IVÁN CORDERO VARGAS

CONTROL DE CALIDAD EN *SOFTWARE*

Material complementario

UNED
UNIVERSIDAD ESTATAL A DISTANCIA
Institución Benemérita de la Educación y la Cultura

Producción académica

y asesoría metodológica

Mario Marín Romero

Revisión filológica

Vanessa Villalobos Rodríguez

Diagramación

Mario Marín Romero

Encargado de cátedra

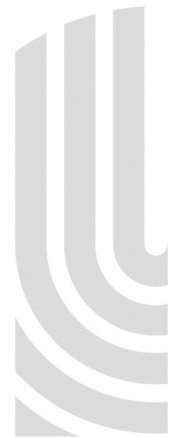
Xinia Chacón Ballestero

Esta guía de estudio se confeccionó en la UNED, en el año 2011, para ser utilizada en la asignatura "Control de calidad en *software*", código 03094, que se imparte en el programa de Ingeniería Informática.

Universidad Estatal a Distancia

Vicerrectoría Académica

Escuela de Ciencias Exactas y Naturales

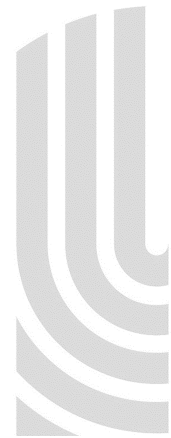


PRESENTACIÓN

Este documento define conceptos básicos para el control de calidad del *software*, definición de estrategias y uso de buenas prácticas. La prosa utilizada explica de manera muy natural los temas, con un lenguaje técnico, pero entendible a la vez. También, le brinda a usted ejemplos y gráficos que le ayudan a interpretar mejor el tema en cuestión.

El documento, además de explicar conceptos, ofrece soluciones tangibles de herramientas útiles que puede tomar en cuenta para su organización; por ejemplo, las informáticas, licenciadas u *open source*.

Por la manera en que se invita a la mejora continua, el texto le motivará a seguir mejorando e investigando nuevas soluciones, y por supuesto, a tomar muy en cuenta las recomendaciones ofrecidas en él.



CONTENIDOS

PRESENTACIÓN iii

MÓDULO I. HERRAMIENTAS BÁSICAS PARA EL CONTROL DE LA CALIDAD 1

Tema 1. Herramientas de *testing* 2

1.1 Generación y definición de datos de prueba 3

1.2 Herramientas de seguimiento de defectos y testing 11

Tema 2. Estrategias de *testing* 35

2.1 Costos del testing 35

2.2 TDD y TAC 42

2.3 Refactorización del código 45

Ejercicios de autoevaluación 47

MÓDULO II. ACTIVIDADES DE LA CALIDAD TOTAL 49

Tema 3. Tipos de pruebas 50

3.1 El ciclo de vida de pruebas según RUP 50

3.2 Etapas de pruebas según RUP 54

3.3 Cobertura de pruebas 56

3.4 Ideas de pruebas 56

3.5 Tipos de pruebas específicas 57

Tema 4. Validación y verificación de *software* 63
4.1 Validación 63
4.2 Verificación 63
4.3 Participación temprana en el proyecto 63
4.4 Insumos de entrada para la validación y verificación de *software* 66
4.5 Planificación de la validacióin y verificación 67
4.6 Técnica de verificación: análisis estático y automatizado de código 69
4.7 Inspecciones de *software* 70

Ejercicios de autoevaluación 71

MÓDULO III. FUNDAMENTOS DE LAS PRUEBAS DE *SOFTWARE* 73

Tema 5. Testing de aplicaciones web 74
5.1 Test de funcionalidad 75
5.2 Test de usabilidad 77
5.3 Test de interfaz 79
5.4 Test de compatibilidad 80
5.5 Pruebas de rendimiento 82
5.6 Pruebas de seguridad técnica 83

Tema 6. Pruebas automatizadas 84
6.1 Importancia 85
6.2 Ventajas 86
6.3 Desventajas 88
6.4 Herramientas automatizadas: desarrollo de una prueba 89

Ejercicios de autoevaluación 114

RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN 115

REFERENCIAS 123

HERRAMIENTAS BÁSICAS PARA EL CONTROL DE LA CALIDAD

I

Sumario

- Herramientas de *testing*
- Estrategias de *testing*

Objetivos

Al finalizar el estudio de este módulo, entre otras habilidades, usted será capaz de:

- Identificar los componentes necesarios de entrada y salida en un proceso de pruebas.
- Utilizar herramientas que existen en el mercado, licenciadas u *open source*, que le ayudarán a gestionar mejor el control de calidad del *software*.
- Aplicar estrategias relevantes de *testing* en sus organizaciones.

Introducción

En este módulo se presenta una descripción de las herramientas básicas para el control de la calidad de *software*. Mediante su estudio, usted descubrirá conceptos, estrategias y técnicas de pruebas, también, sabrá cómo definir qué elementos se deben incluir para el control de *software*, y encontrará tecnologías automatizadas para el *testing* y su seguimiento. Estas herramientas en conjunto le ayudarán a formarse un criterio claro acerca de lo existente en el mercado, y será de gran utilidad para formar su metodología de mejoramiento de calidad en sus organizaciones.

Tema 1. Herramientas de *testing*

En este tema se explican los insumos y generadores mínimos (entradas y salidas) que debe tener un proceso de pruebas de *software*; por ende, le permitirá al estudiante entender su importancia en el Ciclo de Vida de Desarrollo de *Software* (SDLC por sus siglas en inglés) y lo capacitará sobre dicho proceso.

Por último, se muestran algunas herramientas existentes en el mercado y se brinda un *background* (conocimiento general básico) que le inste a continuar investigando y descubriendo herramientas para la mejora continua del control de calidad de *software* en sus organizaciones.

1.1 Generación y definición de datos de prueba

Al preparar una prueba de *software*, es necesario saber qué se probará, entender los requerimientos solicitados por el cliente, usuarios o negocio (usuarios dueños del sistema y conocedores de la operativa), así como conocer también la arquitectura y plataforma sobre la cual se ejecuta el sistema.

Para iniciar con la preparación de la prueba, y estar al tanto del *background*, qué se probará y bajo qué contexto, primero identifique:

- ✓ los eventos y los elementos dinámicos del sistema
- ✓ los límites e interfaces del sistema
- ✓ los elementos de la infraestructura de pruebas

Identificación de elementos y eventos dinámicos del sistema

Este apartado se refiere a utilizar, analizar y comprender la documentación disponible de los requerimientos y diseño con que se desarrolló el *software*, tales como: diagrama de diseño, modelo entidad-relación de la base de datos, diagramas de clases del Lenguaje Unificado de Modelado (UML, por sus siglas en inglés, *Unified Modeling Language*), casos de uso, etc.

Identificación de los límites e interfaces del sistema

El encargado de pruebas debe hacer una exploración para comprender mejor, más allá de los límites del *software*, las expectativas de los sistemas relacionados. Este conocimiento le dará al encargado de pruebas una comprensión profunda de qué necesita, tanto en términos de validación de interfaces como en términos de la infraestructura de pruebas requerida para probar y, posiblemente, simular estas interfaces.

Identificación de los elementos de la infraestructura de pruebas

El ambiente de prueba en que la infraestructura se determina. Para un proceso exitoso, es básico identificarla y mantenerla apropiadamente. La infraestructura adecuada de pruebas es necesaria para el proceso de pruebas, tanto automatizado como manual.

Una vez que se tengan las definiciones previas, se identifican las entradas y salidas de este. Todo proceso tiene insumos, ejecuta dichos insumos y genera los resultados.

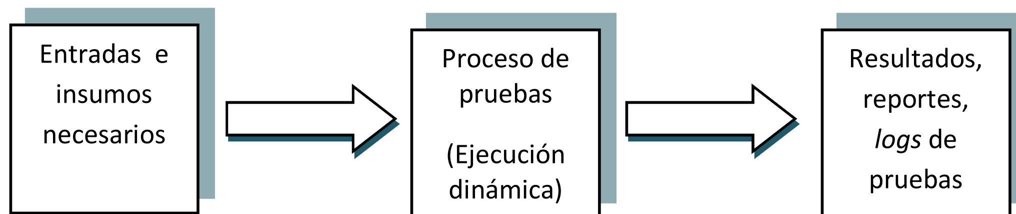


Figura 1. Proceso de pruebas

Entiéndase como *artefacto* todo componente tangible e identificable, que se utiliza como parte una solución informática, por ejemplo: un documento de requerimiento, un archivo *.dll*, *.com* o *.exe*, o de cualquier otra extensión, un diagrama, un caso de uso, un modelo *entidad-relación*, el diagrama de una topología de red, *scripts* de base de datos, etc.

A continuación, se detallan algunos artefactos mínimos de entrada y salida de un proceso de pruebas:

Artefactos de entrada	Artefactos de salida
✓ *datos de prueba* ✓ *caso de prueba* ✓ *plan de pruebas* ✓ *modelo de Diseño* ✓ *guías para pruebas* ✓ *configuración de ambiente de pruebas* ✓ *script de pruebas* ✓ *componente de Pruebas*	✓ *logs o Bitácora de pruebas* ✓ *resultado o reporte de las pruebas* ✓ *solicitud de cambios o mejoras* ✓ *hallazgo de defectos* ✓ *actualización de documentos de pruebas*

Tabla 1. Entradas y salidas de un proceso de pruebas

Un *script* de base de datos es una instrucción o sentencia escrita por el usuario o generada automáticamente por un *software,* con órdenes lógicas y estructuradas para realizar acciones de consulta, inserción, eliminación o actualización de registros en las bases de datos. Estas sentencias son escritas en lenguaje SQL (por sus siglas en inglés *Structured Query Language*) y el motor de base datos lo interpreta para ejecutar la acción.

Artefactos de entrada

Datos de prueba: son necesarios para preparar una prueba de *software*. Asimismo, determinan los escenarios de pruebas, reinician claves de usuarios por utilizar y definen variables iniciales.

Los datos de prueba son la definición (usualmente formal) de la colección de valores de entrada para pruebas, que son consumidos (se utilizan) durante la ejecución de la misma.

Antes de la aplicación de los datos de pruebas, se deben tomar en cuentas las precondiciones requeridas de la configuración del ambiente, para que la prueba sea válida.

El conjunto de datos de pruebas puede almacenarse, desde un simple archivo de texto ASCII (por sus siglas en inglés *American Standard Code for Information Interchange*), libros de MS-Excel o *scripts*, hasta en un complejo ambiente de base de datos, en el cual la información requerida para un escenario específico de pruebas se pueda acceder al instante.

Plan de pruebas: se refiere a la definición de las metas y los objetivos de las pruebas dentro del alcance del proyecto. También, se deben definir el enfoque por utilizar, los recursos solicitados y los entregables que se producirán.

El propósito del plan de pruebas es delinear y comunicar el intento de un esfuerzo de pruebas para un cronograma dado. En primer lugar, como con otros documentos de planeamiento, el objetivo principal es ganar aceptación y aprobación. El documento no detallará lo que no se puede entender, o lo que sería considerado irrelevante en el esfuerzo de las pruebas.

El plan de pruebas especifica los tipos de pruebas que se aplicarán al proyecto; por ejemplo, funcionales, de rendimiento y de regresión o aceptación.

Caso de prueba: se refiere a un conjunto específico de entradas de pruebas, con las condiciones de su ejecución y los resultados esperados, a fin de evaluar algún aspecto particular en el *software*.

El propósito de los casos de prueba es identificar y comunicar, formalmente, las condiciones específicas que serán validadas, para evaluar aspectos particulares de la aplicación (el *software* desarrollado). Los casos de prueba podrían ser motivados por muchas razones pero, usualmente, incluirán un subconjunto de requerimientos.

Un caso de prueba puede fijarse en un documento, en una herramienta de *testing*, o en una prueba automatizada.

Modelo de diseño: describe la realización de los requerimientos del *software*, y abstrae el código fuente (modelo de implementación).

Por ser una abstracción, es una herramienta usada temporalmente para llegar al código fuente (sentencias programadas que cuando se ejecutan conforman la aplicación), y sirve para que el encargado de pruebas conozca cómo está implementado el *software* y pueda definir escenarios con base en él.

El modelo de diseño se usa para concebir el diseño detallado del sistema de *software*. Mejor dicho, se trata de un artefacto amplio y compuesto que contiene todas las clases, subsistemas, paquetes y las relaciones entre estos; insumo esencial para las pruebas y el control de calidad.

Guías de prueba: son documentos de cualquier proceso de control y promulgación de decisiones, estándares añadidos, o guías de buenas prácticas seguirán por los encargados de pruebas.

Las guías de prueba tienen dos propósitos:

- ✓ Ajustar registros (a menudo tácticos) durante la promulgación del proceso.
- ✓ Capturar las prácticas específicas descubiertas en un proceso.

Las guías funcionan como un insumo necesario de pruebas, que ayudan al encargado a tomar en cuenta aspectos relevantes y otros que no se deben tolerar para realizar el proceso de manera correcta.

Configuración de ambiente de pruebas: se refiere a la definición específica de *hardware*, *software*, así como a las propiedades del ambiente, asociadas y requeridas para dirigir adecuadamente las pruebas que habiliten la evaluación de los elementos objeto de estas.

Cada configuración proporciona un escenario adecuado y controlado para dirigir las pruebas requeridas y las actividades de evaluación. Ahora bien, el suministro de un ambiente conocido y controlado en el cual conducen estas actividades, asegura que los resultados sean exactos, válidos, y tengan una alta probabilidad de ejecutarse en el ambiente de producción (ambiente real). Una configuración del ambiente de pruebas bien controlada es un aspecto esencial del análisis eficiente de fallas y resolución de defectos.

***Script* de pruebas:** se refiere a las instrucciones, paso a paso, del proceso que se efectúa en cada una de las pruebas, para habilitar su ejecución. Los *scripts* pueden tomar la forma de instrucciones textuales documentadas que se ejecutan manualmente, o bien pueden ser interpretadas por la computadora que habilita la ejecución de pruebas automatizadas. Estos *scripts* de pruebas son, básicamente, lo que se debe ejecutar, para probar un flujo básico o alterno de un caso de uso.

El propósito de un *script* es proporcionar la implementación de un subconjunto de pruebas, requeridas de una manera eficiente y efectiva.

Componentes de pruebas: se refieren a las soluciones, aplicaciones o proyectos de *software* desarrollados, que:

- ✓ Contienen diversas clases de pruebas.
- ✓ Poseen las pruebas unitarias automatizadas.
- ✓ Verifican el funcionamiento de un componente de sistema (sus clases y métodos).

En un proyecto de *software*, al crear un nuevo componente, se debe generar también su correspondiente homólogo de pruebas unitarias. Si se modifica uno sin prueba asociada, se debe crear uno nuevo y generar la prueba unitaria que valide los cambios realizados.

Artefactos de salida

***Logs* o bitácora de pruebas:** se refiere a la evidencia de haber ejecutado los casos de prueba. Un *log* de pruebas puede generarse de forma automática, por medio de una herramienta (ejemplo *TestManager*, *FitNesse*, etc.), o bien de forma manual, mediante de un documento diseñado por el encargado de pruebas. El *log* de pruebas debe contener, al menos, las fechas de inicio y final, así como indicar el lapso de la ejecución de la prueba. La bitácora es necesaria porque funciona como insumo de estimación de proyectos futuros. El *log* debe guardarse en un repositorio (base de datos de *logs* de pruebas) controlado.

Resultado o reporte de las pruebas: se refiere al resultado que arroja la ejecución de los casos de prueba (una prueba es exitosa si encuentra un defecto). Un resultado de pruebas puede generarse a partir de la ejecución de diferentes tipos de test; entre otros, los hay de:

- ✓ regresión
- ✓ funcionalidad
- ✓ aceptación
- ✓ caja negra
- ✓ unitario
- ✓ carga
- ✓ estrés
- ✓ seguridad
- ✓ concurrencia

El reporte de pruebas debe especificar el escenario ejecutado, el resultado esperado (A) y el resultado obtenido (B), así como la fecha y hora. Si al finalizar la prueba, A es igual a B, concluye la ejecución; si son diferentes, cada resultado de prueba debe tener un código identificador para darle seguimiento a su resolución. Lo inesperado se reporta luego como hallazgos.

Solicitud de cambios o mejoras: es un elemento que puede surgir cuando se ejecutan las pruebas; es la materialización de un requerimiento técnico o de información. Estas pueden ser identificadas por los clientes internos o externos del servicio. Generalmente, una mejora surge a partir de la ejecución de las pruebas de aceptación, es decir, cuando el usuario final (quien utilizará el sistema) se entera de que realmente lo que necesita es otra cosa.

Las *solicitudes de cambio* pueden darse en cualquier otro tipo de test o, inclusive, con el objeto en producción.

También, deben ser escaladas al líder del servicio o sistema (junto con el líder del negocio), para que se analice el impacto que tendrá, y para que se analice el efecto tecnológico que conlleva.

Hallazgo de defectos: aunque lo ideal es evitarlos desde el inicio de un proyecto para ahorrar costos, encontrar defectos es uno de los objetivos principales del proceso de control de calidad del *software*, y esto debe suceder antes del pase a la producción. Los defectos pueden hallarse en cualquier tipo de test, y deben documentarse con todos los detalles, ya sea de forma manual o por medio de una herramienta automatizada, con un identificador para darle seguimiento a su resolución.

Actualización de documentos de pruebas: cuando se realiza un proceso de control de calidad de *software*, para proyectos existentes y nuevos, puede que el encargado de pruebas identifique la necesidad de actualizar sus casos, guías, escenarios automatizados, creación de nuevas ideas de test, etc. Esta es una labor importante en el proceso para que el control de calidad de *software* siempre se ejecute con la información actualizada.

1.2 Herramientas de seguimiento de defectos y *testing*

La importancia de la utilización de herramientas automatizadas es vital para gestionar grandes procesos de control de calidad del *software*, esto mejorará su eficiencia y eficacia. A continuación, se describirán las de mayor uso.

1.2.1 Herramientas de Rational

Rational ClearCase

Este recurso soluciona la administración de la configuración del *software* (control adecuado del *versionamiento* de los elementos de *software*); es decir, se trata de una gestión esencial del control de calidad.

Antes de empezar a usar *ClearCase*, se requiere conocer si usted usará *UCM ClearCase*, un modelo de uso predeterminado, o *ClearCase* base, el cual provee un conjunto de herramientas que pueden usarse para construir otros modelos de uso.

También, se debe crear un espacio de trabajo; este es similar al "espacio de trabajo privado" de los desarrolladores, donde ellos pueden desplegar y cambiar código en un área controlada. El espacio de trabajo de integración es donde los desarrolladores (integradores) del sistema y subsistema se convencen a sí mismos que los componentes creados y probados, en forma separada, pueden construirse y trabajar juntos como un producto.

Para administrar efectivamente la configuración, es necesario la gestión de líneas base. En *UCM* de *Rational ClearCase*, una línea base es un objeto que típicamente representa una configuración estable de un componente. Además, identifica una versión de cada elemento en un componente, de hecho, actúa como una única versión.

En el modelo *UCM* de *ClearCase*, las modificaciones a fuentes son capturadas en forma de actividades de *UCM*. Esta última se constituye por un conjunto de cambio, el cual identifica todas las versiones creadas, mientras se trabaja en una tarea.

Si busca que el trabajo de su área esté disponible para el equipo de proyecto, entregue versiones asociadas con actividades *UCM* desde el flujo de desarrollo hasta el de integración del proyecto.

ClearCase asocia las versiones que se entregan desde el flujo de desarrollo (cuando se desarrolla el *software*) con las versiones en el flujo de integración (cuando se une o integra el *software*), según se necesite. Sin embargo, los cambios que se entregan no se hacen permanentes en este punto, de hecho, se prueban y se verifica la entrega con otro trabajo en el flujo de integración. Después de probarlos, puede cancelar la operación de entrega o completarla, haciendo permanentes los resultados de la entrega.

En el modelo de *UCM*, las actividades (trabajo) entregadas, desde múltiples fuentes, se integran y se organizan en líneas base. Por lo general, las líneas se someten a un proceso de pruebas y de corrección de defectos hasta alcanzar un nivel de estabilidad satisfactorio. Cuando una línea base logra este nivel, su administrador de proyecto la designa como la línea base recomendada.

Otro aspecto importante de *ClearCase* es el establecimiento de políticas (reglas). Las políticas de proyecto permiten la ejecución de buenas prácticas de desarrollo entre un equipo. Al establecerlas se puede, por ejemplo, minimizar problemas cuando integra trabajo, mediante una política que requiera que los desarrolladores actualicen sus áreas de trabajo con las líneas base recomendadas. Esta práctica reduce la posibilidad de introducir defectos en el código.

Una línea base identifica actividades y una versión de cada elemento visible en uno o más componentes. *ClearCase* posse una herramienta llamada *Component Tree Browser*, es una interfaz gráfica de usuario (*Graphic User Interface*-GUI) que despliega la historia de la línea base de un componente; se usa para comparar los contenidos de dos líneas base.

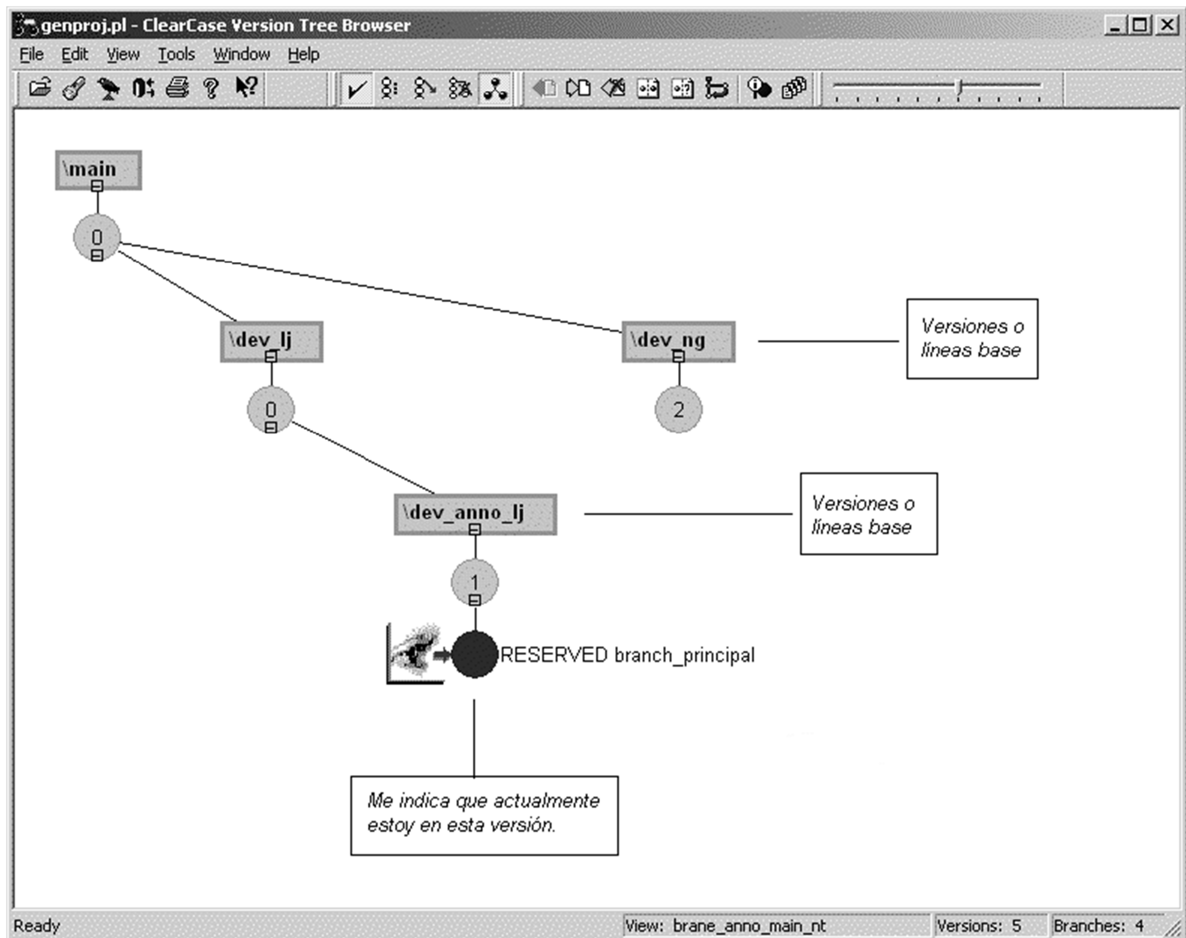


Figura 2. Administración de la configuración en ClearCase

Rational ClearQuest

Herramienta óptima para el seguimiento de defectos y para la administración de solicitudes de cambio.

Por un lado, almacena las solicitudes de cambio en una base de datos. Por otro lado, el administrador puede crear distintos tipos de registros para diferentes propósitos de un proyecto. Además, cada tipo de registro puede tener campos únicos y requerimientos de datos.

ClearQuest le ofrece al usuario una manera sencilla para enviar, modificar, rastrear y seguir la trayectoria de las solicitudes de cambio, conforme estas se mueven a través del sistema de gestión de cambios.

El administrador de *ClearQuest* también puede crear un conjunto a la medida de tipos de registros. Entre estos últimos, se puede usar: registro de defecto, registros para solicitudes de cambio o peticiones de mejora, o registro de solicitudes de documentos.

La herramienta *ClearQuest* provee reportes predefinidos (el administrador también puede personalizar los suyos) que les permiten ver el estado de un proyecto.

Estas estadísticas y reportes que les ofrece a los administradores responde a preguntas como

- ✓ ¿Cuántos defectos han estado abiertos por menos de una semana?
- ✓ ¿Cuántos defectos han estado abiertos por más de tres semanas?
- ✓ ¿Cuántos defectos han estado pospuestos por más de dos meses?
- ✓ ¿Cuántos defectos han sido resueltos?
- ✓ ¿Cuántos defectos están en proceso de pruebas?

Este recurso de *Rational* es un sistema de administración de solicitud de cambio (*change-request management*-CRM) para la naturaleza dinámica e interactiva del desarrollo de *software*. Con *ClearQuest*, se controlan las actividades de cambio, asociadas al desarrollo de *software*, incluso, solicitudes de mejoras, reportes de defectos, y modificaciones a la documentación.

La herramienta no se limita a la rastreabilidad de defectos; posee una herramienta llamada *ClearQuest Designer* que se emplea para personalizar los esquemas (flujos de trabajo) según las necesidades de una compañía. Por ejemplo, el administrador define el tipo de información que el usuario envía, recupera y observa en reportes. Luego determina cuáles usuarios pueden realizar acciones sobre los registros, y a cuáles de estas acciones se les permite acceder.

En esta herramienta, se aprovecha el correo electrónico de dos maneras: envío/modificación (desde el usuario hacia la base de datos), y notificación (desde la base de datos hacia el usuario). También, el administrador de *ClearQuest* puede configurarlo para enviar correos electrónicos que notifiquen a los usuarios acerca de solicitudes de cambio. El administrador usa el tipo de registro llamando *E_Mail Rules* para determinar las condiciones de la notificación; asimismo establece los receptores de la notificación y su contenido.

ClearQuest facilita a los usuarios especificar, mediante una consulta, qué tipo de registros extrae desde la base de datos de esta herramienta. Si un usuario ejecuta una consulta sin definir criterios de selección (filtros), *ClearQuest* enlista todos los registros en la base de datos. A continuación, se mencionan dos ejemplos de consultas típicas:

- ✓ Consulta de los desarrolladores sobre defectos que les han sido asignados para corregir.
- ✓ Consulta de administradores de proyecto sobre los registros, se filtran por la fecha en que fueron creados, para monitorear el progreso de quienes realizan las pruebas.

ClearQuest ejecuta la consulta y luego despliega un conjunto de resultados. Un miembro del equipo puede examinar dicho conjunto y selecciona cuáles registros ver, modificar o usar como base para un reporte o cuadro.

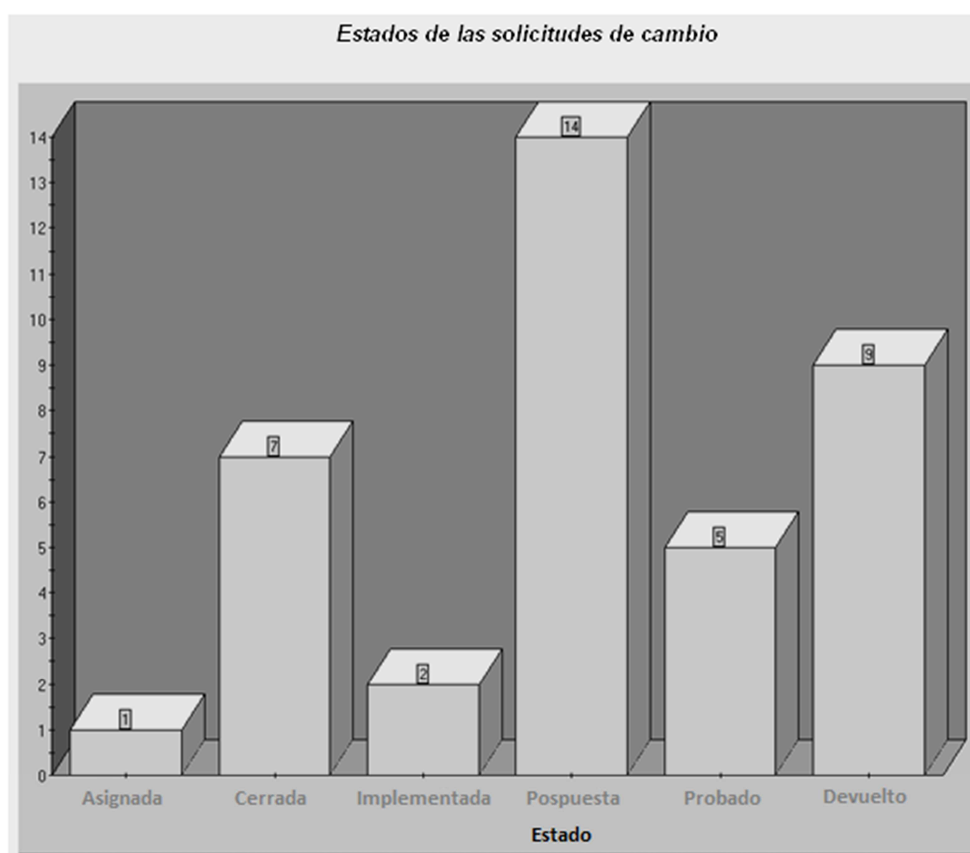


Figura 3. Cantidad de solicitudes de cambio por estado en forma gráfica

Rational TestManager

Es la herramienta de *Rational* que administra y controla las actividades relacionadas con las pruebas.

