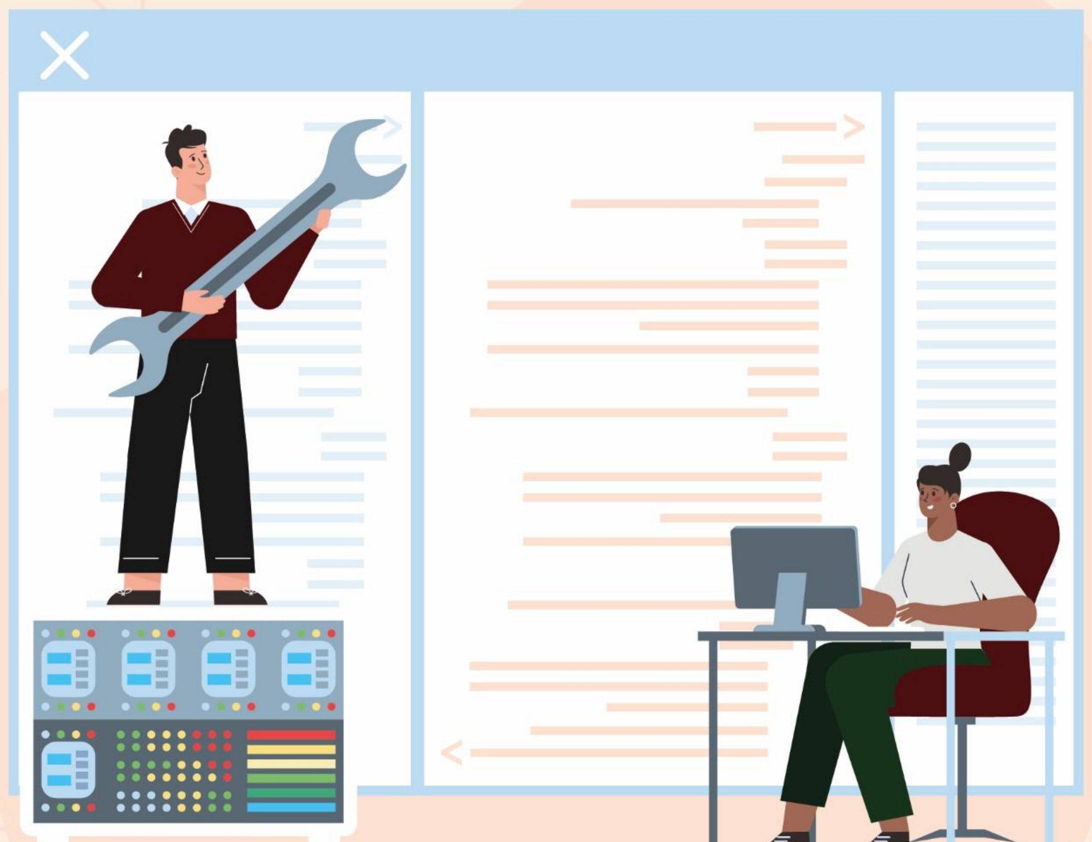
Ingeniería del Software II

Implementación, Pruebas y Mantenimiento

Ricardo Javier Celi Párraga
Miguel Fabricio Boné Andrade
Aldo Patricio Mora Olivero
Juan Carlos Sarmiento Saavedra



EDITORIAL GRUPO AEA

Ingeniería del Software II

Implementación, Pruebas y Mantenimiento

Autor/es:

Ricardo Javier Celi-Párraga

Aldo Patricio Mora-Olivero

Miguel Fabricio Boné-Andrade

Juan Carlos Sarmiento-Saavedra

Título del libro:
Ingeniería del Software II: Implementación, Pruebas y Mantenimiento.
Primera Edición, 2023
Editado en Santo Domingo, Ecuador, 2023
ISBN: 978-9942-7014-8-0

© Abril, 2023
© Editorial Grupo AEA, Santo Domingo - Ecuador
© Celi Párraga Ricardo Javier, Boné Andrade Miguel Fabricio, Mora Olivero Aldo Patricio, Sarmiento Saavedra Juan Carlos.
Editado y diseñado por Comité Editorial del Grupo AEA
Hecho e impreso en Santo Domingo - Ecuador

Cita.

Celi-Párraga, R. J., Boné-Andrade, M. F., Mora-Olivero, A. P., Sarmiento-Saavedra J. C. (2023). Ingeniería del Software II: Implementación, Pruebas y Mantenimiento. Editorial Grupo AEA.

Cada uno de los textos de la Editorial Grupo AEA han sido sometido a un proceso de evaluación por pares doble ciego externos (double-blindpaperreview) con base en la normativa del editorial.

Grupo AEA

Grupo de Asesoría Empresarial y Académica

www.grupo-aea.com

www.editorialgrupo-aea.com

Grupo de Asesoría Empresarial & Académica

Grupoaea.ecuador

Editorial Grupo AEA

Aviso Legal:

La información presentada, así como el contenido, fotografías, gráficos, cuadros, tablas y referencias de este manuscrito es de exclusiva responsabilidad del autor y no necesariamente reflejan el pensamiento de la Editorial Grupo AEA.

Derechos de autor ©

Este documento se publica bajo los términos y condiciones de la licencia Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0).

El "copyright" y todos los derechos de propiedad intelectual y/o industrial sobre el contenido de esta edición son propiedad de la Editorial Grupo AEA y sus Autores. Se prohíbe rigurosamente, bajo las sanciones en las leyes, la producción o almacenamiento total y/o parcial de esta obra, ni su tratamiento informático de la presente publicación, incluyendo el diseño de la portada, así como la transmisión de la misma de ninguna forma o por cualquier medio, tanto si es electrónico, como químico, mecánico, óptico, de grabación o bien de fotocopia, sin la autorización de los titulares del copyright, salvo cuando se realice confines académicos o científicos y estrictamente no comerciales y gratuitos, debiendo citar en todo caso a la editorial.

Autores

Reseña de Autores

Ricardo Javier Celi Párraga

Universidad Técnica Luis Vargas Torres de Esmeraldas

Correo: ricardo.celi@utelvt.edu.ec

Orcid: https://orcid.org/0000-0002-8525-5744

Ingeniero en sistemas informáticos, Máster en ingeniería del software y sistemas informáticos, Docente de La Universidad Técnica Luis Vargas Torres de Esmeraldas, Sede Santo Domingo de los Tsáchilas.

