UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Clasificación de Herramientas Open Source para Pruebas de Aplicaciones Web – Un caso de estudio
Autores: Rodríguez, Anahí Soledad – Soria, Valeria
Director: Lic. Díaz, Francisco Javier
Codirector: Lic. Banchoff T., Claudia M.
Asesor profesional:
Carrera: Licenciatura en Sistemas

Resumen

En esta tesina se destaca la importancia de realizar pruebas de software (o testing de software) que eviten el lanzamiento de sistemas con baja calidad. Si bien hay pruebas que deben ser realizadas en forma manual, existen numerosas herramientas, tanto privativas como de código abierto, que facilitan la ejecución de las mismas. El uso de estas herramientas no siempre garantiza el éxito dado que en muchos casos, el tiempo de aprendizaje de las mismas o la correcta selección implica un esfuerzo adicional que no en todos los proyectos puede ser tenido en cuenta.
Para la selección de herramientas se generó una plantilla con diferentes criterios de evaluación, la cual fue creada en base a cinco (5) métodos para evaluar la madurez. Se utilizó dicha plantilla para mostrar el uso de la misma, con diferentes herramientas para los distintos tipos de pruebas.
Luego, se muestra el uso de dos (2) herramientas para pruebas sobre un sistema Web (Kimkëlen: un sistema de gestión de alumnos para colegios secundarios). Se realizaron pruebas de unidad sobre el sistema y pruebas de rendimiento sobre el servidor.

Palabras Claves

Pruebas/Testing de Software – Software Libre – FLOSS – Licencias de Software – Aplicaciones Web – Modelo en V – Calidad – Bug – Error – Defecto – Falla – Caso de Prueba – CMM – Métodos de Evaluación de Madurez – Automatización de Pruebas – Herramientas para Pruebas de Software –Pruebas de Unidad – Pruebas de Rendimiento.

Trabajos Realizados

Se comenzó la tesina realizando una investigación sobre métodos de evaluación de madurez y en base a ellos se generó una plantilla propia para poder evaluar herramientas.
Se ejecutaron pruebas de unidad sobre la clase Student del sistema para las funciones más importantes.
Se ejecutaron diferentes pruebas de rendimiento sobre el servidor para poder observar el desempeño del mismo ante una sobrecarga de información.

Conclusiones

El primer paso es poder seleccionar una herramienta que se ajuste a las necesidades de cada proyecto en base a su madurez. Se planteó un caso de estudio concreto sobre un sistema Web desarrollado por el CeSPI. Se realizaron pruebas de unidad utilizando PHPUnit y pruebas de rendimiento usando JMeter. Como resultado de esta tesina, debemos destacar que hemos podido establecer un modelo que nos permite seleccionar las herramientas y un procedimiento para la ejecución de pruebas de unidad y de rendimiento.

Trabajos Futuros

Revisar periódicamente las planillas presentadas en el capítulo 7, y de ser necesario agregar nuevas herramientas, dado el gran movimiento de la comunidad de Software Libre.
Seguir probando el sistema con diferentes herramientas, aquellas que no han sido incluidas en esta tesina, y aplicarlas a los diferentes tipos de pruebas, y así poder realizar un plan de pruebas más completo.
Se podría extender la tesis agregando pruebas de usabilidad y accesibilidad, focos que no fueron tomados en este trabajo.

Fecha de la presentación: Mayo de 2012

Agradecimientos

A mi hija Melina

En primer lugar quiero agradecer al amor de mi vida que me acompañó, aguantó y ayudó durante gran parte de la carrera y a su familia (mi nueva y linda familia) que me brindo todo su amor.

A mi primer grupo de amigas Clau, Dani, Vale, Lili y Cari; que han sido muy importantes para mi vida durante la carrera.

A mis “nuevas” amigas que me acompañaron en todo momento.

A mis compañeros/as de trabajo, en especial al grupo soporte, lihuenes y a Car y su grupo de trabajo, que nos brindaron mucha ayuda para poder concretar este trabajo.

A mi AMIGA y compañera de tesina que sin su apoyo y amistad no podría haber terminado la carrera.

A mi director y codirectora de tesis que me dieron la oportunidad de crecer profesionalmente.

También un agradecimiento distinguido a mi amiga y diseñadora que nos ayudó a que este trabajo se vea lindo!!!

Y ni más ni menos a mis abuelos, mi mamá y mi hermano que fueron la guía, el apoyo y el sostén para poder realizar una carrera universitaria.

Anhi

Definitivamente escribir los agradecimientos es la parte más difícil.

¿Por dónde empezar?

Principalmente agradecer a mis padres y a mis hermanos, quienes estuvieron siempre para mí, algunos no en presencia física, pero si espiritual.

Obviamente a mi amiga y compañera, Anhi! Por elegirme para completar la última etapa de la carrera, por aguantarme tanto en el estudio como en el trabajo y por compartir conmigo todos estos años de amistad.

Quiero agradecerles a nuestros directores, Javier y Claudia, quienes nos guiaron y ayudaron a realizar este trabajo durante estos últimos años.

También a Ari, nuestra diseñadora amiga, que nos hizo todos gráficos, las tapas de la tesina y del CD.

A Christian y a su equipo de desarrollo, a los chicos de soporte y al grupo Lihuen.

Y por último pero no menos importante, a todas las personas que me acompañaron en cada paso, tanto de mi carrera como de mi vida personal: amigos de la vida, del trabajo, de la facu, a mi familia, conocidos... a todos ellos: GRACIAS!

Vale

Índice

Capítulo 1: Introducción y Motivación 7
1.1 Detalle 7
1.2 Objetivo 8
1.3 Estructura del informe 8
1.4 Referencias 9
Capítulo 2: Pruebas de Software en el proceso de desarrollo en Software Libre 11
2.1 Introducción 11
2.2 Modelos de desarrollo de software 11
2.3 Modelo de desarrollo de Software Libre 15
2.4 Licencias de software 16
2.4.1 Definición de la licencia GNU GPL (General Public License) 17
2.4.2 Licencias de software compatibles con GNU GPL 17
2.4.3 Licencias de software incompatibles con GNU GPL 19
2.4.4 Tabla comparativa 20
2.5 Resumen 21
2.6 Referencias 21
Capítulo 3: La etapa de pruebas de software 23
3.1 Introducción 23
3.2 Definiciones de la etapa de pruebas 23
3.2.1 Dificultades en el proceso de testing 26
3.3 Principios básicos en las tareas de testing 26
3.4 ¿Para qué sirve probar? 27
3.5 Calidad 30
3.6 Validación y verificación 32
3.7 Error, defecto y falla 32
3.8 Ambiente de pruebas 33
3.9 Resumen 34
3.10 Referencias 34
Capítulo 4: Pruebas de software 37
4.1 Introducción 37
4.2 Caso de prueba 38
4.2.1 Diseño del caso de prueba 38
4.3 Ciclo de las pruebas 39
4.4 Tipos de pruebas 39

4.4.1 Pruebas de unidad 39
4.4.2 Pruebas de Integración 40
4.4.3 Pruebas de Sistema 41
4.4.4 Pruebas de Aceptación 41
4.5 Técnicas de prueba 41
4.5.1 Estáticas 41
4.5.2 Funcionales 45
4.5.3 No Funcionales 47
4.5.4 Estructural 48
4.6 Resumen 49
4.7 Referencias 49
Capítulo 5: Criterios de evaluación de herramientas FLOSS para pruebas de software 51
5.1 Métodos para evaluar la madurez de los proyectos Open Source 51
5.1.1 OSMM (Open Source Maturity Model de Capgemini) 52
5.1.2 OSMM (Open Source Maturity Model de Navica) 52
5.1.3 BRR (Business Readiness Rating) 53
5.1.4 Qualification and Selection of Open Source Software (QSOS) 54
5.1.5 NASA's Reuse Readiness Levels (RRL) 55
5.2 Criterios de evaluación de herramientas: plantilla propuesta 58
5.3 Resumen 61
5.4 Referencias 61
Capítulo 6: Automatización de las pruebas y clasificación de herramientas 63
6.1 Automatización de las pruebas 63
6.2 Clasificación de herramientas de prueba según el Modelo en V 65
6.2.1 Clasificación de herramientas 66
6.3 Resumen 67
6.4 Referencias 67
Capítulo 7: Análisis de herramientas para pruebas de software 69
7.1 Introducción 69
7.2 Clasificación de las herramientas según la etapa 69
7.2.1 Herramientas para la gestión de pruebas 69
7.2.2 Herramientas para pruebas estáticas de código 73
7.2.3 Herramientas para pruebas de unidad 77
7.2.4 Herramientas para pruebas de rendimiento 81
7.2.5 Herramientas para pruebas funcionales 84
7.3 Resumen 91

7.4 Referencias....91
Capítulo 8: Caso de estudio: Sistema de Gestión de Alumnos....93
8.1 Introducción....93
8.2 Características de los servidores....96
8.2.1 Características del primer servidor:....96
8.2.2 Características del segundo servidor:....96
8.3 Un caso de estudio....96
Capítulo 9: Pruebas de Unidad....99
9.1 Introducción....99
9.2 Selección de una herramienta: PHPUnit....99
9.2.1 Instalación de PhpUnit....100
9.2.2 Instalación de sfPhpunitPlugin....100
9.2.3 Convenciones para la definición de las pruebas....101
9.2.4 Ejecución de las pruebas....102
9.2.5 Utilización de Fixtures....103
9.3 Caso de estudio de PhpUnit en el sistema Kimkëlen....104
9.3.1 Creación de las pruebas....105
9.3.2 Análisis de los resultados....108
9.4 Resumen....109
9.5 Referencias....109
Capítulo 10: Pruebas de Rendimiento....111
10.1 Introducción....111
10.2 Pruebas de rendimiento. Definición y Características....111
10.3 Selección de una herramienta: JMeter....112
10.3.1 Instalación de JMeter....113
10.4 Plan de pruebas - Definición y composición....113
10.4.1 Análisis de Resultados....116
10.4.2 Ejecución de las pruebas....117
10.5 Caso de estudio de JMeter en el sistema Kimkëlen....118
10.5.1 Prueba 1: Acceso a la página principal del sistema (“home”)....118
10.5.2 Prueba 2: Ingreso con login (con xml sin logout)....123
10.5.3 Prueba 3: Ingreso con login mejorado con logout (con csv con logout)....128
10.5.4 Prueba 4: Ingreso con login, listado y logout (csv)....135
10.5.5 Prueba 5: Ingreso con login, guardar asistencia y logout....138
10.6 Resumen....143
10.6 Referencias....143

Capítulo 11: Conclusiones 145
Capítulo 12: Trabajo a futuro 147
Índice de figuras 149
Índice de tablas 151

Capítulo 1: Introducción y Motivación

1.1 Detalle

Según Meyer: "*El testing de software es un proceso diseñado para verificar que el código desarrollado hace lo que está especificado en los requerimientos del usuario y no en funciones no deseadas. Al realizar pruebas a un software aumentamos la calidad y fiabilidad del programa. Con fiabilidad nos referimos a encontrar y eliminar errores. Por lo tanto, no se prueba un programa para demostrar que funciona, sino que debe comenzar con la suposición de que el programa contiene errores (una suposición válida para casi cualquier programa) y luego probar el programa para encontrar la mayor cantidad de errores posible. Por lo tanto, una definición más adecuada es la siguiente:* ***La prueba es el proceso de ejecución de un programa con la intención de encontrar errores***". [1]

Según Pressman: "*Las pruebas del software son un elemento crítico para la garantía de calidad del software y representa una revisión final de las especificaciones, del diseño y de la codificación.*" [2].

Todas las etapas en el proceso del desarrollo de software son sumamente relevantes, pero, quizás la etapa de pruebas sea la menos sistematizada y tenida en cuenta. Evidencia de ello, es la gran cantidad de parches y versiones surgidas luego del lanzamiento de una versión final de un software, destinadas a cubrir "agujeros de seguridad" o simplemente, malos funcionamientos. Por este motivo, es muy importante realizar pruebas de software (o testing de software, se usará indistintamente cualquiera de ambos términos) que eviten sistemas de baja calidad y aumenten la confianza de los usuarios.

Si bien, hay pruebas que deben ser realizadas en forma manual, existen numerosas herramientas, tanto privativas como de código abierto, que facilitan la ejecución de las mismas. El uso de estas herramientas no siempre garantiza el éxito dado que en muchos casos, el tiempo de aprendizaje de las mismas o la correcta selección implica un esfuerzo adicional que no en todos los proyectos puede ser tenido en cuenta. Muchas herramientas están focalizadas en un tipo de prueba específica y otras son más generales. Una correcta elección no siempre es una tarea sencilla.

La evolución de las Metodologías de Certificación de Calidad de Sofware tuvo un fuerte impacto en la necesidad de focalizar y formalizar las pruebas de software. El presente trabajo recopila diferentes herramientas y muestra su aplicabilidad.

La motivación de esta tesina nace de la necesidad de crear un equipo de testing para los grupos de desarrollo del Laboratorio de Investigación en Nuevas Tecnologías Informáticas (LINTI) al cual pertenecemos. Hemos realizado diferentes cursos y capacitaciones referidos al tema desde el año 2008. A partir del conocimiento adquirido hemos podido realizar pruebas sobre diferentes aplicaciones desarrolladas por estos equipos, utilizando distintas herramientas libres las cuales permiten automatizar las pruebas y, por ejemplo, generar repeticiones de una manera más simple, aportando mayor fiabilidad al software.

Es por esto que decidimos escribir esta tesina, para brindar nuestro conocimiento. Durante este período también hemos escrito varios artículos (citados en la misma) y varios de ellos han sido seleccionados para participar de diferentes congresos nacionales e internacionales.

1.2 Objetivo

Esta tesina está focalizada al análisis de herramientas Open Source para prueba de software. Ante la gran diversidad y variedad de las mismas, se analizarán modelos de madurez que nos ayuden a seleccionar de manera metodológica y objetiva. En este trabajo se recopila información (dispersa) de herramientas de testing, se agrupa y se unifica la terminología para describirlas a las mismas.

Para simplificar la consulta y el posterior uso del estudio realizado se construyó una planilla que sintetiza los principales criterios de evaluación de madurez.

Para ilustrar la validez del trabajo realizado se incluye una prueba de concepto sobre una aplicación Web.

1.3 Estructura del informe

En el capítulo 2 se describen los modelos de desarrollo del software y se hará una introducción a las licencias de uso del software.

En el capítulo 3 se muestra la etapa de pruebas de software, algunas definiciones de calidad, que es la validación y verificación, las diferencia entre error, defecto y falla, etc.

En el capítulo 4 se presenta lo que es un caso de prueba, cómo es su diseño y se introduce las definiciones de los distintos tipos de pruebas y las diferentes técnicas.

En el capítulo 5 se presentan diferentes métodos para medir la madurez en software libre y en base a estos se definió uno propio para poder realizar una selección de herramientas según diferentes criterios.

En el capítulo 6 se muestra la importancia de la automatización de las pruebas, y se realiza una clasificación de herramientas según el Modelo en V el cual fue descripto en el capítulo 2.

En el capítulo 7 se realiza la comparación de las diferentes herramientas para pruebas según la clasificación especificada en el capítulo 6 utilizando el método propuesto en el capítulo 5.

En el capítulo 8 se da una breve descripción del sistema Kimkëlen sobre el cual se van a realizar las pruebas y las características de los servidores donde se instaló el mismo.

En los capítulos 9 y 10 se muestra la aplicación de las herramientas seleccionadas para mostrar su uso y las cuestiones encontradas durante las pruebas.

Y finalmente en los capítulos 11 y 12 se presentaran las conclusiones de esta tesina y el trabajo a futuro respectivamente.

1.4 Referencias

[1] The Art of Software Testing, Glenford J. Meyer – Segunda edición.

[2] Ingeniería del Software: Un Enfoque Práctico, Roger Pressman – Quinta Edición.

Capítulo 2: Pruebas de Software en el proceso de desarrollo en Software Libre

2.1 Introducción

Existen distintos modelos para el desarrollo de software. La implementación basada en Software Libre (SL) posee, además, características distintivas basadas, principalmente en las formas de compartir el trabajo. En este sentido, es esencial el tipo de licencia con el cual se lo libera para su posterior utilización y/o adecuación.

En este capítulo se describirán tanto los distintos modelos para el desarrollo de software haciendo hincapié en el desarrollo de Software Libre como los distintos tipos de licencias compatibles con la licencia GNU GPL (creada por la Free Software Foundation para proteger la libre distribución, modificación y uso de software, más adelante se dará una descripción de la misma).

2.2 Modelos de desarrollo de software

Los modelos de desarrollo de software sirven para ordenar las tareas de ingeniería de software. Según Ian Sommerville: "*Un modelo de proceso de software es una descripción simplificada de un proceso del software que presenta una visión de ese proceso*" [1].

Existen diferentes modelos y cada uno organiza las distintas tareas involucradas de manera diferente según la política del mismo, pero tienen algunas actividades comunes:

- *Especificación del software*, es donde se define la funcionalidad y las restricciones del mismo.
- *Diseño e implementación del software*, es la etapa donde se realiza la producción del mismo cumpliendo con la especificación.
- *Validación del software*, donde se realizan las pruebas para asegurar que el software hace lo que tiene que hacer.
- *Evolución del software*, abarca las tareas de mantenimiento del software ante los cambios requeridos por el cliente.

Según Sommerville, la mayor parte de los modelos se software se basan en los siguientes paradigmas:

1. Cascada: Las actividades se representan en etapas separadas, cada una se ejecuta y una vez terminada se pasa a la siguiente.
2. Desarrollo iterativo: Se entrelazan las actividades de especificación, desarrollo y validación.
3. Basada en componentes: Esta técnica supone de la existencia de componentes reutilizables. Se basa en la integración de estas partes, más que desarrollarlas desde el principio.

Estos tres modelos son genéricos y muy utilizados en la ingeniería de software actual, no son mutuamente excluyentes y por lo general se utilizan juntos, especialmente para desarrollos de sistemas grandes.

Según Roger Pressman en su libro "*Ingeniería del Software: Un Enfoque Práctico*" [2], los modelos de desarrollo de software tratan de ordenar las tareas de los ingenieros de software, y todos los modelos exhiben características del "*Modelo del Caos*", el cual, según el autor Raccoon [3], define cuatro (4) etapas: status Quo, definición del problema, desarrollo técnico e integración de soluciones, estas etapas se ejecutan de forma iterada hasta la resolución del problema.

A continuación resumiremos otros modelos que describe Pressman:

1. Modelos de construcción de prototipos: En este modelo como primer paso, se recolectan requisitos definidos por el usuario. Así se define un diseño rápido de los requerimientos del usuario o cliente, que lleva a la construcción de un prototipo.
2. Modelo incremental: Es un modelo evolutivo, que se caracteriza por desarrollar versiones del software cada vez más completas, combina el modelo lineal secuencial con la filosofía de la construcción de prototipos.
3. Modelo en espiral: Es un modelo evolutivo, que une la manera iterativa de la construcción de prototipos con aspectos del modelo lineal secuencial. Las versiones se desarrollan de manera incremental.

Esta tesina se focaliza especialmente en las pruebas de software, por lo que si separamos las tareas correspondientes a dichas pruebas según el nivel de prueba que se pueden tratar, podemos identificar los siguientes niveles:

1. *Nivel de prueba de componente:* Se prueba cada componente por separado (Prueba de Unidad).
2. *Nivel de prueba de integración:* Se prueba la integración de cada uno de los componentes.
3. *Nivel de prueba de sistema:* se prueban todos los componentes integrados.
4. *Nivel de pruebas de aceptación:* se realizan pruebas ante un cliente o usuario.

Si vemos las tareas de pruebas separadas por los niveles antes mencionados, podemos ordenar las tareas de pruebas según el "*Modelo en V*" [4 y 5], el cual paraleliza las tareas de desarrollo con las tareas de pruebas. Este modelo no es más que una evolución del Modelo en cascada.

Existen muchas variaciones y definiciones del Modelo en V, el cual presenta un desarrollo "*top-down*" y una implementación y verificación "*bottom-up*", como se muestra en la Figura 1 [4]. La planificación de las pruebas es la tarea que une las dos aristas de la V, dado que se validan los requerimientos en los planes de prueba del sistema, se verifica el diseño en los planes de prueba de integración, y en el vértice de la V se realiza la planificación de las pruebas unitarias y de código [6].

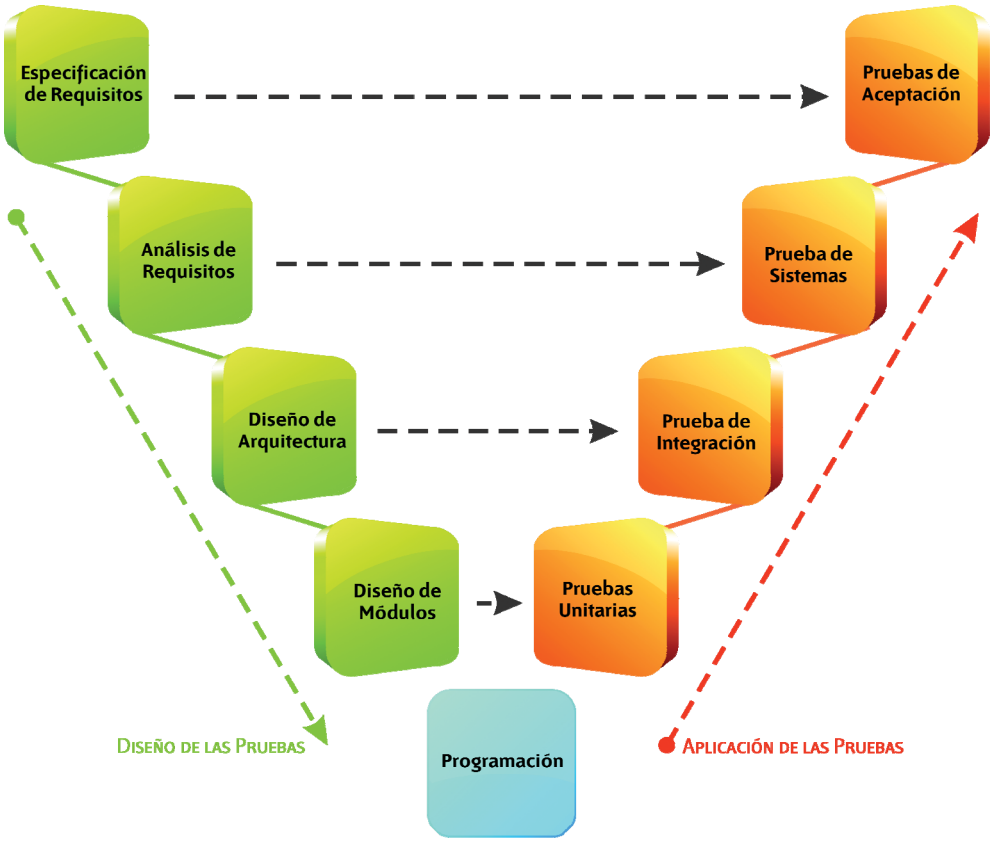


Figura 1: Modelo en V.

Este modelo posee algunas ventajas y desventajas de acuerdo a los aspectos analizados.

Algunas ventajas de este modelo:

- Es fácil de entender por su simplicidad.
- Es muy parecido al Modelo en cascada, es una evolución del mismo.
- Este modelo, junto al Modelo en cascada, es más ordenado (facilitando la gestión de proyectos) que los modelos iterativos que son desordenados e impredecibles.
- Permite realizar pruebas en etapas tempranas y a su vez mantiene ordenadas las actividades en el tiempo.
- Se adapta muy fácilmente a las necesidades de los desarrolladores y al equipo de pruebas.
- Permite validar y verificar cada etapa del desarrollo en el momento en que se está realizando. Es decir, se ejecuta la etapa y una vez finalizada se realiza la verificación (de la parte izquierda de la V) y la validación (pruebas) que es el nexo entre las dos aristas de la V.

Algunas desventajas de este modelo:

- No es posible realizar iteraciones para poder realizar cambios.
- El Modelo en V presenta falencias en el área de usabilidad, dado que la participación del usuario se da en etapas futuras y no se tiene en cuenta en las etapas de especificación, desarrollo e implementación. Esto puede generar problemas graves a la hora de la aceptación del cliente.

Nuestro objetivo es analizar herramientas Free/Open Source (de ahora en adelante FLOSS) para testing de software, para luego poder seleccionar y realizar una clasificación de las mismas. Para ello nos centraremos en un marco de trabajo basado en el Modelo en V. Aunque las herramientas que sugerimos pueden ser utilizadas en cualquier modelo de desarrollo (cambiando principalmente el momento en donde se aplica cada una de ellas).

2.3 Modelo de desarrollo de Software Libre

El modelo de desarrollo en Software Libre tiene algunas particularidades que lo hace atípico en referencia a los desarrollos tradicionales. Comencemos por clarificar la definición de Software Libre, que según la "Free Software Foundation - FSF" [7] y el proyecto GNU [8], dice lo siguiente:

"*El software libre es una cuestión de la libertad de los usuarios de ejecutar, copiar, distribuir, estudiar, cambiar y mejorar el software. Más precisamente, significa que los usuarios de programas tienen las cuatro libertades esenciales:*

- *La libertad de **ejecutar** el programa, para cualquier propósito (libertad 0).*
- *La libertad de **estudiar** cómo trabaja el programa, y **cambiarlo** para que haga lo que usted quiera (libertad 1). El acceso al código fuente es una condición necesaria para ello.*
- *La libertad de **redistribuir copias** para que pueda ayudar al prójimo (libertad 2).*
- *La libertad de **distribuir copias de sus versiones modificadas** a terceros (libertad 3). Si lo hace, puede dar a toda la comunidad una oportunidad de beneficiarse de sus cambios. El acceso al código fuente es una condición necesaria para ello.*" [9].

Hacia el año 1998 el movimiento del software libre se separó en dos ramas: una más orientada a la "definición conceptual y filosófica" (Free Software) y otra, un poco más práctica (Open Source). Richard Stallman, en su artículo "*Por qué el código abierto pierde el punto de vista del software libre*" [10] destaca, entre sus diferencias principales, que "*El código abierto es una metodología de programación, el software libre es un movimiento social"*.

Para evitar estas diferencias, se suele hablar también de FLOSS ("Free/Libre/Open Source Software") para designar este tipo de software sin entrar en las discusiones planteadas anteriormente [11].

El ensayo "*La catedral y el bazar*" de Eric S. Raymond [12], presentó un hito en el análisis de este tipo de software. Raymond analiza dos modelos de producción de software:

- La catedral, considerado un modelo de desarrollo vertical y hermético, generalmente utilizado para el desarrollo de software propietario.
- El bazar, donde se da un desarrollo horizontal, colaborativo y comunitario.

El modelo de desarrollo en Software Libre sigue una estructura colaborativa donde los usuarios pueden jugar un rol de beta-testers informando los errores encontrados o dando sugerencias y opiniones durante el desarrollo y mantenimiento del producto, de manera tal que es mucho más fácil mejorar el código. Por la naturaleza que caracteriza al Software Libre se liberan versiones con frecuencia, y así se puede ir "*escuchando*" las necesidades de los usuarios, para una mejora continua.

Por las características antes definidas, se eligió este modelo para la selección de herramientas. Los proyectos de software en general son diferentes entre si, así como los aspectos en los que es necesario enfocar las pruebas, es por esto que seleccionar una herramienta FLOSS nos permite adaptarla a las necesidades del equipo de pruebas.

2.4 Licencias de software

Como en esta tesina se analizarán herramientas FLOSS, y una característica esencial a la hora de elegir una herramienta es la licencia por la cual fue compartida, veremos en esta sección, que existen muchos tipos de licencias que aplican a software libre, las cuales difieren en algunas cuestiones que debemos tener en mente a la hora de optar por una u otra.

Las licencias de uso determinan la forma en que una obra será utilizada, en nuestro caso un software. En la página oficial del proyecto GNU [13], podemos encontrar una definición más formal: "*Una Licencia de Software es la autorización o permiso concedida por el autor para utilizar su obra de una forma convenida habiendo marcado unos límites y derechos respecto a su uso*."

La forma más sencilla y conocida en la comunidad FLOSS para indicar que un programa o aplicación es Software Libre es atribuirle una licencia "*Copyleft*", la cual exige que todas las versiones modificadas y extendidas del mismo sean también libres.

2.4.1 Definición de la licencia GNU GPL (General Public License)

La Licencia Pública General de GNU es utilizada por la mayoría de los programas de GNU y más de la mitad de los paquetes de Software Libre. La última versión es la 3, con fecha 29/06/2007.

GNU GPL es una licencia copyleft y libre para software y otros tipos de obras. Pretende garantizar la libertad de compartir y modificar todas las versiones de un programa, para asegurar que sigue siendo software libre para todos sus usuarios.

Está diseñada para asegurar que se tiene la libertad de distribuir copias de software libre (y cobrar por ellas si lo desea), que se pueda conseguir el código fuente, cambiar el software o piezas del mismo y crear nuevos programas libres [14].

Dentro de la comunidad FLOSS surgieron diferentes propuestas que introducen variantes a esta postura dando lugar a diversas tipos de licencias. Sin entrar en demasiados detalles, dado que no es la finalidad de esta tesina profundizar en este tópico, describiremos las más destacadas, entre las que se encuentran [15]:

- Licencias de software libre compatibles con GNU GPL.
- Licencias de software libre incompatibles con GNU GPL.

2.4.2 Licencias de software compatibles con GNU GPL

A la fecha (Marzo 2012) existen diferentes licencias que son compatibles con la licencia GNU GPL. Entre ellas mencionaremos algunas de las más conocidas y utilizadas.

GNU LGPL (Lesser LGL)

La Licencia Pública General Reducida de GNU se utiliza en algunas de las bibliotecas GNU. La última versión es la 3, con fecha 29/06/2007.

Esta versión incorpora los términos y condiciones de la licencia GNU GPL versión 3, complementado por permisos adicionales para las bibliotecas de software.

GNU LGPL es menos restrictiva que GNU GPL, ya que no tiene Copyleft fuerte, permitiendo que el software se enlace con módulos no libres. Aunque la GNU LGPL no protege tanto las libertades del usuario, asegura que el usuario de un programa, que esté enlazado con la biblioteca, tenga la libertad y los medios para ejecutar ese programa usando una versión modificada de la biblioteca [16].

BSD

Es una licencia de software libre permisiva, creada por la Universidad de California, que principalmente tienen los sistemas BSD (*Berkeley Software Distribution*).

Permite la redistribución, uso y modificación del software. Esta licencia no tiene Copyleft, ya que permite cambiar la licencia del software [17].

Apache

La licencia de software Apache ("ASL") es una licencia de software libre creada por la Apache Software Foundation. La última versión es la 2.0, con fecha Enero de 2004 y es compatible con la versión 3 de GNU GPL.

Permite el uso del software para cualquier propósito, distribuirlo, modificarlo, y distribuir versiones modificadas del mismo. En su licencia se indica que se prohíbe el uso de la marca de ASL pueda afirmar o implicar que la Fundación apoya a la distribución modificada. Y exige que cualquier distribución que incluya el software de Apache tenga una clara atribución a ASL.

ASL no tiene Copyleft, al igual que la licencia BSD, y las versiones anteriores de esta licencia no son compatibles con GNU GPL [18].

Artistic License

Es una licencia para software Free/Open Source. Su última versión es la 2.0.

Esta licencia establece los términos en que un determinado paquete de software libre se puede copiar, modificar, distribuir, y/o redistribuir. La intención es que el autor mantenga el control artístico sobre el desarrollo de ese paquete, mientras el paquete se mantiene disponible como software Free/Open Source.

Esta licencia no tiene Copyleft (implica que algunas copias o versiones modificadas del mismo pueden no ser completamente libres) y sus versiones anteriores no son compatibles con GNU GPL [19].

W3C Software License

Se encuentra activa desde el 31/12/2002. La misma da permiso para copiar, modificar y distribuir el software y la documentación de la W3C que tenga vinculada esta licencia.

Esta versión elimina el copyright de tal manera que se puede utilizar con otros materiales que de la W3C, y elimina la ambigüedad concesión de "*uso*". Como en el caso de las licencias anteriores, tampoco tiene Copyleft [20].

MIT License

La licencia MIT fue creada el Instituto Tecnológico de Massachusetts. Es una licencia de Software Libre simple y permisiva sin Copyleft.

Permite reutilizar el software tanto para Software Libre como para software no libre, permitiendo no liberar los cambios realizados al programa original.

También permite licenciar dichos cambios con licencia BSD, GNU GPL, u otra cualquiera que sea compatible [21].

MPL

La Licencia Pública de Mozilla (MPL) es una licencia para software Free/Open Source. La versión actual es la 2.0.

La sección 3.3 proporciona una compatibilidad indirecta entre esta licencia y la versión 2.0 de la GNU GPL, la versión 2.1 de la GNU LGPL, la versión 3.0 de la AGPL de GNU y todas las versiones posteriores de estas licencias. La sección 3.3 autoriza a distribuir el trabajo cubierto por la MPL bajo los términos de dichas licencias de GNU, con una condición: debe asegurarse que los archivos, que originalmente estaban disponibles bajo la MPL, sigan estando disponibles bajos los términos de la MPL. El resultado final estará cubierto por la/s licencia/s de GNU. Los que reciban la combinación tendrán la opción de usar los ficheros, originalmente cubiertos por la MPL, bajo los términos de la MPL o distribuir el trabajo, en el todo, bajo las condiciones de una de las licencias de GNU sin más restricciones [22].

2.4.3 Licencias de software incompatibles con GNU GPL

CPL (Common Public License)

CPL es una licencia para software Free/Open Source publicado por IBM aprobada por la “Free Software Foundation” y la “Open Source Initiative”.

IPL (IBM Public License) fue la primera licencia de código abierto de IBM. CPL es la siguiente versión de IPL. CPL fue escrita para generalizar los términos de uso de IPL para que cualquier autor de código abierto pueda utilizar los términos que se encuentran en IPL.

La versión 1.0 fue la última versión de CPL y la versión inicial utilizada por el proyecto Eclipse[1]. Desde entonces, CPL ha sido sustituida por la Licencia Pública Eclipse (EPL).

[1] *Eclipse* es un proyecto de código abierto enfocado en una plataforma de desarrollo para la integración de herramientas de desarrollo de aplicaciones

CPL tiene Copyleft y tiene algunos términos similares a los de GNU GPL, pero también existen algunas diferencias. Una similitud es la distribución de un programa modificado: el código fuente de un programa modificado debe estar a disposición de otros. Pero ya que tiene un copyleft débil y además tiene una cláusula de ley del Estado de Nueva York, esto hace que sea incompatible con la GNU GPL [23, 24 y 25].

Python License (CNRI Python License)

Es una licencia de Software Libre no compatible con GNU GPL, ya que la principal incompatibilidad es que está sujeta a las leyes del estado de Virginia, en los EE.UU., y la GNU GPL no lo permite [26 y 27].

2.4.4 Tabla comparativa

Como se ha visto, existen muchas formas distintas de compartir y publicar software libre. Si bien en la sección anterior hemos mencionado sólo algunas de estas posibilidades, como se ha podido ver, existen muchas similitudes entre ellas.

A continuación se presenta una tabla comparativa (Tabla 1) en la cual se sintetizan algunas características de las licencias nombradas anteriormente:

Licencia	Ultima Versión	Copyleft	Compatible GNU GPL
Apache	2.0	✗	✓
Artistic License	2.0	✗	✓
BSD	S/D	✗	✓
CPL (Common Public License)	1.0	✓	✗
GNU GPL	V3	✓	✓
GNU LGPL (Lesser LGL)	V3	✓	✓
MPL	2.0	✓	✓
MIT license	S/D	✗	✓
Python License (CNRI Python License)	S/D	✓	✗
W3C Software License	S/D	✗	✓

Tabla 1: Comparación de licencias.