Por ejemplo, en *Rational TestManager* se administran las siguientes actividades: planeamiento, diseño, implementación, ejecución y análisis. Además, integra las pruebas con el resto de actividades de desarrollo, a fin de proveer un único punto para visualizar el estado exacto del proyecto.

Por otro lado, *TestManager* provee un *suite* o representación jerárquica de la tarea y de la carga de trabajo que se desea ejecutar y probar. En él, se muestran elementos tales como grupos de computadoras, recursos asignados a cada uno de estos grupos, los *scripts* de pruebas que ejecutan los grupos de computadores y cuántas veces se efectúan.

Cuando se lleva a cabo un *suite*, se provee información específica de tiempo de ejecución; en cada una se prueban los elementos que le fueron asignados. Los resultados se almacenan en bitácoras de pruebas. Después de ejecutarlo, se pueden generar reportes para analizar los datos almacenados en las bitácoras y para desplegar los resultados en forma de gráficos.

En el suite de Rational TestManager, también se ofrece la oportunidad de controlar pruebas de rendimiento; permite ejecutar *scripts* de pruebas y emular las acciones de usuarios reales al acceder a una aplicación multiusuario. Un *suite* puede ser tan simple como simular que un *tester* virtual ejecute un *script* de pruebas (*test script*), o tan complejo como miles de *tester* virtuales que efectúen una variedad de *scripts* de pruebas.

Se puede crear una *suite* de rendimiento de alguna de las siguientes maneras:

- ✓ Usando el *wizard performance testing suite*.
- ✓ Basado en una sesión de *Robot* existente.
- ✓ Usando *blank performance testing suite.*

La herramienta de *Rational* también ofrece la posibilidad de crear planes de prueba, que es una colección de información organizada. En otras palabras, representa el acuerdo y el entendimiento acerca de lo que probará y cuándo se ejecutará. Un proyecto puede tener múltiples planes, los cuales representan diferentes fases o aspectos de un proceso. Los planes de prueba se organizan alrededor de conjuntos de características, funcionalidades o tipos de test.

Al utilizar la metodología de desarrollo de sistemas RUP en *TestManager*, el conjunto de características de funcionalidades se resume en el Caso de uso —documento de requerimientos—, por lo cual se elabora un plan de pruebas según cada caso de uso elaborado.

Cada plan de pruebas puede contener múltiples carpetas de casos de prueba (*test case folders*) y casos de pruebas (flujos básicos y alternos de un Caso de Uso). Además, define todos los elementos clave, tal y como se representan en la jerarquía de las carpetas de casos de prueba.

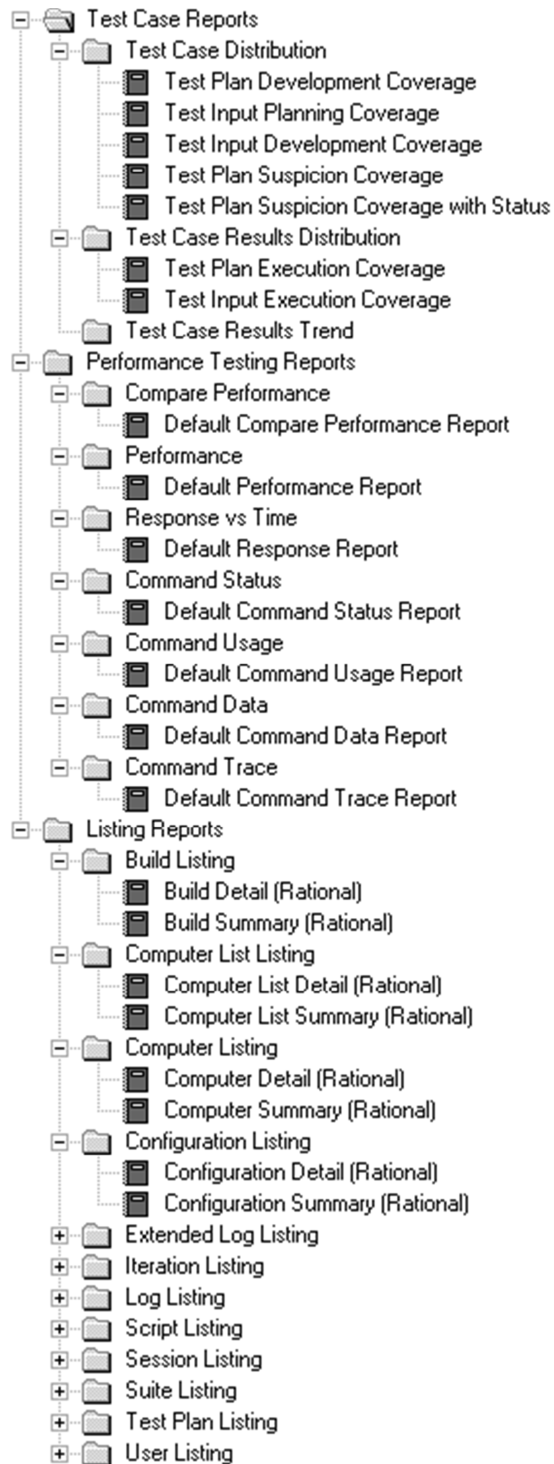


Figura 4. Reportes de TestManager

1.2.2 Herramientas de Microsoft

Microsoft Visual Studio Team System (VSTS)

Microsoft ofrece, a los *testers* (encargados de realizar pruebas) y desarrolladores, la posibilidad de una plataforma para trabajar en la ejecución de pruebas; esa plataforma está inserta en *Visual Studio* (herramienta de desarrollo de *software*) y posibilita los siguientes tipos de test o pruebas:

- ✓ unitarias (*unit test*)
- ✓ manuales (*manual test*)
- ✓ web (*web test*)
- ✓ de carga (*load test*)
    - o de estrés (*stress test*)
    - o de rendimiento (*performance test*)
    - o de capacidad (*Capacity planning test*)
- ✓ generales (*generic test*)
- ✓ de orden (*Ordered test*)

Unit test

Las pruebas unitarias se encargan de probar métodos del código fuente; VSTS ofrece la posibilidad de crear pruebas unitarias utilizando *Test Class* y *Test Method*.

Tan pronto como los desarrolladores terminan un código, es necesario saber si la funcionalidad produce los resultados esperados. Los *unit test* son pruebas dirigidas para los implementadores, que validan la funcionalidad individual y básica de los métodos, antes de pasar el *software* a un proceso mayor de control de calidad.

Manual test

Las pruebas manuales son las más antiguas y más simples, pero aún son cruciales para las de *software*. No se necesita ninguna herramienta automatizada.

VSTS ofrece la posibilidad de crear *manual test* utilizando *Microsoft Office Word* o *Notepad,* lo cual solamente muestra al encargado de pruebas una forma de ejecutar paso a paso el test. VSTS puede tomar en cuenta un *manual test* como parte de los casos de prueba.

Asimismo, recomienda el uso de un *manual test* en los siguientes casos:

- ✓ No hay suficiente presupuesto para la automatización.
- ✓ Las pruebas son muy complicadas o muy difíciles como para convertirse en automatizadas.
- ✓ Las pruebas se ejecutarán solamente una vez.
- ✓ No hay suficiente tiempo como para automatizar las pruebas.
- ✓ Las pruebas automatizadas consumirían mucho tiempo en su creación y ejecución.

Web test

Los *web test* se utilizan para probar la funcionalidad de una página web, *web application*, *web site*, *web services* y la combinación de todos ellos.

Un *web test*, generalmente, se crea mediante la grabación de la iteración del usuario con el *web application* en el *Browser*. También, el *web test* puede usarse para realizar pruebas de rendimiento, se pueden realizar diferentes reglas de validación y de extracción. Estas últimas pueden emplearse para la validación de campos, de textos, y *tags* (etiquetas, bloques), en la página *web* solicitada.

En algún momento, necesitaremos los datos devueltos por la página *web*, los datos a futuro o una colección de datos para la prueba. En esos casos, usamos la extracción de reglas para obtener los datos retornados por la página solicitada. Usando estos procesos, se extraen campos, textos, o valores en la página web y se almacenan en el contexto o colección.

Un *web test* se clasifica en *Simple Web test* y *Code Web test,* ambos son soportados por VSTS.

Simple Web test

Estas pruebas son muy simples de crear y de llevar a cabo. Una vez ejecutadas no se puede intervenir. La desventaja es que no puede ser condicional. Por último, es una serie de flujos validados.

Code Web test

En este caso, son más complejos, pero proveen mucha flexibilidad. Por ejemplo, si necesitamos de alguna condición, las pruebas se pueden generar mediante *C#* o *Visual Basic (*lenguajes de programación*)*; además, si se usa un código programable podemos tener control del flujo de los eventos. La desventaja es que su complejidad y el mantenimiento son muy altos.

Load test

Método de pruebas que se usa en diferentes tipos, su propósito es identificar el rendimiento de una aplicación (sistema, *software*), basado en diferentes escenarios. En las pruebas de carga, se pueden agregar *web test* y *unit test*, para ejecutarse simultáneamente.

Antes de liberar un *web site* a los usuarios, deberíamos chequear el rendimiento de la aplicación, tal que soporte el gran grupo de usuarios que la accederían. En este momento, las pruebas de carga o *load test* son útiles.

Cuando un *web test* se agrega a un *load test*, se simula múltiples usuarios abriendo simultáneas conexiones al mismo web *application* y haciendo varias solicitudes *HTTP*. El resultado de la pruebas puede guardarse en un repositorio para comparar el *set* de resultados y mejorar su rendimiento.

Ordered test

Esta herramienta le ofrece al encargado de pruebas la posibilidad de establecer su orden; en ocasiones se establece qué prueba se ejecuta primero y cuál después, para que el escenario sea el correcto. VSTS ofrece la posibilidad de definir el flujo de pruebas requerido por quien se encarga del test.

Generic test

Existen situaciones los cuales se deben ejecutar pruebas de caja negra (tipos de pruebas cuyo código de la aplicación no conocemos); VSTS ofrece la posibilidad de gestionar los tipos de pruebas de ejecutables y binarios.

1.2.3 Herramientas de Aldon

Las herramientas de Aldon ayudan en la gestión del cambio del *software* (*Software Change Management*) al proveer las siguientes facilidades:

- ✓ *administración de solicitudes de cambio y requerimientos*

  Esta herramienta verifica, aprueba y almacena las solicitudes de cambio de forma automática, así como los requerimientos. Además, proporciona visibilidad del ciclo de vida del desarrollo, y rastrea defectos y solicitudes de cambio (peticiones de mejora). Finalmente, ofrece la opción de rastreabilidad desde un requerimiento hasta la solicitud.

- ✓ *administración de los flujos de trabajo*

  Aldon le permite crear, administrar y personalizar flujos de trabajo, para establecer reglas y condiciones complejas propias de la compañía.

- ✓ *administración de la configuración del* software

  La herramienta genera, organiza y mantiene un repositorio central de los componentes de las aplicaciones, y los ordena según la función del componente en el negocio. Asimismo, se puede acceder a cualquier componente de las aplicaciones de distintos y múltiples proyectos y grupos de trabajo.

- ✓ *administración del deployment (puesta en marcha, liberación en producción)*

  Aldon ofrece la posibilidad de realizar el proceso de *deployment* en forma automática y evita el proceso manual que tiende a cometer errores. Aldon se asegura de que los archivos correctos se liberen de manera automática a las ubicaciones correctas.

La herramienta ofrece una pantalla –GUI– en la cual se administran los *paquetes* (versión de *software* estable por liberar). En ella se pueden distribuir a la ubicación destino adecuado en cualquier etapa, de acuerdo con los parámetros de autorización que corresponda.

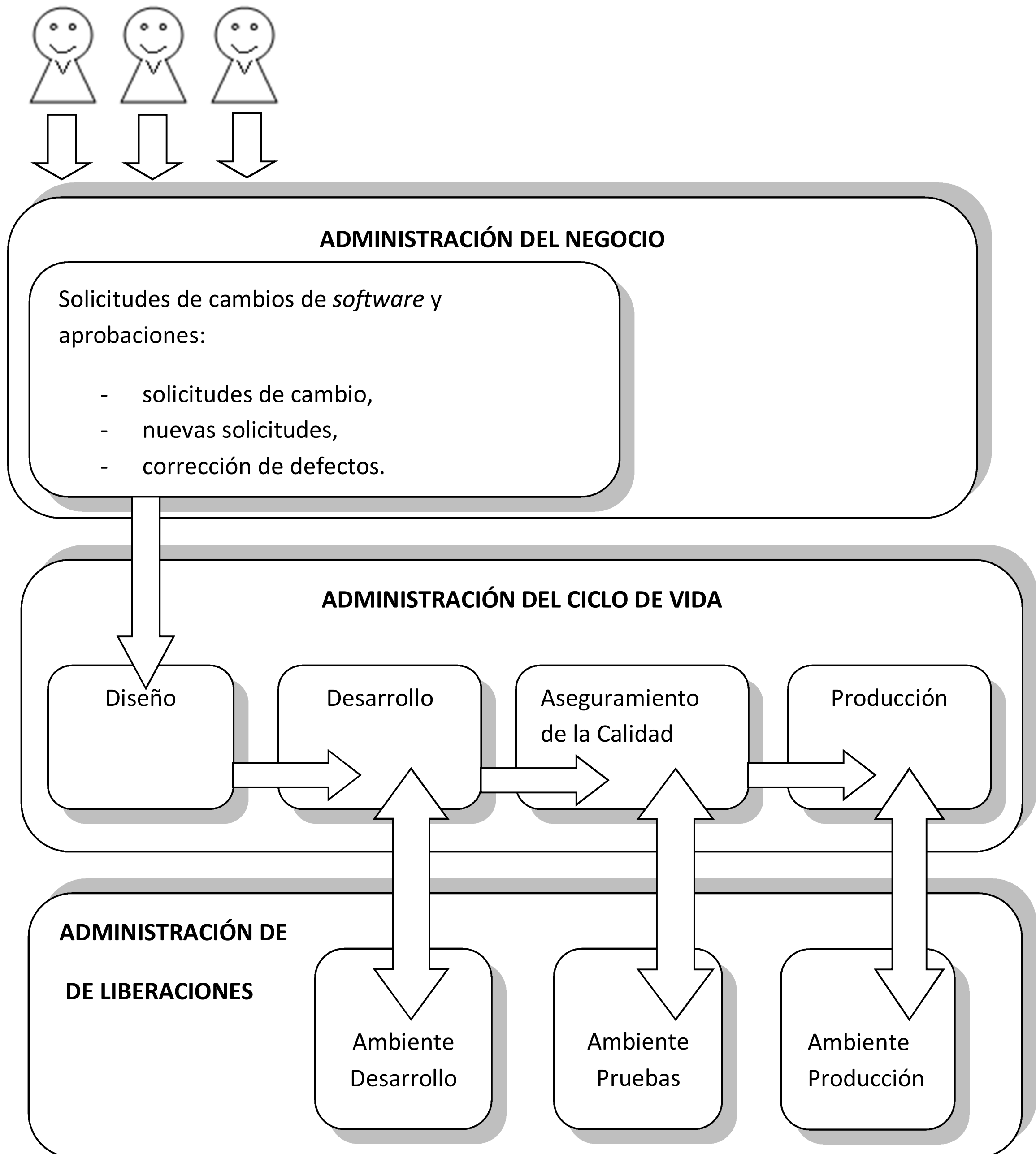


Figura 5. Solución de Aldon

1.2.4 Herramientas Open Source

El *software open source* (código abierto), le ofrece a la comunidad informática el código de su aplicación para ser utilizado, revisado, modificado y mejorado; de esta forma, el *software* se va convirtiendo, cada vez más, en un producto de calidad. El término difiere un poco en relación con lo que es *free software* (libre). En la actualidad, los dos movimientos trabajan en conjunto para obtener un *software* de mejor calidad, con la filosofía de darle a la comunidad informática la posibilidad de leer, modificar y redistribuir el código.

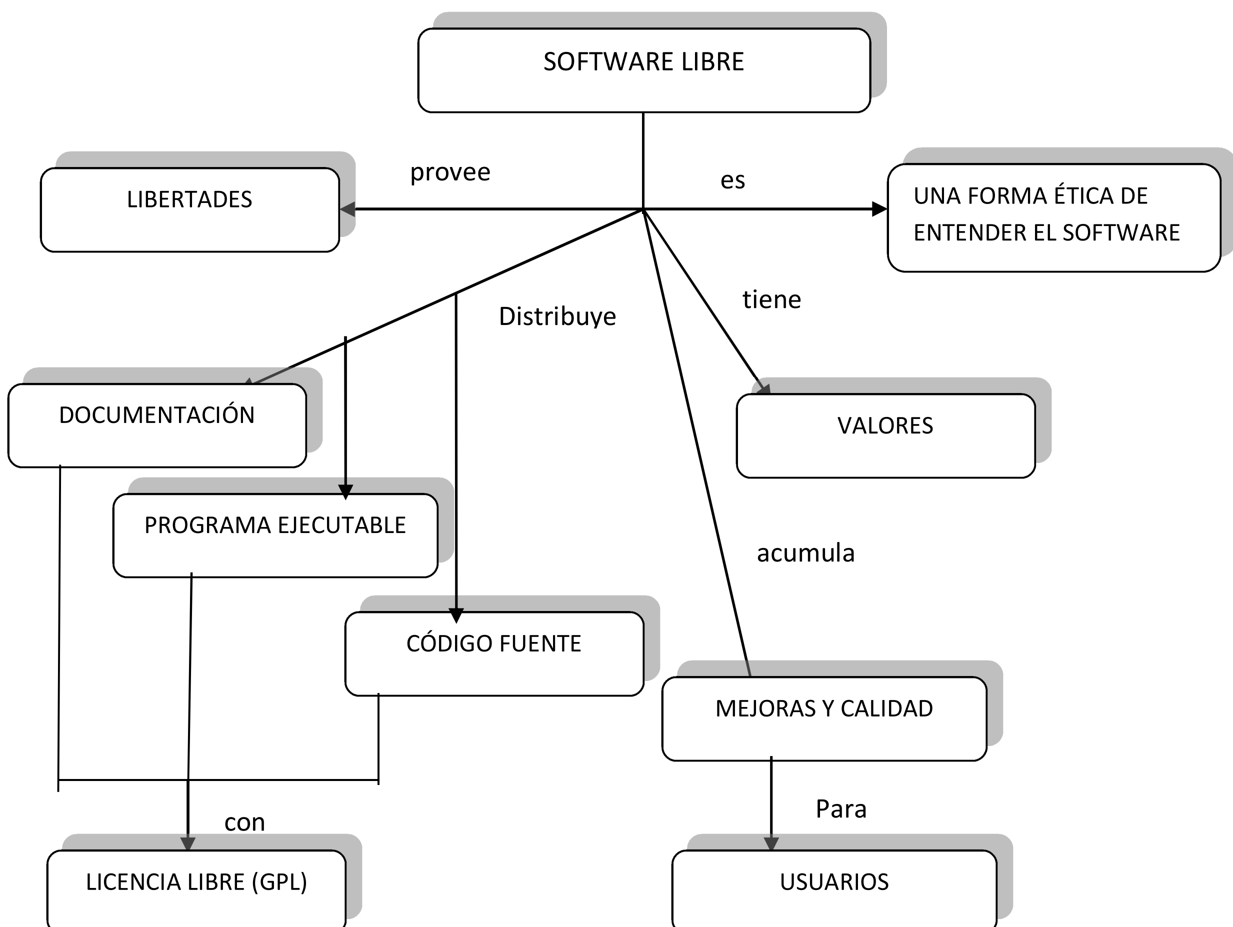


Figura 6. Mapa conceptual del movimiento *software* libre

Herramientas de este tipo son una oportunidad más, a muy bajo costo, que nos ayudan a preparar pruebas tales como:

- ✓ de aceptación
- ✓ estáticas de código
- ✓ unitarias
- ✓ funcionales
- ✓ de rendimiento

A continuación, se ofrece una gran variedad de ellas, que serán de mucha ayuda en las organizaciones para establecer controles de calidad del *software*.

Herramientas para pruebas de aceptación

FitNesse

Herramienta que compara lo que debe hacer el *software* contra lo que realmente hace. Con este instrumento, se realizan pruebas de aceptación y de negocio.

FitNesse permite, a los usuarios finales, insertar en el sistema entradas en un formato especial (propio de este *software*). Estas entradas se interpretan y se crea la prueba automáticamente; luego son ejecutadas por el sistema y el resultado se le muestra al usuario. La ventaja de esta herramienta es que se obtiene una retroalimentación pronta del usuario final.

Los lenguajes soportados para *FitNesse* son:

✓ Java ✓ .NET ✓ C++ ✓ Delphi	✓ Python ✓ Ruby ✓ Smalltalk ✓ Perl

Avignon

Es un *Framework* (plataforma de trabajo) de pruebas de aceptación hecho *in-house (software* desarrollado en casa, no se paga, ni se compra*)* por el equipo de programación de *NOLA Computer Service,* y que utiliza la metodología *eXtreme Programming* (XP). Avignon les permite a los usuarios expresar las pruebas de una manera no ambigua, antes de iniciar su desarrollo.

Avignon utiliza XML (por sus siglas en inglés *Extensible Markup Language*) el cual define el propio lenguaje de pruebas de aceptación. Cada *tag* o bloque del XML tiene una clase *Java* asociada que ejecuta la acción requerida para la parte de esa prueba. Esta plataforma trabaja en conjunto con JUnit, HTTPUnit, etc.

A continuación, se presenta una tabla comparativa entre las dos herramientas anteriores:

Herramienta	Interfaz de usuario	Licencia	Plataforma	Lenguaje	Última actualización	Documentación
FitNesse	Web	GPL	Windows / Linux	Java, C#, PHP, Ruby, .NET, etc.	Julio 2009	Guía de usuario
Avignon	GUI	GPL	Windows / Linux	Java, .NET, etc.	Octubre 2006	Insuficientes

Tabla 2. Comparación básica entre FitNesse y Avignon

Herramientas para pruebas estáticas de código

PHPLint

Es una herramienta que mejora las tareas de programación, ya sea comenzando la codificación con ella, o mejorando código ya existente.

El *PHPLint* permite dar seguridad en el código, encuentra errores de sintaxis, variables no utilizadas, código muerto (código del *software* que nunca será utilizado), etc.

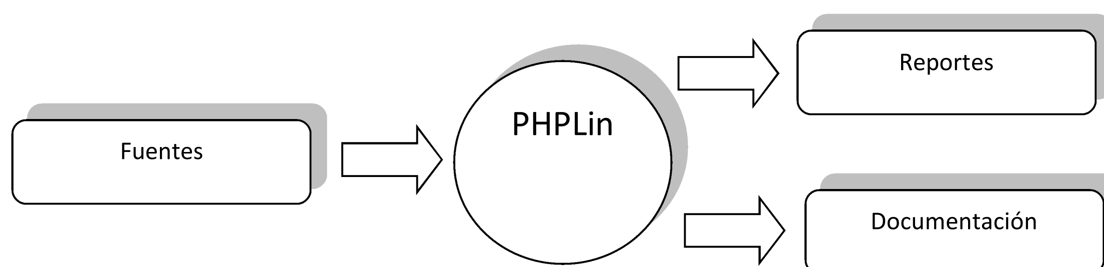


Figura 7. Esquema de trabajo de PHPLint

RATS

Rough Auditing Tool for Security, es una herramienta que chequea la seguridad en el código, y determina la criticidad de fallos, así como su evaluación. Principalmente, chequea (*scanning*) código en C, C++, Perl, PHP y Pyphon.

YASCA

Es una herramienta de análisis de código fuente, escrita por Michael V. Scovetta en el 2007. YASCA puede revisar código escrito en Java, C/C++, HTML, JavaScript, ASP, ColdFusion, PHP, COBOL, .NET, y otros lenguajes.

También, fácilmente, se integra con las siguientes herramientas:

- ✓ FindBugs
- ✓ PMD
- ✓ JLint
- ✓ JavaScript Lint
- ✓ PHPLint
- ✓ CppCheck
- ✓ ClamAV
- ✓ RATS
- ✓ Pixy

PMD

Es una herramienta que puede integrarse a otras como *JDeveloper, Eclipse, JEdit*, etc. *PMD* permite encontrar en el código fuente, errores en el manejo de excepciones, código muerto, sin optimizar o duplicado.

FindBugs

Es una herramienta que puede integrarse a Eclipse, chequea el código fuente y encuentra errores comunes, malas prácticas de programación, código vulnerable, problemas de rendimiento y de seguridad.

FindBugs es una aplicación desarrollada por la Universidad de Maryland y ha sido descargada más de 700 000 veces hasta ahora.

A continuación, se presenta una tabla comparativa entre estas herramientas:

Herramienta	Interfaz de usuario	Licencia	Plataforma	Lenguaje	Última actualización	Documentación
PHPLints	GUI	BSD	Windows / Linux	PHP	Mayo 2009	Tutorial / Manual
RATS	CLI	GPL	Windows / Linux	C++, Perl, PHP, Python	Setiembre 2009	Sin datos
YASCA	CLI	GPL	Windows / Linux	Java, .NET, PHP, HTML, CSS	Mayo 2009	Manual
PMD	CLI	BSD	Windows / Linux	Java	Febrero 2009	Tutorial / Manual
FindBugs	GUI / CLI	GPL	Windows / Linux	Java	Marzo 2009	Tutorial / manual

Tabla 3. Comparación básica entre PHPLint, RATS, YASCA, PMD y FindBugs

CLI: interfaz por línea de comandos, *Command Line Interface*, por sus siglas en inglés.

BSD: las iniciales de *Berkeley Software Distribution*, y se usa para identificar un sistema operativo derivado del sistema *Unix*, a partir de los aportes realizados a ese sistema por la Universidad de California de Berkeley.

Herramientas para pruebas unitarias

JUnit

Herramienta que automatiza las pruebas unitarias y de integración. Provee clases y métodos que facilitan la tarea de realizar pruebas en el sistema y así asegurar la consistencia y funcionalidad.

PHPUnit

Permite crear y ejecutar pruebas unitarias de manera simple, y luego genera los resultados de dichos test; se basa en el *framework JUnit* para *Java*.

SimpleTest

Herramienta para realizar pruebas unitarias en *PHP* y pruebas web. Cuenta con un navegador web interno, esto admite que las pruebas naveguen los sitios, ingresen datos en formularios y páginas de pruebas.

A continuación, se presenta una tabla comparativa entre estas herramientas:

Herramienta	Interfaz de usuario	Licencia	Integración	Última actualización	Documentación
JUnit	Integrada	CPL	Eclipse / NetBeans	Mayo 2009	Foro / FAQ
PHPUnit	CLI	PHP	No aplica	Junio 2009	Manual
SimpleTest	CLI	LGPL	Eclipse	Abril 2008	Tutorial

Tabla 4. Comparación básica entre JUnit, PHPUnit y SimpleTest

Herramientas para pruebas funcionales (verificadores de enlaces)

XENU

Encuentra los enlaces "rotos" (que no tienen acceso) en un análisis a profundidad, revisa *links*, imágenes, *frames*, *plug-ins*, *scripts* de Java, entre otros; al final, genera un reporte al encargado de pruebas, el cual puede ordenar según el criterio que desee.

LINK Checker W3C

Herramienta *on-line* (accedida desde Internet y no es necesario instalarla), encuentra enlaces "rotos", mal definidos, advierte sobre redirecciones, etc.

DRKSpider

Herramienta que encuentra enlaces rotos, tiene niveles de profundidad y genera un árbol jerárquico con los enlaces del sitio en prueba, con información detallada.

Link Evaluator

Herramienta diseñada para ayudar a los usuarios a evaluar la disponibilidad de recursos en línea vinculados a partir de una determinada página web. Cuando se inicia, sigue todos los enlaces en la página actual, y evalúa las respuestas de cada una de las URL/enlaces.

Link Evaluator (enlace evaluador) examina el código de estado HTTP, el contenido de la página devuelta por cada una de las URL, y los intentos de distinguir entre los diversos resultados tales como enlaces rotos, red de tiempos de espera, error de autenticación y contenido correcto o incorrecto de la URL.