Aldo Patricio Mora Olivero

Universidad Técnica Luis Vargas Torres de Esmeraldas

Correo: aldo.mora.olivero@utelvt.edu.ec

Orcid: https://orcid.org/0000-0002-4337-7452

Ingeniero en sistemas y computación, Magíster en tecnologías de la información, Docente de La Universidad Técnica Luis Vargas Torres de Esmeraldas, Sede Santo Domingo de los Tsáchilas.

Miguel Fabricio Boné Andrade

Universidad Técnica Luis Vargas Torres de Esmeraldas

Correo: miguel.bone@utelvt.edu.ec

Orcid: https://orcid.org/0000-0002-8635-1869

Ingeniero de sistemas y computación, Magíster en sistemas de telecomunicaciones, Magíster en tecnologías de la información mención en seguridad de redes y comunicaciones, Docente de La Universidad Técnica Luis Vargas Torres de Esmeraldas, Sede Santo Domingo de los Tsáchilas.

Juan Carlos Sarmiento Saavedra

Universidad Técnica Luis Vargas Torres de Esmeraldas

Correo: juan.sarmiento@utelvt.edu.ec

Orcid: http://orcid.org/0000-0001-8114-9410

Ingeniero en sistemas e informática, Magíster en docencia y desarrollo del currículo, Magíster en tecnologías de la información, Docente de La Universidad Técnica Luis Vargas Torres de Esmeraldas, Sede Santo Domingo de los Tsáchilas.

Índice

Reseña de Autores IV

Índice V

Introducción VII

Capítulo I: Implementación del Software 1

1.1. Conflictos de implementación 3

1.2. Administración de la configuración 6

1.3. Desarrollo de código abierto 9

Capítulo II: Pruebas y Validación del Software 13

2.1. Validación de software 18

2.2. Pruebas de desarrollo 18

2.2.1. Pruebas de unidad 21

2.2.2. Pruebas de componentes 26

2.2.3. Pruebas de interfaz 28

2.2.4. Pruebas del sistema 29

2.3. Desarrollo dirigido por pruebas 31

2.4. Pruebas de versión 34

2.5. Pruebas basadas en requerimientos 35

2.6. Pruebas de rendimiento 38

2.7. Pruebas de usuario 40

2.8. Pruebas de seguridad 44

2.9. Pruebas de despliegue 48

2.10. Pruebas de usabilidad 49

2.11. Pruebas de compatibilidad 53

Capítulo III: Evolución, Confiabilidad y Seguridad Del Software 57

3.1. Proceso de evolución 62

3.2. Evolución dinámica del programa 64

3.3. Mantenimiento del software 67

3.4. Administración de sistemas heredados.. 73

3.5. Propiedades de confiabilidad 76

3.5.1. Disponibilidad y fiabilidad 79

3.5.2. Protección 80

3.5.3. Seguridad.. 81

Capítulo IV: Metodologías de Desarrollo de Aplicaciones Web Seguras......... 85

4.1. OWASP: Open Web Application Security Project............................. 87

4.2. Open-Source Security Testing Methodology Manual (OSSTMM)...... 91

Referencias Bibliográficas.. 95

Introducción

La ingeniería de software es una disciplina que se encarga de aplicar principios y metodologías para el desarrollo de software de manera eficiente y efectiva. Uno de los aspectos más importantes de esta disciplina es la implementación, que se refiere al proceso de transformar el diseño del software en código ejecutable.

Una vez que el software ha sido implementado, es necesario someterlo a pruebas para asegurarse de que funcione correctamente y cumpla con los requerimientos especificados en el diseño. Las pruebas pueden ser de diferentes tipos, desde pruebas unitarias hasta pruebas de integración y aceptación.

Finalmente, una vez que el software ha sido implementado y probado, es importante mantenerlo y actualizarlo de forma regular para asegurarse de que siga funcionando correctamente y se adapte a las nuevas necesidades del negocio o de los usuarios. El mantenimiento del software puede incluir actividades como corrección de errores, actualizaciones de seguridad y mejoras de rendimiento.

En resumen, la implementación, pruebas y mantenimiento son aspectos clave de la ingeniería de software que permiten garantizar la calidad y confiabilidad del software.

Capítulo I: Implementación del Software

Implementación del Software

La implementación es parte del proceso en el que los ingenieros de software programan el código para el proyecto de trabajo que está en relación de las demandas del software, en esta etapa se realizan las pruebas de caja blanca y caja negra.

1.1. Conflictos de implementación

La ingeniería de software incluye todas las actividades implicadas en el desarrollo de software, desde los requerimientos iniciales del sistema hasta el mantenimiento y la administración del sistema desplegado. Una etapa crítica de este proceso es, desde luego, la implementación del sistema, en la cual se crea una versión ejecutable del software. La implementación quizá requiera el desarrollo de programas en lenguajes de programación de alto o bajo niveles, o bien, la personalización y adaptación de sistemas comerciales genéricos para cubrir los requerimientos específicos de una organización.

Se supone a este nivel los estudiantes comprenderán los principios de programación y tendrán alguna experiencia al respecto. Como este capítulo tiene la intención de ofrecer un enfoque independiente de lenguaje, no se centró en conflictos de la buena práctica de programación, pues para esto se tendrían que usar ejemplos específicos de lenguaje. En su lugar, se introducen algunos aspectos de implementación que son muy importantes para la ingeniería de software que, por lo general, no se tocan en los textos de programación.

Éstos son:

Reutilización

La mayoría del software moderno se construye por la reutilización de los componentes o sistemas existentes. Cuando se desarrolla software, debe usarse el código existente tanto como sea posible.

De la década de 1970 a la de 1990, gran parte del nuevo software se desarrolló desde cero, al escribir todo el código en un lenguaje de programación de alto nivel. La única reutilización o software significativo era la reutilización de funciones y objetos en las librerías de lenguaje de programación. Sin embargo, los costos y la presión por fechas significaban que este enfoque se volvería cada vez más inviable, sobre todo para sistemas comerciales y basados en Internet. En consecuencia, surgió un enfoque al desarrollo basado en la reutilización del software existente y ahora se emplea generalmente para sistemas empresariales, software científico y, cada vez más, en ingeniería de sistemas embebidos.