2.5 Resumen

En este capítulo hemos mencionado los distintos tipos de desarrollo de software, haciendo especial hincapié en el modelo de desarrollo de Software Libre. En este sentido es muy importante la forma en el que el mismo es liberado, dado que la licencia que se le aplique podrá limitar o no, su posterior uso o incorporación a otros proyectos.

La heterogeneidad de la comunidad de Software Libre se evidencia en la cantidad de formas distintas que hemos encontrado para compartir los productos. Se han tomado las licencias más conocidas para mencionar o destacar algunas de sus características y las implicancias que las mismas pueden llegar a tener en caso de ser utilizadas.

2.6 Referencias

[1] Ingeniería de Software, Ian Sommerville – Séptima Edición.

[2] Ingeniería del Software: Un Enfoque Práctico, Roger Pressman – Quinta Edición.

[3] The Chaos Model and the Chaos Life Cycle, L. B. S. Raccoon.

[4] Phase I - Systems Engineering Management Plan: A Process Review and Appraisal of the Systems Engineering Capability for the Florida Department of Transportation (FDOT) - Version 2.

[5] ISTQB, ISEB Lecture Notes- 2 Software Testing Foundation Level Certification Chapter-2.

[6] Ciclos de Vida de Ingeniería del Software, Richard Rojas e Israel Boucchechter.

[7] http://www.fsf.org

[8] http://www.gnu.org/

[9] http://www.gnu.org/philosophy/free-sw.es.html

[10] http://www.gnu.org/philosophy/open-source-misses-the-point.es.html

[11] Aprendiendo Tecnología con Software Libre – Capítulo 1, Javier Díaz, Claudia Banchoff, Viviana Harari, Christian Rodríguez, Einar Lanfranco, Mauricio Foster, Alejandra Osorio, Paola Amadeo, Fernando López, Sofía Martin, Anahí Rodríguez, Valeria Soria – Edición: Nacional – INET. Lugar: La Plata, Pcia. de Buenos Aires.

[12] http://www.catb.org/~esr/writings/cathedral-bazaar/

[13] http://www.gnu.org/philosophy/free-sw.es.html

[14] http://www.gnu.org/licenses/gpl.html

[15] http://www.gnu.org/licenses/license-list.es.html

[16] http://www.gnu.org/copyleft/lgpl.html

[17] http://www.opensource.org/licenses/bsd-license.php

[18] http://www.apache.org/licenses/LICENSE-2.0

[19] http://www.opensource.org/licenses/artistic-license-2.0

[20] http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231

[21] http://www.gnu.org/licenses/license-list.html

[22] http://www.mozilla.org/MPL/2.0/

[23] http://www.opensource.org/licenses/cpl1.0

[24] http://www.ibm.com/developerworks/library/os-cplfaq.html

[25] http://www.gnu.org/licenses/license-list.html#CommonPublicLicense10

[26] http://www.gnu.org/licenses/license-list.html

[27] http://www.opensource.org/licenses/pythonpl

Capítulo 3: La etapa de pruebas de software

3.1 Introducción

Producir software de alta calidad implica desarrollar sistemas que cumplan con la totalidad de las necesidades especificadas, o implícitas, establecidas por el usuario. El testing mejora la calidad del producto e interviene en cada etapa del desarrollo de software, desde el análisis de requisitos a la codificación y pruebas, incluyendo el diseño, la documentación y el mantenimiento. Cada una de estas etapas debe estar acompañada por los distintos tipos de pruebas (aceptación, sistema, integración, etc.).

En este capítulo se dará una descripción de las tareas del área de testing, presentando a su vez los distintos conocimientos sobre los aspectos más relevantes de dicha etapa.

3.2 Definiciones de la etapa de pruebas

Existen muchas definiciones de testing de software. A continuación mostraremos las definiciones de distintos autores:

- Según Glenford J. Meyer, en su libro, "The Art of Software Testing" [1]: "*El testing de software es un proceso diseñado para verificar que el código desarrollado hace lo que está especificado en los requerimientos del usuario y no en funciones no deseadas. Al realizar pruebas a un software aumentamos la calidad y fiabilidad del programa. Con fiabilidad nos referimos a encontrar y eliminar errores. Por lo tanto, no se prueba un programa para demostrar que funciona, sino que debe comenzar con la suposición de que el programa contiene errores (una suposición válida para casi cualquier programa) y luego probar el programa para encontrar la mayor cantidad de errores posible. Por lo tanto, una definición más adecuada es la siguiente:* ***La prueba es el proceso de ejecución de un programa con la intención de encontrar errores***".
- La IEEE, Standard for Sotware Test Documentation, 1983, define: "*El testing de software es el proceso de detectar diferencias (errores) entre el comportamiento esperado y el comportamiento actual del sistema*".
- Bill Hetzel [2] define las tareas de testing como los mecanismos para: "*establecer fehacientemente que un programa hace lo que se supone que tiene que hacer*".

- Edward Kit [3] define al testing como: *"una disciplina profesional que requiere personas capacitadas y entrenadas*".
- La ANSI – 1983 – std 829, define a las tareas de testing como: "*el proceso de analizar un ítem de software para detectar la diferencia entre las condiciones requeridas (errores) y evaluar las características de los ítems del software*".
- Luego la ANSI – 1990 – std 610.12, define a las tareas de testing como: "*el proceso de ejecutar un sistema o componente bajo condiciones especificas observando o registrando sus resultados y realizando una evaluación de algún aspecto del sistema o componente*".

Podemos decir entonces que:

Probar un software es someterlo a ciertas condiciones que puedan demostrar si es válido o no con respecto a los requerimientos planteados.

Siempre se deben definir las pruebas asumiendo que el software tiene errores, verificando que se ajustan a los requerimientos establecidos y validar que las funciones se implementen correctamente. Realizando pruebas al software no sólo agregamos valor al producto si no que también al proceso de desarrollo del mismo, considerando los resultados generados.

Si las pruebas detectan errores se dice que son pruebas exitosas.

El proceso de testing debe comenzar junto con la etapa de definición de requerimientos del software y debe acompañar todo el proceso de desarrollo del mismo. De esta manera podemos evitar que sistemas de baja calidad lleguen a los usuarios finales. Comenzando la tarea de testing desde la definición de requerimientos, podemos mejorar los costos de resolución de errores. La Figura 2 [4] nos muestra los beneficios de la corrección de errores en etapas tempranas:

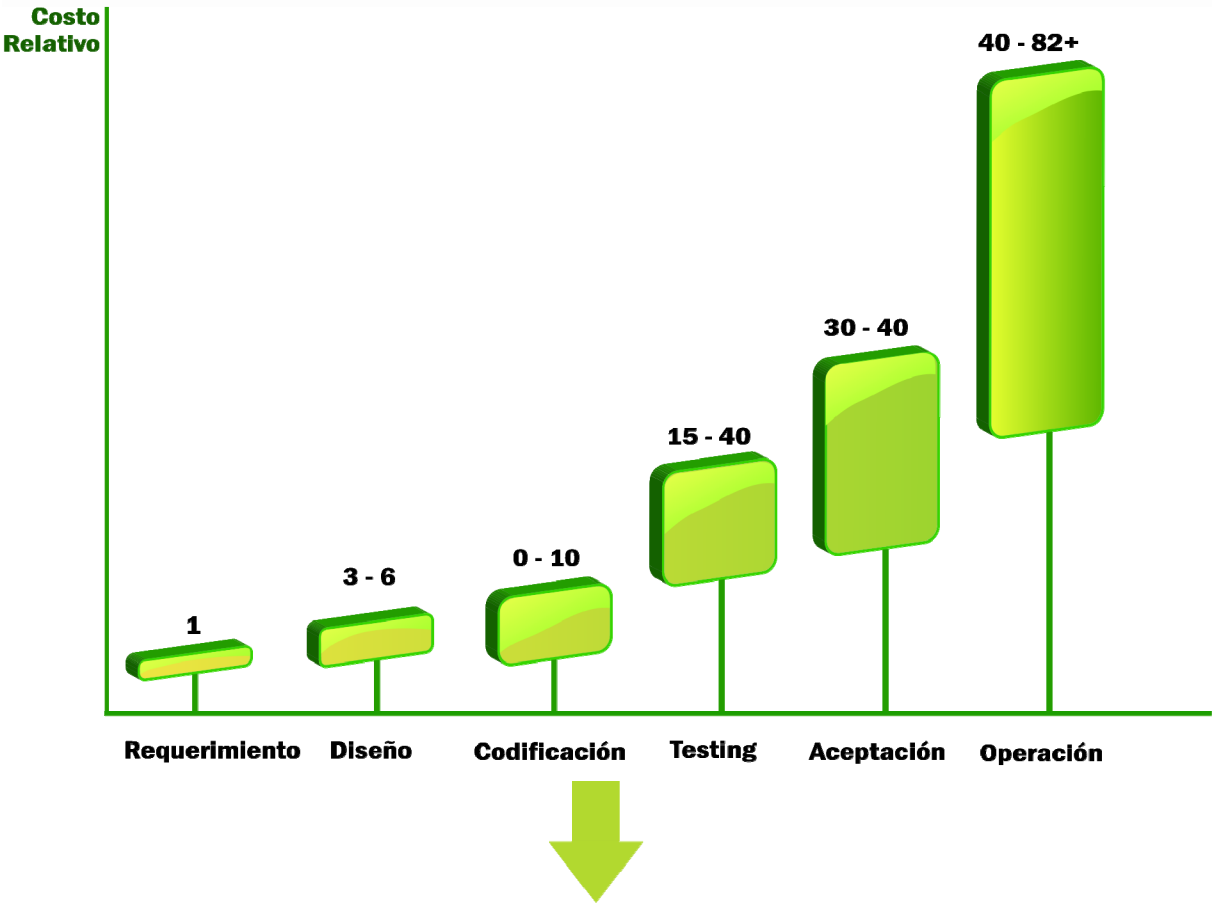


Figura 2: Costos relativos de resolución según la etapa en la que se encuentre el error.

Algo muy común es confundir las tareas de testing con las de debbuging. Éste último es un proceso que se utiliza para encontrar, analizar, evaluar la corrección y eliminación de las fallas en el software y, como vimos anteriormente, las pruebas se encargan de detectar los errores.

Las personas encargadas de realizar las pruebas deben ser ajenas al desarrollo del software, ya que deben estar comprometidas con la búsqueda de errores y no deben defender el producto. Los desarrolladores, por la naturaleza del ser humano, tienden a defender el desarrollo realizado.

Se puede decir que:

El testeo de software es comparable, en su esencia al trabajo del editor para un escritor de libros, pues quien desarrolla esta inmerso en el proyecto y necesita de una vista objetiva para perfeccionar su producto.

3.2.1 Dificultades en el proceso de testing

En la actualidad, el desarrollo de software se ha transformado en una actividad que requiere de cronogramas ajustados. Esto suele lleva a realizar diseños incompletos, pruebas ineficientes, productos de mala calidad y de alto costo en mantenimiento, etc.

Por todas estas razones se hace indispensable la realización de una etapa de prueba, e invertir cada vez más en esta actividad. Las pruebas llegan generalmente demasiado tarde, haciendo que la tarea de testing se realice al final de proyecto, con poco tiempo y con pocos recursos. Por lo que en estos casos sucede que el equipo encargado de estas tareas no puede intervenir en decisiones ya tomadas. Por ejemplo, si el testeador interviene en la etapa de análisis de requerimientos, puede aportar sus conocimientos, para desambiguar requisitos, así como poder organizar los tiempos, de manera de que la etapa de prueba no sea acotada entre que se termina de codificar y la fecha de entrega.

Las tareas de testing pueden comenzar antes de recibir el código o el sistema terminado, dado que se pueden comenzar a diseñar los casos de prueba a partir de la documentación de la especificación.

3.3 Principios básicos en las tareas de testing

Según la ISTQB [5] (International Software Testing Qualification Board, norma superior que certifica el conocimiento de los profesionales sobre las estrategias, métodos y practicas integrales que intervienen en el testing de Software de alto nivel), se han definido siete (7) principios que describen directrices generales que surgen en las tareas de pruebas.

Principio 1: Las pruebas ponen en evidencia defectos. Al realizarse un plan de pruebas a un software determinado, se reduce la cantidad de defectos que el mismo tendrá.

Principio 2: Las pruebas exhaustivas son imposibles. No se pueden realizar pruebas de **todas** las posibles combinaciones de entradas. Se debe priorizar y realizar un análisis de riesgo para poder seleccionar los **casos de prueba** a ejecutar. Según el autor Glenford J. Meyer [1]: "*Un conjunto de pruebas 'ideal', sería poner a prueba todas las permutaciones posibles de un programa pero en la mayoría de los casos, sin embargo, esto no es posible. Incluso un programa aparentemente simple puede tener cientos o miles de de combinaciones de entrada y de salida. Crear casos de prueba para todas las posibilidades no es práctico. Realizar todas las pruebas posibles en una aplicación compleja, sería una tarea muy extensa y requiere muchos recursos humanos y no llega a ser viable económicamente*".

Principio 3: Probar en fases tempranas. Es recomendable comenzar las tareas de testing al inicio del desarrollo del software, realizando pruebas estáticas de los requerimientos, diseño y código. Los errores encontrados en los requerimientos pueden costar solucionarlos en la etapa de implementación mucho más que si se arreglan en la etapa de la especificación [4]. La corrección de defectos en los requerimientos puede tener un alto costo si son detectados en la implementación.

Principio 4: Defectos agrupados. Algunos módulos son los que contienen la mayoría de los defectos. Realizar pruebas a estos módulos sería una forma de aumentar la eficiencia de las pruebas y acotar los casos de prueba.

Principio 5: Paradoja "del pesticida". Realizar siempre las mismas pruebas, puede llegar a ocultar algunos defectos y las pruebas pierden eficacia. Por ese motivo los casos de prueba necesitan revisarse regularmente y se deben ir añadiendo nuevos casos para probar las diferentes partes del software.

Principio 6: Las pruebas son dependientes del contexto. Las pruebas se deben definir según el contexto, dado que sistemas similares (en apariencia) pueden tener objetivos muy diferentes y requisitos muy diferentes.

Principio 7: Falacia de la ausencia de errores. Es importante que el software realice lo que se establece en la definición de los requerimientos del usuario, y además que no contenga errores. Comenzar las pruebas asumiendo que el software no tiene errores, es un error, dado que las pruebas se deben comenzar pensando que el software si contiene errores.

3.4 ¿Para qué sirve probar?

Es necesario realizar pruebas dado que así podemos demostrar que el software funciona correctamente, reduciendo costos, tiempos involucrados y riesgos. Las fallas en sistemas críticos pueden traer consecuencias graves que van desde pérdidas monetarias, y repercusiones económicas a las pérdidas de vidas humanas. A continuación de mostrarán algunas situaciones reales que ejemplifican lo detallado.

- *División en coma flotante en Intel Pentium (1994):* Thomas Nicely descubrió que la operación de división devolvía siempre un valor erróneo, el problema era que se redondeaban mal los dígitos decimales en determinadas divisiones en coma flotante [6].
- *Desintegración del Ariane 5 (1996):* El Ariane 5 era un cohete que a causa de una serie de errores, se desintegró a los 39 segundos de haber sido lanzado, dado

que se reutilizaron partes del software del Ariane 4 que no fueron correctamente probados para el nuevo cohete [6].

- *Bugs en software bancario (1996):* Por error se le acreditaron más de 924 millones de dólares a 823 clientes de un importante banco de EEUU (First National Bank of Chicago) a causa de un cambio de software [6].
- *Error en equipo de Cisco (1998)*: un error en un equipo de ruteo (“switch”) de Cisco en uso por AT&T se propagó por cientos de equipos de ruteo en su red de alta velocidad, dejando fuera de servicio miles de cajeros automáticos y lectores de tarjetas de crédito [6].
- *Error del milenio (2000):* el error Y2K se remonta a la década del '60, cuando los programadores adoptaron la convención de representar el año con dos (2) dígitos (al cual se le agregaba adelante el número 19) en lugar de cuatro (4) dígitos. Otro problema no solo para la generación de las fechas, sino también el cálculo de años bisiestos [6].
- *Fallo de seguridad en las cámaras IP de TRENDnet (2012):* accediendo con una dirección IP, cualquier usuario puede acceder, sin necesidad de ingresar con un nombre de usuario y contraseña, a visualizar en tiempo real lo que están captando dichas cámaras. En algunos casos dichos enlaces iban acompañados de una localización en Google Maps. TRENDnet (cuyo eslogan es "Redes en las que las personas confían") ha puesto a disposición de los usuarios "actualizaciones críticas" para "mejorar la seguridad". [7].

Existen varios ejemplos que ilustran el problema de no tener un plan de pruebas adecuado, como se muestran en las siguientes imágenes:

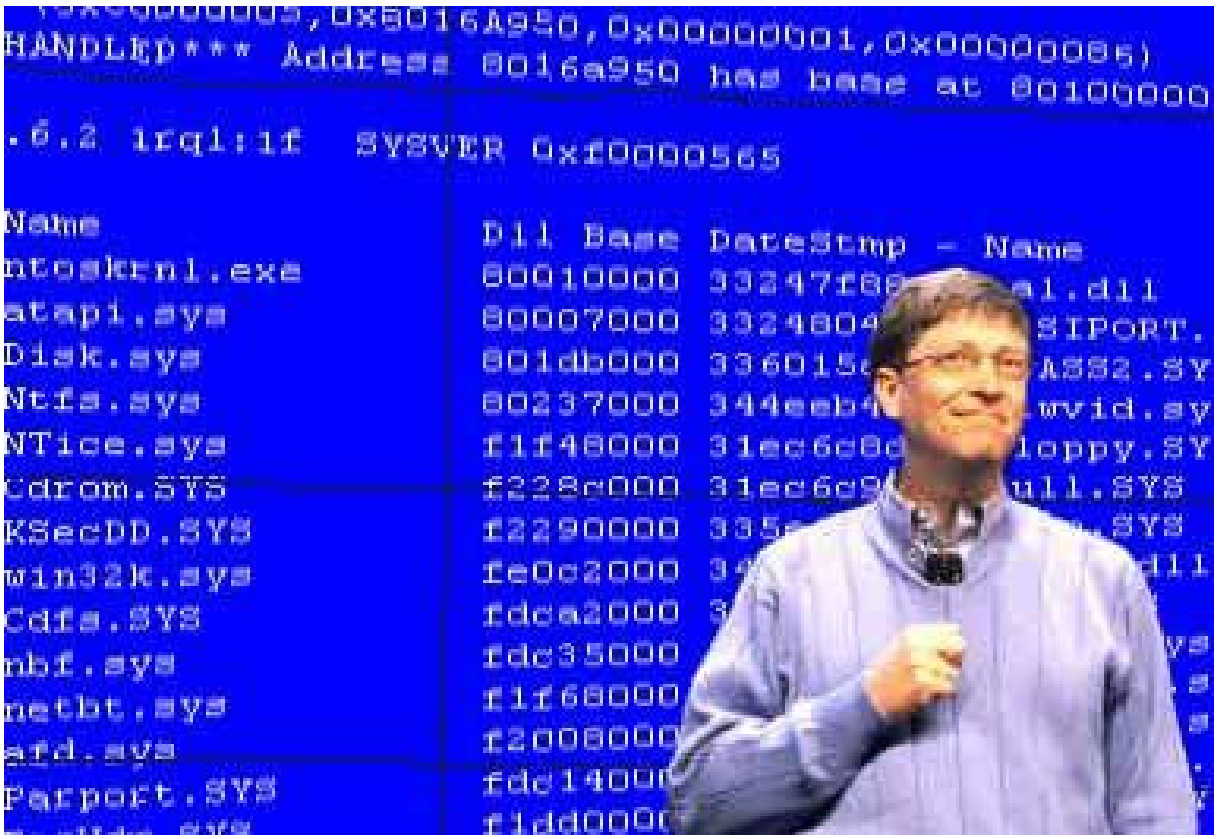


Figura 3: BSoD (pantallazo azul de la muerte) presentación Windows 98.

Figura 4: BSoD en la inauguración de los Juegos Olímpicos de verano de Beijín 2008.

Figura 5: BSoD en cartel de publicidad.

Como puede verse, algunas de estas situaciones son de extremada gravedad, con consecuencias tan graves como daños a la salud humana, pero en todas las situaciones planteadas, la falta de pruebas adecuadas puede terminar en pérdidas intangibles, como la pérdida de credibilidad de las empresas ante sus usuarios, dentro de la empresa puede bajar la confianza y la imagen del desarrollador del software, etc.

3.5 Calidad

La calidad del software no tiene una sola definición, si no que es un concepto multidimensional. Es un concepto complejo que puede describirse desde distintas perspectivas. Por ejemplo, vemos como distintos autores definen calidad:

- Juran [8]: definió la calidad como "*la adaptabilidad al uso*".
- Crosby [9]: definió la calidad como "*el asentimiento con los requerimientos*".
- Feigenbam [10]: definió la calidad como "*cumplimiento con los requerimientos de una persona*".

La ISO y la IEEE definen la calidad como:

Según la ISO 8402-1986 [11 y 12], la calidad es la totalidad de aspectos y características de un producto o servicio que se sustentan en su capacidad de cumplir las necesidades especificadas o implícitas.

La calidad del software hace referencia al proceso de desarrollo de un software que una empresa cumple en función a algún modelo o estándar requerido, o bien por el cliente o bien por la organización misma.

Este concepto de calidad de software es un concepto muy distinto al de cualquier producto o servicio del sector industrial, ya que el software según la IEEE std. 610 es "*el conjunto de programas de ordenador, los procedimientos y posiblemente, la documentación asociada y los datos relativos a la operación del sistema informático*", y tiene unas características muy especiales ya que se desarrolla y no se fabrica; es un producto lógico y no físico; y no se degrada con el uso.

El concepto de calidad de software no es tan sencillo como parece, pues cuidar de la calidad del software significa cuidar todos y cada uno de los elementos enumerados en la definición de software y no sólo el código fuente.

Según la IEEE std. 610-1991 [13] la calidad es: "*El grado con el que un sistema, componente o proceso cumple los requisitos especificados y cubre las necesidades o expectativas del cliente o el usuario*".

La Calidad de Software no se certifica, lo que se certifica son los procedimientos para construir un software de calidad, los procedimientos deben ser correctos y estar en función de la Normalización como por ejemplo la ISO 9000 [14].

Aumentar la calidad del software es una tarea muy importante en la cual interviene el equipo de testing. Para darle más calidad al producto, el mismo debe cumplir con la totalidad de las necesidades especificadas (tanto implícitas como explícitas), establecidas por el usuario. Además de que el producto llegue con la menor cantidad de errores posibles al usuario final [15].

3.6 Validación y verificación

La verificación y la validación son dos términos que están muy relacionados con la calidad del software, cada uno de ellos significa cosas diferentes.

La verificación y la validación proporcionan una **evaluación objetiva** para el software durante todo el ciclo de vida. Esta evaluación permite definir si el software es correcto, completo, exacto, coherente y comprobable.

Así, las mejoras continuas y en cada una de las etapas llevan a tener bajos costos en la resolución de los errores [16].

Según Pressman [17] la ***verificación*** se refiere al conjunto de actividades que aseguran que el software implementa correctamente una función específica y la ***validación*** se refiere a un conjunto diferente de actividades que aseguran que el software construido se ajusta a los requisitos del cliente.

Bohem [18] define estas dos palabras con preguntas tales como:

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

Otra distinción entre estos términos es el momento en el cual los podemos aplicar. Por ejemplo, la verificación puede realizarse al final de cada fase del desarrollo de software, verificando que cada uno de los componentes reúna las condiciones fijadas en el comienzo de esa fase. La validación tiene lugar luego que la verificación ha sido completada. Generalmente se realiza con la prueba de aceptación del cliente.

3.7 Error, defecto y falla

A la hora de detectar un fallo en el software es importante diferenciar entre error, falla y defecto. La IEEE definió un sistema de transición de estados modelando múltiples niveles de confiabilidad, y un software puede comenzar en cualquiera de estos estados, como se muestra en la Figura 6 [19]:

Figura 6: Transición de estados.

Si bien en muchos casos los términos anteriores se toman como sinónimos, no lo son.

A continuación se definen tres términos importantes involucrados en el diagrama anterior:

- *Falla:* Es la diferencia entre los resultados esperados y los reales, el software no se comporta adecuadamente. Esta propiedad del programa se manifiesta en ejecución.
- *Defecto:* existe en el código fuente y esto puede causar una falla. Ejemplo: utilizar el símbolo > en lugar de < en una comparación numérica.
- *Error:* es una equivocación humana, provoca una o varias fallas. Ejemplo: malinterpretar un requerimiento de un usuario.

Relacionando estos términos, podemos decir que:

Un ***error*** genera uno o más ***defectos*** y estos generan cero, una o más ***fallas***.

3.8 Ambiente de pruebas

Un punto muy importante en el momento de establecer las distintas pruebas, es el ambiente en el cual se realizarán las mismas. El ambiente donde se ejecutarán las pruebas debe ser independiente del resto de las áreas, o sea independiente del entorno de desarrollo y del entorno de producción, garantizando independencia y estabilidad en los datos y elementos a probar, de modo que los resultados obtenidos sean objetivamente representativos, punto especialmente crítico en pruebas de rendimiento.

El entorno debe ser reinstalable, así ante pruebas que modifican el ambiente, se puede volver a un ambiente libre de cambios.

Éste debe incluir:

- Todo el hardware necesario para la prueba: Procesadores, periféricos, drivers, red, etc.
- Todo el software necesario para las pruebas: Sistema Operativo, Software de BD, Software a probar, etc.
- Interfaz en general: Archivos recibidos de otros sistemas, etc.

Es importante tener presente algunos conceptos a la hora de la evaluación que se realizará en esta tesina.

Una ***prueba exitosa*** es aquella que encuentra muchos defectos.

Por lo tanto, el desarrollo exitoso puede llevar a un conjunto de pruebas no exitoso (no se encuentran defectos críticos)

Un *caso de Prueba Bueno*, es aquel que tiene alta probabilidad de detectar un defecto aún no descubierto.

Un *Caso de Prueba exitoso* es aquel que detecta un defecto aún no descubierto.

3.9 Resumen

Es este capítulo analizamos los conceptos relacionados con los beneficios de la corrección de errores en etapas tempranas. Esto nos demuestra que es necesario probar las aplicaciones antes de su lanzamiento al público. También hemos introducido conceptos y definiciones sobre calidad de software y cómo aumenta cuando se aplica un plan de pruebas durante el desarrollo, y sobre como debe ser el ambiente de las pruebas.

3.10 Referencias

[1] The Art of Software Testing, Glenford J. Meyer – Segunda edición.

[2] The Complete Guide to Software Testing, Bill Hetzel - Second edition.

[3] Software Testing In The Real World: Improving The Process, Edward Kit.

[4] Mitigating Risk With Effective Requirements Engineering, Borland.

[5] Foundation Level Syllabus (2011) – ISTQB.

[6] Ingeniería de Software Orientada a Objetos Con Java e Internet, Alfredo Weitzenfeld – 2005.

[7] http://www.bbc.co.uk/mundo/noticias/2012/02/120207_tecnologia_seguridad_camara_internet_privacidad_az.shtml

[8] Juran's Quality Control Handbook, Joseph M.Juran y A. Blanton Godfrey – Quinta edición.

[9] The Changing Of Quality In America, Philip Crosby – 1987.

[10] http://www.qualitygurus.com/gurus/list-of-gurus/armand-feigenbaum

[11] http://www.iso.org/

[12] http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=15570

[13] 610-1991 – IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries – Agosto 2002.

[14] Calidad del Software, Juan Manuel Cueva Lovelle – 1999.

[15] Herramientas open source para testing de aplicaciones Web. Evaluación y usos, Lic. J. Díaz, Lic. C. Banchoff, A.C. A. Rodríguez y A.C. V. Soria – CACIC 2009.

[16] IEEE Standard for Software Verification and Validation.

[17] Ingeniería del Software: Un Enfoque Práctico, Roger Pressman – Quinta Edición.

[18] Verifying and Validating Software Requirements and Design Specifications, Barry W. Boehm – IEEE.

[19] Defect, Fault, Error,..., or Failure?, Behrooz Parhami – University of California, Santa Barbara – IEEE.

Capítulo 4: Pruebas de software

4.1 Introducción

Las pruebas del software ayudan a garantizar la calidad del mismo y representan una revisión final de las especificaciones, del diseño y de la codificación.

Según Myers [1]: "*realizar pruebas, aunque parezcan creativas y completas, no puede garantizar la ausencia de todos los errores. Es importante diseñar los casos de pruebas y no realizarlas sin previa planificación, ya que los resultados podrían no ser exitosos. Otra premisa de Myers es que las pruebas deben ser necesariamente incompletas ya que no es posible probar el sistema en su totalidad*".

Una célebre frase de Dijkstra [2] dice que: "*El testing puede ser usado para mostrar la presencia de bugs, pero nunca para mostrar su ausencia"*.

El objetivo de las pruebas es encontrar la mayor cantidad de errores posibles, preferentemente antes de que los encuentre el usuario final.

Como se mencionó en el capítulo anterior, el diseño de las pruebas debe comenzar al inicio del proyecto, pues si las pruebas se realizan una vez que el proyecto está terminado (o casi terminado), el volumen de los casos de prueba es demasiado extenso y aún así no se puede asegurar que se ha probado todo.

Como introducción a las pruebas, podemos decir que existen dos formas o estrategias para probar un software:

1. Pruebas de **Caja Blanca**: permite examinar la estructura interna del programa. Las pruebas se basan en examinar la lógica del mismo.
2. Pruebas de **Caja Negra**: el objetivo de estas pruebas es encontrar errores de tipo funcionales. No hacen hincapié en el comportamiento interno ni en la estructura del programa. Se enfocan en encontrar circunstancias en las que el programa no se comporta de acuerdo a sus especificaciones.

Existen también diferentes tipos de prueba. Basándonos en los niveles del Modelo en V, se tienen:

- Pruebas de unidad
- Pruebas de integración
- Pruebas de sistema

- Pruebas de aceptación

Estos tipos de pruebas serán analizados más adelante en este capítulo, pero previamente debemos definir algunos otros conceptos importantes.

4.2 Caso de prueba

Un caso de prueba, en teoría se debe focalizar en un ítem por vez, y luego realizar un análisis del resultado esperado.

Un caso de prueba consta de cuatro elementos:

1. Objetivo: la característica del sistema a probar.
2. Datos de entrada y de ambiente: datos a introducir al sistema que se encuentra en condiciones preestablecidas.
3. Comportamiento esperado: la salida o la acción esperada en el sistema de acuerdo a los requerimientos del mismo.
4. Comprobación del resultado esperado: método o forma de realizarlo.

Hay que tener en cuenta que si se descubre un nuevo caso de prueba mientras se están ejecutando las mismas, se lo debe documentar para luego poder recrear los pasos que nos hicieron llegar a este nuevo caso.

4.2.1 Diseño del caso de prueba

En el libro “The Art Of Software Testing” [1] Myers dice que para el diseño de casos de prueba, se necesitan dos tipos de información:

- La especificación del módulo.
- El código fuente del módulo.

La especificación define los parámetros de entrada y salida del módulo y su función.

El procedimiento de diseño de las pruebas de un módulo se inicia con el análisis de la lógica del módulo usando uno o más de los métodos de caja blanca, y luego complementar estos casos de prueba mediante la aplicación de métodos de caja negra a la especificación del módulo. Estos conceptos se definen más adelante.

4.3 Ciclo de las pruebas

Al igual que el desarrollo de software, las pruebas también tienen diferentes etapas [3]:

1) Planificación y control: en esta etapa se define el objetivo de las pruebas y se evalúa el nivel de riesgo, experiencias anteriores ayudan al líder del grupo de pruebas para llevar a cabo estas tareas.
2) Análisis y diseño: en esta etapa se deben transformar los objetivos planteados en la etapa anterior en casos de pruebas concretos. También se determina cuáles son las pruebas que pueden automatizarse y cuáles no.
3) Implementación y ejecución: en esta etapa se debe preparar el entorno para realizar las pruebas y ejecutar las mismas.
4) Evaluación: en esta etapa se evalúan los resultados de los casos de prueba con relación a los resultados esperados. Se preparan reportes de los resultados de las pruebas realizadas, y se determina si es necesario realizar más pruebas.
5) Cierre: en esta etapa es necesario recolectar la información de las pruebas realizadas. Se analizan los resultados, se escriben las conclusiones y se guardan los entornos y los resultados de las pruebas.

4.4 Tipos de pruebas

Como se mencionó en el capítulo anterior, para poder hablar de calidad del software es imprescindible asegurarnos que cada etapa de su desarrollo ha sido probada acorde al tipo de aplicación y funcionalidad. Es por esto que vamos a trabajar con los distintos tipos de pruebas que muestra el Modelo en V.

4.4.1 Pruebas de unidad

Este tipo de prueba permite examinar cada módulo de manera individual para asegurar el correcto funcionamiento.

Comienza una vez que se ha desarrollado, revisado y verificado el código fuente. Como los módulos no son independientes, se necesita un software que conduzca las

pruebas. Para ello, se pueden utilizar "*resguardos*" (como por ejemplo objetos MOCK[2]) los cuales simulan la interacción con otros módulos, y que realicen una mínima manipulación de datos y devuelvan control al módulo de prueba que lo invocó.

4.4.2 Pruebas de Integración

Estas pruebas se realizan para poder ver si existen fallas en la interacción de un módulo con otros módulos. Ayudan a verificar que un gran conjunto de partes de software funcionen juntos.

La pregunta es: si cada módulo anda bien por separado, cuando se integra, ¿funciona con el resto de los módulos? La respuesta es NO SIEMPRE, porque aquí entra en juego la interacción de los mismos: los datos se pueden perder en una interfaz; un módulo puede tener un efecto adverso e inadvertido sobre otro; las subfunciones, cuando se combinan, pueden no producir la función principal deseada, etc.

Existen dos técnicas para realizar pruebas de integración:

1. *No Incremental*: Esta es la técnica del *Big Bang, cuyo objetivo* es integrar todos los módulos juntos y así llegamos a un caos total, dado que los errores pueden aparecer en grandes cantidades sin poder distinguir el origen de cada uno.
2. *Incremental*: En este caso, el programa se construye y se prueba en pequeños segmentos en los que los errores son más fáciles de aislar y de corregir. Se puede realizar de dos maneras:
    - *Ascendente (bottom-up)*: la construcción comienza con los niveles más bajos. Dado que se integra de abajo hacia arriba, los módulos subordinados ya se encuentran integrados entonces no se necesitan los resguardos. Los módulos se agrupan por funcionalidad.
    - *Descendente (top-down)*: se integran comenzando por el módulo de control principal y luego se integran los módulos subordinados. Puede ser en profundidad o en anchura.

Se pueden combinar ambas técnicas (ascendente y descendente). Pero primero se deben identificar los módulos críticos, en especial aquellos más complejos, los que tienen varios requerimientos, mayor nivel de control, etc.

[2] MOCK: objeto de dominio simulado que se limita a afirmar que se invocó al método correcto, con los parámetros esperados, en el orden correcto, es generado de manera ad-hoc para la prueba del módulo.

4.4.3 Pruebas de Sistema

Este tipo de pruebas permite analizar al sistema como un todo, verificando que cumple con los requisitos especificados, viendo la interacción del mismo con el hardware, lo cual lleva a realizar pruebas de rendimiento, seguridad, resistencia, de recuperación, etc.

