

Professional Testing Master

Centro de e-Learning SCEU UTN - BA.

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

www.sceu.frba.utn.edu.ar/e-learning

Unidad 5: Otros conceptos de testing



Presentación:

En esta Quinta Unidad del curso, completaremos el conocimiento sobre testing con conceptos, metodologías y técnicas complementarios de gran relevancia en el mercado actual así como herramientas que facilitan la tarea del tester.



Objetivos:

Al terminar la Unidad los participantes:

Se habrán familiarizado con los diferentes tipos de pruebas que se realizan en distintos momentos del desarrollo.

Se habrán introducido en los conceptos fundamentales de planificación de test.

Conocerán herramientas que ayudan con la tarea de test, incluyendo automatización de pruebas.

Habrán aprendido sobre nuevas metodologías de desarrollo y testing de software.

Conocerán las características de las certificaciones de testing más importantes a nivel internacional.



Bloques temáticos:

1. Pruebas: regresión, humo, alfa, beta, usabilidad
2. Planificación de test
3. Automatización y herramientas
4. Test Driven Development y Programación Extrema
5. Certificaciones

Contenido

Unidad 5: Otros conceptos de testing	2
Presentación:	3
Objetivos:	4
Al terminar la Unidad los participantes:	4
Bloques temáticos:.....	5
Contenido.....	6
Consignas para el aprendizaje colaborativo	8
Tomen nota.....	9
1 Prueba de regresión.....	10
2 Prueba de humo.....	11
3 Pruebas alfa y beta.....	12
4 Prueba de usabilidad.....	13
5 Test Plans	15
6 Herramientas y automatización.....	17
6.1 Herramientas para Testing.....	18
6.1.1 Herramientas para planificación y administración de pruebas.....	18
6.1.2 Herramientas para diseño de casos de prueba	19
6.1.3 Monitores y visores.....	19
6.1.4 Drivers	22
6.1.5 Stubs.....	22
6.1.6 Herramientas de carga y stress.....	23
6.2 Herramientas para automatización	25
6.2.1 Grabación y reproducción de macros.....	25
6.2.2 Macros programables	27
6.2.3 Herramientas de testing totalmente programables.....	29
6.2.4 Herramientas de testing aleatorio	31
7 Test Driven Development	31
7.1 Introducción	31
7.2 TDD y Programación Extrema (XP).....	33

7.2.1	Conceptos de Testing Extremo	34
8	Certificaciones para testing	37
8.1	International Institute of Software Testing®	37
8.1.1	Certificaciones.....	37
8.2	International Software Testing Qualifications Board®	44
8.2.1	Niveles de certificación	44
	Bibliografía utilizada y sugerida	47



Consignas para el aprendizaje colaborativo

En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

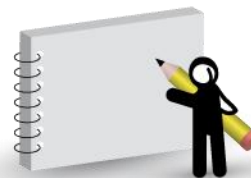
- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.

** El MEC es el modelo de E-learning colaborativo de nuestro Centro.*



Tomen nota

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.

1 Prueba de regresión

Recuerde la prueba de integración de la Unidad 1. Cada vez que se agrega un nuevo módulo como parte de las pruebas de integración, el software cambia. Se establecen nuevas rutas de flujo de datos, ocurren nuevas operaciones de entrada/salida y se invoca nueva lógica de control. Dichos cambios pueden causar problemas con las funciones que anteriormente funcionaban sin fallas. En el contexto de una estrategia de prueba de integración, la *prueba de regresión* es la nueva ejecución de algún subconjunto de pruebas que ya se realizaron a fin de asegurar que los cambios no propagaron efectos colaterales no deseados.

La *prueba de regresión* es una importante estrategia para reducir “efectos colaterales”. Ejecute pruebas de regresión cada vez que se realiza un cambio importante al software (incluida la integración de nuevos componentes).

En un contexto más amplio, las pruebas exitosas (de cualquier tipo) dan como resultado el descubrimiento de errores, y los errores deben corregirse. Siempre que se corrige el software, cambia algún aspecto de la configuración del software (el programa, su documentación o los datos que sustenta). Las pruebas de regresión ayudan a garantizar que los cambios (debidos a pruebas o por otras razones) no introducen comportamiento no planeado o errores adicionales.

Las pruebas de regresión se pueden realizar manualmente, al volver a ejecutar un subconjunto de todos los casos de prueba o usando herramientas de captura/reproducción automatizadas. Las *herramientas de captura/reproducción* (sección 6.2.1) permiten al tester capturar casos de prueba y resultados para una posterior reproducción y comparación. La *suite de prueba de regresión* (el subconjunto de pruebas que se va a ejecutar) contiene tres clases diferentes de casos de prueba:

- Una muestra representativa de pruebas que verificará todas las funciones de software.
- Pruebas adicionales que se enfocan en las funciones del software que probablemente resulten afectadas por el cambio.
- Pruebas que se enfocan en los componentes del software que cambiaron.

A medida que avanza la prueba de integración, el número de pruebas de regresión puede volverse muy grande. Por tanto, la suite de pruebas de regresión debe diseñarse para incluir solamente aquellas que aborden una o más clases de errores en cada una de las

funciones del programa. Es impráctico e ineficiente volver a ejecutar toda prueba para cada función del programa cada vez que ocurre un cambio.

2 Prueba de humo

La *prueba de humo* (smoke test) es un enfoque de prueba de integración que se usa cuando se desarrolla software de producto. Se diseña como un mecanismo de ritmo para proyectos críticos en el tiempo, lo que permite al equipo del software evaluar el proyecto de manera frecuente. La prueba de humo puede caracterizarse como una estrategia de integración continua. El software se reconstruye (con el agregado de nuevos componentes) y se prueba todos los días.

En esencia, el enfoque de prueba de humo abarca las siguientes actividades:

1. Los componentes de software traducidos en código se integran en una *construcción (build)*. Un build incluye todos los archivos de datos, bibliotecas, módulos reutilizables y componentes que se requieren para implementar una o más funciones del producto.
2. Se diseña una serie de pruebas para exponer los errores que impedirán al build realizar adecuadamente su función. La intención debe ser descubrir errores “paralizantes” que tengan la mayor probabilidad de retrasar el proyecto.
3. El build se integra con otros, y todo el producto (en su forma actual) se somete a prueba de humo diariamente. El enfoque de integración puede ser descendente o ascendente.

La frecuencia diaria de las pruebas de todo el producto puede sorprender a algunos lectores. Sin embargo, las pruebas constantes brindan, tanto a gerentes como a profesionales, una evaluación realista del progreso de la prueba de integración.

La prueba de humo debe verificar todo el sistema de extremo a extremo. No tiene que ser exhaustiva, pero debe poder exponer los problemas principales. La prueba de humo debe ser suficientemente profunda para que, si el build pasa, pueda suponer que es suficientemente estable para probarse con mayor profundidad.

La prueba de humo proporciona algunos beneficios cuando se aplica sobre proyectos de software complejos y críticos en el tiempo:

- *Se minimiza el riesgo de integración.* Puesto que las pruebas de humo se realizan diariamente, las incompatibilidades y otros errores paralizantes pueden descubrirse tempranamente, lo que reduce la probabilidad de impacto severo sobre el cronograma cuando se descubren errores.
- *La calidad del producto final mejora.* Es probable que la prueba de humo descubra errores funcionales así como errores de diseño de arquitectura o en el componente debido a que el enfoque está orientado a la construcción (integración). Si tales errores se corrigen temprano, se tendrá una mejor calidad del producto.
- *El diagnóstico y la corrección de errores se simplifican.* Como todo enfoque de prueba de integración, es probable que los errores descubiertos durante la prueba de humo se asocien con “nuevos incrementos de software”; es decir, el software que se acaba de agregar a los builds es causa probable de un error recientemente descubierto.
- *El progreso es más fácil de evaluar.* Con cada día que transcurre, más software se integra y se muestra que funciona. Esto incrementa la moral del equipo y brinda a los gerentes un buen indicio de que se está progresando.

3 Pruebas alfa y beta

Es imposible que un desarrollador de software prevea cómo usará el cliente realmente un programa. Las instrucciones para usarlo pueden malinterpretarse; regularmente pueden usarse combinaciones extrañas de datos; la salida que parecía clara a quien realizó la prueba puede ser confusa para un usuario.

Cuando se construye software a medida para un cliente, se realiza una serie de pruebas de aceptación a fin de permitir al cliente validar todos los requerimientos. Realizada por el usuario final en lugar de por los ingenieros de software, una prueba de aceptación puede variar desde una “prueba de manejo” informal hasta una serie de pruebas planificadas y ejecutadas sistemáticamente. De hecho, la prueba de aceptación puede realizarse durante un periodo de semanas o meses, y mediante ella descubrir errores.

Si el software se desarrolla como un producto que va a ser usado por muchos clientes, no es práctico realizar pruebas de aceptación formales con cada uno de ellos. La mayoría de los constructores de productos de software usan un proceso llamado prueba alfa y prueba beta para descubrir errores que al parecer sólo el usuario final es capaz de encontrar.

Las pruebas *alfa* y *beta* se pueden considerar como parte de las pruebas de validación tratadas en la Unidad 1.

¿Cuál es la diferencia entre una prueba alfa y una prueba beta?

La *prueba alfa* se lleva a cabo en las oficinas del desarrollador por un grupo representativo de usuarios finales. El software se usa en un escenario natural con el desarrollador “mirando sobre el hombro” de los usuarios y registrando los errores y problemas de uso. Es un ambiente controlado.

La *prueba beta* se realiza en uno o más sitios del usuario final. A diferencia de la prueba alfa, por lo general el desarrollador no está presente. Por tanto, la prueba beta es una aplicación “en vivo” del software en un ambiente que no puede controlar el desarrollador. El cliente registra todos los problemas (reales o imaginarios) que se encuentran durante la prueba beta y los reporta al desarrollador periódicamente. Como resultado de los problemas reportados durante las pruebas beta, es posible hacer modificaciones y luego preparar el lanzamiento (release) del producto de software a toda la base de clientes.

En ocasiones se realiza una variación de la prueba beta, llamada *prueba de aceptación del cliente*, cuando el software se entrega a un cliente bajo contrato. El cliente realiza una serie de pruebas específicas con la intención de descubrir errores antes de aceptar el software del desarrollador. En algunos casos (por ejemplo, un gran sistema corporativo o gubernamental) la prueba de aceptación puede ser muy formal y abarcar muchos días o incluso semanas de prueba.

4 Prueba de usabilidad

Otra categoría importante de casos de prueba de sistema es un intento de encontrar problemas de factor humano, o usabilidad. Hace 30 años, la industria de la computación prestaba muy poca atención al estudio y la definición de buenas prácticas considerando el factor humano en los sistemas.