La reutilización de software es posible en algunos niveles diferentes:

1. El nivel de abstracción: En este nivel no se reutiliza el software directamente, sino más bien se utiliza el conocimiento de abstracciones exitosas en el diseño de su software. Los patrones de diseño y los arquitectónicos son vías de representación del conocimiento abstracto para la reutilización.
2. El nivel objeto: En este nivel se reutilizan directamente los objetos de una librería en vez de escribir uno mismo en código. Para implementar este tipo de reutilización, se deben encontrar librerías adecuadas y descubrir si los objetos y métodos ofrecen la funcionalidad que se necesita. Por ejemplo, si usted requiere procesar mensajes de correo en un programa Java, tiene que usar objetos y métodos de una librería JavaMail.
3. El nivel componente: Los componentes son colecciones de objetos y clases de objetos que operan en conjunto para brindar funciones y servicios relacionados. Con frecuencia se debe adaptar y extender el componente al agregar por cuenta propia cierto código. Un ejemplo de reutilización a nivel componente es donde usted construye su interfaz de usuario mediante un marco. Éste es un conjunto de clases de objetos generales que aplica manipulación de eventos, gestión de despliegue, etcétera. Agrega conexiones a los datos a desplegar y escribe el código para definir detalles de despliegue específicos, como plantilla de la pantalla y colores.

4. El nivel sistema: En este nivel se reutilizan sistemas de aplicación completos. Usualmente esto implica cierto tipo de configuración de dichos sistemas. Puede hacerse al agregar y modificar el código (si reutiliza una línea de producto de software) o al usar la interfaz de configuración característica del sistema. La mayoría de los sistemas comerciales se diseñan ahora de esta forma, donde se adapta y reutilizan sistemas COTS (comerciales) genéricos. A veces este enfoque puede incluir la reutilización de muchos sistemas diferentes e integrarlos para crear un nuevo sistema.

Al reutilizar el software existente, es factible desarrollar nuevos sistemas más rápidamente, con menos riesgos de desarrollo y también costos menores. Puesto que el software reutilizado se probó en otras aplicaciones, debe ser más confiable que el software nuevo. Sin embargo, existen costos asociados con la reutilización:

- Los costos del tiempo empleado en la búsqueda del software para reutilizar y valorar si cubre sus necesidades o no. Es posible que deba poner a prueba el software para asegurarse de que funcionará en su entorno, especialmente si éste es diferente de su entorno de desarrollo.
- Donde sea aplicable, los costos por comprar el software reutilizable. Para sistemas comerciales grandes, dichos costos suelen ser muy elevados.
- Los costos por adaptar y configurar los componentes de software o sistemas reutilizables, con la finalidad de reflejar los requerimientos del sistema que se desarrolla.
- Los costos de integrar elementos de software reutilizable unos con otros (si usa software de diferentes fuentes) y con el nuevo código que haya desarrollado. Integrar software reutilizable de diferentes proveedores suele ser difícil y costoso, ya que los proveedores podrían hacer conjeturas conflictivas sobre cómo se reutilizará su software respectivo.

Cómo reutilizar el conocimiento y el software existentes sería el primer punto a considerar cuando se inicie un proyecto de desarrollo de software. Hay que contemplar las posibilidades de reutilización antes de diseñar el software a

detalle, pues tal vez usted quiera adaptar su diseño para reutilización de los activos de software existentes.

Para un gran número de sistemas de aplicación, la ingeniería de software significa en realidad reutilización de software.

1.2. Administración de la configuración

Durante el proceso de desarrollo se crean muchas versiones diferentes de cada componente de software. Si usted no sigue la huella de dichas versiones en un sistema de gestión de configuración, estará proclive a incluir en su sistema las versiones equivocadas de dichos componentes.

En el desarrollo de software, los cambios ocurren todo el tiempo, de modo que la administración del cambio es absolutamente esencial. Cuando un equipo de individuos desarrolla software, hay que cerciorarse de que los miembros del equipo no interfieran con el trabajo de los demás. Esto es, si dos personas trabajan sobre un componente, los cambios deben coordinarse. De otro modo, un programador podría realizar cambios y sobrescribir en el trabajo de otro.

También se debe garantizar que todos tengan acceso a las versiones más actualizadas de componentes de software; de lo contrario, los desarrolladores pueden rehacer lo ya hecho. Cuando algo salga mal con una nueva versión de un sistema, se debe poder retroceder a una versión operativa del sistema o componente.

Administración de la configuración es el nombre dado al proceso general de gestionar un sistema de software cambiante. La meta de la administración de la configuración es apoyar el proceso de integración del sistema, de modo que todos los desarrolladores tengan acceso en una forma controlada al código del proyecto y a los documentos, descubrir qué cambios se realizaron, así como compilar y vincular componentes para crear un sistema.

Por lo tanto, hay tres actividades fundamentales en la administración de la configuración:

- **Gestión de versiones**, donde se da soporte para hacer un seguimiento de las diferentes versiones de los componentes de software. Los sistemas de gestión de versiones incluyen facilidades para que el desarrollo esté coordinado por varios programadores. Esto evita que un desarrollador sobrescriba un código que haya sido enviado al sistema por alguien más.
- **Integración de sistema**, donde se da soporte para ayudar a los desarrolladores a definir qué versiones de componentes se usan para crear cada versión de un sistema. Luego, esta descripción se utiliza para elaborar automáticamente un sistema al compilar y vincular los componentes requeridos.
- **Rastreo de problemas**, donde se da soporte para que los usuarios reporten bugs y otros problemas, y también para que todos los desarrolladores sepan quién trabaja en dichos problemas y cuándo se corrigen.

Desarrollo huésped-objetivo

La producción de software no se ejecuta por lo general en la misma computadora que el entorno de desarrollo de software. En vez de ello, se diseña en una computadora (el sistema huésped) y se ejecuta en una computadora separada (el sistema objetivo). Los sistemas huésped y objetivo son algunas veces del mismo tipo, aunque suelen ser completamente diferentes.

La mayoría del desarrollo de software se basa en un modelo huésped-objetivo. El software se desarrolla en una computadora (el huésped), aunque opera en una máquina separada (el objetivo). En un sentido más amplio, puede hablarse de una plataforma de desarrollo y una plataforma de ejecución. Una plataforma es más que sólo hardware. Incluye el sistema operativo instalado más otro software de soporte como un sistema de gestión de base de datos o, para plataformas de desarrollo, un entorno de desarrollo interactivo.