4.4.4 Pruebas de Aceptación

Este tipo de pruebas permiten encontrar errores que sólo el usuario final puede descubrir. Hay situaciones en las cuales no podemos predecir cómo el cliente o usuario usará realmente el sistema y para ello se pueden realizar dos tipos de pruebas:

1) *Alfa*: es conducida por el cliente en el lugar de desarrollo. Se usa el software de forma natural, con el encargado del desarrollo supervisando al usuario y registrando errores y problemas de uso. Las pruebas alfa se llevan a cabo en un entorno controlado.
2) *Beta*: se lleva a cabo en uno o más lugares de clientes por los usuarios finales del software. A diferencia de la prueba alfa, el encargado del desarrollo no está presente. El cliente registra todos los problemas (reales o imaginarios) que encuentra durante la prueba beta e informa a intervalos regulares al equipo de desarrollo.

4.5 Técnicas de prueba

Existen cuatro técnicas para la implementación de las pruebas: estáticas, funcionales, no funcionales y estructurales, las técnicas estáticas y estructurales se realizan sobre el código fuente, y las otras sobre el sistema. A continuación se describen cada una de ellas.

4.5.1 Estáticas

Esta técnica se aplica al código fuente del sistema y permite examinar la estructura interna del programa.

Esta técnica permite examinar el código de una manera exhaustiva para poder garantizar la correcta ejecución del sistema, en particular es muy importante realizarlas en sistemas críticos como por ejemplo software para elementos de salud.

Pruebas de caja blanca: es un método de diseño de casos de prueba que usa la estructura de control del diseño procedimental para obtener los casos de prueba. Las mismas garantizan que [4]:

- Se ejecutan por lo menos una vez todos los caminos independientes de cada módulo.
- Se ejecutan todas las decisiones lógicas en sus valores verdadero y falso.
- Se ejecutan todos los bucles en sus límites y con sus límites operacionales.
- Se ejecutan las estructuras internas de datos para asegurar su validez.

Las pruebas de caja blanca son necesarias ya que descubren errores que pueden ser: errores de tipeo, errores en las condiciones, etc.

Dentro de las pruebas de caja blanca, se encuentran dos técnicas:

1) *Prueba de camino básico*: esta técnica fue propuesta por Tom McCabe y permite obtener una medida de la complejidad lógica de un diseño procedimental y usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución. Los casos de prueba obtenidos del conjunto básico garantizan que, durante la prueba, se ejecuta por lo menos una vez cada sentencia del programa.

   La *complejidad ciclomática*, define el número de "caminos independientes" en el conjunto básico y brinda un límite superior para el número de casos necesarios para ejecutar todas las instrucciones al menos una vez.
   Un *camino independiente* es cualquier camino del programa que introduce, por lo menos, un nuevo conjunto de sentencias de proceso o una nueva condición.

   El cálculo de la complejidad ciclomática nos dice cuántos caminos independientes se tienen. Se puede calcular de tres formas, y se explica mejor con un ejemplo, supongamos el siguiente grafo de flujo sobre una función o procedimiento:

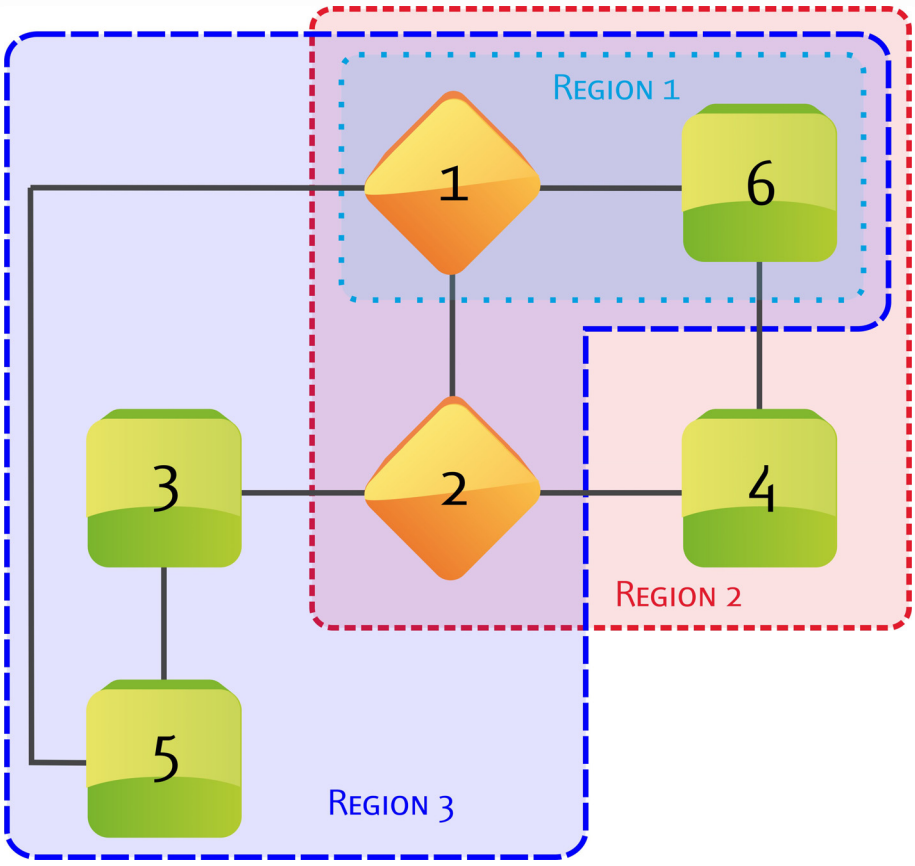


Figura 7: Grafo de flujo.

I. El número de regiones del grafo de flujo coincide con la complejidad ciclomática, en el ejemplo se observan tres (3) regiones.

II. La complejidad ciclomática, V(G), de un grafo de flujo G se define como:

$$V(G) = A - N + 2$$

donde *A* es el número de aristas del grafo de flujo y *N* es el número de nodos del mismo.

En el ejemplo se tiene: A = 7, N = 6, entonces: V(G) = A – N + 2 →

$$V(G) = 7\text{-}6\text{+}2 \rightarrow \mathbf{V(G) = 3}$$

III. La complejidad ciclomática, V(G), de un grafo de flujo G también se define como:

$$V(G)=P+ 1$$

donde *P* es el número de nodos predicado contenidos en el grafo de flujo *G*.

En el ejemplo P = 2, entonces **V(G) = 3**.

Los caminos encontrados son:

Camino 1: 1 y 6.

Camino 2: 1, 2, 4 y 6.

Camino 3: 1, 2, 3, 5, 1 y 6.

2) *Prueba de estructura de control*: La técnica de prueba del camino básico no es suficiente por sí sola. Las siguientes técnicas de estructura de control amplían la cobertura de la prueba y mejoran la calidad de la prueba de caja blanca.

2.1) Pruebas de condición: El método se focaliza a probar cada condición en un programa. Si un conjunto de pruebas es efectivo para detectar defectos en las condiciones, probablemente lo es para detectar otros defectos.

2.2) Prueba del flujo de datos: Se seleccionan caminos de prueba de un programa de acuerdo con la ubicación de las definiciones y los usos de las variables del programa. Las estrategias de prueba de flujo de datos son útiles para seleccionar caminos de prueba de un programa que contenga sentencias *if* o *loops* anidados.

2.3) Pruebas de bucles: Se focaliza exclusivamente en la validez de las instrucciones de bucles (loops). Existen 4 clases:

2.3.1) Simples: se le debe aplicar las siguientes pruebas:

I. Pasar por alto el bucle.
II. Pasar una sola vez por el bucle.
III. Pasar 2 veces por el bucle.
IV. Hacer *m* pasos por el bucle con m < n (n número máximo de pasos permitidos en el bucle).
V. Hacer n-1, n y n+1 pasos por el bucle.

2.3.2) Anidados: si se extienden los pasos anteriores a los bucles anidados, el número de pruebas crecería conforme crecen los límites superiores de los bucles. Para ello se proponen las siguientes pruebas:

I. Comenzar por el bucle interior y establecer los demás bucles con sus valores mínimos.
II. Llevar a cabo las pruebas de bucles simples para el bucle más interior, mientras se mantienen los parámetros de iteración de los bucles externos en sus valores mínimos. Añadir otras pruebas para valores fuera de rango o excluidos.
III. Progresar hacia fuera, llevando a cabo pruebas para el siguiente bucle, pero manteniendo todos los bucles externos en sus valores mínimos y los demás bucles anidados en sus valores “típicos”.
IV. Continuar hasta que se hayan probado todos los bucles.

2.3.3) Concatenados: se pueden probar como bucles simples, siempre que sean independientes del resto. Sin embargo, si hay dos bucles concatenados y se usa el controlador del bucle 1 como valor inicial del bucle 2, entonces los bucles no son independientes, para lo cual se recomienda aplicar las pruebas para bucles anidados.

2.3.4) No estructurados: siempre que sea posible, esta clase de bucles se deben rediseñar para que se ajusten a las construcciones de programación estructurada.

4.5.2 Funcionales

Esta técnica de pruebas se centra en la funcionalidad del sistema. No se llama prueba funcional porque se refiere a su función si no porque se la ve como una función matemática a la cual se le ingresa un valor, y retorna otro, de ahí el nombre de caja "negra u opaca".

Prueba de caja negra: se centran en los requisitos funcionales del software. Se debe contar con un conjunto de condiciones de entrada que ejerciten completamente todos los requisitos funcionales de un programa. Este tipo de pruebas intenta encontrar errores de:

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

Existen diferentes técnicas de pruebas de caja negra:

1. *Técnica aleatoria*: es bastante relativo, pero siempre se realiza las pruebas a una milésima parte. Se deberían elegir, pocos casos de prueba, pero buenos. No probar casos triviales, por ejemplo aquellos que están salvados con excepciones y que se sabe que funcionan correctamente.

2. *Técnica por división de clase de equivalencia*: es un proceso sistemático que identifica clases de pruebas representativas de grandes conjuntos de otras pruebas posibles; la idea es que el producto bajo prueba se comportará de la misma manera para todos los miembros de la clase.

Esta técnica tiene como puntos ventajosos que:

- Es sistemático con relación a las especificaciones.
- Permite un estudio sistemático de los casos.
- Minimiza los casos a ser probados.
- Permite detectar inconsistencias o incompletitud en las especificaciones.

Pero se pueden mencionar las siguientes desventajas:

- No tiene en cuenta las posibles interacciones entre condiciones de las pruebas.
- No resuelve problemas de estabilidad del software. Por ejemplo: repetición de la misma función, series que combinan funciones de cierta forma, etc.

3. *Análisis de valores de frontera*: Es una variante del particionamiento de equivalencias que, en vez de seleccionar cualquier elemento como representativo de una clase de equivalencia, se seleccionan los bordes de una clase, dado que los errores suelen darse en los valores límites del campo que en el centro. Por ejemplo si una clase de equivalencia está compuesta por un intervalo de números enteros que van del 1 al 10, tomar como valores de entrada los números del borde del intervalo 1 y 10.

4. *Tablas de decisión*: es una presentación sintética y matricial de condiciones, acciones y reglas asociadas. Se utilizan para el análisis de reglas complejas.
   Se debe utilizar una matriz, orientada a probar las *reglas*. Al menos un línea o caso de prueba por cada regla.
   La tabla de decisión se lee en forma ordenada de izquierda a derecha y de arriba abajo.
   Las *Condiciones* representan los valores de entrada y condiciones de ejecución de las pruebas. Las *Acciones* representan los resultados esperados.
   Se debe incluir los números de regla en los casos para llevar la trazabilidad de las reglas y asociar los casos a las reglas.
   Se deber verificar la completitud, al menos una línea o caso de prueba por regla.

4.5.3 No Funcionales

Las pruebas no-funcionales son indispensables. Gran parte de la percepción de calidad del software se determina en estos atributos. Una vez resuelto los aspectos funcionalmente básicos, importa la forma en que los mismos son resueltos. Dentro de este tipo de pruebas se encuentran las pruebas de usabilidad, robustez, rendimiento, seguridad entre otras, como se menciona a continuación:

Usabilidad: Estas pruebas verifican la facilidad de uso de una interfaz y también definen una metodología de trabajo en la etapa de diseño. Por lo general, la usabilidad se prueba cuando el sistema se encuentra casi terminado, y se aplican las pruebas Alpha y Beta descriptas en la sección 4.4.4 de este capítulo en pruebas de aceptación.

Existen diferentes pautas de usabilidad, las primeras fueron establecidas por Jakob Nielsen [5]. Con el correr de los años y el avance de las tecnologías, estas pautas fueron actualizadas y se desarrolló una norma ISO 9241_11 [6] para tal fin.

Entre las pautas más relevantes, se encuentran aquellos que responden a las siguientes preguntas:

- ¿Los mensajes de error son claros?
- ¿Se utiliza el mismo estilo de letra?
- ¿Los nombres de las acciones de los botones y/o enlaces son claros?
- ¿En los formularios se indican claramente los campos requeridos? ¿Se validan?
- ¿El sistema funciona en diferentes máquinas con diferentes componentes de Hardware y/o Sistemas Operativos?
- ¿El sistema cuenta con ayuda?

Robustez: se debe probar cómo se comporta el software bajo condiciones que no son normales. No existen normas que indiquen qué es lo que se debe probar. Se debe evaluar los eventos que podrían ocurrir, la probabilidad de que ocurran y las consecuencias.

Se puede probar, por ejemplo, cómo se comporta el software si:

- Se interrumpen las transacciones.
- Se reciben datos erróneos.
- Se invoca un bucle infinito.
- Se corta la energía.
- Se cae el sistema operativo.
- Etc.

Rendimiento: se pueden realizar diferentes pruebas para observar el comportamiento del software frente a la sobrecarga de datos (pruebas de volumen), de usuarios (pruebas de carga) o en una fusión de ambos (pruebas de stress). Para este tipo de pruebas en imprescindible la utilización de herramientas que las automaticen, por ejemplo, para simular el acceso concurrente de una determinada cantidad de usuarios o la múltiple carga de formularios.

Seguridad: estas pruebas consisten en elaborar casos que alteren los controles de seguridad del programa. Una forma de elaborar estos casos de prueba es estudiar los problemas de seguridad conocidos en sistemas similares y generar casos de prueba que traten de generar los mismos problemas en el sistema que está probando.

Las aplicaciones que funcionan en red, y en particular las aplicaciones Web, necesitan pruebas de seguridad de mayor nivel que la mayoría de las aplicaciones, por ejemplo para los sitios de e-commerce, también para sistemas que manejen información confidencial como por ejemplo el sistema Guaraní el cual mantiene información de alumnos y docentes de la universidad.

Otros atributos no funcionales son: Flexibilidad y Evolución, Portabilidad, Simplicidad de Mantenimiento, Disponibilidad y Eficiencia.

4.5.4 Estructural

Este tipo de pruebas se basa en la estructura interna del software. Se realiza recorriendo el código y se debe probar diferentes caminos posibles.

La cobertura estructural es el porcentaje de pruebas realizadas con relación a la totalidad de los casos posibles. Se aspira a cubrir el 100% de cobertura estructural (aunque no siempre posible).

Existen diferentes tipos de cobertura estructural, como ser: de sentencias, de decisiones y de condiciones.

Pueden realizarse en todos los niveles de prueba, especialmente en pruebas de unidad y de integración.

4.6 Resumen

En este capítulo se destacó la necesidad de probar un software y lo importante que es tener un plan de pruebas. El objetivo de diseñarlo es obtener un conjunto de pruebas que tengan la mayor probabilidad de descubrir los defectos del software.

Las técnicas de prueba descriptas se pueden aplicar a cualquier tipo de sistema, pero dado que nuestra tesina está enfocada a un tipo específico, nos centraremos en aplicaciones Web.

En esta sección de la tesina revisamos distintas técnicas para probar un software. Se explicó qué es un caso de prueba y cómo se define, también se definieron los distintos tipos de pruebas y cuáles son las diferentes técnicas que se pueden aplicar.

4.7 Referencias

[1] The Art of Software Testing, Glenford J. Meyer – Segunda edición.

[2] On the cruelty of really teaching computing science, Edsger W. Dijkstra.

[3] Herramientas open source para testing de aplicaciones Web. Evaluación y usos, Lic. J. Díaz, Lic. C. Banchoff, A.C. A. Rodríguez y A.C. V. Soria – CACIC 2009.

[4] Ingeniería del Software: Un Enfoque Práctico, Roger Pressman – Quinta Edición.

[5] Prioritizing Web Usability – Jakob Nielsen, Hoa Loranger.

[6] ISO 9241 (parte 11).

Capítulo 5: Criterios de evaluación de herramientas FLOSS para pruebas de software

5.1 Métodos para evaluar la madurez de los proyectos Open Source

CMM es un proceso de evaluación de madurez del desarrollo del software que clasifica las empresas en cinco (5) niveles de madurez (desde el nivel 1 que es el nivel inicial hasta el 5 que es el nivel óptimo). CMM ayuda a identificar los puntos importantes que se deben estudiar y trabajar para mejorar tanto el proceso como la calidad del software.

Este modelo es demasiado complejo y la infraestructura del proceso que utiliza no es aplicable a los procesos de desarrollo de software libre [1].

Es por esto que en los últimos años se han desarrollado varios métodos para definir un proceso de evaluación de software FLOSS. Algunos se centran en ciertos aspectos como la madurez, la durabilidad y la estrategia de la organización en torno al proyecto Open Source en sí. Otras metodologías agregan aspectos funcionales al proceso de evaluación.

A la hora de seleccionar un software FLOSS, es necesario contar con un método de clasificación, examinando las limitaciones y los riesgos que tiene este tipo de software.

Algunas preguntas generales para hacer antes de seleccionar un software son las siguientes:

- ¿Cuál es la durabilidad del software?
- ¿Qué nivel de estabilidad tiene o se espera?
- ¿Es fácil y factible agregarle funcionalidad al software?
- ¿Tiene una comunidad activa que lo avale?

Es por esto que se definieron los modelos de madurez de proyectos de software FLOSS, los cuales en su mayoría cuentan con:

1. Una puntuación según las características del software.
2. Una elección final, según los criterios de puntuación establecidos

A continuación se hará una descripción general de los métodos más populares:

5.1.1 OSMM (Open Source Maturity Model de Capgemini)

Este modelo fue uno de los primeros, se definió en el año 2003. Posee una licencia privada pero con autorización para su distribución [2 y 3]. Fue desarrollado para comparar y decidir la utilización de un producto.

Define veintisiete (27) parámetros, de los cuales doce (12) de éstos están divididos en cuatro (4) categorías genéricas del producto: Producto, Integración, Uso y Aceptación. Por ejemplo, la edad (inicio del proyecto), el tipo de licencia, la comunidad de desarrollo, el tipo de colaboración con otros productos, la adecuación a estándares, etc. Los restantes quince (15) son características basadas en las necesidades del usuario. Por ejemplo: usabilidad, seguridad, interfaz de usuario, independencia de la plataforma, soporte, etc.

Cada uno de estos parámetros puede tener un puntaje entre 1 (menos importante) y 5 (más importante).

En líneas generales, podemos decir que este modelo tiene las siguientes características:

1. La ponderación de los parámetros es realizada por 2 (dos) evaluadores.
2. Permite determinar la madurez del producto.
3. Ayuda a determinar si el producto cumple con los requerimientos de la empresa u organización a la cual está dirigido.
4. Permite comparar los productos con alternativas comerciales.
5. Muestra la importancia de utilizar productos Open Source.

5.1.2 OSMM (Open Source Maturity Model de Navica)

Este método [4 y 5] es más compacto que el propuesto por Capgemini. Se creo un año después y sólo tiene en cuenta seis (6) características. También define algunos elementos que permiten una evaluación parcial del proceso de desarrollo.

El proceso consta de cuatro (4) fases:

1. Primera Fase: Seleccionar el software que se va a evaluar.
2. Segunda Fase: Ponderación según las siguientes categorías:
    I. Software (4 ptos.).
    II. Soporte (2 ptos.).
    III. Documentación (1 pto.).
    IV. Capacitación (1 pto.).
    V. Integración (1 pto.).

VI. Servicios de profesionales (1 pto.).

Los puntajes marcados son definidos por Navica. Cada una de estas características tiene una plantilla asociada.

3. Tercer Fase: En esta fase se aplica cada una de las plantillas correspondientes a las categorías nombradas en la fase anterior en las cuales se sugiere qué aspectos dentro de la categoría deben ser evaluados ponderando cada uno de ellos.
4. Cuarta Fase: consiste en multiplicar la puntuación de cada categoría por su ponderación para producir una puntuación final de cero (0) a cien (100).

5.1.3 BRR (Business Readiness Rating)

El proyecto BRR fue anunciado en el 2005 y plantea un método que permite evaluar software FLOSS [6].

Dado que existen miles de proyectos FLOSS, en diferentes etapas de desarrollo (abandonados, iniciados, avanzados, etc.) BRR ayuda a determinar en qué etapa se encuentra actualmente el proyecto y si es probable que siga progresando a etapas posteriores.

El modelo de evaluación BRR incluye cuatro (4) pasos:

1. En primer lugar, se crea una lista con el software a evaluar.
2. Luego se clasifican y ponderan los criterios de selección.
3. Se recopilan los datos para cada criterio.
4. Por último, se realiza el cálculo y publicación de los resultados.

BRR define doce (12) criterios que pueden ser utilizados para evaluar el software una vez que se estableció la lista inicial de productos. Se sugiere que sólo seis (6) o siete (7) criterios son realmente utilizados en la evaluación. Los criterios sugeridos son los siguientes:

1. Funcionalidad: ¿El software satisface las necesidades del usuario?
2. Usabilidad: ¿El software es intuitivo, fácil de instalar, fácil de configurar y fácil de mantener?
3. Calidad: ¿El software está bien diseñado, implementado y probado?

4. Seguridad: ¿Qué tan seguro es el software?
5. Rendimiento: ¿Cómo se comporta el software ante ciertas circunstancias?
6. Escalabilidad: ¿Puede hacer frente a grandes volúmenes de datos?
7. Arquitectura: ¿El software es modular, portable, flexible, extensible y abierto? ¿Puede ser integrado con otros componentes?
8. Soporte: ¿La comunidad cuenta con apoyo profesional?
9. Documentación: ¿Existe documentación de buena calidad?
10. Adopción: ¿El software ha sido adoptado por la comunidad, el mercado y la industria?
11. Comunidad: ¿La comunidad del software es activa?
12. Profesionalismo: ¿Que nivel de profesionalismo presentan el proceso de desarrollo y la organización del proyecto?

BRR no ha tenido el nivel de aceptación que se esperaba. A partir de Julio de 2010, se ha dado de baja el proyecto, al parecer porque el proyecto no ha sido capaz de crear la comunidad próspera que esperaban. Los creadores están aplicando actualmente planes para revitalizar la comunidad.

Si bien el proyecto no está activo nos resultó interesante incluirlo en esta tesina ya que se tomaron algunas de sus características como base para poder definir nuestra propia plantilla, la cual será descripta en la siguiente sección del informe.

5.1.4 Qualification and Selection of Open Source Software (QSOS)

QSOS es un método [7], diseñado para calificar, seleccionar y comparar software FLOSS de una manera objetiva, bajo la licencia GNU Free Documentation License.

El proceso consiste en cuatro (4) pasos. Cada una de estas etapas son independientes entre si e iterativas.

Paso 1 – Definición: El objetivo de este paso es definir los elementos, que servirán para los siguientes tres pasos. Estos elementos son:

- Clasificación del software y descripción funcional.
- Clasificación de Licencias.
- Clasificación de la comunidad.

Paso 2 – Evaluación: Se evalúa el software en tres (3) ejes: cobertura funcional, riesgo del usuario y los riesgos de los desarrolladores.

El objetivo de este paso es llevar a cabo la evaluación del software. Algunos ejemplos son:

1. Información general del software: nombre, fecha de creación, autor, tipo de software, licencia, sistemas operativos en el cual pueda ser utilizado, etc.
2. Servicios ofrecidos: documentación, capacitación, etc.
3. Aspectos funcionales y técnicos: tecnología en la cual esta implementado, prerrequisitos de instalación, etc.

En esta etapa también se determina la puntuación de cada uno de estos criterios.

Paso 3 – Calificación: Se definen los filtros en base a los puntos que se evaluarán. Por ejemplo, se pueden considerar familias de software que sean compatibles con un sistema operativo.

Esta etapa sirve para eliminar el software que no satisface las necesidades del usuario en un contexto específico.

Paso 4 – Selección: Aplicación del paso tres (3), con los datos dados por los primeros pasos, formulando consultas, comparaciones y la selección del producto.

El objetivo de este paso es identificar el software que cumple con los requisitos del usuario de dos modos posibles:

- Una selección estricta: se elimina el software si no cumple con los requerimientos definidos en el paso tres (3).
- Una selección más amplia: en vez de eliminar el software que no cumple con los requisitos, los clasifica mientras mide los huecos con los filtros aplicados.

Este método también provee plantillas y cuadros para poder evaluar y aplicar los pasos anteriores.

5.1.5 NASA's Reuse Readiness Levels (RRL)

La NASA Earth Science Data Systems (ESDS) Software Reuse Working Group (WG) reconoció la necesidad de medir la madurez del software para su reutilización y propone una serie de niveles de preparación para la reutilización (RRLs o “Reuse Readiness Levels” en inglés). La madurez de una tecnología en

particular se puede medir de varias maneras, un método conocido es el Technology Readiness Levels (TRLs) [8].

El SRWG recomienda la adopción de RRL, ya que tratan específicamente la madurez del software en el sentido de reutilización como un medio para estimular y facilitar la reutilización del software.

El SRWG identificó nueve (9) áreas temáticas que se consideran importantes para medir la madurez en la reutilización del software:

- Documentación: la información que describe el software y cómo usarlo, como por ejemplo: guías de instalación, manuales de usuario, foros, tutoriales, etc.
- Extensibilidad: la posibilidad de incorporarle nueva funcionalidad o que permita modificarla.
- Cuestiones de Propiedad Intelectual: que tenga bien definido cuáles son las reglas para usar, modificar y distribuir el software.
- Modularidad: grado de modularidad para mejorar el mantenimiento y la reusabilidad.
- Empaquetado: es importante para garantizar la integridad, permitir la distribución, y simplificar la instalación del software.
- Portabilidad: la capacidad de ser instalado o ejecutado en varias plataformas maximiza el potencial de reutilización y aumenta la flexibilidad y el reuso del software.
- Cumplimiento de estándares: Al cumplir con estándares, el software incrementa la posibilidad de aceptación.
- Soporte: Disponibilidad de asistencia al usuario, como ser: apoyo técnico, comunidad de usuarios, ayuda online, documentación, etc.
- Verificación y Pruebas: si el software es verificado y probado, aumenta la confianza y reduce los riesgos y costos potenciales de la reutilización.

Siguiendo el formato de TRL, hay nueve (9) niveles de RRL que van de 1 (menos maduro) a 9 (más maduro):

RRL 1 – Reutilización limitada, el software no está recomendado para su reutilización.

RRL 2 – Reutilización inicial; la reutilización del software no es práctica.

RRL 3 – Reutilización básica, el software podría ser reutilizable por usuarios especializados con un esfuerzo, costo y riesgo sustancial.

RRL 4 – La reutilización es posible, el software puede ser reutilizado por la mayoría de los usuarios con un poco de esfuerzo, costo y riesgo.

RRL 5 – La reutilización es práctica, el software puede ser reutilizado por la mayoría de los usuarios con un costo y riesgo razonable.

RRL 6 – El software es reutilizable, puede ser reutilizado por la mayoría de los usuarios, aunque puede haber algún costo y riesgo.

RRL 7 – El software es altamente reutilizable, puede ser reutilizado por la mayoría de los usuarios con un costo y riesgo mínimo.

RRL 8 – Se puede demostrar una reutilización local, el software ha sido reutilizado por varios usuarios.

RRL 9 – Se puede demostrar una reutilización extensa, el software está siendo reutilizado por muchas clases de usuarios a través de una amplia gama de sistemas.

Este método define una tabla donde se tienen las nueve (9) áreas junto a los nueve (9) niveles con una breve descripción (Reuse Readiness Level (RRL) Topic Area Levels Summary).

RRL proporciona una guía para la reutilización, reduciendo el número de posibles soluciones que deben considerar en detalle. RRL y los niveles de área temática, pueden servir como un indicador de las zonas que se centran en la hora de crear activos reutilizables, como una guía para los proveedores.

5.2 Criterios de evaluación de herramientas: plantilla propuesta

Se toma como base de conocimiento los métodos enunciados previamente, adoptando algunos criterios de evaluación propuestos por los mismos. Estos criterios de evaluación no están basados en un método específico, dado que el objetivo de la evaluación que se va a realizar no es determinar la madurez del software sino seleccionar, bajo ciertos criterios, un subconjunto de herramientas del total existente que mejor se adapten a la funcionalidad pedida: en nuestro caso se seleccionarán herramientas de soporte para el testeo de software [9].

Los puntos más importantes a ser evaluados son:

- Descripción general: una breve descripción de la funcionalidad de la herramienta, mencionando la URL del sitio oficial, el lenguaje en el que está desarrollada, etc.
- Documentación: el tipo de documentación asociada existente. Se debe encontrar en el sitio oficial o dentro del producto.
    1. Guía de instalación.
    2. Manual de usuario.
    3. FAQs.
    4. Soporte online: foro, blog, lista de e-mails, etc.
- Madurez: Se relevarán datos referidos a la herramienta que permitan medir el nivel de madurez del software. Algunos aspectos a analizar son:
    1. Inicio del proyecto.
    2. Última versión de la herramienta.
    3. Grado de actualización.
    4. Actividad en lanzamientos.
    5. Actividad en el reporte de errores.
- Licencia: se nombrará el tipo de licencia a la que pertenece la herramienta.
- Plataforma: se nombrará el Sistema Operativo (SO) en el que pueden ser instaladas dichas herramientas (portabilidad).
- Interfaz: se nombrará el tipo de interfaz (GUI o consola) con el que cuenta la herramienta.

Para cada una de las herramientas, se generará un cuadro con la información provista por la misma, y otro con el grado de cumplimiento de los criterios elegidos. Las siguientes tablas muestran las plantillas para medir la madurez desarrolladas por nosotras.

Por cada herramienta se genera una plantilla con la siguiente información:

Nombre:

Descripción:

Sitio oficial:

Datos generales

Herramienta	Versión	Fecha de inicio del proyecto	Licencia	Plataforma	Tipo de Interfaz	Lenguaje

Criterios de documentación

Herramienta	Guía de instalación	Manual de instalación	Preguntas frecuentes	Soporte Online			Adicional
				Foro	Lista de mail	Blog	

Criterios de madurez [*1]

Herramienta	Inicio del proyecto[*2]	Grado de actualización[*3]	Actividad en lanzamientos[*4]	Actividad en reportes de errores[*5]

([*1]) Se completa con las siguientes siglas:

M = Malo

B = Bueno

R = Regular

MB = Muy bueno

S/D = Sin datos

[*2] Se refiere a que si es muy joven quizás no tenga una madurez suficiente.

[*3] La última versión se encuentra cercana al año actual.

[*4] Se refiere a la frecuencia en la publicación de versiones.

[*5] Se refiere a la frecuencia con que se resuelven y reportan errores.

Tabla 2: Plantillas para la selección de herramientas.

5.3 Resumen

En este capítulo hemos descriptos distintos modelos para medir la madurez. En base a ellos seleccionamos los puntos más importantes para poder generar una plantilla propia que nos permita realizar una comparación entre diferentes herramientas, para luego elegir la más adecuada para nuestro propósito.

5.4 Referencias

[1] El CMM y la mejora continua del proceso de software, Diego J. Bodas Sagi.

[2] Open Source Maturity Model, Capgemini Expert Letter – 2003.

[3] http://www.osspartner.com/portail/sections/accueil-public/evaluation-osmm

[4] http://www.oss-watch.ac.uk/resources/osmm.xml

[5]http://web.archive.org/web/20080507024544/http://www.navicasoft.com/pages/osmm.htm

[6] http://www.oss-watch.ac.uk/resources/brr.xml

[7] http://www.qsos.org

[8] Reuse Readiness Levels (RRLs) - NASA Earth Science Data Systems.

[9] Evaluación de herramientas Free/Open Source para pruebas de software, Lic. J. Díaz, Lic. C. Banchoff, A.C. A. Rodríguez y A.C. V. Soria – WICC 2011.

Capítulo 6: Automatización de las pruebas y clasificación de herramientas

6.1 Automatización de las pruebas

No en todo equipo de desarrollo existe un equipo que se encargue de realizar las pruebas. El equipo de testing puede obtener muchos beneficios en sus tareas con la utilización de herramientas, aunque a veces estas puedan traer aparejados costos en cuanto a tiempo de aprendizaje de la herramienta, costo monetario en la compra de una herramienta que luego no cumple con las expectativas, etc.

La automatización de las pruebas debe ayudar a realizar el trabajo repetitivo y automático, facilitar las tareas de regresión, comparación de grandes volúmenes de datos, simulación de comportamiento, así el equipo de pruebas puede enfocarse en otras tareas tales como encontrar problemas en el software, analizar requerimientos, detectar la funcionalidad más críticas, etc.

El uso de una herramienta no garantiza el éxito en las pruebas, dado que para obtener resultados útiles es necesario conocer cómo utilizar estas herramientas y cómo interpretar sus resultados. También, en muchos casos, se requiere un esfuerzo adicional para lograr obtener beneficios de las mismas, como por ejemplo, aprender cómo se utiliza, o si se necesita un conocimiento de un lenguaje propio de la herramienta para la generación de scripts de pruebas, etc. [1].

En el caso de algunas herramientas, su compra también puede llegar a tener precios elevados, a lo cual la organización o empresa que la utilice debe realizar un estudio de su utilización contando beneficios y riesgos de su uso.

El foco de esta tesina es utilizar herramientas FLOSS, ya que las mismas pueden ser adaptadas a las necesidades de la organización o empresa, y pueden adaptarse a las necesidades del proyecto [1].

Para Whittaker [2], la utilización de una herramienta puede ser útil para mejorar y agilizar las pruebas a realizar, por ejemplo, a la hora de simular una cantidad determinada de usuarios accediendo a un sitio al mismo tiempo, o la carga de datos en un sistema, etc. Aunque, en el momento del análisis de los resultados, es necesaria la presencia de un especialista que pueda interpretar y analizar los datos arrojados por la misma y así llegar a una conclusión.

El uso de herramientas en las pruebas, posee tanto beneficios como riesgos [3]:
Entre los beneficios encontrados podemos citar:

- *El trabajo repetitivo se reduce*: por ejemplo, en el caso de realizar pruebas de regresión en las cuales se deben volver a ejecutar los casos de pruebas ya realizados, o el reingreso de los datos al volver a cero o restaurar la aplicación.
- *Una mayor coherencia y repetibilidad*: al estar la prueba automatizada es mucho más cómodo y fácil de automatizar y no cometer errores.
- *Evaluación Objetiva*: las herramientas nos pueden proporcionar medidas estáticas o de cobertura.
- *Fácil visualización de los resultados*: algunas herramientas proporcionan gráficos, tablas, etc., los cuales muestran resultados.