Los sistemas de software de hoy en día, sobre todo aquellos diseñados para un mercado masivo, se someten a extensos estudios de factor humano, y los programas modernos,

por supuesto, se benefician de los miles de programas y sistemas que los han precedido. Sin embargo, un análisis de los factores humanos sigue siendo una cuestión muy subjetiva. La siguiente es una lista de consideraciones que pueden ser evaluadas:

1. ¿Se ha adaptado cada interfaz de usuario a la inteligencia, la formación académica, y las presiones ambientales del usuario final?
2. ¿Los resultados del programa son significativos, respetuosos, y carentes de léxico técnico informático?
3. ¿Los diagnósticos de error, tales como mensajes de error son sencillos, o el usuario necesita un doctorado en ciencias de la computación para comprenderlos? Por ejemplo, ¿el programa genera mensajes como "IEK022A OPEN ERROR ON FILE 'SYSIN' ABEND CODE=102?". Mensajes como este no eran tan fuera de lo común en los años 70 y 80. Los sistemas para el mercado masivo de hoy en día mejoraron bastante, pero todavía se ven mensajes de error como "Ha ocurrido un error desconocido" o "El programa ha realizado una operación no permitida y debe cerrarse". A modo de ejemplo reciente, en la Figura 4-1 se muestra un mensaje de error real, el cual fue experimentado por el autor en 2021, al intentar realizar una operación en el sitio de online banking de uno de los principales bancos de Argentina. Los programas no deberían estar plagados de mensajes de error inútiles de este tipo. Incluso si usted no participa del desarrollo y se ocupa del testing, puede presionar para que se realicen mejoras con respecto a la interfaz humana.
4. ¿La totalidad de las interfaces de usuario muestran integridad conceptual, consistencia, uniformidad de sintaxis, convenciones, semántica, formato, estilos y abreviaturas?
5. Donde la precisión es vital, como por ejemplo un sistema de online banking, ¿hay suficiente redundancia en la entrada? Por ejemplo, para realizar una transferencia el sistema debería solicitar el nro. de cuenta del destinatario, pero también el nro. de documento y/o nombre para evitar transferir a un destinatario equivocado por un error en el ingreso de algún dato.
6. ¿El sistema contiene una cantidad excesiva de opciones, u opciones que resulta improbable que se usen? La tendencia en el software moderno es a mostrar solamente las opciones de menú más probablemente usables, basándose en consideraciones de testing y de diseño. Por lo tanto un programa bien diseñado puede aprender del usuario y empezar a presentar solamente las opciones del menú más frecuentemente usadas. Sin embargo, aun con menús inteligentes, los programas se deben diseñar para que las opciones se puedan acceder de una forma lógica e intuitiva.
7. ¿El sistema devuelve feedback instantáneo para todas las entradas? Por ejemplo donde la entrada es un clic del mouse, el ítem elegido puede cambiar de color o un botón se puede mostrar como presionado. Si se espera que el usuario elija un valor de una lista, el valor elegido se debe mostrar en la pantalla al momento de realizar

la selección. Más aún, si la opción elegida implica una operación que requiere tiempo de procesamiento (como es habitual en el software que accede sistemas remotos), se debe mostrar un mensaje informando al usuario lo que sucede.

8. ¿El programa es fácil de usar? Por ejemplo ¿el programa es case-sensitive (diferencia mayúsculas/minúsculas) sin dejar claro esto al usuario? Además, si el programa requiere navegar por una serie de menús u opciones ¿queda claro cómo volver al menú principal? ¿es fácil moverse un nivel arriba o abajo?

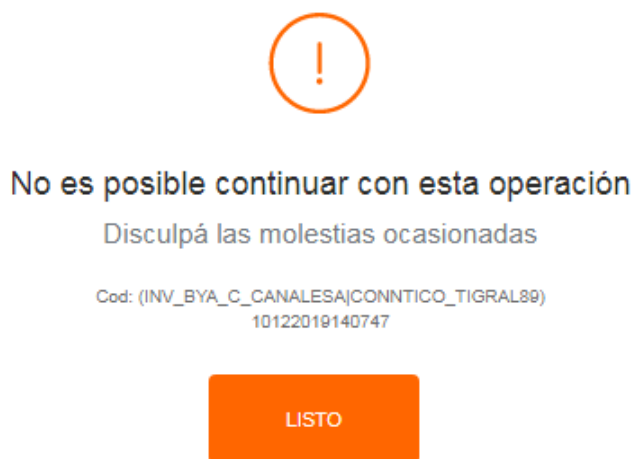


Figura 4-1. Mensaje de error de un sitio de Online Banking de Argentina en 2021

5 Test Plans

Como se mencionó en la Unidad 1, el error más frecuente en la planificación del test, es la suposición de que no se encontrarán errores. El resultado más obvio de esto es la subestimación de los recursos requeridos (personal, plazos, hardware, etc.).

Se podrían dedicar libros completos a la planificación de tests, pero se intentarán resumir los componentes más importantes que debe tener un buen plan:

1. **Objetivos.** Deben estar definidos los objetivos de cada fase de testing.
2. **Criterios de finalización.** Los criterios deben diseñarse para indicar cuándo será considerada completa cada fase de prueba. Este tema se discute en [Mye12].
3. **Cronogramas.** Se necesitan para cada fase. Deben indicar cuando se diseñan, escriben, y ejecutan los casos de prueba. Algunas metodologías de software como

Extreme Programming y Test Driven Development (discutidas más adelante) exigen que el diseño de los casos de prueba se realice antes de que comience la codificación de la aplicación.

4. **Responsabilidades.** Para cada fase, las personas que van a diseñar, escribir, ejecutar y controlar los casos de prueba, y la gente que se compromete a corregir los errores descubiertos, deben ser identificados. Dado que en los grandes proyectos por desgracia surgen disputas sobre si determinados resultados de las pruebas representan errores, se debe definir el árbitro.
5. **Bibliotecas y estándares de casos de prueba.** En un proyecto grande, son necesarios métodos sistemáticos para identificar, escribir y almacenar casos de prueba.
6. **Herramientas.** Las herramientas de prueba requeridas (sección 6) deben ser identificadas, incluyendo un plan para definir quién las va a desarrollar o adquirir, cómo se van a utilizar y cuándo.
7. **Tiempo de hardware.** Este es un plan para la cantidad de tiempo de computadora necesario para cada fase de prueba. Esto incluye los servidores utilizados para compilar aplicaciones, si son necesarios; las máquinas de escritorio requeridas para las pruebas de instalación; servidores Web para aplicaciones basadas en Web; dispositivos de red, si hacen falta; dispositivos móviles, etc.
8. **Configuración de hardware.** Si se necesitan configuraciones especiales de hardware o dispositivos, se requiere un plan que describa los requerimientos, cómo se cumplirán, y cuando se necesitan.
9. **Integración.** Parte del plan de pruebas es una definición de cómo se armará el sistema en conjunto (por ejemplo, integración incremental de arriba hacia abajo – Unidad 1–). Un sistema que contiene subsistemas o programas puede ser integrado gradualmente, utilizando el enfoque descendente o ascendente, por ejemplo, pero donde los componentes básicos son los programas o subsistemas, en lugar de módulos. Si este es el caso, es necesario un plan de integración de sistemas. El plan de integración del sistema define el orden de integración, la capacidad funcional de cada versión del sistema, y las responsabilidades para la producción de código que simula la función de los componentes que no existen.
10. **Procedimientos de seguimiento.** Se deben definir los medios para seguir diversos aspectos del progreso de las pruebas, incluida la ubicación de los módulos propensos a errores y la estimación de los avances respecto a la planificación, los recursos y criterios de finalización.
11. **Procedimientos de depuración.** Se deben definir los mecanismos para reportar los errores detectados, realizar el seguimiento de los progresos en las correcciones, y la aplicación de las correcciones al sistema. Los cronogramas (schedules), las responsabilidades, las herramientas y el tiempo de computadora o recursos también deben formar parte del plan de depuración.

12. Pruebas de regresión. Las pruebas de regresión (sección 1) se llevan a cabo después de hacer una mejora o corrección funcional en el programa. Las pruebas de regresión son importantes ya que los cambios y correcciones de errores tienden a ser mucho más propensos a errores que el código de programa original (de la misma manera que los errores tipográficos en la mayoría de los periódicos son el resultado de cambios en la redacción de último minuto, en lugar de errores en la copia original). Es necesario un plan de pruebas de regresión (quién, cómo, cuándo).

6 Herramientas y automatización

El testing de incluso un proyecto pequeño puede implicar la necesidad de miles de casos de prueba, aun usando técnicas como partición de equivalencia para reducir la cantidad de casos. Además recuerde que las pruebas de regresión implican la re-ejecución de los casos de prueba cada vez que ocurre una modificación del programa.

El problema es que usted necesita probar más pero no tiene el tiempo. La respuesta es la automatización, que ofrece las siguientes ventajas:

- **Velocidad:** piense en cuánto le llevaría ejecutar manualmente un caso de prueba para la aplicación calculadora de Windows, seguramente más de 5 segundos. Con automatización se podría incrementar esta velocidad en 100 a 1000 veces.
- **Eficiencia:** mientras usted ejecuta tests manualmente no puede hacer otra cosa. Mientras se ejecutan tests automatizados, usted puede usar ese tiempo para planificar o pensar en nuevos tipos de tests.
- **Precisión:** luego de ejecutar muchos tests manuales, un tester humano pierde la atención y empieza a cometer errores. Una herramienta automatizada ejecuta los test y verifica los resultados perfectamente una y otra vez.
- **Resistencia:** una herramienta automatizada nunca se cansa ni se rinde.

Sin embargo, las herramientas automáticas de test no son un sustituto para los testers humanos, solamente son una ayuda para que hagan su trabajo más fácilmente. El testing automatizado no siempre es la respuesta correcta. En algunos casos no existe un sustituto para el testing manual.

También existen herramientas para asistir en la planificación y el seguimiento

6.1 Herramientas para Testing

Existe una gran cantidad de herramientas de software que asisten al tester en la gestión y ejecución de las pruebas. En esta sección se mencionan solamente algunas de ellas agrupadas en las categorías principales:

6.1.1 Herramientas para planificación y administración de pruebas

Objetivo: Estas herramientas ayudan a planificar la estrategia de pruebas que se elija y a administrar el proceso de prueba mientras se lleva a cabo.

Mecánica: Las herramientas en esta categoría abordan la planificación de las pruebas, el almacenamiento, administración y control de las mismas; el seguimiento de los requerimientos, la integración, el rastreo de errores y la generación de reportes. Los líderes de proyecto los usan para complementar las herramientas calendarizadas del proyecto. Quienes realizan las pruebas usan estas herramientas para planificar actividades de testing y controlar el flujo de información a medida que avanza el proceso de pruebas.

Herramientas representativas:

- *qTest*, desarrollada por Tricentis (<https://www.tricentis.com/products/agile-dev-testing-qtest/>), es una herramienta para la administración de casos de prueba con una interfaz gráfica amigable que permite organizar, realizar seguimiento y crear reportes de los tests.
- *TestRail*, de GuruRock (<http://www.gurock.com/testrail>), es otra herramienta de administración de casos de prueba que se puede integrar con herramientas de seguimiento de errores e incidencias tales como Jira y muchas otras.
- *Zephyr Enterprise*, de Smarbear (<https://www.getzephyr.com>), es una plataforma de administración que contempla todos los aspectos relacionados con la calidad del software. Está disponible como servicio en la nube o para instalar en un servidor privado.
- *TestLink* (<http://testlink.org>), es una herramienta de código abierto para gestión de pruebas, con interfaz web. Proporciona especificación de pruebas, planes de prueba y ejecución, informes, especificación de requisitos y colabora con conocidas herramientas de seguimiento de errores (bug trackers). El propósito de TestLink es

responder preguntas como: ¿Para qué requisitos necesitamos escribir o actualizar casos de prueba? ¿Qué pruebas ejecutar para esta versión? ¿Cuánto se progresó al probar esta versión? ¿Qué casos de prueba están fallando actualmente y cuáles son los errores? ¿En qué versión se ejecutó por última vez este grupo de casos de prueba?, ¿Es hora de que los ejecutemos nuevamente? Y finalmente: ¿esta versión del producto es apta para el lanzamiento (release)?

- *TestOps* de Katalon (<https://testops.katalon.io>) Permite administrar, planificar, ejecutar y realizar seguimiento de los tests. Ofrece versión de evaluación gratuita por tiempo limitado.