En ocasiones, las plataformas de desarrollo y ejecución son iguales, lo que posibilita diseñar el software y ponerlo a prueba en la misma máquina. Sin embargo, es más común que sean diferentes, de modo que es necesario mover

el software desarrollado a la plataforma de ejecución para ponerlo a prueba, u operar un simulador en su máquina de desarrollo.

Una plataforma de desarrollo de software debe ofrecer una variedad de herramientas para soportar los procesos de ingeniería de software.

Éstas pueden incluir:

- Un compilador integrado y un sistema de edición dirigida por sintaxis que le permitan crear, editar y compilar código.
- Un sistema de depuración de lenguaje.
- Herramientas de edición gráfica, tales como las herramientas para editar modelos UML.

Las herramientas de desarrollo de software se agrupan con frecuencia para crear un entorno de desarrollo integrado (IDE), que es un conjunto de herramientas de software que apoyan diferentes aspectos del desarrollo de software, dentro de cierto marco común e interfaz de usuario. Por lo común, los IDE se crean para apoyar el desarrollo en un lenguaje de programación específico, como Java. El lenguaje IDE puede elaborarse especialmente, o ser una ejemplificación de un IDE de propósito general, con herramientas de apoyo a lenguaje específico.

Como parte del proceso de desarrollo, se requiere tomar decisiones sobre cómo se desplegará el software desarrollado en la plataforma objetivo. Esto es directo para sistemas embebidos, donde el objetivo es usualmente una sola computadora. Sin embargo, para sistemas distribuidos, es necesario decidir sobre las plataformas específicas donde se desplegarán los componentes.

Los conflictos que hay que considerar al tomar esta decisión son:

1. **Los requerimientos de hardware y software de un componente**: Si un componente se diseña para una arquitectura de hardware específica, o se apoya en algún otro sistema de software, tiene que desplegarse por supuesto en una plataforma que brinde el soporte requerido de hardware y software.
2. **Los requerimientos de disponibilidad del sistema**: Los sistemas de alta disponibilidad pueden necesitar que los componentes se desplieguen en más de una plataforma. Esto significa que, en el caso de una falla de

plataforma, esté disponible una implementación alternativa del componente.

3. **Comunicaciones de componentes**: Si hay un alto nivel de tráfico de comunicaciones entre componentes, por lo general tiene sentido desplegarlos en la misma plataforma o en plataformas que estén físicamente cercanas entre sí. Esto reduce la latencia de comunicaciones, es decir, la demora entre el tiempo que transcurre desde el momento en que un componente envía un mensaje hasta que otro lo recibe.

1.3. Desarrollo de código abierto

El desarrollo de código abierto es un enfoque al desarrollo de software en que se publica el código de un sistema de software y se invita a voluntarios a participar en el proceso de desarrollo (Raymond, 2001). Sus raíces están en la Free Software Foundation (http:// www.fsf.org), que aboga por que el código fuente no debe ser propietario sino, más bien, tiene que estar siempre disponible para que los usuarios lo examinen y modifiquen como deseen. Existía la idea de que el código estaría controlado y sería desarrollado por un pequeño grupo central, en vez de por usuarios del código.

El software de código abierto extendió esta idea al usar Internet para reclutar a una población mucho mayor de desarrolladores voluntarios. La mayoría de ellos también son usuario s del código. En principio al menos, cualquier contribuyente a un proyecto de código abierto puede reportar y corregir bugs, así como proponer nuevas características y funcionalidades. Sin embargo, en la práctica, los sistemas exitosos de código abierto aún se apoyan en un grupo central de desarrolladores que controlan los cambios al software.

Desde luego, el producto mejor conocido de código abierto es el sistema operativo Linux, utilizado ampliamente como sistema servidor y, cada vez más, como un entorno de escritorio. Otros productos de código abierto importantes son Java, el servidor Web Apache y el sistema de gestión de base de datos mySQL. Los protagonistas principales en la industria de cómputo, como IBM y Sun, soportan el movimiento de código abierto y basan su software en productos

de código abierto. Existen miles de otros sistemas y componentes de código abierto menos conocidos que también podrían usarse.

Por lo general, es muy barato o incluso gratuito adquirir software de código abierto. Usualmente, el software de código abierto se descarga sin costo. Sin embargo, si usted quiere documentación y soporte, entonces tal vez deba pagar por ello; aun así, los costos son por lo común bastante bajos. El otro beneficio clave para usar productos de código abierto es que los sistemas de código abierto mayores son casi siempre muy confiables.

Para una compañía que desarrolla software, existen dos conflictos de código abierto que debe considerar:

- ¿El producto que se desarrollará deberá usar componentes de código abierto?
- ¿Deberá usarse un enfoque de código abierto para el desarrollo del software?

Las respuestas a dichas preguntas dependen del tipo de software que se desarrollará, así como de los antecedentes y la experiencia del equipo de desarrollo. Si usted diseña un producto de software para su venta, entonces resultan críticos tanto el tiempo en que sale al mercado como la reducción en costos.

Si se desarrolla en un dominio donde estén disponibles sistemas en código abierto de alta calidad, puede ahorrar tiempo y dinero al usar dichos sistemas. Sin embargo, si usted desarrolla software para un conjunto específico de requerimientos organizativos, entonces quizás el uso de componentes de código abierto no sea una opción. Tal vez tenga que integrar su software con sistemas existentes que sean compatibles con los sistemas en código abierto disponibles. No obstante, incluso entonces podría ser más rápido y barato modificar el sistema en código abierto, en vez de volver a desarrollar la funcionalidad que necesita.

Cada vez más compañías de productos usan un enfoque de código abierto para el desarrollo. Sus modelos empresariales no dependen de la venta de un producto de software, sino de la comercialización del soporte para dicho

producto. Consideran que involucrar a la comunidad de código abierto permitirá que el software se desarrolle de manera más económica, más rápida y creará una comunidad de usuarios para el software. A pesar de ello, de nuevo, esto sólo es aplicable realmente para productos de software en general, y no para aplicaciones específicas de la organización.

Publicar el código de un sistema no significa que la comunidad en general necesariamente ayudará con su desarrollo. Los productos más exitosos de código abierto han sido productos de plataforma, en vez de sistemas de aplicación. Hay un número limitado de desarrolladores que pueden interesarse en sistemas de aplicación especializados. En sí, elaborar un sistema de software en código abierto no garantiza la inclusión de la comunidad.

