difícil realizar pruebas exhaustivas de todos los caminos que puede tener un programa, se han definido un conjunto de criterios de cobertura lógica, lo que nos permite decidir qué sentencias o caminos se deben examinar. Se mencionan y describen brevemente a continuación.

- Cobertura de Sentencias: Se describen casos de pruebas suficientes para que cada sentencia en el programa se ejecute, al menos, una vez.

- Cobertura de Decisión: Se describen casos de pruebas suficientes para que cada decisión en el programa se ejecuten una vez con resultado verdadero y otro con el falso.

- Cobertura de Condiciones: Se escriben casos de pruebas suficientes para que cada condición en una decisión tome todas las posibles salidas, al menos una vez, y cada decisión tome todas las posibles salidas, al menos una vez.

- Cobertura Decisión/Condición: Se escriben casos de pruebas suficientes para que cada condición en una decisión tome todas las posibles salidas, al menos una vez, y cada decisión tome todas las posibles salidas, al menos una vez.

- Cobertura de Condición Múltiple: Se escriben casos de pruebas suficientes para que todas las combinaciones posibles de resultados de cada condición se invoquen al menos una vez.

- Cobertura de Caminos: Se escriben casos de pruebas suficientes para que se ejecuten todos los caminos de un programa. Entendiendo camino como una secuencia de sentencias encadenadas desde la entrada del programa hasta su salida.

2.2 Pruebas Black Box – Funcionales

Son conocidas, también, como Pruebas de Comportamiento, basadas en la funcionalidad o componente a ser probado para elaborar los casos de pruebas. El componente se ve como una caja negra, ya que su comportamiento puede ser determinado estudiando y analizando las entradas y salidas obtenidas a partir del mismo.

Al igual que en las pruebas White Box, la dificultad para llevar a cabo pruebas que conlleven todas las entradas y salidas, sólo se selecciona un conjunto de ellas para el estudio.

Para seleccionar el conjunto de entradas y salidas sobre las que trabajar, hay que tener en cuenta que en todo programa existe un conjunto de entradas que causan un determinado comportamiento erróneo en el sistema, y como consecuencia producen una serie de salidas que revelan la presencia de defectos.

Existen algunos criterios para llevar a cabo los casos de pruebas en Black Box. Algos de ellos son:

- Particiones de Equivalencia
- Análisis de Valores Límites
- Métodos basados en grafos
- Pruebas de Comparación
- Análisis Causa-Efecto

Ahondaremos en las dos primeras: Particiones de Equivalencia y Análisis de Valores Límites.

2.2.1 Particiones de Equivalencia

Éste método divide el campo de entrada en clases de datos de los que se pueden derivar casos de pruebas. La partición equivalente se dirige a una definición de casos de prueba que descubran clases de errores, reduciendo así el número total de casos de usos de prueba que hay que desarrollar.

En palabras más simple, el método divide el dominio de los valores de entrada en un número finito de clases de equivalencia. De esta manera se puede asumir que una prueba realizada con un valor representativo de cada clase es equivalente a una prueba realizada con cualquier otro valor de la misma clase.

Por lo visto, el método está formado por dos fases:

1. Identificación de clases de equivalencia
2. Identificación de casos de pruebas

2.2.1.1 Clase de equivalencia

Representa un conjunto de estados válidos y no válidos para las condiciones de entrada. Éstas clases se van identificando examinando las condiciones de entrada y dividiéndola en grupos. Generalmente se definen dos tipos de clases de equivalencia, las clases de equivalencia válidas, y las clases de equivalencias no válidas, representando casos erróneos.

2.2.1.2 Casos de Prueba

El objetivo principal es disminuir la cantidad de casos de pruebas a partir de la presencia de clases de equivalencias, pero para ello es necesario que:

- Cada clase de equivalencia tenga asignada un número único
- Todas las clases de equivalencias sean cubiertas por los casos de pruebas
- Todas las clases de equivalencias no válidas sean cubiertas

2.2.2 Análisis de Valores Límites

Según estudios y la misma experiencia de tester, los casos de pruebas que exploran las condiciones límite aportan y producen un mejor resultado que aquellos que no lo hacen. Las condiciones límites son aquellas que se encuentran en los márgenes de las clases de equivalencias, tanto para los valores de entrada como para los de salida.

Por lo tanto, el análisis de los valores límites es una técnica que complementa a la anterior, partición de equivalencias de forma que:

- En vez de seleccionar cualquier valor de una clase de equivalencia válida o no, se elige los casos de prueba de los valores extremos.

- En lugar de centrarse solo en el dominio de entrada, los casos de prueba se diseñan también considerando el dominio de salida.

2.3 Gray Box

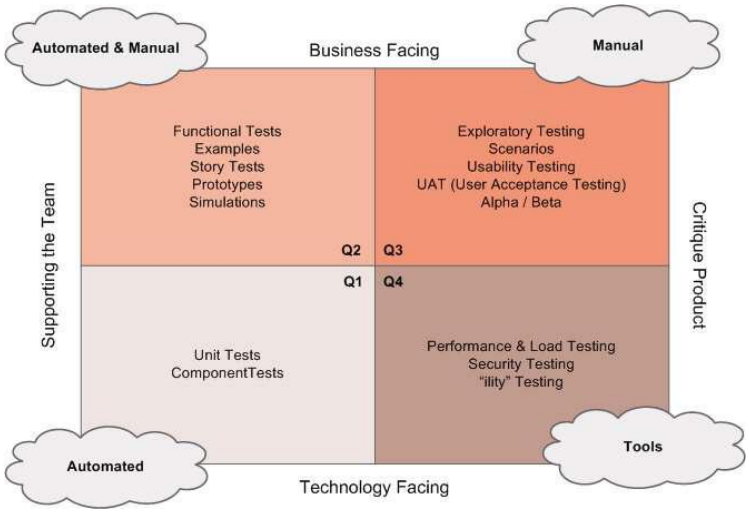
El método es simple, y está basado en la realización de testing Black Box teniendo en cuenta los casos de pruebas realizados por personas (desarrollador) que conocen el programa por dentro. Se puede entender que de esta forma las pruebas realizas pueden ser más efectivas ya que se conoce a priori las secciones del código que presentan mayor conflicto: por complejidad, acoplamiento, etc…

2.4 Testing Manual vs Testing Automático

La definición de ambos tipos de pruebas conlleva a la presencia y participación de las personas. Mientras que las pruebas manuales se llevan a cabo por personas, en las pruebas automáticas son los sistemas de automatización los que tienen el rol principal, aunque estos sean realizados por personas.

En estos términos suele surgir una pregunta: ¿Pruebas manuales o automáticas?

La respuesta rápida puede surgir del pensamiento de la sustitución de horas de pruebas manuales por la rapidez de las pruebas automáticas, que es cierto, pero que no nos asegura tener los resultados esperados ya que dejando de lado las pruebas manuales se está dejando de lado también, el conocimiento, la experiencia de los tester, que aporta mucho valor. Es por ello que de acuerdo a los tipos de pruebas que se estén planteando se utilicen las estrategias más adecuadas. A continuación se muestra un cuadro que resume algunas estrategias usadas.



- Q1: Pruebas unitarias y de componentes. (Automáticas)
- Q2: Pruebas Funcionales, simulaciones, prototipos (Automática/manual)
- Q3: Pruebas de usabilidad, aceptación, exploración, alpha /beta (Manual)
- Q4: Herramientas que se hacen con herramientas

Más adelante se ahondara en las Pruebas Automáticas, en su impacto, métricas, etc.

2.4 Testing Estático y Testing Dinámico

Las pruebas estáticas son aquellas independientes del tiempo, y que se clasifican de esta manera, porque no involucran necesariamente la participación de pruebas automáticas o manuales, como así

también la ejecución del código de la aplicación. Están referidas más que nada a pruebas de revisión de documentos, análisis de flujos, etc. (13)

Las pruebas dinámicas son aquellas que si son dependientes del tiempo e involucran la ejecución de una secuencia específica de instrucciones.

2.5 Taxonomía de las Técnicas de Testing

Se muestra en Anexos una clasificación de distintos esquemas de tipos de pruebas y su relación con las técnicas antes mencionadas (Manual, Automatizadas, Estáticas, Dinámicas, Funcionales, Estructurales) (13)

3. Ciclo de vida del Testing

El desarrollo de software cuenta con varios tipos o estilos de ciclos de vidas con diferentes filosofías, etapas, estructuras, etc. El Testing, si bien forma parte del desarrollo de software y es una etapa en el mismo, también cuenta con un ciclo de vida propio. En el desarrollo de este trabajo se muestran los ciclos de vida más frecuentes, recomendando un ciclo de vida que se adapte mejor a las características del entorno de trabajo tomando las cualidades positivas de cada uno, y en el que se pueda encontrar un marco de colaboración continua.

3.1 Definición y Fases

El testing forma parte del ciclo de vida del Desarrollo de Software, cual sea el enfoque que éste tenga, pero a su vez, el testing también cuenta con su propio ciclo de vida. Claro, esto dependerá mucho de la organización en la que se esté llevando a cabo, como así también, y como lo hemos nombrado anteriormente, del entorno de desarrollo específico (comercial, científico, etc), y dependiendo de esto es que el ciclo de vida contará con más o menos fases, pero en la generalidad de los casos se pueden establecer un conjunto de fases/pasos comunes (según el enfoque propuesto por “UML (The Unified Software Development Process) ISBN 0-201-57169-2”). (14)

1. ***Planificación***
2. ***Análisis***

3. ***Diseño***
4. ***Ejecución***
5. ***Ciclos***
6. ***Pruebas Finales e Implementación***
7. ***Producción***

Surge una pregunta muy importante con respecto al ciclo de vida del testing y sus fases: ¿En qué momento del ciclo de vida se deben comenzar con las pruebas?

La respuesta es “lo antes posible”, que suena sencillo, pero no comúnmente aplicado en los entornos de desarrollo. Donde uno de los eslabones iniciales consiste en que al menos el responsable de calidad del equipo forme parte y asista a las reuniones de toma de requisitos. (13)

En la descripción de las fases del ciclo de vida del testing, encontramos el siguiente desglosamiento:

1. ***Planificación de las pruebas (Fase de definición del Producto):*** Durante esta fase se debe concretar el llamado Test Plan (Plan de Pruebas), el cual representa un documento a alto nivel en el que se describen las estrategias a seguir durante el desarrollo de las pruebas.
   ¿Qué contiene un Test Plan? (Se contemplan algunos de los contenidos de un Test Plan)
   -Alcance de las pruebas
   -Comienzo y fin (planificación)
   -Estrategias (Black Box, White Box)
   -Niveles de pruebas (Integration Testing, Regression Testing, etc)
   -Limitaciones
   -Riesgos
   -Revisiones y entregables
   -Técnicas de Pruebas
   -Hitos
   -Métricas

2. ***Análisis de Pruebas:*** Esta fase es una extensión de la anterior, pero aquí se comenzará a documentar los planes más detallados. Se comienza a analizar los casos de pruebas.
   a. Revisión de Inputs: Se tiene en cuenta el Documento de Requisitos y demás documentos acerca de la planificación del proyecto. El Test Plan comienza a desmenuzarse en pequeñas partes que luego formarán parte de los Test Case (Casos de Pruebas)
   b. Formatos: Generalmente se suele crear una matriz de validación basada en los requisitos, la cual nos ayuda a la hora de ejecutar los Test Case. En esta fase se suelen diseñar las métricas a utilizar.
   c. Test Case: Siguiendo la matriz de validación y los documentos de entrada (inputs) se redactan los Test Case. Se comienza a vincular los requisitos con cada Test Case.

d. Automatización: Se identifica que Test Case pueden ser automatizados.
e. Plan de Regresión: Representa los ciclos de pruebas, es decir la cantidad de veces que se probará la aplicación para verificar que los defectos encontrados no han introducido nuevos errores.

3. ***Diseño de Pruebas:*** Teniendo en cuenta que el ciclo de vida del Testing va transcurriendo en paralelo al del desarrollo, una vez que estemos en esta fase probablemente ya se haya comenzado a desarrollar código software. Durante esta fase los Test Plan, Test Case son revisados y finalizados. Se pueden añadir Test Case y realizar el documento de riesgos. Es el momento para llevar a cabo los scripts de las pruebas automáticas. Finalmente, los informes de pruebas.

4. ***Ejecución de las Pruebas:*** Es el momento en el que el equipo de desarrollo cuenta con una versión estable y lista para testear. Se recomienda es comenzar con un Test de cualificación de Versión (Qualificationn Testing) para evaluar que la aplicación es lo suficientemente estable para comenzar la ejecución de las pruebas. Lo importante es lograr un mínimo de estabilidad. Finalizado esto se puede comenzar a ejecutar los Test Plan, los Test Automáticos, etc. Para llevar el seguimiento de los Test Case se utiliza la matriz PASSED/FAILED para tener una mejor visión del resultado de las pruebas.

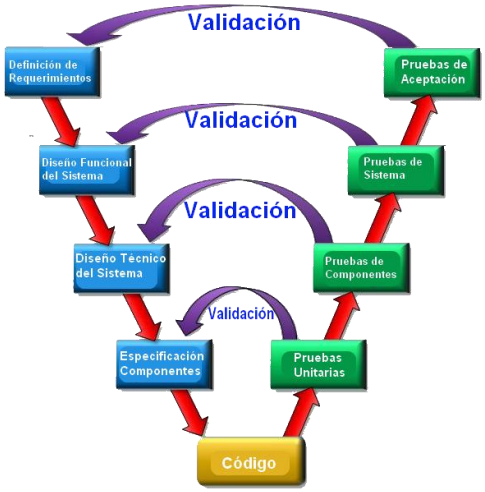
5. ***Ciclos de Pruebas (Regresión):*** En esta fase, una ronda de pruebas seguramente ha sido completada, y errores han sido reportados para que el equipo de desarrollo los solucione. Y una vez que ello suceda, se debería comenzar con la segunda ronda de prueba, la cual podría centrarse solamente en aquellas funcionalidades que habían sido afectadas o se puede aplicar Regressión Testing, probando toda la aplicación nuevamente para descartar que se hayan afectados otras partes de código. Aquí las pruebas automáticas son de gran ayuda para repetir el mismo Test Case una y otra vez.

3.2 Modelos de Ciclos de Vida

Veremos algunos ejemplos de Testing a lo largo del Ciclo de Vida del Desarrollo de Software.

3.2.1 Modelo V

Según lo explica ISTQB ("International Software Testing Qualifications Board") es uno de los modelos de desarrollo más utilizados. Éste modelo está formado por dos ramas, una que representa las fases de desarrollo, y la otra el testing, y dónde para cada etapa del desarrollo existe una correspondiente de testing. (15)

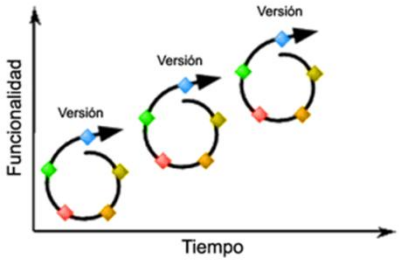


Dependiendo de los proyectos, los productos y los entornos de desarrollo, el Modelo V puede contener más o menos niveles, proponiendo realizar testing en cada una de las fases, desde el comienzo y avanzando durante el desarrollo del proyecto.

Fue creado en respuesta a las deficiencias que presentaba el modelo tradicional en cascada (Watarfall).

3.2.2 Modelo Iterativo

Forma parte de las alternativas a los modelos secuenciales. Aquí se propone una cantidad N de ciclos de desarrollo pequeños, y donde se irán agregando distintas características y funcionalidades.



El testing en cada iteración va a ir siendo mayor ya que debe realizar Regression Testing para asegurar que las nuevas características y funcionalidades desarrolladas no afecten a las ya desarrolladas. En este

modelo cobran relevante importancia las pruebas automáticas ya que nos permite probar las funcionalidades en menor tiempo, teniendo en cuenta que se deben probar en conjunto de acuerdo a lo planteado en los ciclos.

Este modelo tiene la particularidad de que el cliente puede tener soluciones de manera más rápida.

3.2.3 Principios que deberían tener todos los modelos

- Cada actividad del desarrollo de software debería ser testeada
- Ninguna porción del software debería quedar sin probar
- Cada nivel de prueba debería ser probado de forma específica
- Cada nivel de pruebas cuenta con sus propios objetivos de prueba
- Las pruebas llevadas a cabo en cada nivel deben reflejas estos objetivos
- El proceso de pruebas comienza con mucha anticipación a la ejecución de las pruebas
- Revisión de documentos que contengan los conceptos, especificaciones y el diseño global

3.2.4 Testing en entornos Ágiles

Suele presentarse una frase común "…Implementamos metodología ágil (scrum), pero seguimos haciendo testing al final, no tenemos tiempo. El testing siempre retrasa al equipo de desarrollo…"

En un proyecto ágil uno de los principales objetivos es mejorar la calidad del software, y que la compañía que lo lleva adelante puede responder con mayor movilidad, desarrollando de a pequeños pasos y validar dicho desarrollo con el cliente, lo que nos brinda saber si se va por buen camino o si hay que corregir lo desarrollado.

Pero para poder responder rápido es necesario un entorno colaborativo, formado por equipos multidisciplinares capaces de producir incrementos y mejoras en el software con la calidad esperada. Los equipos suelen estar formados generalmente por:

-Técnica (desarrolladores)

-Negocio

-Testers, quienes se encuentran entre la parte técnica y la de negocio. Ayudando a la parte de negocio a traducir lo que el cliente desea e pruebas, que es lo que debe cumplir el software, lo que en definitiva deberán implementar los desarrolladores.

Los tester ágiles tienen una visión intermediaria entre el desarrollo y el negocio, entiende el punto de vista del usuario, pero a la vez, contiene conocimientos acerca del desarrollo de software.

Testers, desarrollo y negocio son un equipo en común, donde si una parte no termina, el trabajo de todos no habrá terminado.

Concluyendo que la frase “QA retrasa al equipo de desarrollo… la culpa es de QA…” no forma parte de un pensamiento ágil.

3.2.5 Pruebas automáticas en Testing Agil

La inviabilidad de tener un equipo de testing manual ejecutando pruebas de regresión cada vez que se hace una integración de código resulta lento, pesado y por consiguiente no ágil. En este enfoque es donde se debe balancear ente pruebas automáticas y manuales.

Las pruebas automáticas permiten verificar seguido y con resultados rápidos, entre otras cualidades adelante descriptas.

Pero esta automatización no significa la eliminación del testing manual. Automatizar significa facilitar el trabajo a los tester para las llamadas “happy paths” (rutas simples), y por otro lado aportar valor desde la búsqueda de errores difíciles de encontrar, prueba de nuevas funcionalidades, y en definitiva aplicar más la experiencia obtenida en el rol de tester.

Por otro lado, la automatización de una conjunto de pruebas y que éstas brinden un informe de que no se ha producido ningún error, proporciona una cuota de seguridad a los desarrolladores, lo que resulta imprescindible para realizar refactorizaciones, mejoras en el código: siendo una herramienta que garantice que los cambios que se han realizado no han modificado el comportamiento del código respecto a lo que había antes.

3.2.6 Recomendación: ¿Qué modelo adoptar?

Anteriormente se han mencionado modelos de ciclos de vidas comúnmente utilizados, pero al momento de elegir alguno de ellos, se recomienda tomar aquellas particularidades que sean positivas de cada uno y que a la vez se puedan adaptar al entorno en el que se apliquen. Es por ello, que teniendo en cuenta las particularidades y características del entorno de desarrollo científico-técnico (mencionadas en los primeros capítulos), se recomienda adoptar las características principales del ciclo de vida iterativo, basándose en Regression Testing para asegurar que las nuevas características y funcionalidades desarrolladas no afecten a las ya desarrolladas. No obstante, también es necesario tener en cuenta las características que brinda los modelos Ágiles, como lo son el tener un equipo multidisciplinario, de espíritu colaborativo, en el que todos converjan en la búsqueda de la calidad de lo desarrollado.

4. Clasificación de las Pruebas según su nivel de componentes

Se menciona a continuación un conjunto de tipos de pruebas de sistemas, cómo los son las pruebas unitarias (mayormente utilizadas/referenciadas en este proyecto), pruebas de integración, sistema, aceptación. (16) (17)

4.1 Pruebas de Unidad

Son pruebas focalizadas en lo que se denomina "unidad de prueba" que, dependiendo del contexto, puede referirse a una clase, un método o un subsistema. Según ANSI/IEEE 1008/1987, define la unidad de prueba de la siguiente forma:

Un conjunto de uno o más módulos de un programa, junto con los datos de control asociados, procedimientos de uso y procedimientos de operación que satisfagan las siguientes condiciones:

1) Todos los módulos pertenecen a un único programa

2) Al menos uno de los módulos nuevos o cambiados del conjunto no ha pasado las pruebas unitarias (dado que una unidad de prueba puede contener uno o más módulos previamente probados)

3) El conjunto de módulos junto con sus datos y procedimientos asociados son el único objetivo del proceso de pruebas

Estas pruebas ocurren, generalmente con el soporte del entorno de desarrollo, tales como un marco de trabajo de prueba de unidad o una herramienta de depuración, y en la práctica, usualmente involucran al programador.

En la orientación a objetos se asume que la unidad de prueba es la clase, por lo que se comprueba que el estado en el que queda la instancia de la clase se está probando es correcto para los datos que se le pasan como entrada.

4.2 Pruebas de Integración

Las pruebas de integración se emplean para comprobar que las unidades de prueba, que han superado sus pruebas de unidad, funcionan correctamente cuando se integran, de manera que lo que se tiende a ir probando es la arquitectura software. Generalmente, las técnicas que más se utilizan durante las pruebas de integración, son las pruebas Black Box, aunque se pueden llevar a cabo algunas pruebas White Box para asegurar que se cubren los principales flujos de comunicación entre las unidades.

Estas pruebas se encargan de las interfaces entre componentes, interacciones a diferentes partes de un sistema, tales como el sistema operativo, sistema de archivos, hardware o interfaces entre sistemas.

En la orientación a objetos, las pruebas de integración pretenden asegurar que los mensajes que fluyen desde los objetos de una clase o componente se envían y reciben en el orden adecuado en el objeto receptor, como así también que produzcan en éste los cambios de estado que se esperaban.

4.2.1 Tipos fundamentales de integración

4.2.1.1 Integración Incremental

Se combina el siguiente módulo que se debe probar con el conjunto de módulos que ya han sido probados.

Ascendente: Se comienza por los módulos hoja.

Descendente: Se comienza por el módulo raíz.

4.2.1.2 Integración No Incremental

Se prueba cada módulo por separado y luego se integran todos de una vez y se prueba el programa completo.

4.2.1.3 Comparación entre Pruebas de Integración Incremental y No Incremental

	Incremental	No Incremental
Ventajas	• Los defectos y errores en las interfaces se detectan antes, ya que se empieza antes a probar las uniones entre los módulos. • La depuración en mucho más fácil, ya que si se detectan los síntomas de un defecto en un paso de la integración hay que atribuirlo muy probablemente al	• Requiere menos tiempo de para las pruebas, ya que se prueba de una sola vez la combinación de módulos. • Existen más oportunidades de probar módulos en

	último módulo incorporado. • Se examina con mayor detalle el programa, al ir comprobando cada interfaz poco a poco.	paralelo.

4.2.1.4 Ad-hoc

Los componentes serán probados, si fuera posible, inmediatamente después de haber sido finalizada su construcción y se hayan finalizado las pruebas de componentes.

4.2.1.5 Prueba de Regresión

Los componentes vuelven a ser probados a la luz de los cambios realizados, ya sea por mantenimiento o desarrollo de alguna nueva versión, buscando efectos adversos en otras partes.

4.3 Pruebas de Sistema

Las pruebas de sistema están relacionadas con el comportamiento de un sistema completo como está definido por el alcance de un proyecto o programa de desarrollo.

En este tipo de pruebas, el entorno de prueba de corresponder al objetivo o entorno de producción final tanto como sea posible para minimizar el riesgo de fallas de entorno específico que no están siendo encontradas en la prueba.

Las pruebas de sistema deben investigar tanto los requisitos funcionales como los no funcionales.

Incluyen: Prueba de funcionalidad, usabilidad, performance, documentación y procedimientos, seguridad y controles, volumen, stress, recuperación, múltiple sitios.

4.3.1 Prueba de Desempeño

Miden tiempo de respuesta, índices de procesamiento de transacciones y otros requisitos sensibles al tiempo. Su objetivo principal es el de verificar y validar los requisitos de desempeño que se han especificado.

Estas pruebas se ejecutan varias veces, por lo general, utilizando distintos tipos de cargas en el sistema, dónde algunas de las características comunes que afectan al desempeño son:

- Errores lógicos
- Procesamientos ineficientes
- Diseños pobres: muchas interfaces, instrucciones i/o.
- Cuellos de botella
- Tiempos de respuestas
- Capacidad de almacenamiento
- Tasas de i/o.
- Número de transacciones que se pueden manejar en simultáneo.

4.3.2 Prueba de Carga

Estas pruebas miden la capacidad del sistema para continuar funcionando apropiadamente bajo diferentes condiciones de carga.

Su meta es determinar y asegurar que el sistema funciona apropiadamente aún más allá de la carga de trabajo máxima esperada.

4.3.3 Prueba de Stress

Nos ayudan a verificar que el sistema funciona apropiadamente y sin errores bajo condiciones de stress como lo son:

- Memoria baja o no disponible
- Número máximo de conexiones
- Múltiple acceso a la misma transacción o funcionalidad con los mismos datos, entre otras.

Su meta es proponerse encontrar errores debidos a recursos o condiciones no apropiadas en el funcionamiento del sistema.

4.3.4 Pruebas de Volumen

Las pruebas de volumen hacen referencia a grandes cantidades de datos para determinar los límites en que se causa que el sistema falle. También identifican la carga máxima o volumen que el sistema puede manejar en un periodo dado. El objetivo es someter al sistema a grandes volúmenes de datos para determinar si el mismo puede manejar el volumen de datos especificados en los requisitos.

4.3.5 Pruebas de Recuperación y Tolerancia a fallas

Estas pruebas aseguran que un sistema se recupere de una variedad de anomalías de hardware, software o red con pérdidas de datos o fallas de integridad.

Las pruebas de tolerancia a fallas aseguran que, para aquellos sistemas que deben mantenerse corriendo, cuando una condición de falla ocurre, los sistemas alternos o de respaldo pueden tomar control del sistema sin pérdida de datos o transacciones.