6.1.2 Herramientas para diseño de casos de prueba

Objetivo: Ayudar al equipo de software en el desarrollo de un conjunto completo de casos de prueba tanto de caja negra como de caja blanca.

Mecánica: Estas herramientas se clasifican en dos categorías amplias: las herramientas **estáticas** y las herramientas **dinámicas**. Existen dos tipos de herramientas estáticas: herramientas basadas en código, y herramientas basadas en requerimientos. Las primeras aceptan código fuente como entrada y realizan algunos análisis que dan como resultado la generación de casos de prueba. Las herramientas basadas en requerimientos aíslan requerimientos de usuario específicos y sugieren casos de prueba (o tipos de pruebas) que revisarán los requerimientos. Las herramientas dinámicas interactúan con un programa en ejecución, comprueban la cobertura de ruta, prueban las afirmaciones (assertions) sobre el valor de variables e instrumentan el flujo de ejecución del programa.

Herramientas representativas:

- *DevPartner*, desarrollada por Microfocus (<https://www.microfocus.com/es-es/products/devpartner>), analiza el código fuente y detecta errores comunes como fugas de memoria, y cuellos de botella que afectan el rendimiento. También verifica que la codificación cumpla los estándares y buenas prácticas
- *McCabeTest*, desarrollada por McCabe (www.mccabe.com), implementa una variedad de técnicas de prueba de trayectoria derivadas de una evaluación de complejidad ciclomática y de otras mediciones de software.

6.1.3 Monitores y visores

Son herramientas que permiten ver detalles del funcionamiento del software que normalmente no se podrían ver. Un ejemplo son los analizadores de protocolo. Usando estas herramientas, por ejemplo, se podría ejecutar un caso de prueba en la computadora 1, ver los resultados en la computadora 2 y un reporte de las comunicaciones en la

computadora 3. De esta forma, si los resultados no son los esperados, se puede saber si el problema está en el software de la computadora 1 o 2.

Una excelente herramienta de este tipo es Wireshark (<https://www.wireshark.org>) que es una herramienta open source que funciona en las plataformas más utilizadas (Linux, Windows, Mac OS, Solaris, BSD, etc.). Esta herramienta permite visualizar y guardar los paquetes que viajan por la red. Por lo tanto es fundamental para hacer testing y debugging de sistemas que se comunican por red (ej. aplicaciones Web o cliente / servidor). Para realizar la captura se selecciona una interfaz (ejemplo placa de red) y luego se pueden aplicar filtros para capturar sólo los paquetes en los que estamos interesados. Estos filtros pueden incluir múltiples condiciones combinadas con operadores lógicos (and, or, not). Las condiciones pueden aplicarse sobre protocolos de diversos niveles como ser transporte (ej. Puerto TCP 80, es decir http), red (ej. dirección IP de origen 192.168.1.1), enlace (ej. Dirección Ethernet/MAC de destino FF:FF:FF:FF:FF:FF –o sea broadcast-) etc. (Figura 6-1).

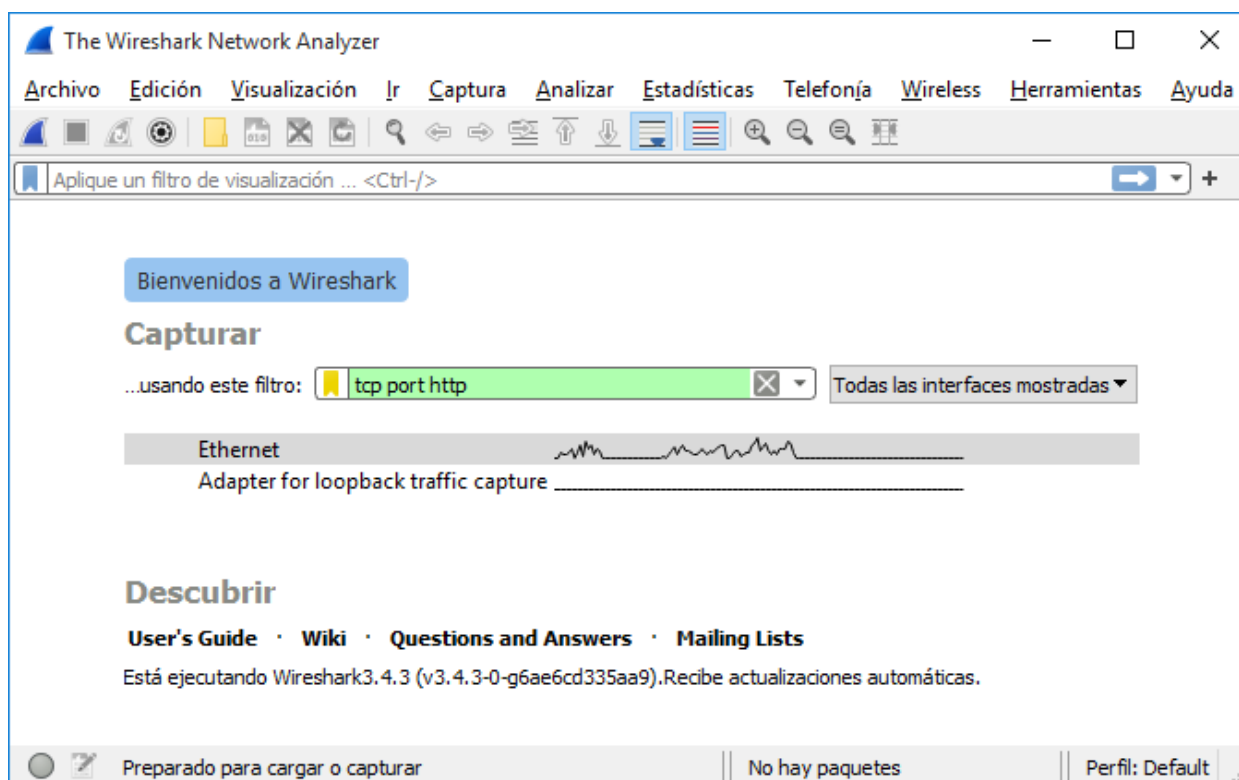


Figura 6-1. Selección de interfaz y aplicación de filtro de protocolo y puerto en Wireshark

Los paquetes capturados se pueden visualizar y/o almacenar. En la visualización la herramienta interpreta y muestra automáticamente los campos de las diferentes capas de los protocolos así como el contenido (datos o carga útil del paquete). En la Figura 6-2 se muestra una captura de pantalla de esta herramienta.

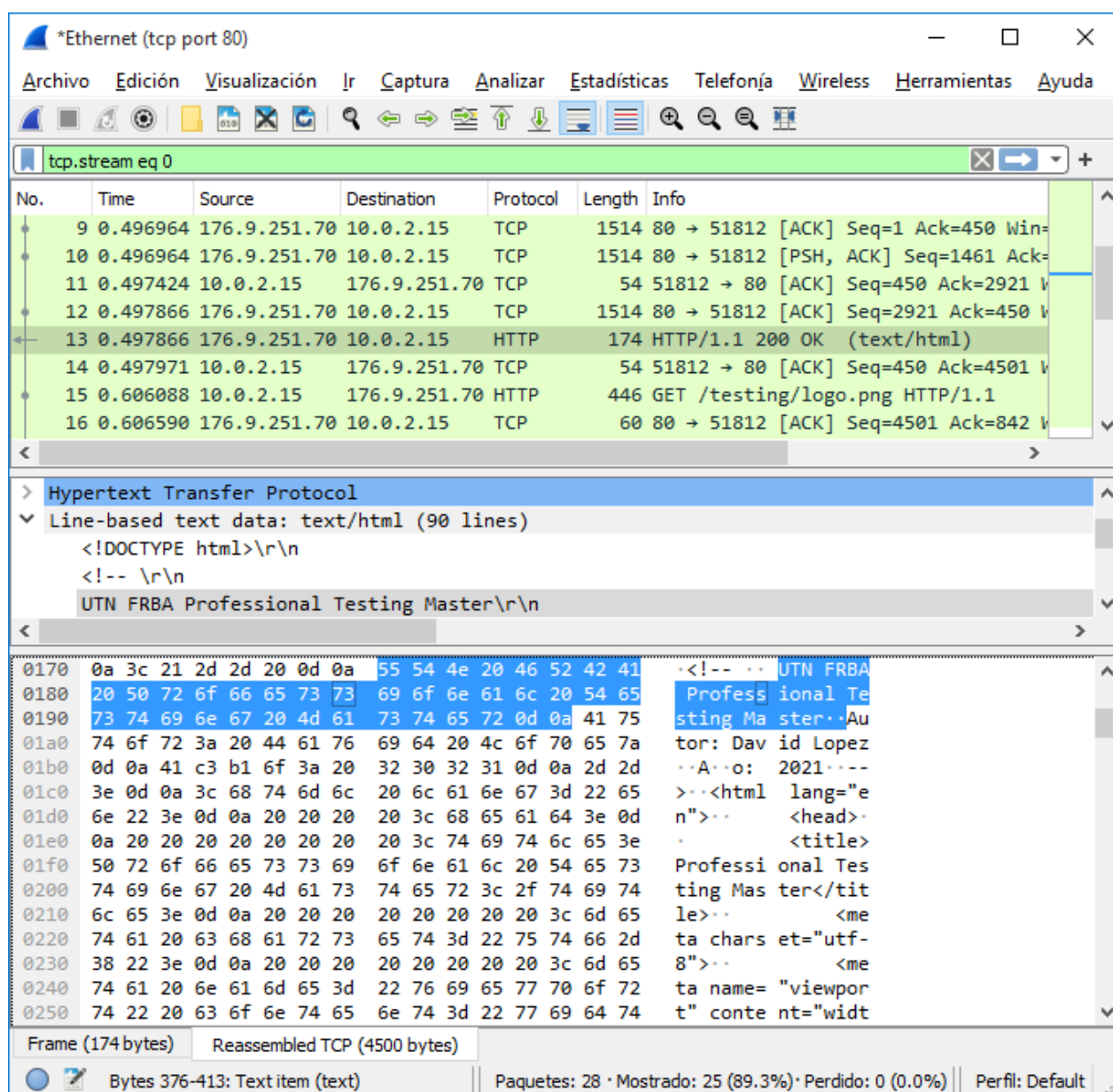


Figura 6-2. Visualización de los paquetes monitoreados en Wireshark

Otro ejemplo de monitores son los depuradores (debuggers) que vienen con la mayoría de los compiladores y entornos de desarrollo (IDEs), ya que permiten hacer testing de caja blanca, al poder visualizar el contenido de las variables internas y la ruta de ejecución de los programas.

6.1.4 Drivers

Son herramientas que controlan y ejecutan el software que está siendo probado.

El ejemplo más simple de driver son los archivos batch, que consisten en una lista de comandos que se deben ejecutar, y pueden incluir cierta lógica simple como por ejemplo condiciones o ciclos. Estos archivos batch se pueden reemplazar por sofisticados scripts usando lenguajes de scripting tan poderosos como Perl, PowerShell o Python. También se pueden incluir herramientas de scheduling como el programador de tareas de Windows o la herramienta Cron de Linux para lanzar la ejecución de ciertas tareas en un horario predefinido.

Existe otro tipo de driver. Si el software que está probando requiere gran cantidad de ingresos mediante el teclado y el mouse, puede usar un software especial y programar en él las entradas de teclado y movimientos de mouse. Ejemplos de esta herramienta son *xdotool* en Linux, *AutoHotkey* y *Autolt* en Windows, o *SikuliX* (Java multiplataforma).