Licencia de código abierto

Aunque un principio fundamental del desarrollo en código abierto es que el código fuente debe estar disponible por entero, esto no significa que cualquiera puede hacer lo que desee con el código. Por ley, el desarrollador del código (una compañía o un individuo) todavía es propietario del código. Puede colocar restricciones sobre cómo se le utiliza al incluir condiciones legales en una licencia de software de código abierto (St. Laurent, 2004).

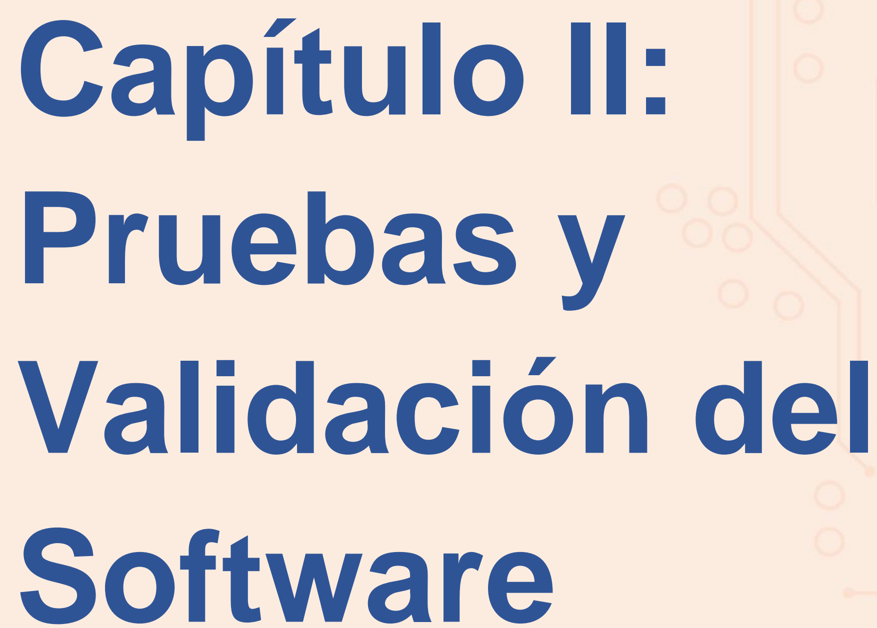
Algunos desarrolladores de código abierto creen que si un componente de código abierto se usa para desarrollar un nuevo sistema, entonces dicho sistema también debe ser de código abierto. Otros están satisfechos de que su código se use sin esta restricción. Los sistemas desarrollados pueden ser propietarios y venderse como sistemas de código cerrado.

La mayoría de las licencias de código abierto se derivan de uno de tres modelos generales:

1. La licencia pública general GNU se conoce como licencia “recíproca”; de manera simple, significa que si usted usa software de código abierto que esté permitido bajo la licencia GPL, entonces debe hacer que dicho software sea de código abierto.

2. La licencia pública menos general GNU es una variante de la licencia anterior, en la que usted puede escribir componentes que se vinculen con el código abierto, sin tener que publicar el código de dichos componentes. Sin embargo, si cambia el componente permitido, entonces debe publicar éste como código abierto.
3. La licencia Berkeley Standard Distribution es una licencia no recíproca, lo cual significa que usted no está obligado a volver a publicar algún cambio o modificación al código abierto. Puede incluir el código en sistemas propietarios que se vendan. Si usa componentes de código abierto, debe reconocer al creador original del código.

El modelo empresarial de software está cambiando. Se ha vuelto cada vez más difícil edificar una empresa mediante la venta de sistemas de software especializado. Muchas compañías prefieren hacer su software en código abierto y entonces vender soporte y consultoría a los usuarios del software. Es probable que esta tendencia se incremente, con el uso creciente de software de código abierto y con cada vez más software disponible de esta forma.



Capítulo II: Pruebas y Validación del Software

Pruebas y Validación del Software

Las pruebas intentan demostrar que un programa hace lo que se intenta que haga, así como descubrir defectos en el programa antes de usarlo. Al probar el software, se ejecuta un programa con datos artificiales. Hay que verificar los resultados de la prueba que se opera para buscar errores, anomalías o información de atributos no funcionales del programa.

El proceso de prueba tiene dos metas distintas:

1. Demostrar al desarrollador y al cliente que el software cumple con los requerimientos. Para el software personalizado, esto significa que en el documento de requerimientos debe haber, por lo menos, una prueba por cada requerimiento. Para los productos de software genérico, esto quiere decir que tiene que haber pruebas para todas las características del sistema, junto con combinaciones de dichas características que se incorporarán en la liberación del producto
2. Encontrar situaciones donde el comportamiento del software sea incorrecto, indeseable o no esté de acuerdo con su especificación. Tales situaciones son consecuencia de defectos del software. La prueba de defectos tiene la finalidad de erradicar el comportamiento indeseable del sistema, como caídas del sistema, interacciones indeseadas con otros sistemas, cálculos incorrectos y corrupción de datos.

Nombre: Crear Usuarios

Descripción: COMO administrador QUIERO ingresar, modificar la información de usuarios PARA asignarles un perfil con sus respectivos privilegios.
Escenarios de prueba: Dado la creación de usuarios CUANDO se pulse el botón guardar, ENTONCES se presenta el mensaje de creación correcta. Dado el ingreso de datos incorrectos en la creación de usuarios CUANDO se pulse el botón guardar, ENTONCES se presenta el mensaje de alerta.

Nota: Fuente: Freire, C. & Eras, S. (2017). PUCE SD.

La primera meta conduce a la prueba de validación; en ella, se espera que el sistema se desempeñe de manera correcta mediante un conjunto dado de casos de prueba, que refleje el uso previsto del sistema. La segunda meta se orienta a pruebas de defectos, donde los casos de prueba se diseñan para presentar los defectos. Las pruebas no pueden demostrar que el software esté exento de defectos o que se comportará como se especifica en cualquier circunstancia. Siempre es posible que una prueba que usted pase por alto descubra más problemas con el sistema.

Las pruebas pueden mostrar sólo la presencia de errores, mas no su ausencia.

Las pruebas se consideran parte de un proceso más amplio de verificación y validación (V&V) del software. Aunque ambas no son lo mismo, se confunden con frecuencia. Barry Boehm, pionero de la ingeniería de software, expresó de manera breve la diferencia entre las dos (Boehm, 1979):

- "Validación: ¿construimos el producto correcto?".
- "Verificación: ¿construimos bien el producto?".