A continuación, se presenta una tabla comparativa entre estas herramientas:

Herramienta	Interfaz de usuario	Licencia	Procesamiento	Plataforma	Última actualización	Documentación
XENU	GUI	Freeware	Remoto / Local	Windows	Abril 2009	FAQ
Link Checker W3C	Web	GPL	Remoto	Windows / Linux	No aplica	Manual
DRKSpider	GUI	GPL	Remoto / Local	Windows	Abril 2009	Foro
Link Evaluator	Web	Apache	Remoto / Local	Windows / Linux	Mayo 2009	Ejemplos

Tabla 5. Comparación básica entre XENU, Link Checker W3C, DRKSpider y Link Evaluator

Herramientas para pruebas funcionales (Funcionalidad)

Selenium IDE

Un *plug-in* (módulo de *hardware* o *software* que añade una característica a un sistema más grande) del Firefox; graba *clicks*, editar y realizar pruebas. *Slenium IDE* no es solamente una herramienta de grabación, sino que es un *IDE* completo (*Integrated development environment*, por sus siglas en inglés; se refiere a un ambiente integrado para desarrollo). Con él, efectuará cualquier tipo de pruebas.

Selenium IDE se exporta en distintos lenguajes para su posterior adaptación y uso.

HTTPUnit

Este se basa en la metodología XP (*Extreme Programming*). Se realizan pruebas funcionales antes de que estén generadas las páginas Web. No se basa en los controles de la página, sino en los valores de entrada que el usuario pueda ingresar.

Badboy

Esta herramienta graba y reproduce las acciones hechas por los usuarios; el *script* generado puede usarse en otras herramientas como *JMeter*. Se puede integrar al navegador web Internet Explorer.

SAHI

SAHI graba y reproduce *scripts* para la ejecución de pruebas web. SAHI "corre" en cualquier *browser* (navegador de Internet) que soporte Javascript.

A continuación, se presenta una tabla comparativa entre las herramientas anteriores:

Herramienta	Interfaz de usuario	Licencia	Plataforma	Última actualización	Documentación
Selenium IDE	GUI	Apache	Varios	Junio 2008	Tutorial / Wiki / Manual
HTTPUnit	WEB	Propia	Windows / Linux	Mayo 2008	Tutorial / FAQ / Manual
Badboy	WEB	LGPL	Windows	Diciembre 2008	Foro / Manual
Sahi	GUI	Apache	Windows / Linux	Mayo 2009	FAQ / Manual

Tabla 6. Comparación básica entre Selenium IDE, HTTPUnit, Badboy y Sahi

Herramientas para pruebas de rendimiento

JMeter

Herramienta que efectúa pruebas de rendimiento, estrés, carga y volumen; sobre recursos estáticos o dinámicos.

OpenSTA

Instrumento que permite captar las peticiones del usuario generadas en un navegador web; estas también se guardan y es posible su edición para uso posterior.

WEbLoad

Este realiza pruebas de rendimiento a través de un entorno gráfico, en el cual se pueden desarrollar, grabar y editar *scripts* de pruebas.

Grinder

Un *framework* escrito en Java, con el cual se efectúan pruebas de rendimiento a través de *scripts* escritos en lenguaje *Jython*. Además, graba las peticiones del cliente sobre un navegador web para ser reproducido luego.

A continuación, se muestra una tabla comparativa entre estas herramientas, que ayudan al lector a tomar una mejor decisión:

Herramienta	Interfaz de usuario	Licencia	Plataforma	Última actualización	Documentación
JMeter	GUI	Apache	Windows	Junio 2009	Tutorial
OpenSta	GUI	GPL	Windows / Linux	Octubre 2007	Guía de usuario
Web Loader	GUI	GPL	Windows	Abril 2007	Tutorial
Grinder	GUI	GPL	Windows / Linux	Febrero 2009	Guía de usuario / FAQ

Tabla 7. Comparación básica entre JMeter, OpenSTA, Web Loader y Grinder

En la siguiente lista, se presentan los *links* de interés, relacionados con los utilitarios *OpenSource*, en donde usted podrá obtener más información y realizar la descarga de la aplicación:

FitNesse: http://fitnesse.org/

Avignon: http://www.nolacom.com/avignon/index.asp

PHPLint: http://www.icosaedro.it/phplint/

RATS: http://www.fortify.com/security-resources/rats.jsp

YASCA: http: //www.yasca.org/

PMD: http://pmd.sourceforge.net/

FindBugs: http://findbugs.sourceforge.net/

JUnit: http://www.junit.org/

PHPUnit: http://www.phpunit.de/

SimpleTest: http://www.simpletest.org/

XENU: http: //home.snafu.de/tilman/xenulink.html

LINK Checker W3C: http://validator.w3.org/checklink

DRKSpider: http: //www.drk.com.ar/index.php

Link Evaluator: https://addons.mozilla.org/es-ES/firefox/addon/4094

Selenium IDE: http://seleniumhq.org/projects/ide

HTTPUnit: http://httpunit.sourceforge.net/index.html

Badboy: http://www.badboy.com.au/

SAHI: http: //sahi.co.in/w/

JMeter: http://jakarta.apache.org/

OpenSTA: http: //www.opensta.org/

WEbLoad: http: //www.webload.org

Grinder: http: //grinder.sourceforge.net/

Tema 2. Estrategias de testing

Este tema explica conceptos básicos y útiles de estrategias de *testing* para que usted conozca sobre ellos y se desenvuelva mejor en relación con el tema de control de calidad en cada una de sus organizaciones.

Asimismo, se detallarán conceptos esenciales y se identificarán los atributos de calidad con que debe contar un *software*; además, se analizarán los tipos de pruebas que el encargado de test realiza para garantizar cada uno de esos elementos de calidad. Con ejemplos sencillos, pero muy significativos, usted aprenderá cómo desarrollar una estrategia de pruebas basada en los requerimientos funcionales y no funcionales del *software*.

2.1 Costos del testing

Al igual que el costo de calidad en general, el costo del *testing* en una organización es sinónimo de ahorro, ya que es conveniente producir *software* de calidad para que las devoluciones, las pérdidas de dinero y otros factores, disminuyan.

Si el producto de *software* liberado en producción ha tenido un adecuado proceso de *testing,* los defectos reportados serán mínimos, la atención de incidentes críticos disminuirá, y la asignación de recurso técnico para solventar los problemas de forma inmediata no se dará. De esta forma, la compañía ahorrará sustancialmente en la parte económica.

En una organización, la calidad debe de ser parte fundamental dentro de cualquier proceso y cualquier departamento. Al igual que en la calidad general, en el *testing,* los programas de capacitaciones para los encargados de pruebas son sumamente importantes en el adecuado manejo de los recursos disponibles.

El costo del *testing* es parte del de la calidad del *software* (CoQ), y la calidad del *software* contribuye a que el producto final contenga factores tales como:

- ✓ fiabilidad
- ✓ eficiencia
- ✓ seguridad (Integridad)
- ✓ facilidad de uso
- ✓ facilidad de mantenimiento
- ✓ flexibilidad
- ✓ facilidad de prueba
- ✓ portabilidad
- ✓ reutilización
- ✓ interoperabilidad

2.1.1 Testing de *software* en desarrollo

En todo desarrollo de *software*, un elemento básico en su ciclo de vida es el *testing,* con el fin de garantizar el óptimo control de calidad y asegurarle al usuario final que sus requerimientos serán satisfechos.

Existen diversas técnicas para realizar *testing* de *software*; para mencionar algunas de ellas: TDD, TAC y *Testing estructural*, las cuales se explicarán más adelante.

En la etapa de pruebas o *testing*, se diseña un plan el cual define qué pruebas deben ser aplicables al *software*, estas pueden ser de los siguientes tipos:

- ✓ unitarias
- ✓ de función
- ✓ de seguridad
- ✓ de volumen
- ✓ de usabilidad
- ✓ de integridad
- ✓ de estructura
- ✓ de estrés
- ✓ de *benchmark*
- ✓ de concurrencia
- ✓ de carga
- ✓ de configuración
- ✓ de instalación
- ✓ de caja blanca
- ✓ de caja negra
- ✓ de regresión
- ✓ de aceptación

Cada uno de estos tipos de pruebas se estudiará detenidamente en el módulo II.

2.1.2 Testing estructural

Esta forma de *testing* utiliza la información sobre la estructura interna del sistema; básicamente es un *testing* de caja blanca (se ve el código del *software* para realizar la prueba). Se prueba lo que el programa hace y no lo que se supone que debe hacer.

El *testing* estructural tiene las siguientes características:

- ✓ Puede automatizarse en gran medida.
- ✓ Requiere que haya finalizado la fase de codificación para poder empezar las pruebas.
- ✓ Si se cambia la estructura del código, los casos de prueba deben recalcularse.
- ✓ Se definen los valores de entrada para ejercitar flujos particulares.

2.1.3 Test funcional y no funcional

Los casos para pruebas funcionales (test funcional) se derivan de los flujos básicos y alternos de los documentos de requerimientos o casos de uso.

Los flujos básicos se refieren al comportamiento habitual y al normal que debe seguir un sistema, mientras que los flujos alternos se refieren a cómo debe comportarse el sistema en caso de que se dé cierta circunstancia en la ejecución del flujo básico.

Los casos de prueba deben desarrollarse para cada uno de los escenarios existentes en el flujo básico y para cada uno de los flujos alternos.

Este ejemplo del caso de uso (documento de requerimientos) muestra el comportamiento de “retiro de dinero” un cajero automático:

Tipo de flujo	Descripción del caso de uso (especificaciones funcionales)
Flujo básico	Comienza con el cajero en estado listo. 1. Iniciar retiro. El cliente inserta la tarjeta en el lector del cajero automático. 2. Verificar la tarjeta del banco. Se lee el código de la cuenta desde la banda magnética en la parte de atrás de la tarjeta y revisa si es una tarjeta aceptable. 3. Ingresar el PIN. El cajero pregunta al cliente su código de PIN (4 dígitos). 4. Comprobar el código de la cuenta y el PIN. El código de la cuenta y el PIN se verifican para determinar si la cuenta es válida y si el PIN ingresado es el correcto. 5. Opciones del cajero automático. El cajero despliega las diversas alternativas disponibles. En este flujo, el cliente siempre selecciona "retiro de efectivo". 6. Ingresar la cantidad. Se digita el monto por retirar. 7. Autorización. El cajero inicia el proceso de verificación con el sistema bancario al enviar IDTarjeta, PIN, Cantidad, e Información de la cuenta, como una transacción. Para este flujo, el sistema bancario está en línea y replica con la autorización para completar el retiro de dinero exitoso y, como consecuencia, actualiza el balance de la cuenta. 8. Dispensar. El dinero es dispensado. 9. Retornar tarjeta. La tarjeta es devuelta. 10. Entregar recibo. El recibo es impreso y dispensado. Se actualiza la bitácora interna respectivamente. El caso de uso termina con el cajero automático preparado para realizar una nueva transacción.

Flujo alterno 1. La tarjeta no es válida.	En el paso 2 del flujo básico (verificar tarjeta bancaria), si la tarjeta no es válida, es expulsada con un mensaje apropiado.
Flujo alterno 2. El cajero no tiene dinero.	En el paso 5 del flujo básico (opciones del cajero automático), si el cajero no tiene dinero, la opción "retiro de dinero" no estará disponible.
Flujo alterno 3. Fondos insuficientes en el cajero.	En el paso 6 del flujo básico (ingresar la cantidad), si el cajero contiene fondos insuficientes para dispensar el monto solicitado, un mensaje apropiado será desplegado, y se retorna al paso 6 del flujo básico, ingresar la cantidad.
Flujo alterno 4. PIN incorrecto.	En el paso 4 del flujo básico (verificar la cuenta y el PIN), el cliente tiene tres intentos para ingresar el PIN correcto. Si un PIN incorrecto es ingresado, el cajero despliega el mensaje apropiado, y si todavía quedan intentos, se retorna al paso 3 del flujo básico, ingresar el PIN. Si en el intento final, se ingresa un PIN incorrecto, la tarjeta es retenida, el cajero vuelve a quedar preparado para nuevas transacciones, y este caso de uso termina.
Flujo alterno 5. La cuenta no existe.	En el paso 4 del flujo básico (verificar cuenta y PIN), si el sistema bancario retorna un código indicador de que la cuenta no pudo ser encontrada o no permite retiros, el cajero despliega el mensaje apropiado y retorna al paso 9 del flujo básico, devolver tarjeta.
Flujo alterno 6. Fondos Insuficientes en cuenta.	En el paso 7 del flujo básico (autorización), si el sistema bancario retorna un código e indica que el balance de la cuenta es menor que la cantidad ingresada en el paso 6, el ATM despliega el mensaje apropiado y retorna al paso 6 del flujo básico.
Flujo Alterno 7. "Log" de error.	Si en el paso 10 del flujo básico (recibo), el "log" no puede ser actualizado, el cajero ingresa en "modo seguro", en el cual todas las funciones son suspendidas y una alarma apropiada es enviada al sistema bancario para indicar que el cajero ha suspendido la operación.
Flujo Alterno 8. Abandonar ("cancelar").	El cliente puede, en cualquier momento, decidir terminar la transacción ("cancelar"). La transacción se detiene y la tarjeta es expulsada.

El ejemplo anterior es un sencillo caso de uso (un documento de requerimientos de *software*). Por un lado, el test funcional es aquel tipo de prueba que ejercita cada uno de los casos de escenarios identificados, para garantizar el correcto funcionamiento del *software*, el cual está plasmado en un documento de requerimientos. La identificación de los casos de prueba debe ser tanto para el flujo básico como para los flujos alternos.

No todos los requerimientos para un objetivo de pruebas serán reflejados en los documentos mencinoados. Por otro lado, los requerimientos no funcionales, como el rendimiento del *software*, la seguridad y el acceso especifican comportamientos adicionales o características para otro tipo de objetivos de prueba (test no funcional), los cuales ejercitan aquello que no es específicamente la funcionalidad del *software*, pero sí es parte importante para un producto de calidad.

La especificación adicional que se haga en este sentido, es el insumo primordial para derivar las pruebas no funcionales.

Los siguientes son algunos de los tipos de prueba que se deben realizar para los test no funcionales:

- ✓ de rendimiento
- ✓ de seguridad / acceso
- ✓ de configuración
- ✓ de instalación

Pruebas de rendimiento

Estas se derivan de la información suplementaria del documento de requerimientos; además, estos datos determinan criterios de rendimiento tales como el tiempo de transacción o el número de transacciones por usuario. Con base en esa información, se determinan y realizan las pruebas para comprobar lo solicitado; muchas veces este tipo de pruebas se realizan de forma automatizada.

Los escenarios típicos de estas pruebas son:

Carga de trabajo	Condición	Resultado esperado
1000 cajeros simultáneos	Transacción de retiro completa	Transacciones completas menores o iguales a 25 segundos.
2000 cajeros simultáneos	Transacción de retiro completa	Transacciones completas menores o iguales a 35 segundos.
10 000 cajeros simultáneos	Transacción de retiro completa	Transacciones completas menores a 50 segundos.

Pruebas de seguridad o acceso

Validan los derechos que tienen los diferentes roles sobre el sistema, dependiendo de la condición dada.

Los escenarios típicos de estas pruebas son:

Condición	Tarjeta	Lector de tarjeta	Red del banco	Resultado esperado
No puede leer tarjeta	Inválida	Válida	Válida	Muestre mensaje de advertencia y expulse la tarjeta
Tarjeta reportada como robada	Inválida	Válida	Válida	Muestre mensaje de advertencia y retiene la tarjeta
Tarjeta expiró	Inválida	Válida	Válida	Muestre mensaje de advertencia y retiene la tarjeta

Pruebas de configuración

Generalmente, se realizan en escenarios distribuidos, es decir, donde existen muchas combinaciones de *hardware* y *software* en los cuales el sistema se debe ejecutar.

Ejemplo de estas combinaciones son: diferentes sistemas operativos, navegadores, velocidades de procesador, *software* instalado de diferentes tipos, etc.

Para preparar escenarios de este tipo de pruebas, se debe tomar en consideración elementos como los siguientes:

- ✓ Identificar las configuraciones más comunes para que sean probadas previamente, incluyen: impresoras, conexiones de red, configuraciones de servidor.
- ✓ Identificar tipos de *software* y licencias instalados (en cliente y en servidor).
- ✓ Identificar los requerimientos mínimos de *hardware*.
- ✓ Identificar los recursos mínimos (procesador mínimo, capacidad mínima de disco, capacidad mínima de memoria, resolución mínima).

Pruebas de instalación

Estas garantizan que el sistema se instale bajo todos los escenarios posibles, los cuales deben contemplar, al menos, los siguientes objetivos:

- ✓ Instalar correctamente la primera vez.
- ✓ Instalar correctamente una versión más reciente (*upgrade*).
- ✓ Instalar correctamente una versión personalizada.

2.2 TDD y TAC

El *Test Driving Development* (TDD por sus siglas en inglés, “desarrollo guiado por pruebas”, su significado en español) dirige la implementación con las pruebas (probar antes de la codificación final), lo cual agiliza el desarrollo del producto. El *testing* y la codifificación son inseparables elementos del desarrollo de *software*, por lo que se obtiene un mejor resultado con los encargados de pruebas y los desarrolladores trabajando juntos.

El TDD prepara casos de prueba de alta calidad antes de escribir el código; la habilidad de los *tester* para ayudar al negocio experto a clarificar o esclarecer requerimientos o características particulares de funcionalidad, es usada a fin de identificar lo que el código necesita hacer.

De esta forma, los *tester* y los desarrolladores trabajarán juntos para diseñar pruebas, codificar, diseñar más pruebas y codificar más; se efectúan tantas iteraciones como sea necesario, hasta que los requerimientos del negocio se alcancen en su totalidad.

A continuación, se desarrolla un ejemplo acerca del trabajo entre un *tester* y un desarrollador, utilizando el concepto de TDD:

Suponga que el desarrollador y el *tester* están trabajando juntos para implementar del cálculo del costo de envío de un producto, basado en el peso y en el código postal destino (suponga compra por *Internet*).

El desarrollador, como primer paso, diseña su prueba e implementa la solución en su *unit test*:

Peso	Código postal destino	Costo
5 kg	80104	$ 7,50

Para validar la implementación, el desarrollador envía las entradas correspondientes y obtiene el cálculo de forma correcta; luego, muestra esto al *tester*, él lo ve bien pero prueba más casos: realiza pruebas enviando al código postal *default* (por defecto), el sistema se comporta correctamente, pero cuando intenta enviar a códigos postales diferentes, de otros países, aparece una excepción, un error. El *tester*, entonces, muestra el resultado al desarrollador y se percata que no aparece la implementación en su *unit test* para que soporte envíos a diferentes códigos postales. Ahora la implementación del sistema cambia, y sería la siguiente:

Peso	Código postal destino	País	Costo
5 kg	80104	US	$ 7,50
5 kg	T2J 2M7	CA	$ 9,40

Una vez realizadas las pruebas, se refactoriza de código (este concepto se explica más adelante) para que vaya quedando claro, ágil y de fácil mantenimiento, y así ir desarrollado la aplicación.

De esta forma, se puede observar cómo el TDD es ventajoso en el sentido de que las pruebas se realizan desde los primeros pasos en la codificación, y no cuando esté toda la aplicación terminada, además, desde el primer instante valida y verifica los requerimientos del usuario.

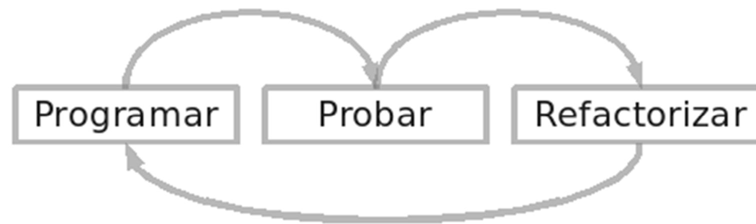


Figura 7. Desarrollo TDD (Fuente: UNED, 2010, *Control de la Calidad del Software - 3094*)

El TAC o TAD (*Test After Coding – Test After Development*, en español "probar después de codificar o desarrollar") se refiere a la técnica de realizar primero la aplicación, y luego, someterla al proceso de pruebas.

Los pasos de esta técnica son:

a) Levantar los requerimientos.

b) Considerar el mejor diseño para la aplicación.

c) Implementar el código de la aplicación siguiendo el diseño.

d) Implementar los *unit test* para el código.

e) Repetir los pasos c) y d) hasta que la aplicación esté completa.

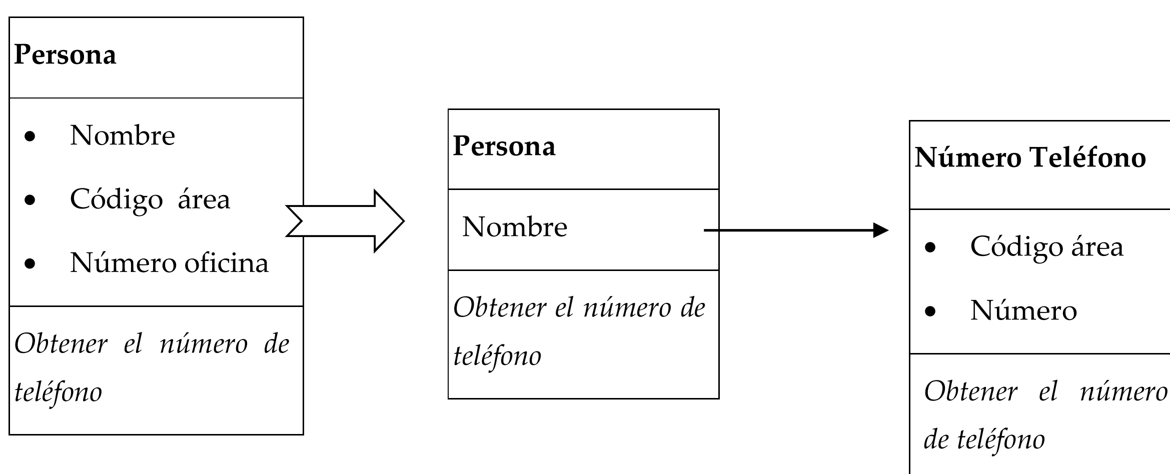
Como se puede observar, primero se codifica la aplicación y luego las pruebas unitarias, esto provoca que cada vez que se haga un cambio en la aplicación se deben reescribir las pruebas unitarias.

2.3 Refactorización del código

Refactoring es una técnica disciplinada para reestructurar un código existente, pues altera su estructura interna sin cambiar su comportamiento externo. La esencia es ir realizando pequeñas transformaciones; cada una de ellas (llamado *refactoring*) es pequeña, pero una secuencia puede producir una significante reestructuración.

Es probable que al realizar una reestructuración pequeña, el sistema no funcione igual, por eso hay que asegurarse de que funcione completa y adecuadamente cada vez que se realiza una pequeña reestructuración de código, para no dar así oportunidad a que el problema se agrave.

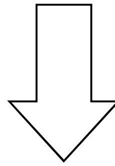
Ejemplo de refactorización de la clase *Persona*:



En el anterior ejemplo, se tiene una clase (*Persona*), quien hace el trabajo que debería hacer dos (*Persona*, *Número Teléfono*). Para *refactorizar* el código, se crea una nueva clase y se mueven los campos relevantes, así como los métodos desde la antigua *clase* a las nuevas.

Ejemplo sencillo de refactorización de código en un *script* de base de datos:

```
select ac.Id,ac.Nombre, bc.primer_apellido,bc.segundo_apellido from Cliente ac, ClienteRelacion bc
where ac.Nombre = 'Omar' and ac.Id in
(select id from ClienteRelacion where primerapellido = 'Vargas' and segundoapellido = 'Porras')
and ac.Id = bc.id
```



```
select
  a.Id
 ,a.Nombre
 ,b.Primer_apellido
 ,b.Segundo_apellido

from
 Cliente a
 ,ClienteRelacion b

where a.Id = b.Id
and a.Nombre        = 'Omar'
and b.Primer_apellido = 'Vargas'
and b.Segundo_apellido = 'Porras'
```

En el anterior código, se observa cómo el *refactoring* ordena de manera clara, las sentencias; de esta forma, reestructura el código interno, y lo muestra más claro, sin modificar el comportamiento externo.

Ejercicios de autoevaluación

Tema 1. Herramientas de *testing*

1. En un proceso de *testing*, ¿cuáles son los elementos básicos de entrada y qué se obtiene como salida?
2. Para el adecuado control de la calidad, IBM Rational ofrece múltiples herramientas, entre ellas, Rational ClearCase, Rational ClearQuest y Rational Test Manager. Eplique la función de cada una.
3. Mencione dos herramientas *openSource* útiles para realizar pruebas de aceptación.

Tema 2. Estrategias de *testing*

1. ¿Qué es TDD y cuál es su principal ventaja?
2. ¿Qué es test funcional?
3. ¿Qué es test no funcional?

Caso de análisis. Desarrollo de un proceso de *testing*

Suponga que usted es uno de los *tester* del área de control de calidad de su organización, el área de fabricación de *software* concluye un desarrollo totalmente nuevo, el cual contiene:

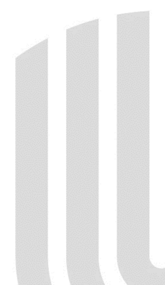
Documentación de la aplicación:

a) requerimientos funcionales documentados
b) requerimientos no funcionales documentados
c) diagramas de clase documentados
d) diagrama de entidad-relación documentado
e) diagrama de infraestructura documentado

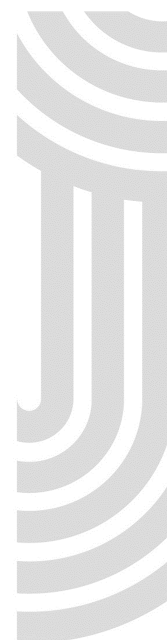
Carácterísticas de la aplicación:

a) Ayuda en línea desarrollada
b) Perfiles de usuario
c) La aplicación es web
d) Los usuarios pueden utilizar la aplicación desde cualquier lugar del mundo.

Suponga que a usted le corresponde realizar el control de calidad a este nuevo desarrollo; con base en lo estudiando en el módulo I, prepare un procedimiento en el cual presente las entradas, los procesos y las salidas del control de calidad, así como su adecuada documentación y posibles herramientas y estrategias por utilizar.



ACTIVIDADES DE LA CALIDAD TOTAL II



Sumario

- Tipos de pruebas
- Validación y verificación de *software*

Objetivos

Al finalizar el estudio de este módulo, entre otras habilidades, usted será capaz de:

- Ejecutar los diferentes tipos de prueba que existen para evaluar un *software*.
- Conocer los conceptos de validación y verificación de *software*.
- Aplicar las mejores prácticas recomendadas en la validación y verificación del *software*.

Introducción

Este módulo explica qué es un ciclo de pruebas, cuáles son sus etapas, y además los tipos de test más importantes que se pueden realizar a un *software*, con el fin de que se comprenda su finalidad, su beneficio, importancia y aplicación.

Se detallan, también, los conceptos de validación y verificación de *software* y su aplicación, la importancia de la participación temprana del *tester* en los proyectos y los insumos de pruebas.

Tema 3. Tipos de pruebas

En este tema se desarrollan los tipos de pruebas más comunes y destacables para validar la calidad del *software*. Usted aprenderá su concepto, entenderá para qué se realiza cada prueba y sabrá en qué casos debe aplicarlas.

Además, aprenderá sobre las fases de test, en específico, la metodología que nos ofrece RUP; de esta forma, usted podrá ver cómo las diferentes metodologías que nos brinda el mercado mejoran en gran medida, la calidad del desarrollo de *software* en una organización, y le motivará en la investigación de nuevas y diferentes opciones metodológicas.

3.1 El ciclo de vida de pruebas según RUP

En RUP, el ciclo de vida de desarrollo se va depurando a través de iteraciones (Ej.: versiones de *software* operacionales, entregables completos). Para el caso específico de *software* ejecutable, en cada iteración el equipo de desarrollo producirá una o más versiones operativas que deben ser probadas.

Las pruebas implementadas y ejecutadas en cada versión se depuran, agregan o eliminan. Algunas de estas se retendrán y acumularán como una unidad de pruebas, las cuales se usan para versiones operacionales siguientes de pruebas de regresión en cada ciclo futuro de pruebas. Esta estrategia las revisa durante todo el proceso.