6.1.5 Stubs

Los stubs ya fueron mencionados en la Unidad 1, y son lo contrario de los drivers, en el sentido que no controlan la ejecución del software a probar, sino que responden a datos enviados por éste. Por ejemplo, si se debe probar la salida de impresión de un programa una opción es utilizar una impresora real e imprimir cientos o miles de copias. Esto funcionaría pero sería ineficiente, lento y hasta ineficaz, ya que podría haber errores en unos pocos píxeles que pasarían inadvertidos ante una inspección visual. En lugar de esto se puede reemplazar la impresora por una computadora ejecutando un software stub, que recibiría y analizaría los datos. Los stubs también se utilizan si el software debe comunicarse con dispositivos externos que durante el desarrollo no están disponibles o son escasos.

Posiblemente usted haya escuchado el nombre emulador (emulator) para describir un dispositivo o software que es un reemplazo para un dispositivo real. Un programa que actúa como una impresora, recibiendo comandos de impresión y respondiendo al software

como si fuera la impresora es un emulador. La diferencia con un stub es que el stub además provee al tester un modo de visualizar o interpretar los datos recibidos.

6.1.6 Herramientas de carga y stress

Una aplicación como un procesador de texto podría funcionar bien cuando se ejecuta como única aplicación en una computadora teniendo todos los recursos disponibles, pero podría empezar a fallar a medida que se queda sin espacio en memoria o disco, o sin recursos de procesamiento. Usted podría empezar a copiar archivos para ocupar el disco, y ejecutar programas que consuman memoria, pero esto sería trabajoso. Una herramienta de stress hace esta tarea mucho más fácil.

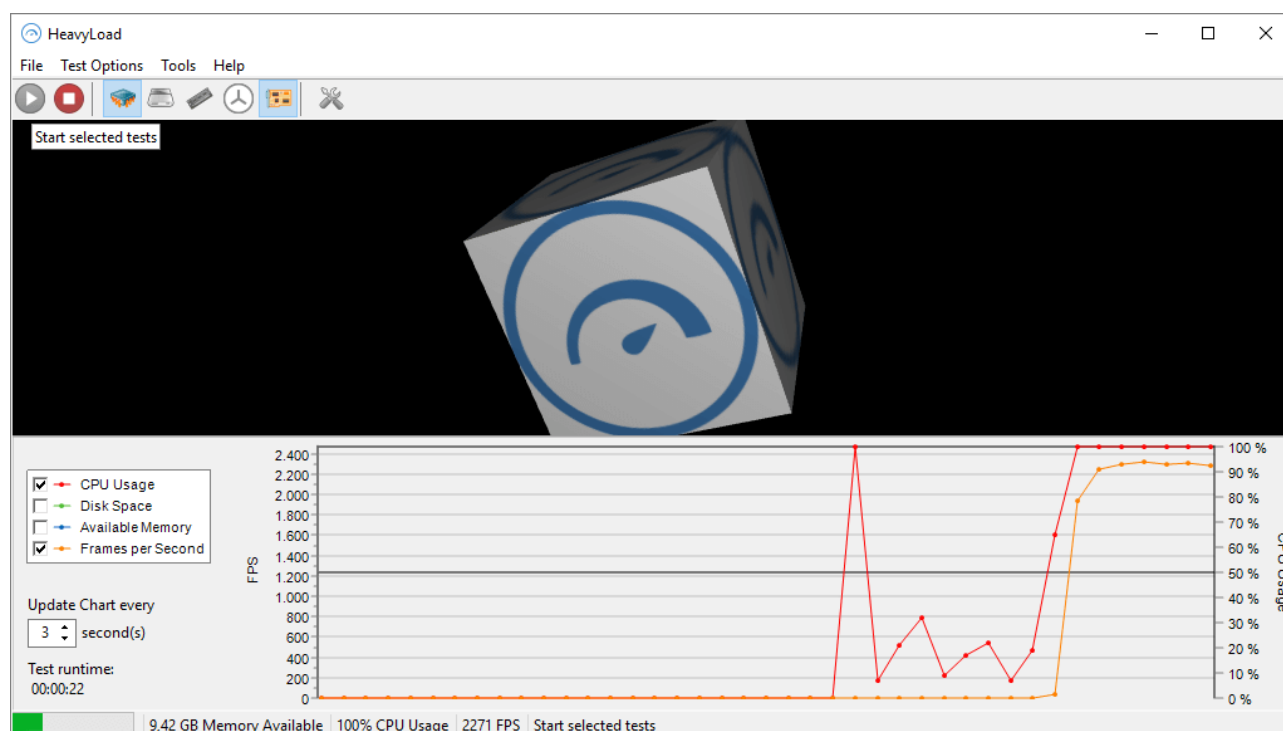


Figura 6-3. Heavy Load

La Figura 6-3, muestra la herramienta gratuita HeavyLoad para Windows (<https://www.jam-software.com/heavyload>). Otros sistemas operativos tienen herramientas similares. Establecer los valores de recursos disponibles cercanos a cero hará que el programa a probar tenga que ejecutar otros caminos para lidiar con la falta de recursos. Idealmente, el software debería ejecutarse sin colgarse ni perder datos.

Simplemente debería ejecutarse más lento o mostrar mensajes indicando que la memoria es escasa o similares.

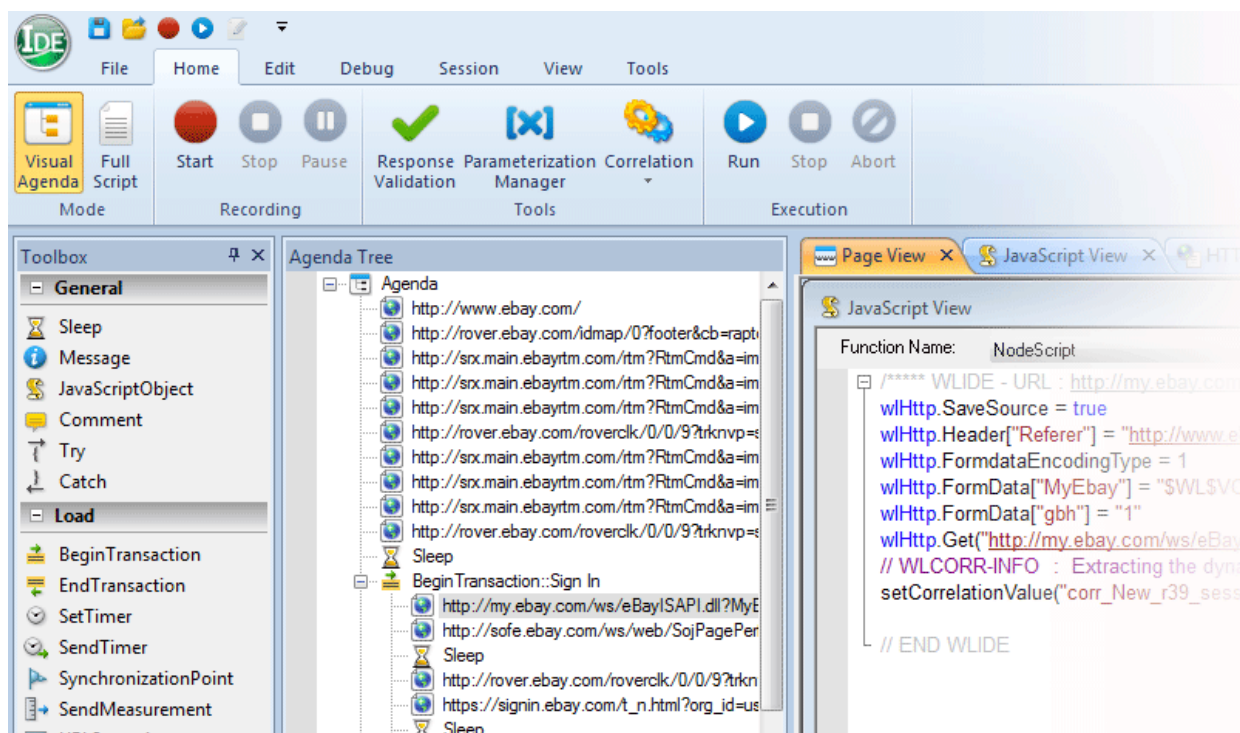


Figura 6-4. IDE de WebLOAD

Las herramientas de carga son similares en cuanto a que generan situaciones que de otra forma serían difíciles de reproducir. Por ejemplo, existen herramientas comerciales que se ejecutan en servidores web para cargarlos simulando una cierta cantidad de conexiones y accesos. Usted podría querer probar una webapp con 10000 o más usuarios simultáneos y uno o varios millones de accesos por día. Con estas herramientas simplemente ingresa estos valores, ejecuta los tests y verifica el tiempo de respuesta. Un buen ejemplo de este tipo de herramientas es *WebLOAD*, desarrollada por Radview (<https://www.radview.com>). Soporta aplicaciones web y mobile, mide la performance del lado del servidor y del cliente. Soporta tecnologías HTTP/HTTPS, WebSocket, PUSH, AJAX, SOAP, REST, HTML5, WebDAV, JavaScript, AngularJS, entre otras. Tiene una versión comercial y otra gratuita. En la Figura 6-4 se muestra la interfaz de la versión gratuita de WebLOAD.

Una alternativa de código abierto que también ofrece funciones de prueba de carga para analizar y medir el desempeño de una variedad de servicios, con énfasis en aplicaciones web, es Apache Jmeter (<http://jmeter.apache.org>).


6.2 Herramientas para automatización


6.2.1 Grabación y reproducción de macros

El tipo más simple de automatización es grabar todas las acciones de mouse y teclado que el tester realiza durante los tests y luego reproducirlas cuando se necesita ejecutar nuevamente los tests. Los grabadores y reproductores de macros (record and playback) entran en la categoría de drivers, ya que se encargan de ejecutar el software a probar.

Para realizar esta tarea con aplicaciones de escritorio en Windows, una de las primeras y más sencillas herramientas fue Macro Magic, la cual se encuentra discontinuada aunque aún se puede conseguir en Internet una versión de evaluación 30 días perfectamente funcional en Windows 10.

Una alternativa más moderna y gratuita es Pulover's Macro Creator (<http://www.macrocreator.com/>). Esta herramienta de código abierto para Windows provee un IDE¹ para grabación y reproducción de macros que también pueden ser editadas desde el código. Está basada en el lenguaje de automatización AutoHotKey, el cual incluye ciclos y ramificaciones de tipo IF. Su interfaz se muestra en la Figura 6-5.

Para grabar una macro con esta herramienta, simplemente debe presionar el botón de grabación , luego presionar F9 para iniciar la grabación, realizar las acciones (ej. Iniciar, operar y cerrar la aplicación, y presionar nuevamente F9 para finalizar la grabación).

Luego puede ejecutar la macro grabada mediante el ícono de reproducción . La macro también se puede repetir una cantidad determinada de veces, ejecutarse en modo acelerado o más lento, entre otras opciones.

Con respecto a la grabación se puede configurar qué se captura: puede grabar solo teclas o teclas y acciones de mouse como mover y clicar o solo clicar.