El objetivo de esta prueba es evaluar las características de contingencia construidas en el sistema para procesar interrupciones y para volver a puntos específicos en el ciclo de procesamiento del sistema. La recuperación debe ser considerada en el proceso de diseño.

4.3.6 Pruebas de múltiples sitios

El propósito general es la evaluación del funcionamiento del sistema o subsistema en múltiples ambientes e instalaciones.

4.3.7 Pruebas de Compatibilidad y Conversión

Estas pruebas tienen por objetivo la comprobación de que el sistema se comporta cómo se espera en relación a la compatibilidad de sus componentes. Aquí se plantean pruebas de compatibilidad entre programas y pruebas de conversión de datos.

Por lo general, las pruebas de compatibilidad se desarrollan cuando se producen reemplazos de partes deficientes en los sistemas.

4.3.8 Pruebas de Integridad de Datos y Base de Datos

La Base de Datos y los procesos de Base de datos deben ser probados como sistemas separados del proyecto, preferiblemente sin realizar el uso de interfaces de usuario.

4.3.9 Pruebas de Seguridad y Control de Acceso

Las pruebas de Seguridad y control de acceso se centran en dos áreas claves de seguridad:

- Seguridad de la aplicación: verifica que un actor solo pueda acceder a las funciones y datos que su usuario tiene permitido.

- Seguridad del sistema: verifica que solo los actores con acceso al sistema y a la aplicación están habilitados para accederla.

El objetivo principal de esta prueba es evaluar el funcionamiento correcto de los controles de seguridad del sistema para asegurar la integridad y confidencialidad de los datos. El foco principal es probar la vulnerabilidad del sistema frente a accesos o manipulaciones no autorizadas.

4.3.10 Enfoque para las Pruebas Funcionales

4.3.10.1 Basadas en Requisitos

Los casos de pruebas se derivan directamente de la especificación de requisitos.

4.3.10.2 Basadas en Procesos de Negocios

Cada proceso de negocio sirve como fuente para derivar/generar pruebas. El orden de relevancia de los procesos de negocio puede ser aplicado para asignar prioridades a los casos de prueba.

4.3.10.3 Basadas en Casos de Uso

Los casos de pruebas se derivan/generan a partir de las secuencias de usos esperados o razonables. Las secuencias utilizadas con mayor frecuencia reciben una prioridad más alta.

4.4 Pruebas de Aceptación

Estas pruebas son generalmente desarrollada y ejecutada por el cliente o un especialista de la aplicación y es conducida a determinar cómo el sistema satisface sus criterios de aceptación validando los requisitos que han sido relevados para el desarrollo, incluyendo la documentación y proceso de negocio.

Evalúan la predisposición del sistema para el despliegue y uso, aunque no es necesariamente el nivel final de la prueba.

Las formas típicas de las pruebas de aceptación incluyen lo siguiente:

4.4.1 Pruebas de aceptación de usuario

Verifica típicamente la aptitud para el uso del sistema por los usuarios del negocio.

4.4.2 Pruebas operacionales

La aceptación del sistema por sus administradores, incluyendo:

- Pruebas de respaldo/restauración
- Recuperación ante desastre
- Gestión del usuario
- Tareas de mantenimiento
- Chequeos periódicos de las vulnerabilidades de seguridad.

4.4.3 Pruebas de aceptación de contrato y regulación

Son realizadas contra criterios de aceptación del contrato para llevar a cabo el software. Estos criterios deben ser definidos cuando el contrato es acordado.

4.4.4 Pruebas alfa y beta (de campo)

4.4.4.1 Pruebas Alfa

Pruebas de aceptación para detectar errores en sistema bajo un ambiente controlado. Son realizadas en el entorno de la organización desarrolladora.

4.4.4.2 Pruebas Beta

Prueba de aceptación donde la validación involucra el uso del software en un ambiente real. Estas pruebas por los usuarios/ usuarios potenciales en su propio entorno.

5. Plan de Pruebas

Introducción

El propósito general de un Test Plan es el de proporcionar la base para llevar a cabo el conjunto de pruebas de una manera organizada.

Desde el punto de vista de la Gestión de Pruebas, es uno de los documentos de mayor importancia dado el grado de ayuda que genera su presencia. Un Test Plan debe encontrarse completo y cuidadosamente desarrollado para cumplir con su misión. (11) (13)

A continuación se mencionan algunos atributos de un buen Test Plan:

- Es una buena oportunidad para detectar la mayoría de los defectos
- Es flexible
- Es ejecutado con facilidad y de forma automática
- Es repetible
- Documenta los resultados esperados
- Define claramente los objetivos de la prueba
- Clarifica las estrategias de pruebas
- Define claramente los criterios de las salidas de las pruebas
- No es redundante
- Identifica riesgos
- Documenta los requerimientos de la prueba

Componentes de un Test Plan (IEEE 829 - Documentación de Pruebas)

Identificador de Plan

Código que representa al Test Plan relacionándolo con su alcance de una forma rápida y clara. Debe contar además con la versión y fecha del Plan.

Ej: TP-Seguridad-v01

Alcance

El alcance especifica el tipo de prueba que se lleva a cabo, como así también las propiedades/elementos a ser probados.

Elementos a Probar

Indica la configuración a probar dentro del alcance del test plan y las condiciones mínimas que debe cumplir para llevar a cabo el Plan.

Estrategia

Engloba la técnica, patrón y herramienta a utilizar en el diseño de los Test Case. Cómo se ha visto en capítulos anteriores, una de las técnicas que se podría utilizar, es la de “Black Box”, o sus variaciones. Se puede especificar la cantidad de Test Case a diseñar, y el grado de automatización a aplicar.

Categorización de la configuración

Determina las condiciones bajo las cuales el Test Plan deber ser:

- Suspendido
- Repetido
- Finalizado

Entregables - Tangibles

Expresa los documentos, herramientas y/o componentes a entregarse al finalizar el proceso previsto por el Plan.

Procedimientos especiales

Establece las herramientas necesarias, como grafos, para preparar y ejecutar las pruebas.

Recursos

Detalla las propiedades necesarias y deseables del ambiente de prueba. Incluye, las características de Hardware y Software, como así también sistemas externos necesarios para llevar a cabo las pruebas, y sus configuraciones.

En esta sección se tiene en cuenta los recursos humanos necesarios para el proceso, y requerimientos especiales del proceso (licencias, tiempos de producción, seguridad, etc.)

Calendario

Se establece los hitos del proceso de pruebas. Puede estar ayudado de un grafo de dependencia con las tareas a realizar.

Manejo de Riesgos

Se explicita los riesgos del Plan, las acciones mitigantes y las de contingencias.

Responsables

Determina quién o quiénes son los responsables de las tareas realizadas en el proceso de pruebas.

6. Test Case

En el desarrollo de software de calidad, tanto el Test Plan como los Test Case son muy importantes. El primero para determinar el ambiente de aplicación de los recursos como el calendario de actividades de las pruebas, el dominio y las características a probar. (13)

Los Test Case bien diseñados tiene gran probabilidad de llegar a resultados más fiables y eficientes, mejorar el rendimiento del sistema, reducir los costos en tres categorías:

a) Productividad - menos tiempo para escribir y mantener los test case,

b) Capacidad de prueba - menos tiempo para ejecutarlos,

c) Programar la fiabilidad - estimaciones más fiables y efectivas.

Definiciones

Se mencionan a continuación una serie de definiciones acerca de los Casos de Prueba:

IEEE 610:

"Conjunto de entradas de prueba, condiciones de ejecución, y resultados esperados desarrollados con un objetivo particular, tal como el de ejercitar un camino en particular de un programa o el verificar que cumple con un requerimiento específico"

Brian Marick:

"Una idea de prueba es una breve declaración de algo que debería ser probado. Ej: Si se prueba la función de una raíz cuadrada, una idea de prueba sería probar un número que sea menor que cero. La idea es chequear si el código maneja un caso de error."

Cem Kaner:

"Un caso de prueba es una pregunta que se le hace al programa. La intención de ejecutar un caso de prueba es la de obtener información."

Componentes de los Test Case

Los casos de pruebas con un conjunto de acciones con resultados y salidas previstas, basadas en requisitos de especificación del sistema. Está formado por:

Propósito: Objetivo de la prueba o descripción del requisito que se está probando.

Método: forma o técnica con la que se probará

Flujo: Secuencia de pasos a ejecutar

Versión: Versión de la aplicación de prueba, el hardware, software, etc.

Datos de entrada: valores que se dan para iniciar la prueba.

Resultados: acciones y resultados esperados o entradas/salidas.

Documentación: Información detallada acerca de la prueba, acompañada de anexos.

Lo más importante que define a un caso de prueba puede ser considerado su flujo, o sea la serie de pasos a ejecutar sobre el sistema, ya sea un conjunto de pasos sobre el camino normal o alternativo; las entradas de datos y por último las salidas esperadas.

Oráculo

Oráculo es el mecanismo, sea este manual o automático, de verificar si el comportamiento del sistema es el deseado o no. Para esto, el oráculo deberá comparar el valor esperado contra valor obtenido, el

estado final esperado con el estado final alcanzado, el tiempo de respuesta aceptable con el tiempo de respuesta obtenido, etc.

Cobertura de Pruebas

Es una medida de la calidad de las pruebas. Primero se identifican cierto tipo de entidades sobre el sistema, y luego se intenta cubrirlas con las pruebas. Es un indicador de si lo que se ha probado es suficiente, y/o si es necesario realizar pruebas sobre otros aspectos.

Factores de calidad de los Test Case

A continuación se listan propiedades que deben cumplir los Test Case para mantener los estándares de calidad:

- Correcto: Debe ser apropiado para los tester y el entorno.
- Exacto: Demostrar que su descripción se puede probar.
- Económico: Tener sólo los pasos y campos necesarios para su propósito.
- Confiable y repetible: Ser un experimento controlado con el que se obtiene el mismo resultado cada vez que se ejecute, sin importar lo que se pruebe.
- Trazable: Saber qué requisitos del caso de uso se prueban.
- Medible: Ser de utilidad a la hora de trabajar con métricas y reportes para los análisis de proyecto.

Formato de los Test Case

- Paso a Paso: Este formato es utilizado en:
    - Reglas de procesamiento
    - Casos de pruebas únicos y diferentes
    - GUI
    - Escenarios de movimiento en interfaces diferentes
    - Entradas y salidas complicados para representar en matrices

- Matrices
    - Formularios con información variada, mismos campos, valores y archivos de entradas diferentes
    - Mismas entradas, diferentes plataformas, navegadores y configuraciones

- Pantallas basadas en caracteres
- Entradas y salidas con mejor presentación matricial

- Scripts automatizados: La decisión de utilizar la automatización de las pruebas depende de la organización y del proyecto que se esté probando.

Los Test Case paso a paso suelen ser más verbales, mientras que el de las matrices más numérico, y los automatizados presentan disminución de los tiempos. Lo ideal es hacer uso de ellos de acuerdo a las necesidades, recursos y entorno. (3)

Proyecto No.: Nombre del Proyecto:	Página No.:
Caso No.: Nombre del Caso:	Ejecución No.: Nombre: Estado de la prueba:
Marca/Subsistema/Módulo/Nivel/Función/Código de la Unidad bajo prueba:	Requisito No.: Nombre:
Escrito por: Fecha:	Ejecutado por: Fecha:
Descripción del caso de prueba (propósito y método):	
Configuración de la prueba para (H/W, S/W, N/W, datos, pre-requisitos de prueba, seguridad y tiempo):	

Paso	Acción	Resultados esperados	Pasado/Fallido
1			
2			
...			

Proyecto No.:			Nombre del proyecto:					Página:		
Nombre de la prueba:			Construcción No.: Fecha de ejecución: Nombre ejecutor:					Ejecución No.:		
Escrito por:			Fecha:					Requisito No.:		
Descripción del caso de prueba (propósito y método):										
Configuración de la prueba:										
Pasado/ Fallido	Usuario	Visualiza	Edición	Adición	Borrado	Reconst.	Auditar	Report.	Seguir	Result.
	1									
	2									
	3									
	...									

Buenas prácticas para el mejoramiento de los Test Case

- Lenguaje: Los Test Case deben estar descriptos de una forma clara y precisa, sin ambigüedades, de manera que sea claro lo que se deba hacer.

- Longitud: Los Test Case no deben contar con muchos pasos, entre 8 y 16 se considera un buen promedio para los métodos paso a paso. Con Test Case "cortos" se pueden reducir tiempos, cometer menos errores. En los casos de los scripts, la longitud no es un parámetro que interese demasiado, ya que la ejecución se realiza en breves segundos, pero es necesaria la administración y mantenimiento.

- Casos acumulativos: Son casos que dependen de otros. Muchas veces es necesario mantener la prueba de forma autónoma hasta que sea posible, ya que esto brinda una mayor flexibilidad en la programación, reduce los costos y tiempos de mantenimiento.

- Administrar las pruebas: La administración de las pruebas incrementa la productividad, su escritura, consulta, movimiento, relación se realicen con mayor practicidad.

Tabla de comprobación de calidad de un Test Case

Atributo	***Lista***	***S/N***
Calidad	Correcto: Apropiado para los Tester y entorno	
	Exacto: Su descripción se puede probar	
	Económico: Tiene sólo los pasos o los campos necesarios para su objetivo	
	Confiable y repetible: Se obtiene el mismo resultado	
	Trazable: Se sabe que requisito se prueba	
	Medible: Representación de métricas útiles para el análisis	
Estructura y capacidad de prueba	Tiene nombre y número	
	Tiene objetivo claro	
	Tiene descripción del método de prueba	
	Especifica la información de configuración, entorno, datos, pre-requisitos, seguridad, etc	
	Tiene flujo de acciones y resultados esperados	

	Guarda el estado de las pruebas, como informes o capturas de pantalla	
	Mantiene el entorno de pruebas limpio	
	No supera los 16 pasos	
	La matriz no demora más de 20 minutos para probarse	
	El script automatizado tiene propósitos, entradas y resultados esperados.	
	La configuración ofrece alternativas a los pre-requisitos de la prueba cuando es posible.	
Administración	Emplea convenciones de nomenclatura y numeración	
	Su versión coincide con el software bajo prueba	
	Incluye objetos de prueba necesarios para el caso, ej: BD	
	Realiza copias de seguridad	

7. Automatización

Consiste en ejecutar los casos de prueba en forma automática, leyendo la especificación del mismo de alguna forma, que puede ser un lenguaje genérico o propio de una herramienta, hojas de cálculo, etc...

La principal razón para automatizar las pruebas de software es el tiempo, y realizando este tipo de pruebas se puede ahorrar mucho en el mediano y largo plazo.

Otra ventaja importante es la inexistencia de error humano, ya que realizar la misma prueba una y otra vez es rutinario y propenso a errores. (3)

Algunas ventajas son:

- Reduce riesgos: proveyendo más cobertura de pruebas, las pruebas automatizadas reducen riesgos de fallos, ya que una vez escritas se ejecutan una y otra vez de la misma forma.

- Permite ejecuciones más rápidas: Debido a que por medios informatizados se pueden lograr órdenes de procesamiento de una magnitud más rápida que los humanos. Ejecutando scripts, más pruebas pueden hacerse en menos tiempo.

- Facilita una gran cobertura de pruebas: Los test automáticos soportan la ejecución de scripts a través de los navegadores populares, sistemas operativos y más. Las pruebas de regresión para proyectos cambiantes y ambientes automatizados son más fáciles que con el proceso manual. Con este tipo de pruebas cualquier cantidad de transacciones y trabajo acumulado puede ser emulada. Cualquier set de entradas puede ser probado rápidamente.

- Encuentra más defectos temprano: Este tipo de pruebas da a los desarrolladores una forma rápida de replicar y documentar los defectos en el software, ayudando a agilizar el proceso de desarrollo mientras se verifica la correcta funcionalidad a través de todos los ambientes, data-sets y procedimientos de negocio.

- Formaliza el proceso: La introducción de test automatizados fomenta a los equipos de desarrollo a formalizar el proceso, resultando en más consistencia y documentación.

- Facilita el reúso de los test: Una vez que los test han sido guionados (scripting), los desarrolladores pueden reusarlos a medida que se hacen cambios en la aplicación. No hay necesidad de hacer nuevos test para la misma funcionalidad.

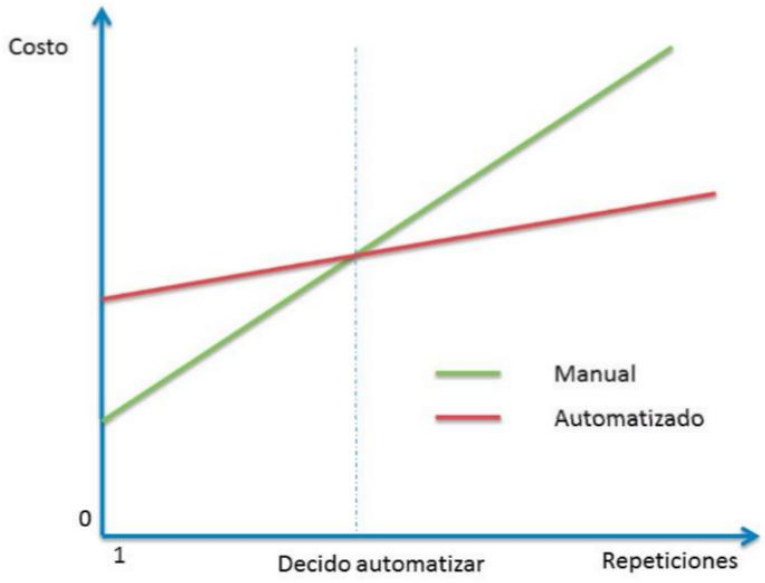
Este tipo de pruebas también acarrea ciertas desventajas:

- El esfuerzo inicial es mayor: La creación de un guion (script) automatizado suele ser más costoso que la prueba de un test case manual
- No suele ser viable automatizar referencias visuales en una pantalla, como colores o la ubicación de un objeto
- El mantenimiento de los guiones (scripts) puede ser muy costoso, en caso que los requerimientos cambien constantemente
- Las herramientas de automatización no pueden medir la usabilidad de la aplicación
- Se requieren conocimientos de programación para adaptar los guiones (scripts) automatizados a los requerimientos

Otros aspectos a tener en cuenta es que el costo de las pruebas automáticas depende considerablemente del conocimiento de la herramienta que se planea usar para realizar el mismo. También hay que considerar la cantidad de ciclos de testing, ya que la utilidad de este tipo de pruebas se ve en los casos en donde se repiten muchas veces, tiempos de pruebas, conocimientos de la herramienta y de programación.

Como se dijo anteriormente, el automatizar o no las pruebas de un proyecto es algo que hay que examinar cuidadosamente en las primeras etapas del ciclo de desarrollo. Hay que tener en cuenta factores como la cantidad de versiones del producto a probar, las distintas plataformas en las que se probara, o incluso los datos, ya que probar reiteradamente lo mismo con distintos datos de pruebas es algo a tener presente. Más allá de estas condiciones, lo más importante a tener en cuenta es ver si el esfuerzo de realizar las automatizaciones (configurar entorno, planear las pruebas, escribir los scripts, etc...) es menor al esfuerzo de realizar las pruebas manuales.

Como se ve en el siguiente gráfico, la cantidad de veces que las pruebas se repetirán es un factor clave a la hora de automatizar o no una prueba.



Como se ve el costo de una repetición es mayor para una prueba automatizada que para una manual. El punto en donde las rectas se cruzan, es el número de repeticiones que marca la inflexión. Si el caso de prueba lo ejecutaremos menos de esa cantidad de veces es mejor no automatizar, si vamos a ejecutarlo más veces entonces es mejor automatizar.

8 Roles

Las actividades de pruebas de los proyectos pueden ser realizadas por diferentes perfiles de personas dependiendo muchas variables, como son: el tipo de proyecto, su organización, el tipo de organización, la experiencia de los miembros, etc...

A pesar de esta variedad, las actividades de prueba se pueden agrupar en dos roles principales, *el test leader y el tester.*

Las actividades y tareas desarrolladas por las personas en estos roles depende del proyecto, producto, persona en el rol y de la organización.

Test Leader

El test leader es llamado también test manager o test coordinator.

El rol del test leader puede ser ejecutado por un project manager, develoment manager, quality assurance manager o el manager del grupo de test. Generalmente el test leader planea, monitoriza y controla las actividades de test. (18) (15)

Típicamente las tareas de un test leader incluyen:

- Coordinar la estrategia de pruebas y el plan con el project manager y otros interesados.
- Escribir y revisar la estrategia de pruebas del proyecto y las políticas de pruebas de la organización
- Contribuir con la perspectiva de prueba a otras actividades, como la planificación de integración
- Planificar los tests (considerando el contexto y entendiendo los objetivos y riesgos) incluyendo enfoques de pruebas, estimando el tiempo, esfuerzo y costos, adquisición de recursos, definición de niveles de pruebas, ciclos y planificación del manejo de incidentes.
- Iniciar la especificación, preparación, implementación y ejecución de los test, monitorear sus resultados y chequear sus salidas.
- Adaptar la planificación basada en los resultados de las pruebas y los progresos realizados, tomando las acciones necesarias para compensar los problemas
- Preparar la configuración adecuada del testware para realizar la trazabilidad.
- Introducir métricas adecuadas para medir el progreso de las pruebas y evaluar la calidad del testing y del producto.
- Decidir qué debería ser automatizado, en qué nivel y cómo.

- Seleccionar las herramientas para ayudar en las pruebas y organizar cualquier capacitación sobre las mismas.
- Decidir sobre la implementación del ambiente de pruebas.
- Escribir un reporte resumido de las pruebas que se realizaron durante el proyecto

Tester

Típicamente las tareas de un tester incluyen:

- Revisar y contribuir al test plan.
- Analizar, revisar y evaluar los requerimientos de usuario, especificaciones y modelos de la capacidad de prueba.
- Crear las especificaciones de prueba.
- Configurar el ambiente de prueba (generalmente coordinando con el administrador de sistemas y redes).
- Preparar y adquirir la información de prueba.
- Implementar las pruebas en todos los niveles, ejecutar y llevar registro (log) de las pruebas, evaluar los resultados y documentar las desviaciones de los resultados esperados.
- Usar las herramientas de administración de pruebas y monitoreo que sean necesarias.
- Automatizar los test (puede necesitar la ayuda de un desarrollador o un experto en automatización de pruebas).
- Medir la performance de los componentes y sistemas (de ser necesario).
- Revisar las pruebas desarrolladas por otros tester.

Las personas que trabajan en análisis de test, diseño de test, o tipos específicos de test o automatización de tests, pueden ser especialistas en sus roles. Dependiendo el nivel de pruebas y los riesgos relacionados con el producto y el proyecto, diferentes personas pueden ocuparse del rol de tester, manteniendo algún grado de independencia.

Típicamente los testers a nivel de componentes e integración son los desarrolladores, los testers a nivel de aceptación suelen ser los expertos de negocio y usuarios, y los testers para la aceptación operacional suelen ser los operarios.

Adicionalmente a estos roles se puede encontrar alguien que organice los distintos proyectos de la organización, a quien se le suelen presentar las métricas finales del proyecto y que asigna recursos en base al rendimiento de estos. Esta persona podría ser un líder de área, jefe de desarrollo, etc...

9 Métricas

Las métricas son una medida cuantitativa del grado en que un sistema, componente o proceso posee un determinado atributo. Estas ayudan a estimar el progreso y la calidad del proceso de pruebas. (19) (20)

El tipo de métricas a usar depende de muchos factores, como son la organización, la experiencia del equipo, el tipo de proyecto, su tamaño, etc.

Entre las ventajas de usar métricas en las pruebas se encuentra:

- Tomar decisiones para la siguiente fase de actividades
- Evidencia de lo que se dijo o de las predicciones planteadas
- Entendimiento del tipo de mejoras requeridas
- Tomar decisiones respecto a cambios en el proceso o en el uso de tecnologías.

9.1 Tipos de métricas

Métricas de proceso: Pueden ser usadas para mejorar la eficiencia del proceso de desarrollo.

Métricas de producto: Miden la calidad del producto de software.

Métricas de proyecto: Pueden ser usadas para medir la eficiencia del equipo del proyecto o herramientas usadas por el equipo.

9.2 Métricas de pruebas manuales

Las métricas de pruebas manuales son clasificadas en dos clases, base y calculadas.

Métricas Base

Las métricas base es información "cruda" obtenida durante el desarrollo y ejecución del proceso de pruebas (número de casos de prueba ejecutados, número de casos de prueba, etc).

Métricas Calculadas

Por otro lado las métricas calculadas son derivadas de la información obtenida de las métricas base (Porcentaje de casos de pruebas completos, porcentaje de casos de prueba exitosos, etc)

9.3 Ejemplos de métricas

Entre las métricas base que se pueden usar se encuentran:

- Número de casos de prueba ejecutados
- Número de casos de prueba que resultaron exitosos
- Número de casos de prueba que fallaron
- Número de casos de prueba bloqueados
- Número de casos de prueba no ejecutados
- Número de defectos identificados
- Gravedad de los defectos identificados

Algunas métricas calculadas partiendo de las métricas bases identificadas son:

- ***Porcentaje de casos de prueba ejecutados:*** Usado para obtener el porcentaje de casos de prueba ejecutados

  R = (N° de casos de prueba ejecutados / N° de casos de prueba totales) * 100

- ***Porcentaje de casos de prueba no ejecutados:*** Usado para obtener el porcentaje de casos de prueba no ejecutados

  R = (N° de casos de prueba no ejecutados / N° de casos de prueba totales) * 100

- ***Porcentaje de casos de prueba exitosos:*** Usado para obtener el porcentaje de casos de prueba que pasaron las mismas

  R = (N° de casos de prueba exitosos / N° de casos de prueba totales) * 100

- ***Porcentaje de casos de prueba fallidos:*** Usado para obtener el porcentaje de casos de prueba que fallaron las pruebas

  R = (N° de casos de prueba fallidos / N° de casos de prueba totales) * 100

- ***Porcentaje de casos de prueba bloqueados:*** Usado para obtener el porcentaje de casos de prueba que quedaron en estado bloqueado

  R = (N° de casos de prueba bloqueados / N° de casos de prueba totales) * 100

- ***Densidad de defectos:*** Esta métrica es la cantidad de defectos encontrados por tamaño. El tamaño varía de acuerdo a las características del proyecto, siendo por ejemplo la cantidad de defectos por 100 líneas de código.