Los procesos de verificación y validación buscan comprobar que el software por desarrollar cumpla con sus especificaciones, y brinde la funcionalidad deseada por las personas que pagan por el software. Dichos procesos de comprobación comienzan tan pronto como están disponibles los requerimientos y continúan a través de todas las etapas del proceso de desarrollo.

La finalidad de la verificación es comprobar que el software cumpla con su funcionalidad y con los requerimientos no funcionales establecidos. Sin embargo, la validación es un proceso más general.

La meta de la validación es garantizar que el software cumpla con las expectativas del cliente. Va más allá del simple hecho de comprobar la conformidad con la especificación, para demostrar que el software hace lo que el cliente espera que haga.

Por lo general, un sistema de software comercial debe pasar por tres etapas de pruebas:

1. **Pruebas de desarrollo**, donde el sistema se pone a prueba durante el proceso para descubrir errores (bugs) y defectos. Es probable que en el desarrollo de prueba intervengan diseñadores y programadores del sistema.
2. **Versiones de prueba**, donde un equipo de prueba por separado experimenta una versión completa del sistema, antes de presentarlo a los usuarios. La meta de la prueba de versión es comprobar que el sistema cumpla con los requerimientos de los participantes del sistema.
3. **Pruebas de usuario**, donde los usuarios reales o potenciales de un sistema prueban el sistema en su propio entorno. Para productos de software, el “usuario” puede ser un grupo interno de marketing, que decide si el software se comercializa, se lanza y se vende. Las pruebas de aceptación se efectúan cuando el cliente prueba de manera formal un sistema para decidir si debe aceptarse del proveedor del sistema, o si se requiere más desarrollo.

En la práctica, el proceso de prueba por lo general requiere una combinación de pruebas manuales y automatizadas. En las primeras pruebas manuales, un examinador opera el programa con algunos datos de prueba y compara los resultados con sus expectativas. Anota y reporta las discrepancias con los desarrolladores del programa. En las pruebas automatizadas, éstas se codifican en un programa que opera cada vez que se prueba el sistema en desarrollo.

Comúnmente esto es más rápido que las pruebas manuales, sobre todo cuando incluye pruebas de regresión, es decir, aquellas que implican volver a correr pruebas anteriores para comprobar que los cambios al programa no introdujeron nuevos bugs.

El uso de pruebas automatizadas aumentó de manera considerable durante los últimos años. Sin embargo, las pruebas nunca pueden ser automatizadas por completo, ya que esta clase de pruebas sólo comprueban que un programa haga lo que supone que tiene que hacer. Es prácticamente imposible usar pruebas automatizadas para probar sistemas que dependan de cómo se ven las cosas

(por ejemplo, una interfaz gráfica de usuario) o probar que un programa no presenta efectos colaterales indeseados.

2.1. Validación de software

La validación de software o, más generalmente, su verificación y validación (V&V), se crea para mostrar que un sistema cumple tanto con sus especificaciones como con las expectativas del cliente. Las pruebas del programa, donde el sistema se ejecuta a través de datos de prueba simulados, son la principal técnica de validación. Esta última también puede incluir procesos de comprobación, como inspecciones y revisiones en cada etapa del proceso de software, desde la definición de requerimientos del usuario hasta el desarrollo del programa. Dada la predominancia de las pruebas, se incurre en la mayoría de los costos de validación durante la implementación y después de ésta.

Con excepción de los programas pequeños, los sistemas no deben ponerse a prueba como una unidad monolítica. De manera ideal, los defectos de los componentes se detectan oportunamente en el proceso, en tanto que los problemas de interfaz se localizan cuando el sistema se integra. Sin embargo, conforme se descubran los defectos, el programa deberá depurarse y esto quizá requiera la repetición de otras etapas en el proceso de pruebas. Los errores en los componentes del programa pueden salir a la luz durante las pruebas del sistema. En consecuencia, el proceso es iterativo, con información retroalimentada desde etapas posteriores hasta las partes iniciales del proceso.

2.2. Pruebas de desarrollo

Las etapas en el proceso de pruebas son:

1. Prueba de desarrollo: Las personas que desarrollan el sistema ponen a prueba los componentes que constituyen el sistema. Cada componente se prueba de manera independiente, es decir, sin otros componentes del sistema. Éstos pueden ser simples entidades, como funciones o clases de objeto, o agrupamientos coherentes de dichas entidades. Por lo

general, se usan herramientas de automatización de pruebas, como JUnit (Massol y Husted, 2003), que pueden volver a correr pruebas de componentes cuando se crean nuevas versiones del componente.

2. Pruebas del sistema: Los componentes del sistema se integran para crear un sistema completo. Este proceso tiene la finalidad de descubrir errores que resulten de interacciones no anticipadas entre componentes y problemas de interfaz de componente, así como de mostrar que el sistema cubre sus requerimientos funcionales y no funcionales, y poner a prueba las propiedades emergentes del sistema. Para sistemas grandes, esto puede ser un proceso de múltiples etapas, donde los componentes se conjuntan para formar subsistemas que se ponen a prueba de manera individual, antes de que dichos subsistemas se integren para establecer el sistema final.
3. Pruebas de aceptación: Ésta es la etapa final en el proceso de pruebas, antes de que el sistema se acepte para uso operacional. El sistema se pone a prueba con datos suministrados por el cliente del sistema, en vez de datos de prueba simulados. Las pruebas de aceptación revelan los errores y las omisiones en la definición de requerimientos del sistema, ya que los datos reales ejercitan el sistema en diferentes formas a partir de los datos de prueba. Asimismo, las pruebas de aceptación revelan problemas de requerimientos, donde las instalaciones del sistema en realidad no cumplan las necesidades del usuario o cuando sea inaceptable el rendimiento del sistema.