¹ IDE: Integrated Development Environment (Entorno de desarrollo integrado)

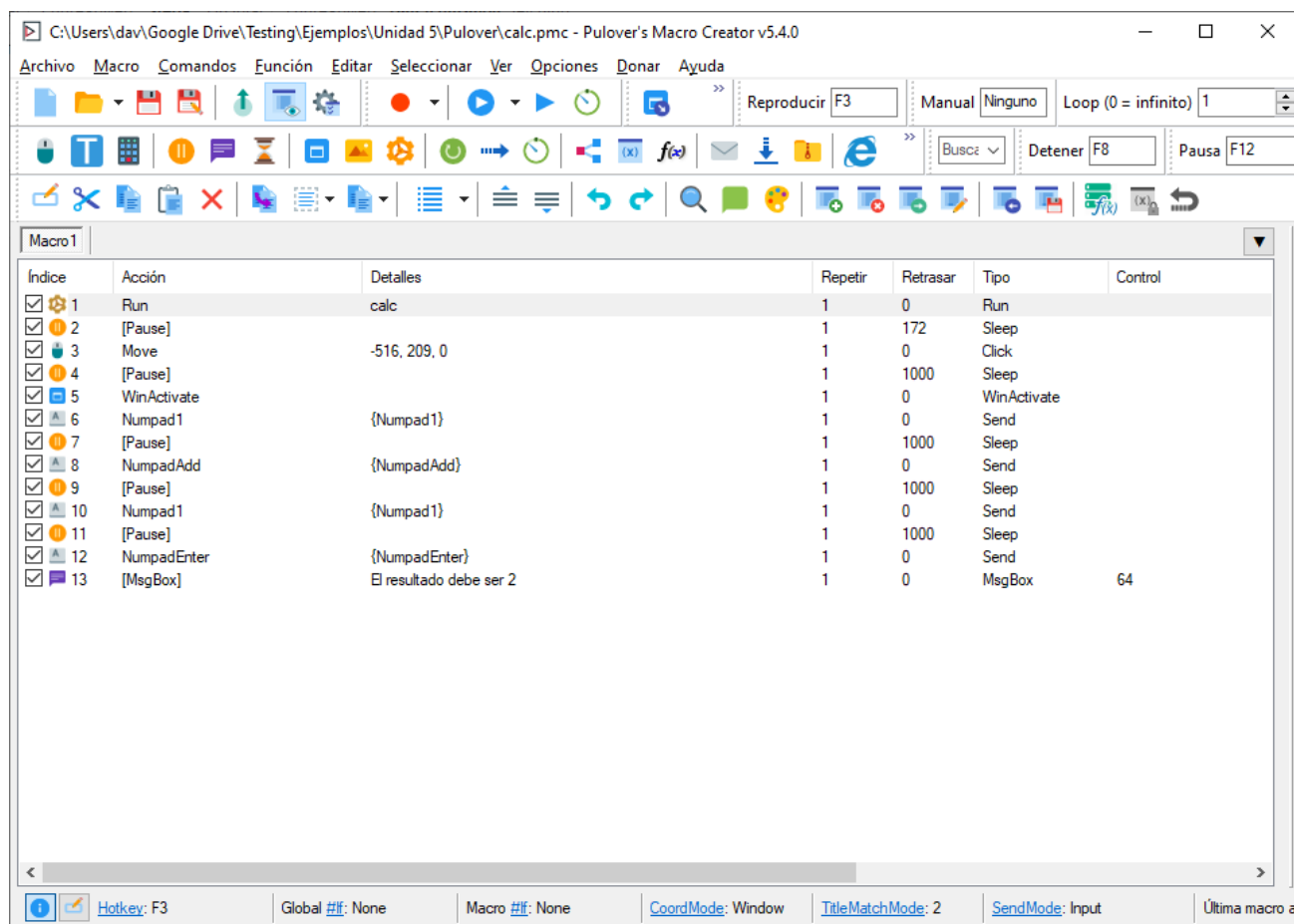


Figura 6-5. Herramienta Pulover's Macro Creator

Ahora es un buen momento para experimentar con macros. Descargue algún software de macros y pruébelo en algunos programas simples como la Calculadora o el Notepad y fíjese qué le parece.

Lo que encontrará es que si bien el software de grabación de macros puede hacer algo de automatización haciendo mucho más fácil y rápido volver a ejecutar tests, no es perfecto. El mayor problema es la falta de verificación. La macro puede ingresar 100-99 en la calculadora, pero no puede verificar que el resultado es 1. Usted debe hacerlo. Esto es un problema, claro, pero muchos testers estarán felices de eliminar el ingreso por teclado y los movimientos de mouse. Es mucho más fácil simplemente observar las macros ejecutándose y confirmar que los resultados son los esperados.

Otra dificultad con las macros es la velocidad de reproducción. Si bien se puede configurar, esto no siempre es suficiente para mantenerlas sincronizadas. Una operación podría tardar 1 ó 10 segundos. Se puede configurar la macro para el peor caso esperado, pero esto haría que el test funcione lento aun si el software funciona más rápido. Y si el software inesperadamente tarda 15 segundos en cargar, sus macros podrían confundirse cliqueando elementos equivocados en momentos equivocados.

A pesar de estas limitaciones, la grabación y reproducción de macros es una forma popular de automatizar tareas simples de test. También es un buen comienzo para testers que están aprendiendo cómo automatizar sus tareas.

6.2.2 Macros programables

Las macros programables son un paso evolutivo respecto de las macros de grabación y reproducción. En lugar de crearlas grabando las acciones cuando se ejecuta el caso de prueba por primera vez, se crean mediante una simple instrucción para que la siga el programa de reproducción. Un programa de macros muy simple (creado con Pulover's Macro Creator) se ve como se muestra a continuación (Figura 6-6):

```

1 Run, calc
2 WinActivate, Calculadora
3 Sleep, 1000
4 Send, {Numpad1}
5 Sleep, 1000
6 Send, {NumpadAdd}
7 Sleep, 1000
8 Send, {Numpad1}
9 Sleep, 1000
10 Send, {NumpadEnter}
11 Sleep, 1000
12 MsgBox, 64, , El resultado debe ser 2
13 WinClose, Calculadora
  
```

Figura 6-6. Script en lenguaje AutoHotkey

Esta macro se puede programar seleccionando acciones de un menú de opciones. No es necesario escribir comandos. Con Pulover's también se genera automáticamente (con pequeñas diferencias) al grabar las acciones como se explicó antes.

La línea 1 ejecuta la calculadora de Windows (aplicación calc.exe). La línea 2 busca la ventana con título "Calculadora" y la activa (pone el foco para que tome los ingresos). Las

líneas 3, 5, 7, 9, 11, generan esperas de 1 segundo (1000 milisegundos) para dar tiempo a la aplicación a responder. Las líneas 4, 6, 8, 10 simulan el uso del teclado numérico para ingresar la operación “1+1=”. La línea 12 muestra un mensaje diciendo que la respuesta esperada es 2. La línea 13 cierra la ventana Calculadora y termina el test.

Tenga en cuenta que las macros programadas como ésta tienen algunas ventajas sobre las macros grabadas. A pesar de que todavía no se puede llevar a cabo la verificación de los resultados de las pruebas, se puede detener su ejecución con un resultado esperado y una consulta para pedir al tester (véase la **¡Error! No se encuentra el origen de la referencia.**) que confirme si la prueba se ha superado o no.

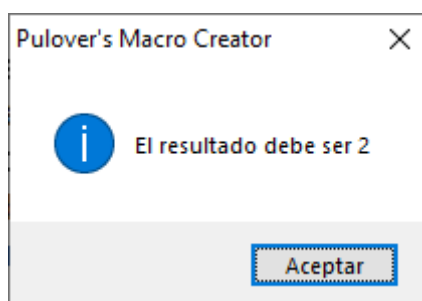


Figura 6-7. Pedido de confirmación de resultado

Las macros programadas también pueden resolver muchos problemas de tiempo de las macros grabadas, por no depender de los retrasos absolutos, sino esperar a que se produzcan ciertas condiciones antes de seguir adelante. En el ejemplo de la calculadora, se puede usar el comando RunWait en lugar de Run para esperar a que el programa se cargue antes de continuar con el test, un método mucho más confiable que usar un Sleep².

Hasta ahora, todo bien. Con las macros programadas se puede recorrer un largo camino hacia la automatización de las pruebas. Usted tiene un lenguaje de macros fácil de usar, comandos genéricos para el manejo de su software, y un medio para que le solicite información. Para muchas tareas de prueba, esto es más que suficiente, y se ahorrará una gran cantidad de tiempo automatizando las pruebas de esta manera.

² En el ejemplo no se usó RunWait ya que el siguiente comando es WinActivate y ya realiza una cierta espera para activar la ventana.

Sin embargo, aún está faltando la capacidad de comprobar automáticamente los resultados de la prueba. Para esto, es necesario pasar a una herramienta de pruebas completamente automatizada.

6.2.3 Herramientas de testing totalmente programables

¿Qué pasaría si tuviera la potencia de un lenguaje de programación combinada con macros para manejar el software bajo prueba y capacidad de verificar los resultados?. ¡Tendría la herramienta ideal!

Las herramientas automatizadas como Katalon Studio o similares proveen los medios para crear tests muy poderosos.

Un lenguaje de testing también le ofrece mejores funciones de control que simplemente clicar áreas específicas de la pantalla o simular el pulsado de teclas individuales. Por ejemplo, hay mecanismos para presionar un botón OK sin saber sus coordenadas en la pantalla. El software de prueba se encarga de buscarlo, encontrarlo, y hacer clic en él tal como lo haría un usuario. Del mismo modo, hay comandos de menús, casillas de verificación, botones de opción, cuadros de lista, y así sucesivamente. Los comandos de este tipo proporcionan una gran flexibilidad en la escritura de sus tests, haciéndolos mucho más legibles y confiables. La característica más importante que viene con estas herramientas de automatización es la capacidad de realizar la verificación, comprobando que el software está haciendo lo que se espera. Hay varias maneras de hacer esto:

- **Capturas de pantalla.** La primera vez que ejecute las pruebas automatizadas, se puede capturar y guardar imágenes de pantalla en puntos clave que sabe que son correctos. En futuras ejecuciones de tests, la automatización podría entonces comparar las pantallas guardadas con las pantallas actuales. Si son diferentes, algo inesperado sucedió y la automatización puede marcarlo como un error.

Tenga en cuenta que las pruebas que utilizan capturas de pantalla pueden requerir un enorme esfuerzo para mantenerse. Incluso un cambio de un píxel podría hacer que la comparación falle³. A menos que la interfaz de usuario del software no cambie, usted podría terminar comparando manualmente y volviendo a capturar las

³ Existen herramientas como AppliTools que incorporan técnicas de inteligencia artificial para mitigar este problema al comparar solo los rasgos más relevantes de las imágenes como lo haría un humano. Igualmente no son perfectas.

pantallas con cada ejecución de las pruebas. Esto va en contra del propósito de la automatización.

- **Valores de control.** En lugar de capturar las pantallas, se puede comprobar el valor de los elementos individuales de la ventana del software. Si está probando la calculadora, la automatización podría leer el valor del campo de visualización mediante reconocimiento de caracteres (OCR) y compararlo con lo que se esperaba. También podría determinar si se ha pulsado un botón o se seleccionó una casilla de verificación. Existen herramientas que permiten hacer esto dentro de su script de pruebas.
- **Archivos y otras salidas.** Del mismo modo, si el programa guarda los datos en un archivo, (por ejemplo un procesador de textos) la automatización podría leerlo después de su creación, y compararlo con un archivo correcto conocido. Las mismas técnicas se aplican si el software que está probando envía los datos a través de una red local o Internet. La automatización puede configurarse para leer los datos y compararlos con los datos que espera.

Al igual que con las capturas de pantalla, las comparaciones de archivos también pueden tener problemas. Si el formato de archivo incluye una fecha, un contador, u otros valores cambiantes, la comparación de archivos fallará. Usted tendrá que programar su herramienta de automatización para ignorar estas diferencias.