  R = N° de defectos /tamaño

- ***Eficacia en la eliminación de defectos:*** Es usado para identificar la eficiencia de las pruebas del sistema

  R = (N° de defectos encontrados en QA / (N° de defectos encontrados en QA + N° de defectos encontrados por el usuario final)) * 100

- ***Limpieza de código:*** Muestra que tanto falla el código entregado para probar

  R = test que fallaron alguna vez / test totales

- ***Defectos introducidos:*** Cantidad de defectos introducidos tras la primera ejecución de pruebas

  R = (defectos totales - defectos encontrados en primera ejecución de pruebas)/ defectos encontrados en primera ejecución de pruebas

Concreción del Modelo

Investigación y Elección de herramienta de gestión de pruebas

Planteada la necesidad de integrar gestión de pruebas a la arquitectura del proyecto PIDDEF 4211, se realizó la investigación y la posterior comparación y elección de herramientas open-source que den respuesta a la problemática.

A continuación se menciona una serie de herramientas con sus respectivas descripciones de propiedades básicas, adjuntando para cada caso la fuente oficial de información.

Seguido de ello se plantea un cuadro comparativo, con distintas consideraciones planteadas de acuerdo a las necesidades de la arquitectura de desarrollo. Luego, se pondera dichos atributos, para que por un criterio matemático quede definido cuál es la herramienta que se considera que mejor se puede adaptar y responder como solución.

TestLink

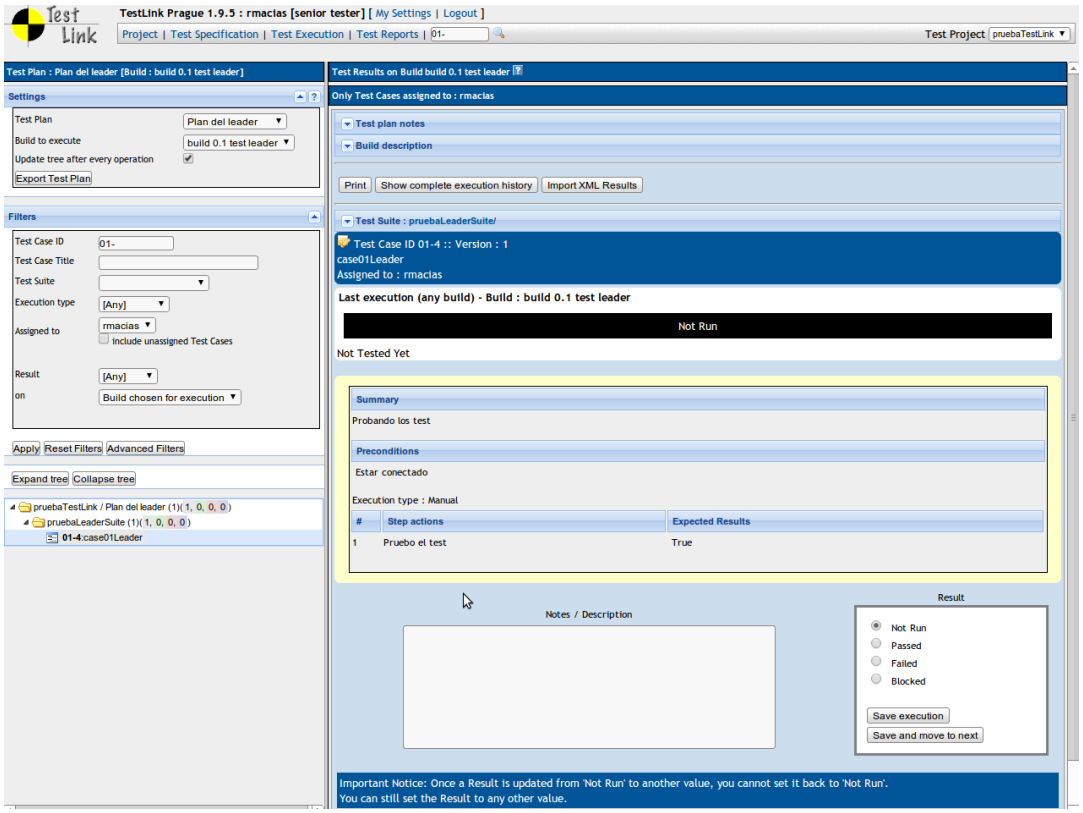
TestLink es una herramienta open-source para la gestión de pruebas que facilita la aseguración de la calidad de software. Desarrollado y mantenida por TeamTest.

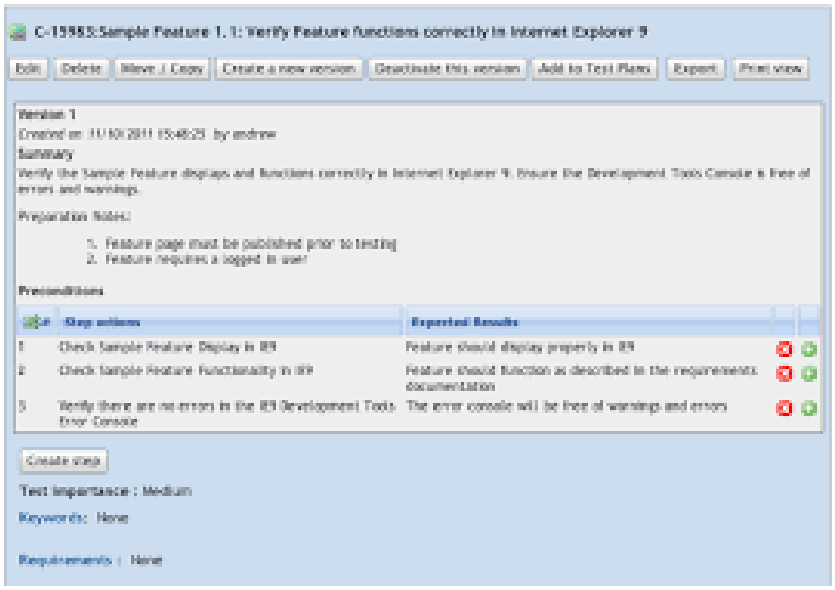
TestLink ha mostrado una larga evolución con el paso de los años, ofreciendo en sus últimas versiones mayor estabilidad y proyección, permitiendo la posibilidad de desarrollar la gestión integral de testing mediante la gestión de Test Plan.

Se encuentra desarrollada en PHP y MySQL.

Para mayor conocimiento de la herramienta se recomienda visitar su página web: http://testlink.org/

A continuación se muestran imágenes ilustrativas de TestLink:


TestLink Prague 1.9.5 : rmacias [senior tester] [My Settings | Logout]
Project | Test Specification | Test Execution | Test Reports | 01-
Test Project pruebaTestLink
Test Plan : Plan del leader [Build : build 0.1 test leader]
Settings
Test Plan
Plan del leader
Build to execute
build 0.1 test leader
Update tree after every operation
Export Test Plan
Filters
Test Case ID
01-
Test Case Title
Test Suite
Execution type
[Any]
Assigned to
rmacias
include unassigned Test Cases
Result
[Any]
on
Build chosen for execution
Apply
Reset Filters
Advanced Filters
Expand tree
Collapse tree
pruebaTestLink / Plan del leader (1) (1, 0, 0, 0)
pruebaLeaderSuite (1) (1, 0, 0, 0)
01-4:case01Leader
Test Results on Build build 0.1 test leader
Only Test Cases assigned to : rmacias
Test plan notes
Build description
Print
Show complete execution history
Import XML Results
Test Suite : pruebaLeaderSuite/
Test Case ID 01-4 :: Version : 1
case01Leader
Assigned to : rmacias
Last execution (any build) - Build : build 0.1 test leader
Not Run
Not Tested Yet
Summary
Probando los test
Preconditions
Estar conectado
Execution type : Manual
Step actions Expected Results
1 Pruebo el test True
Notes / Description
Result
Not Run
Passed
Failed
Blocked
Save execution
Save and move to next
Important Notice: Once a Result is updated from 'Not Run' to another value, you cannot set it back to 'Not Run'.
You can still set the Result to any other value.



Testopia

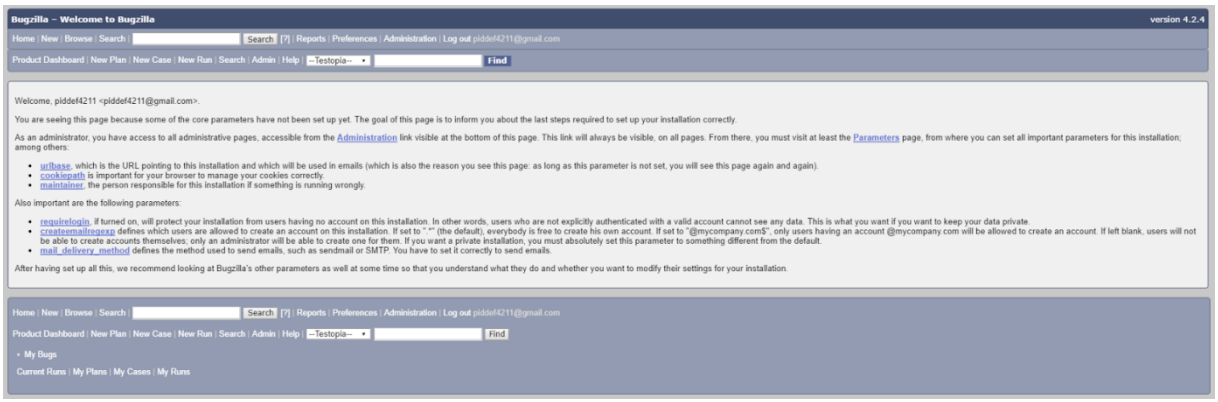
Testopia es un administrador de casos de pruebas, realizado por la fundación Mozilla para que funcione como una extensión de Bugzilla. Es una de las herramientas más utilizadas en la gestión de pruebas open-source, por lo que cuenta con un gran nivel de aporte por parte de sus usuarios.

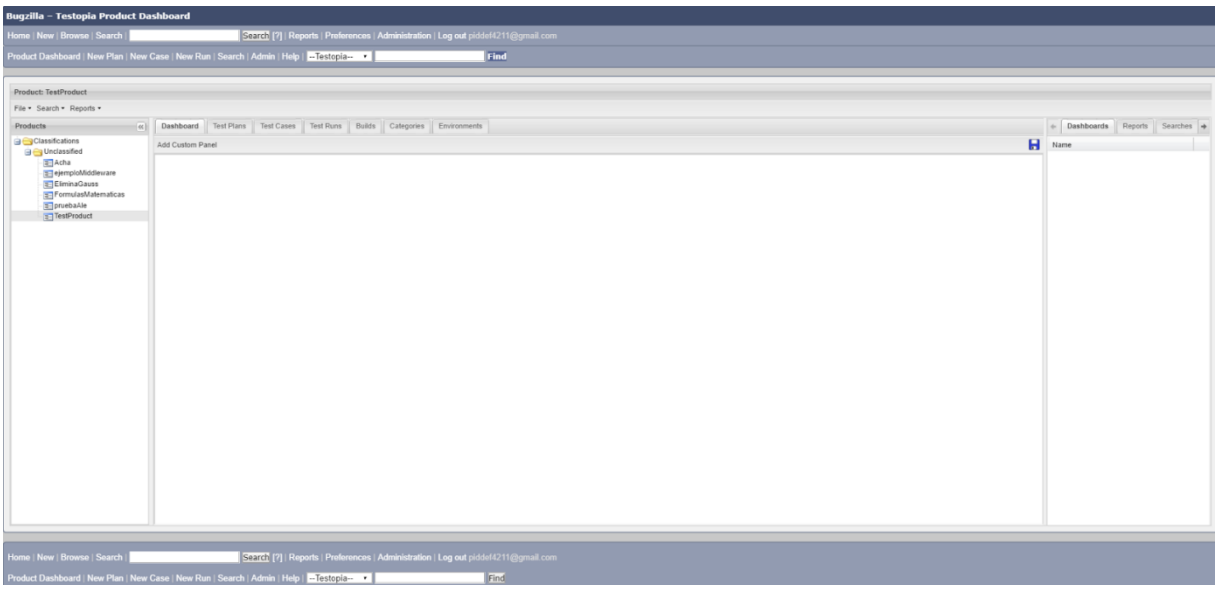
Presenta un sistema de fácil uso, mantenimiento y un amplio soporte.

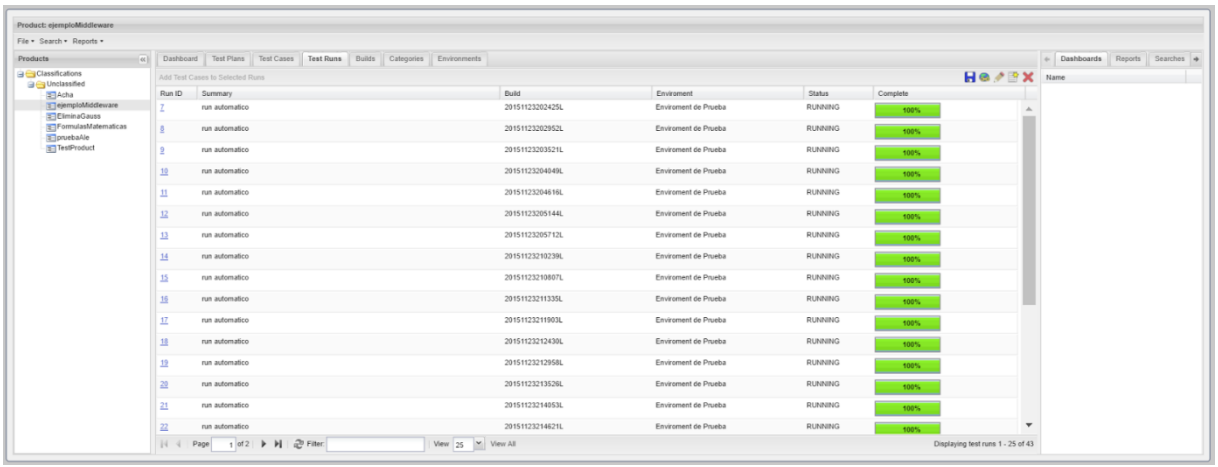
Ha sido desarrollado para tener compatibilidad con las bases de datos MySQL y Postgres.

Para acceder a más información se puede acceder al siguiente link: https://developer.mozilla.org/es/docs/Mozilla/Bugzilla/Testopia

Imágenes ilustrativas:







RTH - RTH Turbo

Requirements and Testing Hub es una herramienta de gestión de pruebas open source, que además incorpora la gestión de requisitos y la capacidad de seguimiento de errores.

RTH Turbo, surge originalmente de una optimización de la versión 1.2 de RTH.

Para un mayor conocimiento de las características de la herramienta se puede acceder a la siguiente dirección:

Donde además se puede acceder a una versión Demo.

Imágenes ilustrativas:

Radi

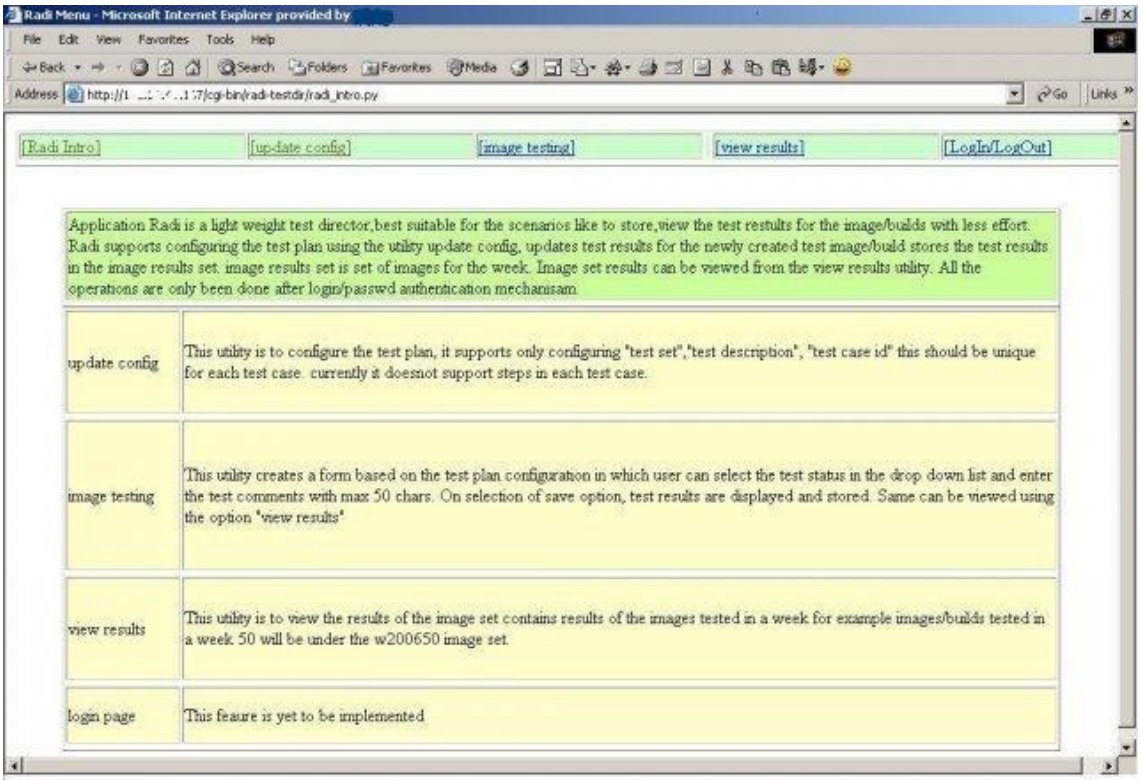
Radi - testdir

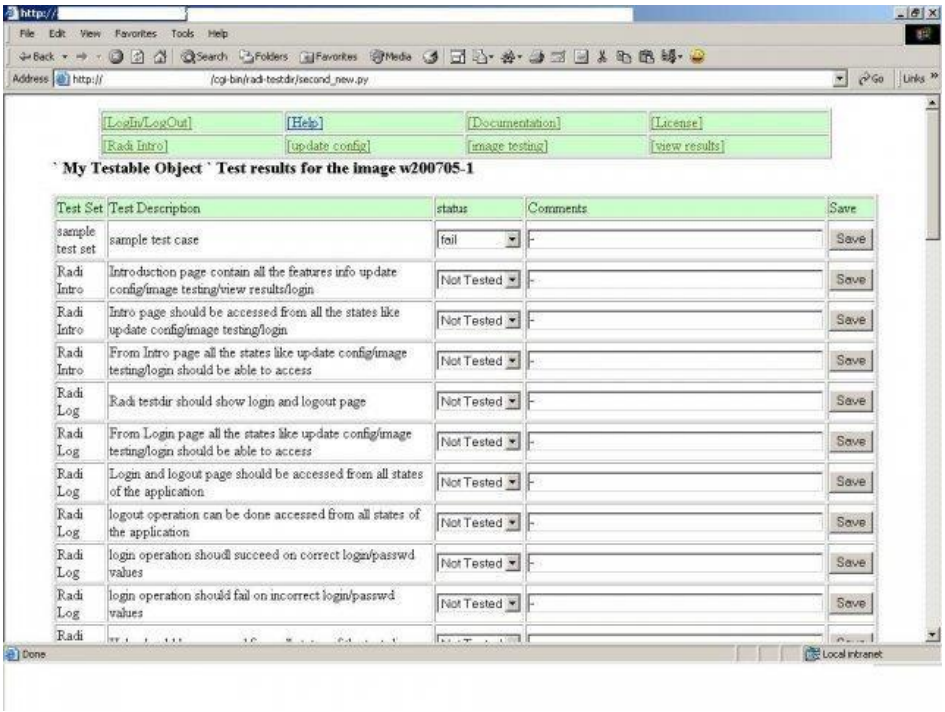
Radi Testdir es una herramienta de gestión de pruebas implementado en Python. Esta herramienta requiere de operaciones simples y mínimas por parte del usuario, a diferencia de otras herramientas.

Para una mayor información de la herramienta se sugiere visitar su página web:

http://radi-testdir.sourceforge.net/Radi_Home.html

Imágenes ilustrativas:


Radi Menu - Microsoft Internet Explorer provided by
File Edit View Favorites Tools Help
Address http://1 ..1 :7/cgi-bin/radi-testdir/radi_intro.py
[Radi Intro] [update config] [image testing] [view results] [LogIn/LogOut]
Application Radi is a light weight test director,best suitable for the scenarios like to store,view the test results for the image/builds with less effort. Radi supports configuring the test plan using the utility update config, updates test results for the newly created test image/build stores the test results in the image results set. image results set is set of images for the week. Image set results can be viewed from the view results utility. All the operations are only been done after login/passwd authentication mechanism.
update config
This utility is to configure the test plan, it supports only configuring "test set","test description", "test case id" this should be unique for each test case. currently it doesnot support steps in each test case.
image testing
This utility creates a form based on the test plan configuration in which user can select the test status in the drop down list and enter the test comments with max 50 chars. On selection of save option, test results are displayed and stored. Same can be viewed using the option "view results"
view results
This utility is to view the results of the image set contains results of the images tested in a week for example images/builds tested in a week 50 will be under the w200650 image set.
login page
This feaure is yet to be implemented


http://
File Edit View Favorites Tools Help
Address http:// /cgi-bin/radi-testdir/second_new.py
[LogIn/LogOut] [Help] [Documentation] [License]
[Radi Intro] [update config] [image testing] [view results]
` My Testable Object ` Test results for the image w200705-1
Test Set | Test Description | status | Comments | Save
sample test set | sample test case | fail | - | Save
Radi Intro | Introduction page contain all the features info update config/image testing/view results/login | Not Tested | - | Save
Radi Intro | Intro page should be accessed from all the states like update config/image testing/login | Not Tested | - | Save
Radi Intro | From Intro page all the states like update config/image testing/login should be able to access | Not Tested | - | Save
Radi Log | Radi testdir should show login and logout page | Not Tested | - | Save
Radi Log | From Login page all the states like update config/image testing/login should be able to access | Not Tested | - | Save
Radi Log | Login and logout page should be accessed from all states of the application | Not Tested | - | Save
Radi Log | logout operation can be done accessed from all states of the application | Not Tested | - | Save
Radi Log | login operation shoudl succeed on correct login/passwd values | Not Tested | - | Save
Radi Log | login operation should fail on incorrect login/passwd values | Not Tested | - | Save
Done
Local intranet

Salomé

Salomé es una herramienta open source para la gestión de pruebas que ofrece la posibilidad de trabajar con pruebas definidas según la norma ISO 9646.

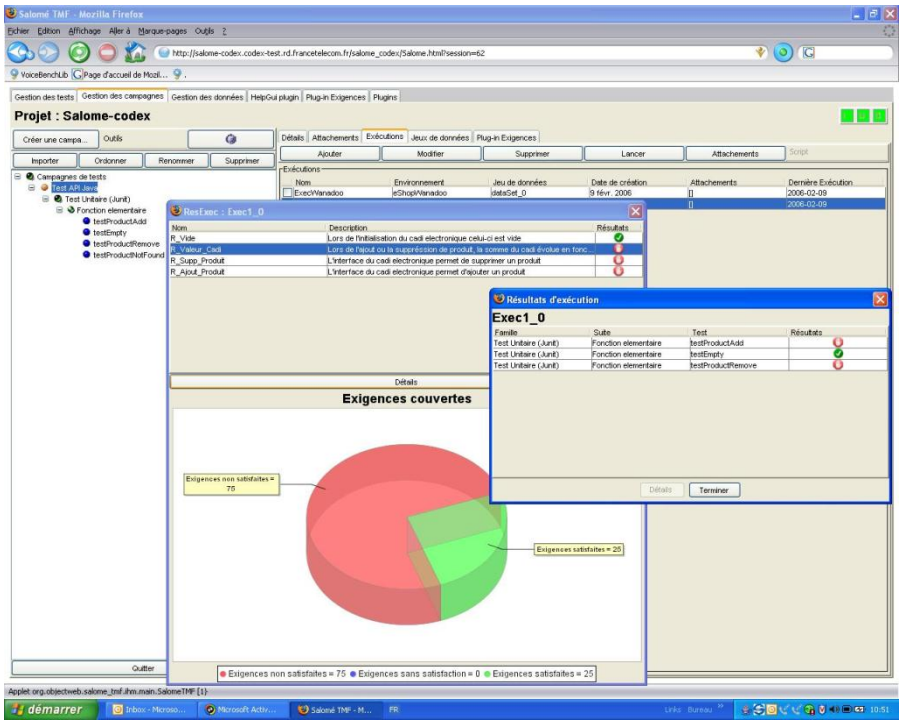
Además brinda la posibilidad de realizar tanto pruebas manuales como automáticas y organizarlas según conjunto de datos.

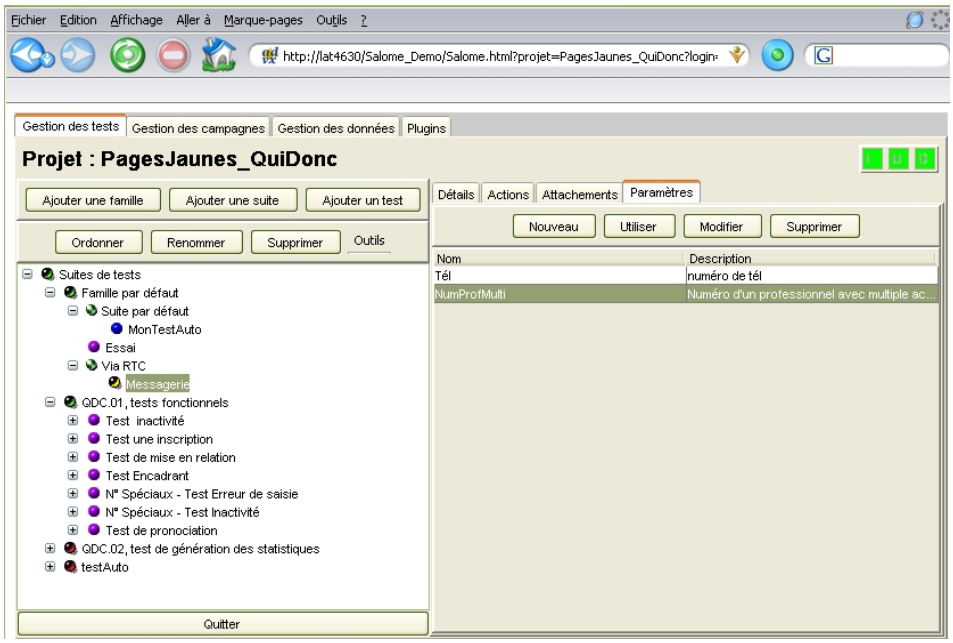
Se encuentra desarrollada completamente en Java.