Algunos de los riesgos o desventajas que podemos encontrar son:

- *Se pueden llegar a tener expectativas irrealistas sobre las herramientas*: el uso de las mismas puede ser fácil o difícil dependiendo de la persona que las utiliza, o también puede suceder que se crea que es para una funcionalidad en particular, y en realidad la herramienta fue creada para otro propósito.
- *Sobrestimar el tiempo*: el tiempo que puede llevar la familiarización con la herramienta puede ser mucho más alto de lo que se había estimado. También puede llevar mucho tiempo tener experiencia en toda la funcionalidad que la herramienta provee.
- La incorporación de la herramienta a las pruebas realizadas puede demandar más tiempo que el previsto. (Tiempo de adaptación de la herramienta al ciclo de las pruebas).

En resumen la automatización de las pruebas y el uso de herramientas pueden ser de gran ayuda pero requiere de un esfuerzo inicial que puede ser muy grande. En la sección siguiente planteamos una clasificación que nos podrá permitir agrupar y analizar herramientas para distintos tipos de pruebas.

6.2 Clasificación de herramientas de prueba según el Modelo en V

En el capítulo 2 se presentaron los distintos modelos de desarrollo de software, en base al análisis realizado, se eligió el Modelo en V para seleccionar las herramientas para los distintos tipos de pruebas.

En la Figura 8, se puede observar una clasificación de las herramientas dependiendo del tipo de pruebas:

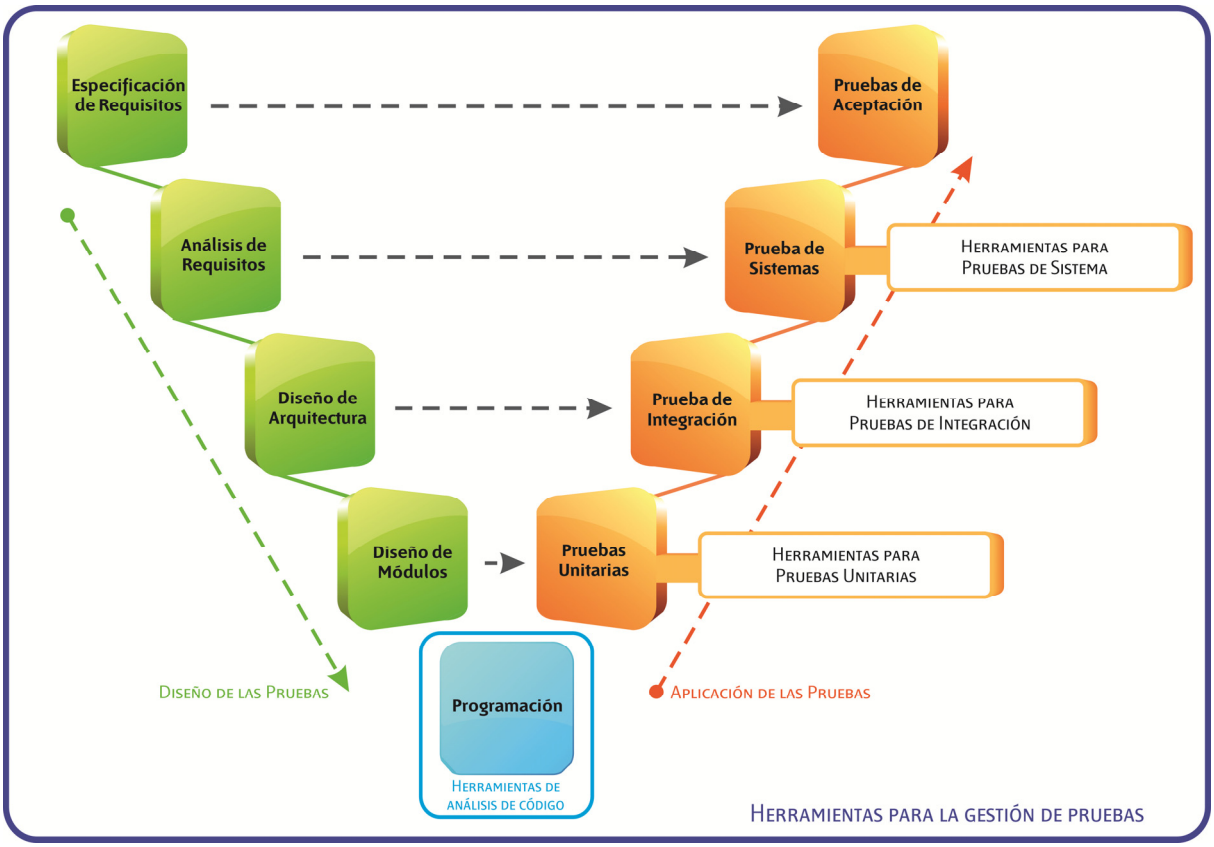


Figura 8: Modelo en V definiendo los tipos de herramientas.

Se eligió el Modelo en V, como un marco para planificar las actividades de verificación y validación con respecto a cada etapa del desarrollo del producto.

Se realiza una abstracción de niveles para mostrar la conexión entre las actividades de desarrollo y de pruebas.

Se sugiere la automatización de las pruebas en el lado derecho de la imagen (aplicación de prueba), y en el vértice (etapa de programación).

Se puede utilizar una herramienta que permita la planificación de las pruebas que abarque todo el modelo.

6.2.1 Clasificación de herramientas

Para realizar una clasificación de herramientas, se las divide según el modelo:

Gestión de las pruebas: Estas herramientas son utilizadas para mejorar, ordenar y gestionar las tareas del grupo de pruebas, permitiendo un seguimiento dentro del ciclo de vida de las mismas.

Programación: para las pruebas de análisis de código, es muy importante la utilización de herramientas, dado que es más fácil encontrar determinados errores que el ojo humano no detecta tan fácilmente.

Pruebas Unitarias: las herramientas para pruebas unitarias facilitan tareas como probar una clase o un módulo en particular, tienen por objetivo asegurar que un componente funciona según las especificaciones definidas. Las herramientas dependen del lenguaje utilizado. En algunos casos son frameworks que se pueden integrar a los IDE de programación según corresponda.

Pruebas de Integración: para esta etapa no existen herramientas específicas, sin embargo, se pueden utilizar y combinar herramientas de pruebas de unidad, que provean objetos mock para realizar las tareas de integración. Este tipo de herramientas permite facilitar la integración entre los distintos módulos.

Pruebas de sistema: este tipo de herramientas se puede utilizar una vez que el sistema ya tiene integrados todos los módulos y la interfaz se encuentra funcionalmente utilizable. Como se mencionó anteriormente, este tipo de prueba se subdivide en pruebas funcionales y no funcionales. Dado que nuestra tesina está enfocada en aplicaciones Web, los tipos de herramientas a analizar serán: para pruebas funcionales: las funcionales propiamente dichas, chequeadores de enlace, etc., y para pruebas no funcionales: herramientas específicas de la arquitectura, de rendimiento, etc. A continuación se describen los tipos de pruebas funcionales y no funcionales:

1. *Funcionales:* dentro de este tipo de pruebas, se pueden encontrar varias “categorías”, como ser: herramientas de funcionalidad, chequeadoras de enlaces, manejo de cookies, javascript, formularios, etc. Sólo analizaremos las dos primeras, dado que las otras categorías no aplican en la evaluación realizada en la presente tesina.

2. *No funcionales:* dentro de este tipo de pruebas se encuentran los siguientes:
    - *Pruebas de Rendimiento:* este tipo de pruebas, sirve para demostrar si el sistema cumple con determinados criterios de rendimiento, o para ver si un sistema responde mejor que otro, también para medir el comportamiento del sistema ante la carga de usuarios, etc.
    - *Pruebas de Aceptación:* este tipo de pruebas puede realizarse de varias maneras, con herramientas que capturan las acciones que realiza el usuario para luego analizar si le resulto complejo encontrar algo en particular. Otro ejemplo es que el personal de pruebas utilice planillas en las cuales se marque si cumple o no con características basadas en usabilidad y accesibilidad, etc.

6.3 Resumen

Es este capítulo analizamos las ventajas y desventajas de la automatización de las pruebas. Como punto principal destacamos que familiarizarse con una herramienta para automatizar las pruebas puede llevar un tiempo considerable, como así también evaluar si la misma es adecuada para nuestro objetivo. Utilizar herramientas no es la solución a todo, pero si ayuda en varias tareas como por ejemplo la organización de las pruebas, la repetición de las mismas, las pruebas de carga (tanto de usuarios como de datos), la detección de errores en el código, etc.

Para que las mismas sean ordenadas nos basamos en el Modelo en V, y se dividieron las pruebas junto con las herramientas según la etapa en la que se encuentra el desarrollo.

6.4 Referencias

[1] Herramientas open source para testing de aplicaciones Web. Evaluación y usos, Lic. J. Díaz, Lic. C. Banchoff, A.C. A. Rodríguez y A.C. V. Soria – CACIC 2009.

[2] What Is Software Testing? And Why Is It So Hard?, James A. Whittaker – Enero/Febrero 2000 – IEEE.

[3] http://www.istqb.org/download.htm

Capítulo 7: Análisis de herramientas para pruebas de software

7.1 Introducción

En este capítulo se mostrarán un conjunto reducido de herramientas las cuales estarán agrupadas según la clasificación expuesta en el capítulo anterior.

En una primera selección nos basamos en un listado de herramientas Open Source de los sitios Web opensourcetesting.org [1], testingfaqs.org [2] y SoftwareQATest.com [3], dado que los listados que presentan son muy completos y son sitios reconocidos.

Luego, buscamos la popularidad de las mismas según la puntuación de SourceForce [4] y el ranking en las búsquedas de Google.

Para poder determinar la puntuación de las herramientas según los votos de SourceForge, se calculó el porcentaje de votos positivos sobre el total de los votos. De esta manera logramos establecer la siguiente escala, según nuestro criterio:

- De 0 a 19 → Bajo.
- De 20 a 49 → Medio.
- De 50 a 75 → Bueno.
- Mayor a 75 → Muy bueno.

7.2 Clasificación de las herramientas según la etapa

Durante la investigación para la presente tesina, se evaluaron una gran cantidad de herramientas, sólo se presentará un grupo reducido de las mismas seleccionando aquellas más destacadas.

La última revisión de estas plantillas se llevó a cabo en el mes de Marzo 2012.

7.2.1 Herramientas para la gestión de pruebas

Este tipo de herramientas deben permitir definir planes de pruebas, organizándolos en distintos grupos (Test Suite) de casos de prueba según sea necesario para el proyecto abordado [5]. El equipo de pruebas puede definir si la ejecución de los casos de pruebas será manual o automatizada, ver los resultados de manera dinámica, generar informes, asignar tareas, etc.

Como característica opcional, se podría integrar a sistemas de seguimiento de errores (BTS), permitiendo mayor facilidad en la asociación de un posible error encontrado durante la ejecución del caso de prueba.

A continuación daremos una breve explicación sobre las herramientas analizadas en la presente sección.

Testlink

Es una aplicación Web, desarrollada en lenguaje PHP con motor de base de datos MySQL, Postgres o MS-SQL. Permite definir distintos niveles de usuarios. Se pueden exportar e importar los casos de pruebas a un formato XML. Puede ser integrada a herramientas BTS como Bugzilla, Mantis, Redmine, etc.

Puntuación: tiene muy buena puntuación en SourceForge (150 votos positivos y 18 votos negativos – 89% recomendado). Aparece dentro de las primeras posiciones en los resultados de las búsquedas de Google para herramientas de gestión de pruebas. Se encuentra en uno (1) de los listados mencionados en la sección 7.1 de este capítulo.

Sitio oficial: http://www.teamst.org/

RTH

Es una aplicación Web, desarrollada en lenguaje PHP, utiliza MySQL como motor de base de datos. Permite definir distintos niveles de usuarios, manejo de requerimientos y tiene facilidades para el seguimiento de errores.

Puntuación: tiene pocos votos en SourceForge (29 votos positivos y10 votos negativos – 74% recomendado). Se encuentra en dos (2) de los listados mencionados en la sección 7.1 de este capítulo.

Sitio oficial: http://sourceforge.net/projects/rth/

Testopia

Es una aplicación Web, desarrollada por la comunidad de Mozilla, escrita en lenguaje PERL con motor de base de datos MySQL o Postgres. Es una extensión de la herramienta para seguimiento de errores Bugzilla con lo cual se puede integrar al mismo. Se encuentra en uno (1) de los listados mencionados en la sección 7.1 de este capítulo.

Puntuación: no esta disponible en SourceForge, pero la incorporamos al listado dado que pertenece a la comunidad de Mozilla. Se encuentra en uno (1) de los listados mencionados en la sección 7.1 de este capítulo.

Sitio oficial: http://www.mozilla.org/projects/testopia/

Las siguientes tablas muestran las planillas para medir la madurez correspondiente a las herramientas mencionadas:

Datos generales

Herramienta	Versión	Fecha de inicio del proyecto	Licencia	Plataforma	Tipo de Interfaz	Lenguaje
Testlink	V 1.9.3 Julio 2011	Octubre 2003	GNU GPL	Independiente	Web	PHP
RTH	V 1.7.2 Abril 2009	Octubre 2006	GNU GPL	Independiente	Web	PHP
Testipoia	V 2.4 Octubre 2010	Marzo 2007	MPL	Independiente	Web	PERL

Criterios de documentación

Herramienta	Guía de instalación	Manual de instalación	Preguntas frecuentes	Soporte Online			Adicional
				Foro	Lista de mail	Blog	
Testlink	SI	SI	NO	SI	NO	NO	Guía para desarrolladores
RTH	SI	NO	SI	SI	NO	SI	S/D
Testipoia	SI	SI	SI	NO	SI	NO	Wiki

Criterios de madurez

Herramienta	Inicio del proyecto	Grado de actualización	Actividad en lanzamientos	Actividad en reportes de errores
Testlink	B	B	MB	MB (último movimiento en marzo 2012)
RTH	R	B	R	M (no hubo casi movimiento: 1 bug abierto en 2011, otro en 2010 y luego varios en 2009)
Testipoia	B	MB	MB	MB (último movimiento en marzo 2011)

Tabla 3: Plantilla para la selección de herramientas para la gestión de pruebas.

7.2.2 Herramientas para pruebas estáticas de código

Con el uso de este tipo de herramientas, es posible encontrar más fácilmente errores tales como: código muerto, código fuente no documentado o mal finalizado, variables no utilizadas y/o no inicializadas, invocación a funciones o módulos inexistentes, verificación de cantidad u posición de los parámetros, problemas de seguridad, etc.

A continuación daremos una breve explicación sobre las herramientas analizadas en la presente tesina.

Yasca

Es una herramienta desarrollada en Java y PHP. El análisis de código se puede realizar en los siguientes lenguajes Java, C, HTML, JavaScript, ASP, ColdFusion, PHP, COBOL. Se puede integrar a otras herramientas tales como: FindBugs, PMD, JLint, JavaScript Lint, PHPLint, CppCheck, ClamAV, RATS y Pixy.

Puntuación: tiene pocos votos en SourceForge (24 votos positivos y 2 votos negativos – 92% recomendado).

Sitio oficial: http://www.scovetta.com/yasca.html

FindBugs

Es una herramienta desarrollada en Java por la Universidad de Maryland. Analiza código Java y permite la personalización de los aspectos utilizados para el análisis. Se puede integrar al IDE de programación Eclipse.

Puntuación: tiene buena puntuación en SourceForge (77 votos positivos y 4 votos negativos – 95% recomendado). Aparece dentro de las primeras posiciones en los resultados de las búsquedas de Google para herramientas de análisis estático de código. Se encuentra en uno (1) de los listados mencionados en la sección 7.1 de este capítulo.

Sitio oficial: http://findbugs.sourceforge.net/

W3C – Markup Validation Service

Es un servicio on-line que ofrece la W3C. Ayuda a validar documentos en HTML, XHTML, MathML, SMIL y SVG. Este validador sigue los siguientes estándares: ISO/IEC 15445 e ISO 8879.

Puntuación: Aparece dentro de las primeras posiciones en los resultados de la búsqueda de Google. Además está desarrollado con ayuda de la fundación Mozilla y con el apoyo de las donaciones de la comunidad.

Sitio oficial: http://validator.w3.org/

W3C – CSS Validation Service

Es un servicio on-line que ofrece la W3C. Ayuda a validar CSS. Puede utilizarse on-line o bien como un programa desktop en Java. Esta herramienta permite encontrar los siguientes errores: errores comunes cometidos en los archivos CSS, usos incorrectos, riesgos en cuanto a la usabilidad, etc.

Puntuación: Aparece dentro de las primeras posiciones en los resultados de las búsquedas de Google. Además está desarrollado con ayuda de la fundación Mozilla y con el apoyo de las donaciones de la comunidad.

Sitio oficial: http://jigsaw.w3.org/css-validator/

Las siguientes tablas muestran las planillas para medir la madurez correspondiente a las herramientas mencionadas:

Datos generales

Herramienta	Versión	Fecha de inicio del proyecto	Licencia	Plataforma	Tipo de Interfaz	Lenguaje
Yasca	V 2.21 Noviembre 2010	Febrero 2008	BSD, GNU GPL y GNU LGPL	Independiente	Consola	JAVA y PHP
Findbug	V 2.0.0 Diciembre 2011	Diciembre 2003	GNU LGPL	Independiente	Consola – GUI[*1]	JAVA
W3C – Markup Validation Service	S/D	S/D	W3C Software license[*2]	Independiente	Web	HTML / JS
W3C – CSS Validation Service	S/D	S/D	W3C Software license[*2]	Independiente	Web	JAVA

[*1] Puede añadirse como plugin al eclipse

[*2] http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231

Tabla 4: Plantilla para la selección de herramientas para pruebas estáticas de código.

Criterios de documentación

Herramienta	Guía de instalación	Manual de instalación	Preguntas frecuentes	Soporte Online			Adicional
				Foro	Lista de mail	Blog	
Yasca	SI	SI	NO	SI	NO	SI	S/D
Findbug	SI	SI	SI	SI	SI	SI	S/D
W3C – Markup Validation Service	SI	SI	SI	NO	SI	NO	S/D
W3C – CSS Validation Service	SI	SI	SI	NO	SI	NO	S/D

Criterios de madurez

Herramienta	Inicio del proyecto	Grado de actualización	Actividad en lanzamientos	Actividad en reportes de errores
Yasca	B	MB	B	M (poco movimiento. Último creado diciembre 2010 y último cerrado en mayo 2010)
Findbug	MB	B	MB	MB (último movimiento en marzo 2012)
W3C – Markup Validation Service	S/D	S/D	S/D	B (Poco movimiento. Última creada octubre 2011)
W3C – CSS Validation Service	S/D	S/D	S/D	B (Poco movimiento. Última creada octubre 2011)

7.2.3 Herramientas para pruebas de unidad

Este tipo de herramientas son dependientes del lenguaje de programación utilizado. Por tal motivo las hemos clasificado de acuerdo al lenguaje de programación que soporta.

En esta tesina se evaluaron herramientas sólo para los lenguajes de aplicaciones Web más populares tales como: Java, JavaScript, PHP y Python, y sólo presentaremos una herramienta (la más popular) para cada lenguaje, dado que no es el objetivo de esta tesina como mencionamos anteriormente.

Herramienta para el lenguaje Java

JUnit

Es un framework desarrollado en Java. Se pueden ejecutar varias pruebas al mismo tiempo y es fácil de interpretar los resultados. JUnit proporciona herramientas para definir el conjunto de pruebas a ejecutar y luego muestra los resultados.

Puntuación: tiene muy buena puntuación en SourceForge (642 votos positivos y 277 votos negativos – 70% recomendado). Aparece dentro de las primeras posiciones en los resultados de las búsquedas de Google para herramientas para pruebas unitarias para el lenguaje Java. Se encuentra en los tres (3) listados mencionados en la sección 7.1 de este capítulo.

Sitio oficial: http://www.junit.org/

Herramienta para el lenguaje JavaScript

JsUnit

Es un framework desarrollado en JavaScript. Incluye una plataforma para la automatización de la ejecución de las pruebas en varios navegadores y múltiples máquinas ejecutando en diferentes sistemas operativos.

Puntuación: tiene pocos votos en SourceForge (16 votos positivos y 1 votos negativos – 94% recomendado). Aparece dentro de las primeras posiciones en los resultados de las búsquedas de Google para herramientas para pruebas unitarias para el lenguaje JavaScript. Se encuentra en los tres (3) listados mencionados en la sección 7.1 de este capítulo.

Sitio oficial: http://www.jsunit.net/ y https://github.com/pivotal/jsunit

Herramienta para el lenguaje PHP

PHPUnit

Es un framework que se ejecuta por línea de comandos, se integra a diferentes IDEs de programación como Eclipse, NetBeans, PhpEdit. También se lo puede integrar aun proyecto Symfony utilizando el plugin sfPhpUnitPlugin, lo que facilita escribir y ejecutar las pruebas (http://www.symfony-project.org/plugins/sfPhpunitPlugin).

Es fácil de ejecutar y de analizar los resultados.

Puntuación: no esta disponible en SourceForge. Aparece dentro de las primeras posiciones en los resultados de las búsquedas de Google para herramientas para pruebas unitarias para el lenguaje PHP. Se encuentra en dos (2) de los listados mencionados en la sección 7.1 de este capítulo.

Sitio Oficial: https://github.com/sebastianbergmann/phpunit

Herramienta para el lenguaje Python

PyUnit

Es la versión en Python del JUnit, forma parte de la Librería Estándar de Python (versión 2.1).

Permite escribir pruebas fácilmente y se pueden ejecutar en modo texto o GUI.

La documentación proporciona un “cookbook” para escribir las pruebas, instrucciones de uso, problemas conocidos, planes de futuro, y enlaces a fuentes de información relacionada.

Puntuación: tiene pocos votos en SourceForge (4 votos positivos y 0 voto negativo – 100% recomendado). Aparece dentro de las primeras posiciones en los resultados de las búsquedas de Google para herramientas para pruebas unitarias para el lenguaje Python. Se encuentra en dos (2) de los listados mencionados en la sección 7.1 de este capítulo.

Sitio oficial: http://pyunit.sourceforge.net/

Las siguientes tablas muestran las planillas para medir la madurez correspondiente a las herramientas mencionadas:

Datos generales

Herramienta	Versión	Fecha de inicio del proyecto	Licencia	Plataforma	Tipo de Interfaz	Lenguaje
JUnit	V 4.10 Septiembre 2010	S/D	Common Public License – V 1.0	Independiente	Consola	JAVA
JsUnit	V 2.2 Noviembre 2009	2001	GNU GPL	Independiente	Web	JavaScript
PHPUnit	V 3.6 Diciembre 2011	S/D	GNU GPL	Independiente	Consola	PHP
PyUnit	V 1.4.1 Agosto 2001	1999	Pytho License (CNRI Python)	Independiente	GUI o texto	Python

Criterios de documentación

Herramienta	Guía de instalación	Manual de instalación	Preguntas frecuentes	Soporte Online			Adicional
				Foro	Lista de mail	Blog	
JUnit	NO	NO	SI	NO	SI	NO	JavaDoc Cookbok
JsUnit	SI	SI	NO	NO	NO	NO	Documentación online y dentro de la herramienta
PHPUnit	SI	SI	NO	NO	SI	NO	S/D
PyUnit	SI	SI	NO	NO	SI	NO	S/D

Criterios de madurez

Herramienta	Inicio del proyecto	Grado de actualización	Actividad en lanzamientos	Actividad en reportes de errores
JUnit	S/D	B	B	MB (Último movimiento octubre 2011)
JsUnit	B	R	M	N/A
PHPUnit	S/D	MB	MB	MB (Último movimiento marzo 2012)
PyUnit	B	M	M	M (Poco movimiento. Última cerrado junio 2005)

Tabla 5: Plantilla para la selección de herramientas para pruebas de unidad.

7.2.4 Herramientas para pruebas de rendimiento

Este tipo de herramientas deben permitir realizar pruebas de carga y de rendimiento sobre diferentes tipos de servidores Web, de correo, sobre HTTP y HTTPS, bases de datos, etc.

JMeter

Es una aplicación de escritorio desarrollada en Java, diseñada para medir el rendimiento y el comportamiento de los sistemas ante las pruebas de sobrecarga.

Se puede utilizar para probar el rendimiento tanto de los recursos estáticos como dinámicos (archivos, Servlets, scripts de Perl, Java Objects, bases de datos y consultas, servidores FTP, etc.). Puede ser utilizado para simular una sobrecarga en un servidor, una red o un objeto, para poner a prueba su resistencia o para analizar el rendimiento global para diferentes tipos de carga. Puede usarse para hacer un análisis gráfico de rendimiento o para probar su servidor / script / comportamiento del objeto con sobrecargas concurrentes.

Puntuación: no esta disponible en SourceForge. Aparece dentro de las primeras posiciones en los resultados de las búsquedas de Google para herramientas para pruebas de rendimiento. Se encuentra en los tres (3) listados mencionados en la sección 7.1 de este capítulo. Además pertenece al proyecto Apache Jakarta.

Sitio oficial: http://jakarta.apache.org/jmeter/

OpenSTA

Es un conjunto de herramientas que tiene la capacidad de realizar secuencias de comandos HTTP y HTTPS para pruebas de sobrecarga, para medir el rendimiento de aplicaciones en plataformas Win32. Permite captar las peticiones del usuario generadas en un navegador Web, luego guardarlas, y poder editar para su posterior uso.

Puntuación: tiene pocos votos en SouceForge (18 votos positivos y 3 votos negativos – 86% recomendado). Aparece dentro de las primeras posiciones en los resultados de las búsquedas de Google para herramientas para pruebas de rendimiento. Se encuentra en los tres (3) listados mencionados en la sección 7.1 de este capítulo.

Sitio oficial: http://opensta.org/

WebLoad

Permite realizar pruebas de rendimiento, a través de un entorno gráfico en el cual se pueden desarrollar, grabar y editar script de pruebas.

Puntuación: tiene pocos votos en SourgeForce (17 votos positivos y 15 votos negativos – 53% recomendado). Aparece dentro de las primeras posiciones en los

resultados de las búsquedas de Google para herramientas para pruebas de rendimiento. Se encuentra en los tres (3) listados mencionados en la sección 7.1 de este capítulo.

Sitio oficial: http://www.webload.org/

Las siguientes tablas muestran las planillas para medir la madurez correspondiente a las herramientas mencionadas:

Datos generales

Herramienta	Versión	Fecha de inicio del proyecto	Licencia	Plataforma	Tipo de Interfaz	Lenguaje
JMeter	V 2.6 Enero 2012	Diciembre 1998	Apache License V 2.0	Independiente	GUI	JAVA
OpenSTA	V 1.4.4 Octubre 2007	Febrero 2001	GNU GPL	Windows	GUI	C++
WebLoad	V 8.1.0 Octubre 2007	Febrero 2007	GNU GPL y Professiona (no es libre)	Windows	GUI	C++ y JAVA

Criterios de documentación

Herramienta	Guía de instalación	Manual de instalación	Preguntas frecuentes	Soporte Online			Adicional
				Foro	Lista de mail	Blog	
JMeter	NO	SI	SI	SI	SI	NO	Wiki JavaDocs (API)
OpenSTA	NO	SI	SI	NO	SI	NO	S/D
WebLoad	NO	NO	NO	NO	SI	NO	S/D

Criterios de madurez

Herramienta	Inicio del proyecto	Grado de actualización	Actividad en lanzamientos	Actividad en reportes de errores
JMeter	B	B	B	MB (último movimiento marzo 2012)
OpenSTA	B	R	R	M (Poco movimiento último cerrado octubre 2006 y ultimo abierto febrero 2012)
WebLoad	R	M	M	N/A

Tabla 6: Plantilla para la selección de herramientas para pruebas de rendimiento.

7.2.5 Herramientas para pruebas funcionales

A continuación daremos una breve explicación sobre las herramientas analizadas en la presente tesina.

7.2.5.1 Categoría 1 – Herramientas para pruebas funcionales: estas herramientas deben validar partes de las aplicaciones, pueden simular la navegación del usuario, realizar peticiones y comprobar las respuestas.

SfBrowser (Symfony)

Es una herramienta integrada al framework de desarrollo Symfony.

En Symfony, las pruebas funcionales se ejecutan a través de un navegador especial, implementado por la clase sfBrowser. Actúa como un navegador adaptado a la aplicación y conectado directamente a él, sin necesidad de un servidor Web. Permite el acceso a todos los objetos de Symfony antes y después de cada solicitud.

Con SfBrowser, cada escenario se puede reproducir automáticamente una y otra vez mediante la simulación de la experiencia que tiene un usuario en un navegador.

Sitio oficial: http://www.symfony-project.org/

Selenium

Es un conjunto de herramientas para automatizar pruebas de aplicaciones Web. Puede ser controlado por muchos lenguajes de programación y frameworks de pruebas.

Selenium tiene distintas herramientas:

1) Selenium IDE es un complemento de Firefox que graba clics, tipeo de teclado y otras acciones para hacer una prueba, que se puede reproducir en el navegador
2) Selenium Remote Control (RC) ejecuta las pruebas en múltiples navegadores y plataformas. Se puede personalizar por ejemplo para distintos lenguajes.
3) Selenium Grid extiende de Selenium RC para distribuir las pruebas en varios servidores, y así se ahorra tiempo al ejecutar las pruebas en paralelo.

En la presente tesina sólo se evaluará el Selenium IDE.

Puntuación: no esta disponible en SourceForge. Aparece dentro de las primeras posiciones en los resultados de las búsquedas en Google para herramientas de pruebas funcionales. Se encuentra en dos (2) de los listados mencionados en la sección 7.1 de este capítulo.

Sitio oficial: http://seleniumhq.org/

Badboy

Badboy trabaja monitoreando las actividades del explorador Web Internet Explorer mientras se navega, grabando los eventos para que luego se puedan reproducir. Tiene un plugin para FireFox (JBadboy V. 0.1 de noviembre 2008), es una versión alfa, que sólo cuenta con un subconjunto de las herramientas de Badboy.

Ofrece algunas características elementales pruebas de estrés, pero al no ser muy completo, se integra con JMeter (aunque con algunas limitaciones dado que algunas de las características de ambos no son exactamente iguales).

Está diseñado para soportar la grabación en sitios y aplicaciones Web que utilizan AJAX, y luego son capaces de reproducirlos.

Puntuación: no esta disponible en SourceForge. Se encuentra en dos (2) de los listados mencionados en la sección 7.1 de este capítulo.

Sitio oficial: http://www.badboy.com.au/

Las siguientes tablas muestran las planillas para medir la madurez correspondiente a las herramientas mencionadas:

Datos generales

Herramienta	Versión	Fecha de inicio del proyecto	Licencia	Plataforma	Tipo de Interfaz	Lenguaje
SfBrowser	V 1.4.17 Marzo 2012	Octubre 2005	Creative Commons Attribution-No Derivative Works 3.0 Unported License	Independiente	N/A	PHP
Selenium IDE	V 1.7.2 Marzo 2012	Junio 2006	Apache 2.0	Independiente	GUI	S/D
Badboy	V 2.1 Diciembre 2011	S/D	Propia basada en GNU LGPL	Windoews	GUI	S/D

Criterios de documentación

Herramienta	Guía de instalación	Manual de instalación	Preguntas frecuentes	Soporte Online			Adicional
				Foro	Lista de mail	Blog	
SfBrowser	SI	SI	SI	SI	NO	SI	API Libros
Selenium IDE	SI	SI	NO	SI	SI	SI	Grupos en Google Wiki
Badboy	NO	SI	SI	SI	Tiene e-mail	NO	Guía para el desarrollador

Criterios de madurez

Herramienta	Inicio del proyecto	Grado de actualización	Actividad en lanzamientos	Actividad en reportes de errores
SfBrowser	MB	B	B	MB (Proyecto Symfony último movimiento marzo 2012)
Selenium IDE	MB	MB	B	MB (Ultimo reporte marzo 2012)
Badboy	S/D	MB	B	N/A

Tabla 7: Plantilla para la selección de herramientas para pruebas funcionales.

7.2.5.2 Categoría 2 – Herramientas chequeadoras de enlaces: estas herramientas indican si existen enlaces rotos o muertos en nuestra aplicación, también si no existen imágenes o archivos, etc.

LINK Checker W3C

Esta herramienta se encuentra disponible On-Line y en versión CGI. La misma lee un documento HTML o XHTML o una hoja de estilos CSS y extrae de una lista que referencia enlaces. Se puede comprobar de forma recursiva una parte de un sitio Web, el número de documentos que pueden ser chequeados recursivamente es limitado.

Puntuación: está desarrollado con ayuda de la fundación Mozilla y con el apoyo de las donaciones de la comunidad. Aparece dentro de las primeras posiciones en los resultados de las búsquedas de herramientas para pruebas funcionales para chequeadores de enlaces.

Sitio oficial: http://validator.w3.org/checklink/

DRKSpider

Es una aplicación de escritorio desarrollada en C++. Permite la navegación por enlaces internos, externos, imágenes, .css y otros archivos. Genera como resultado un árbol jerárquico con los enlaces del sitio en prueba, con información detallada de cada uno de los enlaces encontrados.

En marzo de 2012 se lanzó una versión desarrollada en Java para que la aplicación funcione en cualquier sistema operativo.

Puntuación: no esta disponible en SourceForge, pero la incorporamos al listado dado que se encontraba en uno (1) de los listados mencionados en la sección 7.1 de este capítulo.

Sitio oficial: http://www.drk.com.ar/spider.php

Link Evaluator

Es un plugin para Firefox. Sólo procesa la página actual y no realiza un análisis en profundidad, el resultado se puede visualizar los resultados en la misma página resaltando en distintos colores el estado de los enlaces.

Puntuación: no esta disponible en SourceForge, pero la incorporamos al listado dado que se encontraba en uno (1) de los listados mencionados en la sección 7.1 de este capítulo.

Sitio oficial: https://addons.mozilla.org/es-ES/firefox/addon/4094/ y
http://evaluator.oclc.org/

Las siguientes tablas muestran las planillas para medir la madurez correspondiente a las herramientas mencionadas:

Datos generales

Herramienta	Versión	Fecha de inicio del proyecto	Licencia	Plataforma	Tipo de Interfaz	Lenguaje
LINK Checker W3C	V 4.81 Octubre 2011	Agosto 1998	GNU GPL	Independiente (procesamiento local y remoto)	Web	Perl
DRK Spider	V 3.2 Marzo 2012	S/D	GNU GPL	Independiente (hay un .exe y una aplicación Java)	GUI	C++ y JAVA
Link Evaluator	V 0.9.9.7 Mayo 2009	Diciembre 2006	Apache 2.0	Independiente	GUI	S/D

Criterios de documentación

Herramienta	Guía de instalación	Manual de instalación	Preguntas frecuentes	Soporte Online		Adicional
				Foro	Lista de mail	Blog	
LINK Checker W3c	SI	SI	NO	NO	SI	NO	S/D
DRK Spider	SI	NO	NO	SI	NO	NO	S/D
Link Evaluator	SI	SI	NO	NO	NO	NO	S/D

Criterios de madurez

Herramienta	Inicio del proyecto	Grado de actualización	Actividad en lanzamientos	Actividad en reportes de errores
LINK Checker W3C	MB	MB	S/D	MB (Ultimo movimiento febrero 2012)
DRK Spider	S/D	S/D	S/D	S/D
Link Evaluator	B	B	R	S/D

Tabla 8: Plantilla para la selección de herramientas para pruebas funcionales – Chequeadoras de enlaces.

7.3 Resumen.

En este capítulo se presentaron un conjunto de herramientas utilizadas para el apoyo de las tareas de testing. Las mismas se ponderaron según el cálculo propuesto al inicio de éste capítulo, según la puntuación de SourceForce, la cantidad de descargas y el ranking de los resultados de las búsquedas de Google, luego se completaron las tablas generadas en el capítulo 4, para poder realizar una comparación más objetiva de las mismas, y así permitir la selección de una herramienta más adecuada para la aplicación en la etapa de testing de la presente tesina.