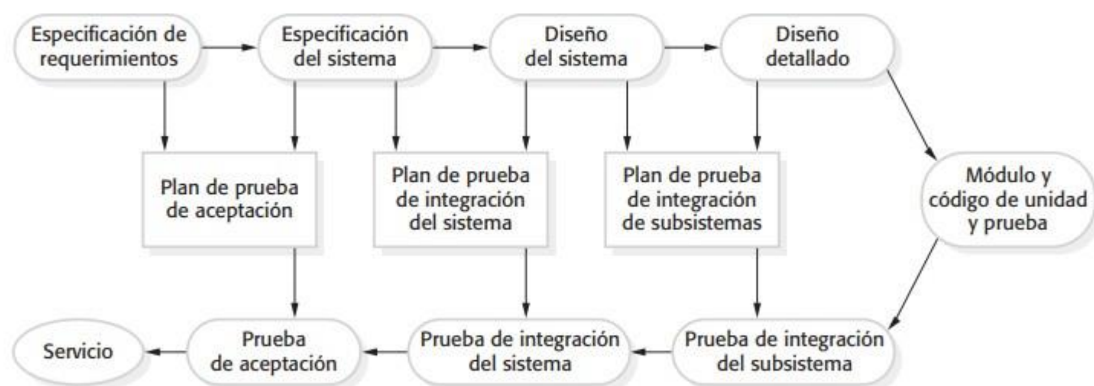
Por lo general, los procesos de desarrollo y de pruebas de componentes están entrelazados. Los programadores construyen sus propios datos de prueba y experimentan el código de manera incremental conforme lo desarrollan. Éste es un enfoque económicamente sensible, ya que el programador conoce el componente y, por lo tanto, es el más indicado para generar casos de prueba.

Si se usa un enfoque incremental para el desarrollo, cada incremento debe ponerse a prueba conforme se diseña, y tales pruebas se basan en los requerimientos para dicho incremento. En programación extrema, las pruebas se desarrollan junto con los requerimientos antes de comenzar el desarrollo. Esto

ayuda a los examinadores y desarrolladores a comprender los requerimientos, y garantiza que no haya demoras conforme se creen casos de prueba.

Cuando se usa un proceso de software dirigido por un plan (como en el desarrollo de sistemas críticos), las pruebas se realizan mediante un conjunto de planes de prueba. Un equipo independiente de examinadores trabaja con base en dichos planes de prueba pre formulados, que se desarrollaron a partir de la especificación y el diseño del sistema.

Probando fases en un proceso de software dirigido por un plan



En ocasiones, a las pruebas de aceptación se les identifica como “pruebas alfa”. Los sistemas a la medida se desarrollan sólo para un cliente. El proceso de prueba alfa continúa hasta que el desarrollador del sistema y el cliente estén de acuerdo en que el sistema entregado es una implementación aceptable de los requerimientos.

Cuando un sistema se marca como producto de software, se utiliza con frecuencia un proceso de prueba llamado “prueba beta”. Ésta incluye entregar un sistema a algunos clientes potenciales que están de acuerdo con usar ese sistema. Ellos reportan los problemas a los desarrolladores del sistema. Dicho informe expone el producto a uso real y detecta errores que no anticiparon los constructores del sistema. Después de esta retroalimentación, el sistema se modifica y libera, ya sea para más pruebas beta o para su venta general.

Pruebas de desarrollo

Las pruebas de desarrollo incluyen todas las actividades de prueba que realiza el equipo que elabora el sistema. El examinador del software suele ser el

programador que diseñó dicho software, aunque éste no es siempre el caso. Algunos procesos de desarrollo usan parejas de programador/examinador donde cada programador tiene un examinador asociado que desarrolla pruebas y auxilia con el proceso de pruebas.

Para sistemas críticos, puede usarse un proceso más formal, con un grupo de prueba independiente dentro del equipo de desarrollo. Son responsables del desarrollo de pruebas y del mantenimiento de registros detallados de los resultados de las pruebas.

Durante el desarrollo, las pruebas se realizan en tres niveles:

1. **Pruebas de unidad**, donde se ponen a prueba unidades de programa o clases de objetos individuales. Las pruebas de unidad deben enfocarse en comprobar la funcionalidad de objetos o métodos.
2. **Pruebas del componente**, donde muchas unidades individuales se integran para crear componentes compuestos. La prueba de componentes debe enfocarse en probar interfaces del componente.
3. **Pruebas del sistema**, donde algunos o todos los componentes en un sistema se integran y el sistema se prueba como un todo. Las pruebas del sistema deben enfocarse en poner a prueba las interacciones de los componentes.

Las pruebas de desarrollo son, ante todo, un proceso de prueba de defecto, en las cuales la meta consiste en descubrir bugs en el software. Por lo tanto, a menudo están entrelazadas con la depuración: el proceso de localizar problemas con el código y cambiar el programa para corregirlos.

2.2.1. Pruebas de unidad

Las pruebas de unidad son el proceso de probar componentes del programa, como métodos o clases de objetos. Las funciones o los métodos individuales son el tipo más simple de componente. Las pruebas deben llamarse para dichas rutinas con diferentes parámetros de entrada.

Cuando pone a prueba las clases de objetos, tiene que diseñar las pruebas para brindar cobertura a todas las características del objeto. Esto significa que debe:

- Probar todas las operaciones asociadas con el objeto.
- Establecer y verificar el valor de todos los atributos relacionados con el objeto.
- Poner el objeto en todos los estados posibles. Esto quiere decir que tiene que simular todos los eventos que causen un cambio de estado.

Considere, por ejemplo, el objeto de estación meteorológica del modelo analizado en la unidad anterior. Tiene un solo atributo, que es su identificador (identifier). Ésta es una constante que se establece cuando se instala la estación meteorológica. Por consiguiente, sólo se necesita una prueba que demuestre si se configuró de manera adecuada. Usted necesita definir casos de prueba para todos los métodos asociados con el objeto, como reportWeather, reportStatus, etcétera. Aunque lo ideal es poner a prueba los métodos en aislamiento, en algunos casos son precisas ciertas secuencias de prueba. Por ejemplo, para someter a prueba el método que desactiva los instrumentos de la estación meteorológica (shutdown), se necesita ejecutar el método restart (reinicio).

EstaciónMeteorológica
identificador
reportWeather ( ) reportStatus ( ) powerSave (instruments) remoteControl (commands) reconfigure (commands) restart (instruments) shutdown (instruments)

La generalización o herencia provoca que sea más complicada la prueba de las clases de objetos. Usted no debe poner únicamente a prueba una operación en la clase donde se definió, ni suponer que funcionará como se esperaba en las subclases que heredan la operación. La operación que se hereda puede hacer conjeturas sobre otras operaciones y atributos. Es posible que no sean válidas en algunas subclases que hereden la operación. Por consiguiente, tiene que poner a prueba la operación heredada en todos los contextos en que se utilice.