Para mayor información se sugiere visitar la siguiente página web:

http://wiki.ow2.org/salome-tmf/

Imágenes ilustrativas:





Tarántula

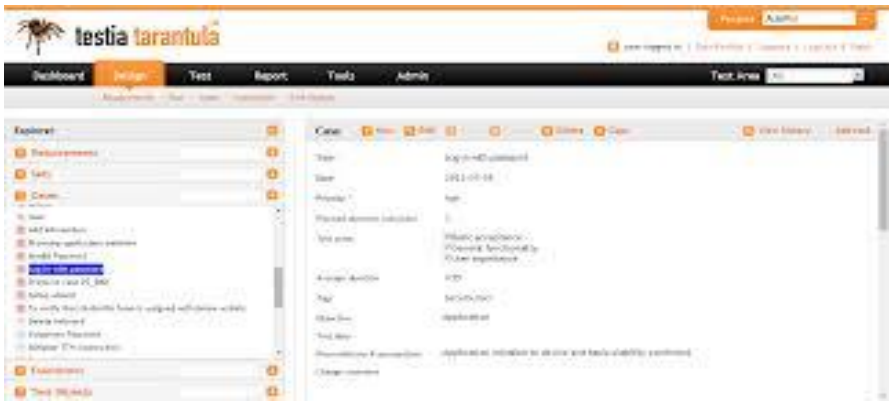
Tarántula es una de las herramientas más modernas para la gestión de pruebas destinada principalmente para entornos de desarrollo ágiles.

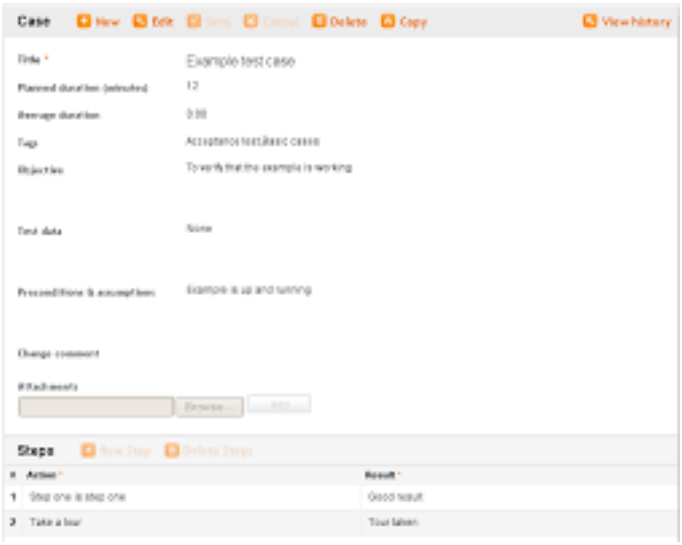
Tarántula es libre, licenciada como software de código abierto bajo la licencia GNU GPLv3. Sus principales objetivos y focos son: pruebas ágiles, gestión de las pruebas, informes y usabilidad.

Para un mayor conocimiento de la herramienta se recomienda visitar la siguiente página web:

http://www.testiatarantula.com/

Imágenes ilustrativas:





Criterios de elección

Luego de realizar la instalación de las distintas herramientas, se eligieron y establecieron de acuerdo a las necesidades de la arquitectura y su entorno de desarrollo un conjunto de criterios para su evaluación. A dichos criterios se le estableció un rango de valores y posteriormente se evaluó cada herramienta. Por último se obtuvo un criterio ponderado matemáticamente, que da como resultado la herramienta que posee los atributos que responden de mejor manera a la situación planteada.

Criterios

Usabilidad: Según lo establece la ISO 9241 “es el grado en el que un producto puede ser utilizado por usuarios específicos para conseguir objetivos específicos con efectividad, eficiencia y satisfacción en determinado contexto de uso”.

El rango de valores que puede adoptar: 0 a 10*

Y para esta valoración se tiene en cuenta los siguientes criterios, teniendo cada uno de ellos un peso de 0 a 2 puntos (siendo 0 la menor valoración y 2 la mayor) en la valoración final:

- Coherencia (con respecto a otros sistemas)
- Operatividad (Navegación, funcionalidad, control)
- Eficiencia (Velocidad de respuesta, frecuencia de ayuda)
- Identificación de tareas
- Potenciación de habilidades

Integración Bugzilla: Referencia a la posibilidad de integrar la herramienta con Bugzilla.

*El rango de valores que puede adoptar: 0 o 1 (Se integra o no)***

Carga de Requerimientos: Referenciado a la posibilidad de llevar a cabo una gestión y carga de requerimientos.

*El rango de valores que puede adoptar: 0 o 1 (Dispone o no)***

Soporte: Referenciado al asesoramiento existente tanto de la organización responsable de la herramienta, como también de la comunidad que hace uso de la misma.

*El rango de valores que puede adoptar: 0 a 10**

Y para esta valoración se tiene en cuenta los siguientes criterios, teniendo cada uno de ellos un peso de 0 a 2 puntos (siendo 0 la menor valoración y 2 la mayor) en la valoración final:

- Foro
- E-mails
- Chats
- Manual

- Comunidad activa de consulta

Actualizaciones: Referenciado a la periodicidad con la que se realizan actualizaciones de la herramienta.

El rango de valores que puede adoptar: 0 a 2

Correspondiente a los siguientes criterios:

- 0: Más de 3 años sin actualizaciones
- 1: entre 2 y 3 años sin actualizaciones
- 2: Menos de 2 años de la última actualización

Base de Datos: Motor de base de datos del que hace uso la herramienta, y su integración con los demás componentes de la arquitectura

*El rango de valores que puede adoptar: 0 a 2****

Manejo de usuarios: Capacidad de gestión de usuarios.

*El rango de valores que puede adoptar: 0 o 1**

Idioma soportado: Capacidad de ser una herramienta multilenguaje.

*El rango de valores que puede adoptar: 0 o 1**

Lenguaje Base: Lenguaje en el que se encuentra desarrollada la herramienta. Necesario saber por si se desea realizar alguna personalización, y no se cuenta con los recursos necesarios.

Entorno: Entorno en el que se accede a la herramienta. Puede ser vía Web, aplicación nativa, etc.

*Rango: (0 a 2: No satisfactorio; 3-4: Satisfactorio, 5-6: bueno-aceptable, 7-8: Muy bueno, 9-10: Excelente)

**Rango: (0: No posee), (1: Posee)

***Rango: (0: Insatisfactorio, 1: Bueno, 2: Muy Bueno)

Pesos ponderados

Los niveles de los pesos fueron ponderados de acuerdo a consideraciones de importancia relacionada a la convivencia de la herramienta en un entorno de desarrollo científico, a una integración con los distintos componentes de la arquitectura, su mantenimiento, soporte, etc.

Para ello se estableció un rango de que va desde 1 a 8, siendo 0 el valor de menos consideración y 8 el peso con mayor importancia. De esta manera quedan establecidos los siguientes pesos.

1-2	Criterio sin mayor importancia. Su presencia o ausencia no brinda valor
3-4	Criterio necesario pero no es de vital importancia.
5-6	Criterio a tener en cuenta, y de importancia a la hora de tomar decisiones
7-8	Criterio de vital importancia para tomar decisiones

Tabla de ponderación

	Pesos	**Testopia**	Ponderación	ST	**TestLink**	Ponderación	ST	**RTH**	Ponderación	ST
Usabilidad (0-10)	7		7	49		4	28		4	28
Integración Bugzilla (0-1)	8		1	8		1	8		0	0
Requerimiento (0-1)	5		0	0		1	5		1	5
Soporte (0-10)	8		8	64		7	56		5	40
Actualizaciones (0-10)	7		2	14		2	14		0	0
BD (0-2)	6	Mysql	2	12	Mysql	2	12	Mysql	2	12
Usuarios (0-1)	5		1	5		1	5		1	5
Multi-idioma (0-1)	3		1	3		1	3		1	3
Lenguaje		Perl			PHP			PHP		
Interfaz de usuario		Web			Web			Web		
TOTAL				**155**			131			93

	Pesos	**Radi**	Ponderación	ST	**Salomé**	Ponderación	ST	**Tarantula**	Ponderación	ST
Usabilidad (0-10)	7		4	28		5	35		8	56
Integración Bugzilla (0-1)	8		0	0		0	0		1	8
Requerimiento (0-1)	5		1	5		0	0		1	5
Soporte (0-10)	8		4	32		5	40		6	48
Actualizaciones (0-10)	7		1	7		1	7		2	14
BD (0-2)	6	xml	0	0	Mysql	2	12	Mysql	1	6
Usuarios (0-1)	5		1	5		1	5		1	5
MultIdioma (0-1)	3		1	3		1	3		1	3
Lenguaje		Python			JAVA			RUBY		
Interfaz de usuario		Web			Web			Web		
TOTAL				80			102			145

Como se puede observar, la herramienta con mayor puntaje en la tabla de ponderación es Testopia, que por consiguiente, será la elegida para integrar la Gestión de Pruebas a la Arquitectura de Desarrollo del Proyecto PIDDEF 4211.

Motivación

Testopia es la herramienta elegida para poder llevar a cabo la integración de la Gestión de Pruebas en la arquitectura de desarrollo de la Institución que nos permita la integración entre los componentes ya existentes en dicha arquitectura, una mayor documentación, organización, trazabilidad de las distintas pruebas que se lleven a cabo. Además se brindarán un conjunto de actividades, pasos y procesos que hacen a las buenas prácticas, las cuales no se encuentran presentes en entornos de desarrollo científico-técnico, ya que como se ha mencionado anteriormente no es un entorno en el que se cuente con conocimientos de conceptos específicos del desarrollo de software. Testopia junto con la implementación de procesos, técnicas establecidas y las buenas prácticas dan soporte a dos conceptos de suma importancia a la hora de hablar de aseguramiento de calidad, como lo son la validación y la verificación de software, y más aun teniendo en cuenta la criticidad de los desarrollos llevados a cabo por los equipos de I+D del Instituto Universitario Aeronáutico.

A continuación se comenzará con el desafío del proyecto de poder llevar a cabo la integración de la Gestión de Pruebas, empezando por la descripción de las distintas posibilidades que nos brinda Testopia, siguiendo luego con el desarrollo de un middleware para la integración de la misma con los distintos componentes de la arquitectura.

Testopia

Definición

La gestión de casos de pruebas (TC) es el proceso de seguimiento de los resultados de las pruebas en un conjunto de casos de pruebas (TC) para un determinado conjunto de entornos.

Testopia ha sido diseñado para para proporcionar una repositorio central que sea de utilidad para la comunidad de testers. Sirve tanto como repositorio de TC como un sistema de gestión. Su estructura encuadra para satisfacer las necesidades de la comunidad de Tester, independientemente del tamaño del grupo y de la organización a la que pertenezcan.

Aunque Testopia fue diseñado para la prueba de software, puede ser utilizado también para el seguimiento de cualquier tipo de TC.

Y, al ser, de código abierto, también permite ser modificado de acuerdo a las necesidades del entorno.

Testopia y Bugzilla

Bugzilla es una herramienta de Bug Tracking, seguimiento de errores, más usada y de código libre de los sistemas disponibles. Presenta un sistema de fácil uso, mantenimiento y con un amplio soporte.

Como los casos de pruebas son, y deben estar, muy ligados a los defectos, los TC deben ser escritos para verificar que el defecto ha sido solucionado en versiones futuras.

Y para esto se diseñó Testopia, que es un complemento para Bugzilla, y que permite en una misma experiencia de usuario lograr el seguimiento de defectos y la gestión de casos de pruebas.

Fundación Mozilla y el Proyecto Testopia

Para hacer un poco de historia, Testopia primeramente fue llamado *"Bugzilla TestRuner"*. Fue desarrollado por Maciej Maczynski mientas corría el año 2001, pero luego, la titularidad pasó a manos de Ed Fuentetaja, quien pasará los derechos de desarrollo a Greg Hendricks en 2006. Allí es llamado como más se lo conoce, Testopia, antes de que fuera liberada la versión 1.0 en mayo del mismo año. Actualmente reside como un Proyecto Mozilla. Su página web: http://mozilla.org/projects/testopia.

Requerimientos

	Versión
Bugzilla	3.x
MySQL	5.0
Modulos PERL	
JSON	1.14
Text::Diff	
Librerías	
Ext JS toolkit	2.0.1

¿En qué tipos de Testing nos puede ayudar Testopia?

Cómo se ha mencionado en el Marco Teórico del presente trabajo, existen dos clasificaciones generales para el Testing de software. Ellos son, Caja Negra (black box) y Caja Blanca (White box).

El testing de Caja Negra (Black Box testing), es una estrategia en la que solamente se basa en los requerimientos y las especificaciones, sin tener en cuenta los caminos internos. En cambio, en las pruebas de Caja Blanca, se tiene acceso al código fuente, algoritmos y las estructuras internas que se han utilizado.

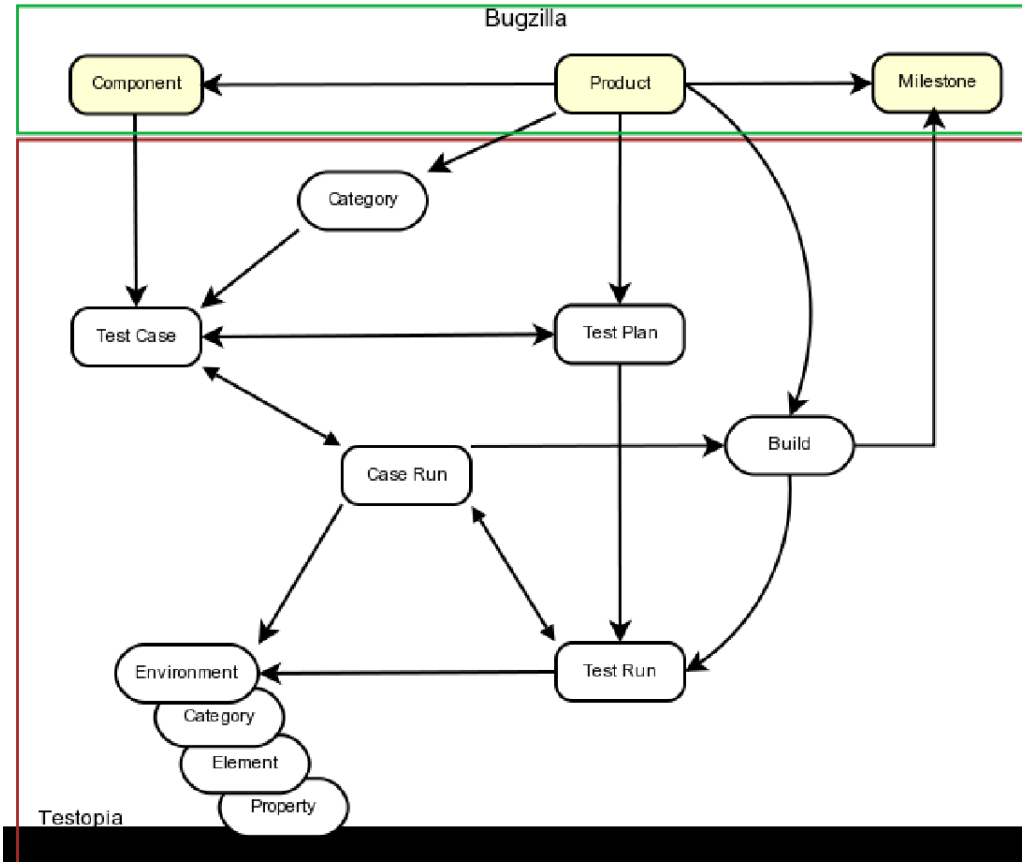
Pruebas de Caja Negra

Testopia ha sido diseñado para llevar a cabo, principalmente, este tipo de pruebas, en donde los requerimientos se traducen directamente en casos de prueba. Si un requerimiento no se cumple, el caso de prueba falla, en caso contrario, pasa con éxito.

Pruebas de Caja Blanca

Testopia no ha sido diseñado para manejar este tipo de pruebas, pero a la vez puede proporcionar un repositorio de resultados de las pruebas.

Arquitectura Bugzilla/Testopia



Test Plan (Planes de Pruebas)

En la parte superior de la jerarquía de Testopia encontramos a los Test Plan. Y antes de realizar cualquier acción, lo primero que se necesita es crear un Plan de Pruebas.

Los Test Plan están asociados con un producto único en Bugzilla, aunque se puede tener varios Test Plan por cada producto.

Nuestro Plan de Pruebas nos servirá como un punto de almacenamiento para todos los Test Case (casos de pruebas) relacionados y los Test Run (pruebas de funcionamiento), y actuará como dashboard para el testing. Además permitirá determinar quién tendrá acceso para actualizar los Casos de Prueba (Test Case).

Test Case

Consideraremos a los casos de prueba como el centro, el corazón de todo el testing. Ellos nos explican los pasos a seguir durante la ejecución de una prueba y qué resultados puede esperarse. En el caso de que algunos de los pasos no sea el esperado, la prueba fallará.

En Testopia, los Test Case son semi-independientes. Cada Caso de Prueba puede ser asociado a muchos Planes de Prueba, pero debe tenerse especial cuidado cuando se actualiza algún caso de prueba, para que este no interfiera con las pruebas de otro Plan de Pruebas.

Para cada Test Case se encontrará una lista de los Test Plan asociados a éste.

Test Run

Los Test Run son el centro del esfuerzo de los Test. Una vez que se hayan definido un conjunto de Casos de Prueba, se comienza a realizar los Test Run. Donde, cada ejecución se asocia a un único Test Plan, y puede consistir en cualquier número de casos de prueba de ese plan.

Test Run Environment

Si consideramos que los Casos de Pruebas son el "qué" de las pruebas, podremos considerar a los Entornos como el "dónde".

Ninguna de las pruebas se puede realizar en el vacío. Tener en cuenta esto es importante, porque el "donde" es de igual importancia que el "cómo" de la ejecución. Y esto se debe a que el Software está diseñado para desplegarse en algún hardware y bajo condiciones específicas. Las cuales serán capturadas en el Test Run Environment.

Builds

El desarrollo de software es generalmente un "asunto" iterativo. En donde los desarrolladores escriben el código, los compilan, y luego éstos forman parte de algún sistema.

Los errores y las demandas de nuevos requerimientos aparecen, y son los desarrolladores quienes tienen que reescribir o modificar el código para poder solucionar dichos pedidos.

En Testopia, a cada iteración se la denomina Builds. Las Builds, generalmente se las relaciona con los hitos de un proyecto, y para poder realizar un Test Run, mínimamente se tendrá que definir una Build.

Test Case Run

Estos representan el registro de cómo ha sido el resultado de un Test Case en una ejecución, en una determinada Build y en un entorno en particular.

Cuando se crea un Test Run, se crean registros por cada caso de prueba interviniente. Por defecto, estos asumen la Build y el Environment del Test Run, sin embargo esto puede ser modificado de acuerdo a las necesidades y a las condiciones específicas que se requieran para los Test Case.

Lineamiento del Proceso de Testing con Testopia

En esta sección se expondrán las formas en que Testopia maneja a forma general los distintos conceptos antes mencionados, como se crean y se modifican, y consideraciones a tener en cuenta a la hora de trabajar con ellos. Seguidamente se planteará los pasos necesarios para que Testopia quedé correctamente configurado para la interacción con los demás componentes de la arquitectura.

Dashboard

Es el punto de lanzamiento y comienzo para todas las acciones en Testopia. Se podrá visualizar los distintos planes por producto, ver y guardar informes y reportes, manipular Builds y categorías.

El dashboard nos presenta un conjunto de pestañas donde podremos encontrar la lista de planes, casos, ejecuciones, builds, entornos y categorías.

Agregar Categorías y Builds

Las categorías se utilizan para clasificar a los Test Case.

Cada producto tiene una categoría determinada (default category), y éstas se pueden utilizar para dividir los Test Case.

Como se ha mencionado anteriormente, antes de crear un Test Run, se debe especificar al menos una Build.

Para añadir una Build o una categoría, se deberá selección <<add>>.

Para editar las categorías y Builds, se deberá realizar doble click sobre el campo en la lista presentada, y desde allí, se podrán editar y eliminar individualmente las categorías y Builds.

Nuevo Test Plan

Todas las pruebas deben comenzar con un Test Plan, por eso a continuación se describen los pasos a seguir para crear un nuevo Test Plan en Testopia.

1. Hacer click en <<New Plan>>

2. Ingresar el nombre
3. Seleccionar un producto de la lista presentada
4. Seleccionar un tipo para este Plan
5. Seleccionar la versión de producto. Será la versión por defecto para las nuevas ejecuciones.
6. Escribir o pegar el documento del plan en el editor de documento del Plan.
7. Seleccionar <<add>>

Visualizar un Test Plan

En la parte superior del nuevo plan creado se podrá ver una sección de información general que contiene la información perteneciente al plan.

Además, se podrá ver el autor del plan, fecha de creación, versión del documento que se está visualizando.

Adjuntar Archivos

Durante la creación, o ya habiendo sido creado el Test Plan, puede añadirse archivos adjuntos. Para ellos será necesario hacer click en <<browse>>, localizar el archivo que se quiera adjuntar, escribir una descripción si se desea, y por último, adjuntar.

Editar campos del Plan

En la sección de información, se podrán ver los campos a los cuales se les está permitido ser modificados, identificados con la imagen de un lápiz.

Cualquier cambio que se realice en estos campos, serán guardados inmediatamente.

Historial

Nos permite tener un cierto control sobre las acciones que se van realizando, como son los distintos cambios que se realiza, quien los realiza, y cuándo. También se pueden realizar comparaciones entre distintas versiones para ver las modificaciones realizadas.

Crear Test Case

Una vez que tengamos listo nuestro Test Plan, podremos crear los Test Case que puedan ser almacenados en él.

Para ellos deberemos hacer click en <<New Case>> en la página de Planes. O sino, podrá elegirse el Plan, hacer click derecho, y seleccionar <<Add Test Case>>

Pasos:

1. Click en <<Create New Test Case>>
2. Realizar una breve descripción del Test Case en el campo de Resumen
3. Seleccionar una Categoría.
4. Añadir un tester por defecto, o seleccionar un componente para asiganar el Test Case al contacto Bugzilla QA para ese componente.
5. Listar los pasos para la prueba en el campo de <<Action>>
6. Proporcionar los resultados que se esperan en el campo de <<Results Field>>
7. Seleccionar <<Add>>

Nota: Al existir la posibilidad de que un solo Test Case pueda ser vinculado a múltiples Test Plan, se estará permitido seleccionar a qué Test Plan se desea hacer la vinculación.

Añadir y Remover Componentes y Etiquetas (Tags)

Como en los apartados anteriores, se podrá ver en detalle la información perteneciente al Test Case, y allí será posible añadir componentes adicionales a los Test Case. Así también se está permitido remover los componentes que se deseen. De la misma forma se podrá hacer con las Etiquetas relacionadas a los Test Case.

Visualizar Resultados de las ejecuciones

Seleccionar <<CaseRun History>>, y desplegará el historial del Test Case en todas las ejecuciones. Allí podremos observar el estado del Test Case, si éste ha pasado la prueba con éxito o no, en cada ejecución.

Adjuntar Archivos

Es posible adjuntar archivos a los Test Case, de la misma manera que fue descrito en el apartado de Test Plan.

Adjuntar Bugs

A diferencia de los Test Plan, en esta sección se podrá adjuntar Bugs a los Test Case.

A Test Case se le podrá adjuntar varios Bugs, y para ello será necesario ingresar el número de Bug en el campo correspondiente, y luego seleccionar <<Add>>. También se podrá ver una tabla con todos los Bugs adjuntos.

Editar Campos del Test Case

De la misma manera que en Test Plan, se podrán editar y actualizar los valores introducidos. Los cambios se podrán visualizar en el historial correspondiente.

Dependencias de los Test Case

Las dependencias es un nuevo concepto que se aplica en los Test Case. A menudo cuando se realizan pruebas de Test Case, el orden en que se realizan las pruebas depende de las pruebas que se encontraban antes en lista.

Otra posibilidad es que, si un determinado Test Case falla, impida que otros Test Case puedan ejecutarse con éxito.

Para esto es que se usan los campos de dependencias, que nos ayudan a establecer las relaciones entre los distintos Test Case.

En el caso de que algún Test Case bloquee la ejecución de otro, se deberá introducir el ID de ese otro Test Case en el campo especificado. Y, si ese Test Case requiere que se ejecute otro Test Case primero, se deberá introducir el ID de dicho Test Case en el campo de dependencia. Si un Test Case falla, y bloquea a otro Test Case, y ambos se encuentran en la misma ejecución, el Test Case bloqueado recibirá <<Blocked>> como estado.

Crear Entornos (Environments)

Como se ha descrito anteriormente, los entornos representan el “dónde” de las pruebas. Y en las pruebas de software, éste “donde” podría incluir por ejemplo, el sistema operativo, hardware, plataformas, etc…

El entorno más básico consiste en un sistema operativo y una plataforma elegida a partir de listas que presenta Bugzilla. No obstante, pueden formar parte del entorno conjuntos de aplicaciones y otros productos, navegadores u otros paquetes.

En Testopia la creación de Entornos requiere de dos pasos.

1. Definir un conjunto de variables que se utilizan en el entorno.

2. Crear el entorno desde el conjunto de elementos posibles.

Administración de Entornos (Environment Administration)

Al momento de instalar Testopia, se deberán definir un conjunto de variables de entorno que serán utilizadas para construir, luego, los entornos.

Las variables de entorno se encontrarán dispuesta en forma de una jerarquía de objetos, representadas como un árbol. Existen cuatro niveles principales: Categoría, Elementos, Propiedades, Valores de Propiedades.

Categorías

Las categorías nos proporcionan un mecanismo de clasificación para los elementos que componen el entorno.

Cada categoría es asociada a un solo producto. O también, se puede hacer uso de la etiqueta <<All>>, lo que denota todas las categorías de elementos que no son específicas de ningún producto.

Para crear una categoría, deberemos hacer click derecho sobre el producto o <<All>>, y seleccionar <<Add Category>>. A continuación, se selecciona la categoría nueva, recientemente creada, y en el formulario que se presenta se podrán editar el nombre de la categoría, como así también el producto al que se encuentra asociada.

Elementos

Los elementos son lo esencial de nuestro entorno. Para poder crearlos, primeramente tenemos que tener creada alguna categoría, o usar <<All>>. Seguidamente, se selecciona la categoría a la cual se le desee añadir elementos, se hace click derecho en dicha categoría y se selecciona <<create element>>. Esto crea un nuevo elemento, llamado <<nuevo elemento>>, el cual puede ser editado haciendo click derecho sobre el mismo y seleccionando <<edit>>.