Para obtener más información acerca de los productos de automatización de pruebas disponibles, puede visitar los siguientes sitios web:

- *Katalon Studio* (usado por Ford, Toyota, Samsung y Oracle entre otros clientes). Disponible para Windows, Linux y MacOS <https://www.katalon.com/>
- *Applitools* (uno de los sponsors del proyecto Selenium y usado por Microsoft, Bayer, Sony y Adidas entre otros). Requiere el uso de Webdriver o un framework similar. <https://applitools.com/>
- *UFT One* de Microfocus (que también es sponsor de Selenium) :
- <https://www.microfocus.com/es-es/products/uft-one/overview>
- *Test Complete* de Smartbear:
<https://smartbear.com/product/testcomplete/overview/>
- *Lambdatest* <https://www.lambdatest.com/>
- *Ranorex* (sponsor de Selenium) <https://www.ranorex.com/>

Estos paquetes pueden ser caros para los particulares ya que están dirigidos principalmente a los equipos de test corporativos. Pero si usted está interesado en ganar

algo de experiencia con ellos, solicite una copia de evaluación. La mayoría de las empresas le ayudará con la esperanza de que le guste su producto y lo recomiende a otros. Otra opción son las herramientas de código abierto aunque varían mucho en su capacidad y calidad, así que tendrá que investigar cuáles podrían funcionar mejor para sus necesidades. En este sentido no se puede dejar de mencionar las herramientas Selenium WebDriver para el caso de aplicaciones web y Appium para aplicaciones móviles y de escritorio, aunque requieren escribir código:

- Selenium WebDriver <http://www.seleniumhq.org>
- Appium <http://appium.io/>

6.2.4 Herramientas de testing aleatorio

Otro tipo de herramientas de test automatizado, los llamados monos (monkeys), se basan en el concepto de que un millón de monos escribiendo con un millón de teclados durante un millón de años podría eventualmente escribir una obra de Shakespeare, por pura casualidad estadística. De esta manera, herramientas realizando acciones al azar durante varias horas o días podrían encontrar errores utilizando el software de formas que al tester no se le hubieran ocurrido, pero que podrían darse con el software en producción luego de cierto tiempo de uso por parte de una gran cantidad de usuarios. Existen herramientas con distinto nivel de inteligencia. Un ejemplo de estas herramientas para aplicaciones web es el sitio <https://monkeytest.it/> en el que uno ingresa la URL de la webapp a probar y recibe un informe de los problemas encontrados.

7 Test Driven Development

7.1 Introducción

Normalmente, los requerimientos impulsan el diseño y éste establece una base para la construcción. Esta simple realidad en ingeniería del software funciona razonablemente bien y es esencial cuando se crea una arquitectura de software. Sin embargo, un cambio sutil puede proporcionar beneficios significativos.

En *Test Driven Development* (TDD, Desarrollo Basado en Pruebas), los requerimientos para un componente de software funcionan como la base para la creación de una serie de casos de prueba que verifiquen la interfaz y que intenten encontrar errores en las estructuras de datos y en la funcionalidad del componente. TDD no es una nueva tecnología, sino más bien una tendencia que enfatiza el diseño de casos de prueba **antes** de la creación de código fuente (ver sección 7.2).

El proceso TDD sigue el flujo que se ilustra en la Figura 7-1. Antes de crear el primer pequeño fragmento de código, se crea una prueba para verificar el código (intentar que fracase el código). Entonces el código se escribe para satisfacer la prueba. Si la pasa, se crea una nueva prueba para el siguiente segmento de código que se va a desarrollar. El proceso continúa hasta que el componente está completamente codificado y todas las pruebas se ejecutan sin error. Si alguna prueba encuentra un error, el código existente se “refactoriza” (corrige, del inglés refactor) y todas las pruebas creadas para dicho punto se vuelven a ejecutar. Este flujo iterativo continúa hasta que no hay pruebas pendientes de crear, lo que implica que los componentes satisfacen todos los requerimientos.

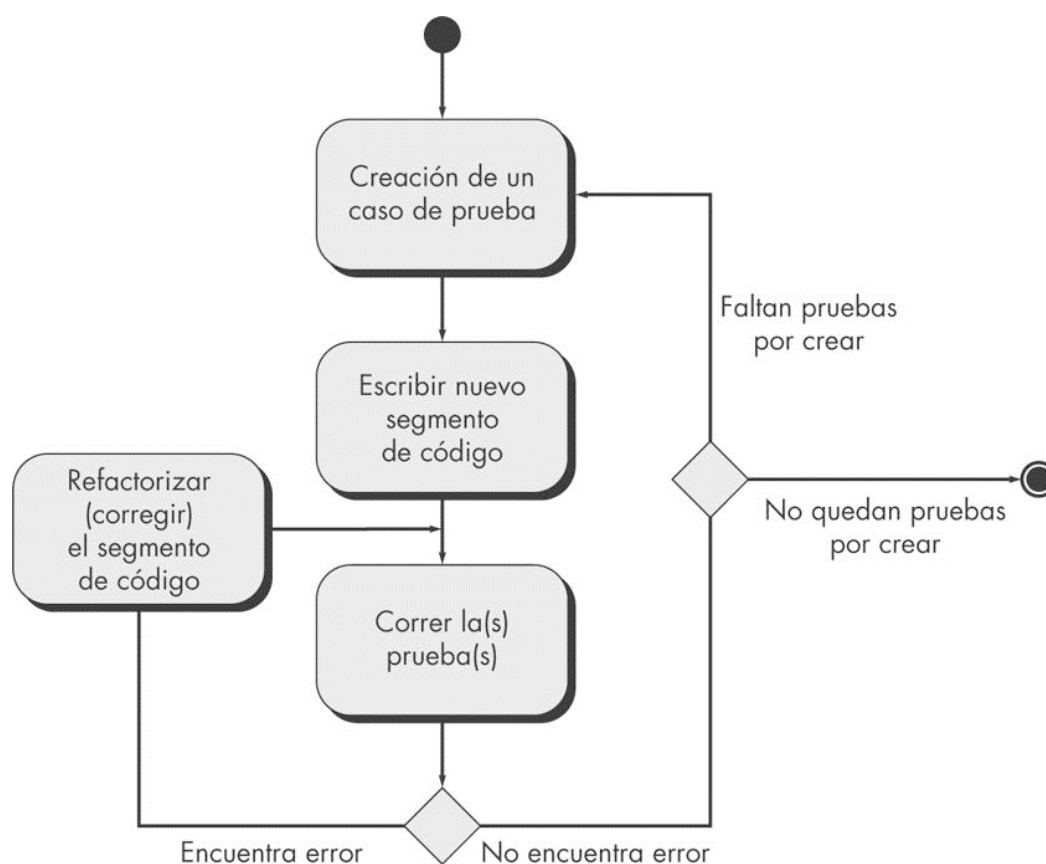


Figura 7-1. Flujo de proceso de TDD

Durante el TDD, el código se desarrolla en incrementos muy pequeños (una subfunción a la vez) y no se escribe código hasta que exista una prueba que lo verifique (que al ejecutarse debe fallar). Debe observar que cada iteración resulta en una o más pruebas nuevas que se agregan a una suite de pruebas de regresión que se ejecutan con cada cambio. Esto se hace para garantizar que el nuevo código no generó efectos colaterales que causen errores en el código anterior.

En TDD, las pruebas impulsan el diseño de componentes detallados y el código fuente resultante. Los resultados de dichas pruebas causan modificaciones inmediatas al diseño de componentes (vía el código) y, más importante, el componente resultante (cuando se completa) se verificó en forma independiente. Si tiene interés en más detalles sobre TDD, consulte [Bec04] o [Ast04].

7.2 TDD y Programación Extrema (XP)

TDD es una tendencia que enfatiza el diseño de casos de prueba antes de la creación de código fuente. Es una disciplina de diseño y programación, donde cada nueva línea de código que escribe un programador es en respuesta a una prueba que ha fallado, también escrita por el programador. Sin ser definida como una metodología de pruebas se apoya fuertemente en las pruebas unitarias y también en pruebas de aceptación, basándose en prácticas formalizadas por *Extreme Programming* (XP, Programación Extrema) [Mye12], una de las más populares metodologías ágiles de desarrollo de software. Es decir que XP enfatiza TDD como parte de su modelo de proceso ágil.

El propósito de la metodología de desarrollo XP es crear programas de calidad en cortos periodos de tiempo. XP se centra en la aplicación de diseños simples, la comunicación entre los desarrolladores y los clientes, poniendo a prueba constantemente su base de código, refactorización para acomodar cambios en las especificaciones, y la búsqueda de feedback de los usuarios. XP tiende a funcionar bien para proyectos pequeños a medianos en entornos que tienen frecuentes cambios en las especificaciones y en donde la comunicación casi instantánea es posible.

El modelo XP se basa en gran medida en las pruebas unitarias y pruebas de aceptación de módulos. En general, debe ejecutar pruebas unitarias para cada cambio de código incremental, por pequeño que sea, para asegurarse de que la base de código todavía

cumple su especificación. De hecho, la prueba es de tal importancia en XP que el proceso requiere crear primero las pruebas de unidad (módulo) y las pruebas de aceptación, y recién a continuación, crear su base de código. Este tipo de prueba se llama, apropiadamente, *Extreme Testing* (XT, Testing Extremo).

7.2.1 Conceptos de Testing Extremo

Para cumplir con el ritmo y la filosofía de XP, los desarrolladores utilizan testing extremo, que se centra en las pruebas constantes. Como se mencionó anteriormente, dos formas de pruebas constituyen la mayor parte de XT: pruebas unitarias y pruebas de aceptación. La teoría que se utiliza al escribir las pruebas no varía significativamente de la teoría presentada en las Unidades 1 y 2. Sin embargo, la etapa en el proceso de desarrollo en el que se crean las pruebas es diferente. Sin embargo, XT y las pruebas tradicionales todavía tienen el mismo objetivo: identificar errores en un programa.

En el resto de esta sección se ofrece más información sobre las pruebas unitarias y pruebas de aceptación, desde la perspectiva de la programación extrema.

7.2.1.1 Pruebas unitarias extremas

La prueba unitaria es el enfoque de prueba principal que se utiliza en XT y tiene dos sencillas reglas: Todos los módulos de código deben tener las pruebas unitarias antes de que comience la codificación, y todos los módulos de código deben pasar las pruebas unitarias antes de ser liberados a producción. A primera vista, esto puede no parecer tan extremo. Sin embargo, la gran diferencia entre las pruebas unitarias como se han descrito anteriormente y XT es que las pruebas unitarias deben ser definidas y creadas antes de la codificación del módulo.

Inicialmente, usted puede preguntarse por qué debe, o cómo se puede, crear los casos de prueba para el código que ni siquiera ha escrito. También puede pensar que no tiene tiempo para crear las pruebas ya que la aplicación debe cumplir con una fecha límite. Estas son preocupaciones válidas, pero fáciles de abordar. La siguiente lista identifica algunos de los beneficios asociados con la escritura de pruebas unitarias antes de empezar a programar la aplicación:

- Se gana la confianza de que su código cumplirá con la especificación.
- Usted expresa el resultado final de su código antes de empezar a programar.
- Usted entiende mejor la especificación y los requerimientos de la aplicación.

- Es posible inicialmente implementar diseños simples y tranquilamente refactorizar el código más adelante para mejorar el rendimiento sin tener que preocuparse por romper la especificación.