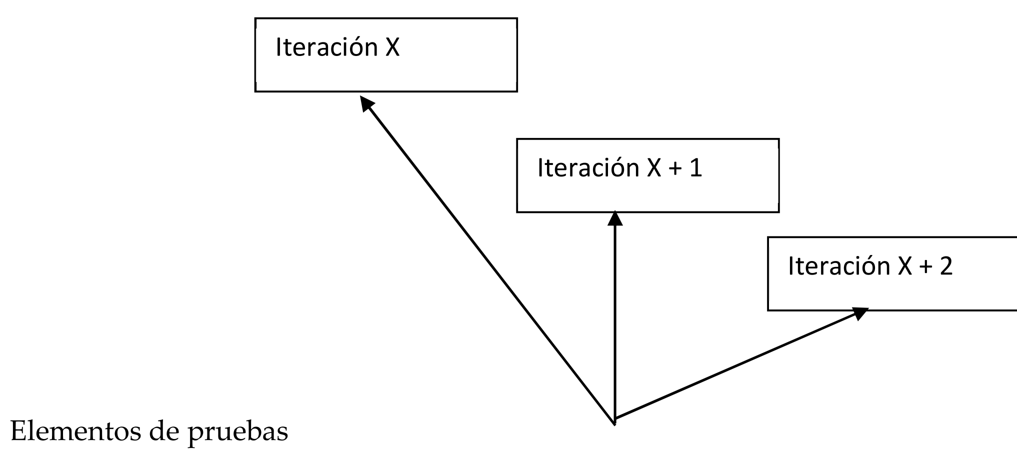


Figura 8. Evolución de las pruebas a través del tiempo

La estrategia iterativa requiere la aplicación de pruebas de regresión (más adelante se explicarán). Cualquiera de las pruebas desarrolladas en la iteración X es un candidato potencial en la iteración X + 1, asimismo, en la iteración X + 2 y así sucesivamente. Si la misma prueba es repetitiva, cabe la posibilidad de evaluar la oportunidad de automatizarla (pruebas automatizadas se explicará en el módulo III); de esta manera, liberará al personal de *testing* para invertir esfuerzos en la implementación de más validaciones.

El ciclo de vida de pruebas para cada iteración es el siguiente:

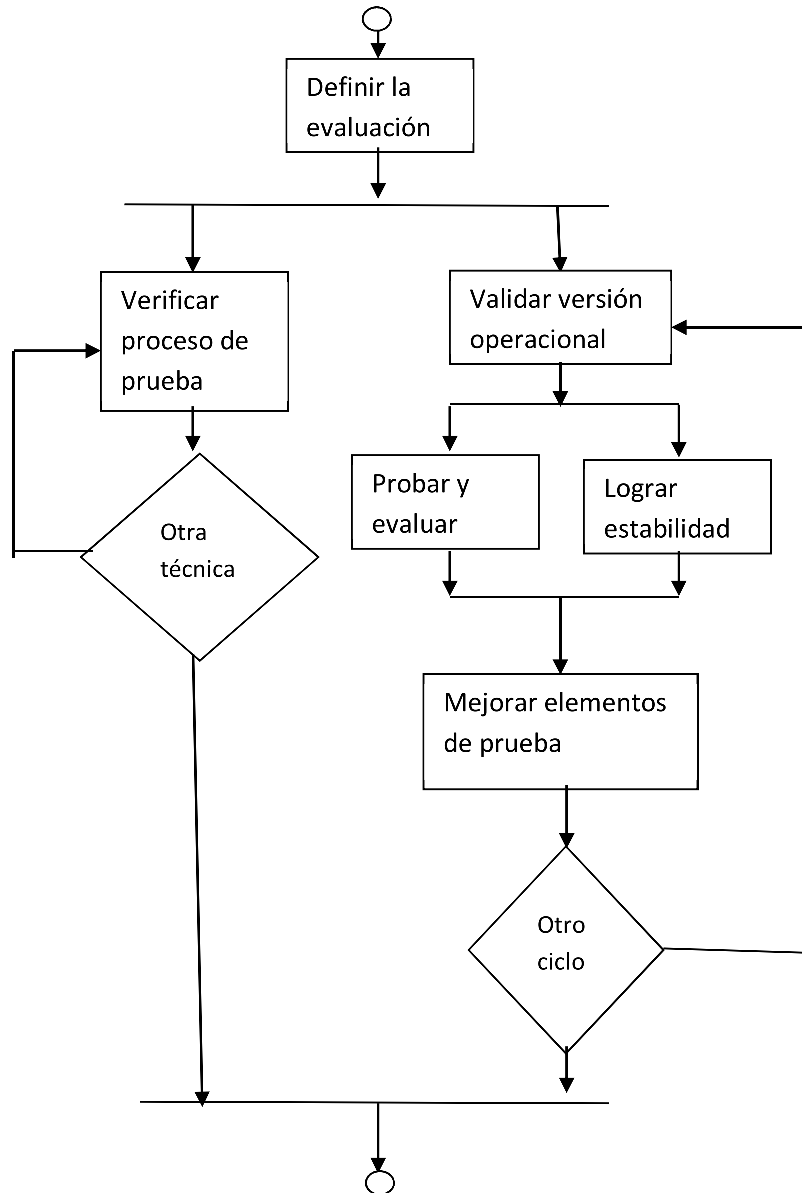


Figura 9. Ciclo de vida de pruebas

La iteración comienza por una investigación del equipo de pruebas y la negociación con los administradores del proyecto; cada una contendrá al menos un ciclo de pruebas. Comúnmente, se desarrollan múltiples versiones operacionales por iteración y se los asigna un ciclo de pruebas.

Al mismo tiempo que las pruebas están en ejecución, un subconjunto de miembros del equipo puede estar investigando nuevas técnicas.

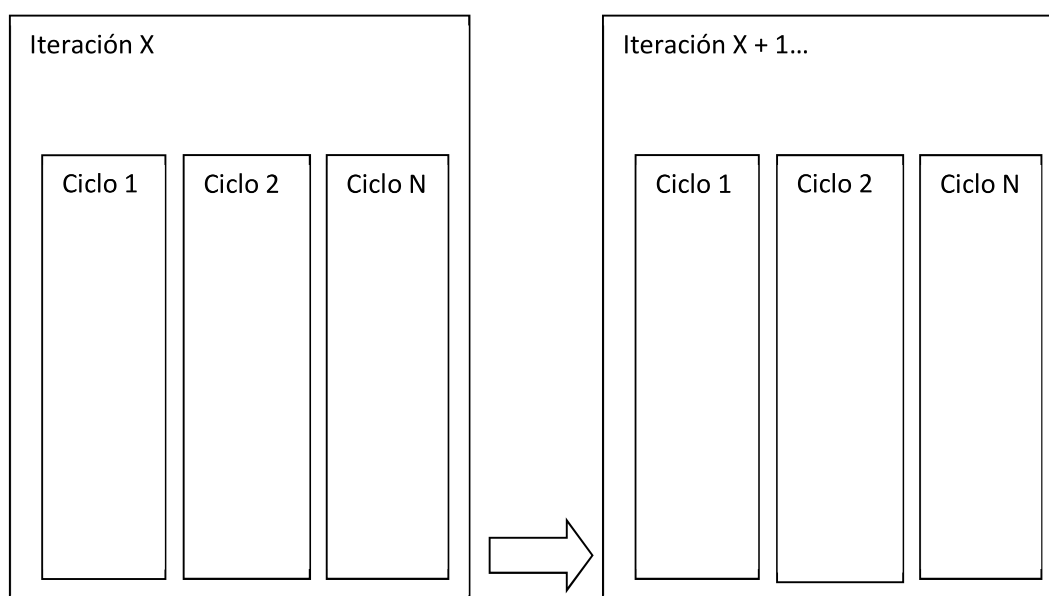


Figura 10. Ciclos de prueba

Un ciclo de pruebas se refiere a ejecutar completamente casos de prueba que se encargarán de ejercitar los componentes de la versión operacional del *software*.

Los problemas encontrados durante una iteración pueden resolverse en la misma iteración o posponerse para la siguiente (el administrador de proyectos decide). Una de las tareas principales para el equipo de pruebas es determinar qué tan completa es la iteración al verificar que se cumplieron los objetivos. De iteración a iteración aparecerán nuevos requerimientos, algo de lo que el *tester* debe tener conciencia y estar preparado para manejarlo.

3.2 Etapas de pruebas según RUP

La realización de pruebas se deben aplicar en diferentes etapas, y estas deben ser hechas por roles experimentados en su diseño y su conducción, y aplicando la técnica adecuada en cada nivel.

Pruebas del desarrollador

Tradicionalmente, las pruebas de desarrollador han sido pensadas, principalmente, en términos de pruebas unitarias con diversos niveles de enfoque, pero puede que no abarque todos los enfoques de una prueba integral. Utilizando este método, se presentan riesgos en la calidad del *software* en esta etapa de test, que a menudo son descubiertos en los otros niveles.

Pruebas independientes y de los implicados

Las pruebas independientes significan el diseño y la ejecución bien elaborados de las pruebas, para que sean aplicadas por personas independientes al equipo de desarrolladores. Además, de las pruebas que el personal independiente debe ejecutar, también ellos pueden agregar sus propias pruebas y hacerlas disponibles.

Una alternativa que brinda las pruebas independientes es que posibilita vincular a los implicados para tomar en cuenta sus puntos de vista, o sea, a conjuntos de personas tales como: soporte técnico, capacitadores, personal de ventas, clientes, etc.

Pruebas unitarias

La pruebas unitarias se enfocan en verificar los elementos “más pequeños” del *software*. La prueba unitaria se aplica, típicamente, a componentes en el modelo de implementación para comprobar que los flujos de control y los de datos estén cubiertos y funcionen como se espera.

Las expectativas de estos test se basan en la forma en que los componentes participan en la ejecución de un caso de uso, lo cual se encuentra en los diagramas de secuencia para ese caso de uso.

Las pruebas unitarias ejercitan, principalmente, los métodos del *software* y sus detalles que están descritos en el proceso de implementación.

Pruebas de integración

La prueba de integración asegura que los componentes en el modelo de implementación operen apropiadamente cuando se mezclen para ejecutar un caso de uso.

Generalmente, los paquetes combinados provienen de diferentes equipos de desarrollo. Las pruebas de integración descubren defectos o errores que se originan al combinar los componentes de un *paquete* (versión operacional funcional a liberar).

Pruebas del sistema

Las pruebas de sistema son hechas, principalmente, cuando el *software* funciona como un todo. Esta etapa de test ocurre tan pronto como subconjuntos bien formados del comportamiento de los casos de uso estén implementados.

El objetivo de esta etapa es probar de extremo a extremo los elementos internos del sistema.

Pruebas de aceptación

Esta es típicamente la final, anterior a la puesta en marcha (liberación en *producción*). El objetivo de estas pruebas es verificar que el *software* está listo y que los usuarios finales puedan usadolas para ejecutar aquellas funciones y tareas para las que fue construido, en las cuales, generalmente, se certifica mediante escenarios de pruebas creados por los mismos usuarios finales.

3.3 Cobertura de pruebas

Las coberturas de pruebas se refieren al porcentaje de código, funcionalidad o rendimiento de un sistema cubierto por los test.

Para establecerlas, existen algunos criterios tales como ejecución de casos de pruebas efectuadas y cobertura de código total del sistema. Por ejemplo, un criterio para permitir el avance del *software* hacia las pruebas de aceptación puede ser cuando el 95% de los casos de prueba hayan sido ejecutados con éxito. Otro criterio podría ser que el 100% del código esté cubierto por las pruebas unitarias.

Existen herramientas que ayudan a determinar la cobertura de pruebas en el código, por ejemplo, la solución de Microsoft Visual Studio Ultimate 2010, que ofrece la posibilidad al *tester* de realizar las pruebas unitarias por clase; luego de la ejecución de los test, la herramienta muestra un reporte de la cobertura de código cubierto y el restante.

3.4 Ideas de pruebas

Las ideas de pruebas son listas o catálogos con escenarios útiles de test que ayudan a mejorar la calidad del *software*, y que están presentes para que se utilicen cada vez que sea necesario probar uno en particular, o de forma integral.

Para que una lista o catálogo de ideas de prueba sea bueno, se debe tomar en cuenta los siguientes criterios:

- ✓ Las ideas de pruebas planteadas pueden encontrar un conjunto grande de fallas subyacentes.
- ✓ Las ideas de pruebas son fáciles de aplicar y se puede identificar las no relevantes a una situación particular.
- ✓ No debe tener ideas de pruebas que nunca se utilizan.

Las ideas de prueba se ejecutan para test de caja blanca y para test de caja negra, luego, pueden ser parte de los casos de prueba.

La lista o catálogo de ideas de prueba lo realizan, generalmente, *tester* expertos, quienes han experimentado las posibles fallas de sistemas, entonces documentan lo que se debe tomar en cuenta para probar y asegurar la calidad del *software*.

3.5 Tipos de pruebas específicas

Para obtener calidad en el *software*, es necesario evaluar más allá que simplemente el comportamiento correcto de la interfaz (cara que le muestra el *software* al cliente, donde este trabaja), de las funciones y el tiempo de respuesta de un sistema. La dimensión de la calidad de un *software* debe enfocarse en las características y los atributos, tales como integridad, facilidad de instalación, ejecución en diferentes plataformas, y buen rendimiento en el manejo de solicitudes simultáneas.

La elavuación de dichas características se logra con diversos tipos de pruebas, cada uno de ellos tiene un objetivo específico y una técnica que evalúa un criterio de calidad del *software*, los cuales se pueden definir en los grupos de:

- ✓ funcionalidad
- ✓ usabilidad
- ✓ confiabilidad
- ✓ desempeño
- ✓ soporte

A continuación, se muestran los tipos de pruebas que validan esos criterios de calidad en el sistema.

Funcionalidad: pruebas de función

Las pruebas de función validan y verifican las funciones del *software* con base en los requerimientos solicitados (los casos de uso por ejemplo). En estas pruebas, se incluyen las unidades de sistemas y su integración (se llaman pruebas de caja negra, ya que el *tester* se concentra en validar la funcionalidad del sistema y los resultados arrojados con base en escenarios, sin importar lo que haga el código del *software* internamente para que arroje dichos resultados). Se confía en las condiciones de entrada y salida para la evaluación de las pruebas.

Funcionalidad: pruebas de seguridad

Estas aseguran que los datos sean accesibles solo para aquellos actores deseados; en otras palabras, es la prueba que garantiza los perfiles de usuario y verifica que tienen acceso solamente a la información necesaria.

Funcionalidad: pruebas de volumen

Las pruebas de volumen validan la capacidad del sistema para manejar grandes volúmenes de datos, ya sea de entrada, salida, o bien residentes en su base de datos. Además, incluyen estrategias como generar un reporte que devuelva la información de la base de datos, o que no retorne dato alguno por la definición de los filtros. También, valida la cantidad máxima de caracteres aceptados en un campo.

Usabilidad: pruebas de usabilidad

Estas pruebas se enfocan en factores humanos, tales como estética, interfaz intuitiva (entendible y fácil de usar), ayudas en línea, contexto entendible, asistentes, documentación de usuario y materiales de apoyo.

Confiabilidad: pruebas de integridad

Estas se enfocan en valorar la robustez del sistema (resistencia a fallos). Por ejemplo, validan el comportamiento adecuado de la unión de componentes individuales de *software* que forman el sistema.

Confiabilidad: pruebas de estructura

Las pruebas de escritura validan la estructura interna del código fuente, buscan las debilidades y errores en él. Este tipo de prueba es realizado, la mayoría de las veces, por los desarrolladores.

Las pruebas de estructura web validan que los enlaces del sitio estén conectados a la página correspondiente, para no mostrar contenido que no pueda ser consultado (contenido huérfano).

Confiabilidad: pruebas de estrés

Esta evaluación valida la respuesta del sistema bajo condiciones anormales; el estrés en el sistema puede incluir cargas de trabajo externas, memoria insuficiente, *hardware* y servicios no disponibles o limitados recursos compartidos.

Las pruebas de estrés son de gran utilidad para entender en qué condiciones el sistema puede fallar; de esta manera, se pueden realizar planes de contingencia para solventar el problema cuando ocurra, además de identificar las mejoras que se le deben realizar.

Desempeño: pruebas de benchmark

Tipo de prueba que compara el desempeño de un sistema al aplicar un test nuevo con uno ya conocido; evalúa el rendimiento de un sistema determinado bajo diferentes configuraciones de *hardware* y *software*.

Desempeño: pruebas de concurrencia

En este caso, se valida el correcto funcionamiento del sistema bajo la demanda de múltiples usuarios o accesos sobre el mismo recurso (memoria, base de datos, *webservices*, etc.)

Desempeño: pruebas de carga

Las pruebas de carga se usan para validar y estimar la aceptación de los límites operacionales de un sistema bajo cargas de trabajo variables.

Los escenarios típicos son: la carga del trabajo permanece constante y la configuración del sistema bajo prueba varía (por ejemplo: menos disco, menos memoria, etc.), o la configuración del sistema permanece constante y la carga del trabajo varía.

Las métricas se basan en el rendimiento de la carga de trabajo y en el tiempo de respuesta de las transacciones en línea. Las variaciones de dicha carga incluirán la emulación de cargas promedio que ocurrirán en condiciones normales de operación.

Desempeño: pruebas de perfil de desempeño

Tipo de prueba que incluye el flujo de ejecución normal, acceso a datos, funciones y llamadas del sistema para identificar posibles "cuellos de botella" y procesos ineficientes.

Soporte: pruebas de configuración

Estas pruebas validan y aseguran que el sistema haga lo deseado en diferentes configuraciones de *hardware* y *software*.

Soporte: pruebas de instalación

Estas pruebas validan que el sistema se instale según lo deseado en diferentes configuraciones de *hardware* y *software*, y realiza las validaciones necesarias bajo diferentes límitantes como espacio en disco; también, valida el correcto funcionamiento ante fallos como interrupción de electricidad.

Además de los anteriores tipos de pruebas para garantizar la calidad, se identifican también los siguientes, que aseguran aún más la estabilidad del sistema:

Pruebas de caja blanca

Estas se basan propiamente en la revisión del código, en los modelos de diseño, inclusive, en detectar funcionamiento indebido del *software*; por ejemplo, suponga que se está evaluando un sistema de pagos electrónicos, funcionalmente el *software* parece que hace lo que debe, pero internamente, tiene una instrucción que acredita un pequeño porcentaje a la cuenta de un funcionario cada vez que se hagan diez transacciones. Este tipo de situaciones, muy probablemente, es más sencillo detectarlas con la revisión de código o pruebas de caja blanca que, de paso, es funcionalidad que el sistema no debería hacer (suponga que así lo dictan los casos de uso); por lo tanto, existiría un defecto.

Pruebas de regresión

Las pruebas de regresión validan que, luego de un cambio, la funcionalidad ya probada en el *software* no se afectó. Por ejemplo, suponga que se tiene un sistema el cual debe realizar lo siguiente: A–B–C–D = Z, se realiza una solicitud de cambio y se desea que ahora al *software* se le agregue un elemento más: E, el *software* entonces debería comportarse así: A–B–C–D–E = Z – E. La prueba de regresión, en este caso, es la unidad de prueba la cual asegura que el primer comportamiento (A – B – C – D = Z) no se afectó y continúa funcionando de igual forma aun con el cambio solicitado.

Pruebas de aceptación

Una prueba de aceptación de “usuario” es típicamente la prueba final anterior a la puesta en marcha en producción del *software*. Además, su intensión es verificar que el *software* está listo y puede llevarlo a los usuarios finales para ejecutar aquellas funciones y tareas para las cuales la aplicación fue creada.

Respecto de la realización de estas pruebas, lo conveniente es que el usuario experto (usuario conocedor del negocio) prepare un plan de pruebas en el cual se detallen las entradas, pasos por seguir, proceso por realizar y resultado esperado. Este plan de pruebas lo ejecutará el usuario experto, y se recomienda que sea sin ayuda técnica para que el usuario identifique errores de todo tipo, inclusive los de usabilidad.

También, existe otro concepto de este tipo de pruebas, que se caracteriza un grupo o equipo no interviene en otro. Por ejemplo, *una prueba de aceptación de una versión operacional* es hecha para aceptar la entrega de una versión de *software* desde el ambiente de desarrollo (donde trabajan los programadores) hacia el ambiente de pruebas (donde se realizan los test).

Tema 4. Validación y verificación de *software*

En este tema se explican los conceptos de validación y verificación en el *software*, así como los insumos necesarios para realizarlas y planearlas; además, se detallan estrategias que se emplean para aumentar la calidad en las aplicaciones.

Con los términos detallados acá, usted tendrá una mejor noción de su importancia, y le instan a continuar investigando herramientas, metodologías y buenas prácticas que le ayuden a realizar, con mayor eficiencia, la labor de evaluación de la calidad.

4.1 Validación

La validación (VAL) se lleva a cabo con el fin de demostrar que un *software* cumple satisfactoriamente su objetivo al ser colocado en su ambiente de trabajo.

4.2 Verificación

La verificación (VER) pretende asegurar que el *software* cumple con los requisitos especificados.

4.3 Participación temprana en el proyecto

En el desarrollo de *software* está presente la etapa de pruebas y, consecuentemente, el rol de los *tester* en el proyecto. Si se analiza detenidamente, es sumamente complejo.

La etapa de pruebas en el *software* tiene como finalidad encontrar los posibles problemas (de todo tipo) antes de que el producto se ofrezca en producción, con el fin de garantizar su calidad.

Típicamente, el ciclo de vida de un desarrollo de *software* se conforma de las siguientes etapas: requerimientos, análisis y diseño, desarrollo, pruebas y puesta en marcha. Conforme aparezcan defectos en las etapas más avanzadas del desarrollo, el costo es mayor. Vea la siguiente tabla:

Etapa en la cual se encuentra un defecto	**Costo relativo de la corrección de un defecto**
Requerimientos	Se multiplica por 1
Análisis y diseño	Se multiplica por 3 o por 6
Desarrollo	Se multiplica por 10
Pruebas	Se multiplica por 50
Puesta en marcha (liberación en producción)	Se multiplica por cientos

Entonces, si se sabe que el *testing* es la labor para encontrar defectos y garantizar la calidad y, además, que por cada error encontrado en etapas avanzadas del desarrollo, el costo relativo de corrección aumenta.

Es aquí donde se recomienda que el *tester* deba tener una temprana participación en el proyecto. Las buenas prácticas de desarrollo de *software* y pruebas establecen como método para garantizar la calidad, reducir costos y reducir esfuerzos, que el *tester* esté presente en las etapas iniciales del ciclo de vida del *software.* Metodologías ágiles como XP (*eXtreme Programming*) lo recomienda al equipo de pruebas desde la etapa de requerimientos (inclusive desde la concepción de la idea), para que se detecten anticipadamente los problemas relativos con la definición de un requerimiento, y no esperar a que se evidencien hasta en las etapas finales, en donde su corrección es sumamente cara.

Veamos un ejemplo sencillo del costo de encontrar un problema en la etapa de pruebas, y no en la etapa de requerimientos:

Etapa de requerimientos

Suponga que el analista se reúne con el usuario y se empieza a definir los requerimientos del *software*, el usuario indica que desea un menú en el cual se ofrezca la funcionalidad del sistema. El analista documenta el requerimiento.

Etapa de análisis y diseño

El analista realiza todos los diagramas UML y los casos de uso, con base en un actor que accede a la funcionalidad del sistema.

Etapa de desarrollo

El implementador desarrolla el *software* con base en lo que dice el caso de uso y diagramas UML.

Etapa de pruebas

El *tester* realiza pruebas de seguridad y detecta la falta de roles, hace las recomendaciones del caso al desarrollador, quien consulta al analista y se da cuenta de que tiene razón.

El analista comenta con el usuario y este último se percata de que sí son necesarios otros roles. El *tester*, entonces, solicita el cambio, lo cual implica:

- ✓ Realizar el cambio en el requerimiento.
- ✓ Modificar los diagramas UML realizados.
- ✓ Rediseñar el caso de uso y volver a solicitar aprobación.
- ✓ Actualizar los casos de prueba.
- ✓ Realizar nuevamente la implementación y ejecutar una vez más pruebas unitarias.

- ✓ Efectuar pruebas de regresión para garantizar que la corrección al defecto no afecta ninguna otra funcionalidad del *software*.
- ✓ Estimar nuevamente los tiempos de entrega.
- ✓ Utilización de recursos y personal.

En fin, una larga lista y mayor costo. Si el error se encuentra en etapas tempranas, esa lista disminuye.

4.4 Insumos de entrada para la validación y verificación de *software*

Como insumo de entrada para la validación y verificación de *software*, muchas veces, se piensa solamente en los requisitos funcionales, que vienen plasmados en el documento de requerimientos o bien en los casos de uso. Esto es parte de los insumos de entrada, pero no es todo.

Como se explicó en el módulo I, los insumos de entrada para el desarrollo de pruebas son:

- ✓ datos de prueba
- ✓ caso de prueba
- ✓ plan de pruebas
- ✓ modelo de diseño
- ✓ guías para pruebas
- ✓ configuración de ambiente de pruebas
- ✓ *script* de pruebas.
- ✓ componente de pruebas

Con base en esos insumos, el *tester* debe realizar el control de calidad necesario en el *software*. Aun así, en el proceso de mejora continua normal de una empresa que revisa constantemente sus procesos, se identifica que para una mejor validación y verificación sea necesario un conjunto de artefactos más, por ejemplo:

- ✓ especificaciones suplementarias o requerimientos no funcionales
- ✓ diagrama de clases de UML
- ✓ modelo de base de datos
- ✓ diagrama de arquitectura del sistema

Con esos insumos extra, el *tester* conocerá y comprenderá más el panorama del sistema, así podrá ver con más claridad las necesidades de pruebas para validar su robustez. Con las "especificaciones suplementarias", se podrán generar pruebas de rendimiento y se podrá validar la capacidad del *software*.

A partir de los "diagramas de clase", el *tester* conocerá la arquitectura interna del código y, aparte de ayudarle para la estimación de pruebas, verificará y medirá el impacto en cada cambio.

El modelo de base de datos le ayudará a conocer la estructura de la base de datos y podrá implementar escenarios de prueba para validar su robustez y su normalización.

Los diagramas de arquitectura del sistema es un insumo necesario, pues le indica al *tester* cuál es la plataforma tecnológica en la que está el sistema, y esto le ayuda a implementar escenarios de seguridad, de carga, concurrencia, etc.

4.5 Planificación de la validacióin y verificación

La planificación es una etapa básica para iniciar el control de calidad, con base en los insumos adquiridos y con base en la solicitud de cambio propuesta, se evalúan los cambios y se analiza su impacto en el sistema de forma integral.

Para realizar tal labor, lo que se genera es un *plan de pruebas*; en él, se especifican los tipos de pruebas que se realizarán, se detallan los elementos por probar, se da la estimación de tiempo, se planifica la configuración del ambiente de pruebas, y se especifican las entregas, entre otros.

Un ejemplo completo de un plan de pruebas por considerar lo ofrece el estándar de la IEEE 829, el cual recomienda los siguientes apartados:

Plan de pruebas

1. Identificador único del documento: se refiere a, por ejemplo, un consecutivo.
2. Introducción y resumen de elementos y características por probar: se describe el alcance, la aproximación, los recursos, la planificación y las actividades necesarias, así como se identifican los elementos de prueba, las tareas y lo que se hará.
3. Elementos
    a. *Software* que se probará: se identifican los módulos de entrada, los procesos y las salidas.
    b. Documentos por probar: se identifican los documentos del ciclo de vida que se desean probar; por ejemplo, el documento de diseño, el de requerimiento, etc.
4. Características por probar: se describe las características del *software* que se examinará, por ejemplo, interfaz de usuario, modularidad, tiempo de respuesta, funcionalidad.
5. Características que no se probarán: se describe lo que no se hará en las pruebas planificadas; por ejemplo, condición de errores no detectadas, pruebas de carga o estrés.
6. Enfoque general de la prueba: se describen las actividades, técnicas y herramientas para realizar pruebas de diseño, unitarias, de requerimientos, de integración, de interfaz.
7. Informes por entregar: se detallan los que se deben entregar, con base a la labor que se realiza en el apartado de “*software* que se probará”. Los documentos son, por ejemplo, el informe de pruebas unitarias, de integración, de interfaz gráfica, de rendimiento.

8. Actividades de preparación y ejecución de pruebas: va el cronograma de actividades del equipo de trabajo de pruebas.
9. Necesidades de entorno: se refiere a los requerimientos de *software* y *hardware* para realizar la ejecución del plan.
10. Necesidades de personal y de formación: son las recomendaciones al tipo de personal que se necesita, se describen las habilidades y el conocimiento que deben tener.
11. Esquema de tiempos: se fija la estimación de las pruebas y se incluye la metodología utilizada por la organización para dicho fin.
12. Aprobaciones: se detalla los empleados o funcionarios aprobadores del plan.

4.6 Técnica de verificación: análisis estático y automatizado de código

El análisis estático de código se refiere al proceso de evaluar el *software* sin ejecutarlo. Es una técnica que se aplica directamente sobre el código fuente tal cual, sin transformaciones previas ni cambios de ningún tipo. Su mayor objetivo es mejorar la base del código manteniendo la semántica original. Existen dos tipos de análisis estático de código: el automatizado y el manual.

El **análisis estático automatizado de código** se realiza por parte de una herramienta que recibe el código fuente de nuestro programa, lo procesa y da sugerencias para mejorarlo. Estas herramientas incluyen analizador léxico y sintáctico y un conjunto de reglas que se aplican sobre determinadas estructuras.

Si el código fuente posee una estructura concreta que el analizador considere como mejorable con base en sus reglas, nos lo indicará y sugerirá la mejora.

Estos analizadores consisten en encontrar partes del código que puedan:

- ✓ Reducir el rendimiento.
- ✓ Provocar errores en el *software*.
- ✓ Complicar el flujo de datos.
- ✓ Suponer un problema en la seguridad.

El **análisis estático manual de código** se basa en el factor humano, en revisar apartados propios de nuestra aplicación; por ejemplo, en determinar si las librerías (componentes de referencia que utiliza nuestro programa, como archivos *.dll*) están siendo utilizadas correctamente por nuestro programa.