Pueden existir elementos anidados, es decir, se pueden crear elementos dentro de otros elementos, de la misma manera en la que se ha creado un elemento simple. Es posible crear tantos niveles de elementos como sea necesario para la complejidad del entorno.

Propiedades

Las Propiedades son las que describen a los elementos. Se pueden añadir propiedades a los elementos haciendo click derecho sobre este y seleccionando <<Add Property>>. Se podrán añadir tantas propiedades como sea necesario. Las Propiedades también pueden ser anidadas.

Valor de las Propiedades

Definida una propiedad para un elemento, se deberá proporcionar una lista de valores para elegir en el entorno. Deberemos hacer click en la Propiedad, y seleccionar <<Add Value>>,y nos creará un valor para la propiedad. Para realizar la edición de la misma, se deberá seleccionar <<Edit>> al hacer click derecho.

Crear nuestro Entorno

Una vez que se han configurado los elementos que se van a utilizar en el entorno, se puede crear entornos propios con esos elementos.

Para ellos, seleccionaremos <<Add>> en la pestaña de Entorno del dashboard. Allí tendremos que ingresar el nombre del entorno a crear. El Producto solo es usado para clasificación. Esto no limitará las opciones de qué elementos se pueden colocar en el entorno.

Una vez creado el entorno, haremos click en él, para editarlo. En el panel de edición podremos ver dos árboles, uno que representa al nuevo entorno, y el otro que contiene la lista de las variables que podremos elegir para el mismo.

El entorno estará formado de los elementos que se han definido anteriormente.

Para agregar un elemento, deberemos buscarlo en la lista y arrastrarlo hasta el árbol perteneciente al entorno. Aquí no es de importancia el orden en el que se encuentren.

Una vez que hemos seleccionado los elementos del entorno, tendremos que seleccionar cuál de los valores de las propiedades se aplicarán al mismo. Para ello, se debe expandir el elemento y la propiedad, y luego hacer click en el valor a usar.

Para eliminar un elemento, bastará con hacer click derecho sobre él, y seleccionar <<Remove>>. Todos los cambios serán guardados inmediatamente.

Crear un Test Run

Pasos

1. Click en <<Create a New Test Run>> en el Test Plan correspondiente.
2. Seleccionar que Test Case se quiere incluir.
3. Ingresamos un resumen de la prueba a realizar.
4. Seleccionamos una Build de la lista existente, o escribimos el nombre para una nueva.

5. Seleccionamos un Entorno de la lista de Entornos.

6. Seleccionamos <<Add>>

Una vez que tengamos un entorno y Tests Cases se está listo para comenzar una prueba. Y esto se hace mediante los Test Runs. La forma más rápida y fácil de hacerlo es mediante el Test Plan creado, ya que solo basta con hacer click derecho sobre él y seleccionar <<Create a New Test Run>>.

Se mostrará una lista de Test Case <<Confirmados>> del Test Plan. Se podrán seleccionar sólo los que se deseen incluir o todos ellos a la vez, utilizando <<Select all>>. También será posible aplicar filtros para añadir solo los Test Case que cumplan con ciertos criterios. En el caso de que se posean gran cantidad de Test Case, se pueden utilizar funciones de localización en la tabla, por defecto sólo es posible visualizar 25 Test Case. *Esto reviste gran importancia, ya que los Test Case a incluir serán solo los que se encuentran visibles en la pantalla*. De todas formas, luego, será posible agregar más Test Case.

El siguiente paso será proporcionar un resumen, y seleccionar un Build.

En el caso de que no se haya creado una Build antes, sólo se tendrá que establecer el nombre de la nueva Build, y será añadido al producto.

Finalmente, es necesario seleccionar un Entorno. Para ello, escribiremos el nombre del Entorno, y a medida que se va escribiendo, aparecerán las coincidencias de los mismos pertenecientes a la lista de Entornos existentes. También se puede seleccionar la flecha desplegable que contiene los entornos del producto.

Información de los Test Run

Al igual que con los otros objetos, podremos observar una sección de información general por encima de la ejecución. Esta sección contiene información similar a la ya explicada en Test Case y Test Plan, con una notable excepción:

Barra de Progreso. Esta barra nos mostrará en porcentajes los Test Case validados, representados mediante colores de acuerdo a los estados.

Por debajo, será posible observar los Logs de los Test Run donde se encuentran los Test Case y allí es donde se puede tener acceso a la información de la ejecución.

Agregar Test Case

Se podrán agregar Test Case, haciendo click en <<Add Cases>> que se encuentra debajo la tabla de <<case-run>>. Esto nos llevará a la lista de Test Case que se nos presentó al momento de la creación del Test Run, salvo por la exclusión de aquellos Test Case que ya se encuentran en ejecución.

Editar campos del Test Run

Al igual que en los Test Case y los Test Plan, es posible actualizar cualquiera de los valores que se han aplicado al crear el Test Run. Tenemos que tener en cuenta cambiar de Build o Entorno no afectara a los casos que se encuentran en ejecución, pero se aplicará para los agregados después del cambio. Todos los cambios se reflejarán en el historial.

Ejecutar Test

Cuando se realiza la ejecución de Test Case puede que sólo quiera hacérselo para una prioridad específica, o para un encargado determinado. O bien, puede incluir un conjunto más amplio de Test Case que pertenezca a un ciclo.

También puede existir la necesidad de que para un determinado día sólo quieran ejecutarse Test Case de mayor prioridad, y dejar lo de más baja para un tiempo posterior. Todo esto se puede realizar utilizando Filtros.

Filtros de Test Case en la Ejecución

Se deberá expandir el Filtro, haciendo click en el triángulo de expansión, y allí se nos presentará una serie de opciones para el filtrado de Test Case, entre los que podemos encontrar:

- Estado
- Categoria
- Build
- Entorno
- Propiedad
- Componente
- Responsable
- Tags
- Resumen

Testopia puede recordar los filtros para que en próximas ejecuciones puedan volver a aplicarse.

Para guardar un filtro, deberemos introducir un nombre, y hacer click en <<Save>>.

Los filtros guardados se encontrarán presentes en la lista de filtros.

Para borrar un filtro, se deberá seleccionar <<Reset>>.

Clasificación de los Test Case

La lista de Test Case puede ser ordenada de acuerdo a los campos establecidos como encabezados. Para ellos se deberá hacer click en alguno de ellos para poder ordenarlos según dicho criterio.

En el caso de que se desee crear un orden personalizado, se deberá escribir el número de índice en el campo de índice y seleccionar <<Change>>.

Este orden reflejará los Test Case en forma ascendente según los índices suministrados. (Testopia solo permite orden ascendente en la actual versión).

Test Case: Éxito y Falla

Una vez que se encuentre todo preparado para comenzar la prueba, se expande el primer Test Case de la lista, y se lee <<Action>> y <<Expected Results>> para dicho caso.

A continuación se puede realizar a prueba. Si se logra el resultado esperado se toma como éxito la prueba y se selecciona el <<Green check>> o <<PASSED>>

Si el resultado no fue lo esperado o un ocurrió algún error, se marca al Test Case como fallido, y se selecciona <<red X>> o <<FAILED>>

Añadir Notas

La actualización de estado agregará una línea al campo de notas describiendo el momento del cambio y quién lo ha realizado.

Además podrán agregarse notas adicionales, seleccionando <<Add Note>> y luego <<Append Note>>.

Adjuntar Bugs

Si un Test Case ha fallado, o por cualquier otra razón, es posible que se desee adjuntar Bugs a la prueba. Testopia nos permite adjuntar bugs existentes o crear nuevos.

Para realizar lo primero, adjuntar Bugs existente, se deberá escribir el campo el número de Bug y seleccionar <<Attach Bug>>. El Bug se mostrará en el campo de Bugs encontrados.

En el caso de que se desee iniciar un nuevo Bug se seleccionará <<New>>, lo que nos llevará a la página de bugs, donde se encuentra información acerca del Test Case, allí se podrá introducir detalles adicionales al Bug.

Reasignación de Test

El campo de asignación se utiliza para ayudar a los Tester a seguir sus Test Case.

Si se desea realizar una reasignación de algún Test Case en partículas, se podrá hacer introduciendo el nombre del usuario Bugzilla en el campo de asignación y seleccionando <<Assign>>.

Cambio de Build o Entorno en un Test Case

Debido a que puede existir la necesidad de realizar cambios en la Buid o en los Entornos, la flexibilidad de Testopia nos permite actualizarlos. Cada vez que se realice se generaran se generaran notas estableciendo el momento del cambio como así también quien lo ha realizado.

Eliminar Case-Runs

A veces, por error, puede añadirse un Test Case a una ejecución, o puede elegirse una combinación de Build y Entorno no válidas. En estos casos, es más fácil eliminar los case-run juntos. Para ello será necesario disponer de los permisos adecuados. Para eliminar un solo Test Case, se deberá seleccionarlo de la lista y elegir <<Delete>>.

Actualizar varias Cases a la vez

Es posible actualizar un grupo de case-runs a la vez seleccionando el mismo en la lista. También se puede usar la barra de herramientas superior, sobre la lista de los Cases. Desde allí se podrá cambiar el estado, adjuntar bugs, actualizar atributos, e incluso eliminar la lista completa.

Finalizar

Una vez finalizadas todas las pruebas en una ejecución, se debe establecer el estado de ejecución <<STOPPED>>, que evitará que se produzcan cambios en los case-runs.

Aspectos generales

Identificadores

Objeto	*Prefijo*
TEST CASE	case, TC, c
TEST PLAN	Plan, TP, P
TEST RUN	Run, TR, r
TEST RUN ENVIRONMENT	Env, TE, e
TEST CASE-RUN	Caserun, TCR, cr
TAGS	tag

Importar y exportar

Los Test Case pueden ser exportados en formato XML o en Comma Separated Value.

Para exportar un Test Case, se deberá hacer click en la sección de exportación que se encuentra en parte inferior de la página del Test Plan, Test Case.

También es posible la exportación en formato CVS que permite la apertura de los Test Case en algún procesador de cálculo donde se pueden manipular los valores y generar informes personalizados.

La importación de Test Case, también es posible en Testopia.

Éstas pueden ser de instalaciones de otros Testopia o de otros sistemas de gestión de Test Case, utilizando la secuencia de comandos <<tr_importxml.pl>>.

Con el fin de ser elegibles para la importación, los Test Case exportados deberán cumplir con la definición de tipo de documento Testopia, que se encuentra en el archivo <<testopia.dtd>>. Esto se puede hacer mediante el uso de hojas de estilo XML, así se transforma el XML para que coincida con el DTD.

Marco Práctico

1. Diseño de la aplicación Middleware

La aplicación desarrollada deberá cumplir la función de interconectar los resultados de las distintas ejecuciones de Cruise Control con el correspondiente test case configurado en la herramienta seleccionada.

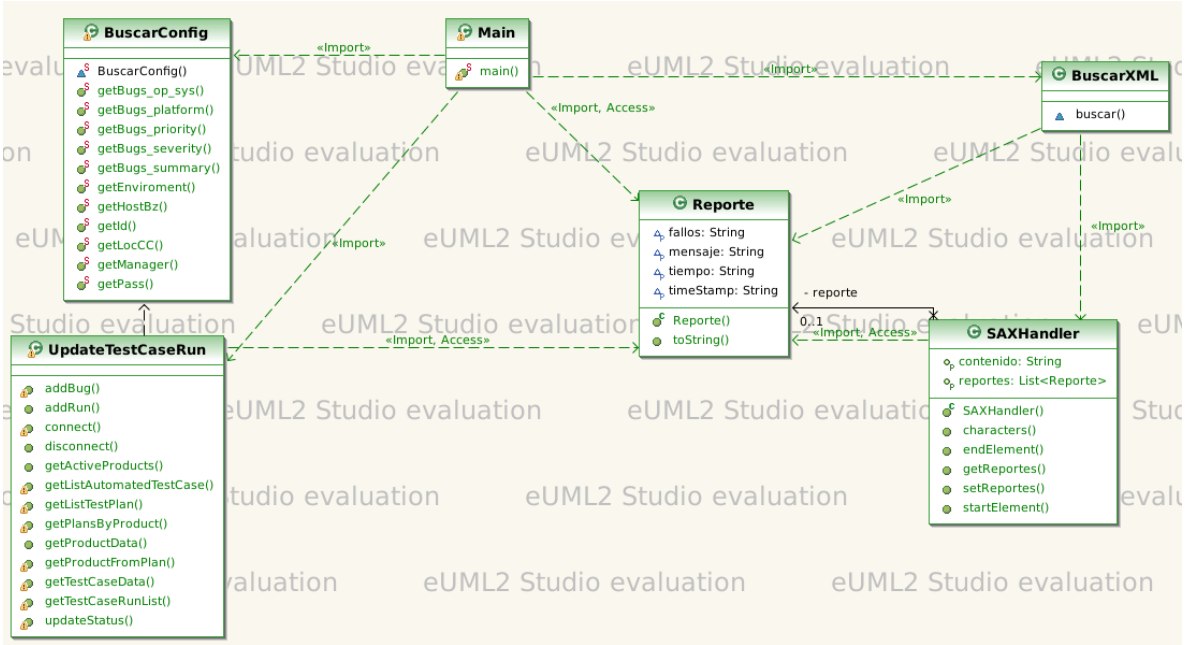
Entre las cosas que se actualizarán se encuentran:

- El resultado de la ejecución del test case
- Nuevo bug, en caso que la ejecución no haya sido correcta
- Desactivar la automatización en caso de que la ejecución no haya sido correcta, con el fin de evitar un exceso de bugs repetidos

La aplicación será desarrollada en el lenguaje de programación Java, debido a los conocimientos de los integrantes del grupo y a la practicidad del lenguaje para comunicarse con las herramientas elegidas.

Además, se hará uso de **API XML-RPC,** proporcionada por Bugzilla/Testopia, para actualizar los datos de los casos de uso dentro de las correspondientes aplicaciones. Esta tecnología será brevemente explicada en los anexos.

La estructura de la aplicación es la siguiente:



A continuación se procede a una explicación brece de las seis clases que se pueden observar en la imagen. (En Anexo se encuentran los Javadoc correspondientes, que contienen información con más detalle)

- **BuscarConfig**: Esta clase tiene la función de buscar las configuraciones de la aplicación, ya sean lugares donde se encuentran los servicios, datos a colocar en los bugs o test runs, enviroment de prueba, etc...

- **BuscarXML**: Esta función se encargara de buscar los xml resultantes de las pruebas de Cruise Control

- **Reporte**: Esta clase contendrá la información obtenida de los xml resultantes de las ejecuciones de las pruebas de Cruise Control

- **SAXHandler**: Clase que se encarga de parsear los xml

- **UpdateTestCaseRun:** Es la clase principal, contiene las llamadas a la api de Bugzilla/Testopia mediante las cuales se realizan las modificaciones a cada una

- **Main**: Clase inicial, contiene la integración del resto de las clases, se encarga de generar los path.

2. Instalación de Testopia

En la realización de este trabajo usaremos la versión 2.5 de Testopia que corresponde a la versión 4.2 de Bugzilla.

Para obtenerlo usaremos el comando "*wget"* a la dirección:

http://ftp.mozilla.org/pub/mozilla.org/webtools/testopia/

```
piddef4211@piddef4211:~/testopia$ wget ftp://ftp.mozilla.org/pub/mozilla.org/web
tools/testopia/testopia-2.5-BUGZILLA-4.2.tar.gz
```

Esto descargara un archivo comprimido que extraemos usando el comando *"tar –xf"* testopia-2.5-BUGZILLA-4.2.tar.gz.

```
tar -xf testopia-2.5-BUGZILLA-4.2.tar.gz
```

En este momento se debe copiar el contenido que se acaba de extraer al directorio de Bugzilla usando el comando *"cp"* con la opción –r para que copie también los directorios.

```
piddef4211@piddef4211:~/testopia$ sudo cp -r ./* /var/www/bugzilla
```

Ahora se mueve a la carpeta raíz de Bugzilla donde se copiaron los archivos en el paso anterior

```
piddef4211@piddef4211:/var/www/bugzilla$ cd /var/www/bugzilla
```

Una vez ubicados en este directorio, se debe ejecutar el archivo *"checksetup.pl"*, que instalará Testopia o, en caso de ser necesario, informará las dependencias que se necesiten para poder realizar esta función.

```
piddef4211@piddef4211:/var/www/bugzilla$ sudo ./checksetup.pl
```

En la siguiente imagen se ven los módulos que faltan en una primera ejecución del archivo.

```
COMMANDS TO INSTALL OPTIONAL MODULES:

            GD: /usr/bin/perl install-module.pl GD
         Chart: /usr/bin/perl install-module.pl Chart::Lines
   Template-GD: /usr/bin/perl install-module.pl Template::Plugin::GD::Image
    GDTextUtil: /usr/bin/perl install-module.pl GD::Text
       GDGraph: /usr/bin/perl install-module.pl GD::Graph
      mod_perl: /usr/bin/perl install-module.pl mod_perl2
Apache-SizeLimit: /usr/bin/perl install-module.pl Apache2::SizeLimit
      Text-CSV: /usr/bin/perl install-module.pl Text::CSV
XML Schema Validator: /usr/bin/perl install-module.pl XML::Validator::Schema

COMMANDS TO INSTALL REQUIRED MODULES (You *must* run all these commands
and then re-run checksetup.pl):

    /usr/bin/perl install-module.pl Text::Diff
    /usr/bin/perl install-module.pl GD::Graph3d

To attempt an automatic install of every required and optional module
with one command, do:

  /usr/bin/perl install-module.pl --all

*** Installation aborted. Read the messages above. ***
```

Para realizar la instalación de los módulos faltantes, ejecutamos el script *"install-modules.pl"* con la opción "–all". Esto tratara de instalar todos los módulos que Bugzilla requiera.

```
piddef4211@piddef4211:/var/www/bugzilla$ sudo /usr/bin/perl install-module.pl --
all
```

Una vez ejecutado install-modules.pl se ejecuta nuevamente para tratar de instalar Testopia. En este momento debería funcionar y mostrar lo siguiente.

```
Now that you have installed Bugzilla, you should visit the 'Parameters'
page (linked in the footer of the Administrator account) to ensure it
is set up as you wish - this includes setting the 'urlbase' option to
the correct URL.
checksetup.pl complete.
```

Una vez realizado estos pasos, Testopia se encontrará correctamente instalado.

Cambio en TestRun.pm

Si bien con los pasos descriptos anteriormente Testopia estará listo para ser usado, es necesario realizar una modificación en uno de sus scripts. Esto es para permitir una buena comunicación con la aplicación middleware que se desarrolló en este trabajo.

El script descripto es ***TestRun.pm***, que se encuentra en:

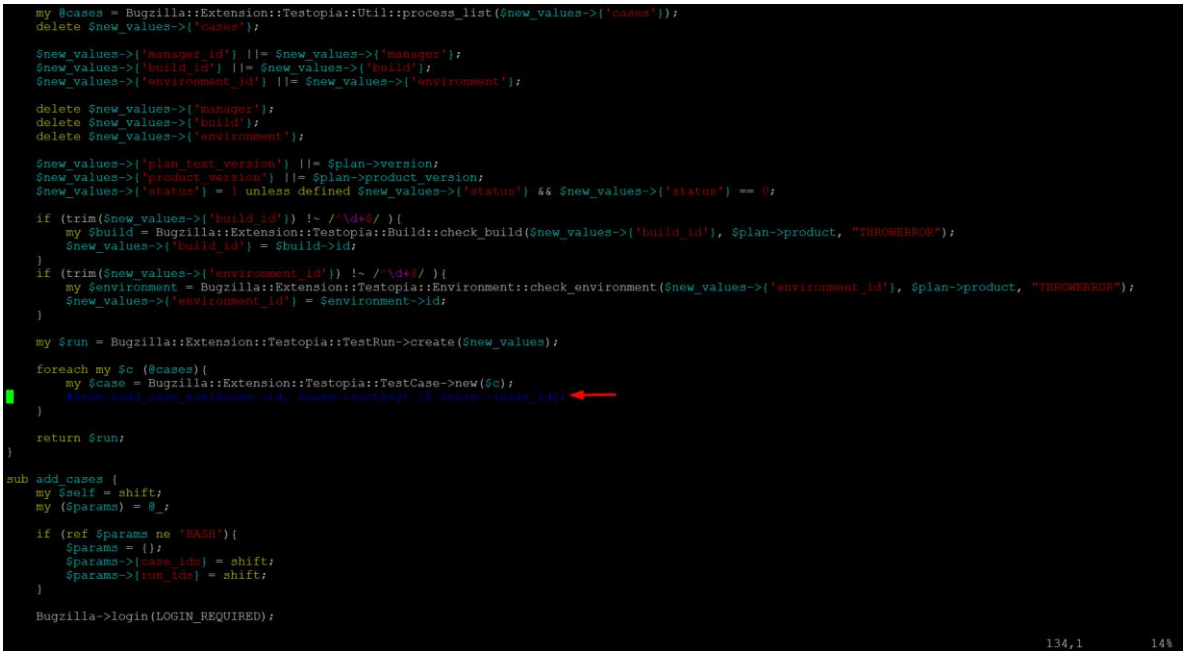
"/DIRECTORIO_ DE_ INSTALACION_ DE_BUGZILLA/extensions/Testopia/lib/WebService"

En el contexto de este trabajo el archivo editado será:

" /var/www/bugzilla/extensions/lib/WebService/TestRun.pm"

El objetivo de este archivo es proveer métodos para manipular los TestRuns de Testopia.

El cambio que se hará, será simplemente comentar la ***línea 134***, de forma que quede como en la siguiente imagen:



Este cambio se realiza con el fin de que Testopia no genere un nuevo Test Case Run cada vez que se genera un nuevo Test run. Esto ocasiona problemas de sincronización, ya que el programa middleware

genera un nuevo Test Run para cada test case que se encuentre configurado en Testopia. Los nuevos Test Runs generan nuevos Test Case Runs, que deben ser utilizados para actualizar el resultado de la ejecución de la prueba.

La solución fue quitar esta parte de la automatización de Testopia, y cuando es necesario, crear un nuevo Test Case Run con los parámetros correspondientes dentro del programa middleware.

3. Uso Cruise Control / Bugzilla / Testopia

La presente sección se divide en tres partes, que son los tres componentes que interactúan entre ellos. Tendremos la configuración de Cruise Control, Bugzilla/Testopia, y por último la visualización de resultados.

3.1 Cruise Control

Para comenzar un nuevo proyecto en Cruise Control, lo primero que se debe hace es **crear una nueva carpeta en el directorio projects**.

En el caso del Proyecto PIDDEF4211, se encuentra en /opt/cruisecontrol. Para crear el directorio, se usa el comando mkdir "nombre del proyecto".

Para todos los comandos ejecutados en este proyecto serán necesarios permisos de súper usuario, que, para obtenerlos, se realizará uso del comando "sudo" seguido del "comando a ejecutar", como se puede observar en la siguiente imagen.

```
piddef4211@integracion-testopia:/opt/cruisecontrol/projects$ sudo mkdir ejemploMiddleware
[sudo] password for piddef4211:
piddef4211@integracion-testopia:/opt/cruisecontrol/projects$
```

Dentro de la carpeta recientemente creada, se crearan tres nuevos directorios usando el mismo comando "mkdir".

- lib: que contendrá las dependencias usadas en el proyecto (en este caso solo junit)
- src: que contendrá los ficheros fuentes
- Test: en el cual estarán ubicados los códigos fuente que prueban el código ubicado en el directorio src.

```
piddef4211@integracion-testopia:/opt/cruisecontrol/projects/ejemploMiddleware$ sudo mkdir src Test lib
piddef4211@integracion-testopia:/opt/cruisecontrol/projects/ejemploMiddleware$ ls
lib  src  Test
piddef4211@integracion-testopia:/opt/cruisecontrol/projects/ejemploMiddleware$
```

En el mismo lugar donde se crearon estas carpetas se genera un nuevo archivo, llamado “build.xml”. Este contendrá las instrucciones para construir el proyecto. En este caso usamos el comando vim “nombre de archivo” para crear y editar el archivo.

```
piddef4211@integracion-testopia:/opt/cruisecontrol/projects/ejemploMiddleware$ sudo vim build.xml
```

A continuación se muestra una imagen del contenido que deberá tener el fichero build.xml de un proyecto llamado “ejemploMiddleware”.

```
<project name="ejemploMiddleware" default="all">
        <target name="all" depends="clean, compile, sleep, test, jar"/>
        <target name="clean">
                <delete dir="target" quiet="true"/>
        </target>
        <target name="compile">
                <mkdir dir="target/classes"/>
                <javac srcdir="src" destdir="target/classes"/>
        </target>
        <target name="sleep">
                <echo message="Sleeping for a while so oyu can se the build in the new dashboard" />
                <sleep seconds="15" />
        </target>
        <target name="test" depends="compile">
                <mkdir dir="target/test-classes"/>
                <javac srcdir="Test" destdir="target/test-classes">
                        <classpath>
                                <pathelement location="target/classes"/>
                                <pathelement location="lib/junit.jar"/>
                        </classpath>
                </javac>

                <mkdir dir="target/test-results"/>
                <junit haltonfailure="no" printsummary="on">
                        <classpath>
                                <pathelement location="target/classes"/>
                                <pathelement location="lib/junit.jar"/>
                                <pathelement location="target/test-classes"/>
                        </classpath>
                        <formatter type="brief" usefile="true" />
                        <formatter type="xml" />
                        <batchtest todir="target/test-results">
                                <fileset dir="target/test-classes" includes="**/*.class"/>
                        </batchtest>
                </junit>
        </target>
        <target name="jar" depends="compile">
                <jar jarfile="target/ejemploMiddleware.jar" basedir="target/classes"/>
        </target>
</project>
```

Una vez terminado de editar el archivo, se guarda con el comando “:x!”.