7.4 Referencias

[1] http://www.opensourcetesting.org/

[2] http://testingfaqs.org/

[3] http://www.softwareqatest.com/

[4] http://http://sourceforge.net/

[5] Software quality: state of the art in management, testing, and tools, Martin Wieczorek, Dirk Meyerhoff – 2001.

Capítulo 8: Caso de estudio: Sistema de Gestión de Alumnos

8.1 Introducción

Para el análisis de las herramientas aplicadas en la presente tesina, se utilizó un sistema de gestión de alumnos para colegios secundarios llamado Kimkëlen, desarrollado en Symfony (Framework de desarrollo en PHP) por el equipo de desarrollo del CeSPI - UNLP. El mismo cuenta con una licencia propia de la UNLP, y actualmente se encuentra en la versión 2.6.10

El sistema Kimkëlen permite administrar los alumnos del colegio, registrar las calificaciones y la asistencia y organizar otros temas relacionados con la dinámica de un colegio secundario. El funcionamiento de Kimkëlen es interactivo y on-line, lo que facilita el acceso y la consulta inmediata por parte de todos los usuarios, sean docentes o no docentes y también el acceso de alumnos para realizar consultas de su historia académica.

En la administración del sistema se llevan a cabo las altas, bajas y modificaciones de los datos referidos a los alumnos, docentes, planes de estudio, etc., y cuenta con la lógica de inscripciones de los alumnos a las diferentes carreras que se estudian en el colegio, a las materias, notas, faltas, etc.

El sistema esta siendo usado en el colegio Liceo Víctor Mercante de la UNLP y por la Escuela Agraria de la UBA. A partir de este año (2012) el sistema Kimkëlen comenzará a ser utilizado por el Bachillerato de Bellas Artes y el colegio Nacional ambos pertenecientes a la UNLP y por los colegios Pellegrini y Nacional de la UBA.

A continuación se muestran algunas pantallas del sistema Kimkëlen:

Figura 9: página de inicio de Kimkëlen.

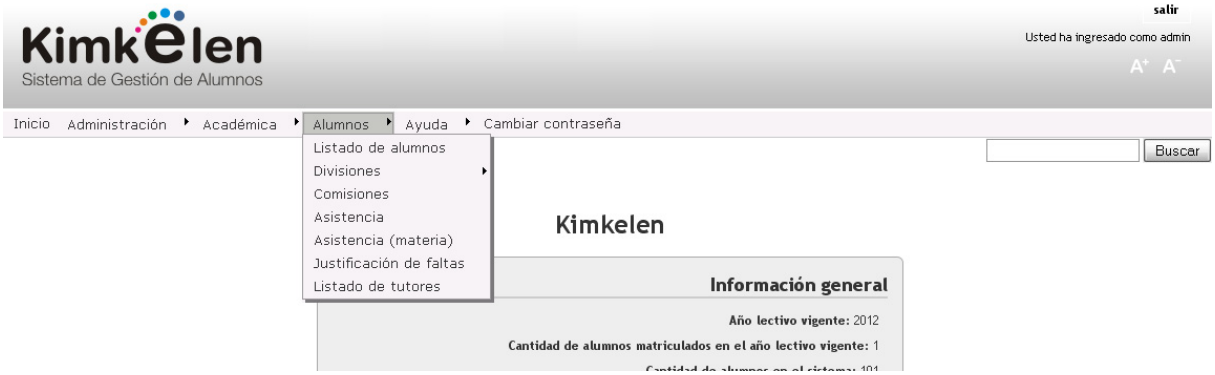


Figura 10: pantalla del menú de alumnos.

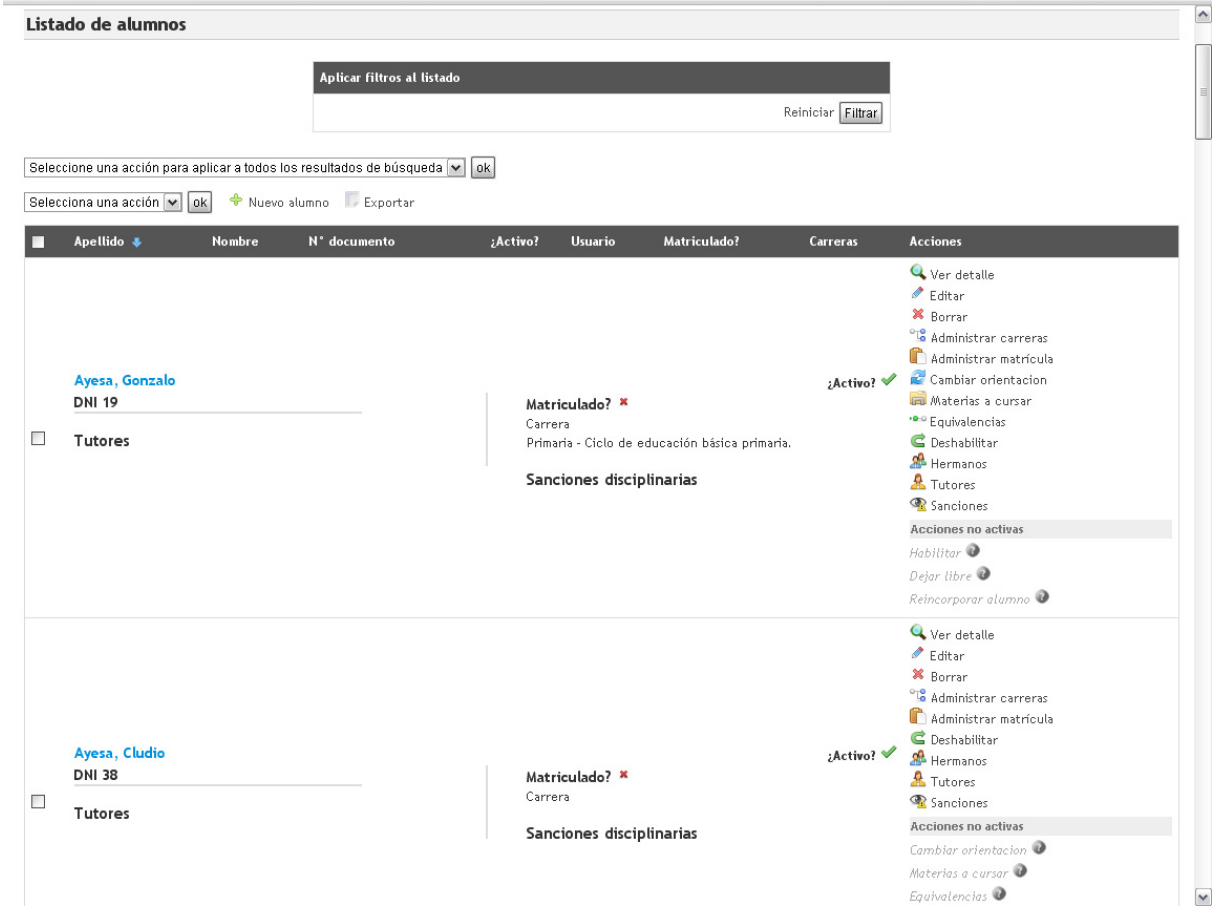


Figura 11: pantalla del listado de alumnos.

8.2 Características de los servidores

El sistema Kimkëlen se instaló en 2 servidores diferentes, uno para las pruebas de unidad en una PC “local” y el otro en un servidor Web dentro de la red de universidad para las pruebas de rendimiento.

8.2.1 Características del primer servidor:

- **PC física**: Intel(R) Core(TM) i3 CPU M 350 @ 2.27GHz 2 GB RAM.

La misma tiene instalada una (1) máquina virtual en la cual se realizaron las pruebas unitarias con las siguientes características:

- Máquina virtual con 512 MB de RAM asignados.
- Sistema Operativo Debian GNU/Linux 6.0 – Lihuen.
- Disco de tamaño variable.

8.2.2 Características del segundo servidor:

- **Servidor físico**: 16 x Intel(R) Xeon(R) CPU E5620 @ 2.40GHz 32 GB RAM.

El mismo tiene instalada una (1) máquina virtual en la cual se realizaron las pruebas de rendimiento con las siguientes características:

- En un principio el servidor virtual con 512 MB de RAM asignados.
- Luego se aumento a 2 GB de RAM.
- Sistema operativo Debian GNU/Linux 6.0.

8.3 Un caso de estudio

En el capítulo anterior hemos analizado una gran cantidad herramientas FLOSS para pruebas de software. Pero nos es imposible mostrar la utilización de todas ellas en la presente tesina, es por ello que se decidió mostrar ejemplos de uso de herramientas en el sistema Kimkëlen para dos (2) tipos de prueba.

La elección de este tipo de pruebas se basó en la las características y utilización del sistema.

Se decidió realizar pruebas de unidad sobre el módulo más importante del sistema, el módulo *Student*, el cual cuenta con la funcionalidad más crítica. Como se mencionó anteriormente este tipo de pruebas son las primeras a ser ejecutadas y son las que encuentran los errores al inicio del desarrollo, lo que significa que el costo de resolución de dichos errores es menor que si se encuentra en etapas posteriores.

En segundo lugar se decidió realizar pruebas de rendimiento, ya que el sistema puede ser accedido por varios usuarios a la vez, lo que implica que se pueda estar cargando las asistencias de los alumnos o las notas de los exámenes, o quizás también los alumnos estén accediendo a ver su boletín. Es necesario realizar pruebas para observar como se comporta tanto el servidor como el sistema ante un acceso masivo.

En los siguientes capítulos se detallarán las pruebas realizadas mostrando las herramientas utilizadas, las cuales facilitan las tareas de testing.

Capítulo 9: Pruebas de Unidad

9.1 Introducción

Como mencionamos en capítulos anteriores, una prueba de unidad es aquella que permite examinar cada módulo de manera individual para asegurar su correcto funcionamiento. En este caso en particular el término módulo se refiere a cada función dentro de la clase.

En este capítulo se mostrará el uso de una herramienta para realizar pruebas de unidad, la misma ha sido seleccionada de acuerdo a los criterios expuestos en el Capítulo 7. Se mostrará tanto el uso como la funcionalidad de la misma, en la aplicación de la herramienta en el sistema Kimkëlen, descripto en el Capítulo 8.

9.2 Selección de una herramienta: PHPUnit

Para la selección de la herramienta para pruebas de unidad nos basamos en los criterios de evaluación analizados en el Capítulo 7, donde se realizó la clasificación y ponderación de herramientas FLOSS para este tipo de pruebas. Dado que la aplicación a analizar está desarrollada en PHP, se seleccionó la herramienta PHPUnit.

PHPUnit es una aplicación de consola desarrollada en PHP y puede funcionar en diversas plataformas. La última versión es la V 3.6.10 con fecha 14 de marzo de 2012, esto muestra una alta actividad en el lanzamiento de sus nuevas versiones. Las pruebas realizadas fueron entre los meses de Junio y Julio de 2011, por lo cual se ha utilizado una versión anterior (versión V 3.4.15).

Como se mencionó en el capítulo 7, esta herramienta se encuentra bajo la licencia "*GNU GPL Versión 3*" por lo que es posible compartir y modificar todas las versiones del software, y asegura que sigue siendo libre para todos sus usuarios.

Es importante destacar que el sistema Kimkëlen está desarrollado utilizando el framework Symfony, y este framework provee una herramienta para pruebas unitarias denominada *Lime*. Lime sólo puede ser utilizada en el entorno de Symfony, por lo que se decidió proceder con la utilización de PHPUnit, ya que éste puede ser utilizado para cualquier sistema desarrollado en PHP y nos permitiría analizar la herramienta independientemente del framework de desarrollo utilizado.

Para esta elección, también fue de ayuda conocer la existencia del plug-in *sfPhpunitPlugin* para Symfony, el cual integra ambos frameworks (PHPUnit y Symfony)

permitiendo que la realización de las pruebas de unidad sea más sencilla y de una manera más estándar. Es por esto que decidimos utilizar el plugin de Symfony.

9.2.1 Instalación de PhpUnit

Como requisitos para la utilización de *sfPhpunitPlugin* se necesita tener instalado PhpUnit 3.4.x [1].

Una de las formas de instalación de la herramienta, es a través de un canal PEAR. Para crear la conexión, desde la consola se deben ejecutar los siguientes comandos:

```
pear channel-discover pear.phpunit.de
pear channel-discover components.ez.no
pear channel-discover pear.symfony-project.com
```

Una vez creada la conexión, se puede instalar PHPUnit a través del siguiente comando:

```
pear install phpunit/PHPUnit
```

Esta forma de instalación es independiente del sistema operativo que se maneje. Tanto sobre plataformas Windows como Linux, se trabaja desde las respectivas consolas de comandos.

9.2.2 Instalación de sfPhpunitPlugin

Una vez instalado el PHPUnit, se debe instalar el plugin de Symfony antes mencionado [2]. Para hacerlo, se ejecuta el siguiente comando en la consola:

```
./symfony plugin:install sfPHPUnitPlugin
```

Luego se deben crear los directorios para las pruebas. Esto se realiza con el siguiente comando:

```
./symfony phpunit:init
```

El comando anterior genera 8 directorios y 2 archivos dentro del directorio /test, como se muestra a continuación (en este caso se muestra la estructura en un sistema Linux):

```
      >> phpunit Created dir /var/www/alumnos/test/phpunit
      >> phpunit Created dir /var/www/alumnos/test/phpunit/unit
      >> phpunit Created dir /var/www/alumnos/test/phpunit/functional
      >> phpunit Created dir /var/www/alumnos/test/phpunit/unit/model
      >> phpunit Created dir /var/www/alumnos/test/phpunit/fixtures
      >> phpunit Created dir /var/www/alumnos/test/phpunit/fixtures/unit
      >> phpunit Created dir
/var/www/alumnos/test/phpunit/fixtures/functional
      >> phpunit Created dir
/var/www/alumnos/test/phpunit/fixtures/unit/model
      >> phpunit Generate file
/var/www/alumnos/test/phpunit/BasePhpunitTestSuite.php
      >> phpunit Generate file
/var/www/alumnos/test/phpunit/phpunit/AllTests.php
```

Se crea el directorio raíz "*/phpunit*" para todos las pruebas y fixtures[3] descriptos en la siguiente sección.

En el subdirectorio "*fixtures*" se almacenan todos los fixtures utilizados en las pruebas, más adelante en este capítulo se profundizará en este tema

Se crean tres subdirectorios para las pruebas: "*unit"* (para las pruebas de unidad), "*unit/model*" (para las pruebas de unidad sobre el modelo), *functional*" (para las pruebas funcionales).

Se crea los archivos "*BasePhpunitTestSuite.php*" que es el archivo raíz para todos los proyectos de pruebas, y "*AllTests.php*", el encargado de ejecutar todas las pruebas a través de un comando estándar de PHPUnit.

9.2.3 Convenciones para la definición de las pruebas

PHPUnit tiene algunas convenciones para escribir las pruebas de unidad. A continuación se mencionan las mismas [1]:

1. Las pruebas para una clase *Ejemplo* se escriben en una clase llamada *EjemploTest*.
2. *EjemploTest* hereda (por lo general) de *PHPUnit_Framework_TestCase*. En nuestro caso, como utilizamos el plugin *sfPhpunitPlugin*, hereda de *sfBasePhpunitTestCase*.
3. Si se van a utilizar fixtures para cargar datos en la base de datos de pruebas, se debe indicar en la clase de la prueba que se va a implementar el

[3] Los fixtures son archivos escritos en notación YAML que sirven para organizar los datos a ser cargados en la base de datos.

sfPhpunitFixturePropelAggregator (en nuestro caso utilizamos propel, ya que es el ORM (Object-Relational Mapping) utilizado en el sistema).

4. Las pruebas son métodos públicos llamados *test**. También se puede utilizar la notación `@test` en los métodos para marcarlos como tales.
5. Dentro de los métodos, se encuentras las afirmaciones [3] (aserciones), como por ejemplo: `assertEquals()` utilizado para afirmar que el valor actual se corresponde con un valor esperado.

El siguiente ejemplo muestra la estructura de una prueba de unidad para la clase *Ejemplo*:

```
      class EjemploTest extends sfBasePhpunitTestCase {
        public function testEjemplo1() {
          // En el cuerpo de la función van las aserciones que se utilicen,
estas pueden ser:
          assertTrue(bool $condition[, string $message = ''])
          // Reporta un error "$message" si "$condition" es Falsa en caso
contrario la prueba se ejecuta sin errores.
          assertFalse(bool $condition[, string $message = ''])
          // Reporta un error "$message" si "$condition" es Verdadera en
caso contrario la prueba se ejecuta sin errores.
          assertEquals(mixed $expected, mixed $actual[, string $message =
''])
          // Reporta un error "$message" si las variables $expected y
 $actual no son iguales en caso contrario la prueba se ejecuta sin errores.
          //También se puede utilizar assertNotEquals() que es la aserción
inversa a la anterior.
          assertEmpty(mixed $actual[, string $message = ''])
          // Reporta un error "$message" si $actual no está vacía. En caso
contrario la prueba se ejecuta sin errores.
          //También se puede utilizar assertNotEmpty() que es la aserción
inversa a la anterior.
            }
      }
```

9.2.4 Ejecución de las pruebas

Una vez escrita/s la/s prueba/s, existen varias maneras de ejecutarla/s desde la consola de comandos [2]:

Correr todas las pruebas del proyecto:

```
      ./symfony phpunit:runtest
```

Todas las pruebas que se encuentran en una carpeta:

```
      ./symfony phpunit:runtest unaCarpeta/
```

Todas las pruebas que se encuentran en una carpeta y en subcarpetas:

```
./symfony phpunit:runtest unaCaperta/*
```

Una prueba en particular:

```
./symfony phpunit:runtest unaCarpeta/UnTestCase.class.php
```

También se puede incluir el path absoluto:

```
./symfony phpunit:runtest
/path_del_proyecto/test/phpunit/unaCarpeta/*
```

Un path relativo al proyecto:

```
./symfony phpunit:runtest test/phpunit/unaCarpeta/*;
```

9.2.5 Utilización de Fixtures

Los fixtures son archivos escritos en notación YAML que tienen información que se cargará en la base de datos. Los mismos se utilizan tanto para cargar datos iniciales en el sistema, como para cargar datos en la base de datos de prueba.

A continuación se muestra un ejemplo:

```
Persona: # nombre de la entidad
  Persona_1: # nombre de la variable
    nombre: 'Lopez' # campo 1
    apellido: 'Claudio' # campo 2
    dni: '12345678' # campo 3
  Persona_2: # nombre de la variable
    nombre: 'Perez' # campo 1
    apellido: 'Juan' # campo 2
    dni: '22333444' # campo 3
```

Con la ayuda de dichos fixtures se pueden preparar los escenarios de pruebas, cargando los datos necesarios para las mismas.

Se pueden cargar los fixtures utilizados en las pruebas, con los siguientes métodos [2]:

1. `loadOwn()` – carga el fixture que se encuentra en el directorio `..\test\phpunit\fixtures\models\UnTest.class\ y` puede se usado únicamente por la clase de prueba `UnTest.class`.
2. `loadPackage()` – carga el fixture que se encuentra en el directorio `..\test\phpunit\fixtures\unit` y puede ser usado únicamente por los casos de pruebas que se encuentran en el mismo directorio.

3. `loadCommon()` – carga el fixture que se encuentra en el directorio `..\test\phpunit\fixtures\common` y puede se usado por cualquier caso de prueba.
4. `loadSymfony()` – carga el fixture que se encuentra en el directorio `..\data\fixtures` (directorio estandar de symfony), y puede se usado por cualquier caso de prueba.

La manera de invocar a los fixtures desde el caso de prueba, es a través de la función `_start()` que se ejecuta antes de cada prueba dentro de la clase, como se muestra en el siguiente ejemplo:

```
class User_Test extends sfBasePhpunitTestCase
  implements sfPhpunitFixturePropelAggregator{
protected function _start(){
$this->fixture()
     ->clean() //Vacía toda la BD
     ->loadOwn('users') //Carga el fixture que se encuentra en:
../test/phpunit/fixtures/unit/UserTestCase/users.propel.yml
     ->loadPackage('users') //Carga el fixture que se encuentra en:
../test/phpunit/fixtures/unit/users.propel.yml
     ->loadCommon('users') //Carga el fixture que se encuentra en:
../test/phpunit/fixtures/common/users.propel.yml
     ->loadSymfony('users'); //Carga el fixture que se encuentra en:
../data/fixture/users.propel.yml
}
//Aquí se escriben las pruebas para la clase User
}
```

9.3 Caso de estudio de PhpUnit en el sistema Kimkëlen

Para presentar el uso de esta herramienta, se realizaron diferentes pruebas de unidad sobre diferentes clases del modelo de datos. En particular, en esta sección, se muestra un caso de prueba sobre la clase *Student*. Dicha clase es la más representativa del sistema ya que concentra la mayor cantidad de actividad, desde el alumno (clase `Student`) se disparan las acciones más importantes de la aplicación, entre las que se destacan:

1. Registrar al alumno en una carrera.
2. Saber si el alumno está (o no) inscripto en una carrera.
3. Registrar al alumno en un año escolar (y en un turno).
4. Saber si el alumno está (o no) inscripto en un año escolar.
5. Contar la cantidad de previas que tiene el alumno.
6. Contar la cantidad de materias desaprobadas del alumno en un año escolar.
7. Saber si el alumno aprobó un grupo de materias dadas.

8. Saber si el alumno aprobó una materia dada.
9. Saber si el estudiante está inscripto al menos en una carrera.
10. Contar la cantidad de inasistencias del alumno.
11. Saber si el alumno ha quedado libre.

9.3.1 Creación de las pruebas

Para realizar las pruebas de unidad, se creó un fixture (`students.propel.yml`) con los datos necesarios para poder probar la clase, la Figura 12 muestra un ejemplo de ello:

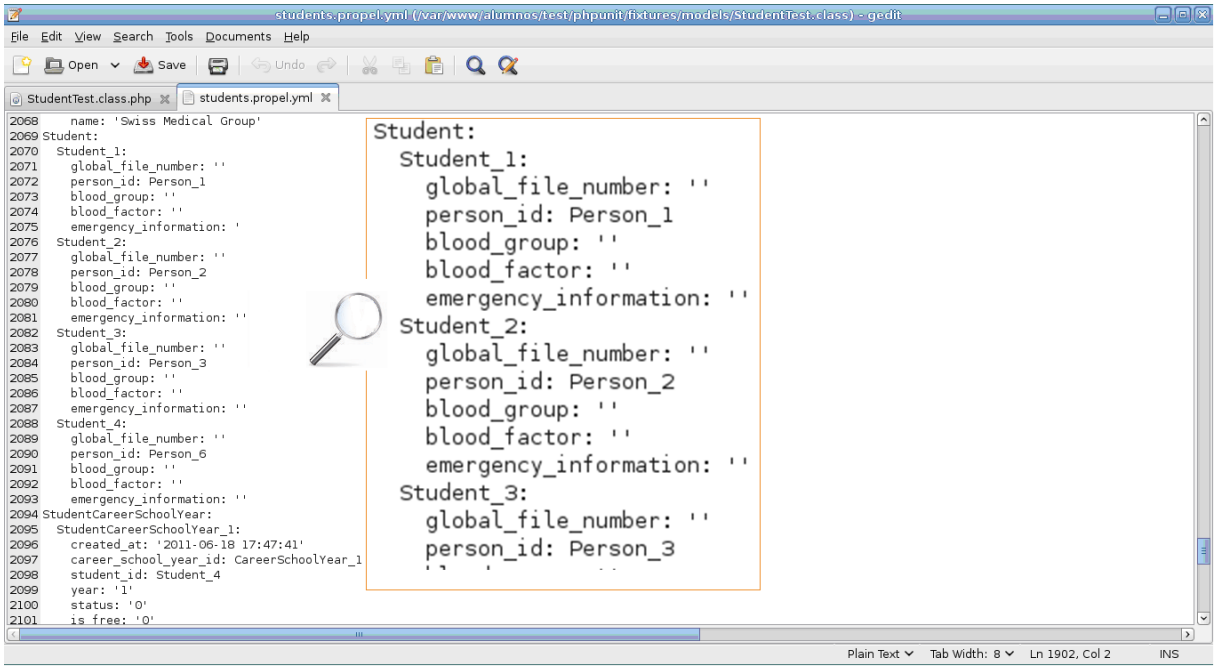


Figura 12: Fixture students.propel.yml.

Se adjunta el archivo `students.propel.yml` en el anexo Pruebas de unidad.

En dicho fixture se tienen los datos necesarios que deben existir en la base de datos para cada una de las pruebas. Existen datos que son comunes a todas las pruebas, como por ejemplo: los usuarios que pueden acceder al mismo, con grupos y permisos, una carrera, un plan de estudios, un año lectivo, materias, estudiantes, etc. A continuación se describen los datos utilizados en cada prueba:

1. *Saber el nombre completo de un alumno*: se prueba la función `getPersonFullName()` y se compara con el nombre del alumno utilizado en la prueba.

2. *Inscribir un alumno en una carrera*: se prueba la función `registerToCareer($carrera)` con un estudiante que no esta inscripto a ninguna carrera y luego se utiliza la función `isRegisteredToCareer($carrera)` la cual devuelve un valor booleano.
3. *Inscribir un alumno en un año escolar*: se prueba la función `registerToSchoolYear($anio_escolar, $turno)` con un estudiante que no esta inscripto en un año escolar y luego se utiliza la función `getIsRegistered($anio_escolar)` la cual devuelve un valor booleano.
4. *Reinscribir un alumno en un año escolar*: se prueba la función `registerToSchoolYear($anio_escolar, $turno)` con un estudiante que ya esta inscripto en un año escolar y luego se utiliza la función `getIsRegistered($anio_escolar)` la cual devuelve un valor booleano. Para esta prueba se utilizan excepciones dado que el modelo de datos no esta contemplando que esta situación pueda suceder, por lo que informa un error.
5. *Saber si un alumno no esta inscripto en un año escolar*: se prueba la función `getIsNotRegistered($anio_escolar)` con un estudiante que no esta inscripto en un año escolar la cual devuelve un valor booleano.
6. *Saber si un alumno puede ser eliminado del sistema*: se prueba la función `canBeDeleted()`, la misma verifica que el estudiante si está inscripto en un curso, ha aprobado una materia o aprobó un curso el mismo no puede ser eliminado. Se probó la función con un estudiante que no está inscripto en ningún curso y otro que sí.
7. *Saber si un alumno puede administrar las materias a las que se puede inscribir*: se prueba la función `canBeManagedForCareerSubjectAllowed()` la cual devuelve un valor booleano. Se probo la función con un estudiante que sí puede y otro que no.
8. *Saber que año cursaba el alumno en un año escolar*: se prueba la función `getCareerYear($anio_escolar)` la cual devuelve el año que cursaba el alumno en un año escolar dado.
9. *Saber cuantas materias previas tiene un alumno*: se prueba la función `countPreviousForCareer($carrera)` la cual devuelve la cantidad de materias previas que adeuda un alumno. Se probó la función con un estudiante que no adeuda materias.
10. *Saber si un alumno puede administrar las equivalencias entre materias*: se prueba la función `canManageEquivalence()` la cual devuelve un valor booleano. Se probó la función con un estudiante que sí puede y otro que no.

11. *Saber la cantidad de materias que un alumno desaprobó en un año escolar*: se prueba la función `countDisapprovedForSchoolYear($anio_escolar)` la cual devuelve la cantidad de materias desaprobadas. Se probó la función con un estudiante que aprobó materias y otro que no.
12. *Saber la cantidad de materias que un alumno aprobó en una carrera*: se prueba la función `hasApprovedCareerSubject()` la cual devuelve la cantidad de materias aprobadas. Se probo la función con un estudiante que aprobó materias y otro que no.
13. *Saber si un alumno es regular*: se prueba la función `canManageRegularity()` la cual devuelve un valor booleano. Se probo la función con un estudiante que sí puede y otro que no.
14. *Saber si un alumno está inscripto en alguna carrera*: se prueba la función `isInscriptedInCareer()` la cual devuelve un valor booleano. Se probó la función con un estudiante que sí está inscripto a una carrera y otro que no.
15. *Saber cuál es el año actual que esta cursando un alumno*: se prueba la función `getCurrentCourseYear()` la cual devuelve el año actual que cursa el alumno. Se probó la función con un estudiante que está inscripto en 1er año.

A continuación se muestra la Figura 13 la clase StudentTest:

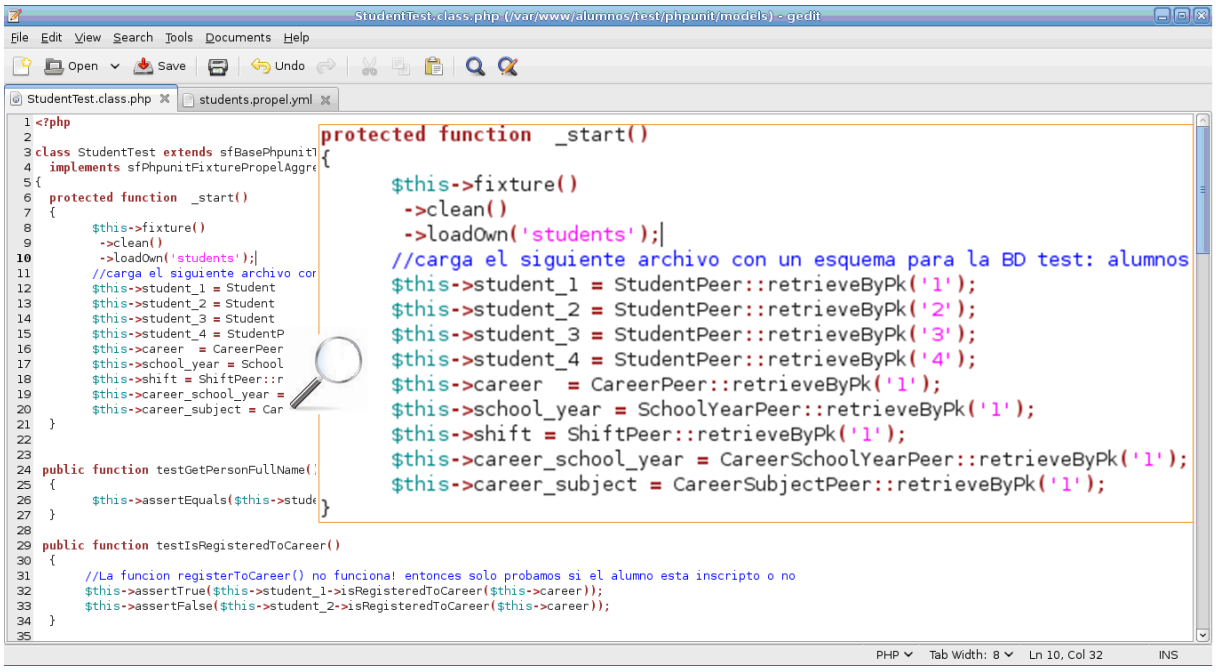


Figura 13: Prueba StudentTest.class.php.

Se adjunta el archivo `StudentTest.class.php` en el anexo Pruebas de unidad.

9.3.2 Análisis de los resultados

La Figura 14 muestra la salida de la ejecución de las pruebas de unidad sobre la clase mencionada, Como puede verse, la misma muestra que se ejecutaron 15 test y 24 aserciones, esto muestra el éxito en la prueba.

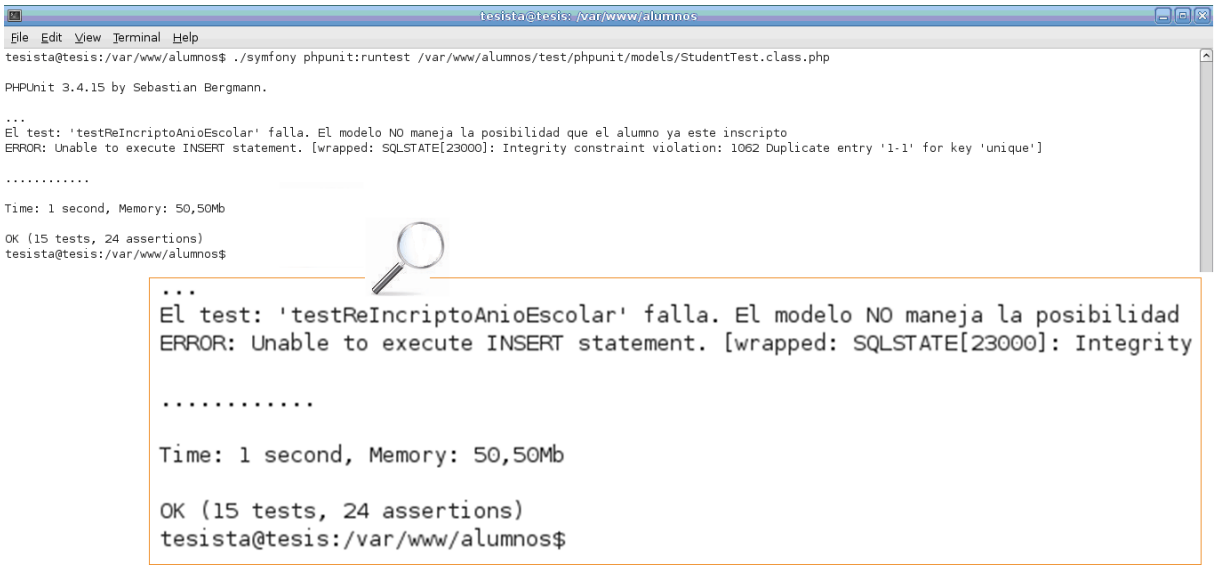


Figura 14: Ejecución de los test.

También se observa que existe un mensaje en el `testReIncriptoAnioEscolar`, esto es porque se intenta reinscribir un alumno nuevamente a un año escolar pero la función del modelo no contempla este caso, entonces se genera una excepción que es capturada en la prueba. La Figura 15 muestra el código de la misma:

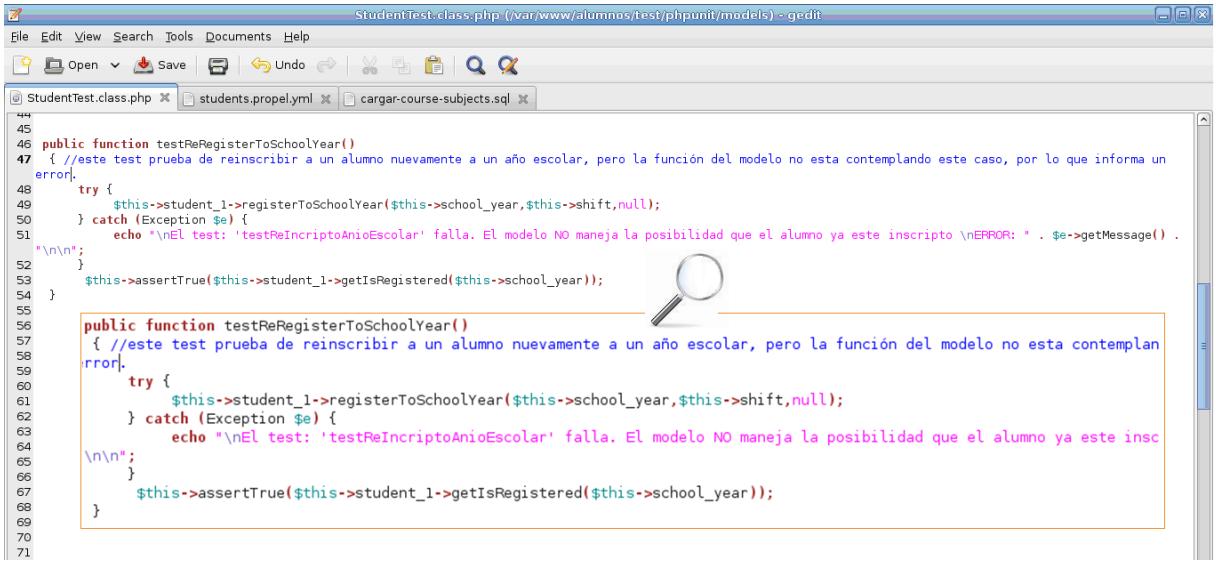


Figura 15: Función testReRegisterToSchoolYear.