4.7 Inspecciones de *software*

La inspección de *software* se refiere al proceso de revisar la documentación generada, revisar los requerimientos, el diseño generado de arquitectura de la aplicación y del modelo de base de datos; y se refiere también a la revisión del código fuente en desarrollo, en fin, es una actividad que se aplica a lo largo del proceso de ingeniería de *software*.

Según la SQA (*Software Quality Assurance*), esto reúne:

- ✓ Un enfoque de gestión de calidad.
- ✓ Tecnología de Ingeniería de *software* efectiva (métodos y herramientas).
- ✓ Revisiones técnicas formales que se aplican durante el proceso del *software.*
- ✓ Una estrategia de prueba.
- ✓ Un control de la documentación del *software* y de los cambios realizados.
- ✓ Un procedimiento que asegure un ajuste a los estándares de desarrollo de *software.*
- ✓ Mecanismos de medición y de generación de informes.

La inspección de *software* se refiere, entonces, a SQA, revisiones y pruebas que se realizan a lo largo del desarrollo para asegurar que cada producto cumple con los requerimientos que le fueron asignados.

El aseguramiento de la calidad consiste en auditar los productos generados, por lo que se va adquiriendo cada vez una garantía más profunda y segura de que su calidad está cumpliendo sus objetivos.

Una revisión, inspección o aseguramiento consiste en:

- ✓ Señalar la necesidad de mejoras en el producto de una sola persona o de un equipo de trabajo.
- ✓ Confirmar las partes del producto en las que no es necesaria una mejora.
- ✓ Conseguir un trabajo de mayor calidad que maximize los criterios de corrección y completitud, principalmente.

Ejercicios de autoevaluación

Tema 3. Tipos de pruebas

1. Con base en la lectura del tema, explique el proceso del ciclo de vida de pruebas, según RUP, que se muestra en la figura 9.

Tema 4. Validación y verificación de *software*

1. Analice: suponga que usted es un *tester* y está ejecutando un caso de prueba; en el proceso de la ejecución, en uno de los pasos del *test case*, se da cuenta de que el *software* falla y se cierra. ¿Esto que usted encontró se refiere al concepto de validación o verificación?

2. Analice: suponga que usted es un *tester* y está ejecutando un caso de prueba; en el proceso de la ejecución, en uno de los pasos del *test case*, se da cuenta de que el *software* realiza la operación A + B = C, pero el caso de pruebas señala que A + B = D. ¿Esto que usted encontró se refiere al concepto de validación o verificación?

Caso de análisis. Identificación de los tipos de prueba necesarios

Suponga que usted es uno de los *tester* del área de control de calidad de su organización; el área de fabricación de *software* concluye un desarrollo totalmente nuevo, pero además, realiza una mejora a un módulo ya existente.

Características de la aplicación:

a) Ayuda en línea desarrollada.

b) Perfiles de usuario.

c) La aplicación es web.

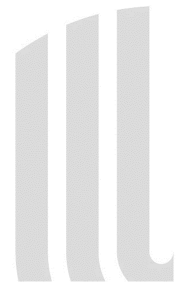
d) Los usuarios pueden utilizar la aplicación desde cualquier lugar del mundo.

e) Tiene una interfaz con otro módulo.

f) Es necesario cumplir con requerimientos funcionales y no funcionales.

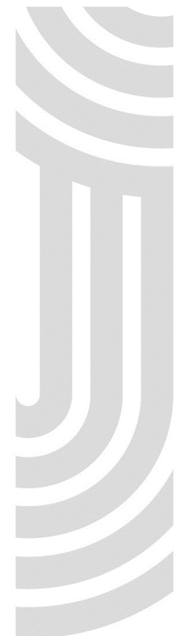
g) La base de datos de la aplicación tiene mucha capacidad de almacenamiento y esto ya fue garantizado en estudios anteriores.

Identifique los tipos de prueba necesarios para que el nuevo desarrollo y la mejora realizada se liberen en producción de la mejor calidad posible.



FUNDAMENTOS DE LAS PRUEBAS DE *SOFTWARE*

III



Sumario

- Testing de las aplicaciones web
- Pruebas automatizadas

Objetivos

Al finalizar el estudio de este módulo, entre otras habilidades, usted será capaz de:

- Definir los tipos de pruebas de seguridad técnica que se pueden aplicar al *software web*.
- Conocer sobre herramientas para el desarrollo de pruebas automatizadas.
- Investigar sobre nuevas formas de automatizar *test cases*.

Introducción

Este módulo está dirigido al *testing* de aplicaciones *web* (aplicaciones a través de un *browser* o llamado también navegador de Internet).

La intensión es ofrecerle la explicación básica para el desarrollo de pruebas en aplicaciones web y que, con base en este, usted pueda crearse un concepto de lo que trata y le motive a la investigación de herramientas, metodologías, estrategias y modelos existentes en ese sentido, tanto para la automatización de pruebas como para la ejecución manual de ellas.

Tema 5. Testing de aplicaciones web

Este tema desarrolla algunos de los tipos de pruebas esenciales para las aplicaciones web, tipos de test mínimos que se deben tomar en cuenta para asegurar la calidad de una aplicación de estas.

Usted entenderá los conceptos de dichas pruebas y notará su importancia, entonces, verá la utilidad de emplearlas en las aplicaciones de su organización.

No se explica cómo las debe hacer, pero sí se indica qué se debe hacer.

5.1 Test de funcionalidad

Se refiere a la aplicación de pruebas para validar la funcionalidad de una página web. Entre las pruebas que se recomiendan están:

Pruebas de enlaces (links)

Consisten en realizar test:

- ✓ A los enlaces de salida de todas las páginas desde un dominio (conjunto de equipos informáticos administrados en una misma red).
- ✓ A todos los enlaces internos, esto es, asegurarse de que todos *links* internos que llaman a otras páginas no fallen.
- ✓ A enlaces que envían información, por ejemplo, correos electrónicos.
- ✓ Para asegurarse de que no hay *links* a páginas "huérfanas", "muertas" o inexistentes o enlaces "rotos".

Para la realización de estos test, es de gran utilidad para los ingenieros de pruebas las herramientas automatizadas que existen en el mercado, las cuales se explicaron en el módulo I.

Pruebas de formularios

Los formularios son pruebas integrales de cualquier sitio web; capturan la información de los usuarios e interactúan con ellos, lo que se debe garantizar es que funcionen correctamente.

Un ejemplo sencillo de un formulario es cuando se le solicita información a usted para registrase en una red social, por ejemplo:

Sign Up
It's free, and always will be.
First Name:
Last Name:
Your Email:
Re-enter Email:
New Password:
I am: Select Sex:
Birthday: Month: Day: Year:
Why do I need to provide this?
Sign Up

Figura 11. Ejemplo de registro a una red social, mediante un formulario (Fuente: registro público de Facebook)

Pruebas de "cookies"

Los *cookies* son archivos de información que se almacenan en la computadora del visitante de un sitio, información que el servidor requiere cada vez que se ingrese a dicha página web. Esto para agilizar la comunicación y disponer de información valiosa, tal como la sesión del usuario.

Para la prueba de cookies, se recomienda:

- ✓ Probar la aplicación activando y desactivando los *cookies* de su *browser*.
- ✓ Probar si los *cookies* están correctamente encriptados (cifrado, con información no entendible para el usuario o herramienta de *software*) antes de almacenarse en el computador del visitante.
- ✓ Probar las sesiones de usuario según los *cookies*.

ivan@rad.msn[1] - Notepad

File Edit Format View Help


```
FC09FB=rad.msn.com/92163195805696302608492132923248301139 62*FC04FB=rad.msn.com/921631958056963026084 9
213272324830113962*format_name63847Counter0rad.msn.com/10881600870430153376782856864 30117166*
format_name63847SinceDateFri 26 Nov 2010 05:53:00 UTCrad.msn.com/1088160087043015337678288686430117166*
format_name63847LastPrintFri 26 Nov 2010 05:53:00 UTCrad.msn.com/1088160087043015337678291686430117166*
format_name_count_63847Counter1rad.msn.com/1088282600870430153376359299686430117166*
format_name_count_63847SinceDateFri 26 Nov 2010 05:53:00 UTCrad.msn.com/10882826008704301533763593036864
30117166*format_name_count_63847LastPrintFri 26 Nov 2010 05:57:41 UTCrad.msn.com/108828260087043015337 6
359306686430117166*format_name_count_61949Counter0rad.msn.com/10882620718976301481693390687136301 11959*
format_name_count_61949SinceDateSun 31 Oct 2010 08:44:11 UTCrad.msn.com/10882620718976301481693390737136
30111959*format_name_count_61949LastPrintSun 31 Oct 2010 08:44:11 UTCrad.msn.com/10882620718976301481 69
339078713630111959*format_name63069Counter0rad.msn.com/108843186432030149580120043248030113370*
format_name63069SinceDateSun 7 Nov 2010 09:00:52 UTCrad.msn.com/108843186432030149580120053248030113370*
format_name63069LastPrintSun 7 Nov 2010 09:00:52 UTCrad.msn.com/108843186432030149580120062248030113370*
format_name_count_63069Counter0rad.msn.com/108843186432030149580120074248030113370*
format_name_count_63069SinceDateSun 7 Nov 2010 09:00:52 UTCrad.msn.com/108843186432030149580120084248 0
30113370*format_name_count_63069LastPrintSun 7 Nov 2010 09:00:52 UTCrad.msn.com/108843186432030149580
120089248030113370*format_name63938Counter0rad.msn.com/108812412250883015017220110832483011396 2*
format_name63938SinceDateWed 10 Nov 2010 07:39:55 UTCrad.msn.com/1088124122508830150172201115324830113962*
format_name63938LastPrintWed 10 Nov 2010 07:39:55 UTCrad.msn.com/1088124122508830150172201119324830113962*
format_name_count_63938Counter2rad.msn.com/108846135590430150176122935406430113966*
```


Figura 12. Ejemplo de un cookie

Pruebas de la base de datos

Estas consisten en validar la integridad de los datos y su consistencia. El objetivo es asegurar que no se presenten errores en el momento de editar, borrar y modificar formularios en el sitio web.

Es necesario chequear que todos los *query* (instrucción estructural interpretada por el motor de base de datos para consultar información almacenada) se ejecuten correctamente.

5.2 Test de usabilidad

La aplicación de pruebas para validar qué tan intuitiva y fácil de usar es la aplicación web para el usuario se denomina test de usabilidad.

Para garantizar la usabilidad de la aplicación, se recomiendan, al menos, las siguientes pruebas:

Pruebas de navegación

Estas pruebas se centran en cómo el usuario navega por la página; para ello, utilizando los diferentes enlaces, botones y formularios.

Las pruebas de navegación deben incluir:

- ✓ Validación de la facilidad de navegación, las instrucciones deben ser claras.
- ✓ Validar si las instrucciones son correctas y cumplen con su objetivo.
- ✓ Validar el correcto funcionamiento de los menús.

Pruebas de contenido

En este caso, llevan a cabo el chequeo exhaustivo del texto escrito en la aplicación y la presentación, en general, de su contenido.

Las pruebas de este tipo incluyen:

- ✓ Validar que el contenido sea lógico y fácil de entender.
- ✓ Revisar los errores gramaticales y ortográficos.
- ✓ Validar que los textos que hacen referencia a otro texto funcionen correctamente.
- ✓ Revisar que las imágenes estén bajo el estándar correcto (tamaño, claridad y presentación).

Pruebas de ayuda

Las pruebas de este tipo revisan la ayuda que puede presentar la aplicación al usuario final, para que sea más fácil el uso del sitio y sin problemas.

Incluye:

- ✓ Validar las opciones de búsqueda a fin de que ayuden realmente al usuario a encontrar páginas que ellos buscan más fácil y rápidamente.

- ✓ Validar que los mapas del sitio presenten los *links* con una vista de navegación en árbol.
- ✓ Validar que la ayuda *online* funcione correctamente y que le muestra al usuario la información requerida por él.

Estos elementos son opcionales, pero que si se presentan, se debe garantizar el correcto funcionamiento.

5.3 Test de interfaz

El test de interfaz se refiere a la validación del correcto comportamiento de enlaces; en este casos las interfaces, entre las capas y las capas son la separación lógica de la presentación, la lógica del negocio y los datos; por ejemplo: cliente–servidor–base de datos.

Entre las pruebas de interfaz, se debe revisar, por ejemplo, lo siguiente:

- ✓ Validar que cuando la base de datos (capa de datos) devuelve un error, estos deberán capturarse y mostrarse adecuadamente al usuario (capa de presentación o cliente).
- ✓ Validar el correcto funcionamiento de la aplicación cuando se interrumpe una transacción, verificar que los datos son íntegros y consistentes.
- ✓ Validar que las instrucciones usadas en la página son correctas.
- ✓ Comprobar el tamaño (peso en KB o MB) de las imágenes, si el texto se muestra rápidamente o no. Para estos casos probablemente, es factible realizar una prueba de ancho de banda, la cual básicamente determina qué tan rápido es el enlace con que trabajamos y el que se recomienda. Por ejemplo, *dependiendo* del tamaño de las imágenes o información por transmitir, se podría recomendar enlaces de 128 KB, 512 KB o de hasta 1 MB.
- ✓ Verificar el cómo los motores de búsqueda ven la aplicación desarrollada.
- ✓ Validar que el sitio web funciona correctamente cuando se desactivan las imágenes. Los usuarios las deshabilitan cuando las conexiones son lentas o se utilizan dispositivos móviles (aplicaciones WAP, protocolo utilizado para móvil).

- ✓ Verificar el comportamiento del sitio cuando se cambia la resolución de la computadora.
- ✓ Verificar que las impresiones que ofrece el sitio sean correctas.

5.4 Test de compatibilidad

Las pruebas de compatibilidad son muy relevantes, pero van sujetas hasta dónde la organización puede certificar su aplicación web.

Una compañía puede sacar al mercado una aplicación web que funcione perfectamente con el browser de Internet Explorer, pero puede que no funcione bien con el *browser* Opera, por ejemplo. En estos casos, la organización debe comunicar y hacer la advertencia al usuario final en su sitio web. Igual pasa con el uso de los sistemas operativos, puede que la aplicación web funcione perfectamente en Microsoft Windows, pero no así en Macintosh, o viceversa.

Los costos para certificar un sitio web en los *browser* y en los sistemas operativos es alto, así que lo recomendable es que las organizaciones certifiquen su aplicación en uno o dos de ellos (los más usados), y en uno o dos sistemas operativos (igualmente en los más usados), y que esto lo muestre en su sitio. Si el usuario desea, aún así, utilizar un sistema operativo o *browser* no certificado por la empresa, esta decisión será bajo su propio riesgo.

Así, los test recomendables, mínimos, para la compatibilidad son:

- ✓ de navegadores
- ✓ de sistema operativo (SO)
- ✓ de navegación móvil
- ✓ opciones de impresión

Pruebas de navegadores

Los navegadores tienen diferentes configuraciones que su aplicación debería tratar. Por lo tanto, el desarrollo de su aplicación debe tener código funcional en los diferentes navegadores que usted desea garantizar.

El navegador más utilizado en el mundo es Microsoft Explorer, y el segundo es Mozilla Firefox, por lo que es común que las aplicaciones web se desarrollen con la intensión de que funcione en al menos esos dos *browser*.

Pruebas de Sistema operativo

Algunas de las funcionalidades de tu aplicación puede que no sean compatibles con los sistemas operativos, puede que surjan problemas en las nuevas tecnologías para el desarrollo web, como los gráficos o interfaces que realicen llamados a los diferentes API (*Application Programming Interface* por sus siglas en inglés). Por eso, debe tomarse en cuenta este importante aspecto para su desarrollo, definir y garantizar en qué sistemas operativos su aplicación se ejecutará correctamente.

Pruebas de navegación móvil

En la era de la información, la actual, cada vez se usa más la navegación móvil (Internet en celulares u otros dispositivos), por eso, poco a poco la globalización obliga a que las organizaciones, además de crear aplicaciones web funcionales, también exige a que estas sean funcionales en dispositivos móviles, para facilidad del usuario final.

Las organizaciones deben tomar en cuenta, también, ese aspecto si quieren incursionar más fuertemente en el mercado actual.

Pruebas de opciones de impresión

Actualmente, la tendencia de imprimir está perdiendo fuerza entre las organizaciones, pues ahora se quiere que todo sea digital y que se muestre en pantalla, para reducir costos y proteger el planeta. No obstante, aún está en proceso de transición, todavía existen personas y organizaciones que se resisten al cambio o lo están haciendo poco a poco, entonces, se les debe ofrecer aún las elecciones de imprimir y garantizar que funcionen correctamente desde la opción de impresión del navegador utilizado o desde la de su aplicación web.

5.5 Pruebas de rendimiento

Las pruebas más necesarias para las aplicaciones web son las de carga y las de estrés; esto garantiza que su aplicación estará diseñada correctamente para el incremento en la demanda de usuarios el cual tendrá en la red de Internet.

Pruebas de carga

Pruebas que se encargan de garantizar que si, por ejemplo, muchos usuarios acceden a la misma página y se realizan múltiples solicitudes al servidor, este responderá en tiempo y funcionalidad correcta.

Pruebas de estrés

Estrés significa llevar un sistema más allá de sus límites. Las pruebas de estrés se realizan con el fin de intentar "botar" el sistema, detenerlo, que deje de funcionar, y así, comprobar su capacidad, su reacción, y verificar su comportamiento ante una "caída". En el módulo II, se explicaron estos tipos de prueba.

5.6 Pruebas de seguridad técnica

En la web, cada vez es más significativo realizar pruebas de seguridad, a causa de los constantes ataques a sitios web por parte de los *cracker* (persona que utiliza sus conocimientos técnicos para violar la seguridad de los sistemas en beneficio propio o para causar daños. No es un *hacker*).

Algunas ideas de pruebas que se pueden mencionar para garantizar la seguridad de su aplicación web son las siguientes:

- ✓ Verificar que si se pone el url de páginas internas directamente en la barra del Navegador sin *loguearse,* esta no se abrirá.
- ✓ Verificar el comportamiento de los formularios cuando se ponen datos falsos.
- ✓ Verificar que las carpetas web y archivos no son accedidos directamente, excepto por las opciones de descarga del sitio.
- ✓ Verificar que las páginas en las cuales se presente vulnerabilidad de ataque a los servidores, por ejemplo, formularios de registro o consulta, ofrezcan la posibilidad de CAPTCHA (acrónimo de *Completely Automated Public*). Estos garantizan que quien está utilizando el sistema es un humano, ya que él mismo dificulta el reconocimiento de la máquina por su distorsión y degradado de fondo.

Figura 13. Ejemplo de un CAPTCHA

- ✓ Verificar el correcto funcionamiento del SSL (*Secure Socket Layer* por sus siglas en inglés). El SSL es un servicio de seguridad creado por Netscape Communications Corporation, el cual se encarga de cifrar la información que viaja entre el servidor y el cliente. El sitio web debe comunicar al usuario cuando este pasa a una página, segura o no.
- ✓ Verificar que el registro de las transacciones, *logueos* de usuarios, intentos de ataque, etc., deben quedar registrados en una bitácora en la base de datos o *log* en el servidor.

- ✓ Verificar el correcto funcionamiento del *time-out* o límite de tiempo de ejecución de transacciones.
- ✓ Verificar cuidadosamente que su aplicación no tenga la vulnerabilidad de inyección de SQL, es decir, a partir de parámetros dados por el usuario para consulta o modificación de la base de datos, el sistema arma el SQL, que será luego ejecutado por el motor de base de datos. Muchos ataques se originan porque el *cracker* se aprovecha de esas vulnerabilidades y arman su propio SQL con fines maliciosos y que luego modifican la base de datos a placer.
- ✓ Verificar que los datos no se pierden en un formulario cuando el usuario da *back* (atrás) o consecuentemente *forward* (adelante).

Tema 6. Pruebas automatizadas

Este tema le enseñará que, además de la existencia de las pruebas manuales tradicionales, también existe la posibilidad de automatizarlas, las cuales brindan un gran valor agregado a su tiempo de ejecución, y amplias ventajas para simular cargas excesivas de usuarios y datos. Así se comprueba el buen diseño de un *software*; sin embargo, no todo se debe automatizar, pues existen desventajas por evaluar en el momento de la automatización. Estos temas serán tratados acá y usted aprenderá a definir, con criterio, qué debe y qué no debe automatizar en su organización.

Se explicará, también, cómo se realiza una automatización de pruebas y, con base en ello, usted podrá, eventualmente, investigar las herramientas que se ofrecen en el mercado y seleccionar la que se ajuste más a sus necesidades.

6.1 Importancia de las pruebas automatizadas

Conforme avanza el desarrollo tecnológico y las aplicaciones, igualmente se ha creado la necesidad de invertir en el desarrollo de pruebas, que garanticen el correcto funcionamiento de *software* nuevo de tecnología de punta.

Ahora ya no es suficiente la creación de pruebas solo para aplicaciones *Winform* (comúnmente conocidas, en donde no interactúa con un *browser*), sino que las tendencias en desarrollo de *software* son aplicaciones en la Web, accedidas por medio de un *browser*, en el cual múltiples usuarios pueden realizar transacciones concurrentemente (procesos que interactúan al mismo tiempo y utilizan los mismos recursos tecnológicos).

Anteriormente, con una única computadora, se podían realizar las pruebas a un sistema; ahora, es necesario simular cómo se comporta un sistema accedido por web si se sabe que una organización, un país, o inclusive, el mundo lo puede usar.

De acuerdo con lo expuesto anteriormente (desarrollos web), las pruebas automatizadas son esenciales, pues es necesario comprobar el rendimiento de una aplicación cuando muchos usuarios la acceden, ya que usted puede simular diferentes escenarios para las pruebas de rendimiento.

Para dar un ejemplo, suponga que usted desarrolló una aplicación para que todo el país consulte el padrón electoral, usted no sabe la capacidad de enlace de una computadora casera (puede ser de 128 KB, de 512 KB o aún de más velocidad como por ejemplo de 1 MB), tampoco usted sabe la capacidad de enlace de una organización ni en qué momento los usuarios decidirán acceder a su aplicación, ¿qué pasa si todos los usuarios deciden acceder al mismo tiempo? Como sucede cuando se ponen a la venta boletos para un evento muy importante como, por ejemplo, la inauguración del nuevo Estadio Nacional de Costa Rica. Si su aplicación no es probada adecuadamente con la ayuda de *testing* automatizado, muy probablemente cuando esté funcionando, en producción, esta falle y, en las primeras cargas de los usuarios, deje de funcionar.

Si desea probar adecuadamente su *software* y simular el comportamiento real cuando esté en producción, usted debe demostrar el rendimiento de su aplicación con test de carga y de estrés (explicados en módulos anteriores) y, para ello, son necesarias las pruebas automatizadas.

Algunas herramientas que ofrecen la posibilidad de automatizar son: Microsoft Visual Studio, HP Quality Center y las aplicaciones *OpenSource* descritas en el módulo I, y algunas de estas herramientas ofrecen no solo la posibilidad de automatizar pruebas para aplicaciones web, sino también para *Winform*.

6.2 Ventajas de las pruebas automatizadas

Las pruebas automatizadas presentan muchas ventajas, pues son útiles para probar el rendimiento de una aplicación, la funcionalidad y la modularidad de ellas.

Entre dichas ventajas están:

- ✓ Se puede automatizar las pruebas unitarias.
- ✓ Se puede automatizar las pruebas funcionales.
- ✓ Se puede realizar pruebas de rendimiento (carga y estrés).
- ✓ El resultado de las pruebas es mucho más rápido.
- ✓ La densidad de las pruebas de calidad aumenta, esto es, se abarcan más puntos de calidad; por ejemplo, la seguridad, la funcionalidad, el rendimiento, etc.
- ✓ Se puede simular diferentes escenarios con mucho más facilidad.
- ✓ Las pruebas diseñadas pueden evitar el error humano.
- ✓ Se puede reutilizar muchas veces.

Por ejemplo, suponga que usted es un *tester* que debe probar una aplicación con alta operatividad, y debe garantizar su funcionamiento para los *browser* Firefox, Microsoft Explorer y Opera. Además debe garantizar que funciona en los sistemas operativos de Microsoft y Macintosh, así como en diferentes ambientes con distintos aplicaciones de usuario final.

¿Qué debe hacer para garantizar todo esto? Para ejecutar una misma prueba en toda esa diversidad de ambientes, debería realizar una prueba automatizada para su aplicación, que garantice, al menos, el rendimiento y la funcionalidad; luego, definirle diferentes escenarios, en los cuales puede combinar las variables de ambiente y, así, determinar si la aplicación funciona correctamente para todo.

Ahora, suponga que usted ya tiene la prueba automatizada y lo único que debe hacer es realizar todas las posibles permutaciones de ambiente válidas para luego ejecutar la prueba. Un ejemplo de los escenarios sería:

Escenario 1		
Browser utilizado	**Sistema operativo**	**Aplicaciones de usuario final**
Microsoft Explorer	Microsoft Windows	Microsoft Office, Microsoft Project

Escenario 2		
Browser utilizado	**Sistema operativo**	**Aplicaciones de usuario final**
Microsoft Explorer	Microsoft Windows	Adobe Reader, Aplicación de Recursos Humanos

Escenario 3		
Browser utilizado	**Sistema operativo**	**Aplicaciones de usuario final**
Mozilla Firefox	Microsoft Windows	Microsoft Office, Microsoft Project

Escenario 4		
Browser utilizado	**Sistema operativo**	**Aplicaciones de usuario final**
Mozilla Firefox	Macintosh	Scribus y otras aplicaciones *Opensource*

Usted obtendrá un resultado para cada escenario, y será insumo para reportar defectos, mejoras o para decidir si es conveniente que la aplicación solamente se garantice para un tipo de navegador, o para un tipo de sistema operativo, o para decidir si no se recomiendan ciertas aplicaciones instaladas en los equipos de los usuarios o, incluso, para definir si el proyecto no se libera en producción en el tiempo determinado.

6.3 Desventajas de las pruebas automatizadas

Aún con todas las ventajas que se pueden identificar de las pruebas automatizadas, también existen desventajas por tomar en cuenta antes de aplicarlas. A la vez, le deben guiar para saber qué automatizar y qué no, y en qué momento. Considere lo siguiente:

- ✓ Requiere de un alto presupuesto.
- ✓ En ocasiones, es complicado realizarla.
- ✓ Se requiere de mucho tiempo.
- ✓ Si cambia el requerimiento o se incluye una mejora, también se debe considerar el cambio en la prueba. El mantenimiento puede ser costoso.

Por ejemplo, imagine que, como *tester*, debe realizar las pruebas de un sistema nuevo que será accedido por múltiples usuarios; este sistema está compuesto por diferentes módulos para diferentes funcionalidades. Usted desarrolla una prueba automatizada para cada módulo, a fin de evitar que la validación y verificación sea manual y que el tiempo de ejecución y el resultado de las pruebas sean más rápidos.

Ahora, suponga que se implementó una mejora, y tres de esos módulos se unieron, además, se eliminó funcionalidad y se agregó otra, por ende, los cálculos se modificaron, ¿qué pasa con la prueba automatizada que usted ya había desarrollado? Probablemente, esa prueba ya no aplica y deberá modificarla también.

Se recomienda, entonces, que una prueba se automatice si tiene características de poco cambio y si se utiliza muy constantemente.

6.4 Herramientas automatizadas: desarrollo de una prueba

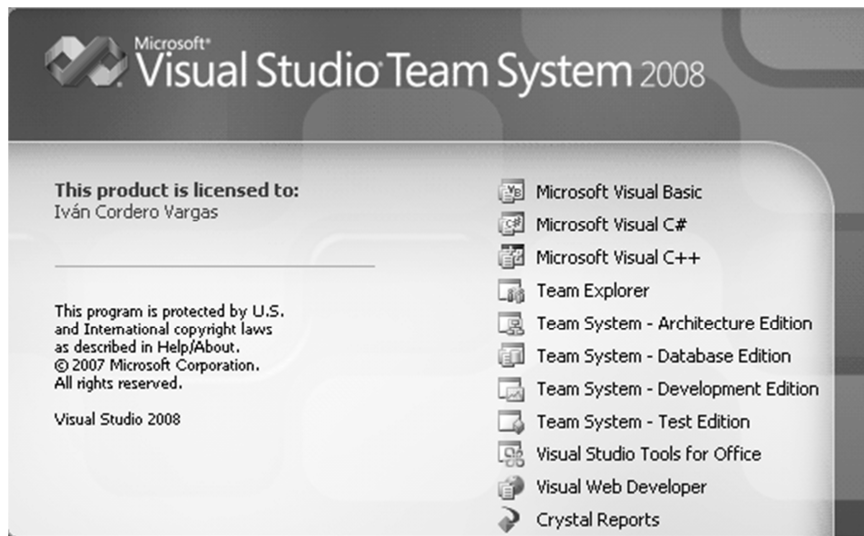
A continuación, se describirá cómo ocurre una prueba automatizada; esta en particular, se desarrolla en la herramienta Microsoft Visual Studio 2008 (VS2008), y se hará para una prueba de carga. El objetivo de mostrarle a usted el desarrollo de una prueba de este tipo es para que se familiarice con ellas, y le motive a investigar sobre lo que ofrecen las herramientas del mercado, a fin de mejorar la calidad de nuestros sistemas. Este ejemplo en particular es muy sencillo, pero una prueba de carga, estrés o funcional puede ser tan compleja como el sistema mismo.