A continuación, se crean los archivos fuentes dentro de la carpeta src. Para navegar entre las carpetas usar el comando “cd” seguido de “nombre de la carpeta a ingresar”.

```
piddef4211@integracion-testopia:/opt/cruisecontrol/projects/ejemploMiddleware$ cd src/
```

En este directorio se crea una nueva carpeta, en este caso se llamará Matematica. Esta carpeta será el package que agrupará las clases del proyecto.

```
piddef4211@integracion-testopia:/opt/cruisecontrol/projects/ejemploMiddleware/src$ mkdir Matematica
```

Se ingresa a la carpeta recién creada (usando el comando cd) y se crea un nuevo archivo (usando el comando vim) con el nombre de la clase que se quiere crear, en este caso Suma.

```
piddef4211@integracion-testopia:/opt/cruisecontrol/projects/ejemploMiddleware/src/Matematica$ vim Suma.java
```

La siguiente es una imagen de una clase que simplemente suma dos números y devuelve su resultado. Cuando se termina de editar se guarda usando el comando: **:x!**

```
package Matematica;

public class Suma {
    private double a;
    private double b;

    public Suma(double a, double b) {
        this.a = a;
        this.b = b;
    }

    public double suma() {
        return a + b;
    }
}
```

Una vez completado este paso se navega a la carpeta Test creada anteriormente (usando el comando cd) para crear las clases de prueba.

```
piddef4211@integracion-testopia:/opt/cruisecontrol/projects/ejemploMiddleware$ cd Test/
```

Se crea una carpeta que será el package de las clases de prueba.

```
piddef4211@integracion-testopia:/opt/cruisecontrol/projects/ejemploMiddleware/Test$ mkdir Matematica
```

A continuación se debe entrar en la nueva carpeta y crear la nueva clase de prueba (usando el comando vim).

```
piddef4211@integracion-testopia:/opt/cruisecontrol/projects/ejemploMiddleware/Test/Matematica$ vim SumaTest.java
```

La siguiente imagen muestra una clase de prueba simple que prueba la clase suma que se creó anteriormente.

Para ello, se le pasa dos parámetros (5, 5), y controla que el resultado de la misma sea 10.

```java
package Matematica;

import static org.junit.Assert.*;

import org.junit.Test;

import Matematica.Suma;

public class SumaPrueba {

        @Test
        public void test() {
                Suma suma = new Suma(5.0, 5.0);
        double resultado = suma.suma();
        assertTrue(resultado == 10.0);
        }

}
```

Luego de esto, se debe navegar hasta la carpeta raíz del proyecto (ubicada en el directorio projects) y crear una nueva carpeta llamada “target” (usando el comando mkdir)

```
piddef4211@integracion-testopia:/opt/cruisecontrol/projects/ejemploMiddleware$ sudo mkdir target
piddef4211@integracion-testopia:/opt/cruisecontrol/projects/ejemploMiddleware$ cd target
```

A la careta creada, se le brindan permisos de escritura, lectura y ejecución. Esto es necesario debido a que en esta carpeta estarán ubicados los .class, .jar y los resultados de las ejecuciones. Para llevar a cabo esta acción se usa el comando “chmod” seguido de “permisos” “directorio o archivo a modificar permisos”. Adicionalmente se usa la opción –R, para hacer que el cambio de permisos sea recursivo dentro de este directorio.

```
piddef4211@integracion-testopia:/opt/cruisecontrol/projects/ejemploMiddleware$ sudo chmod -R 777 target/
```

Posteriormente, se edita el archivo config.xml, ubicado en el directorio principal de Crusie Control (usando el comando vim)

```
piddef4211@integracion-testopia:/opt/cruisecontrol$ sudo vim config.xml
```

En este archivo agregamos un segmento de código como el siguiente para agregar el proyecto recién creado a la ejecución de Cruise Control.

De este código hay que destacar la línea:

<execute command=”java –jar /home/piddef4211/middleware/middleware.jar ${project.name}”></execute>

Esta línea ejecuta el programa middleware que está ubicado en: ***“/home/piddef4211/middleware”*** , pasándole como parámetro el nombre del proyecto.

Este programa buscará en Bugzilla/Testopia un proyecto de prueba con el nombre del parámetro pasado.

Además, para los test cases automatizados tratará de encontrar los resultados de las pruebas, y en caso de encontrarlos actualizará el testrun asociado al test case.

```
<project name="ejemploMiddleware" buildafterfailed="false" requireModification="false">^M
  <!-- Declarar todos los plugins que se van a usar -->^M
  <!--<plugin name="git" classname="net.sourceforge.cruisecontrol.sourcecontrols.Git" />^M
  <plugin name="gitbootstrapper" classname="net.sourceforge.cruisecontrol.bootstrappers.GitBootstrapper" />^M-->
^M
  <bootstrappers>^M
<!--      <gitbootstrapper localworkingcopy="projects/${project.name}/" />^M-->
    <antbootstrapper anthome="apache-ant-1.7.0" buildfile="projects/${project.name}/build.xml" target="clean" />^M
  </bootstrappers>^M
  <listeners>^M
    <currentbuildstatuslistener file="logs/${project.name}/status.txt" />^M
  </listeners>^M
  <log dir="logs/${project.name}">^M
    <merge dir="projects/${project.name}/target/test-results" />^M
  </log>^M
  <schedule interval="300">^M
    <ant anthome="apache-ant-1.7.0" buildfile="projects/${project.name}/build.xml" />^M
  </schedule>^M
 <publishers>^M
    <onsuccess>^M
       <artifactspublisher dest="artifacts/${project.name}" file="projects/${project.name}/target/${project.name}.jar"/>^M
      <execute command="java -jar /home/piddef4211/middleware/middleware.jar ${project.name}"></execute>
    </onsuccess>^M
       <onfailure>^M
          <execute command="/usr/local/ant-parser/script/import.sh /opt/cruisecontrol-bin-2.8.4/logs/${project.name}"/>^M
      </onfailure>^M
  </publishers>^M
^M
 </project>^M
```

Con lo hecho hasta este punto el proyecto quedará creado en Cruise Control. El siguiente paso es configurar Bugzilla/Testopia para que el programa middleware pueda realizar la sincronización entre componentes.

Resumen de pasos para la utilización de Cruise Control

1. Crear una nueva carpeta en el directorio “projects” como sudo (Ej: */opt/cruisecontrol/ejemploMiddleware*)

2. Dentro del directorio creado, se crean 3 directorios nuevos:
   a. lib: contenedor de dependencias*(ejemploMiddleware/lib*)
   b. src: contenedor de ficheros fuentes (*ejemploMiddleware/src*)
   c. Test: estarán ubicados los códigos fuentes que prueban el código ubicado en src. (*ejemploMiddleware/Test*)

3. Se crea un fichero llamado “build.xml” dentro del directorio padre del proyecto. *(“ejemploMiddleware”)*. Este archivo contiene las instrucciones para construir el proyecto.

4. Se edita y guarda el fichero “build.xml” según ejemplo en imagen.

5. Crear ficheros fuentes en el directorio “src” (*ejemploMiddleware/src/Matematica*). Este directorio es el “package que agrupará las clases del proyecto”

6. Dentro del directorio recientemente creado, se crea el fichero de la clase (*Suma.java*)

7. Se edita el fichero creado, para que contenga funcionalidad de clase.

8. Crear directorio que contendrá las clases de pruebas dentro del directorio Test (*ejemploMiddleware/Test/Matematica*)

9. Dentro del directorio creado, se crea la clase de prueba (SumaTest.java)

10. Se edita el fichero creado, para probar la clase Suma.

11. En el directorio raíz del proyecto *(/opt/cruisecontrol/ejemploMiddleware/)*, se crea un nuevo directorio llamado "target" (contendrá los .class .jar y resultados de las ejecuciones). A este directorio se le darán permisos de escritura, lectura y ejecución.

12. Editar el archivo "config.xml", que se encuentra en el directorio principal de Cruise Control, para agregar el proyecto recién creado a la ejecución de CruiseControl.

3.2 Bugzilla/Testopia

Para realizar las configuraciones necesarias en Bugzilla/Testopia se deberá ingresar a la url en donde el mismo este publicado. En el caso del proyecto actual la misma es: http://accesodi.iua.edu.ar/testopia/bugzilla/.

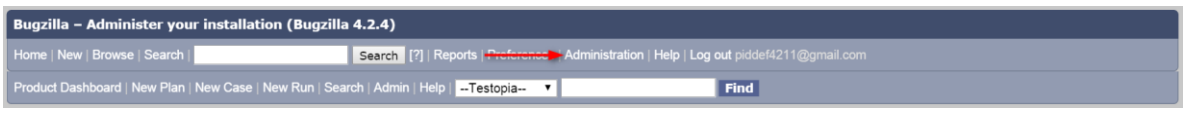
Una vez accedido a la web, se realiza el login usando las credenciales

- usuario: piddef4211@gmail.com

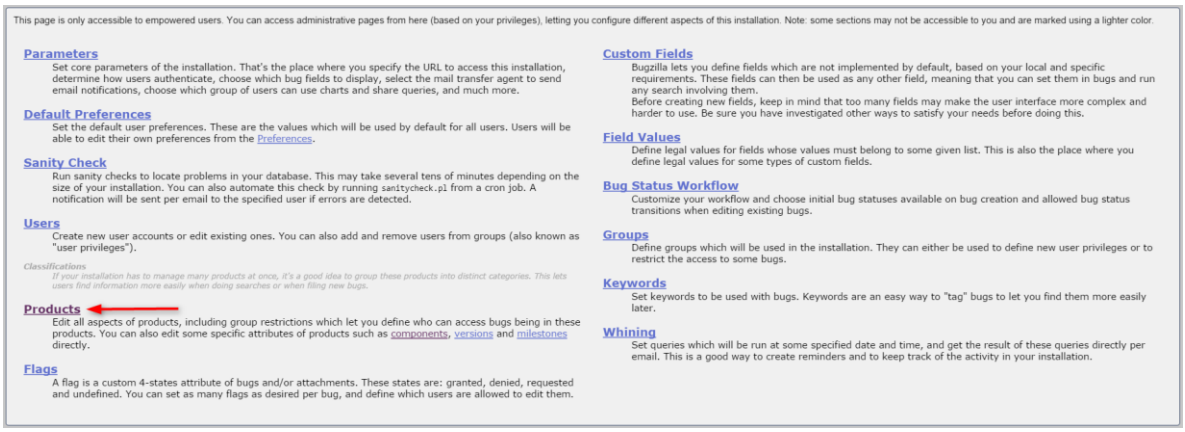
- contraseña: piddef4211.

Una vez que el ingreso sea exitoso, se deberá crear un nuevo producto que será asociado al proyecto creado en Cruise Control.

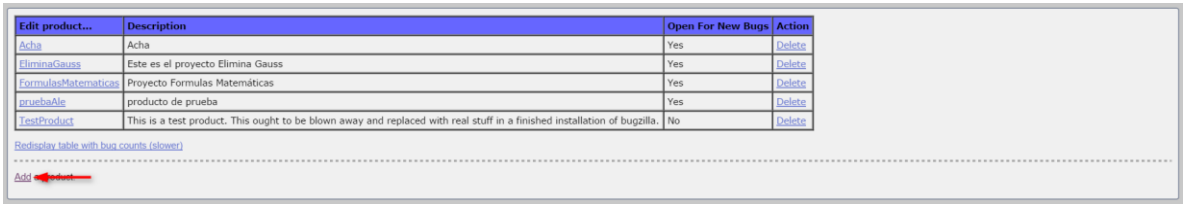
Para poder realizarlo, se deberá ir a la opción "Administración" en la parte superior de la pantalla, como se señala en la siguiente imagen.



En la ventana que se muestra a continuación entrar a la opción "products".



En la nueva ventana se pueden ver los proyectos creados, así como una descripción de los mismos. Debajo de ellos, hay un link con la palabra “Add” mediante el cual se podrá crear un nuevo producto.

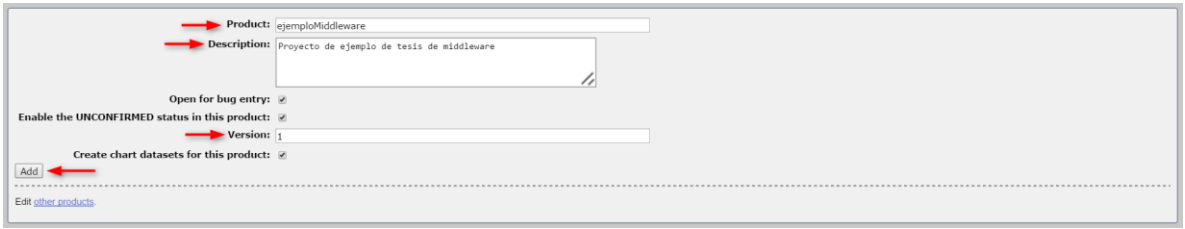


En la ventana de nuevo producto se deberán completar los campos marcados con flechas rojas y luego presionar el botón Add. Los campos a completar son los siguientes: (de arriba hacia abajo)

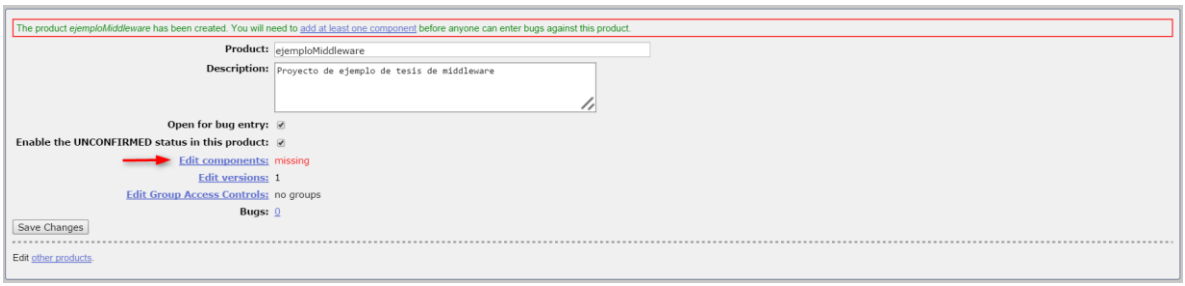
Product: Este campo debe ser llenado con el nombre del producto que se esta creando. Es importante que el producto creado tenga exactamente el mismo nombre que el proyecto creado en Cruise Control en la sección anterior. Esto es debido a que el programa middleware usa estos campos para sincronizar los proyectos.

Description: Descripción del proyecto que se está creando

Version: Versión del producto que se está creando



Una vez presionado el botón “Add” se mostrará la siguiente ventana. Una de las opciones que muestra es la marcada con la flecha, “Edit components”, mediante la cual se pueden agregar componentes al producto recién creado.



Luego de presionar el link “Edit components” se muestra la siguiente ventana, la cual contiene los componentes asociados al producto elegido. Debido a que el producto es nuevo, la ventana se encuentra vacía. Presionando el link “Add” se procederá a crear un nuevo componente.

Se deberán completar los siguientes campos para dar de alta un nuevo componente (de arriba hacia abajo):

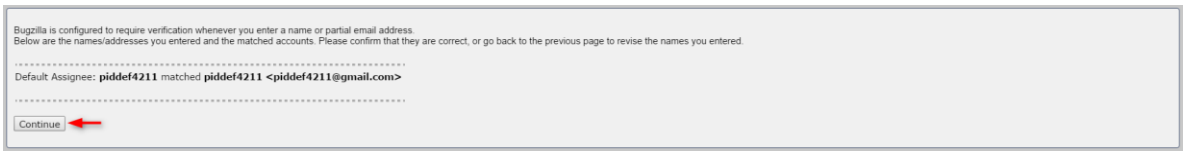
Component: Nombre del nuevo componente

Component Description: Descripción del componente que se está creando

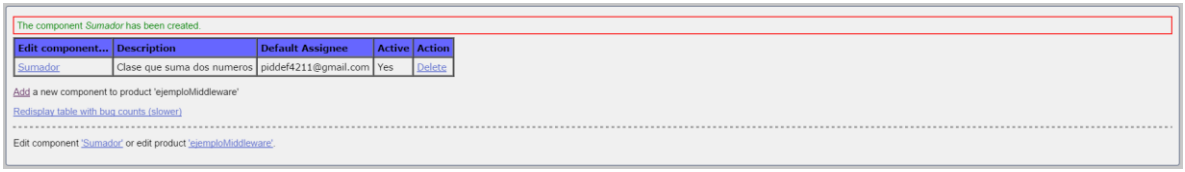
Default Assignee: Persona asignada al nuevo componente

Presionando el botón Add se termina de completar la creación del componente

Una vez presionado el botón, el sistema asocia el componente a la cuenta especificada. Presionando el botón “Continue” se prosigue.



A continuación se ve el nuevo componente creado



Teniendo el producto y el componente creado, se procede a crear el Test Plan dentro del cual se encontrarán los Test Cases.

Para hacer esto en la parte superior de la pantalla se presiona en la opción “New Plan”, como se ve en la imagen.

En la ventana que se abre en esta opción se deben completar los siguientes campos (de izquierda a derecha y de arriba hacia abajo):

Plan name: Nombre del plan que se está creando

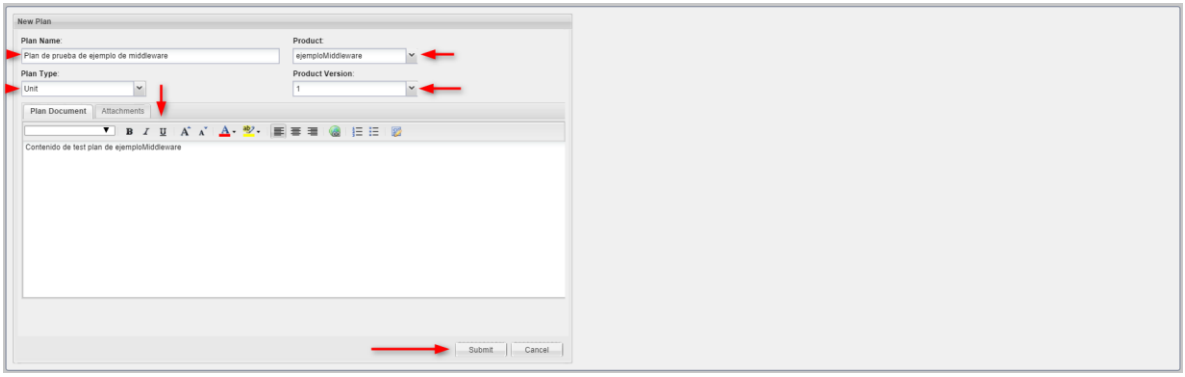
Product: Producto al que el plan estará asociado. En este caso se deberá seleccionar el producto creado en los pasos anteriores (que tiene el mismo nombre que el proyecto de Cruise Control creado en la sección anterior)

Plan Type: En esta selección se especifica el tipo de plan a crear (los tipos disponibles por defecto son: Aceptación, Función, instalación, integración, interoperabilidad, performance, producto, sistema y unitario). Este campo es informativo, no cambia la estructura del proyecto.

Product version: En este campo se puede seleccionar alguna de las versiones del producto elegido (en caso que tenga varias)

Plan document: En este campo es donde se escribe el cuerpo del plan. Adicionalmente se puede adjuntar archivos al mismo

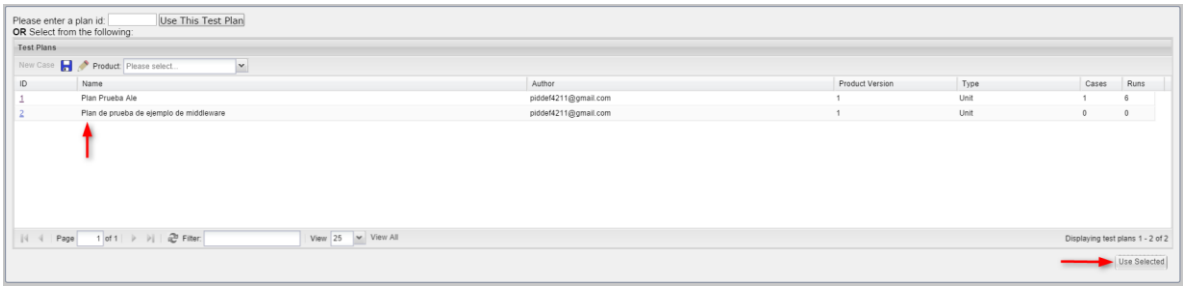
Una vez completados estos campos, presionar el botón “Submit”.



Una vez completados estos pasos se debe proceder a crear un nuevo Test Case.

Para realizarlo, en la parte superior de la pantalla seleccionar la opción “New Case”

Esto abrirá la siguiente ventana, en la cual se debe seleccionar un Test Plan al que estará asociado el nuevo caso de uso. En este caso se selecciona el plan creado en el paso anterior y se presiona el botón “Use Selected”.



Una vez elegido el plan se presenta la siguiente ventana, en la cual se deben completar los campos marcados con flechas (de izquierda a derecha y arriba a abajo):

Summary: Nombre del caso de uso

Default Tester: Persona que estará a cargo del caso de uso

Status: Estado del caso de uso (los casos posibles son: “propuesto”, “confirmado” y “deshabilitado”). Es importante que el nuevo caso de uso quede en estado confirmado, ya que este es el único caso en el que se puede usar (los otros casos son para administrarlo pero no dejan que se ejecute).

Alias: En este campo se selecciona el script que probara el caso de uso en caso de ser automatizado. Para que la automatización mediante middleware funcione es importante que se **escriba correctamente el nombre de la clase de prueba** que va a realizar el caso de uso. **El nombre debería ser package.nombreDeLaClase.**

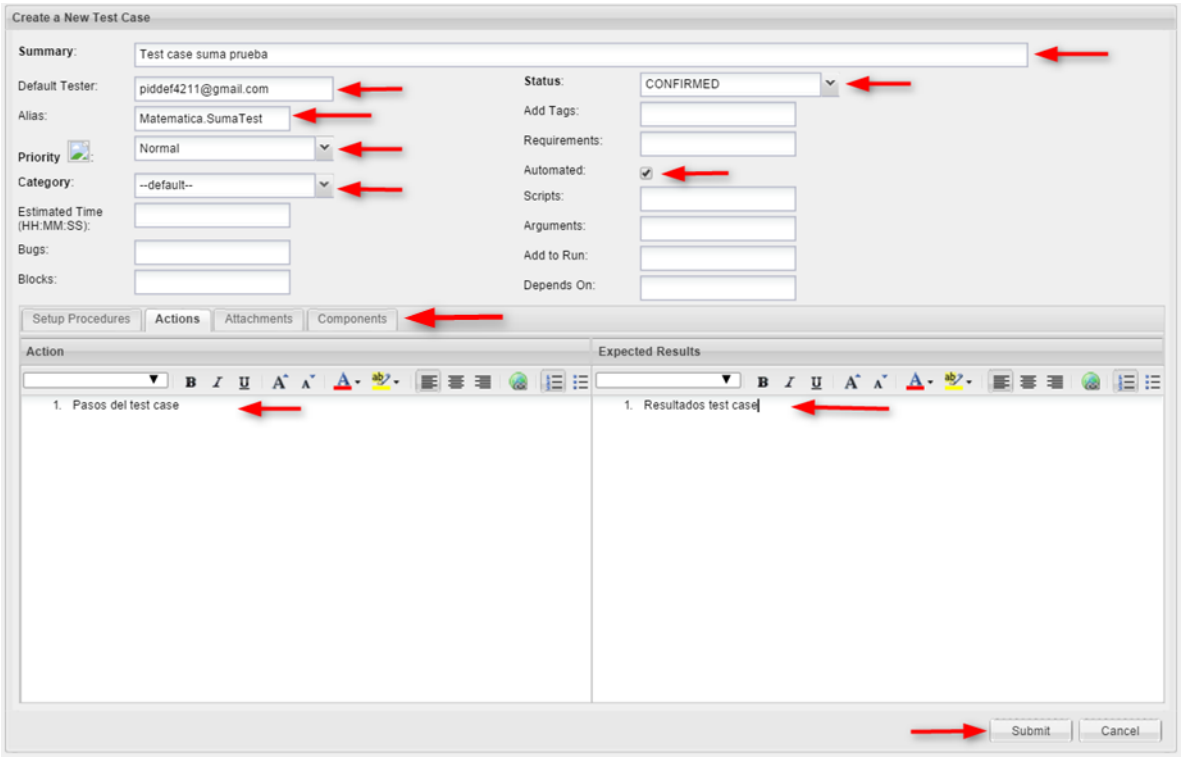
Priority: Prioridad del caso de uso

Category: Categoría del caso de uso, usada para administración de los mismos.

Automated: Campo para decir si el caso de uso es automatizado o no. Para que la ejecución mediante middleware funcione este campo debe estar chequeado. Este campo se deshabilitará automáticamente en caso que el script de prueba falle para evitar spam en la persona encargada. En caso de que falle la prueba, para que continúe ejecutándose automáticamente, este campo debe ser chequeado manualmente.

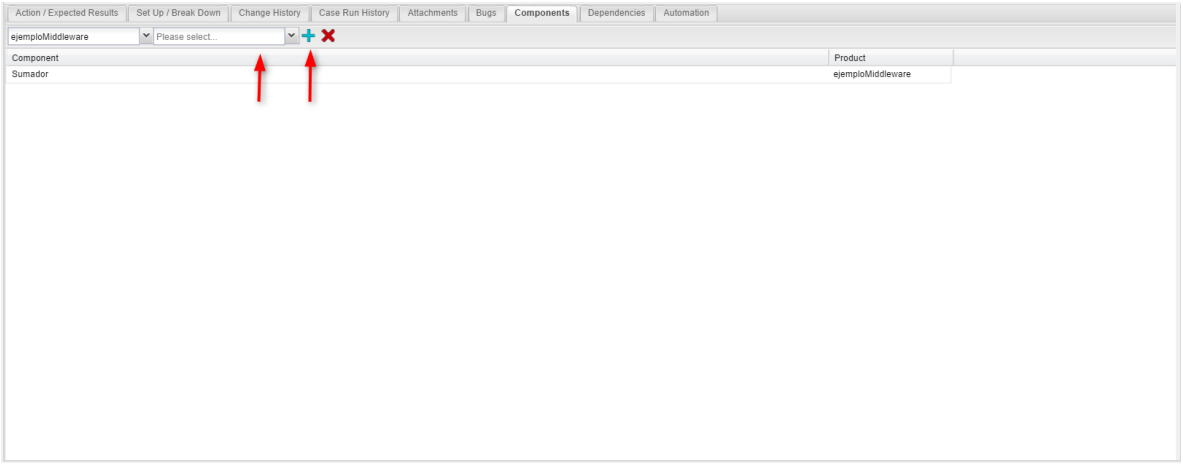
Acciones: acciones a realizar para ejecutar el caso de uso

Expected results: resultados esperados de la ejecución del caso de uso



Una vez completados los campos marcados con flechas, se selecciona la pestaña “components” para agregar componentes del proyecto a este caso de uso.

En la parte inferior de la pantalla se ve seleccionado el producto que se agregó al Test Plan. A la derecha de este campo se ve una lista de componentes de este producto. Para añadir los que el presente Test Case probará, se debe seleccionar y presionar el botón “más” en color azul.



Una vez completados los campos marcados con flechas se presiona el botón “submit” para terminar con este paso.