9.4 Resumen

Para la utilización de esta herramienta no sólo fue necesario el conocimiento del uso de la misma, sino también de la estructura del código del sistema, para lo cual se invirtió tiempo de aprendizaje dado que el grupo de pruebas no forma parte del grupo de desarrollo. Es recomendable que las pruebas de unidad sean escritas por los desarrolladores o que si el trabajo lo realiza un tester, el mismo se encuentre dentro del grupo de desarrollo, para así poder reducir el tiempo que invierte un testeador en conocer el sistema y su estructura.

Una ventaja de la utilización de esta herramienta, es que existe mucha documentación, tanto en la página oficial, como así también en la comunidad de Symfony, como fuera mencionado en el Capítulo 7.

Otra de las ventajas es que la automatización y ejecución de las pruebas es muy simple. Como así también la posibilidad de extender el caso de prueba ante nuevas funcionalidades en el modelo o cambios en el mismo.

La interpretación de los resultados brindado por la herramienta es simple y fácil de entender.

Como ventaja de las pruebas de unidad, es que al hacerlas de manera organizada y temprana durante el desarrollo evita problemas a futuro y el costo de solucionarlos es menor.

9.5 Referencias

[1] https://github.com/sebastianbergmann/phpunit/

[2] https://github.com/makasim/sfPhpunitPlugin

[3] http://www.phpunit.de/manual/current/en/writing-tests-for-phpunit.html#writing-tests-for-phpunit.assertions

Capítulo 10: Pruebas de Rendimiento

10.1 Introducción

Como mencionamos en capítulos anteriores, una prueba de rendimiento es aquella que permite observar el comportamiento del software frente a la sobrecarga de datos (pruebas de volumen), de usuarios (pruebas de carga) o en una fusión de ambos (pruebas de stress).

En este capítulo se mostrará el uso de una herramienta para realizar pruebas de rendimiento que ha sido seleccionada de acuerdo a los criterios expuestos en el Capítulo 7. Se mostrará el uso y la funcionalidad de la misma, en la aplicación en el sistema Kimkëlen, descripto en el Capítulo 8.

10.2 Pruebas de rendimiento. Definición y Características

Las pruebas de rendimiento realizadas sobre computadoras, redes, software u otros dispositivos, son utilizadas para determinar la velocidad y eficiencia de los mismos. Este procedimiento puede involucrar tanto pruebas cuantitativas, por ejemplo, medir tiempos de respuesta o cantidad en millones de líneas de código, como pruebas cualitativas, en las cuales se evalúa fiabilidad, escalabilidad e interoperabilidad. Estas pruebas de rendimiento pueden ser realizadas a través de herramientas que proveen pruebas de stress, que permiten determinar la estabilidad del sistema [1].

En las pruebas de rendimiento se debe evaluar el grado con que un sistema o componente logra la funcionalidad señalada dentro de las restricciones dadas con respecto al tiempo de proceso. Comprobando el comportamiento del sistema ante determinadas situaciones, por ejemplo, cuánto tarda el servidor en responder ante el acceso de varios usuarios al mismo tiempo (Pruebas de Carga), peticiones de datos en casos extremos o resistencia ante el ingreso de grandes volúmenes de datos (Pruebas de Volumen) y verificar que el sistema sea estable mas allá de los limites especificados, permite verificar la robustez y escalabilidad de la infraestructura de la red (Pruebas de Stress) [2].

Las limitaciones en los tiempos de respuesta de un sitio Web están muy relacionadas a la velocidad del enlace donde se esté “navegando”. Según el autor Jakob Nielsen, en el libro “Usability Engineering” [3], existen tres límites importantes en el tiempo de respuesta [4 y 5]:

1. *0,1 segundo*: es el límite en el cual el usuario siente que esta "*manipulando*" los objetos desde la interfaz de usuario.
2. *1 segundo*: es el límite en el cual el usuario siente que está navegando libremente sin esperar demasiado una respuesta del servidor.
3. *10 segundos*: es el límite en el cual se pierde la atención del usuario, si la respuesta tarda más de 10 segundos se deberá indicar algún mecanismo por el cual el usuario pueda interrumpir la operación.

En nuestro caso particular, este tiempo está condicionado a los siguientes puntos:

- El servidor testeado se encuentra en la misma red en la cual se realizaron las pruebas.
- Velocidad de conexión del servidor.
- Velocidad de conexión del cliente.
- Tiempo en el cual el navegador Web tarda para dibujar la página (tiempo muy pequeño, pero debe ser considerado de acuerdo al diseño de la interfaz).
- Rendimiento de la red en el momento de la prueba.

10.3 Selección de una herramienta: JMeter

Como en el caso de las pruebas de unidad, para la selección de la herramienta para pruebas de rendimiento nos basamos en los criterios de evaluación analizados en el Capítulo 7. Entre las herramientas mencionadas se seleccionó JMeter [6], ya que es la que posee mejor puntuación.

JMeter es una aplicación de escritorio desarrollada en Java, es multiplataforma, y pertenece al proyecto Apache Jakarta. La última versión de esta herramienta es la versión 2.6 con fecha de enero 2012, esto muestra una alta actividad en el lanzamiento de sus nuevas versiones. Las pruebas fueron realizadas entre los meses de agosto y septiembre de 2011, por lo cual se ha utilizado una versión anterior (versión V 2.4 r961953).

Como se mencionó en el capítulo 7, esta herramienta se encuentra bajo la licencia "*Apache License, Version 2.0*" por lo que es posible distribuirlo, modificarlo, y distribuir versiones modificadas del mismo.

JMeter está diseñado para medir el rendimiento y el comportamiento de los sistemas ante las pruebas de sobrecarga. Originalmente fue diseñado para probar las aplicaciones web, pero se ha ampliado a otras funciones de prueba.

10.3.1 Instalación de JMeter

Como requisitos para la instalación de JMeter, se necesita tener JMV 1.5 totalmente compatible o superior.

Según la especificación, JMeter funciona correctamente en los siguientes sistemas operativos, dado que es multiplataforma:

- Unix (Solaris, Linux, etc.)
- Windows (XP, 7, etc)
- OpenVMS Alpha 7.3+

En nuestro caso se instaló en un equipo con sistema operativo Windows XP y SP3.

Para instalar JMeter, simplemente se debe descomprimir el archivo zip / tar en el directorio deseado (el mismo no debe contener espacios en blanco ya que podría llegar a haber problemas (sobre todo con el modo cliente-servidor)). Si el JRE / JDK se encuentre instalado correctamente y que la variables de entorno JAVA_HOME también se encuentre configurada, la herramienta se encuentra disponible para su uso.

Al instalarlo se genera la siguiente estructura de directorios:

```
../jmeter
../jmeter/bin
../jmeter/docs
../jmeter/extras
../jmeter/lib/
../jmeter/lib/ext
../jmeter/lib/junit
../jmeter/printable_docs
```

10.4 Plan de pruebas - Definición y composición

Un plan de pruebas es un conjunto de pasos que JMeter irá ejecutando. El mismo cuenta con uno o más grupos de hilos, controladores lógicos, listeners, temporizadores, afirmaciones y elementos de configuración.

Para las pruebas realizadas se utilizaron los siguientes componentes comunes a todas las pruebas y luego en las pruebas se agregará un componente especifico según sea necesario:

Grupo de hilos: El grupo de hilos es el punto de inicio de cualquier plan de prueba, todos los controladores y muestreadores deben estar bajo un grupo de hilos. Otros elementos, por ejemplo los listeners, se pueden colocar directamente en el plan de pruebas, en cuyo caso se aplicará a todos los grupos de hilos.

El grupo de hilos controla el número de hilos que JMeter utiliza para ejecutar la prueba. Los controles para un grupo de hilos le permiten:

- Establecer el número de hilos
- Establecer el período de ramp-up (*Período de Arranque*: este valor se utiliza para saber cuánto tiempo esperar para que comience la ejecución el siguiente hilo)
- Establecer el número de veces para ejecutar la prueba

Cada hilo ejecutará el plan de pruebas en su totalidad y de forma completamente independiente de los otros hilos. La utilización de múltiples hilos sirve para simular conexiones simultáneas al servidor de la aplicación.

Listeners: permite acceder a la información que se va obteniendo durante la ejecución de la prueba. Entre ellos se encuentran:

- *Agreggate Graph*: permite visualizar los resultados de la prueba realizada de manera precisa. Utiliza más memoria, ya que calcula la mediana y la línea al 90%, para lo cual se requiere que todos los datos estén almacenados. También permite generar gráficos de barras y guardarlos en formato .png. Los datos que se presentan son:
    1. *URL*: etiqueta de la muestra
    2. *#Muestras*: cantidad de hilos utilizados para la URL.
    3. *Media*: tiempo promedio en milisegundos para un conjunto de resultados.
    4. *Mediana*: valor en tiempo del percentil 50.
    5. *Línea de 90%*: máximo tiempo utilizado por el 90% de la muestra, al resto de la misma le llevo más tiempo.
    6. *Min*: tiempo mínimo de la muestra de una determinada URL.
    7. *Max*: tiempo máximo de la muestra de una determinada URL.
    8. *%Error*: porcentaje de requerimientos con errores.
    9. *Rendimiento*: rendimiento medido en los requerimiento por segundo / minuto / hora.
    10. *KB/sec*: rendimiento medido en Kbytes por segundo.

- *Árbol de resultados*: muestra un árbol con todas las respuestas de la muestra, también permite ver el tiempo que tardó en obtener esta respuesta, y algunos códigos de respuesta. Existen varias maneras de ver la respuesta: la opción "texto" permite ver todo el texto que figura en la respuesta, la vista HTML intenta representar la respuesta en HTML, etc.

Todas las pruebas propuestas se ejecutaron de dos maneras, para luego comparar los resultados:

1. Pruebas donde se utilizan hilos concurrentes, es decir que no hay tiempo de espera entre el lanzamiento de un hilo y el siguiente.
2. Pruebas donde se espera un tiempo entre el lanzamiento de un hilo y el siguiente.

Elección de la cantidad de hilos utilizados en las pruebas:

Para la primer prueba solo se hacen peticiones HTTP con acceso la pantalla inicial sin ingreso al sistema, por este motivo se llegaron a tener hasta 200 conexiones, con este número se puede probar la performance del servidor.

En cambio en las pruebas posteriores se utilizan usuarios que ingresan al sistema con nombre de usuario y contraseña, y realizan operaciones en el mismo, con lo que aumenta notablemente la comunicación con el servidor. Es por esto que se redujo la cantidad de hilos a utilizar en las pruebas a 50 usuarios como máximo, siendo este número suficiente para probar la performance del servidor.

10.4.1 Análisis de Resultados

Los resultados se analizarán con algunos de los datos reportados por el componente "*Agreggate Graph"* generando una tabla con la siguiente información*:*

Prueba x.x						
# Usuarios	**# Muestras**	**Media**	**Mediana**	**% Error**	**Rendimiento (seg)**	**Kb/seg**

Tabla 9: Resultados de las pruebas.

Dependiendo del tipo de prueba que se realice si la misma tiene un conjunto de enlaces que visitar, esto determinará la variación entre el #Usuarios y el #Muestras.

Si la prueba no tiene problemas de conexión el %Error va a ser igual a 0.

El rendimiento se utiliza para ver cuántas peticiones/segundo es capaz de manejar el sistema. A medida que aumenta la carga se puede apreciar la variación del rendimiento. Normalmente el rendimiento crece por un tiempo, se mantiene y luego cae.

Luego se realiza un análisis del cálculo del tiempo total utilizado por cada una de las mismas con la siguiente fórmula:

```
Tiempo Total = #Muestras * Media = xx milisegundos
```

Con el tiempo total de las pruebas se puede calcular el tiempo promedio total requerido por cada usuario en la prueba, de la siguiente manera:

```
((Tiempo Total /1000)/60)/cantidad de Thread = yy minutos
```

Los datos obtenidos se mostrarán en una tabla como la siguiente:

Análisis x.x				
# Usuarios	**# Muestras**	**Media**	**Tiempo Total (milisegundos)**	**Tiempo Promedio (minutos)**

Tabla 10: Análisis de los resultados de las pruebas.

10.4.2 Ejecución de las pruebas

Una vez generado el plan de pruebas, se debe seleccione la opción "*Arrancar*" del menú "*Lanzar*" para poder ejecutarlo.

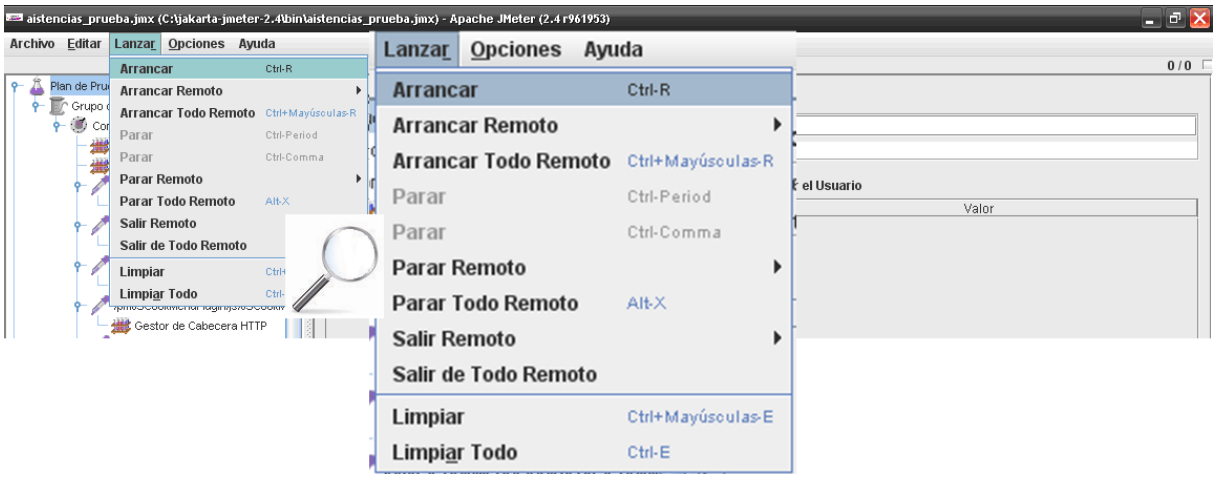


Figura 16: JMeter opción Arrancar.

Cuando JMeter se está ejecutando, se muestra un cuadrado verde pequeño en el extremo derecho debajo de la barra de menú. Los números a la izquierda de la caja verde son el número de hilos activos / número total de hilos. Como se observa en la Figura 17:

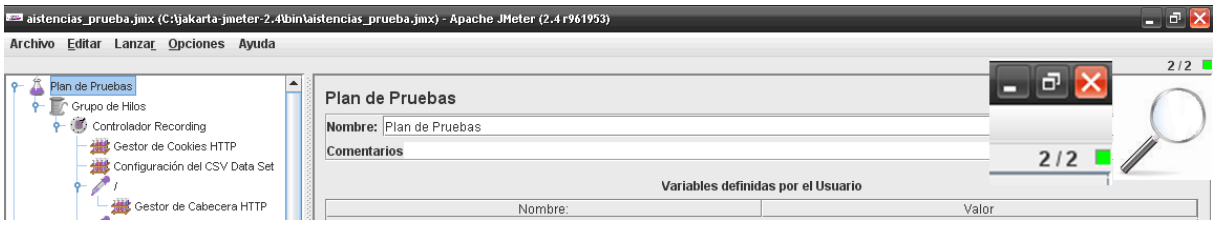


Figura 17: Ejecutando JMeter.

10.5 Caso de estudio de JMeter en el sistema Kimkëlen

10.5.1 Prueba 1: Acceso a la página principal del sistema (“home”)

Como primer prueba se decidió probar el tiempo de respuesta del sistema ante el acceso a la página principal (“*home*") ya que es una prueba básica, esto sirve para tener valores iniciales del tiempo de respuesta para luego comparar con el resto de las pruebas propuestas.

Para ello se configuró a la herramienta con un “*Grupo de Hilos*” para poder simular los accesos de los usuarios.

Al mismo se le agregó el muestreador “*Petición HTTP*” el cual permite el acceso a la página principal del sistema (“*home*”), para ello se ingresó el nombre del servidor en el campo correspondiente (Path):

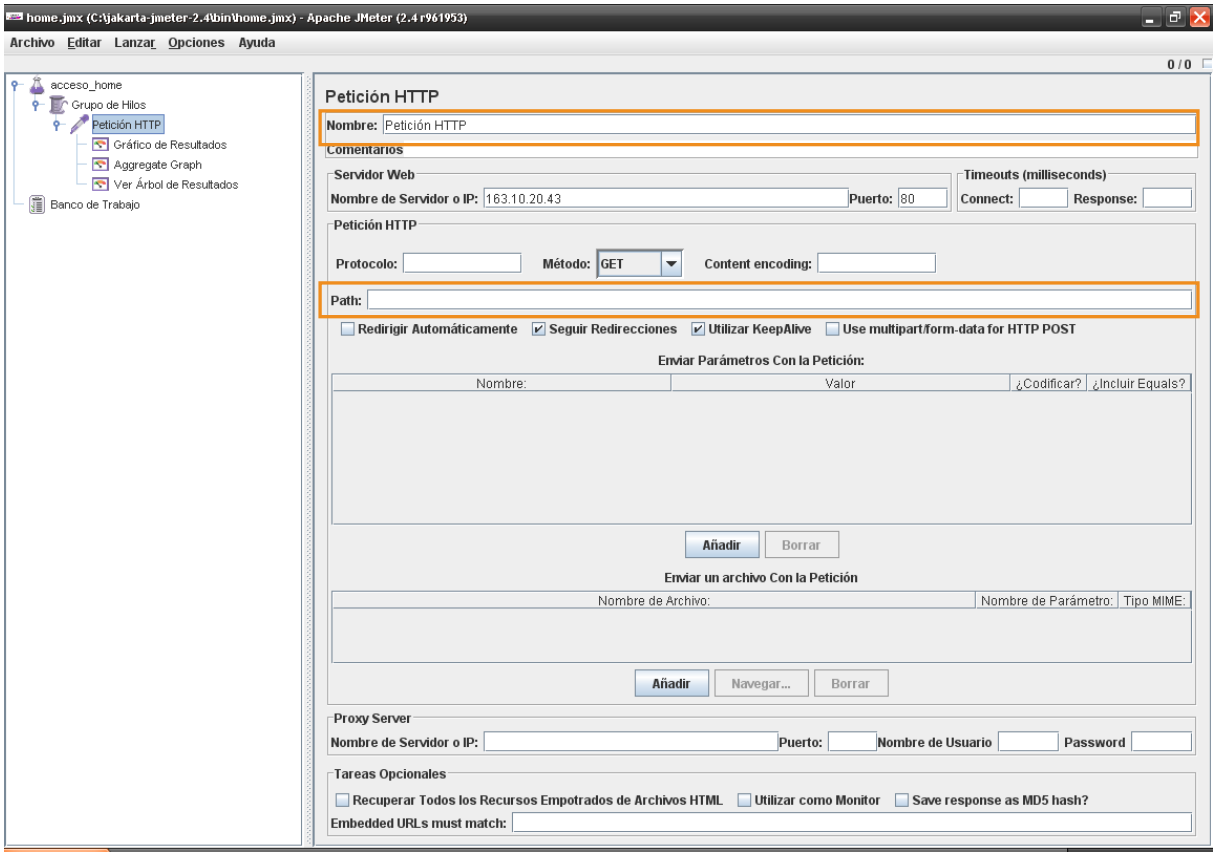


Figura 18: Petición HTTP.

Para poder evaluar los resultados, se agregaron los “*Listeners*” “*Aggregate Graph*” y “*Árbol de resultados*” descriptos anteriormente.

Como se mencionó anteriormente las pruebas se dividieron en dos (2):

Prueba 1.1: Acceso a la página principal del sistema ("home") en la cual se configuraron usuarios concurrentes para las siguientes cantidades: 20, 50, 100 y 200. Los valores totales obtenidos por el componente "*Aggregate Graph*" se muestran en la Tabla 11:

Prueba 1.1-a						
# Usuarios	**# Muestras**	**Media**	**Mediana**	**% Error**	**Rendimiento (seg)**	**Kb/seg**
20	20	6386	3743	0	1,49	4,27
50	50	38101168	2925647	0,56	0,0084	0,01

Tabla 11: Valores correspondientes al acceso concurrente al home.

Nota: Como puede observarse en la tabla anterior el servidor presentó un 56% de errores para 50 usuarios, esto significa que 28 de los hilos ejecutados no pudieron completar la conexión al sistema. A raíz de este problema se decidió aumentar la memoria del servidor apache, pasando de tener 512Mb a 2Gb, ya que el tiempo de respuesta de dicha prueba resulta elevado e inadecuado. A continuación se muestra una captura de pantalla donde se muestra el error reportado en el servidor apache:

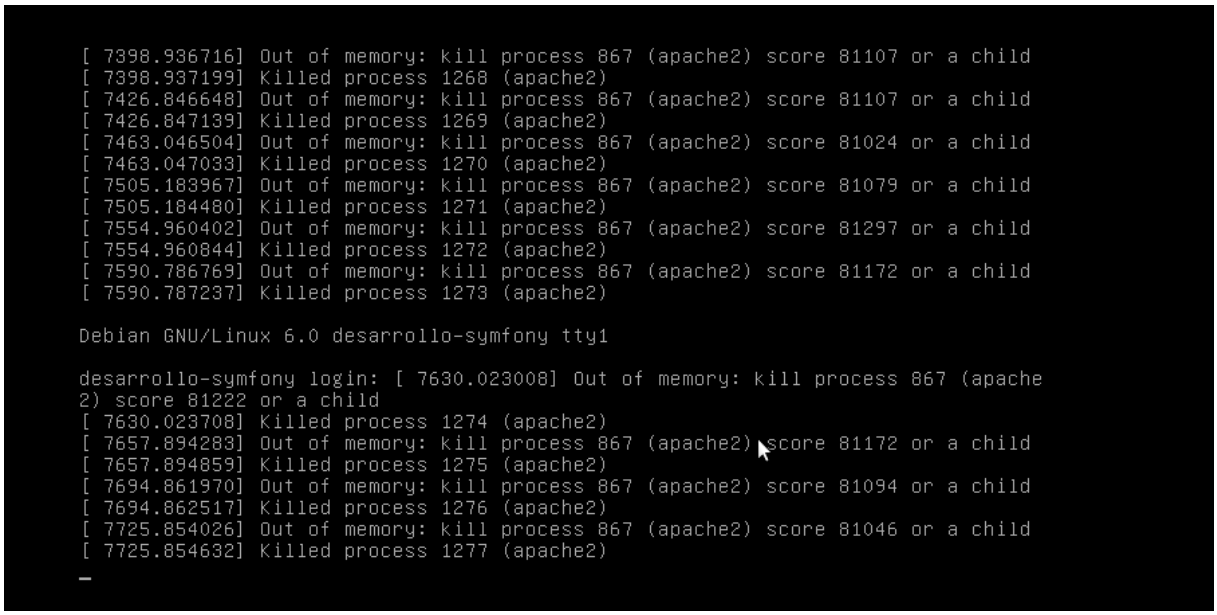


Figura 19: Errores en el Servidor Apache.

Se pueden observar los errores en el *"Árbol de resultados"* el cual marca en color rojo aquellas conexiones que no llegan a conectarse, como se muestra en la Figura 20:

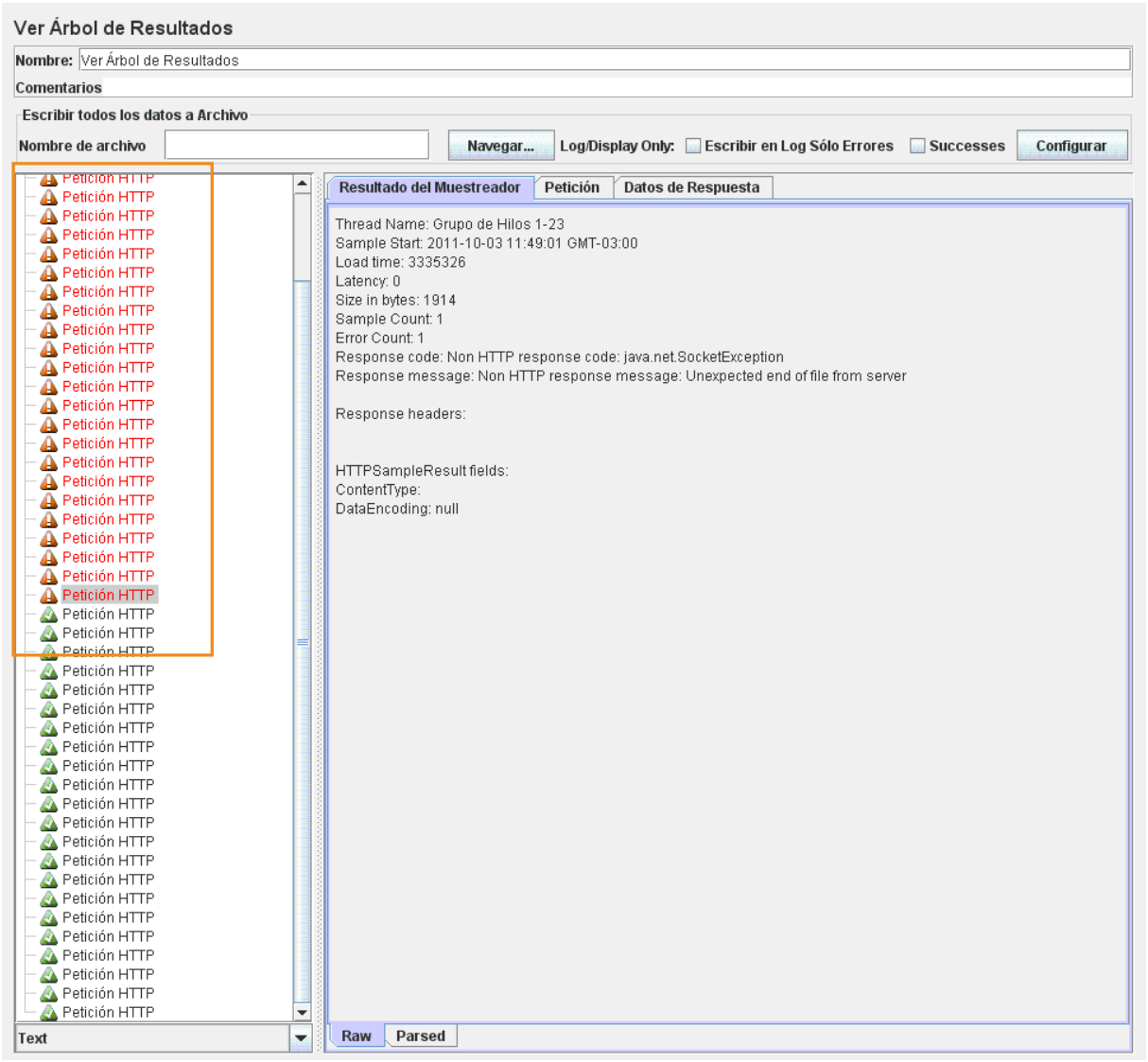


Figura 20: Errores en las peticiones HTTP.

Dado que la prueba de 50 usuarios accediendo concurrentemente al sistema dio como resultado la no conexión al mismo, no se siguió probando con la configuración anterior, se aumentó la memoria y se procedió a la re ejecución de todas las pruebas, completando así las pruebas con 100 y 200 usuarios simultáneos. Los valores totales obtenidos por la componente “Aggregate Graph” se muestran en la Tabla 12:

Prueba 1.1-b						
# Usuarios	# Muestras	Media	Mediana	% Error	Rendimiento (seg)	Kb/seg
20	20	3196	3638	0	4,08	11,69
50	50	8826	10385	0	3,74	10,72
100	100	2607530	3010379	0,34	0,02	0,066
200	200	6308107	6752481	0,64	0,02	0,046

Tabla 12: Valores correspondientes al acceso concurrente al home con mejora en la memoria.

Nota: Como puede observarse en la tabla anterior el tiempo de respuesta mejoró notablemente luego del aumento en la memoria del servidor y no se produjeron errores en la prueba con 50 usuarios. A partir de 100 usuarios aumento el porcentaje de error, mostrando que en la prueba correspondiente a 100 usuarios 34 de ellos no lograron conectarse y en la de 200 usuarios 128 tampoco lograron la conexión.

Análisis realizado

Utilizando las fórmulas mencionadas anteriormente, se pueden observar los resultados en la Tabla 13:

Análisis 1.1-b				
# Usuarios	# Muestras	Media	Tiempo Total (milisegundos)	Tiempo Promedio (minutos)
20	20	3196	63920	0,053266667
50	50	8826	441300	0,1471
100	100	2607530	260753000	43,45883333
200	200	6308107	1261621400	105,1351167

Tabla 13: Valores correspondientes al tiempo promedio y total.

Prueba 1.2: Acceso a la página principal del sistema ("*home*") en el cual se configuraron usuarios separados con 1 segundo de salida entre cada uno, las cantidades configuradas son las siguientes: 20, 50, 100 y 200. Los valores totales obtenidos por la componente "Aggregate Graph" se muestran en la Tabla 14:

Prueba 1.2						
# Usuarios	# Muestras	Media	Mediana	% Error	Rendimiento (seg)	Kb/seg
20	20	425	414	0	1,02	2,93
50	50	409	407	0	1,009	2,89
100	100	412	417	0	1,003	2,87
200	200	409	408	0	1,001	2,86

Tabla 14: Valores correspondientes al acceso con delay de 1 seg. entre cada salida de usuario al home.

Nota: Durante esta prueba se puede observar que no se produjeron errores de conexión dado que los usuarios realizan la petición al home del sistema cada 1 seg.

Análisis realizado

Utilizando las fórmulas mencionadas anteriormente, obtenemos los siguientes resultados mostrados en la Tabla 15:

Análisis 1.2				
# Usuarios	# Muestras	Media	Tiempo Total (milisegundos)	Tiempo Promedio (minutos)
20	20	425	8500	0,007
50	50	409	20450	0,006
100	100	412	41200	0,006
200	200	409	81800	0,006

Tabla 15: Valores correspondientes al tiempo promedio y total.

Conclusiones

Como se mencionó anteriormente en el servidor con 2 GB de RAM funcionaron las pruebas realizadas, ya que cuando el mismo contaba con 512 MB las pruebas con 50 usuarios no llegaron a completarse.

Dado el aumento en la memoria del servidor también se pudo observar un mayor rendimiento al comparar los resultados de la columna rendimiento de las tablas 11 y 12.

Comparando los resultados de las pruebas (de usuarios concurrentes y de usuarios con un tiempo de espera), se puede ver un mejor rendimiento en el acceso no concurrente en todas las cantidades de usuarios.

10.5.2 Prueba 2: Ingreso con login (con xml sin logout)

Como segunda prueba se decidió probar el tiempo de respuesta del sistema ante el acceso al sistema con login de usuarios.

Para ello se configuró a la herramienta con un "*Grupo de Hilos*" para poder simular los accesos de los usuarios.

Al mismo se le agregó el muestreador "*Petición HTTP*" el cual permite el acceso a la página principal del sistema, donde se encuentra el formulario de ingreso, para ello se ingresó el nombre del servidor en el campo correspondiente.

A este muestreador, se le agregó el componente "*Modificador de Parámetro de Usuario HTTP*" del menú "*Pre Procesadores*", el cual permite utilizar un archivo .xml con los datos necesarios para el formulario.

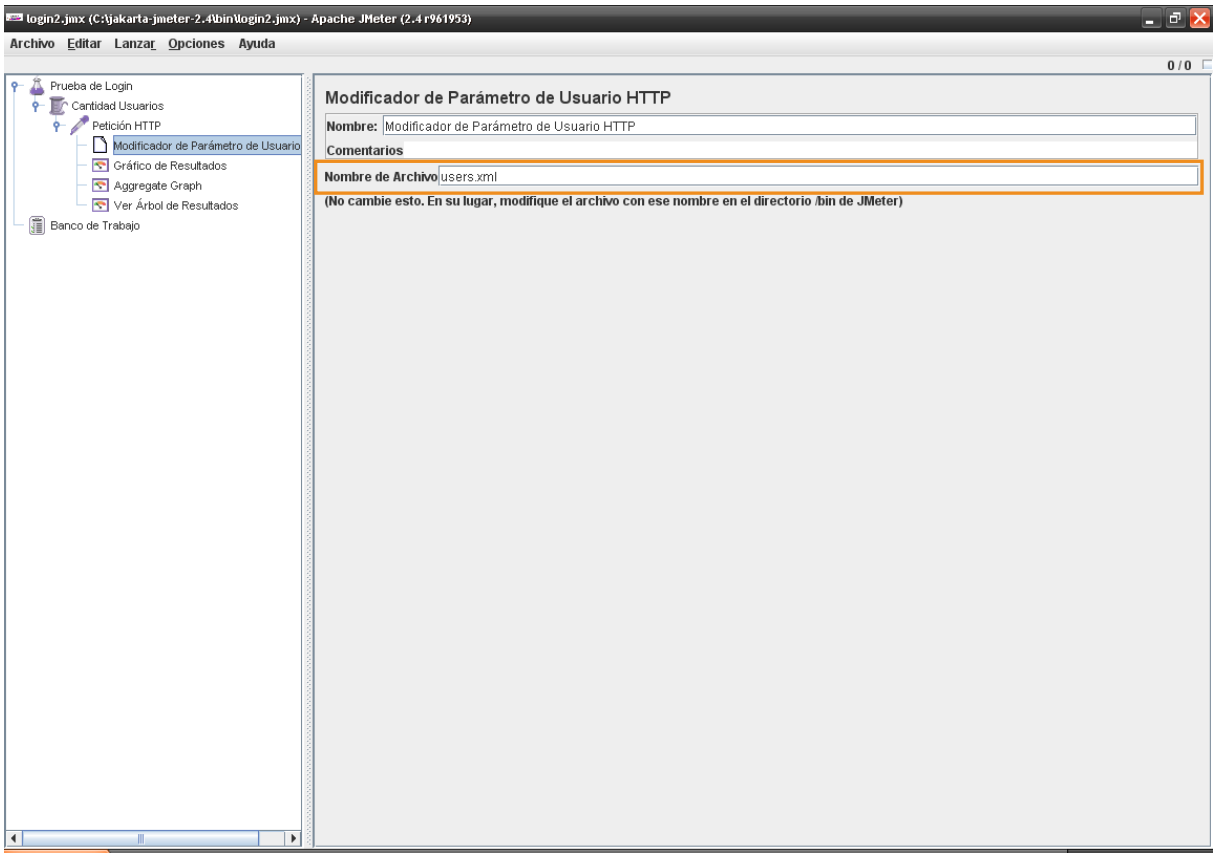


Figura 21: Modificador de Parámetro de Usuario HTTP.