Se desarrollarán dos pruebas: una *web test* y otra *load test* que, simplemente, desean medir el rendimiento de acceso a la aplicación, *log in* y *log out* del usuario, o sea, ingresar y salir de la aplicación, pero con muchos usuarios a la vez.

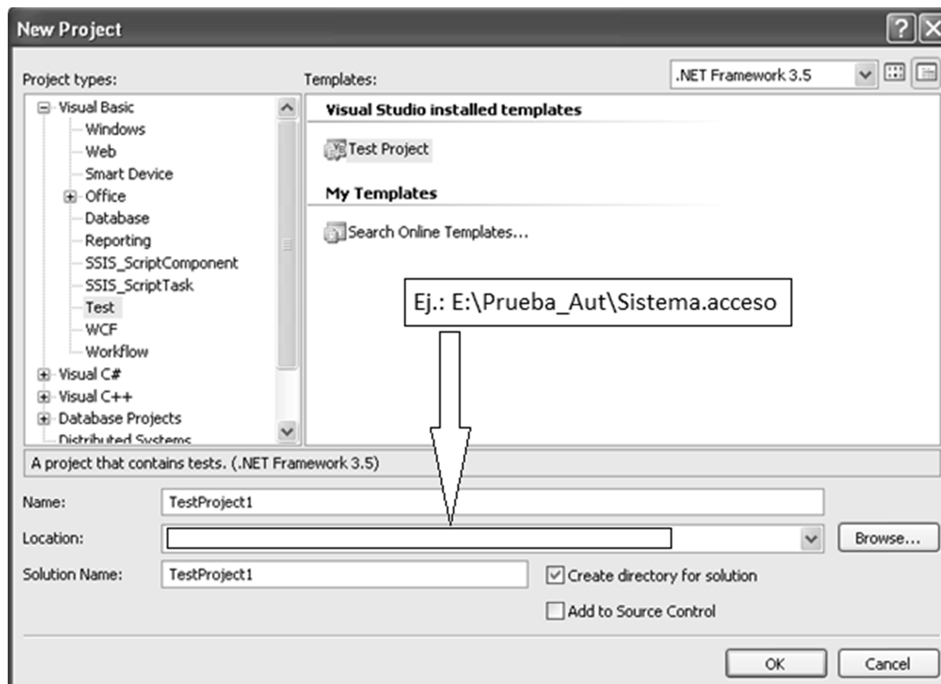
El detalle sobre cómo se debe elaborar una prueba no se explicará a fondo en este apartado, ya que la fabricación de test automatizado en Microsoft Visual Studio 2008 conlleva un alcance muy amplio, y no es posible abarcarlo acá, pues sería un curso completo.

Inicio del proyecto

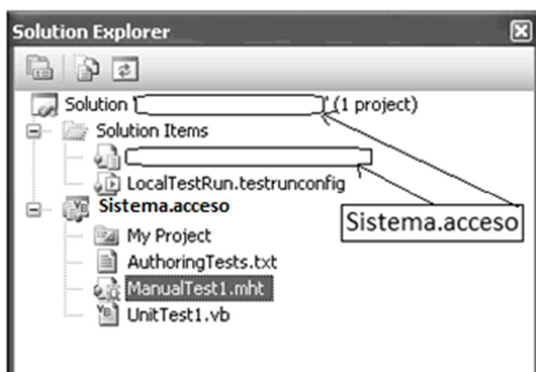
Ingrese a la herramienta para el desarrollo de la prueba automatizada, en este caso, Visual Studio Team System 2008.



Seleccione luego un proyecto tipo *Test Project* y dele una ubicación para su prueba.

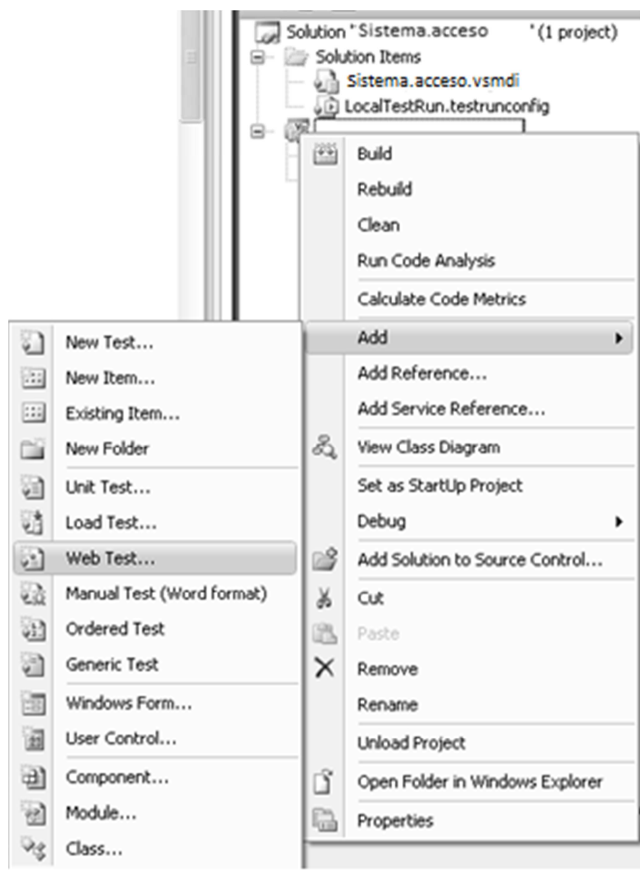


Una vez definido dónde quedará ubicado su proyecto, la siguiente imagen muestra cómo queda una solución típica de una prueba automatizada en esta herramienta (VS2008).



Generación del web test

Para crear un *web test*, cuyo concepto se explicó en módulos anteriores, dé clic derecho sobre el proyecto, luego *Add* y después seleccione *web test.*

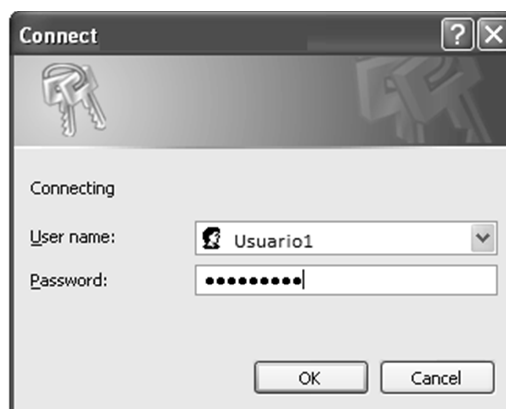


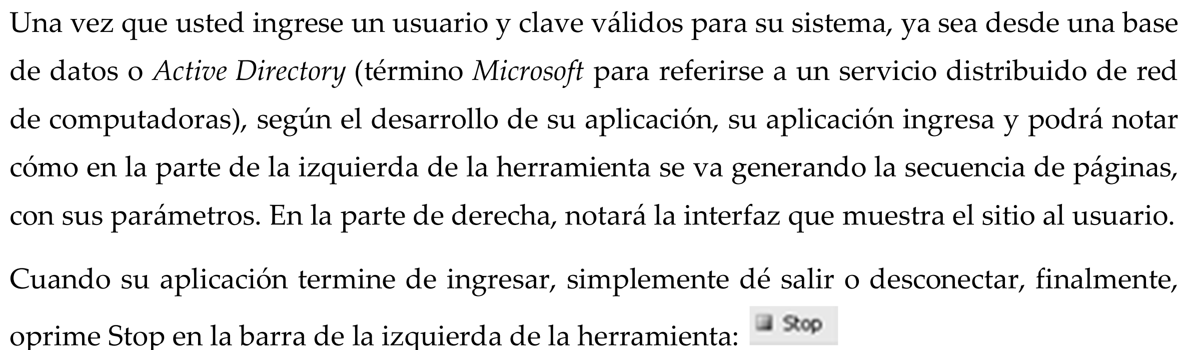
Seguidamente, de la selección del *web test*, aparecerá una página para que usted grabe los pasos de la prueba, aquí usted realiza los pasos los cuales desea que formen parte de su prueba. Para este caso en particular, solamente se desea ingresar y salir del sistema.

En la barra superior del navegador que aparece, usted digita el url de su sistema.

Luego, cuando se inserta el url, y orpime la tecla *Enter*, entrará en funcionamiento su sitio *web*. En este momento, el sitio está siendo grabado por la herramienta de prueba automatizada, para que, posteriormente, cuando lo ejecute, el sitio haga exactamente lo mismo que usted hizo, de forma automática.

En esta prueba particular, el sitio solicita usuario y clave para ingresar:

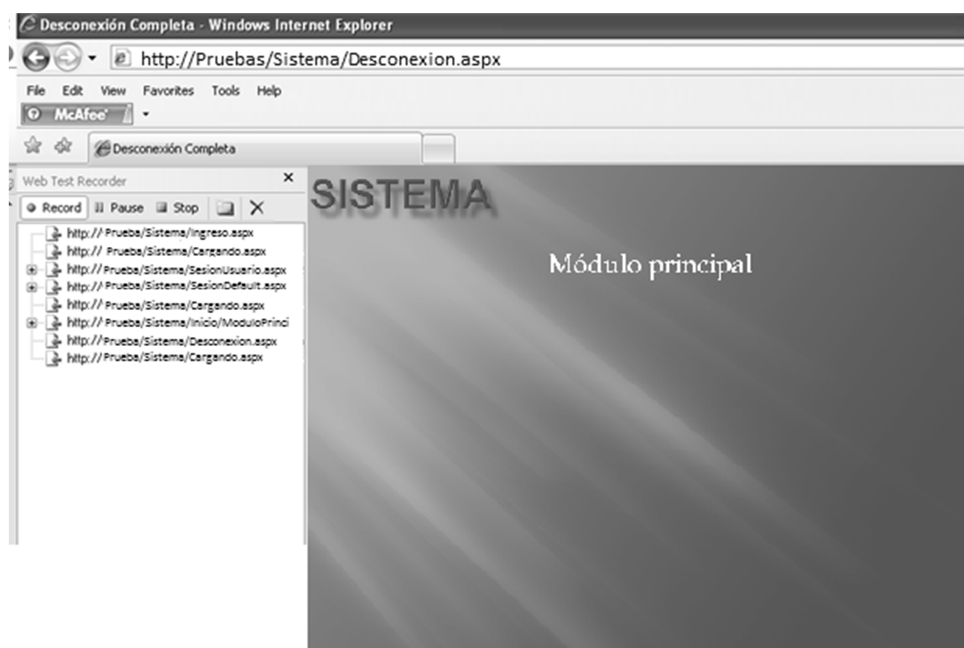




Una vez que usted ingrese un usuario y clave válidos para su sistema, ya sea desde una base de datos o *Active Directory* (término *Microsoft* para referirse a un servicio distribuido de red de computadoras), según el desarrollo de su aplicación, su aplicación ingresa y podrá notar cómo en la parte de la izquierda de la herramienta se va generando la secuencia de páginas, con sus parámetros. En la parte de derecha, notará la interfaz que muestra el sitio al usuario.

Cuando su aplicación termine de ingresar, simplemente dé salir o desconectar, finalmente, oprime Stop en la barra de la izquierda de la herramienta: Stop

La generación de páginas del lado izquierdo es sumamente importante, porque es ahí donde usted podrá manipular su aplicación de pruebas para definir un *pool* de datos (un conjunto de datos), eliminar páginas inútiles, etc.



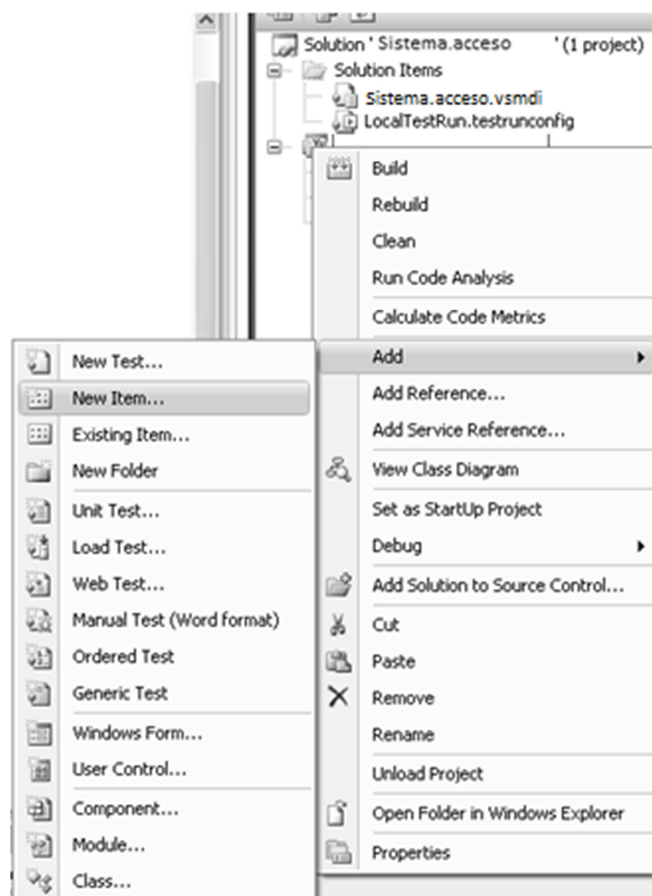
Generación y utilización de un pool *de datos*

Cuando usted grabó el ingreso y salida de su aplicación, esta fue grabada con el nombre de usuario y clave que usted empleó, pero no sería funcional para una prueba de rendimiento usar siempre el mismo usuario, ya que si se realiza la carga de 20, 30 o 50 usuarios al mismo tiempo sobre tu sitio (prueba de carga), el usuario sería solo uno, por lo que estaría siendo utilizado en el primer ingreso.

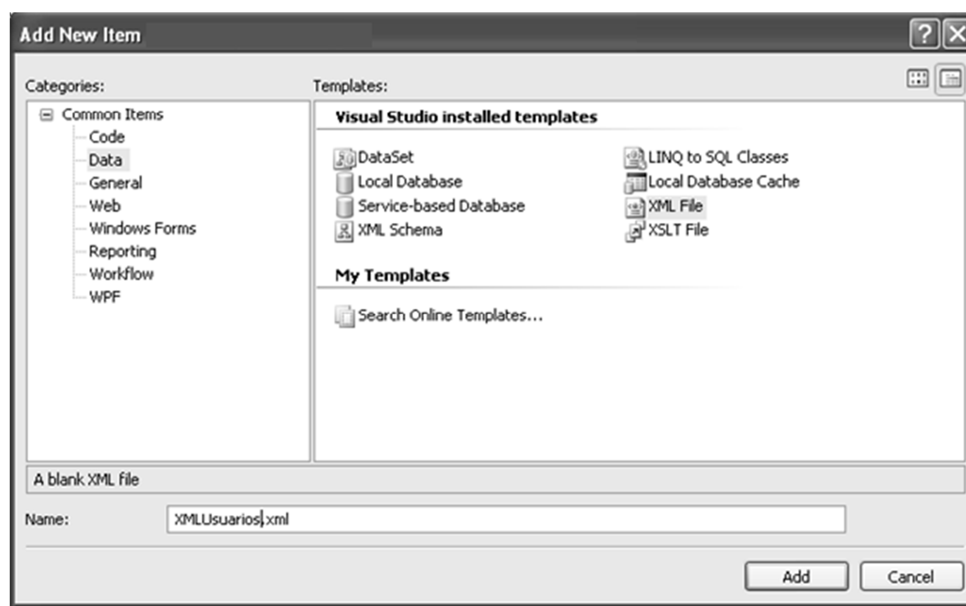
Con base en la última explicación, se hace útil un *pool* de datos, que contenga 50, 100, 200 o miles de usuarios que su sistema pueda utilizar.

Para generar un *pool* de datos, use un archivo XML, una base de datos o un archivo CSV (de formato abierto separado por comas, *comma-separated-values*). Para nuestro ejemplo, utilizaremos un archivo XML.

Si se desea seleccionar estos tipos de fuente de datos, dé clic derecho al proyecto, seleccione *Add* y luego *New Item*.



Una vez concluido el paso anterior, te aparecerá una caja de diálogo en la cual debe seleccionar su ítem. Para nuestro efecto, y para un archivo XML, debe seleccionar, en la izquierda, la opción de Data y, en la derecha, la opción de XML file, luego, abajo pone un nombre, que para este ejemplo se opta por XMLUsuarios.xml.



A continuación, se muestra un ejemplo de cómo usted puede crear un archivo XML que contenga usuarios y claves, el cual funcionará luego como fuente de datos para su aplicación.