De estos beneficios, no se puede subestimar el conocimiento y la comprensión que se obtienen de las especificaciones y los requisitos de la aplicación. Por ejemplo, no se puede entender completamente los tipos de datos aceptables y los límites de los valores de entrada de una aplicación si se inicia la codificación primero. Así que ¿cómo se puede escribir una prueba unitaria para realizar análisis de frontera sin entender las entradas aceptables? ¿La aplicación puede aceptar sólo números, sólo caracteres, o ambos? Si crea las pruebas unitarias en primer lugar, usted debe entender la especificación. La práctica de crear pruebas unitarias primero es el punto brillante de la metodología XP, ya que obliga a entender la especificación para resolver ambigüedades antes de comenzar la codificación.

Como se mencionó en la Unidad 3, usted determina el alcance de la unidad. Dado que los lenguajes de programación más populares de hoy en día, tales como Java, C# y JS son en su mayoría orientados a objetos, los módulos son a menudo las clases. A veces se puede definir un módulo como un conjunto de clases o métodos que representan algunas funciones. Sólo usted, como programador, conoce la arquitectura de la aplicación, y la mejor manera de construir las pruebas unitarias para ello.

Ejecutar manualmente pruebas unitarias, incluso para la aplicación más pequeña, puede ser una tarea desalentadora. A medida que la aplicación crece, puede generar cientos o miles de pruebas unitarias. Por lo tanto, normalmente se utiliza una suite de pruebas automatizadas para aliviar la carga de la ejecución constante de pruebas unitarias. Con estas suites se generan scripts de tests y luego se ejecuta la totalidad o parte de ellos. Además, las suites de pruebas normalmente le permiten crear informes y clasificar los errores que ocurren con frecuencia en su aplicación. Esta información puede ayudarle proactivamente a eliminar errores en el futuro.

Curiosamente, una vez que se crean y validan las pruebas unitarias, el código de prueba llega a ser tan valioso como la aplicación de software que está tratando de crear. Como resultado, usted debe tener las pruebas en un repositorio⁴ de código como protección.

⁴ Un repositorio de código es un sistema de base de datos que se utiliza para almacenar archivos (generalmente de texto, como código fuente) y mantener un control de versiones permitiendo a varias personas hacer modificaciones en paralelo. Ejemplos de estos sistemas son GIT y SVN (SubVersion).

Además, debe asegurarse de que se realizan copias de seguridad adecuadas, así como que se tiene la seguridad necesaria en el servidor.

7.2.1.2 Pruebas de aceptación

Las pruebas de aceptación representan el segundo, e igualmente importante tipo de XT que se produce en la metodología XP. El propósito de las pruebas de aceptación es determinar si la aplicación cumple con otros requisitos, como funcionalidad y facilidad de uso. Usted y el cliente crean las pruebas de aceptación en las fases de diseño / planificación.

A diferencia de las otras formas de pruebas examinadas hasta ahora, las pruebas de aceptación son llevadas a cabo por los clientes, no los testers ni desarrolladores. De esta manera, los clientes proporcionan la verificación imparcial de que la aplicación cumple con sus necesidades. Los clientes crean las pruebas de aceptación a partir de las historias de usuario⁵. Pueden necesitarse varias pruebas de aceptación para cada historia de usuario.

Las pruebas de aceptación en XT pueden o no ser automatizadas. Por ejemplo, se requiere una prueba manual cuando el cliente debe validar que una pantalla de entrada de usuario cumple con su especificación con respecto al color y la disposición de pantalla. Un ejemplo de una prueba automatizada es cuando la aplicación debe calcular los sueldos mediante la entrada de datos como un archivo plano para simular los valores de producción.

Con las pruebas de aceptación, el cliente valida un resultado esperado de la aplicación. Una desviación del resultado esperado se considera un error y se informa al equipo de desarrollo. Si los clientes descubren varios errores, entonces deben asignarles una prioridad antes de pasar la lista al grupo de desarrollo. Después de corregir los errores, o después de cualquier cambio, los clientes deben volver a ejecutar las pruebas de aceptación. De esta manera, las pruebas de aceptación también se convierten en una forma de prueba de regresión.

Un punto importante es que un programa puede pasar todas las pruebas unitarias pero fallar las pruebas de aceptación. ¿Por qué? Debido a que una prueba unitaria verifica si una unidad (componente) de programa cumple con determinadas especificaciones como el cálculo de las deducciones del sueldo, pero no una determinada funcionalidad. La

⁵ El concepto de historia de usuario en XP es similar al concepto de caso de uso en UML o AOO (Unidad 3).

comprensión de la especificación, pero no la funcionalidad, generalmente crea este escenario. También debemos recordar que para una aplicación comercial el “look and feel”⁶ es muy importante y no se debe pasar por alto.

8 Certificaciones para testing

El aumento de la competencia y la demanda de los clientes han hecho del software testing una operación clave que exige hoy en día los mejores talentos. Como resultado, hay un aumento de la demanda de testers expertos y muchas compañías a nivel mundial buscan testers certificados. En las siguientes secciones se discutirán dos de las principales certificaciones para software testing a nivel internacional.

8.1 International Institute of Software Testing®

El Instituto Internacional de Pruebas de Software (IIST®) es un proveedor de educación basada en certificaciones. El Consejo Asesor del IIST, es un grupo de expertos y profesionales de la industria, y ofrece orientación a los esfuerzos para desarrollar educación basada en certificaciones.

8.1.1 Certificaciones

Para lograr el objetivo de la educación basada en certificaciones, el IIST tiene varias certificaciones. Cada certificación se basa en un conjunto bien definido de conocimientos (Body of Knowledge, BOK), aprobado por el Consejo Asesor del IIST. Para conseguir estas certificaciones, el candidato debe realizar una serie de cursos llevados a cabo por un instructor, y aprobar el examen en cada curso. Los exámenes en los cursos de educación basada en certificación no son del tipo multiple choice o verdadero / falso. El Consejo Asesor del IIST acordó que este modelo es muy superior a las certificaciones que

⁶ “look and feel” es el diseño de la interfaz gráfica y la percepción del usuario al interactuar con ella. Este aspecto junto con la usabilidad (facilidad de uso), adecuada funcionalidad de la aplicación, y otros factores forman parte de lo que se conoce como UX (User eXperience, o experiencia de usuario).

se basan en pasar un examen y no requieren de un riguroso curso de estudio. En los siguientes párrafos se describen las principales certificaciones.

8.1.1.1 Certified Software Test Professional – Associate Level (CSTP-A)

Objetivos

- Conseguir un mejor entendimiento de la terminología y conceptos.
- Mejorar la comunicación entre los miembros del equipo de prueba.
- Proveer técnicas para trabajar con requerimientos incompletos.
- Obtener un mejor entendimiento del proceso de prueba.
- Ayudar a probar aun sin requerimientos o con requerimientos pobres.
- Enseñar cómo descomponer los requerimientos en escenarios para un mejor testing.
- Ayudar a los profesionales a desarrollar una prueba de regresión efectiva.
- Enseñar un proceso sistemático para pruebas positivas y negativas.
- Enseñar las mejores formas para documentar su diseño de pruebas.
- Ayudar a obtener un mejor entendimiento de los diferentes niveles de prueba.
- Ayudar a colaborar con los desarrolladores y a cerrar los gaps entre los requerimientos y el código para obtener mejor cobertura de la prueba.
- Ayudar a los profesionales a monitorear el progreso de ejecución de sus pruebas.

Destinatarios

- Cualquier persona nueva en esta área.
- Este curso es esencial para cada profesional de software involucrado en la disciplina de pruebas de software: ingenieros y analistas de testing interesados en desarrollar actividades efectivas en planificación, diseño y monitoreo de todas las actividades de prueba.
- Equipos de pruebas de aceptación de usuario responsables de verificar el desempeño y funcionalidad.
- Equipos de desarrollo interesados en desarrollar testing efectivo unitario y de integración.
- Gerentes o líderes de desarrollo y pruebas interesados en obtener mayor control sobre las actividades de prueba y calidad del producto de software.

- Testers de software con experiencia en métodos empíricos e informales.
- Cualquier persona que desee convertirse en un profesional de testing y obtener un mejor entendimiento en la disciplina de pruebas del software.

Requerimientos de educación formal

Tres días de entrenamiento que cubren las áreas 1 y 2 del [Test Professional Body of Knowledge \(TPBOK\)](#).

Requerimientos de experiencia laboral

Ninguno.

8.1.1.2 Certified Software Test Professional – Practitioner Level (CSTP-P)

Objetivos

Además de lograr todos los objetivos enumerados en los niveles CSTP-Associate, CSTP-Practitioner tiene como meta alcanzar los siguientes objetivos:

- Ayudar a analizar y evaluar los requerimientos, evaluar la “testeabilidad” y el diseño de testing basado en los requerimientos.
- Ayudar a gestionar todos los aspectos del proceso de prueba.
- Proporcionar herramientas para medir y controlar el proceso de prueba.
- Ayudar a tener un mejor control sobre el proceso de ejecución de pruebas.
- Proporcionar las mejores prácticas en la presentación de informes de defectos.
- Ayudar a definir las actividades de prueba, de acuerdo al modelo de ciclo de vida utilizado.
- Ayudar a desarrollar mejores planes de prueba.

Destinatarios

- Personas que tengan activo el nivel de certificación de CSTP-Associate.
- Personas que tengan activo el nivel de certificación de CSTE o CTFL.

Requerimientos de educación formal

1. Finalización de la certificación formal de nivel CSTP-Associate o de una certificación activa de CFTL o CSTE.
2. Cuatro días de entrenamiento de la siguiente manera:

Centro de e-Learning SCEU UTN - BA.

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

- a. Tres días de entrenamiento para cubrir el Área # 3 (Gestión del proceso de prueba), # 4 (Ejecución de prueba), y # 5 (Análisis de requerimientos y pruebas basadas en requerimientos) del BOK.
- b. Un día electivo, que cubra cualquiera de las áreas antes mencionadas u otras áreas de apoyo.

Requerimientos de experiencia laboral

Al menos un año de experiencia en un trabajo relacionado con testing. Este requisito será cumplido por medio de una carta laboral firmada por el gerente o directivo del candidato describiendo el rol específico y responsabilidades durante el período de un año o más.

8.1.1.3 Certified Software Test Professional – Master Level (CSTP-M)

Objetivos

Además de lograr todos los objetivos al que figuran en el nivel CSTP Associate y los del nivel CSTP Practitioner, el nivel CSTP Master tiene como meta alcanzar los siguientes objetivos:

- Ayudar a los profesionales de testing a desarrollar sus habilidades de pruebas de software a través de la educación formal.
- Establecer una habilidad común de acuerdo al BOK.
- Crear un grupo calificado de profesionales de testing.
- Preparar a los candidatos para una amplia gama de tareas de pruebas de software.
- Complementar el entrenamiento formal y programas de formación durante el trabajo.
- Fomentar el reconocimiento profesional y la promoción de las carreras de testing.

Destinatarios

- Cualquier persona nueva en la disciplina de testing.
- Testers de software con experiencia con métodos ad hoc.
- Las personas cuya experiencia en testing es corta (no tienen un punto de vista completo del ciclo de vida).
- Cualquier persona que desee convertirse en un profesional de testing y avanzar en su situación profesional con respecto a la disciplina de pruebas de software.

Requerimientos de educación formal

Centro de e-Learning SCEU UTN - BA.

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

www.sceu.frba.utn.edu.ar/e-learning

Opción 1: Obtener el nivel CSTP-Master como una continuación del nivel CSTP-Practitioner.