Resumen de pasos para la utilización de Bugzilla/Testopia

1) Ingresar a la url donde se encuentran publicados Bugzilla/Testopia. Ej: http://accesodi.iua.edu.ar/testopia/bugzilla/
2) Ingresar utilizando las siguientes credenciales:
    a) Usuario: piddef4211@gmail.com
    b) Contraseña: piddef4211
3) Crear un nuevo producto asociado al proyecto en Cruise Control
    a) Ir a Administración -> Products -> Add

b) Completar campos:

   i) Product: nombre del producto. Deberá ser igual al proyecto creado en Cruise Control.
   ii) Description: descripción del producto
   iii) Version

c) Seleccionar “Add”

d) Seleccionar “Edit Components”
   i) Crear un nuevo componente. Seleccionar “Add”.
   ii) Completar los campos:
      (1) Component: nombre del nuevo componente
      (2) Component Description: descripción del componente
      (3) Default Assignee: persona asignada al nuevo componente
   iii) Seleccionar “Add” para añadir el componente

e) Selección “Continue” para visualizar el componente y producto creado.

4) Crear Test Plan

a) Barra de Producto -> New Plan

b) Completar los campos:
   i) Plan Name: nombre del plan
   ii) Product: producto al que el plan estará asociado. (Creado en los pasos anteriores, de mismo nombre que el proyecto en CC)
   iii) Plan Type: tipo de plan a crear. Es solo informativo
   iv) Product version: versión del producto elegido, en el caso de que éste posea varias versiones.
   v) Plan Document: Se describe el Test Plan en sí. Se pueden adjuntar archivos al mismo.
   vi) Seleccionar “Submit”

5) Crear Test Case

a) Barra de Producto -> New Case

b) Seleccionar Test Plan al que estará asociado

c) Seleccionar “Use Selected”

d) Completar los siguientes campos:
   i) Summary: nombre del caso de uso
   ii) Default Tester: persona a cargo del caos de uso
   iii) Status: estado del caso de uso (propuesto, confirmado, deshabilitado)

iv) Alias: se selecciona el script que probará el caso de uso en caso de ser automatizado. (IMPORTANTE: Para que la automatización mediante middleware funcione se debe **escribir correctamente el nombre de la clase de prueba** que va a realizar el caso de uso. **El nombre debería ser package.nombreDeLaClase.**)
v) Priority: Prioridad del caso de uso
vi) Category: categoría del caso de uso. Necesaria para su administración.
vii) Automated: seleccionar si el caso de uso es automático o no. (IMPORTANTE 1: Para que la automatización mediante middleware funcione este campo de estar chequeado; IMPORTANTE 2: Este campo se deshabilitará automáticamente si el script falla)
viii) Acciones: acciones a realizar para ejecutar el caso de uso
ix) Expected Results: resultados esperados de la ejecución del caso de uso.

e) Seleccionar la pestaña "Componets", para añadir componentes del producto al caso de uso actual.
   i) Se mostrará una lista de componentes del producto asociado al Test Plan
   ii) Elegir, y añadir con el botón "más" en color azul.

f) Seleccionar "Submit" para finalizar

3.3 Configuración de middleware

El programa middleware necesita de parámetros de configuración para poder conectarse tanto con Bugzilla como con Cruise Control.

Estos parámetros se configuran en un archivo llamado ***"config.txt"***, que deberá estar ubicado en el mismo directorio en el que está ubicado el archivo .jar del programa middleware.

El archivo ***"config.txt"*** deberá tener los siguientes campos:

////////////////////////////// Configuración de conexiones//////////////////////////

//URL de Bugzilla

Bugzilla_URL=http://127.0.0.1/bugzilla/xmlrpc.cgi

//Usuario que usará la aplicación para realizar cambios (Debe poder leer y escribir en los proyectos que se quieran automatizar)

Bugzilla_User=piddef4211@gmail.com

//Password del usuario anterior

Bugzilla_Password=piddef4211

//Directorio de la instalación de Cruise Control donde se encuentra la carpeta projects

Cruise_Control_Home=/opt/cruisecontrol/

////////////////////////////Configuración de Bugs Automáticos//////////////////////////

Bugs_platform=PC

Bugs_summary=Prueba de configuración

Bugs_op_sys=Linux

Bugs_priority=Normal

Bugs_severity=Normal

////////////////////////Configuración TestRuns automáticos////////////////////////////

//Id del enviroment en donde se ejecutan las pruebas automatizadas

Enviroment=1

//Id del usuario encargado de la ejecución (a quien se le asignara)

Manager=1

El archivo que se ve anteriormente consta de tres partes.

En la primera se encuentran los parámetros para realizar las conexiones con Bugzilla y Cruise Control.

En la segunda se encuentran las configuraciones que se le aplicaran a los Bugs que se creen automáticamente, por ejemplo si el programa encuentra un test case que no brinde el resultado esperado se creará un bug titulado “Prueba de configuración”.

Por último la tercera parte del archivo de configuración son las configuraciones de los TestRuns que la aplicación creará, en este caso todos los testruns creados tendrán como persona asignada al usuario con id 1 (piddef4211).

(Los campos, como se ve en el ejemplo presentado anteriormente, están comentados para informar a que corresponde cada uno.)

3.4 Prueba de funcionamiento

Lo desarrollado hasta el momento en las secciones anteriores nos permite tener configurado correctamente Cruise Control como también Bugzilla / Testopia. Ahora es el momento de realizar las pruebas de funcionamiento de estos componentes.

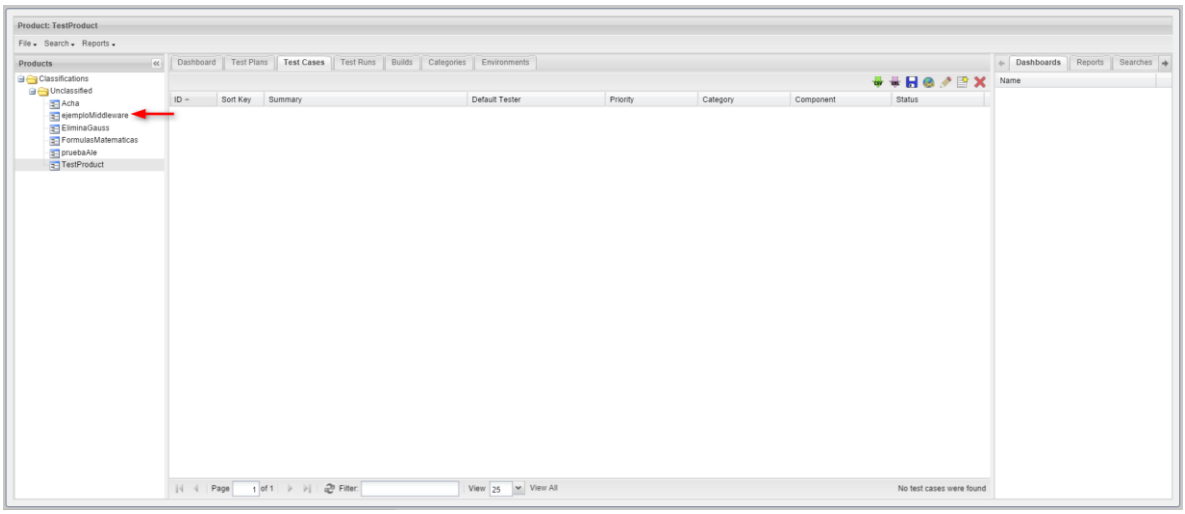
Para probarlos se debe ejecutar el siguiente comando para iniciar Cruise Control:

"service cruisecontrol start"

```
piddef4211@integracion-testopia:/opt/cruisecontrol$ sudo service cruisecontrol start
```

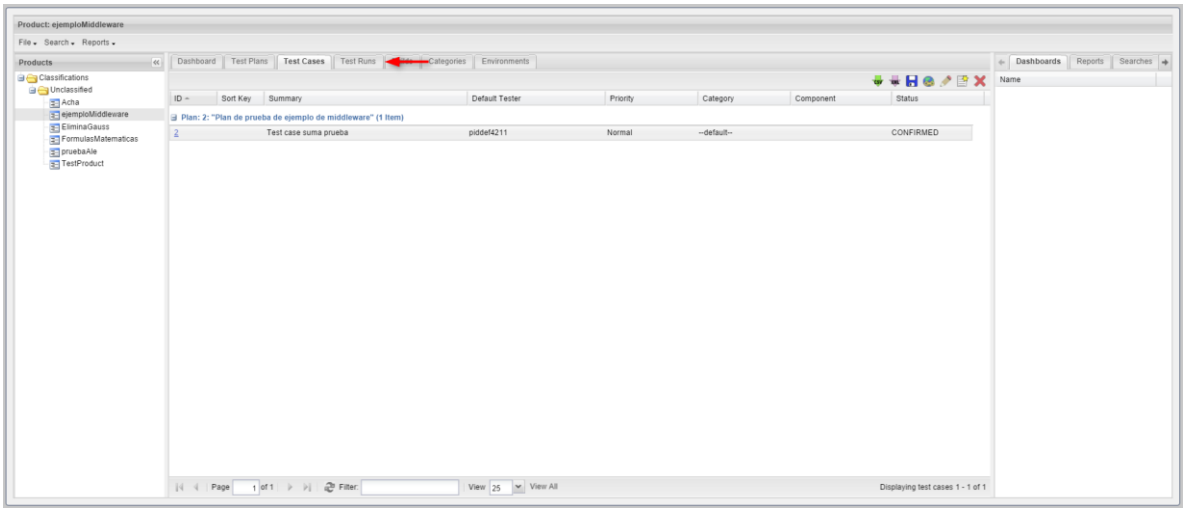
Una vez ejecutado, ingresaremos en la url de Bugzilla/Testopia y elegimos la opción "Product Dashboard" como me puede ver en la siguiente imagen.

Se mostrará en la parte izquierda los productos que se encuentran cargados en Bugzilla/Testopia. En este lugar se debe seleccionar el producto creado en la sección anterior.



Una vez seleccionado el producto correcto se podrá acceder a la información del mismo.

En este caso se necesita tener información de Test Runs (ejecuciones individuales de los casos de prueba). Para obtener esta información, seleccionar la pestaña Test Runs



Caso exitoso

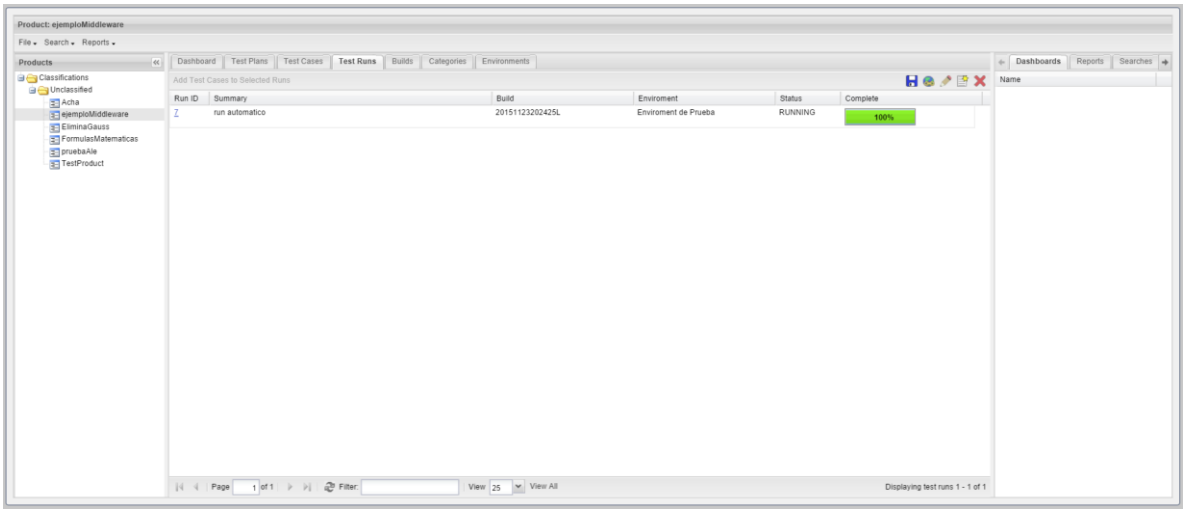
Si todo se encuentra configurado correctamente, cada vez que se ejecute Cruise Control (en este ejemplo se configuró cada cinco minutos) se creará un nuevo Test Run y se mostrará el resultado del mismo.

En la siguiente imagen se puede ver que el test run se ejecutó exitosamente. También se puede ver asociado al Test Run, una Build. La Build tiene el mismo nombre del log de ésta ejecución en Cruise Control (ubicado en el directorio:

“cruisecontrol/logs/nombreDelProyecto”).

Mediante esto se puede obtener el log completo de esta ejecución.

IMPORTANTE: hay que mencionar que en caso de que la prueba no resulte como se esperó, el recuadro verde de la derecha estará rojo, el caso de uso se desactivará y un nuevo bug será creado, asociado al test case.



Caso no exitoso

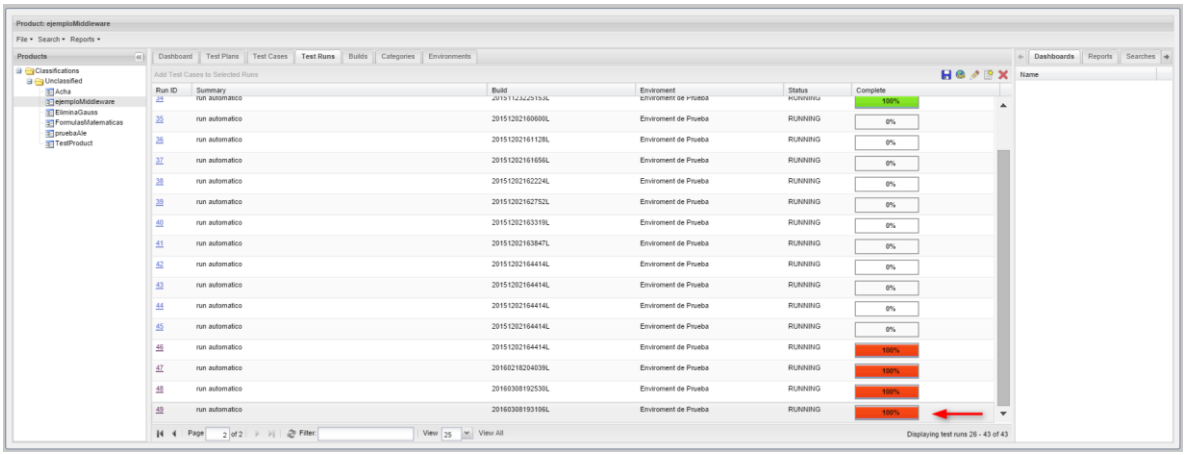
Para probar el caso no exitoso, se tomará la clase de prueba creada anteriormente y se le cambiará el resultado esperado (de modo que el test case falle).

Se cambia la línea *“assertTrue(resultado = 10.0)”* por *“assertTrue(resultado == 11)”*

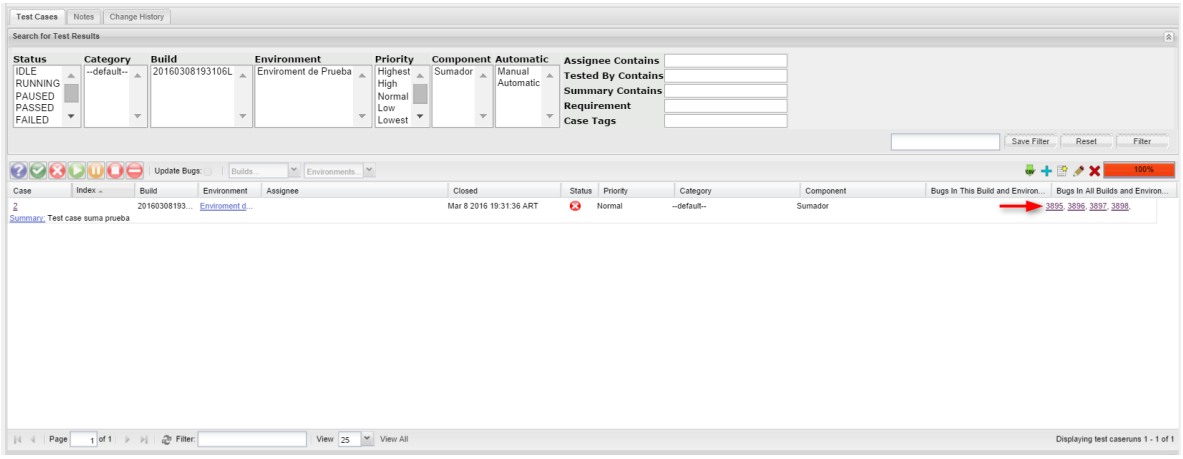
(hace que la clase de prueba espere un 11 del método llamado anteriormente en vez de un 10).

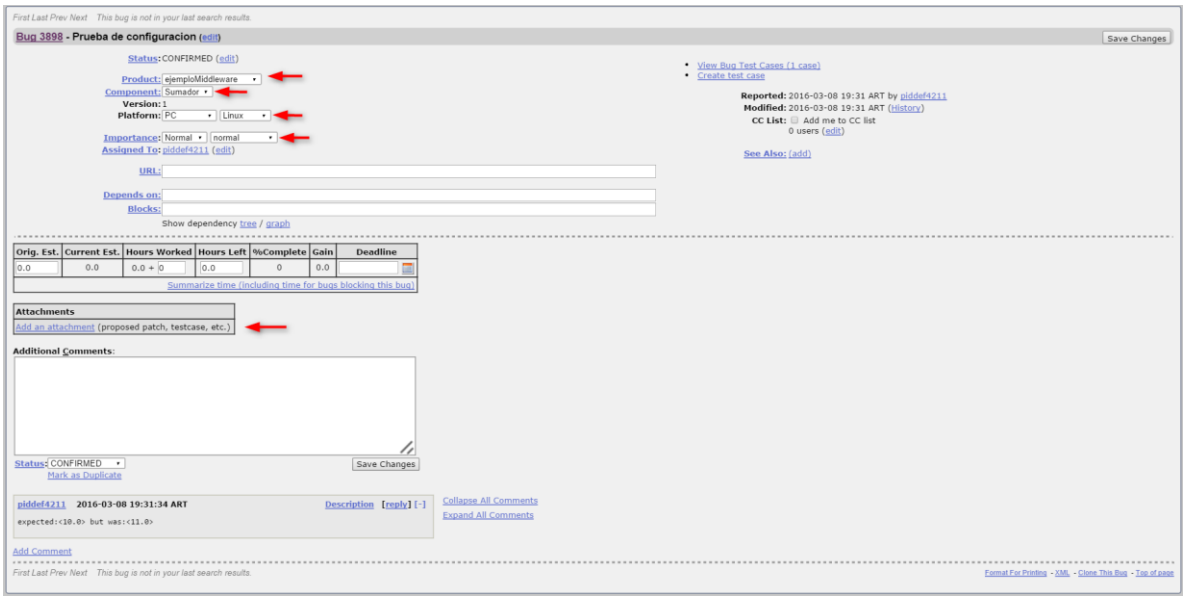
```
package Matematica;

import static org.junit.Assert.*;

import org.junit.Test;

import Matematica.Suma;

public class SumaPrueba {

        @Test
        public void test() {
                Suma suma = new Suma(5.0, 5.0);
        double resultado = suma.suma();
        assertTrue(resultado == 11.0);
        }

}
```

Una vez realizado el cambio, se espera a que se ejecute nuevamente Cruise Control y se podrá ver en el "Product Dashboard" un nuevo test run que habrá fallado, como puede apreciarse en la siguiente imagen.



Si se ingresa a ver el Test Case correspondiente al Test Run que falló, se podrá ver que el mismo tiene un nuevo bug. Este se habrá creado con la información que se configuró en el programa middleware, junto con información obtenida de la ejecución de la prueba por parte de Cruise Control.



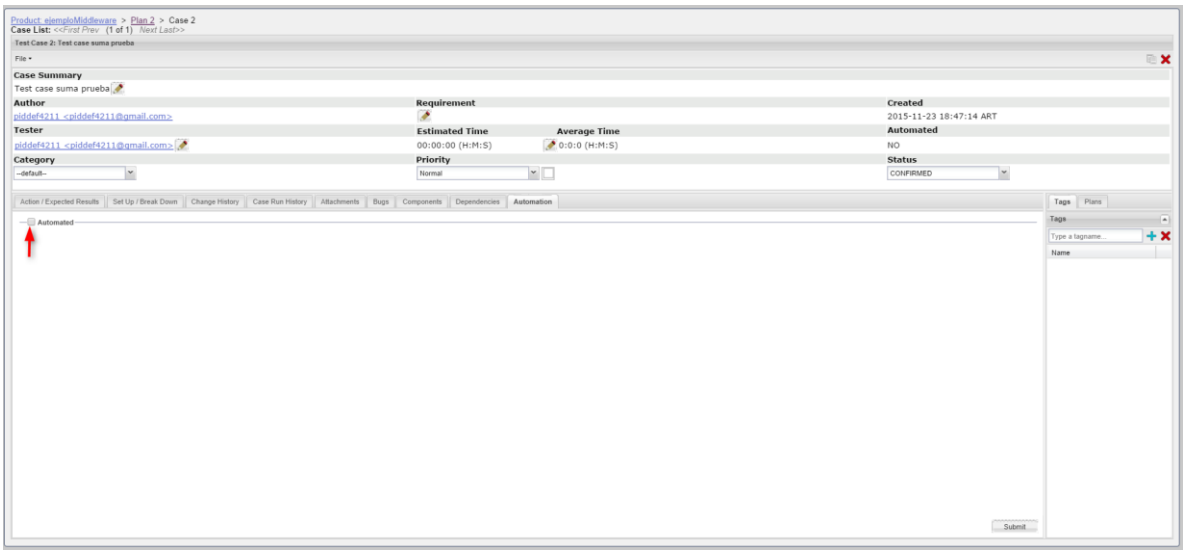


Si se inspecciona el Test Case correspondiente a la prueba ejecutada y se va a la pestaña de automatización, se podrá observar que el Test Case se encuentra sin automatización.

Esto sucede para que Cruise Control no siga generando bugs repetidos.

Una vez que se haya corregido el error que el case encontró, el tester deberá automatizar nuevamente el case, tildando la casilla automatizar.

Hecho esto en la siguiente ejecución de Cruise Control se ejecutará nuevamente el case.

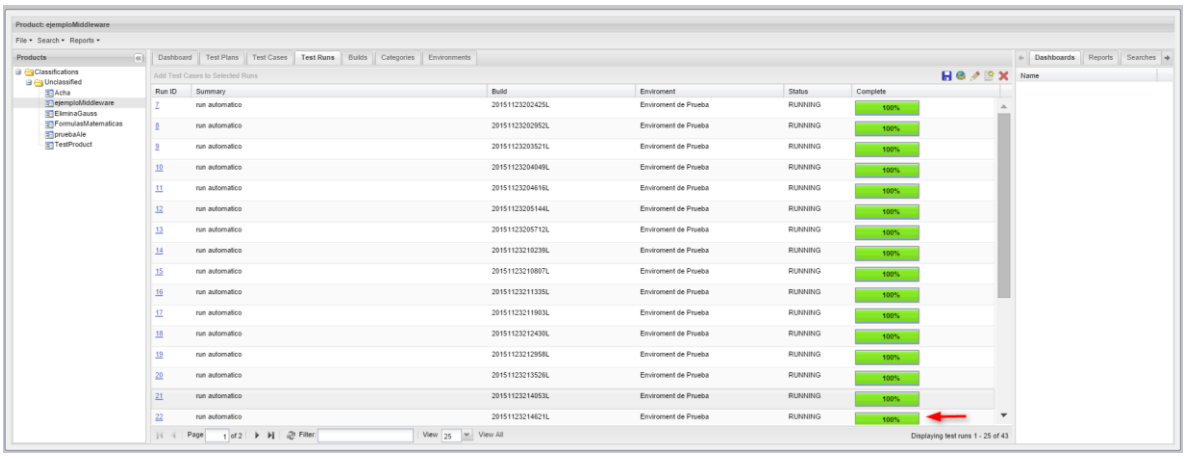


Casos exitosos

En caso que el case se ejecute correctamente, se continuará ejecutando de la misma forma por cada vez que Cruise Control ejecute el proyecto.

Cada una de estas ejecuciones creará un nuevo Test Run, el cual será exitoso a menos que falle.

Para detener la ejecución del case, bastara con abrirlo, ir a la pestaña de automatización y sacar el tilde de la casilla automatizar.



El tester encargado del test case se encargara de revisar el fallo y una vez reparado volver a subir el test case

Resumen de pasos para realizar Prueba de Funcionamiento

1. Ejecutar el comando *"service cruisecontrol start"*

2. Ingresar a url Bugzilla/Testopia: http://accesodi.iua.edu.ar/testopia/bugzilla/

3. Elegir la opción "Product Dashboard" del panel de herramientas.

4. Seleccionar el Producto creado en la sección anterior.

5. Obtener información de los Test Runs (ejecuciones individuales de los casos de prueba):
    a. Seleccionar la pestaña Test Runs
6. Realizar acciones según sea el resultado del case.

Conclusión

El actual proyecto surgió de la necesidad de brindar una solución factible a los problemas derivados de la gestión y ejecución de pruebas, dentro de una arquitectura de desarrollo planteada para el Proyecto PIDDEF 42/11 en el Instituto Universitario Aeronáutico.

Estas prácticas solían dejarse generalmente para las últimas etapas del proceso de desarrollo de software, por lo que se realizaban de una manera apresurada, o en algunos casos, directamente no se llevaban a cabo.

Con la realización del presente trabajo se logró establecer los lineamientos que dan soporte a la validación y verificación del software desarrollado en la Institución favoreciendo al aseguramiento de la calidad de los mismos.

Para lograr y cumplir el objetivo, se llevaron a cabo actividades de investigación respecto al manejo de pruebas en los desarrollos de software en general y al desarrollo del software científico en particular.

Se logró desarrollar un componente middleware que permitió la interconexión entre las distintas tecnologías que forman parte de la Arquitectura de Desarrollo, que facultó automatizar el seguimiento de las tareas de pruebas, favoreciendo a una mayor documentación, organización, trazabilidad de las pruebas que se llevan a cabo, seguimiento de avance e incidencias.

Además se establecieron guías de prácticas referenciales y de roles para los equipos de desarrollo de la institución, que mediante su puesta en práctica guiarán el proceso de desarrollo en vías de lograr la calidad de software deseado. No obstante estas prácticas se encuentran dispuestas a una continua evolución a medida que los equipos se familiaricen con el proceso.