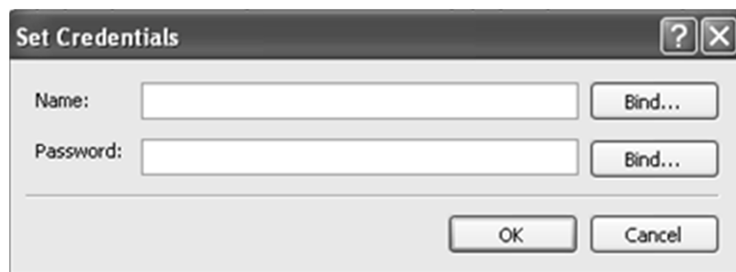
La creación de un archivo XML no se explica en este tema, ya que eso sería una capacitación técnica de otra índole.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<USUARIOS>
  <USUARIO Login="usuario1" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario2" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario3" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario4" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario5" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario6" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario7" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario8" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario9" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario10" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario11" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario12" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario13" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario14" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario15" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario16" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario17" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario18" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario19" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario20" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario21" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario22" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario23" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario24" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario25" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario26" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario27" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario28" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario29" ClaveAcceso="Clave123" />
  <USUARIO Login="usuario30" ClaveAcceso="Clave123" />
```

Ahora bien, una vez creado de forma correcta el archivo XML, lo que necesitamos es asignar el *pool* de *Login* al parámetro de "usuario" y el *pool* de *ClaveAcceso* al parámetro de "clave". El parámetro de "usuario" y "clave" se refieren en la siguiente imagen como "name" y "password".

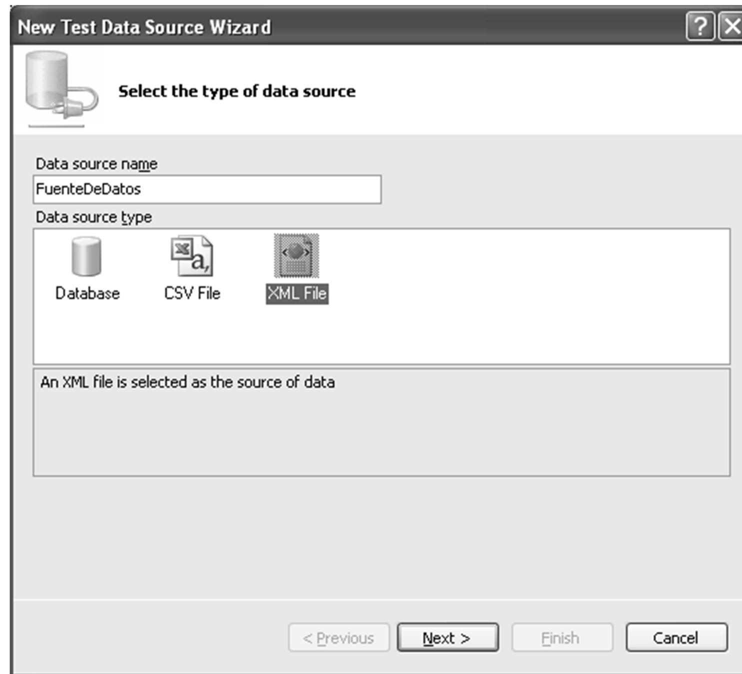
Para asignar el *pool* de datos, se debe presionar el botón llamado "Set Credentials", que se encuentra en la barra de herramientas y que tiene la siguiente figura:

Cuando aparezca la caja de diálogo con la opción de Bind (asociar, ligar, según su traducción en español) para asociar el usuario y la clave, se sigue la secuencia de las siguientes figuras:

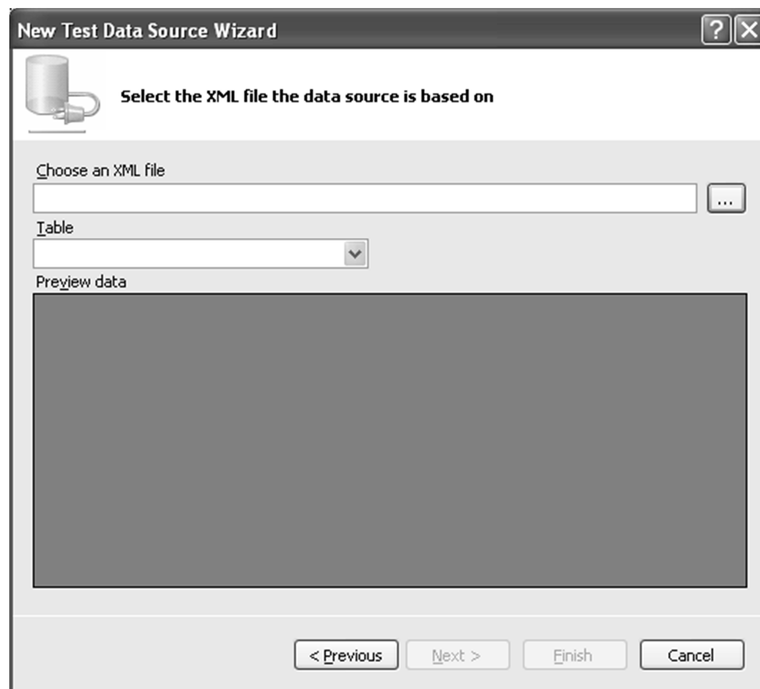


Seleccione Add Data Source para seleccionar su fuente datos, en este caso el archivo XML que usted creó.

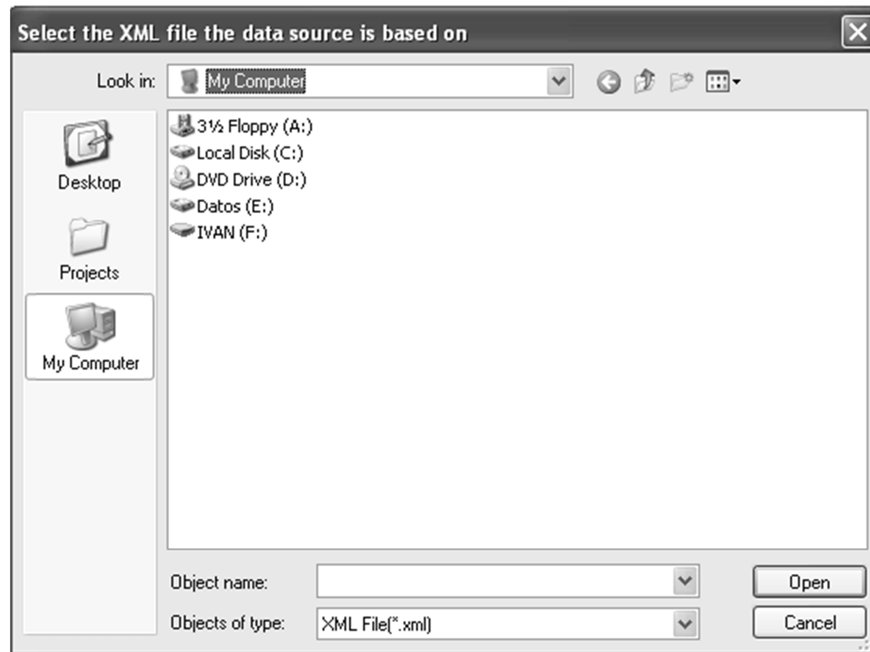
Seleccione el tipo de fuente de datos: XML File.



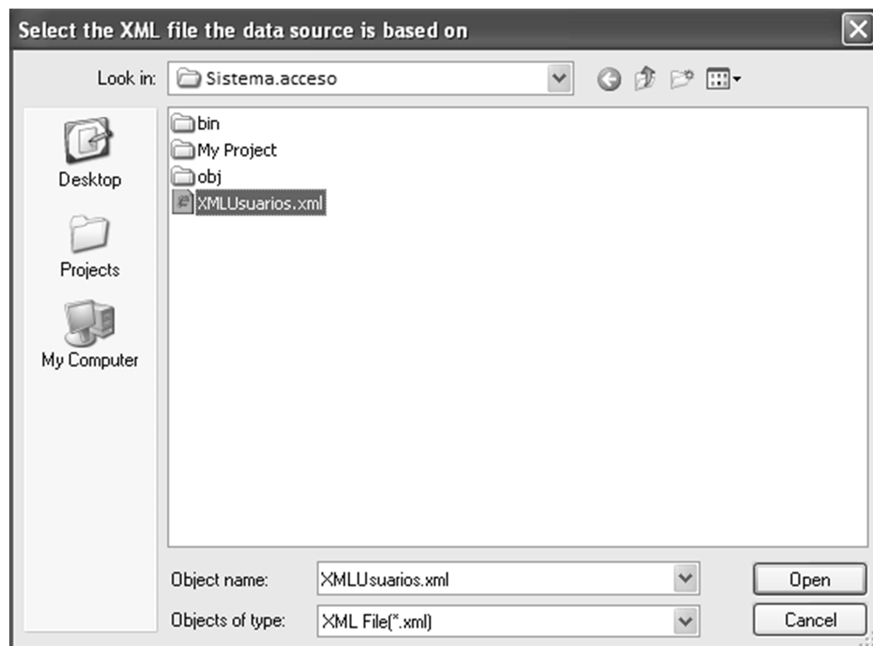
Busque dónde está su archivo XML, o sea, seleccione la ruta de su archivo con el botón



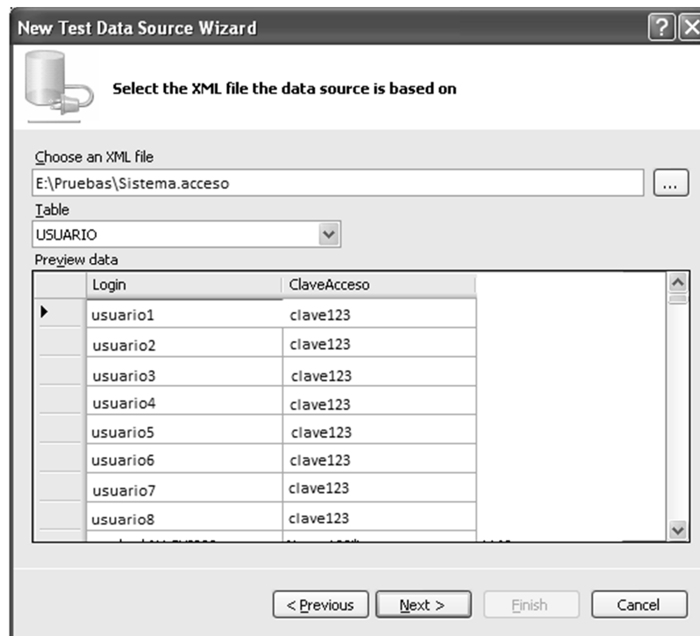
Seleccione el directorio donde está su archivo.



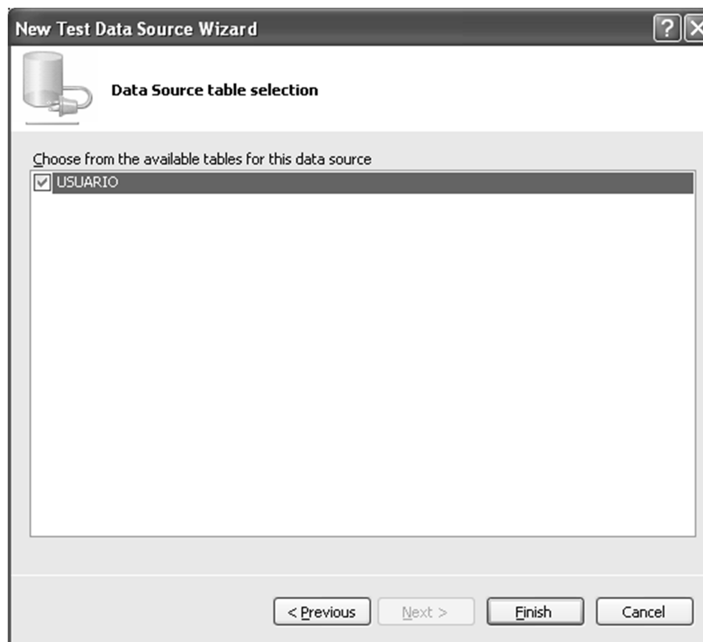
Ubique su archivo, selecciónelo con un clic y presione el botón Open.



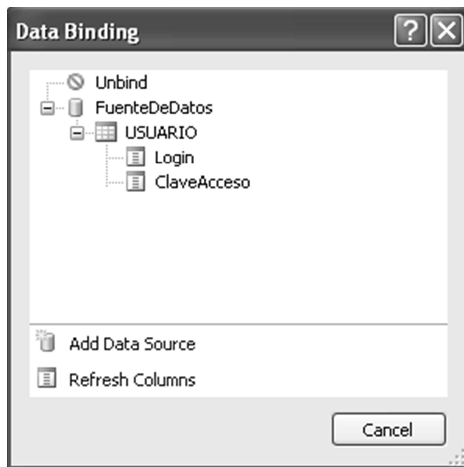
Si su archivo está correctamente creado, aparecerá una pantalla como la siguiente, en donde puede observar sus campos de *Login* y *ClaveAcceso*, con sus respectivos datos. Presione el botón Next >.



Dé clic a la fuente de datos creada por usted (USUARIO es el bloque creado en el archivo XML) y luego dé Finish.

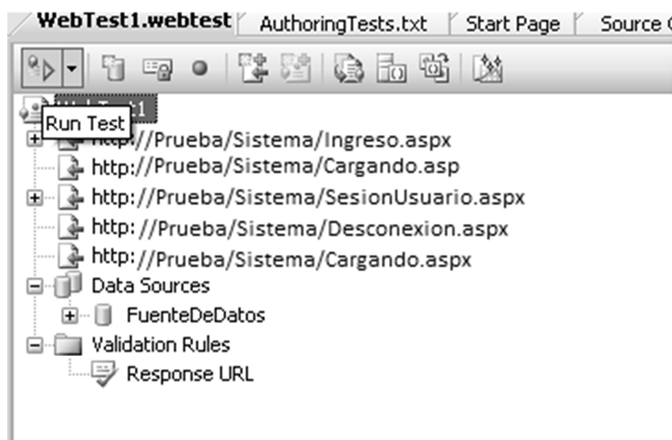


Se abrirá una pantalla que genera su archivo XML en forma de tabla de una base de datos y entonces podrás asociar el campo *Login* al Name y el campo ClaveAcceso al Password, de la caja de diálogo Set Credentials.



Ejecución del web test

Para ejecutar tu *web test* y asegurarse de que funciona con la configuración que realizaste, seleccione su proyecto, se abrirá un *tab* con el nombre de WebTest1.webtest y ahí podrás ejecutar tu prueba presionando el botón con la figura:



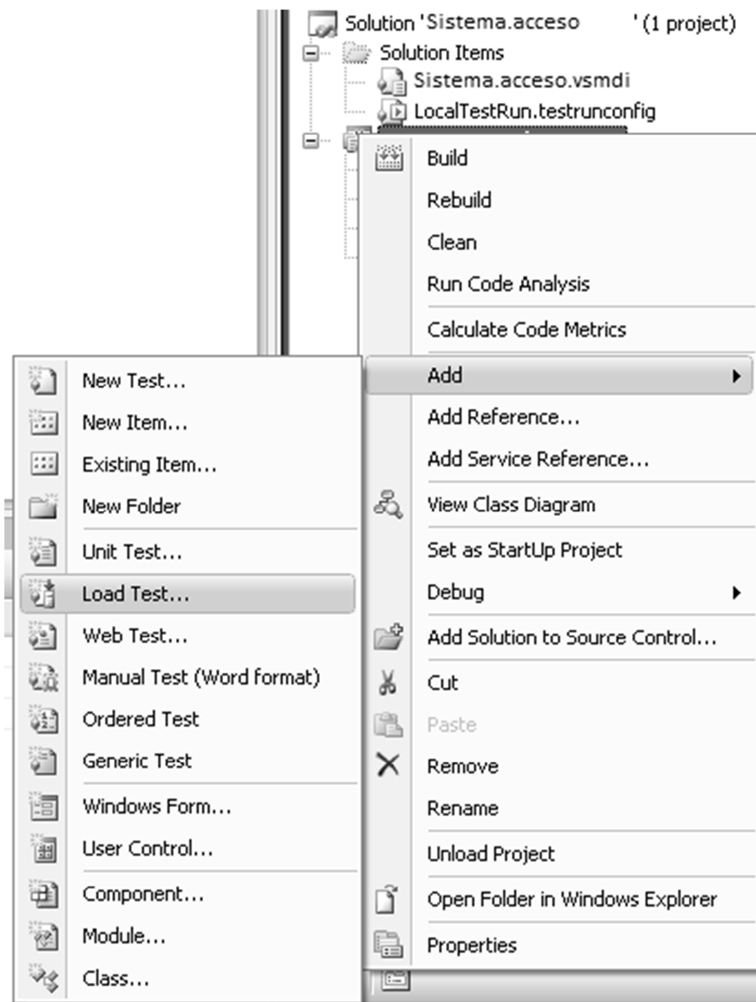
Si el *web test* se ejecuta correctamente, se presentará una pantalla en donde indica que la prueba pasó (*Passed*).

Si el *web test* no es exitoso, por ejemplo, se pone mal la clave de usuario en su archivo XML, entonces, se presentará una pantalla en donde se indica que la prueba falló (*Failed*).

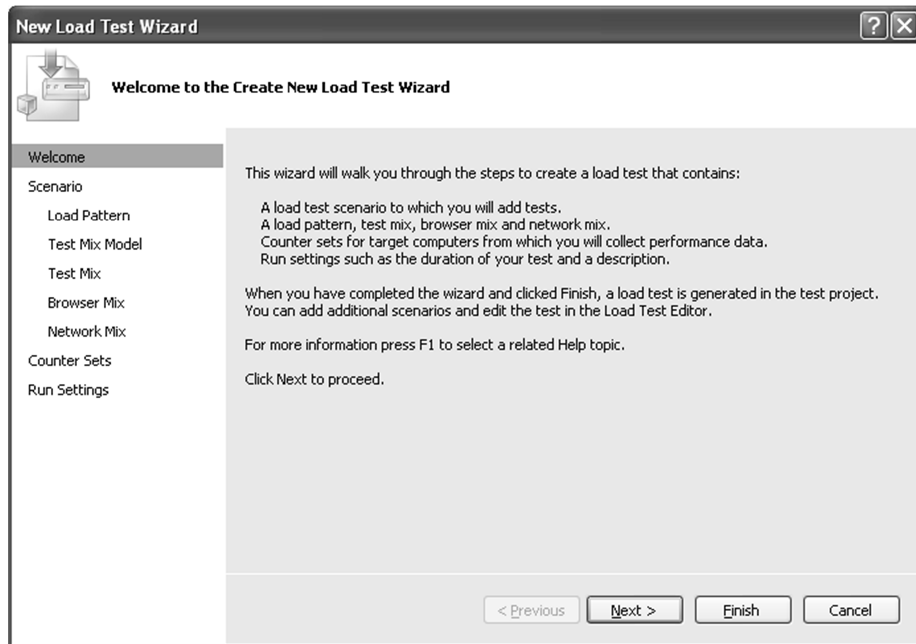
Generación del Load test

El *Load test* es la prueba de carga, la cual ejercita su *web test* con base en los usuarios que usted definió en el archivo XML. En la prueba de carga, usted define el escenario que desee.

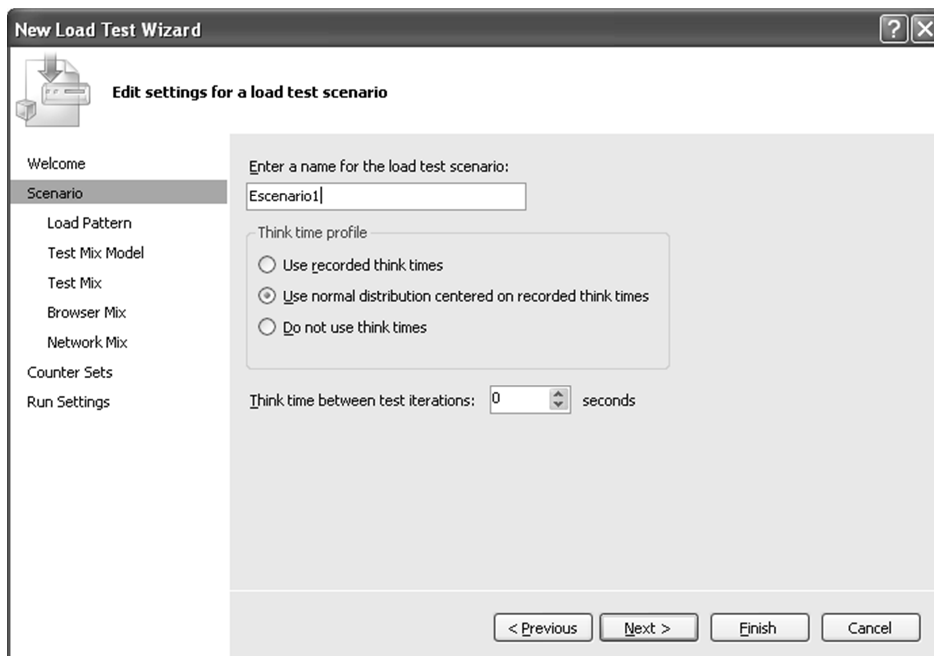
Vaya a su proyecto (Sistema.acceso) y dé clic derecho, seleccione Add y luego seleccione Load Test.



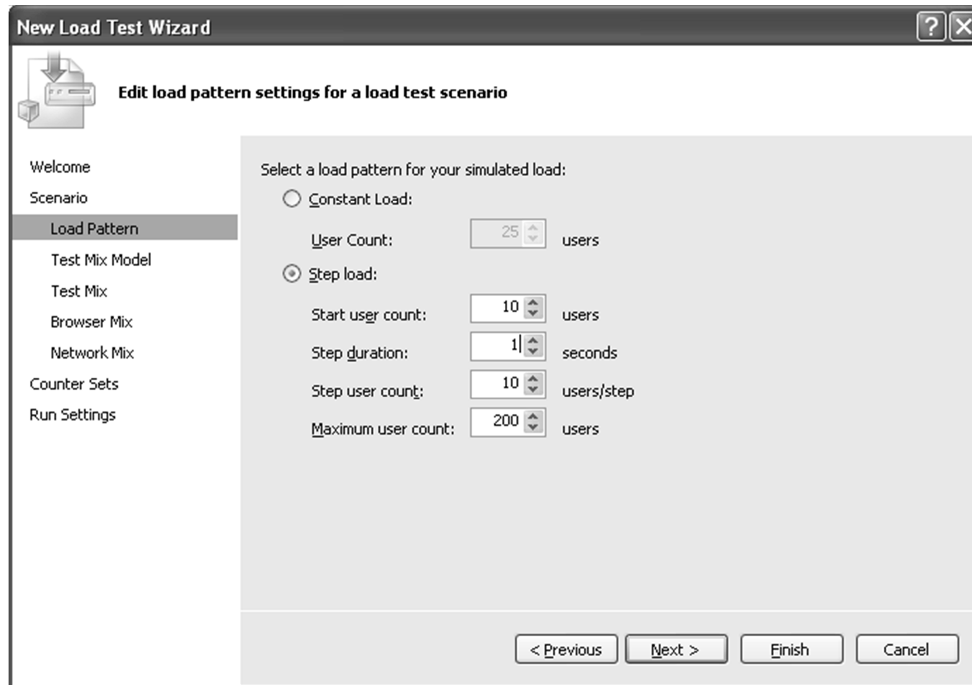
Aparecerá un *wizard* (asistente), el cual usted irá llenando la información necesaria para su escenario.



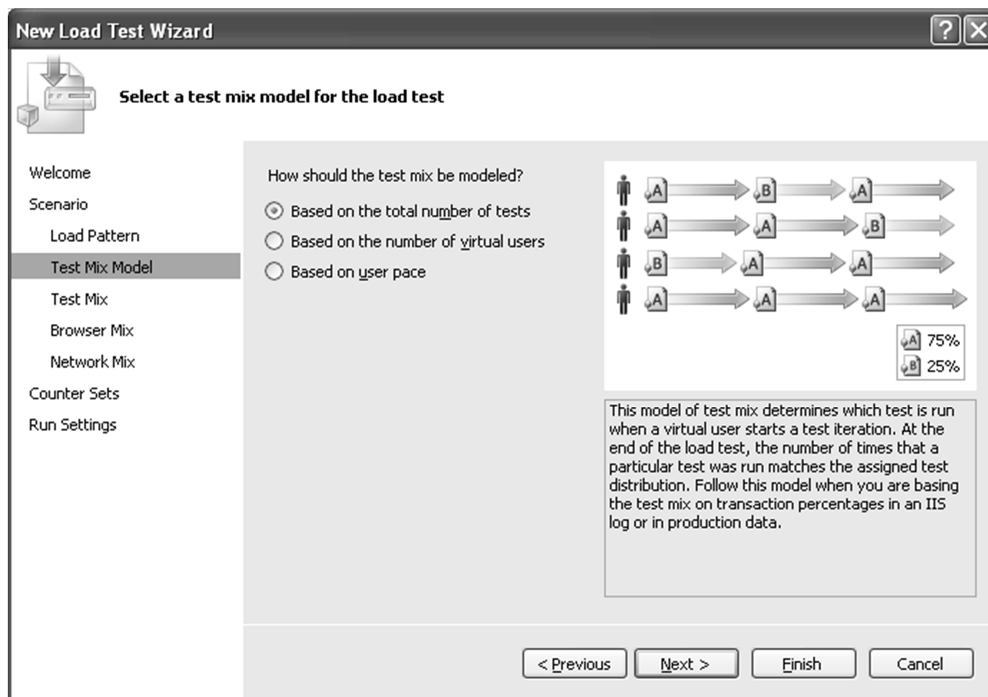
Digite el nombre del escenario y seleccione la simulación del tiempo de espera de un usuario en cada acción.



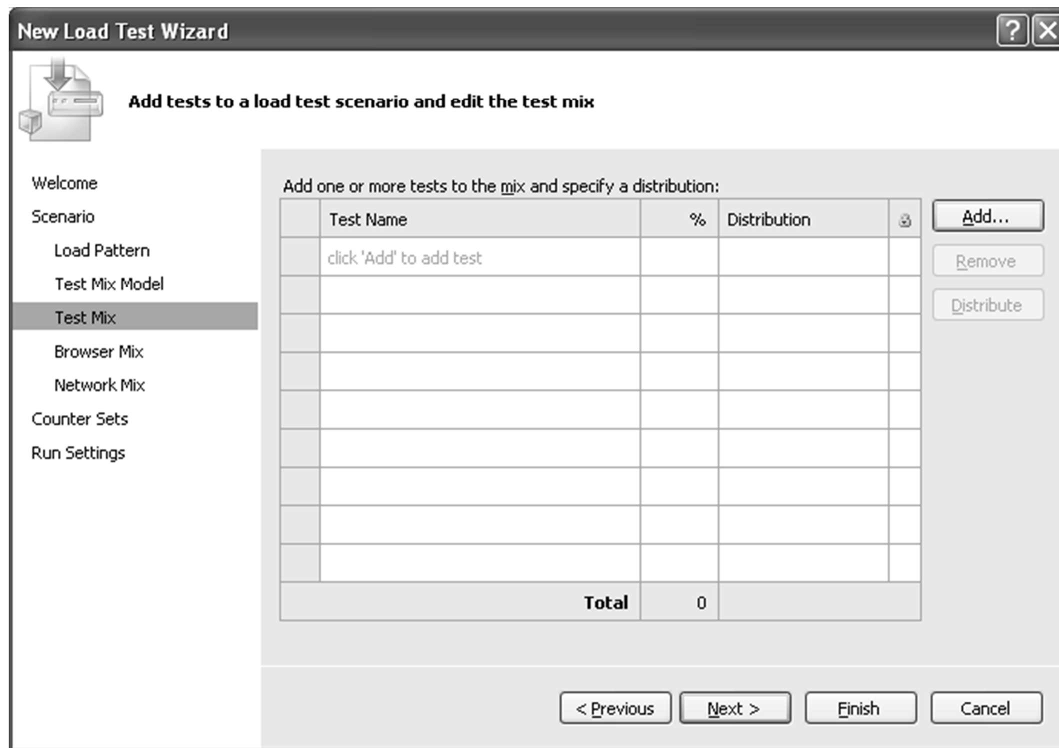
Seleccione el patrón para la prueba de carga.

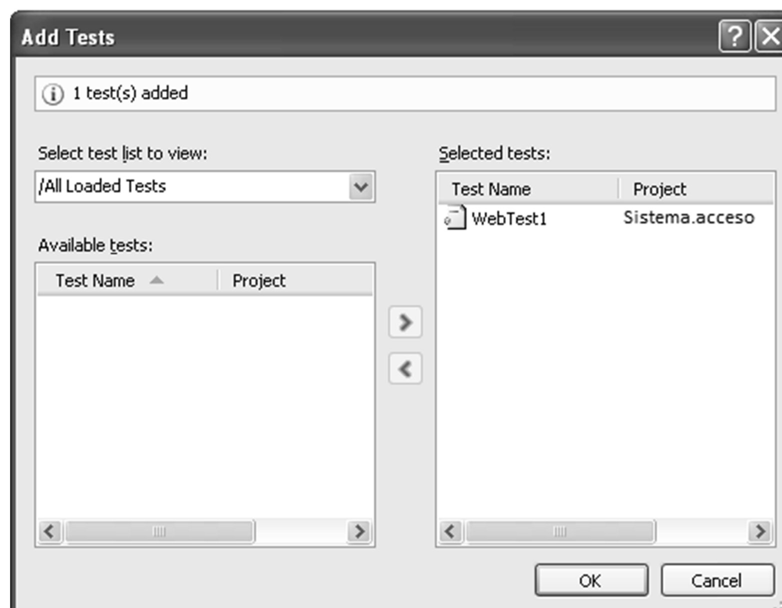


Seleccione las mezclas o combinaciones de su prueba.

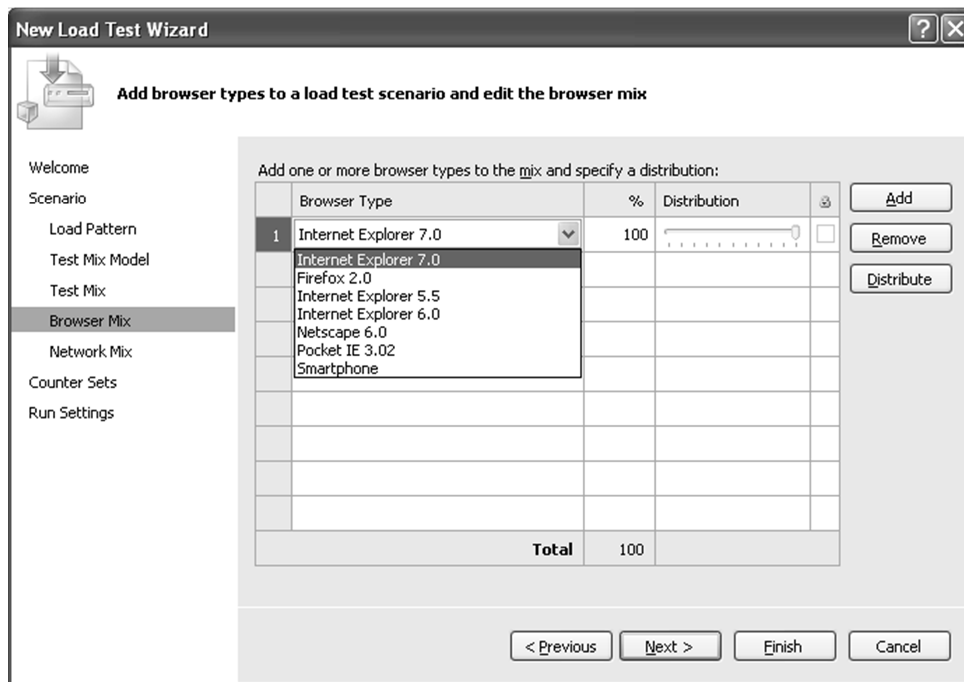


Agregue el *web test* a su *load test* para que simule la carga de usuario.

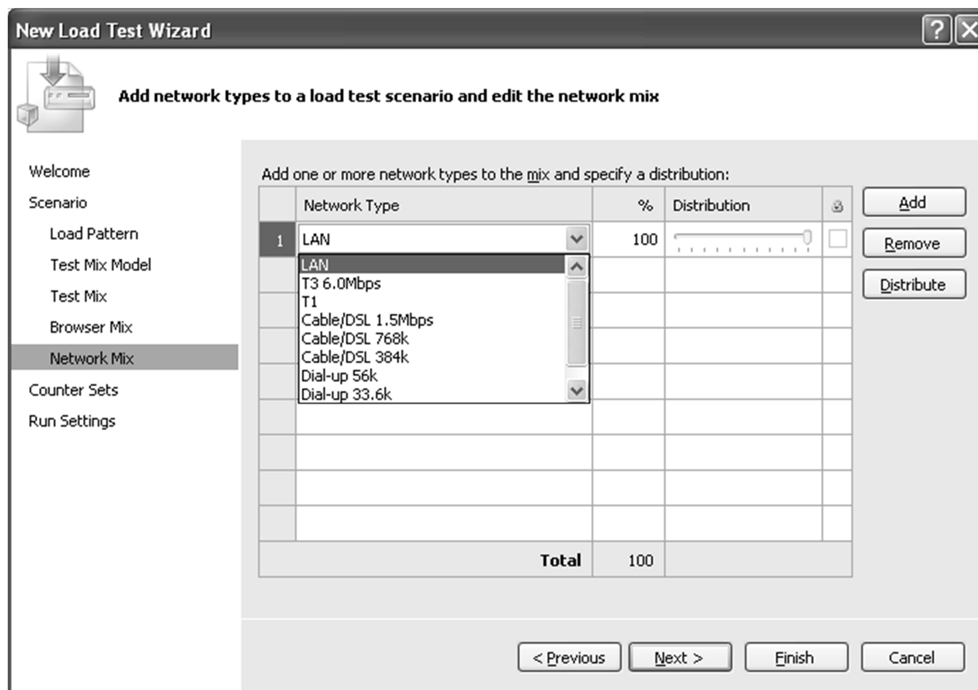




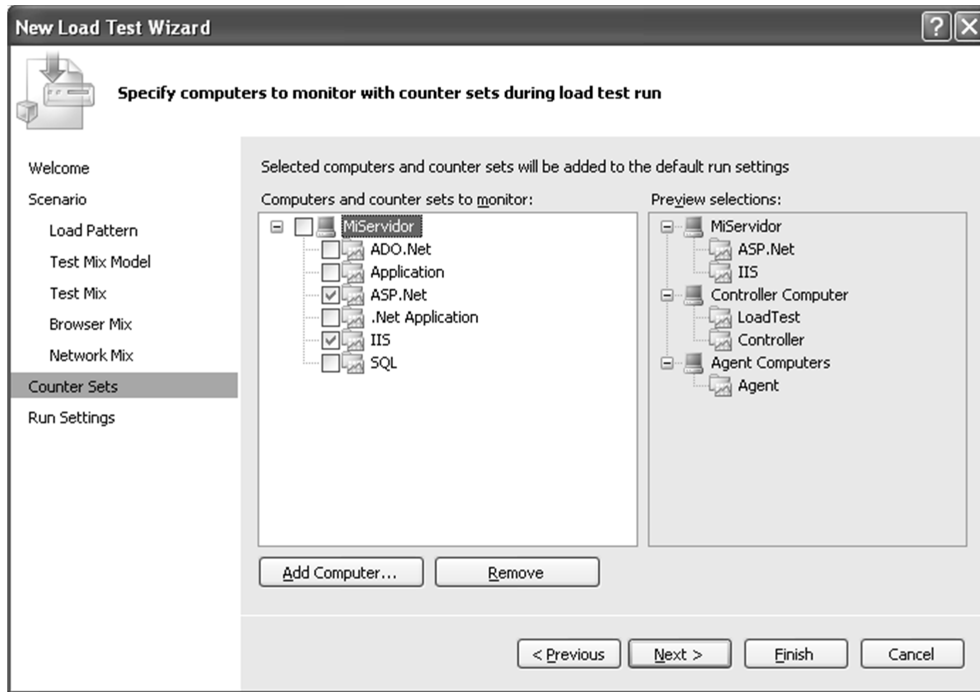
Seleccione el *browser* para su escenario (en ejemplos anteriores, se explicó la importancia de las pruebas automatizadas para diferentes ambientes).



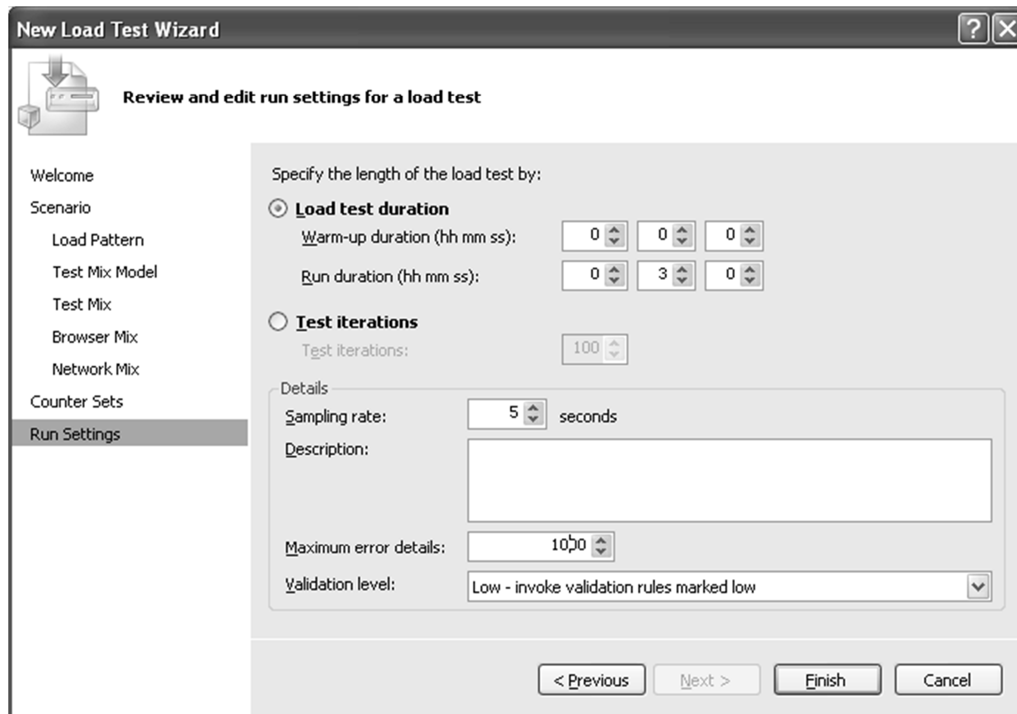
Seleccione el tipo de enlace que simulará su prueba de carga.



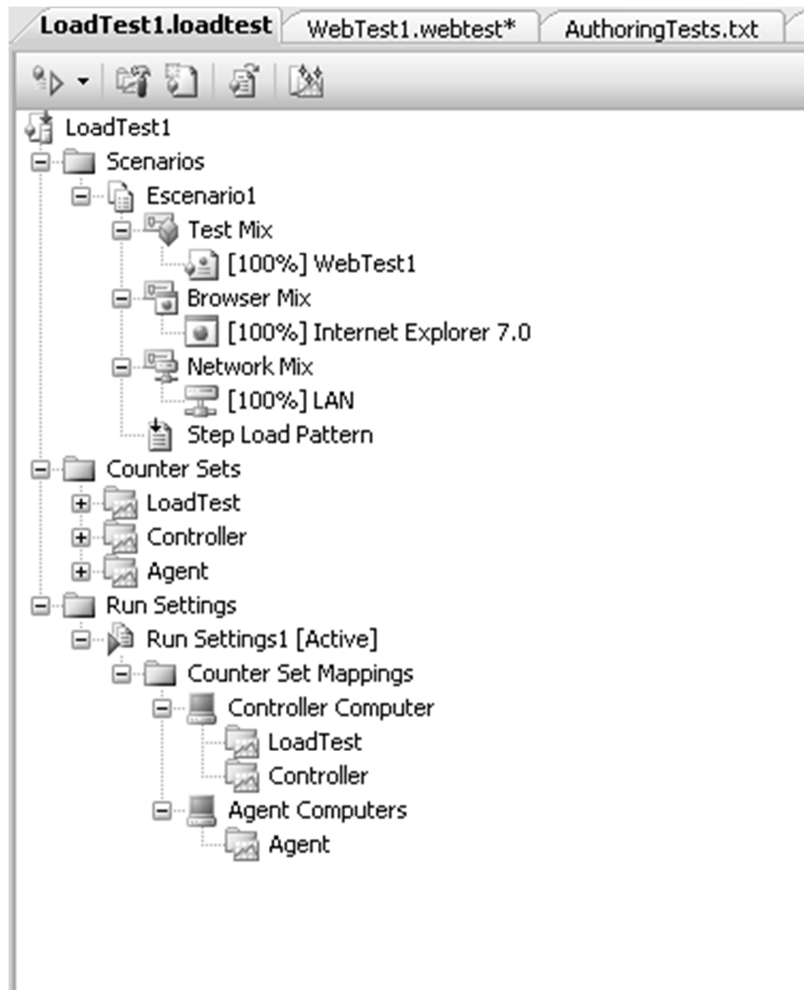
Especifique la computadora por monitorear.



Especifique la cantidad de iteraciones y usuarios que utilizará su prueba.

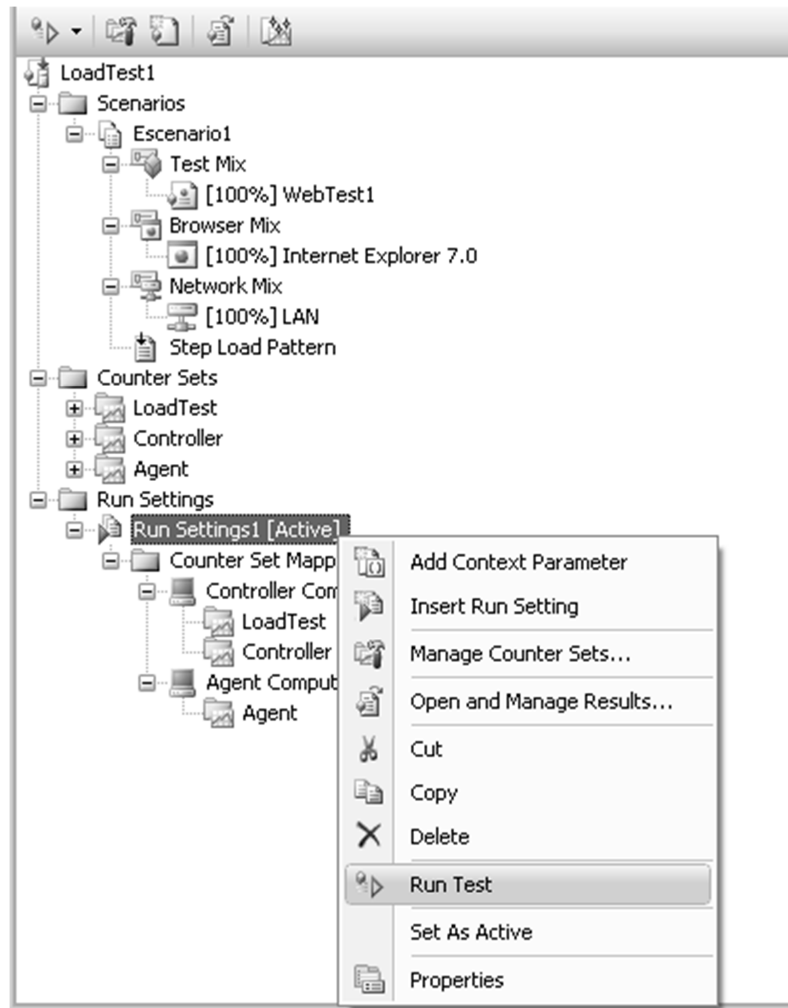


Una vez finalizado el *Wizard*, se generará una pantalla en la cual muestra el escenario que usted creó.

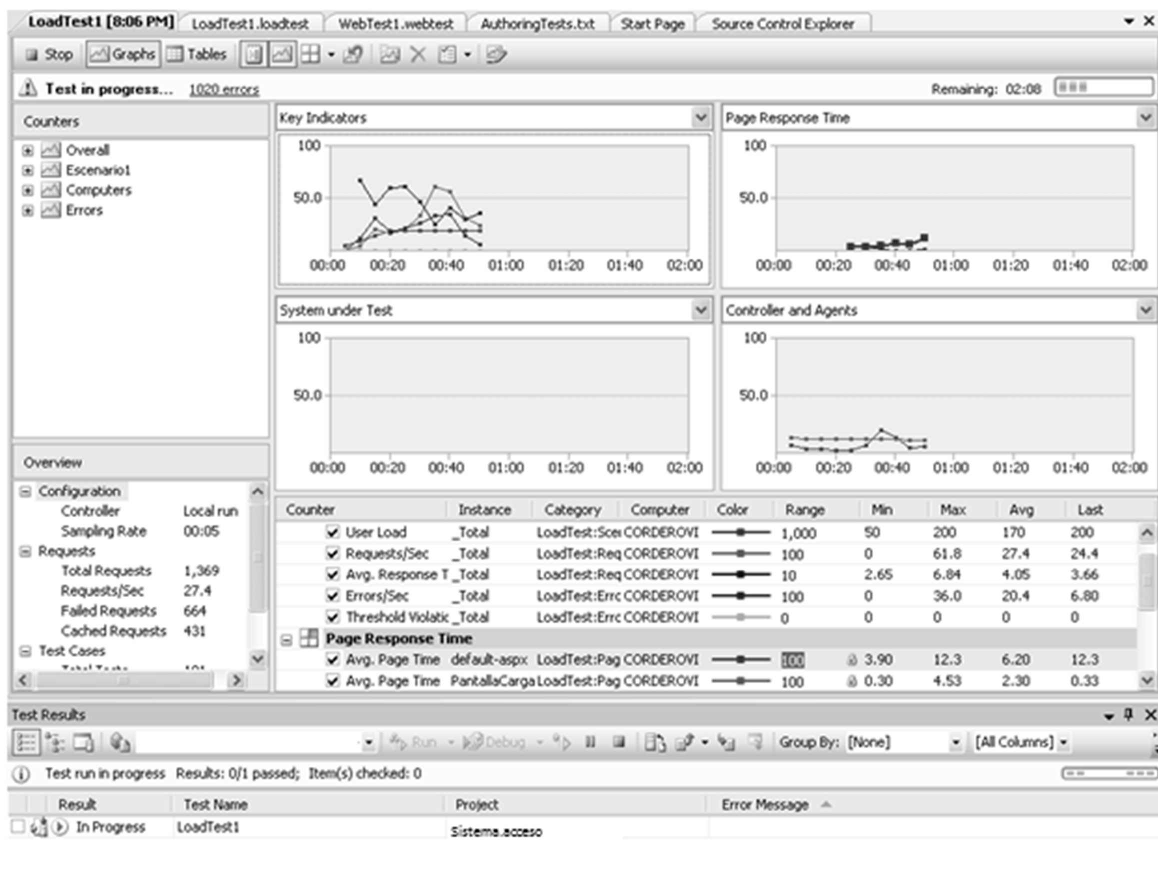


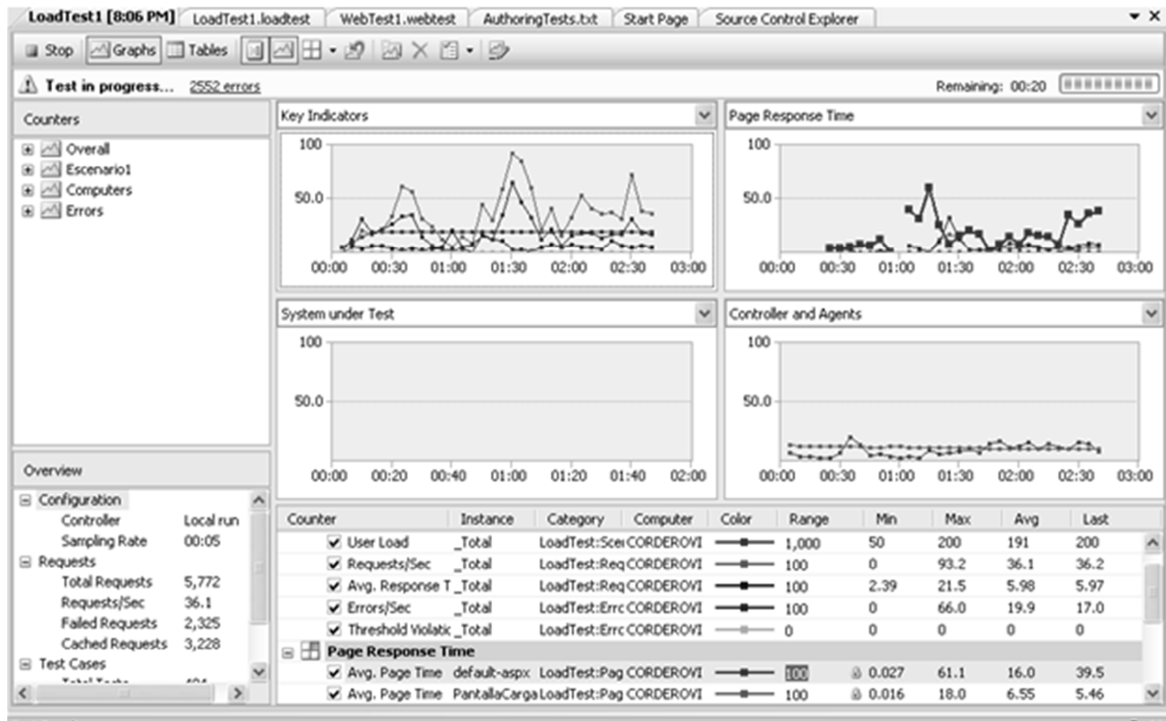
Ejecución del Load test

Para la ejecución del *load test,* lo cual dará inicio a la carga de su escenario, simplemente dé *click* derecho sobre "Run Settings" y seleccione "Run Test".



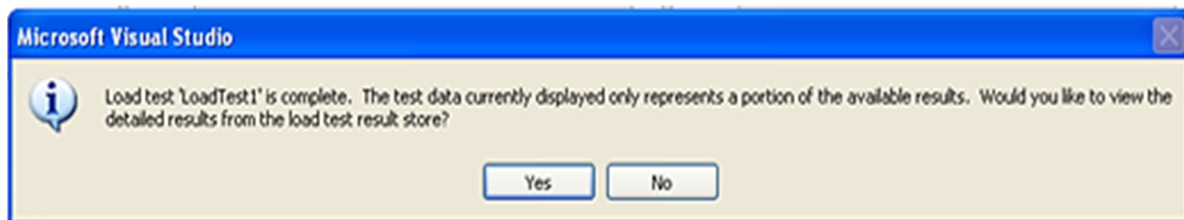
Luego de seleccionada la ejecución de la prueba de carga, aparecerán los gráficos correspondientes para el monitoreo de cada una de las páginas y, con la información necesaria, se irá validando el comportamiento de la prueba.





Finalización del Load test

La prueba se deja ejecutando, mientras usted revisa los errores y el comportamiento de los gráficos (la explicación de cómo se interpreta los gráficos no se detalla en este material). Al finalizar la prueba de carga, la herramienta le consultará de la siguiente manera, en la cual da la opción de generar un reporte o no.



Si se selecciona Yes, se generará un reporte como el que sigue; si se selecciona No, simplemente no se genera ningún reporte.

Load Test Summary

Test Run Information

Load test name	LoadTest1
Description	
Start time	12/3/2010 8:12:43 PM
End time	12/3/2010 8:15:43 PM
Warm-up duration	00:00:00
Duration	00:03:00
Controller	Local run
Number of agents	1
Run settings used	Run Settings1

Key Statistic: Top 5 Slowest Pages

URL (Link to More Details)	Avg. Page Time (sec)
http://Prueba/Sistema/ingreso.aspx	13.2
http://Prueba/Sistema/Cargando.aspx	4.94
http://Prueba/Sistema/SesionUsuario.aspx	4.16
http://Prueba/Sistema/Desconexion.aspx	0.31

Key Statistic: Top 5 Slowest Tests

Name	Avg. Test Time (sec)
WebTest1	29.1

Overall Results

Max User Load	200
Requests/Sec	38.4
Requests Failed	2,913
Requests Cached Percentage	41.7
Avg. Response Time (sec)	5.13
Avg. Content Length (bytes)	17,325
Tests/Sec	3.56
Tests Failed	616
Avg. Test Time (sec)	29.1
Avg. Transaction Time (sec)	0
Avg. Page Time (sec)	5.34

Test Results

Name	Scenario	Total Tests	Failed Tests (% of total)	Avg. Test Time (sec)
WebTest1	Escenario1	640	616 (96.3)	29.1

Lo anteriormente expuesto es el uso de una herramienta particular que, en su futuro, podría actualizarse con nuevas versiones y mejorar su funcionamiento o cambiar su interfaz (presentación al usuario). El objetivo final es mostrarle la existencia de herramientas que permiten la automatización de escenarios de prueba, tanto básicos como alternos, y que puede hacer uso de ellas para el mejoramiento de sus procesos de control de calidad.

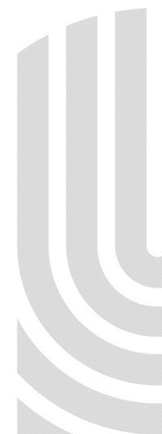
Ejercicios de autoevaluación

Tema 5. *Testing* de aplicaciones web

1. Suponga que se desarrolló una aplicación Windows, pero su ayuda, que antes era en un documento word, ahora se liberará por la Web en la Intranet de la organización. Esta ayuda funciona solamente como consulta y el mecanismo de seguridad es ingresar a la aplicación Windows. ¿Cuáles serían las pruebas mínimas que usted debería realizar para garantizar la calidad de este *software* de ayuda y por qué?

Tema 6. Pruebas automatizadas

2. En la herramienta para automatización de pruebas de Microsoft, descrita en este módulo, ¿cuál es la diferencia entre un *web test* y un *load test*?



RESPUESTAS A LOS EJERCICIOS DE AUTOEVALUACIÓN

Módulo I

Tema 1. Herramientas de *testing*

1. En un proceso de *testing*, los elementos básicos de entrada y salida son:

   - ✓ Elementos básicos de entrada: *datos de prueba, caso de prueba, plan de pruebas, modelo de diseño, guías para pruebas, configuración de ambiente de pruebas,* script *de pruebas y componente de pruebas.*
   - ✓ Elementos de salida: *logs o bitácora de pruebas, resultado o reporte de las pruebas, solicitud de cambios o mejoras, hallazgo de defectos y actualización de documentos de pruebas.*

2. Para el adecuado control de la calidad, IBM Rational ofrece múltiples herramientas; entre ellas, Rational CleaCase, Rational ClearQuest y Rational Test Manager: A continuación, se explica cada una.

   - ✓ Rational ClearCase: herramienta para la administración de la configuración del *software*. Esta administra el versionamiento del *software*, su código fuente y cualquier otro artefacto del desarrollo. Ofrece al ciclo de vida de *software* un manejo seguro y confiable de líneas base.

- ✓ Rational ClearQuest: herramienta para la administración de la gestión del cambio, seguimiento de defectos y nuevas solicitudes. Asimismo, ofrece al usuario la posibilidad de un *tracing* completo del estado de las solicitudes, defectos, etc., así como una gran variedad de reportes y gráficos totalmente personalizables.
- ✓ Rational TestManager: herramienta para el control de las pruebas, en la cual se desarrollan los planes y los casos de prueba, integra con los requerimientos del *software* y una gran variedad de reportes.

3. Dos herramientas *openSource* útiles para realizar pruebas de aceptación son FitNesse y Avignon.

Tema 2. Estrategias de *testing*

1. TDD es probar primero que desarrollar, o desarrollo guiado por pruebas, es una técnica en donde primero se realizan los *unit test*, se prueba el requerimiento y si falla se vuelve a ajustar; luego se *refactoriza* el código para ser utilizado en la aplicación.

   Su principal ventaja es que las pruebas se realizan desde el inicio del desarrollo en los pasos más pequeños de la implementación, así se van detectando los problemas tempranamente y se ajusta el *software* a los requerimientos.

2. Los test funcionales son pruebas dirigidas a los requerimientos funcionales del *software*, o sea, a lo que el *programa* debe hacer, los comportamientos establecidos en el flujo básico y los flujos alternos del documento de requerimientos.
3. Los test no funcionales son pruebas dirigidas a los requerimientos suplementarios establecidos. En otras palabras, no están ligados con el comportamiento habitual del *software* desde el punto de vista funcional, pero sí a su rendimiento, seguridad o control de acceso.

Caso. Desarrollo de un proceso de *testing*

A continuación se resuelve el caso, asimismo, se advierte que no es la única respuesta, ya que en una organización se deben tomar en cuenta las posibles variables. No obstante, esta solución representa una guía de lo que se espera en un proceso normal y básico de *testing*.

Identificación de elementos y eventos dinámicos del sistema

Analice y comprenda la información disponible: requerimientos funcionales y no funcionales, diagrama de clases, diagrama Entidad – Relación y diagrama de infraestructura. Converse con el desarrollador y analista en caso de dudas (puede que en esta etapa se produzcan modificaciones en la documentación, e ahí la importancia de que el *tester* participe en etapas tempranas del ciclo de vida de desarrollo).

Identificación de los límites e interfaces del sistema

Indague sobre interfaces, infraestructura de la aplicación, sistemas relacionados con el nuevo desarrollo.