1. Cumplir las necesidades educativas de ambos niveles CSTP –Master y CSTP-Practitioner.
2. Tres días de entrenamiento de la siguiente manera:
 - a. Dos días para cubrir las áreas #6 (Automatización de pruebas de software) y #7 (Prueba estática) del BOK.
 - b. Un día de entrenamiento electivo, que puede cubrir cualquiera de las áreas antes mencionadas u otras áreas relacionadas con testing.

Opción 2: Alcanzar el nivel de CSTP-Master sin pasar por el nivel CSTP-Practitioner.

Diez días de formación, en los cuales al menos siete días cubran las siete áreas de conocimiento profesional del BOK. El resto de los diez días pueden ser seleccionados por el candidato para cualquiera de los temas de pruebas de software.

Requerimientos de experiencia laboral

Al menos un año de experiencia en testing. Este año es adicional al año que se requiere para la certificación de nivel CSTP-Practitioner si elige la opción # 1. Este requisito será cumplido por medio de una carta laboral firmada por el gerente o directivo del candidato describiendo el rol específico y responsabilidades durante el período de un año o más.

8.1.1.4 Certified Test Manager (CTM)

Objetivos

La certificación CTM se ha desarrollado sobre la base del conjunto de conocimientos de Administración de Pruebas (TMBOK) para llenar el gap y necesidad de habilidades de gestión requeridas por los gerentes y líderes de testing para administrar eficazmente el proceso, el proyecto y la organización de testing. Busca los siguientes objetivos:

- Desarrollar habilidades de gestión de pruebas a través de la educación formal.
- Establecer habilidades comunes para los gerentes y líderes de prueba en base al TMBOK.
- Preparar a los profesionales de testing, en especial aquellos que han logrado la certificación de nivel CSTP-Master, para puestos de dirección, gerencias y de liderazgo en proyectos de pruebas de software.

- Fomentar el reconocimiento profesional y la promoción de las carreras de aquellos que gestionan proyectos de testing.

Destinatarios

- Cualquier persona que haya trabajado en pruebas por al menos 3 años.
- Las personas con el rol de gerente o líder en las pruebas.
- Los responsables del desarrollo o líderes de desarrollo que deseen trasladarse a una gestión de pruebas o posición de liderazgo en pruebas de software.
- Auditores, inspectores y otros que deben evaluar el producto.
- Las personas que hayan completado satisfactoriamente el nivel de CSTP–Master.

Requerimientos de educación formal

Diez días de entrenamiento formal de la siguiente manera:

1. Siete días para cubrir las siete áreas del TMBOK.
2. Tres días electivos para cubrir cualquier área relacionada con testing o calidad. La formación electiva también puede ser seleccionada para cubrir cualquier área del TMBOK con mayor profundidad.

Requerimientos de experiencia laboral

El candidato debe tener al menos tres años trabajando en testing, incluyendo un año de liderazgo o gestión. Este requisito será cumplido al momento de la certificación por medio de una carta que describa el papel del candidato y las responsabilidades en un período de tres años o más. La carta debe ser firmada por alguno de los siguientes:

1. Supervisor o jefe directo / gerente actual o anterior del candidato
2. El cliente del candidato (si es un profesional independiente)
3. Un compañero de trabajo que ocupe actualmente una certificación de CTM y haya trabajado con el candidato en un proyecto de testing.
4. Múltiples fuentes pueden ser presentadas para cubrir el período de tres años.

8.1.1.5 Certified Software Test Automation Specialist (CSTAS)

Objetivos

La certificación de CSTAS busca principalmente los siguientes objetivos:

- Brindar técnicas y métodos para diseñar pruebas con automatización.

- Expandir las pruebas automatizadas más allá de las pruebas funcionales para incluir otras áreas como pruebas de performance, de carga, administración de pruebas, soporte a pruebas automatizadas y automatización de pruebas de código.

Requerimientos de educación formal

Los candidatos deben completar un programa de estudios de diez días de entrenamiento:

- Cinco días para las cinco áreas clave del [Software Test Automation Body of Knowledge \(STABOK\)](#).
- Dos días para cubrir cualquiera de las dos áreas electivas de STABOK
- Tres días para cubrir cualquier tema de testing apropiado para el candidato.

Requerimientos de experiencia laboral

El candidato deberá demostrar conocimientos de al menos una herramienta de prueba funcional automatizada. El requisito será cumplido por medio de una carta firmada el gerente o directivo, o bien el cliente si el candidato es consultor o independiente.

8.1.1.6 Otras certificaciones

Además de las descriptas en los párrafos anteriores, el IIST ofrece otras certificaciones:

Certified Agile Software Test Professional Practitioner Level (CASTP-P)

Esta certificación se centra en el testing usando metodologías ágiles. Nivel inicial.

Certified Agile Software Test Professional Master Level (CASTP-M)

Esta certificación se centra en el testing usando metodologías ágiles. Nivel avanzado.

Certified Software Quality Manager (CSQM)

Esta certificación está orientada a la gestión de la calidad del software.

Certified Mobile Software Test Professional (CMSTP)

Esta certificación apunta al testing de aplicaciones móviles.

Para más información sobre estas certificaciones recurrir a [IIST].

8.2 International Software Testing Qualifications Board®

ISTQB® (International Software Testing Qualifications Board) es una organización de certificación de software testing que opera a nivel internacional. Fundada en Edimburgo en noviembre de 2002, ISTQB es una asociación sin fines de lucro legalmente registrada en Bélgica.

ISTQB ha creado un esquema de certificación para software testing con éxito mundial.

ISTQB ha emitido más de 750.000 certificaciones en más de 120 países, con una tasa de crecimiento de alrededor de 60.000 certificaciones por año.

El plan se basa en un conjunto de conocimientos (un programa de estudios y un glosario) y las normas de examen que se aplican sistemáticamente en todo el mundo, con exámenes y material de lectura disponibles en varios idiomas.

El ISTQB suministra el Programa de Estudio (Syllabus) y el Glosario, en los cuales se definen los estándares internacionales por nivel y se establecen las guías para la acreditación y evaluación de los profesionales de software testing a cargo de los comités de cada país.

8.2.1 Niveles de certificación

El ISTQB contempla tres niveles de certificación.

8.2.1.1 *Foundation Level*

Este es el nivel básico o nivel de inicio, para el cual no se exige experiencia en testing, sin embargo es necesario comprender los conceptos generales sobre los tipos de aplicaciones. El objetivo de este nivel es dar a conocer los fundamentos y los conceptos clave relacionados con las pruebas de software. Igualmente, proporciona al profesional en testing un medio para obtener crecimiento en esta disciplina. Dentro de los temas contemplados en el curso de preparación para la certificación en este nivel están: los objetivos básicos del testing, la psicología del testing, cómo se incorporan las pruebas en el ciclo de desarrollo de software, (los niveles de pruebas), técnicas estáticas y dinámicas de prueba, técnicas de diseño de casos de prueba, la gestión de las pruebas y una

clasificación genérica de las herramientas para pruebas. Es la más difundida de todas las certificaciones.

8.2.1.2 Advanced Level

Este es un nivel avanzado para el cual se requiere 5 años de experiencia en testing y haber cumplido la certificación en el Foundation Level. El objetivo de este nivel de certificación es medir la coherencia entre la experiencia y la ejecución de los proyectos de testing por parte del analista. Para el nivel avanzado se definen tres tipos de certificación: Technical Test Analyst, Test Analyst y Test Manager.

Jefe de Pruebas Nivel Avanzado (Test Manager):

El “Jefe de Pruebas Nivel Avanzado (Test Manager)” deberá ser capaz de:

- Definir objetivos y estrategias globales de pruebas.
- Planificar, programar y seguir tareas.
- Describir y organizar las actividades de pruebas necesarias.
- Seleccionar, adquirir y asignar recursos a cada tarea.
- Organizar y dirigir equipos de pruebas.
- Organizar la comunicación entre los miembros del equipo de pruebas, y entre equipos de pruebas distribuidos.
- Justificar decisiones y proporcionar la información adecuada.

Analista de Pruebas Nivel Avanzado (Test Analyst):

El “Analista de Pruebas Nivel Avanzado (Test Analyst)” deberá ser capaz de:

- Estructurar las tareas definidas en la estrategia de pruebas en términos de requisitos técnicos.
- Analizar la estructura interna del sistema con el suficiente detalle para cumplir con el nivel de calidad esperado.
- Evaluar el sistema en términos de atributos técnicos de calidad, tales como rendimiento, seguridad, etc.
- Proporcionar los elementos necesarios para apoyar las evaluaciones.

Analista Técnico de Pruebas Nivel Avanzado (Technical Test Analyst):

El “Analista Técnico de Pruebas Nivel Avanzado (Technical Test Analyst)” deberá ser capaz de:

- Estructurar tareas definidas en la estrategia de pruebas.
- Analizar sistemas en detalle de manera que cumplan las expectativas de calidad del cliente.
- Evaluar requerimientos.
- Elaborar y ejecutar las tareas de pruebas adecuadas e informar sobre sus progresos.
- Proporcionar los elementos necesarios para apoyar los informes de progreso.
- Implementar las herramientas y técnicas necesarias para alcanzar los objetivos definidos.

8.2.1.3 Expert Level

Este nivel es más avanzado. Amplía el conocimiento obtenido en Advanced Level al proporcionar certificaciones exhaustivas y orientadas a la práctica en una variedad de diferentes temas de testing.

Este nivel actualmente tiene dos módulos:

- Mejorando el proceso de prueba.
- Gestión de pruebas.

Se tiene pensado agregar más módulos en el futuro.

Para obtener la certificación de nivel experto, los candidatos deben:

- Tener la certificación Foundation Level.
- Tener una certificación Advanced Level según el módulo Expert Level deseado.
- Aprobar el examen de nivel Expert Level.
- Tener al menos 5 años de experiencia en testing.
- Tener al menos 2 años de experiencia en el tema específico de Expert Level.



Bibliografía utilizada y sugerida

Libros y otros manuscritos

- [Ast04] Astels, D. Test Driven Development: A Practical Guide. 2004.
- [Bec04] Beck, K. Test-Driven Development: By Example. Second Edition. 2004.
- [Eve07] Everett, Gerald & McLeod, Raymond. Software Testing - Testing Across The Entire Software Development LifeCycle. 2007.
- [Far08] Farrell-Vinay, Peter. Manage Software Testing. 2008.
- [IIST] International Institute of Software Testing <http://www.iist.org>
- [ISTQB] International Software Testing Qualifications Board <http://www.istqb.org>
- [Jor14] Jorgensen, P. Software Testing - A Craftsman's Approach. 4th Edition. 2014.
- [Mye12] Myers, Glenford, Badgett, T. y Sandler, C. The Art of Software Testing. Third Edition. 2012.
- [Pre19] Pressman, Roger y Maxim, B. Software Engineering: A Practitioner's Approach. Ninth Edition. 2019.
- [SSTQB] Spanish Software Testing Qualifications Board <http://www.sstqb.es>

Lo que vimos:

En esta última Unidad hemos abordado los principales conceptos de testing que habían quedado pendientes de las Unidades anteriores.

Esto incluyó distintos tipos de tests, como regresión, smoke, alfa, beta, y usabilidad. También se introdujeron los puntos clave sobre planificación y se ha desarrollado con cierto nivel de detalle el tema de automatización y herramientas de testing.

Hacia el final de la Unidad se ha presentado TDD (Test Driven Development), una moderna y muy interesante metodología de desarrollo de software, junto con su relación con la metodología de desarrollo ágil XP (Extreme Programming).

Para cerrar la Unidad y el Curso, se han descripto dos de las certificaciones de testing más importantes a nivel mundial, esto es IIST, ISTQB.