En el plano personal, éste Trabajo Final de Grado presentó varios desafíos acerca de tecnologías, prácticas y campos de acción que se desconocían con anterioridad, lo que significó una gran motivación para cumplir con los objetivos que se plantearon, y una posterior satisfacción al lograrlos.

Bibliografía

1. **Departamento de Informática, Instituto Universitario Aeronáutico.** *Automatización del proceso de comunicación entre el sistema de integración contínua y el sistema de gestión de incidencias.* Córdoba : s.n. pág. 1.

2. **Aeronáutico, Departamento de Informática Instituto Universitario.** *Testing en el Desarrollo de Software Científico en el marco de la Integración Continua.* Córdoba : s.n.

3. **Rodríguez, Federico Toledo.** *Introducción a las pruebas de sistemas de información.* Montevideo - Uruguay : s.n., 2014.

4. **Wikipedia.** Wikipedia. [En línea] [Citado el: 20 de Octubre de 2015.] https://es.wikipedia.org/wiki/Normas_ISO_9000.

5. **Wikifoundry.** Ingeniería de Software. [En línea] [Citado el: 11 de Noviembre de 2015.] http://clases3gingsof.wikifoundry.com/page/FURPS.

6. **Sur, Departamento Ciencias e Ingeniería de la Computación Univerisdad Nacional del.** Universidad del Sur. [En línea] [Citado el: 3 de Noviembre de 2015.] http://www.cs.uns.edu.ar/~prf/teaching/SQ07/clase6.pdf.

7. **Plata, Universidad Nacional de La.** [En línea] [Citado el: 6 de Octubre de 2015.] http://sedici.unlp.edu.ar/bitstream/handle/10915/3956/3_-_Aseguramiento_de_la_calidad_del_software.pdf?sequence=11.

8. **(R. Dupuis, P. Bourque, A. Abran, J. W. Moore, and L. L. Tripp.** The SWEBOK Project: Guide to the software engineering body of knowledge,. 2003.

9. **Uruguay, Facultad de Ingeniería Universidad de la República.** [En línea] [Citado el: 26 de Noviembre de 2015.] https://www.fing.edu.uy/tecnoinf/mvd/cursos/ingsoft/material/teorico/is09-Verificacion-Validacion.pdf.

10. **Kshirasagar Naik, Priyadarshi Tripathy.** *Software Testing and Quality Assurance - Theory and Practice.* s.l. : Wiley, 2008.

11. **Waterloo, Department of Electrival and Computer Engineering University of.** *Software Testing And Quality Assurance.* Waterloo : Wiley, 2008.

12. **Sevilla, Departamento de Lenguajes y Sistemas Informáticos Universidad de.** Universidad de Sevilla. [En línea] [Citado el: 6 de Octubre de 2015.] http://www.lsi.us.es/docencia/get.php?id=361.

13. **Lewis, William E.** *Software Testing and Continuos Quality Improvement.* s.l. : Aurbach, 2004.

14. **Cycle, Software Testing Life.** [En línea] [Citado el: 7 de Noviembre de 2015.] http://www.softqanetwork.com/software-testing-life-cycle.

15. **Board, International Software Testing Qualifications.** [En línea] [Citado el: 5 de Octubre de 2015.] http://www.istqb.org/.

16. **Software, Tipos de Prueba de.** Ingeniería de Software Blogspot. [En línea] [Citado el: 12 de Octubre de 2015.] http://ing-sw.blogspot.com.ar/2005/04/tipos-de-pruebas-de-software.html.

17. **Información, Universidad de Castillas - Departamento de Tecnologías y Sistemas de.** *Pruebas de Sistemas de Informaicón.*

18. **devBistro.** Testing & The Role of a Test Lead / Manager. [En línea] [Citado el: 29 de Octubre de 2015.] http://www.devbistro.com/articles/Testing/Role-of-Test-Lead-Manager.

19. **Association, Chicago Quality Assurance.** Practical Metrics for Managing and Improving Software Testing. [En línea] [Citado el: 10 de Octubre de 2015.] http://www.cqaa.org/Resources/Documents/Presentations%202010/Practical%20MetricsForTesting.pdf .

20. **Help, Software Testing.** Important Software Test Metrics and Measurements. [En línea] http://www.softwaretestinghelp.com/software-test-metrics-and-measurements/.

21. **Inc, Red Hat.** Fedora Hosted. *Fedora Hosted .* [En línea] [Citado el: 15 de Marzo de 2016.] https://fedorahosted.org/nitrate/wiki/XMLRPC-APIs.

22. **Network, Mozilla Developer.** MozillaWiki. *MozillaWiki.* [En línea] [Citado el: 5 de Noviembre de 2015.] https://wiki.mozilla.org/Bugzilla:Addons#Test_case_management_systems.

23. **Natarajan, Ramesh.** Bugzilla Installation Guide for Linux. *The Geek Stuf.* [En línea] [Citado el: 2015 de Octubre de 27.] http://www.thegeekstuff.com/2010/05/install-bugzilla-on-linux/.

24. **Consulting, Gavaldo.** QA Tools. *XQual.* [En línea] [Citado el: 2015 de Octubre de 20.] http://www.xqual.com/qa/tools.html.

25. **OWN2.** SalomeTMF. *OWN2 Consortium.* [En línea] [Citado el: 6 de Noviembre de 2015.] http://wiki.ow2.org/salome-tmf/.

26. **Excellence, Testing.** Best open source test management tools. *Testing Excellence - Lear Software Testing.* [En línea] [Citado el: 6 de Noviembre de 2015.] http://www.testingexcellence.com/best-open-source-test-management-tools/.

27. **P.C., Exelixis Media.** Parsing XML using DOM, SAX and StAX Parser in Java. *Java Code Geeks - Java Developers resource center.* [En línea] [Citado el: 28 de Noviembre de 2015.] https://www.javacodegeeks.com/2013/05/parsing-xml-using-dom-sax-and-stax-parser-in-java.html.

28. **Help, Software Testing.** Important Software Test Metrics and Measurements. *Software Testing Help.* [En línea] [Citado el: 15 de Marzo de 2016.] http://www.softwaretestinghelp.com/software-test-metrics-and-measurements/.

29. **Association, Chicago Quality Assurance.** Practical Metrics for Managing and Improving Software Testing . *CQAA .* [En línea] [Citado el: 15 de Marzo de 2016.] http://www.cqaa.org.

30. **Tian, Jeff.** *Software Quality Engineering.* New Jersey : Wiley Interscience, 2005.

31. **Elfriede, Dustin, Thom, Garret y Bernie, Gauf.** *Implementing Automated Software Testing.* Massachussets : Pearson Education Inc, 2009.

Anexos

1. Protocolo XMLRPC

Es un protocolo de llamada a procedimiento remoto que usa XML para codificar los datos y HTTP como protocolo de transmisión de mensajes. Está diseñado para ser los más sencillo posible, mientras permite transmitir, procesar y retornar estructuras de datos complejas.

Request

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181


<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
      <value><i4>41</i4></value>
      </param>
    </params>
  </methodCall>
```

Ejemplo de request

Requerimientos de cabecera

El formato de la URI en la primera línea de la cabecera no está especificado. Por ejemplo, puede estar vacío, una barra sola en el caso que el servidor este manejando solo llamadas XML-RPC. Si el servidor

maneja varias peticiones HTTP, se permite la URI para ayudar a enrutar al código que maneja las peticiones.

Un User-Agent y Host deben ser especificados

El Content-Type es text/xml

El campo Content-Length debe ser especificado y debe ser correcto.

Formato del contenido

El contenido esta en XML, una sola estructura <methodCall>.

El tag <MethodCall> debe contener un tag <methodName>, una cadena que contiene el nombre del método al que se llama. La cadena puede tener caracteres alfanuméricos, guiones bajos, puntos, dos puntos y barras diagonales.

Si el método tiene parámetros, el tag <methodCall> debe contener un tag <params>. Este tag puede contener el numero de tags <param> que sea necesario, cada uno de los cuales debe tener un tag <value>

Etiqueta <value>

El contenido de esta etiqueta puede ser escalar, el tipo es indicado usando uno de los sig**uientes tag**

TAG	**Tipo**	**Ejemplo**
<i4> o <int>	Entero con signo de 4 bytes	-12
<boolean>	0 o 1	1
<string>	string	Hola
<double>	Puto flotante con signo (doble precisión)	-12.223
<dateTime.iso8601>	Date/time	19980717T14:08:55
<base64>	Binario codificado en base 64	eW91IGNhbid0IHJlYWQgdGhpcyE=

Una etiqueta <value> puede ser también una estructura. Para ello se usa el tag <struct>

Un <struct> contiene etiquetas <member> y cada uno de estos contiene un tag <name> y un <value>. Las estructuras pueden ser recursivas, por lo que un <value> puede contener una estructura o un arreglo dentro.

Por último, la etiqueta value puede ser del tipo arreglo. Para esto se usa la etiqueta <Array>

Un <Array> contiene un solo tag <data> que contiene el número que se necesite de tags <value>. En los arreglos se pueden mezclar tipos de datos y pueden ser recursivos.

Response

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 426
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:02 GMT
Server: UserLand Frontier/5.1.2-WinNT


<?xml version="1.0"?>
<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
          <name>faultCode</name>
          <value><int>4</int></value>
          </member>
        <member>
          <name>faultString</name>
          <value><string>Too many parameters.</string></value>
          </member>
        </struct>
      </value>
    </fault>
  </methodResponse>
```

Ejemplo de response con error

Formato

A menos que se encuentre un error de nivel más bajo siempre devuelve 200 OK

El content-type es text/xml. Content-Length debe estar presente y ser correcto

El cuerpo es una sola estructura XML, un <methodResponse> que contiene un solo <params> que contiene un solo <param> con un solo <value>

El tag <methodResponse> puede contener tambien un <fault> que contiene un <value> que es un <struct> que contiene dos elementos, uno llamado <faultCode>, un <int> y uno llamdo <faultString>, un <string>.

Un <methodResponse> no pude contener ambos, <fault> y <params>

2. Apache XML-RPC

Apache XML-RPC es una implementación en java de XML-RPC

La clase cliente de esta implementación es org.apache.xmlrpc.XmlRpcClient

En este trabajo se usaron llamadas síncronas a los métodos de Bugzilla. Un ejemplo sencillo de esto es:

```
XmlRpcClient xmlrpc = new XmlRpcClient ("http://localhost:8080/RPC2");
  Vector params = new Vector ();
  params.addElement ("some parameter");
  // this method returns a string
  String result = (String) xmlrpc.execute ("method.name", params);
```

En el trabajo se usaron funciones extra debido a que Bugzilla solicita el ingreso mediante credenciales y guarda esa información en cookies.

3. Javadoc Middleware

En esta sección se agrega el javadoc resultante de la aplicación middleware. Se agregó solo el de la clase *updateTestCaseRun* debido a que es el que tiene la lógica de la aplicación. Las clases resultantes solo se encargan de tareas que facilitan el acceso a datos y configuraciones, como, por ejemplo, abrir y leer los archivos de configuración, parsear los xml, guardar la información leída, etc...

Class UpdateTestCaseRun

- java.lang.Object
    - UpdateTestCaseRun

public class **UpdateTestCaseRun**

extends java.lang.Object

Clase se comunica con el web service de Bugzilla/Testopia, haciendo uso de la clase XmlRpcClient.

- **Constructor Summary**

Constructors
Constructor and Description
UpdateTestCaseRun()

- **Method Summary**

Methods	
Modifier and Type	**Method and Description**
void	**addBug**(java.lang.String product, java.lang.String component, java.lang.String version, **Reporte** reporte, java.lang.Integer caseId) Agrega un bug a Bugzilla y lo asocia al testcase que se le pasa
java.util.HashMap	**addRun**(java.lang.Integer planId, java.lang.Integer caseId, **Reporte** reporte, java.lang.String product)
void	**connect**() Método que realiza la conexión con el web service de Bugzilla/Testopia
void	**disconnect**()

java.util.List<java.util.HashMap>	**getActiveProducts**()
java.util.List<java.util.HashMap>	**getListAutomatedTestCase**(java.lang.Integer plan_id) Método que devuelve una lista de TestCase que están marcados como automáticos en Bugzilla
java.util.List	**getListTestPlan**() Método que obtiene una lista de los Test Plan en Testopia
java.util.List	**getPlansByProduct**(java.lang.String producto) Método que obtiene una lista de los TestPlan en Testopia
java.util.HashMap	**getProductData**(java.lang.Integer id) Retorna la información de un Producto solo si este tiene TestRuns asociados
java.util.HashMap	**getProductFromPlan**(java.lang.Integer planId) Obtiene un producto
java.util.List	**getTestCaseData**(java.lang.Integer idCase) Método que obtiene información de los TestCase de los que se le pasa el id
java.util.List<java.util.HashMap>	**getTestCaseRunList**(java.lang.Integer caseId)
void	**updateStatus**(**Reporte** reporte, java.lang.Integer idCaseRun) Método que cambia el estado de los TestRun

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

- **Constructor Detail**
  - **UpdateTestCaseRun**

public UpdateTestCaseRun()

- **Method Detail**
  - **connect**
  - public void connect()
  - throws java.net.MalformedURLException,

org.apache.xmlrpc.XmlRpcException

Método que realiza la conexión con el web service de Bugzilla/Testopia

Throws:

java.net.MalformedURLException

org.apache.xmlrpc.XmlRpcException

  - **disconnect**
  - public void disconnect()

throws org.apache.xmlrpc.XmlRpcException

Throws:

org.apache.xmlrpc.XmlRpcException

  - **updateStatus**
  - public void updateStatus(Reporte reporte,
  - java.lang.Integer idCaseRun)
  - throws java.net.MalformedURLException,

org.apache.xmlrpc.XmlRpcException

Método que cambia el estado de los TestRun

Parameters:

reporte - Información obtenida de la ejecución de las pruebas automáticas en Cruise Control

idCaseRun - Id del TestCaseRun al que se le cambiara el estado

Throws:

java.net.MalformedURLException

org.apache.xmlrpc.XmlRpcException

- **getListAutomatedTestCase**
- public java.util.List<java.util.HashMap> getListAutomatedTestCase(java.lang.Integer plan_id)
- throws java.net.MalformedURLException,

org.apache.xmlrpc.XmlRpcException

Método que devuelve una lista de TestCase que están marcados como automáticos en Bugzilla

Parameters:

plan_id - Integer que seÃ±ala el TestPlan en el cual se buscaran los TestCase

Returns:

Lista de Hash que contienen la información de los TestCase marcados como automáticos

Throws:

java.net.MalformedURLException

org.apache.xmlrpc.XmlRpcException

- **getListTestPlan**
- public java.util.List getListTestPlan()
- throws java.net.MalformedURLException,

org.apache.xmlrpc.XmlRpcException

Método que obtiene una lista de los TestPlan en Testopia

Returns:

Lista de hash que contiene los test plan de Testopia

Throws:

java.net.MalformedURLException

org.apache.xmlrpc.XmlRpcException

- **getPlansByProduct**
- public java.util.List getPlansByProduct(java.lang.String producto)
- throws java.net.MalformedURLException,

org.apache.xmlrpc.XmlRpcException

Método que obtiene una lista de los TestPlan en Testopia

Returns:

Lista de hash que contiene los test plan de Testopia

Throws:

java.net.MalformedURLException

org.apache.xmlrpc.XmlRpcException

- **getTestCaseData**
- public java.util.List getTestCaseData(java.lang.Integer idCase)
- throws java.net.MalformedURLException,

org.apache.xmlrpc.XmlRpcException

Método que obtiene información de los TestCase de los que se le pasa el id

Parameters:

idCase -

Returns:

Throws:

java.net.MalformedURLException

org.apache.xmlrpc.XmlRpcException

- **getTestCaseRunList**
- public java.util.List<java.util.HashMap> getTestCaseRunList(java.lang.Integer caseId)
- throws java.net.MalformedURLException,

org.apache.xmlrpc.XmlRpcException

Throws:

java.net.MalformedURLException

org.apache.xmlrpc.XmlRpcException

- **getProductFromPlan**
- public java.util.HashMap getProductFromPlan(java.lang.Integer planId)
- throws java.net.MalformedURLException,

org.apache.xmlrpc.XmlRpcException

Obtiene un producto

Parameters:

planId - Id del plan solicitado

Returns:

Hash del plan solicitado

Throws:

java.net.MalformedURLException

org.apache.xmlrpc.XmlRpcException

- **addBug**
- public void addBug(java.lang.String product,
- java.lang.String component,
- java.lang.String version,

- Reporte reporte,
- java.lang.Integer caseId)
- throws java.net.MalformedURLException,

org.apache.xmlrpc.XmlRpcException

Agrega un bug a Bugzilla y lo asocia al testcase que se le pasa

Parameters:

product - Producto en el que se produce el bug

component - Componente en el que se produce el bug

version - Versión en la que se produce el bug

reporte - reporte con la información del bug (mensaje)

caseId - caso de uso al que se le asigna el bug

Throws:

java.net.MalformedURLException

org.apache.xmlrpc.XmlRpcException

- **getActiveProducts**
- public java.util.List<java.util.HashMap> getActiveProducts()
- throws java.net.MalformedURLException,

org.apache.xmlrpc.XmlRpcException

Throws:

java.net.MalformedURLException

org.apache.xmlrpc.XmlRpcException

- **getProductData**
- public java.util.HashMap getProductData(java.lang.Integer id)
- throws java.net.MalformedURLException,

org.apache.xmlrpc.XmlRpcException

Retorna la información de un Producto solo si este tiene TestRuns asociados

Parameters:

id - Id del producto que se está buscando

Returns:

Hash con la información del producto solicitado

Throws:

java.net.MalformedURLException

org.apache.xmlrpc.XmlRpcException

- **addRun**
- public java.util.HashMap addRun(java.lang.Integer planId,
- java.lang.Integer caseId,
- Reporte reporte,
- java.lang.String product)
- throws java.io.IOException,

org.apache.xmlrpc.XmlRpcException

Throws:

java.io.IOException

org.apache.xmlrpc.XmlRpcException

2. Taxonomía de Tipos de Pruebas

Exhibit 2.1. Testing Technique Categories

Technique	Manual	Automated	Static	Dynamic	Functional	Structural
Acceptance testing	x	x		x	x	
Ad hoc testing	x				x	
Alpha testing	x			x	x	
Basis path testing		x		x		x
Beta testing	x			x	x	
Black-box testing		x		x	x	
Bottom-up testing		x		x		x
Boundary value testing		x		x	x	
Branch coverage testing		x		x		x
Branch/condition coverage		x		x		x
Cause-effect graphing		x		x	x	
Comparison testing	x	x		x	x	x
Compatibility testing	x	x				x
Condition coverage testing		x		x		x
CRUD testing		x		x	x	
Database testing		x		x		x
Decision tables		x		x	x	
Desk checking	x			x		x
End-to-end testing	x	x			x	

Exhibit 2.1. Testing Technique Categories (Continued)

Technique	Manual	Automated	Static	Dynamic	Functional	Structural
Equivalence partitioning		x		x		
Exception testing		x		x	x	
Exploratory testing	x			x	x	
Free form testing		x		x	x	
Gray-box testing		x		x	x	x
Histograms	x				x	
Incremental integration testing	x	x		x	x	
Inspections	x		x		x	x
Integration testing	x	x		x	x	
JADs	x				x	x
Load testing	x	x		x		x
Mutation testing	x	x		x	x	
Orthogonal array testing	x		x		x	
Pareto analysis	x				x	
Performance testing	x	x		x	x	x
Positive and negative testing		x		x	x	
Prior defect history testing	x		x		x	
Prototyping		x		x	x	
Random testing		x		x	x	

Range testing		x		x	x	
Recovery testing	x	x		x		x
Regression testing				x	x	
Risk-based testing	x		x		x	
Run charts	x		x		x	
Sandwich testing		x		x		x
Sanity testing	x	x		x	x	
Security testing	x	x				x
State transition testing		x		x	x	
Statement coverage testing		x		x		x
Statistical profile testing	x		x		x	
Stress testing	x	x		x		
Structured walkthroughs	x			x	x	x
Syntax testing		x	x	x	x	
System testing	x	x		x	x	
Table testing		x		x		x
Thread testing		x		x		x
Top-down testing		x		x	x	x
Unit testing	x	x	x			x
Usability testing	x	x		x	x	
User acceptance testing	x	x		x	x	
White-box testing		x		x		x

Technique	Brief Description
Acceptance testing	Final testing based on the end-user/customer specifications, or based on use by end-users/customers over a defined period of time
Ad hoc testing	Similar to exploratory testing, but often taken to mean that the testers have significant understanding of the software before testing it
Alpha testing	Testing of an application when development is nearing completion; minor design changes may still be made as a result of such testing. Typically done by end-users or others, not by programmers or testers
Basis path testing	Identifying tests based on flow and paths of a program or system
Beta testing	Testing when development and testing are essentially completed and final bugs and problems need to be found before final release. Typically done by end-users or others, not by programmers or testers
Black-box testing	Testing cases generated based on the system's functionality
Bottom-up testing	Integrating modules or programs starting from the bottom
Boundary value testing	Testing cases generated from boundary values of equivalence classes
Branch coverage testing	Verifying that each branch has true and false outcomes at least once
Branch/condition coverage testing	Verifying that each condition in a decision takes on all possible outcomes at least once
Cause–effect graphing	Mapping multiple simultaneous inputs that may affect others to identify their conditions to test
Comparison testing	Comparing software weaknesses and strengths to competing products
Compatibility testing	Testing how well software performs in a particular hardware/software/operating system/network environment
Condition coverage testing	Verifying that each condition in a decision takes on all possible outcomes at least once
CRUD testing	Building a CRUD matrix and testing all object creations, reads, updates, and deletions
Database testing	Checking the integrity of database field values
Decision tables	Table showing the decision criteria and the respective actions
Desk checking	Developer reviews code for accuracy

Technique	Brief Description
End-to-end testing	Similar to system testing; the "macro" end of the test scale; involves testing of a complete application environment in a situation that mimics real-world use, such as interacting with a database, using network communications, or interacting with other hardware, applications, or systems if appropriate
Equivalence partitioning	Each input condition is partitioned into two or more groups. Test cases are generated from representative valid and invalid classes
Exception testing	Identifying error messages and exception handling processes and conditions that trigger them
Exploratory testing	Often taken to mean a creative, informal software test that is not based on formal test plans or test cases; testers may be learning the software as they test it
Free form testing	Ad hoc or brainstorming using intuition to define test cases
Gray-box testing	A combination of black-box and white-box testing to take advantage of both
Histograms	A graphical representation of measured values organized according to the frequency of occurrence used to pinpoint hot spots
Incremental integration testing	Continuous testing of an application as new functionality is added; requires that various aspects of an application's functionality be independent enough to work separately before all parts of the program are completed, or that test drivers be developed as needed; done by programmers or by testers
Inspections	Formal peer review that uses checklists, entry criteria, and exit criteria
Integration testing	Testing of combined parts of an application to determine if they function together correctly. The "parts" can be code modules, individual applications, or client and server applications on a network. This type of testing is especially relevant to client/server and distributed systems.
JADs	Technique that brings users and developers together to jointly design systems in facilitated sessions
Load testing	Testing an application under heavy loads, such as testing of a Web site under a range of loads to determine at what point the system's response time degrades or fails
Mutation testing	A method for determining if a set of test data or test cases is useful, by deliberately introducing various code changes ("bugs") and retesting with the original test data/cases to determine if the "bugs" are detected. Proper implementation requires large computational resources.

Technique	Brief Description
Orthogonal array testing	Mathematical technique to determine which variations of parameters need to be tested
Pareto analysis	Analyze defect patterns to identify causes and sources
Performance testing	Term often used interchangeably with "stress" and "load" testing. Ideally "performance" testing (and any other "type" of testing) is defined in requirements documentation or QA or Test Plans
Positive and negative testing	Testing the positive and negative values for all inputs
Prior defect history testing	Test cases are created or rerun for every defect found in prior tests of the system
Prototyping	General approach to gather data from users by building and demonstrating to them some part of a potential application
Random testing	Technique involving random selection from a specific set of input values where any value is as likely as any other
Range testing	For each input identifies the range over which the system behavior should be the same
Recovery testing	Testing how well a system recovers from crashes, hardware failures, or other catastrophic problems
Regression testing	Testing a system in light of changes made during a development spiral, debugging, maintenance, or the development of a new release
Risk-based testing	Measures the degree of business risk in a system to improve testing
Run charts	A graphical representation of how a quality characteristic varies with time
Sandwich testing	Integrating modules or programs from the top and bottom simultaneously
Sanity testing	Typically an initial testing effort to determine if a new software version is performing well enough to accept it for a major testing effort. For example, if the new software is crashing systems every five minutes, bogging down systems to a crawl, or destroying databases, the software may not be in a "sane" enough condition to warrant further testing in its current state
Security testing	Testing how well the system protects against unauthorized internal or external access, willful damage, etc.; may require sophisticated testing techniques

Technique	Brief Description
State transition testing	Technique in which the states of a system are first identified and then test cases are written to test the triggers to cause a transition from one condition to another state
Statement coverage testing	Every statement in a program is executed at least once
Statistical profile testing	Statistical techniques are used to develop a usage profile of the system that helps define transaction paths, conditions, functions, and data tables
Stress testing	Term often used interchangeably with "load" and "performance" testing. Also used to describe such tests as system functional testing while under unusually heavy loads, heavy repetition of certain actions or inputs, input of large numerical values, or large complex queries to a database system
Structured walkthroughs	A technique for conducting a meeting at which project participants examine a work product for errors
Syntax testing	Data-driven technique to test combinations of input syntax
System testing	Black-box type testing that is based on overall requirements specifications; covers all combined parts of a system
Table testing	Testing access, security, and data integrity of table entries
Thread testing	Combining individual units into threads of functionality which together accomplish a function or set of functions
Top-down testing	Integrating modules or programs starting from the top
Unit testing	The most "micro" scale of testing; to test particular functions or code modules. Typically done by the programmer and not by testers, as it requires detailed knowledge of the internal program design and code. Not always easily done unless the application has a well-designed architecture with tight code; may require developing test driver modules or test harnesses
Usability testing	Testing for "user-friendliness." Clearly this is subjective, and will depend on the targeted end-user or customer. User interviews, surveys, video recording of user sessions, and other techniques can be used. Programmers and testers are usually not appropriate as usability testers
User acceptance testing	Determining if software is satisfactory to an end-user or customer
White-box testing	Test cases are defined by examining the logic paths of a system