Identificación de los elementos de la infraestructura de pruebas

Explore el ambiente de pruebas y garantice que es el óptimo para realizar la prueba.

Definir entradas del proceso

Desarrolle el plan de pruebas, los casos de prueba, la configuración del ambiente de pruebas, y defina los tipos de test por realizar (funcionales, de regresión, de rendimiento, de usabilidad, de integridad, etc.).

Utilización de la herramienta IBM Rational TestManager para la documentación de planes y casos de prueba.

Utilización de la herramienta IBM Rational ClearQuest para la gestión de cambios, control y seguimiento de los artefactos.

Verificar la utilidad y reutilización de los datos, *scripts* y componentes de pruebas.

Iniciar el proceso de pruebas

Las estrategias adecuadas se definen: realización de test funcional y no funcional.

Con base en la definición de los casos de prueba y los documentos obtenidos, se inicia el proceso de test.

En el IBM Rational Test Manager, se deben ir ejecutando los casos de prueba y queda evidencia del resultado del test.

En caso de un hallar un defecto o mejora, estos debe reportadarse en IBM Rational ClearQuest, y la solicitud seguirá el flujo implementado por la organización.

La ejecución de pruebas automatizadas con herramientas como *LINK Checker W3C* para la revisión de “enlaces muertos” de la Ayuda en línea. También, se debe tomar en cuenta implementar *web test* y *load test* por las características de la aplicación (en el módulo III se explica una herramienta).

Salidas del proceso de pruebas

En el proceso de testing, deben generarse los *logs* de pruebas, el resultado final, las solicitudes de cambio y defectos encontrados (si los hay), así como actualizar la documentación correspondiente si es necesario.

Módulo II

Tema 3. Tipos de pruebas

1. El proceso del ciclo de vida de pruebas según RUP, que se muestra en la figura 9, se explica así:

   - ✓ Es un ciclo reutilizable en cada iteración, y mejora cada vez que se utiliza.
   - ✓ Definir la evaluación: se determina el alcance de la iteración por ser probada, para delimitar el logro de las pruebas.

- ✓ Validar versión operacional: se refiere a cuando se hace una solicitud al área de pruebas para realizar los correspondientes test, esta área recibe un entregable, pero antes de continuar con el proceso, debe asegurarse de que esa versión entregada es estable, o sea, como mínimo, tiene la responsabilidad de validar que no se falle en las primeras pruebas de verificación.
- ✓ Lograr estabilidad: se refiere a que si el *software* entregable falla, este se reporta al desarrollador para que realice una nueva versión y vuelva a pasar por pruebas hasta que se logre estabilizar.
- ✓ Probar y evaluar: una vez que se tenga un *software* estable, éste ingresa formalmente a un proceso de pruebas, por lo que aquí ya se inician las pruebas aplicables al *software*, basadas en un plan de pruebas y la ejecución de los casos de prueba, ideas de prueba y aplicación de herramientas automatizadas. En este momento, se aplican los tipos de pruebas definidos para el particular, a saber: funcionales, de carga, de estrés, de seguridad, de instalación, de volumen, de regresión, de aceptación, etc. Luego de realizar la ejecución, se evalúan los resultados y los criterios de aceptación y, finalmente, se brindan los reportes del resultado, que pueden generar nuevas solicitudes de cambio.
- ✓ Mejorar elementos de prueba: cada vez que se realiza los procedimientos de *probar y evaluar* y *lograr estabilidad*, siempre existe la posibilidad de mejorar los elementos. En otras palabras, mejorar mis pruebas automatizadas, mis datos de prueba, mis *scripts* de prueba, mis casos de prueba o base de datos. En fin, es mejorar lo que usted utiliza para realizar sus pruebas.
- ✓ Otro ciclo: se refiere a si surgieron solicitudes de cambio o defectos, entonces, es necesario reportarlo al equipo de desarrollo. Este último resuelve la problemática, y se vuelve a ejecutar el proceso.
- ✓ Verificar proceso de prueba: se debe estar en una constante mejora, siempre verificando si es posible mejorar el proceso. Si se identifica otra técnica mejor, se implementa; a esto se refiere la decisión *otra técnica*.

Tema 4. Validación y verificación de *software*

1. En este caso, se refiere al concepto de validación, ya que se está demostrando el objetivo del *software* en un entorno de trabajo. Ejemplo: se valida el correcto funcionamiento cuando pasa a un ambiente de Pruebas.
2. En este caso, se refiere a verificación, ya que se está comprobando que el *software* cumple con los requerimientos solicitados por el usuario final o dueño del sistema.

Caso. Identifique los tipos de prueba necesarios

Para el caso dado, es necesario efectuar, al menos, los siguientes tipos de prueba:

Pruebas de función o funcionalidad, validan el nuevo desarrollo y la mejora ya existente.

Pruebas de seguridad, son necesarias, ya que el nuevo desarrollo posee roles, y se debe garantizar que los perfiles estén bien diseñados y que, solamente, se acceda a la información pertinente.

Pruebas de usabilidad, garantizan el nuevo aplicativo y la mejora; cumplen con los estándares de usabilidad brindados por la organización.

Pruebas de integridad, validan el nuevo desarrollo *convive* (no se presentan conflictos) con el resto de componentes del sistema.

Pruebas de estrés, son esenciales, pues el sistema será utilizado por usuarios del mundo entero. Hay que tomar en cuenta que este tipo de pruebas se usan más dependiendo de la criticidad de la aplicación.

Pruebas de concurrencia, son fundamentales, ya que por la característica de la aplicación, la demanda de acceso de usuarios será mucha y al mismo tiempo.

Pruebas de configuración, estos test son básicos puesto que la aplicación es *web* y, al menos, su funcionamiento debe ser garantizado en dos de los siguientes navegadores: Firefox e Internet Explorer.

Pruebas de instalación, se ejecutan para poner en marcha el producto.

Pruebas de regresión, test sumamente relevante para garantizar que la mejora no haya dañado el correcto funcionamiento del desarrollo ya existente.

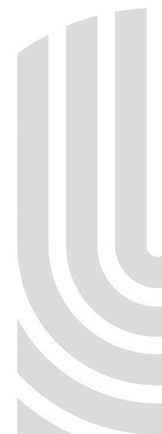
Módulo III

Tema 5. *Testing* de aplicaciones web

1. Las pruebas mínimas que usted debe realizar para garantizar la calidad de este *software* de ayuda son:
    - ✓ Pruebas de enlace para que no existan enlaces "rotos".
    - ✓ Pruebas de navegación para garantizar la facilidad de navegación del sitio.
    - ✓ Pruebas de contenido que garanticen la calidad del texto.
    - ✓ Pruebas de ayuda para asegurarse de que posea las características típicas para lo cual fue creado.
    - ✓ Pruebas de opción de impresión para garantizar su correcto funcionamiento en caso de que el usuario desee utilizar la funcionalidad.

Tema 6. Pruebas automatizadas

2. En la herramienta para automatización de pruebas de Microsoft, es la siguiente:
    - ✓ Un *web test (wt)* es una prueba simple del comportamiento de una aplicación web, su principal característica es que graba los pasos que realiza el usuario y posteriormente los datos variables pueden ser sustituidos por un pool de datos. Cuando se ejecuta el *web test*, se ejecuta solamente una vez y actúa según como el usuario lo grabó, sin que se realice ninguna intervención una vez que inicie la ejecución. El mantenimiento es relativamente sencillo.
    - ✓ El *load test* es una prueba de carga, con mantenimiento alto, las pruebas se pueden generar utilizando código programable, y en relación con el *web test*, el *load test* puede utilizar uno o varios *wt* para simular escenarios.



REFERENCIAS

Aldon. (2011). Aldon. Recuperado el 7 de enero de 2011, de www.aldon.com

Aston, P., & Fitzgerald, C. (2010). The Grinder, a Java Load Testing Framework. Recuperado el 17 de setiembre de 2010, de http://grinder.sourceforge.net/

Avignon. (2011). Home page Avignon. Recuperado el 25 de octubre de 2010, de http://www.nolacom.com/avignon/index.asp

Babalievsky, F. (2009). Link Evaluator. Recuperado el 2010 de diciembre de 2010, de https://addons.mozilla.org/es-ES/firefox/addon/4094

Calkini, I. T. (2011). Verificación de software. Recuperado el 10 de febrero de 2011, de http://www.itescam.edu.mx/principal/sylabus/fpdb/recursos/r56662.PDF

Checker, W. L. (1999 – 2010). W3C Link Checker. Recuperado el 15 de diciembre de 2010, de http://validator.w3.org/checklink

Crispin, L. (2009). Coding and Testing: Testers and Programmers Working Together. Recuperado el 20 de enero de 2011, de http://www.methodsandtools.com/archive/archive.php?id=88

DRK. (2002 – 2011). DRK Open Source Software tools. Recuperado el 16 de diciembre de 2010, de http://www.drk.com.ar/index.php

FindBugs. (2009). Find Bugs in Java Programs. Recuperado el 14 de octubre de 2010, de http://findbugs.sourceforge.net/

FitNesse. (2011). The fully integrated standalone wiki, and acceptance testing framework. Recuperado el 25 de marzo de 2011, de http://fitnesse.org/

Foundation, T. A. (1999 - 2011). The Apache Jakarta Project. Recuperado el 16 de julio de 2010, de http://jakarta.apache.org/

Fowler, M. (2011). Refactoring. Recuperado el 17 de enero de 2011, de www.refactoring.com

Garita, G. (2010). Material Digital para el curso Sistemas de Calidad. Costos de la calidad del software.

Gold, R. (2000 – 2008). HttpUnit. Recuperado el 20 de agosto de 2010, de http://httpunit.sourceforge.net/index.html

IEEE. (2010). Pruebas según el estándar IEEE std. 829. Recuperado el 18 de noviembre de 2010, de http://members.fortunecity.es/jorgechip/IEEE829.doc

Inc, F. S. (2010). RATS. Recuperado el 25 de octubre de 2010, de http://www.fortify.com/security-resources/rats.jsp

Inc., G. (2011). PhpUnit. Recuperado el 21 de octubre de 2010, de http://www.phpunit.de/

InfoEther. (2002 - 2009). SourceForge.net. Recuperado el 25 de octubre de 2010, de http://pmd.sourceforge.net/

JUnit. (2011). Resources for Test Driven Development. Recuperado el 14 de octubre de 2010, de http://www.junit.org/

Ltd., R. S. (2011). WebLOAD Professional vs WebLOAD Open Source Load Generation Engine. Recuperado el 16 de setiembre de 2010, de http://www.webload.org

Ltd., T. S. (2011). Is your testing keeping pace with development? Recuperado el 15 de julio de 2010, de http://sahi.co.in/w/

Luzuriaga, J. M. (s.f.). Monografias.com. Recuperado el 20 de octubre de 2010, de Inspecciones de software: http://www.monografias.com/trabajos6/isof/isof.shtml

Monografias. (s.f.). Inspección de software. Recuperado el 22 de noviembre de 2010, de http://www.monografias.com/trabajos6/isof/isof.shtml#refe

OpenSta. (2007). OpenSTA User Home. Recuperado el 22 de julio de 2010, de http://www.opensta.org/

PHPLint. (2011). Home page PHPLint. Recuperado el 25 de octubre de 2010, de http://www.icosaedro.it/phplint/

Prado, E. R. (s.f.). Casi todas las pruebas de software. Recuperado el 10 de octubre de 2010, de www.sistedes.es/TJISBD/Vol-1/No-4/.../pris-07-raja-ctps.pdf

Rational. (2010). Rational Unified Process (RUP). Recuperado el 22 de setiembre de 2010, de http://www-01.ibm.com/software/awdtools/rup/

Rational. (2011). IBM® Rational® ClearCase® offers complete software configuration management. Recuperado el 10 de febrero de 2011, de http://www-01.ibm.com/software/awdtools/clearcase/index.html

Rational. (2011). IBM® Rational® ClearQuest® offers comprehensive software change management. Recuperado el 10 de febrero de 2011, de http://www-01.ibm.com/software/awdtools/clearquest/

Rational. (2011). Solving the complexity of test and test asset management. Recuperado el 12 de febrero de 2011, de http://www-01.ibm.com/software/awdtools/test/manager/

S., S., & Kumar, N. S. (2008). Software Testing with Visual Studio Team System 2008.

Scovetta, M. V. (2007). Yasca. Recuperado el 1.° de febrero de 2011, de http://www.yasca.org/

SEI. (2010). Validación. Recuperado el 18 de noviembre de 2010, de http://www.sei.cmu.edu/cmmi/tools/cmmiv1-3/upload/DEV-VAL-compare.pdf

SEI. (2010). Verificación. Recuperado el 18 de noviembre de 2010, de http://www.sei.cmu.edu/cmmi/tools/cmmiv1-3/upload/DEV-VER-compare.pdf

SeleniumHQ. (2011). Web application testing system. Recuperado el 20 de agosto de 2010, de http://seleniumhq.org/projects/ide

Sleuth, X. L. (2011). Find broken links on web sites. Recuperado el 15 de noviembre de 2010, de http://home.snafu.de/tilman/xenulink.html

Software, B. (2010). Badboy Software. Recuperado el 21 de agosto de 2010, de http://www.badboy.com.au/

Test, P. (2010). Web test. Recuperado el 1.° de diciembre de 2010, de http://puretest.blogspot.com/2009/11/checklist-for-website-testing-step-1.html

Test, S. (2010). Unit Testing for PHP. Recuperado el 15 de noviembre de 2010, de http://www.simpletest.org/

Vautier, E. (2009). Testdriven. Recuperado el 2 de febrero de 2011, de http://www.testdriven.com/tag/articles_post_tag/

Walther, S. (2009). Test-After Development is not Test-Driven Development. Recuperado el 20 de enero de 2011, de http://stephenwalther.com/blog/archive/2009/04/08/test-after-development-is-not-test-driven-development.aspx