Para esta prueba los datos de ingreso son: "*signin[username]*" para el nombre de usuario y "*signin[password]*" para la contraseña del mismo, a continuación mostramos un ejemplo del archivo:

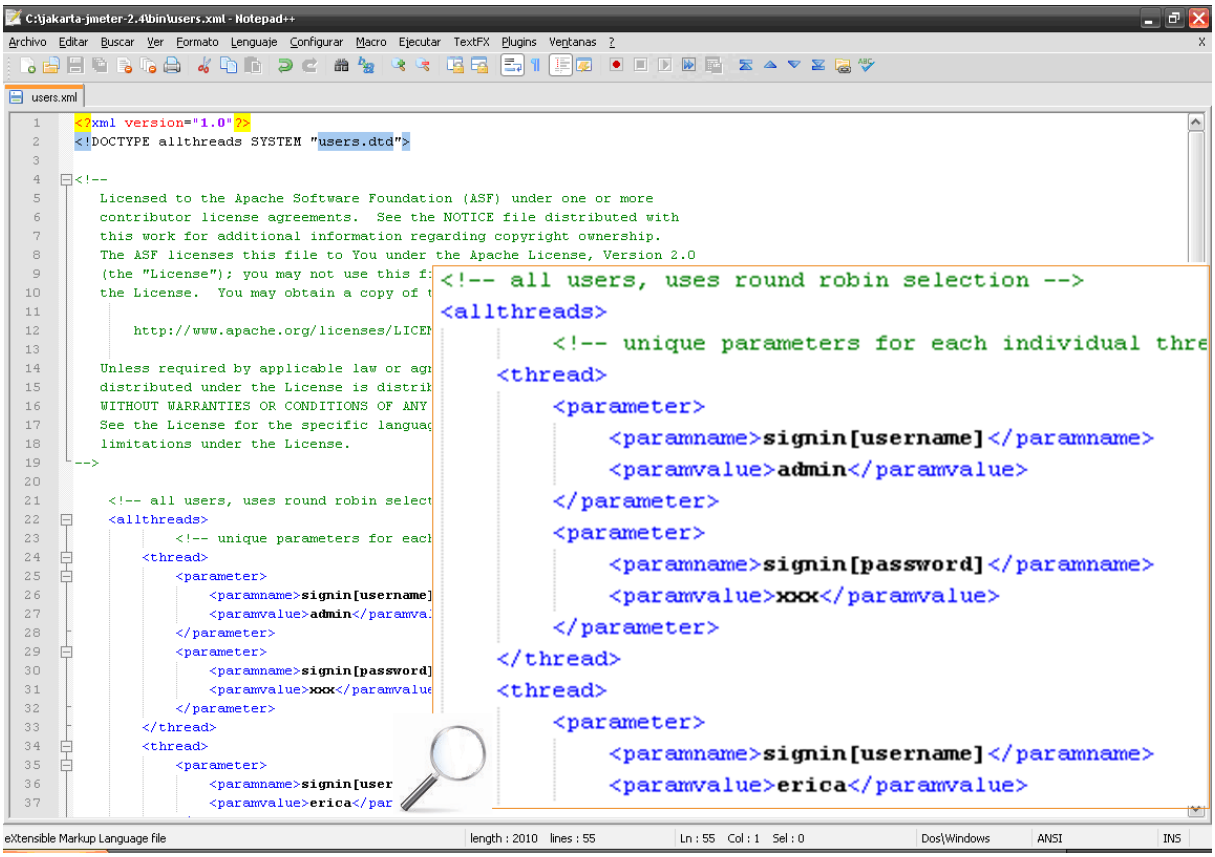


Figura 22: archivo users.xml.

Se adjunta el archivo `users.xml` en el anexo Pruebas de rendimiento.

Para poder evaluar los resultados, se agregaron los siguientes "*Listeners*": "*Aggregate Graph*" y "*Árbol de resultados*" descriptos anteriormente.

Como se mencionó anteriormente las pruebas se dividieron en dos (2):

Prueba 2.1: Acceso al sistema con usuarios concurrentes para las siguientes cantidades: 10, 20, 30, 40 y 50. Los valores totales obtenidos por el componente "*Aggregate Graph*" se muestran en la Tabla 16.

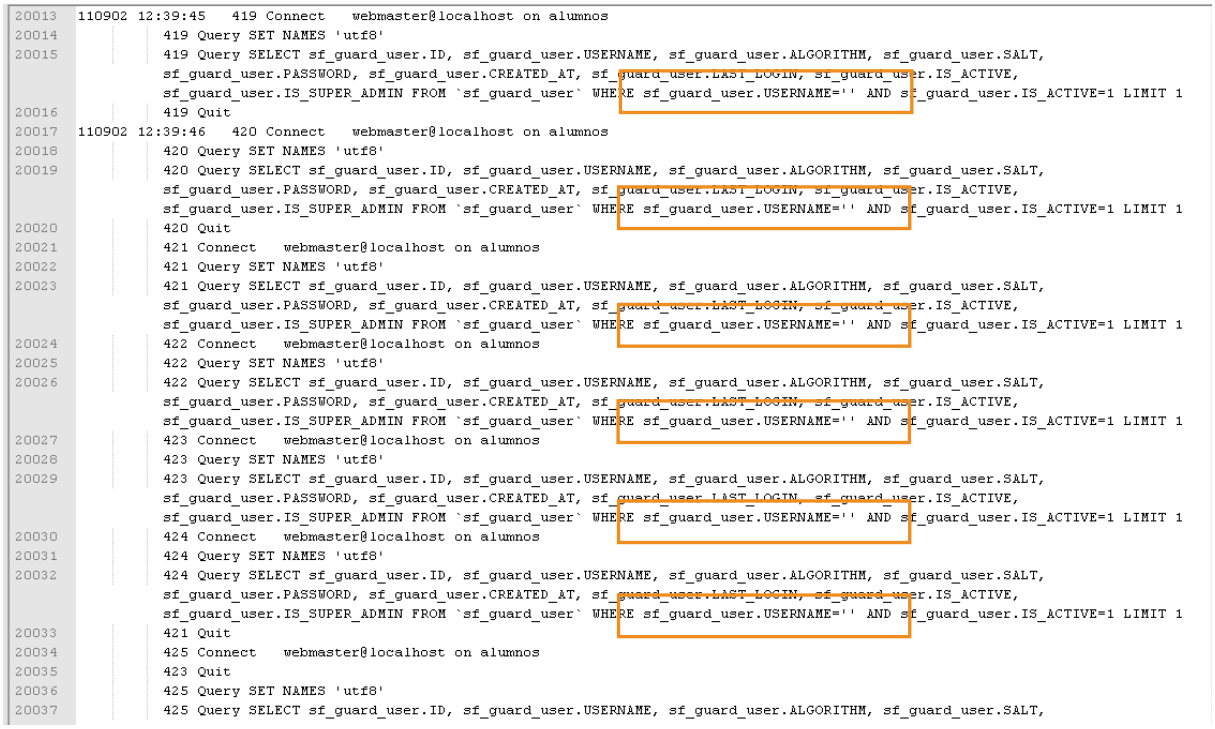
Prueba 2.1						
# Usuarios	# Muestras	Media	Mediana	% Error	Rendimiento (seg)	Kb/seg
10	10	3235	3367	0	2,31	6,82
20	20	6952	6506	0	2,21	6,52
30	30	10219	11536	0	2,28	6,73
40	40	6632	6813	0	3,91	12,01
50	50	10373	12787	0	2,71	8,33

Tabla 16: Valores correspondientes al acceso concurrente con login.

Nota: las pruebas correspondientes a 40 y 50 usuarios concurrentes, no realizaron el ingreso a la aplicación, como puede observarse en el log del MySQL el campo "username" de la consulta no contiene datos, está vacío:

```
19853 110902 12:30:10    379 Connect    webmaster@localhost on alumnos
19854            379 Query SET NAMES 'utf8'
19855            379 Query SELECT sf_guard_user.ID, sf_guard_user.USERNAME, sf_guard_user.ALGORITHM, sf_guard_user.SALT,
                 sf_guard_user.PASSWORD, sf_guard_user.CREATED_AT, sf_guard_user.LAST_LOGIN, sf_guard_user.IS_ACTIVE,
                 sf_guard_user.IS_SUPER_ADMIN FROM `sf_guard_user` WHERE sf_guard_user.USERNAME='' AND sf_guard_user.IS_ACTIVE=1 LIMIT 1
19856            379 Quit
19857            380 Connect    webmaster@localhost on alumnos
19858            380 Query SET NAMES 'utf8'
19859            380 Query SELECT sf_guard_user.ID, sf_guard_user.USERNAME, sf_guard_user.ALGORITHM, sf_guard_user.SALT,
                 sf_guard_user.PASSWORD, sf_guard_user.CREATED_AT, sf_guard_user.LAST_LOGIN, sf_guard_user.IS_ACTIVE,
                 sf_guard_user.IS_SUPER_ADMIN FROM `sf_guard_user` WHERE sf_guard_user.USERNAME='' AND sf_guard_user.IS_ACTIVE=1 LIMIT 1
19860            381 Connect    webmaster@localhost on alumnos
19861            381 Query SET NAMES 'utf8'
19862            381 Query SELECT sf_guard_user.ID, sf_guard_user.USERNAME, sf_guard_user.ALGORITHM, sf_guard_user.SALT,
                 sf_guard_user.PASSWORD, sf_guard_user.CREATED_AT, sf_guard_user.LAST_LOGIN, sf_guard_user.IS_ACTIVE,
                 sf_guard_user.IS_SUPER_ADMIN FROM `sf_guard_user` WHERE sf_guard_user.USERNAME='' AND sf_guard_user.IS_ACTIVE=1 LIMIT 1
19863            382 Connect    webmaster@localhost on alumnos
19864            382 Query SET NAMES 'utf8'
19865            382 Query SELECT sf_guard_user.ID, sf_guard_user.USERNAME, sf_guard_user.ALGORITHM, sf_guard_user.SALT,
                 sf_guard_user.PASSWORD, sf_guard_user.CREATED_AT, sf_guard_user.LAST_LOGIN, sf_guard_user.IS_ACTIVE,
                 sf_guard_user.IS_SUPER_ADMIN FROM `sf_guard_user` WHERE sf_guard_user.USERNAME='' AND sf_guard_user.IS_ACTIVE=1 LIMIT 1
19866            380 Quit
19867            383 Connect    webmaster@localhost on alumnos
19868            383 Query SET NAMES 'utf8'
19869            383 Query SELECT sf_guard_user.ID, sf_guard_user.USERNAME, sf_guard_user.ALGORITHM, sf_guard_user.SALT,
                 sf_guard_user.PASSWORD, sf_guard_user.CREATED_AT, sf_guard_user.LAST_LOGIN, sf_guard_user.IS_ACTIVE,
                 sf_guard_user.IS_SUPER_ADMIN FROM `sf_guard_user` WHERE sf_guard_user.USERNAME='' AND sf_guard_user.IS_ACTIVE=1 LIMIT 1
19870            381 Quit
19871            382 Quit
19872            384 Connect    webmaster@localhost on alumnos
19873            383 Quit
19874            384 Query SET NAMES 'utf8'
19875            384 Query SELECT sf_guard_user.ID, sf_guard_user.USERNAME, sf_guard_user.ALGORITHM, sf_guard_user.SALT,
                 sf_guard_user.PASSWORD, sf_guard_user.CREATED_AT, sf_guard_user.LAST_LOGIN, sf_guard_user.IS_ACTIVE,
                 sf_guard_user.IS_SUPER_ADMIN FROM `sf_guard_user` WHERE sf_guard_user.USERNAME='' AND sf_guard_user.IS_ACTIVE=1 LIMIT 1
19876            385 Connect    webmaster@localhost on alumnos
19877            385 Query SET NAMES 'utf8'
```

Figura 23: MySQL Log Prueba de 40 usuarios.



```
20013  110902 12:39:45    419 Connect    webmaster@localhost on alumnos
20014             419 Query SET NAMES 'utf8'
20015             419 Query SELECT sf_guard_user.ID, sf_guard_user.USERNAME, sf_guard_user.ALGORITHM, sf_guard_user.SALT,
                  sf_guard_user.PASSWORD, sf_guard_user.CREATED_AT, sf_guard_user.LAST_LOGIN, sf_guard_user.IS_ACTIVE,
                  sf_guard_user.IS_SUPER_ADMIN FROM `sf_guard_user` WHERE sf_guard_user.USERNAME='' AND sf_guard_user.IS_ACTIVE=1 LIMIT 1
20016             419 Quit
20017  110902 12:39:46    420 Connect    webmaster@localhost on alumnos
20018             420 Query SET NAMES 'utf8'
20019             420 Query SELECT sf_guard_user.ID, sf_guard_user.USERNAME, sf_guard_user.ALGORITHM, sf_guard_user.SALT,
                  sf_guard_user.PASSWORD, sf_guard_user.CREATED_AT, sf_guard_user.LAST_LOGIN, sf_guard_user.IS_ACTIVE,
                  sf_guard_user.IS_SUPER_ADMIN FROM `sf_guard_user` WHERE sf_guard_user.USERNAME='' AND sf_guard_user.IS_ACTIVE=1 LIMIT 1
20020             420 Quit
20021             421 Connect    webmaster@localhost on alumnos
20022             421 Query SET NAMES 'utf8'
20023             421 Query SELECT sf_guard_user.ID, sf_guard_user.USERNAME, sf_guard_user.ALGORITHM, sf_guard_user.SALT,
                  sf_guard_user.PASSWORD, sf_guard_user.CREATED_AT, sf_guard_user.LAST_LOGIN, sf_guard_user.IS_ACTIVE,
                  sf_guard_user.IS_SUPER_ADMIN FROM `sf_guard_user` WHERE sf_guard_user.USERNAME='' AND sf_guard_user.IS_ACTIVE=1 LIMIT 1
20024             422 Connect    webmaster@localhost on alumnos
20025             422 Query SET NAMES 'utf8'
20026             422 Query SELECT sf_guard_user.ID, sf_guard_user.USERNAME, sf_guard_user.ALGORITHM, sf_guard_user.SALT,
                  sf_guard_user.PASSWORD, sf_guard_user.CREATED_AT, sf_guard_user.LAST_LOGIN, sf_guard_user.IS_ACTIVE,
                  sf_guard_user.IS_SUPER_ADMIN FROM `sf_guard_user` WHERE sf_guard_user.USERNAME='' AND sf_guard_user.IS_ACTIVE=1 LIMIT 1
20027             423 Connect    webmaster@localhost on alumnos
20028             423 Query SET NAMES 'utf8'
20029             423 Query SELECT sf_guard_user.ID, sf_guard_user.USERNAME, sf_guard_user.ALGORITHM, sf_guard_user.SALT,
                  sf_guard_user.PASSWORD, sf_guard_user.CREATED_AT, sf_guard_user.LAST_LOGIN, sf_guard_user.IS_ACTIVE,
                  sf_guard_user.IS_SUPER_ADMIN FROM `sf_guard_user` WHERE sf_guard_user.USERNAME='' AND sf_guard_user.IS_ACTIVE=1 LIMIT 1
20030             424 Connect    webmaster@localhost on alumnos
20031             424 Query SET NAMES 'utf8'
20032             424 Query SELECT sf_guard_user.ID, sf_guard_user.USERNAME, sf_guard_user.ALGORITHM, sf_guard_user.SALT,
                  sf_guard_user.PASSWORD, sf_guard_user.CREATED_AT, sf_guard_user.LAST_LOGIN, sf_guard_user.IS_ACTIVE,
                  sf_guard_user.IS_SUPER_ADMIN FROM `sf_guard_user` WHERE sf_guard_user.USERNAME='' AND sf_guard_user.IS_ACTIVE=1 LIMIT 1
20033             421 Quit
20034             425 Connect    webmaster@localhost on alumnos
20035             423 Quit
20036             425 Query SET NAMES 'utf8'
20037             425 Query SELECT sf_guard_user.ID, sf_guard_user.USERNAME, sf_guard_user.ALGORITHM, sf_guard_user.SALT,
```

Figura 24: MySQL Log Prueba de 50 usuarios.

Análisis realizado

Utilizando las fórmulas mencionadas anteriormente, se pueden observar los resultados en la Tabla 17:

Análisis 2.1				
# Usuarios	**# Muestras**	**Media**	**Tiempo Total (milisegundos)**	**Tiempo Promedio (minutos)**
10	10	3235	32350	0,0539
20	20	6952	139040	0,1159
30	30	10219	306570	0,1703
40	40	6632	265280	0,1105
50	50	10373	518650	0,1729

Tabla 17: Valores correspondientes al tiempo promedio y total.

Prueba 2.2: Acceso al sistema con usuarios separados con 1 segundo de salida entre cada uno: 10, 20, 30, 40 y 50. Los valores totales obtenidos por la componente "*Aggregate Graph*" se muestran en la Tabla 18:

Prueba 2.2						
# Usuarios	# Muestras	Media	Mediana	% Error	Rendimiento (seg)	Kb/seg
10	10	828	824	0	1,01	2,98
20	20	869	868	0	1,006	2,96
30	30	812	794	0	1,002	2,95
40	40	415	410	0	1,01	3,10
50	50	418	415	0	1,01	3,09

Tabla 18: Valores correspondientes al acceso concurrente con login.

En esta prueba todos los usuarios lograron ingresar al sistema con nombre de usuario y contraseña.

Análisis realizado

Utilizando las fórmulas mencionadas anteriormente, se pueden observar los resultados en la Tabla 19:

Análisis 2.2				
# Usuarios	# Muestras	Media	Tiempo Total (milisegundos)	Tiempo Promedio (minutos)
10	10	828	8280	0,0138
20	20	869	17380	0,01448333
30	30	812	24360	0,01353333
40	40	415	16600	0,00691667
50	50	418	20900	0,00696667

Tabla 19: Valores correspondientes al tiempo promedio y total.

Conclusiones

Como se mencionó anteriormente en la primer prueba con 40 y 50 usuarios generaron problemas durante el login a causa de una sobrecarga al motor de bases de datos utilizado (MySql). Se pudo observar que si las conexiones tienen espera de 1 segundo, el motor de bases de datos responde correctamente.

Se puede observar que el rendimiento en la prueba 2.1 es mejor que el de la prueba 2.2, pero hay que aclarar que esto se dio dado que en el caso de 40 y 50 usuarios de la

primer prueba los usuarios no ingresaron por lo que no se pudo medir correctamente la performance de la misma. Por lo que los valores de la prueba 2 son más reales, dado que se realizaron todos los ingresos de los usuarios.

10.5.3 Prueba 3: Ingreso con login mejorado con logout (con csv con logout)

Como tercer prueba se decidió probar otra técnica para el login de usuarios con otra configuración:

Para ello se configuro la prueba de la siguiente manera:

- Se grabó un camino de navegación utilizando el componente "*Servidor Proxy HTTP*", el camino es el siguiente:
    - Acceso al home del sistema ingresando usuario y contraseña
    - Visualización del home una vez que el usuario ha ingresado.
    - Cierre de sesión en el sistema.

- Se agregó el componente "*Configuración del CSV Data Set*" para que se utilicen los datos de ingreso al sistema, al cual se lo debe configurar con los datos que se parametrizarán en el componente "*Petición HTTP*" del login, como se observa en las siguientes imágenes:

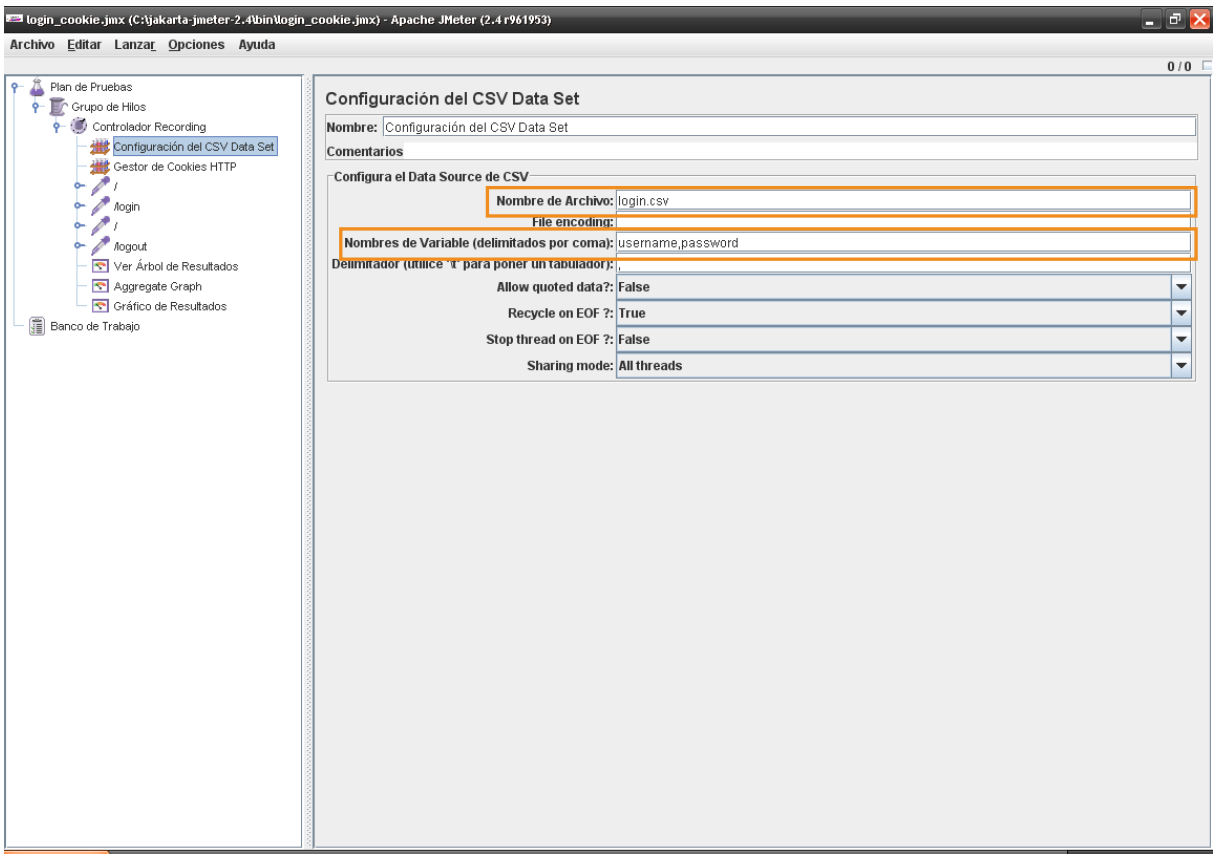


Figura 25: Configuración del CSV data set.

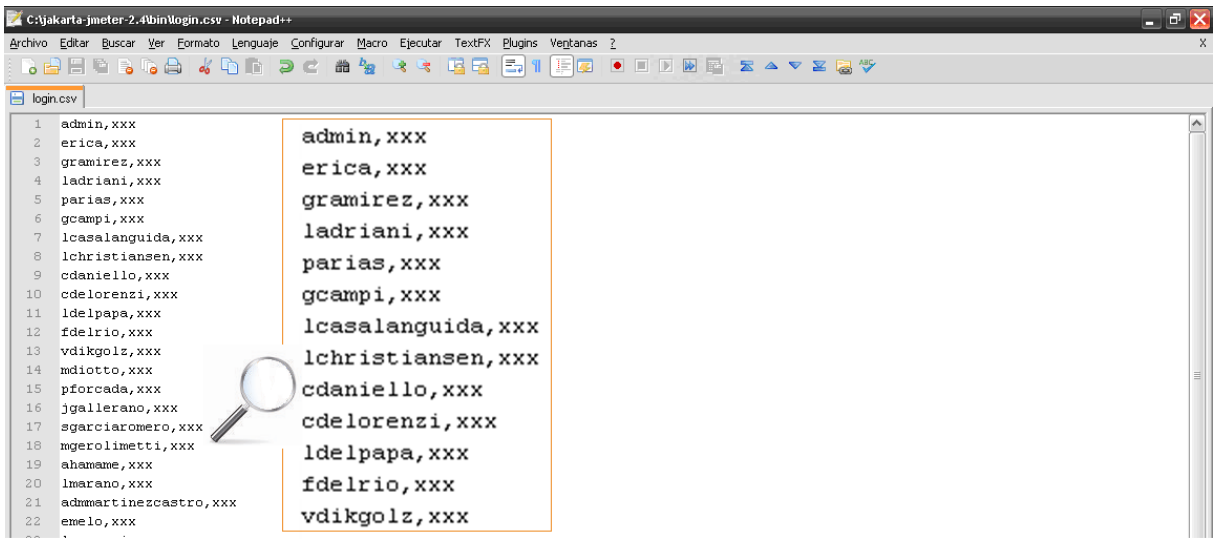


Figura 26: archivo login.csv.

Se adjunta el archivo `login.csv` en el anexo Pruebas de rendimiento.

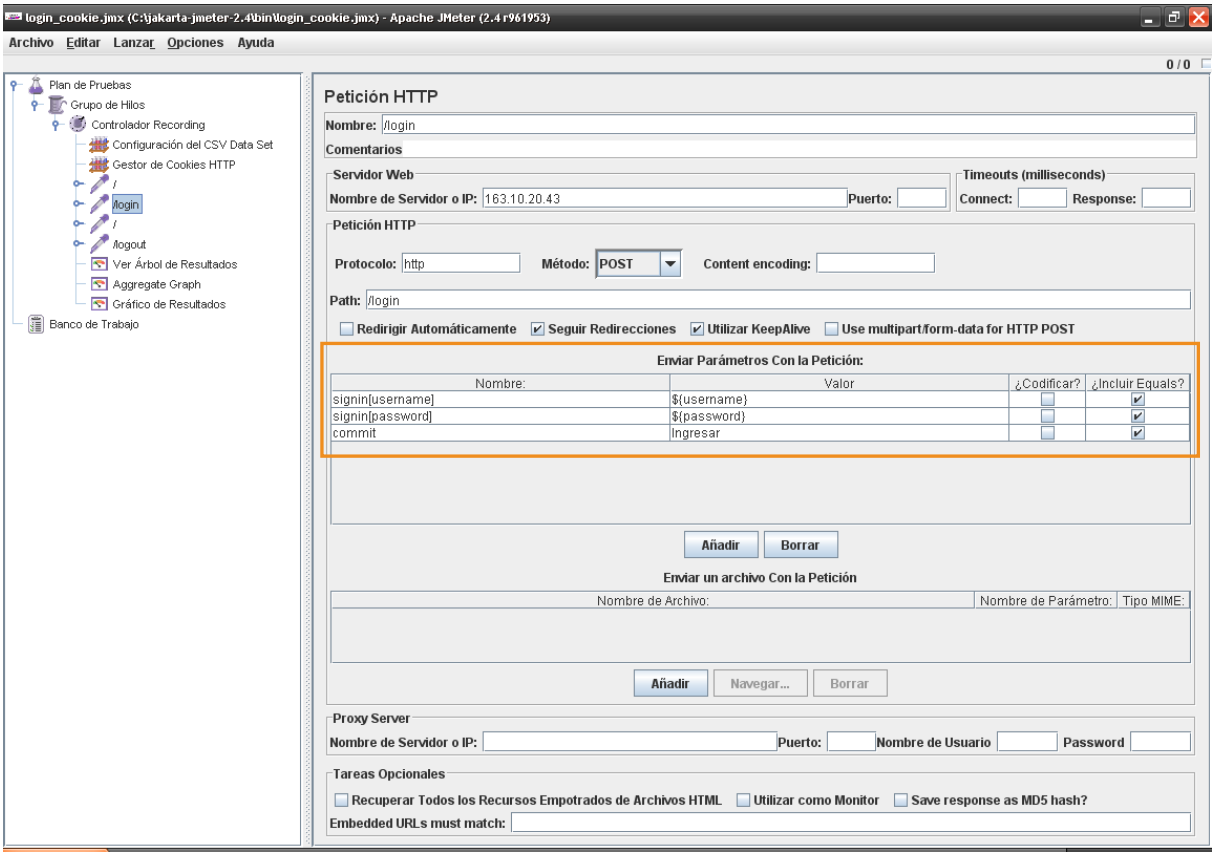


Figura 27: Petición HTTP donde se parametrizan los datos de entrada.

- Luego se agregó el componente "*Gestor de Cookies HTTP*" para que se mantenga la sesión de usuario. Sin este componente, luego de realizar exitosamente el login de usuario, se pierde la sesión del mismo en la siguiente petición.

Para poder evaluar los resultados, se agregaron los siguientes "*Listeners*": "*Aggregate Graph*" y "*Árbol de resultados*" descriptos anteriormente.

Como se mencionó anteriormente las pruebas se dividieron en dos (2):

Prueba 3.1: Acceso al sistema con usuarios concurrentes para las siguientes cantidades de usuarios: 10, 20, 30, 40 y 50. Los valores totales obtenidos por la componente "*Aggregate Graph*" se muestran en la Tabla 20:

Prueba 3.1						
# Usuarios	# Muestras	Media	Mediana	% Error	Rendimiento (seg)	Kb/seg
10	40	2957	2859	0.0	3,19	17,98
20	80	5736	5932	0.0	321,64	17,69
30	120	8560	8608	0.0	32,20	17,61
40	160	18924	11677	0.0	18,49	10,04
50	200	1034337	182669	0.08	0,04	0,18

Tabla 20: Valores correspondientes al acceso concurrente con login.

Nota: Como puede observarse en la tabla anterior la prueba con 50 usuarios presento errores. La misma fue lanzada el día 23/09/2011 a las 15:19 hs., una vez que finalizó se analizó el archivo de log del MySql, en el cual se observa el día y horario en el que se ejecutó la prueba, como se muestra en la siguiente pantalla:

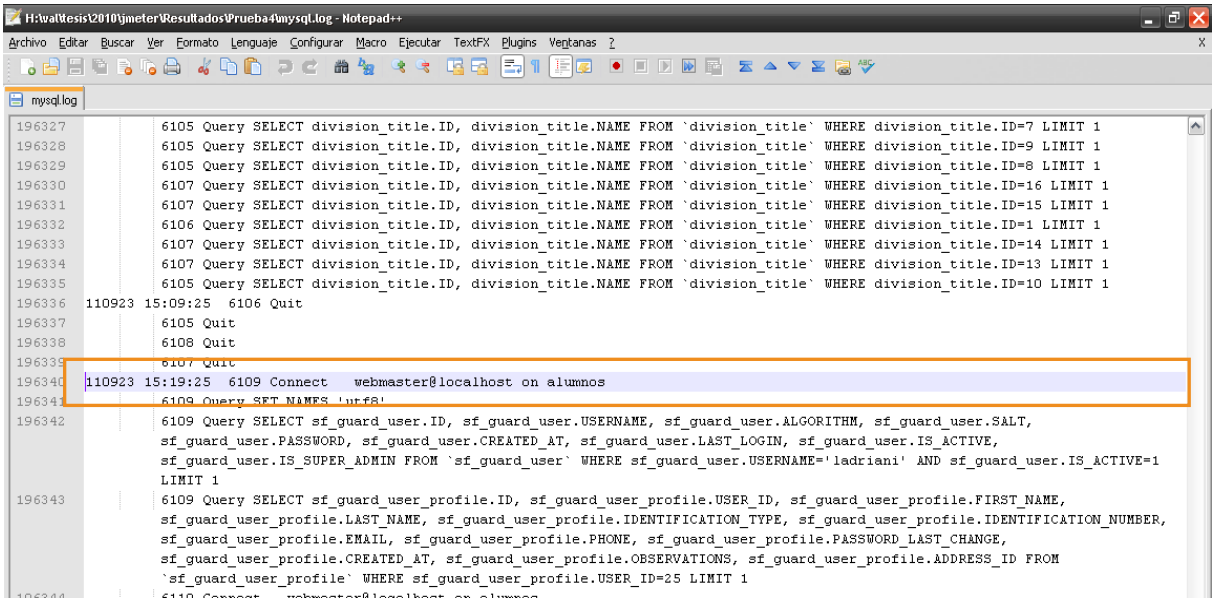


Figura 28: archivo MySQL Log.

Luego se ejecutó una consulta sql (`select * from sf_guar_user`) sobre la tabla de usuarios para ver el último ingreso de los mismos y el resultado de la consulta muestra que hay usuarios que no lograron ingresar, dado que la prueba fue ejecutada el día 23/09/2011 a las 15:19hs. como se mencionó anteriormente. En la siguiente pantalla se observa que la

fecha de ingreso de algunos usuarios es anterior a este horario, lo que implica que los mismos no lograron ingresar al sistema:

id	username	algorithm	created_at	last_login	is_active	is_super_admin
1	admin	sha1	2011-02-02 15:24:38	2011-09-23 15:19:27	1	1
22	erica	sha1	2011-02-09 10:19:56	2011-09-23 16:21:24	1	1
24	gramirez	sha1	2011-02-09 10:27:58	2011-09-23 15:19:26	1	0
25	ladriani	sha1	2011-02-09 10:54:36	2011-09-23 15:19:25	1	0
26	parias	sha1	2011-02-09 11:08:48	2011-09-23 15:19:25	1	0
27	gcampi	sha1	2011-02-09 11:10:26	2011-09-23 15:19:26	1	0
28	lcasalanguida	sha1	2011-02-09 11:12:15	2011-09-23 15:19:27	1	0
29	lchristiansen	sha1	2011-02-09 11:13:55	2011-09-23 15:19:26	1	0
30	cdaniello	sha1	2011-02-09 11:15:10	2011-09-23 15:19:25	1	0
31	cdelorenzi	sha1	2011-02-09 11:16:44	2011-09-23 15:19:30	1	0
32	ldelpapa	sha1	2011-02-09 11:18:33	2011-09-23 15:19:59	1	0
33	fdelrio	sha1	2011-02-09 11:19:54	2011-09-23 15:21:09	1	0
34	vdikgolz	sha1	2011-02-09 11:21:33	2011-09-23 15:19:26	1	0
35	mdiotto	sha1	2011-02-09 11:22:49	2011-09-23 15:19:26	1	0
36	pforcada	sha1	2011-02-09 11:25:46	2011-09-23 15:21:14	1	0
37	jgallerano	sha1	2011-02-09 11:26:50	2011-09-23 15:20:32	1	0
38	sgarciaromero	sha1	2011-02-09 11:29:42	2011-09-23 15:21:14	1	0
39	mgerolimetti	sha1	2011-02-09 11:31:45	2011-09-23 15:20:08	1	0
40	amamame	sha1	2011-02-09 11:34:25	2011-09-23 16:31:05	1	0
42	lmarano	sha1	2011-02-09 11:38:15	2011-09-23 16:32:14	1	0
43	admmartinezcastro	sha1	2011-02-09 11:40:06	2011-09-23 16:32:50	1	0
44	emelo	sha1	2011-02-09 11:41:30	2011-09-23 15:08:22	1	0
45	lpezzani	sha1	2011-02-09 11:42:50	2011-09-23 16:32:08	1	0
46	morellana	sha1	2011-02-09 11:44:30	2011-09-23 16:30:44	1	0
47	asilveti	sha1	2011-02-09 11:45:43	2011-09-23 16:32:44	1	0
48	msimiele	sha1	2011-02-09 11:49:10	2011-09-23 15:20:29	1	0
49	mstrazzeri	sha1	2011-02-09 11:50:43	2011-09-23 15:21:09	1	0
50	myaber	sha1	2011-02-09 11:52:28	2011-09-23 16:32:46	1	0
51	pherrera	sha1	2011-02-09 11:53:45	2011-09-23 16:28:37	1	0
52	hcafasso	sha1	2011-02-09 11:54:50	2011-09-23 16:31:47	1	0
53	acaprio	sha1	2011-02-09 11:56:01	2011-09-23 15:08:28	1	0
54	acappannini	sha1	2011-02-09 11:57:50	2011-09-23 15:08:22	1	0
55	ldossantos	sha1	2011-02-09 11:59:37	2011-09-23 15:08:26	1	0
56	caprea	sha1	2011-02-09 12:00:48	2011-09-23 15:21:09	1	0
57	ngonzalez	sha1	2011-02-09 12:02:12	2011-09-23 16:15:05	1	0
58	mmartinez	sha1	2011-02-09 12:04:00	2011-09-23 15:08:28	1	0
59	apazferrari	sha1	2011-02-09 12:05:49	2011-09-23 16:25:22	1	0
60	cpeña	sha1	2011-02-09 12:06:58	2011-09-23 12:50:29	1	0
61	gperez	sha1	2011-02-09 12:08:46	2011-09-23 15:21:20	1	0
62	framirez	sha1	2011-02-09 12:10:30	2011-09-23 16:27:45	1	0
63	esimiele	sha1	2011-02-09 12:13:59	2011-09-23 12:59:02	1	0
64	msurraco	sha1	2011-02-09 12:15:39	2011-09-23 16:32:22	1	0
66	pacuña	sha1	2011-02-11 12:24:45	NULL	1	0
67	lagratti	sha1	2011-02-11 12:27:01	2011-09-23 16:27:48	1	0
68	galarcon	sha1	2011-02-11 12:27:52	2011-09-23 15:22:59	1	0
69	malba	sha1	2011-02-11 12:28:46	2011-09-23 12:59:08	1	0
70	aalbarracin	sha1	2011-02-11 12:29:29	2011-09-23 16:32:47	1	0
71	nali	sha1	2011-02-11 12:30:14	2011-09-23 12:59:10	1	0
72	malonso	sha1	2011-02-11 12:31:35	2011-09-23 12:59:10	1	0
73	maloy	sha1	2011-02-11 12:32:02	2011-09-23 15:22:03	1	0
74	maltuve	sha1	2011-02-11 12:32:27	NULL	1	0
75	namoroso	sha1	2011-02-11 12:33:00	NULL	1	0

Figura 29: Consulta MySQL.

Análisis realizado

Utilizando las fórmulas mencionadas anteriormente, se pueden observar los resultados en la Tabla 21:

Análisis 3.1				
# Usuarios	# Muestras	Media	Tiempo Total (milisegundos)	Tiempo Promedio (minutos)
10	40	2957	118280	0,19713333
20	80	5736	458880	0,3824
30	120	8560	1027200	0,57066667
40	160	18924	3027840	1,2616
50	200	1034337	206867400	68,9558

Tabla 21: Valores correspondientes al tiempo promedio y total.

Prueba 3.2: Acceso al sistema con usuarios separados con 1 segundo de salida entre cada uno: 10, 20, 30, 40 y 50. Los valores totales obtenidos por el componente "*Aggregate Graph*" se muestran en la Tabla 22:

Prueba .32						
# Usuarios	# Muestras	Media	Mediana	% Error	Rendimiento (seg)	Kb/seg
10	40	1193	1128	0	3,05	17,22
20	80	1647	1559	0	3,19	17,56
30	120	2221	1974	0	3,19	17,48
40	160	2733	2566	0	3,21	17,44
50	200	3647	3366	0	3,15	16,85

Tabla 22: Valores correspondientes al acceso con delay con login.

En esta prueba todos los usuarios lograron ingresar al sistema con nombre de usuario y contraseña.

Análisis realizado

Utilizando las fórmulas mencionadas anteriormente, se pueden observar los resultados en la Tabla 23:

Análisis 3.2				
# Usuarios	**# Muestras**	**Media**	**Tiempo Total (milisegundos)**	**Tiempo Promedio (minutos)**
10	40	1193	47720	0,07953333
20	80	1647	131760	0,1098
30	120	2221	266520	0,14806667
40	160	2733	437280	0,1822
50	200	3647	729400	0,24313333

Tabla 23: Valores correspondientes al tiempo promedio y total.

Conclusiones

Se puede observar que la configuración de la prueba utilizando un archivo .csv es mejor que utilizar un archivo .xml, dado que no contiene tantos errores.

Como se mencionó anteriormente en la primer prueba con 50 usuarios generó problemas durante el login de varios usuarios. Mientras que en la prueba 3.2 no se encontraron errores.

Se puede observar que el tiempo promedio en la prueba 3.2 es mejor que el de la prueba 3.1.

10.5.4 Prueba 4: Ingreso con login, listado y logout (csv)

Como cuarta prueba se decidió agregar al camino anterior la visualización del listado de alumnos. Para ello se utilizó el componente "*Petición HTTP*" para el listado que se encuentra en "*/student*".

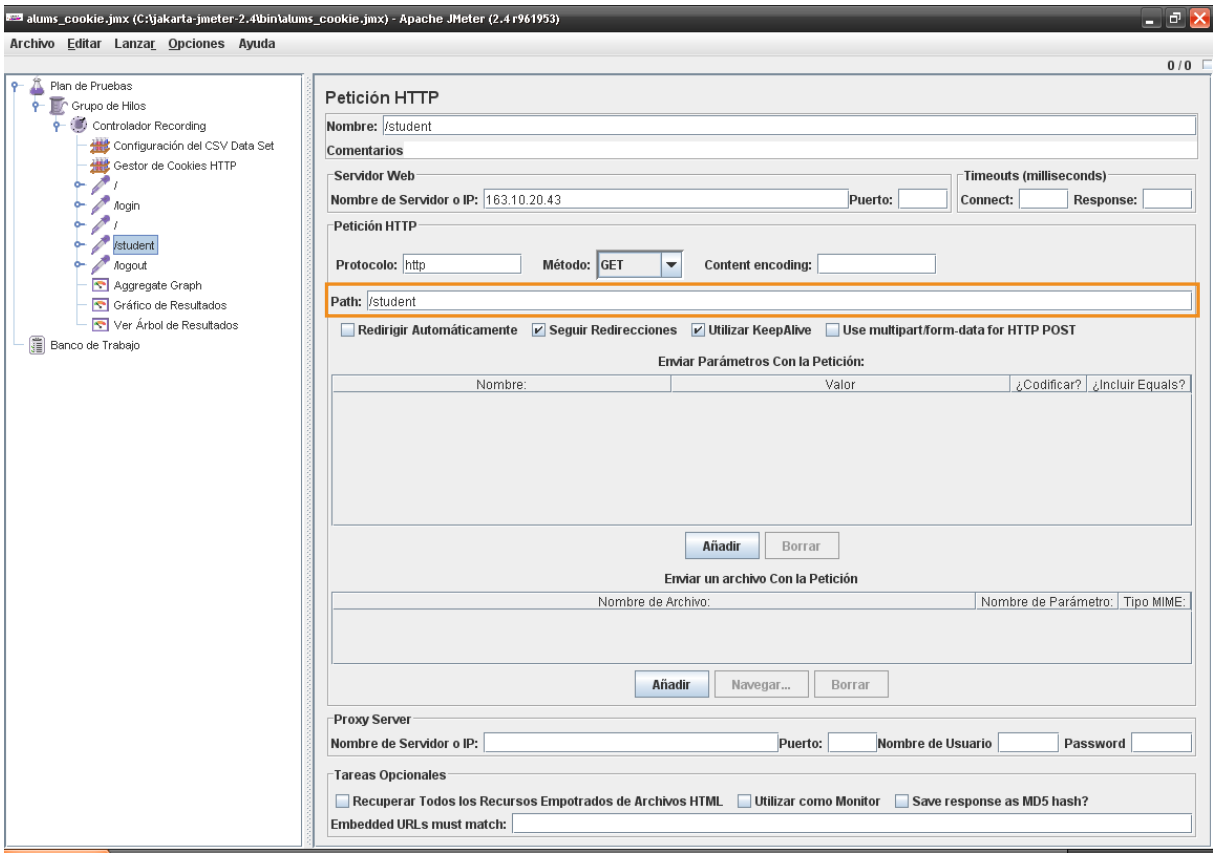


Figura 30: Petición HTTP.

Todos los usuarios utilizados en esta prueba tienen permisos para acceder a dicho listado.

Para poder evaluar los resultados, se agregaron los siguientes "*Listeners*": "*Aggregate Graph*" y "*Árbol de resultados*" descriptos anteriormente.

Como se mencionó anteriormente las pruebas se dividieron en dos (2):

Prueba 4.1: Acceso al sistema con usuarios concurrentes para las siguientes cantidades: 10, 20, 30, 40 y 50. Los valores totales obtenidos por el componente "*Aggregate Graph*" se muestran en la Tabla 24:

Prueba 4.1						
# Usuarios	# Muestras	Media	Mediana	% Error	Rendimiento (seg)	Kb/seg
10	50	3666	2969	0	2,34	33,24
20	100	7788	6118	0	2,22	32,42
30	150	11811	9367	0	2,21	32,50
40	200	136622	40997	0	0,23	3,55
50	250	1412490	221618	0,112	0,03	0,16

Tabla 24: Valores correspondientes al acceso concurrente al sistema con visualización del listado.

Análisis realizado

Utilizando las fórmulas mencionadas anteriormente, se pueden observar los resultados en la Tabla 25:

Análisis 4.1				
# Usuarios	# Muestras	Media	Tiempo Total (milisegundos)	Tiempo Promedio (minutos)
10	50	3666	183300	0,3055
20	100	7788	778800	0,649
30	150	11811	1771650	0,98425
40	200	136622	27324400	11,3851667
50	250	1412490	353122500	117,7075

Tabla 25: Valores correspondientes al tiempo promedio y total.

Nota: Se pueden observar un errores de "Response code = 500" (El servidor encuentra una condición inesperada para la cual no realiza el requerimiento [7]) en el *"Árbol de resultados"* el cual marca en color rojo aquellas conexiones que no pudieron completar la solicitud, como se muestra en la siguiente pantalla:

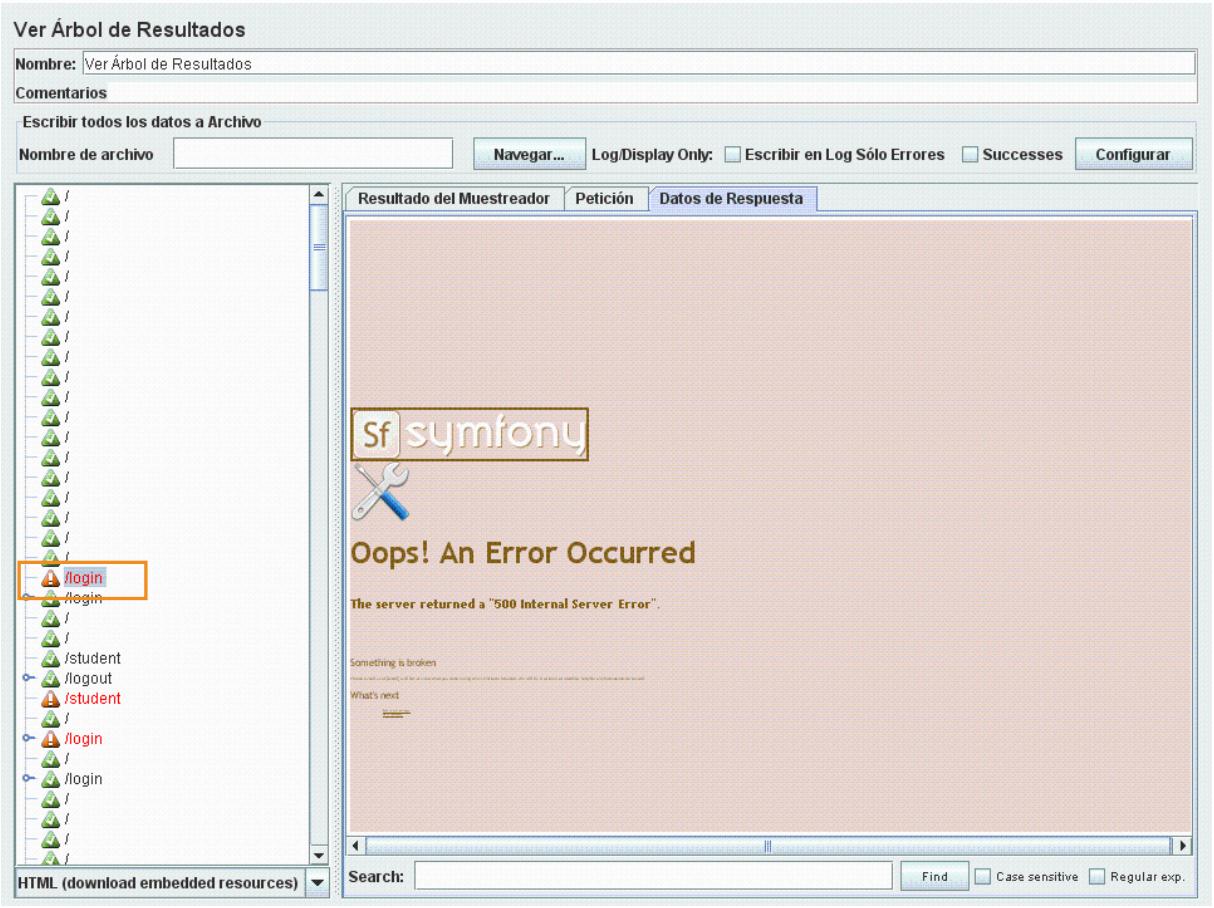


Figura 31: Errores al intentar completar una solicitud.

Como se puede observar en la pantalla anterior, no todos los usuarios lograron ingresar al sistema.

Prueba 4.2: Acceso al sistema con usuarios separados con 1 segundo de salida entre cada uno: 10, 20, 30, 40 y 50.Los valores totales obtenidos por la componente "*Aggregate Graph*" se muestran en la Tabla 26:

Prueba 4.2						
# Usuarios	# Muestras	Media	Mediana	% Error	Rendimiento (seg)	Kb/seg
10	50	2747	2128	0	2,32	33,02
20	100	5155	4031	0	2,21	32,33
30	150	7201	5732	0	2,24	33,07
40	200	9608	7390	0	2,22	33,33
50	250	55244	9244	0	0,58	8,20

Tabla 26: Valores correspondientes al acceso con delay al sistema con visualización del listado.

En esta prueba todos los usuarios lograron ingresar al sistema con nombre de usuario y contraseña.

Análisis realizado

Utilizando las fórmulas mencionadas anteriormente, se pueden observar los resultados en la Tabla 27:

Análisis 4.2				
# Usuarios	**# Muestras**	**Media**	**Tiempo Total (milisegundos)**	**Tiempo Promedio (minutos)**
10	50	2747	137350	0,22891667
20	100	5155	515500	0,42958333
30	150	7201	1080150	0,60008333
40	200	9608	1921600	0,80066667
50	250	55244	13811000	4,60366667

Tabla 27: Valores correspondientes al tiempo promedio y total.

Conclusiones

Las pruebas 4.1 y 4.2 son más complejas dado que realizan un camino de navegación. En la prueba 4.1 se pueden observar errores de "*Response code = 500*" por lo cual cuando las conexiones son concurrentes el sistema se comporta de manera inestable, esto no se ve al realizar una espera de 1 segundo entre cada usuario para cada conexión, esto simularía una utilización de sistema más real.

Se puede observar que el rendimiento en estas dos pruebas es muy similar, no así el tiempo promedio que mejoró notablemente en la prueba 4.2 con respecto a la prueba 4.1.

10.5.5 Prueba 5: Ingreso con login, guardar asistencia y logout

Como última prueba se decidió realizar un camino en el cual los usuarios con rol preceptor (más el usuario administrador) ingresan al sistema y seleccionan la opción "*Asistencias*" generando así las asistencias de un día para los alumnos pertenecientes a cada división.

Todos los usuarios utilizados en esta prueba tienen permisos para acceder a dicho listado.

Para poder evaluar los resultados, se agregaron los siguientes "*Listeners*": "*Aggregate Graph*" y "*Árbol de resultados*" descriptos anteriormente.

Como se mencionó anteriormente las pruebas se dividieron en dos (2):

Prueba 5.1: Acceso al sistema generando asistencias con usuarios concurrentes para las siguientes cantidades: 10, 20, 30, 40 y 50. Los valores totales obtenidos por el componente "*Aggregate Graph*" se muestran en la Tabla 28:

Prueba 5.1						
# Usuarios	# Muestras	Media	Mediana	% Error	Rendimiento (seg)	Kb/seg
10	280	1097	210	0	8,73	227,39
20	560	2222	294	0	8,68	225,72
30	840	3445	242	0	8,41	218,84
40	1120	144424	1474	0,001	0,26	6,53
50	1400	275133	3645	0,02	0,17	4,19

Tabla 28: Valores correspondientes al acceso concurrente guardando la asistencia de los alumnos.

Nota: Se pueden observar un errores de "*Response code = 500*" (El servidor encuentra una condición inesperada para la cual no realiza el requerimiento [7]) en el *"Árbol de resultados"* el cual marca en color rojo aquellas conexiones que no pudieron completar la solicitud, como se muestra en la siguiente pantalla:

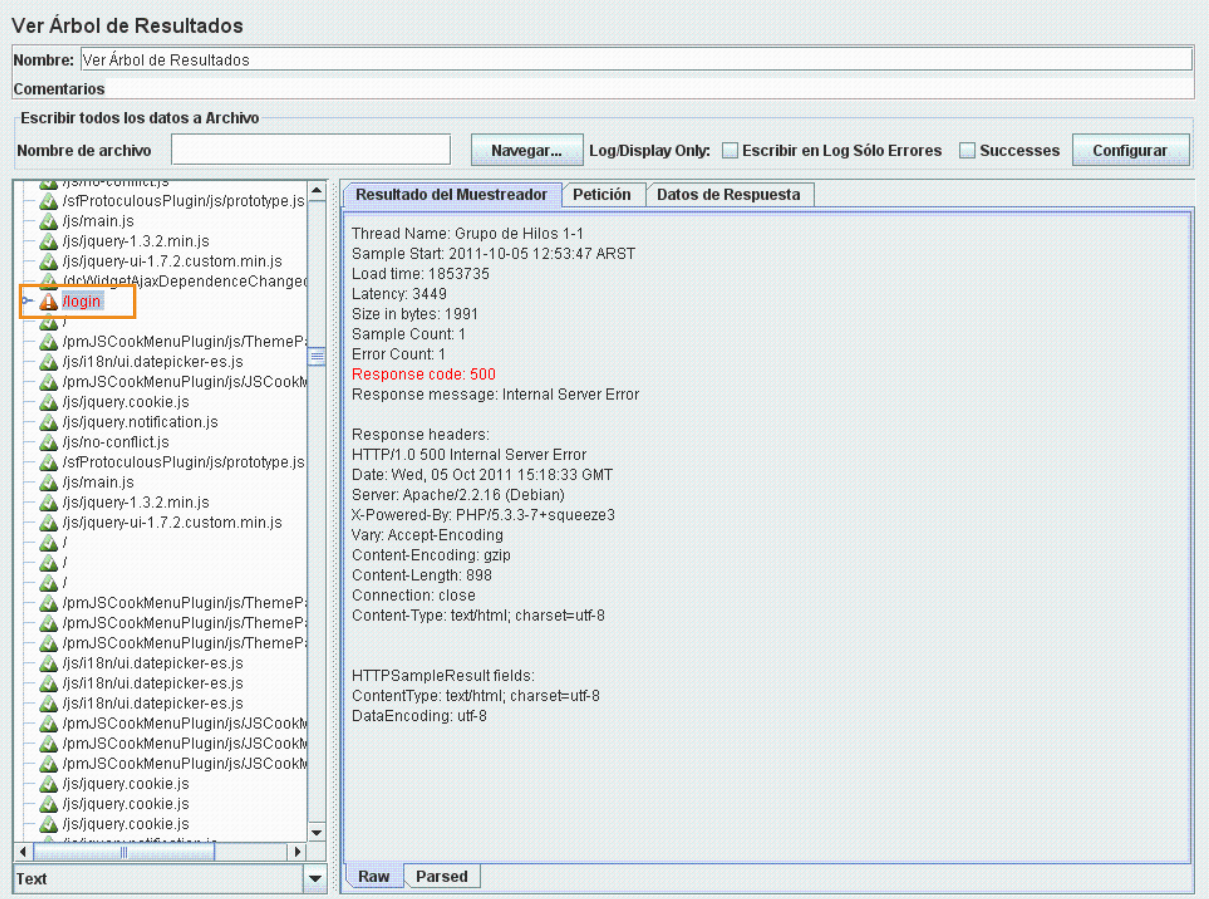


Figura 32: Errores al intentar completar una solicitud.

Como se puede observar en la pantalla anterior, no todos los usuarios lograron ingresar al sistema

Análisis realizado

Utilizando las fórmulas mencionadas anteriormente, se pueden observar los resultados en la Tabla 29:

Análisis 5.1				
# Usuarios	# Muestras	Media	Tiempo Total (milisegundos)	Tiempo Promedio (minutos)
10	280	1097	307160	0,51193333
20	560	2222	1244320	1,03693333
30	840	3445	2893800	1,60766667
40	1120	144424	161754880	67,3978667
50	1400	275133	385186200	128,3954

Tabla 29: Valores correspondientes al tiempo promedio y total.

Prueba 5.2: Acceso al sistema generando asistencias con usuarios separados con 1 segundo de salida entre cada uno: 10, 20, 30, 40 y 50. Los valores totales obtenidos por el componente "*Aggregate Graph*" se muestran en la Tabla 30:

Prueba 5.2						
# Usuarios	**# Muestras**	**Media**	**Mediana**	**% Error**	**Rendimiento (seg)**	**Kb/seg**
10	280	859	379	0	8,63	224,66
20	560	1722	252	0	8,34	217,002
30	840	2505	278	0	8,47	220,37
40	1120	3433	468	0	8,22	213,69
50	1400	107297	1078	0,085	0,37	9,16

Tabla 30: Valores correspondientes al acceso con delay guardando la asistencia de los alumnos.

Nota: Se pueden observar un errores de "*Response code = 500*" (El servidor encuentra una condición inesperada para la cual no realiza el requerimiento [7]) en el *"Árbol de resultados"* el cual marca en color rojo aquellas conexiones que no pudieron completar la solicitud, como se muestra en la siguiente pantalla:

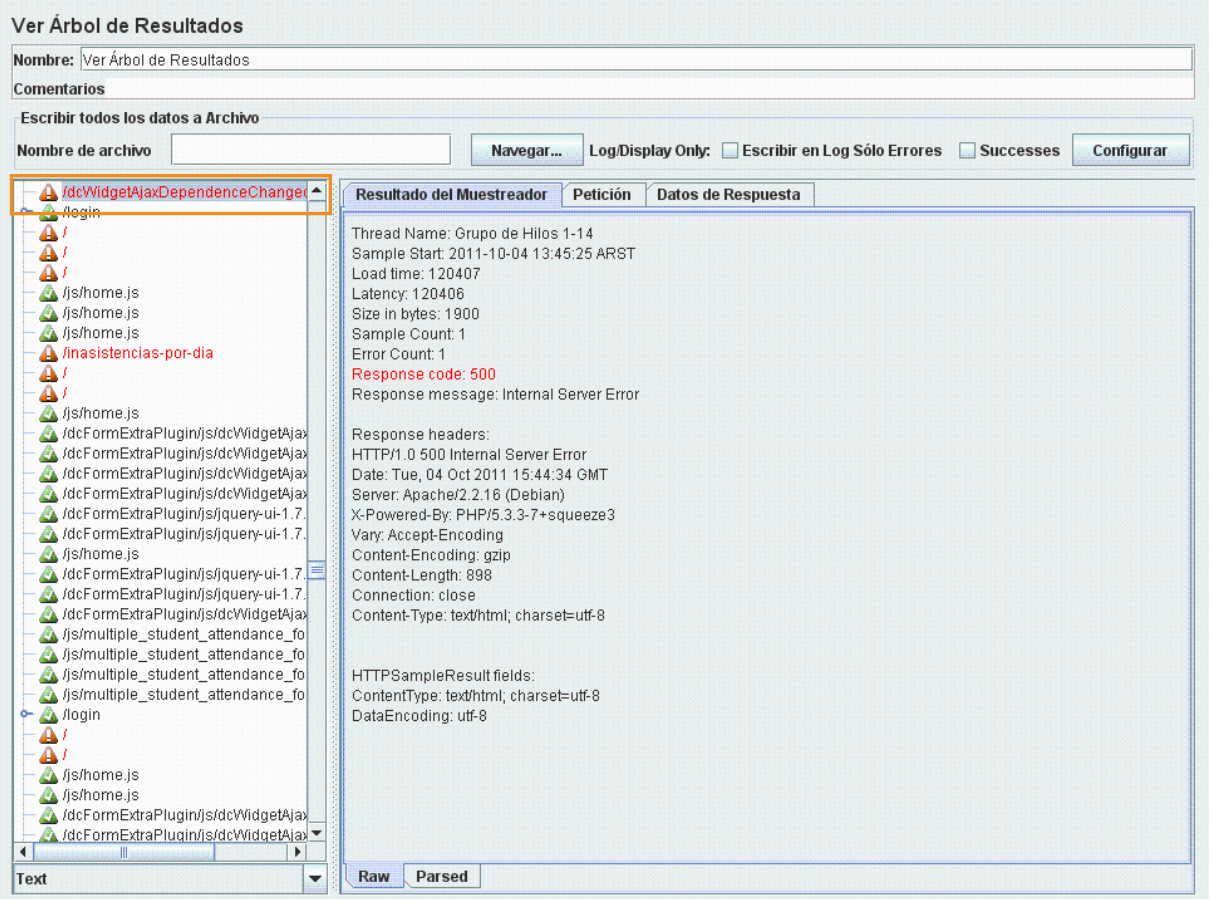


Figura 33: Errores al intentar completar una solicitud.

Como se puede observar en la pantalla anterior, no todos los usuarios lograron ingresar al sistema

Análisis realizado

Utilizando las fórmulas mencionadas anteriormente, se pueden observar los resultados en la Tabla 31:

Análisis 5.2				
# Usuarios	# Muestras	Media	Tiempo Total (milisegundos)	Tiempo Promedio (minutos)
10	280	859	240520	0,40086667
20	560	1722	964320	0,8036
30	840	2505	2104200	1,169
40	1120	3433	3844960	1,60206667
50	1400	107297	150215800	50,0719333

Tabla 31: Valores correspondientes al tiempo promedio y total.

Conclusiones

Las pruebas 5.1 y 5.2 son aún más complejas dado, que realiza una acción de guardado de datos además de un camino de navegación. De esta manera se puede comprobar de una forma más adecuada la performance del sistema Kimkëlen. Tanto en la prueba 5.1 como la 5.2 se presentaron errores de "*Response code = 500*".

Se puede observar que el rendimiento en estas dos pruebas es muy similar, no así el tiempo promedio que mejoró notablemente en la prueba 5.2 con respecto a la prueba 5.1.

10.6 Resumen

En este capítulo hemos mostrado el uso de la herramienta JMeter en pruebas de rendimiento sobre el sistema Kimkëlen.

Podemos decir que la utilización de JMeter ayuda mucho para la ejecución de este tipo de pruebas. Pero también cabe mencionar que lleva un tiempo considerable de aprendizaje y comprensión de la herramienta. Dado los resultados obtenidos se pudo comprobar que era necesaria una reconfiguración del servidor para que el sistema se comporte de manera más estable.

Como síntesis general podemos decir que el sistema se comporta mejor cuando se tiene un tiempo de espera entre cada uno de los hilos de ejecución y que el rendimiento del mismo varia un poco según la prueba.

10.6 Referencias

[1] Usando JMeter para pruebas de rendimiento, Lic. J. Díaz, Lic. C. Banchoff, A.C. A. Rodríguez y A.C. V. Soria – CACIC 2008.

[2] http://www.osmosislatina.com/jmeter/basico.htm

[3] Usability Engineering, Jakob Nielsen.

[4] http://www.useit.com/alertbox/9703a.html

[5] http://www.useit.com/papers/responsetime.html

[6] http://jakarta.apache.org/jmeter/

[7] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

Capítulo 11: Conclusiones

Existen diferentes métodos para evaluar la madurez de un proyecto, en esta tesina se evaluaron cinco (5) de ellos en el capítulo 5 (OSMM de Capgemini, OSMM de Navica, BRR, QSOS y RRL). Cada uno de estos modelos tiene sus propias particularidades, pero presentan varias características comunes que nos permitieron plantear una propuesta más adecuada a nuestros requerimientos.

Como también se vio, existen numerosas herramientas útiles en la etapa de testing y pueden ser usadas en los distintos tipos de prueba aplicable a un software, y el tiempo requerido para seleccionar las más adecuadas puede ser elevado (sin contar el tiempo de aprendizaje de las mismas).

El primer paso es poder realizar una selección adecuada que se ajuste a las necesidades de cada proyecto y, tratándose de herramientas Open Source es imprescindible elegirlas en base a su madurez, es por esto que se tomaron como base de conocimiento los métodos mencionados y se generó una plantilla propia para tal fin. Estos criterios ayudan a agrupar herramientas según su tipo de aplicación, unificar la terminología para describirlas y seleccionar una de entre las muchas existentes.

Sin aplicar criterios para la selección de una herramienta, la elección de la misma llevaría un tiempo considerable, y muchos equipos de desarrollo y de testing no cuentan con mucho tiempo para invertir en dicha elección.

Para poder demostrar el uso de algunas de las herramientas mencionadas, se planteó un caso de estudio concreto realizado para el equipo de desarrollo del CeSPI. Se tomó un caso real, donde hubo comunicación y feedback directos del equipo de desarrollo, quienes nos ayudaron a comprender el manejo y el alcance del sistema, nos facilitaron un manual de usuario y de instalación del mismo, y nos dieron una capacitación básica en el uso del sistema.

Para esta actividad, se realizó una capacitación destinada al equipo de desarrollo, donde se les explicó la importancia de las pruebas de unidad y cómo llevarlas a cabo. También se los capacitó en pruebas de rendimiento. Dichas capacitaciones estuvieron a cargo nuestro. Esto contribuyó no sólo a mejorar la comunicación con el equipo de desarrollo de la aplicación a testear, sino también a demostrar el tiempo que se requiere en aprender la utilización de estas herramientas y la importancia de su aplicación.

Con las pruebas realizadas se logró brindar un aporte para la mejora de la funcionalidad y rendimiento del sistema, dado que en la ejecución de las mismas, se encontraron errores que fueron solucionados por el equipo de desarrollo.

Como resultado de esta tesina, debemos destacar que hemos podido establecer un modelo que nos permite seleccionar las herramientas y un procedimiento para la ejecución de pruebas de unidad y de rendimiento.

Capítulo 12: Trabajo a futuro

Pensamos que esta tesina podría permitir disparar nuevas líneas de trabajo relacionadas. Las líneas que proponemos son:

1. Revisar periódicamente las planillas presentadas en el capítulo 7, y de ser necesario agregar nuevas herramientas, dado el gran movimiento de la comunidad de Software Libre.
2. Seguir probando el sistema con diferentes herramientas, aquellas que no han sido incluidas en esta tesina, y aplicarlas a los diferentes tipos de pruebas, y así poder realizar un plan de pruebas más completo.
3. Se podría extender la tesis agregando pruebas de usabilidad y accesibilidad, focos que no fueron tomados en este trabajo.

Índice de figuras

Figura 1: Modelo en V. 13
Figura 2: Costos relativos de resolución según la etapa en la que se encuentre el error. 25
Figura 3: BSoD (pantallazo azul de la muerte) presentación Windows 98..... 29
Figura 4: BSoD en la inauguración de los Juegos Olímpicos de verano de Beijín 2008..... 29
Figura 5: BSoD en cartel de publicidad. 30
Figura 6: Transición de estados. 32
Figura 7: Grafo de flujo. 43
Figura 8: Modelo en V definiendo los tipos de herramientas..... 65
Figura 9: página de inicio de Kimkëlen. 94
Figura 10: pantalla del menú de alumnos..... 94
Figura 11: pantalla del listado de alumnos. 95
Figura 12: Fixture students.propel.yml. 105
Figura 13: Prueba StudentTest.class.php. 107
Figura 14: Ejecución de los test..... 108
Figura 15: Función testReRegisterToSchoolYear. 108
Figura 16: JMeter opción Arrancar. 117
Figura 17: Ejecutando JMeter..... 117
Figura 18: Petición HTTP. 118
Figura 19: Errores en el Servidor Apache. 119
Figura 20: Errores en las peticiones HTTP..... 120
Figura 21: Modificador de Parámetro de Usuario HTTP. 123
Figura 22: archivo users.xml..... 124
Figura 23: MySQL Log Prueba de 40 usuarios. 125
Figura 24: MySQL Log Prueba de 50 usuarios. 126
Figura 25: Configuración del CSV data set. 129
Figura 26: archivo login.csv. 129
Figura 27: Petición HTTP donde se parametrizan los datos de entrada. 130
Figura 28: archivo MySQL Log. 131
Figura 29: Consulta MySQL. 132
Figura 30: Petición HTTP. 135
Figura 31: Errores al intentar completar una solicitud. 137
Figura 32: Errores al intentar completar una solicitud. 140
Figura 33: Errores al intentar completar una solicitud. 142

Índice de tablas

Tabla 1: Comparación de licencias..... 20
Tabla 2: Plantillas para la selección de herramientas. 60
Tabla 3: Plantilla para la selección de herramientas para la gestión de pruebas..... 72
Tabla 4: Plantilla para la selección de herramientas para pruebas estáticas de código. 76
Tabla 5: Plantilla para la selección de herramientas para pruebas de unidad. 80
Tabla 6: Plantilla para la selección de herramientas para pruebas de rendimiento. 83
Tabla 7: Plantilla para la selección de herramientas para pruebas funcionales. 87
Tabla 8: Plantilla para la selección de herramientas para pruebas funcionales – Chequeadoras de enlaces. 90
Tabla 9: Resultados de las pruebas. 116
Tabla 10: Análisis de los resultados de las pruebas..... 116
Tabla 11: Valores correspondientes al acceso concurrente al home. 119
Tabla 12: Valores correspondientes al acceso concurrente al home con mejora en la memoria. 121
Tabla 13: Valores correspondientes al tiempo promedio y total. 121
Tabla 14: Valores correspondientes al acceso con delay de 1 seg. entre cada salida de usuario al home. 122
Tabla 15: Valores correspondientes al tiempo promedio y total. 122
Tabla 16: Valores correspondientes al acceso concurrente con login. 125
Tabla 17: Valores correspondientes al tiempo promedio y total. 126
Tabla 18: Valores correspondientes al acceso concurrente con login. 127
Tabla 19: Valores correspondientes al tiempo promedio y total. 127
Tabla 20: Valores correspondientes al acceso concurrente con login. 131
Tabla 21: Valores correspondientes al tiempo promedio y total. 133
Tabla 22: Valores correspondientes al acceso con delay con login. 133
Tabla 23: Valores correspondientes al tiempo promedio y total. 134
Tabla 24: Valores correspondientes al acceso concurrente al sistema con visualización del listado. 136
Tabla 25: Valores correspondientes al tiempo promedio y total. 136
Tabla 26: Valores correspondientes al acceso con delay al sistema con visualización del listado. 137
Tabla 27: Valores correspondientes al tiempo promedio y total. 138
Tabla 28: Valores correspondientes al acceso concurrente guardando la asistencia de los alumnos. 139

Tabla 29: Valores correspondientes al tiempo promedio y total. 140
Tabla 30: Valores correspondientes al acceso con delay guardando la asistencia de los alumnos. 141
Tabla 31: Valores correspondientes al tiempo promedio y total. 142