Siempre que sea posible, se deben automatizar las pruebas de unidad. En estas pruebas de unidad automatizadas, podría usarse un marco de automatización de pruebas (como JUnit) para escribir y correr sus pruebas de programa.

Otros ejemplos

- PHPUnit
- Prueba unitaria de Python
- Angular Unit Testing

Un conjunto automatizado de pruebas tiene tres partes:

1) Una parte de configuración, en la cual se inicializa el sistema con el caso de prueba, esto es, las entradas y salidas esperadas.
2) Una parte de llamada (call), en la cual se llama al objeto o al método que se va a probar.
3) Una parte de declaración, en la cual se compara el resultado de la llamada con el resultado esperado. Si la información se evalúa como verdadera, la prueba tuvo éxito; pero si resulta falsa, entonces fracasó.

En ocasiones, el objeto que se prueba tiene dependencias de otros objetos que tal vez no se escribieron o que, si se utilizan, frenan el proceso de pruebas. Si su objeto llama a una base de datos, por ejemplo, esto requeriría un proceso de configuración lento antes de usarse. En tales casos, usted puede decidir usar objetos mock (simulados). Éstos son objetos con la misma interfaz como los usados por objetos externos que simulan su funcionalidad.

Por ende, un objeto mock que aparenta una base de datos suele tener sólo algunos ítems de datos que se organizan en un arreglo. Por lo tanto, puede entrar rápidamente a ellos, sin las sobrecargas de llamar a una base de datos y acceder a discos. De igual modo, los objetos mock pueden usarse para simular una operación anormal o eventos extraños. Por ejemplo, si se pretende que el sistema tome acción en ciertas horas del día, su objeto mock simplemente regresará estas horas, independientemente de la hora real en el reloj.

Elección de casos de pruebas de unidad

Las pruebas son costosas y consumen tiempo, así que es importante elegir casos efectivos de pruebas de unidad. La efectividad significa, en este caso, dos cuestiones:

1. Los casos de prueba tienen que mostrar que, cuando se usan como se esperaba, el componente que se somete a prueba hace lo que se supone que debe hacer.
2. Si hay defectos en el componente, éstos deberían revelarse mediante los casos de prueba.

Aquí se discuten dos estrategias posibles que serían efectivas para ayudarle a elegir casos de prueba.

Se trata de:

1. **Prueba de partición**, donde se identifican grupos de entradas con características comunes y se procesan de la misma forma. Debe elegir las pruebas dentro de cada uno de dichos grupos.
2. **Pruebas basadas en lineamientos**, donde se usan lineamientos para elegir los casos de prueba. Dichos lineamientos reflejan la experiencia previa de los tipos de errores que suelen cometer los programadores al desarrollar componentes.

Pruebas de caja negra

Cuando se usa la especificación de un sistema para reconocer particiones de equivalencia, se llama “pruebas de caja negra”. Aquí no es necesario algún conocimiento de cómo funciona el sistema. Sin embargo, puede ser útil complementar las pruebas de caja negra con “pruebas de caja blanca”, en las cuales se busca el código del programa para encontrar otras posibles pruebas. Por ejemplo, su código puede incluir excepciones para manejar las entradas incorrectas. Este conocimiento se utiliza para identificar “particiones de excepción”: diferentes rangos donde deba aplicarse el mismo manejo de excepción.

Pruebas de caja blanca

Las pruebas de caja blanca son un método de evaluación de software que se utiliza para examinar la estructura interna, el diseño, la codificación y el funcionamiento interno del software. Los desarrolladores utilizan este método de prueba para verificar el flujo de entradas y salidas a través de la aplicación, mejorando la usabilidad y el diseño y reforzando la seguridad. El concepto se llama "caja blanca" porque es simbólicamente transparente, ya que el código es visible para el probador durante el examen. En comparación, cuando el código interno no es visible, se denomina prueba de caja negra.

Usted también puede usar lineamientos de prueba para ayudarse a elegir casos de prueba. Los lineamientos encapsulan conocimiento sobre qué tipos de casos de prueba son efectivos para la detección de errores. Por ejemplo, cuando se prueban programas con secuencias, arreglos o listas, los lineamientos que pueden ayudar a revelar defectos incluyen:

1. Probar software con secuencias que tengan sólo un valor único. Los programadores naturalmente consideran a las secuencias como compuestas por muchos valores y, en ocasiones, incrustan esta suposición en sus programas. En consecuencia, si se presenta una secuencia de un valor único, es posible que un programa no funcione de manera adecuada.
2. Usar diferentes secuencias de diversos tamaños en distintas pruebas. Esto disminuye las oportunidades de que un programa con defectos genere accidentalmente una salida correcta, debido a algunas características accidentales de la entrada.
3. Derivar pruebas de modo que se acceda a los elementos primero, medio y último de la secuencia. Este enfoque revela problemas en las fronteras de la partición.

El libro de Whittaker incluye muchos ejemplos de lineamientos que se pueden utilizar en el diseño de casos de prueba. Algunos de los lineamientos más generales que sugiere son:

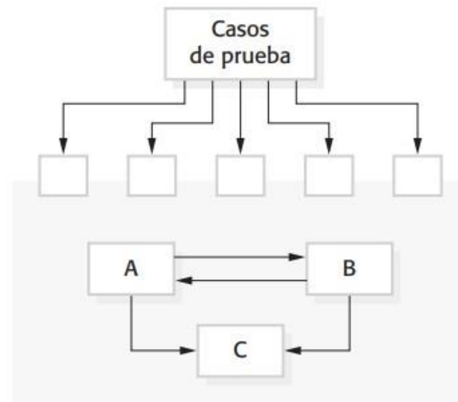
- Elegir entradas que fuercen al sistema a generar todos los mensajes de error.
- Diseñar entradas que produzcan que los buffers de entrada se desborden.
- Repetir varias veces la misma entrada o serie de entradas.
- Forzar la generación de salidas inválidas.
- Forzar resultados de cálculo demasiado largos o pequeños.

2.2.2. Pruebas de componentes

En general, los componentes de software son componentes compuestos constituidos por varios objetos en interacción. Por ejemplo, en el sistema de la estación meteorológica, el componente de reconfiguración incluye objetos que tratan con cada aspecto de la reconfiguración. El acceso a la funcionalidad de dichos objetos es a través de la interfaz de componente definida. Por consiguiente, la prueba de componentes compuestos tiene que enfocarse en mostrar que la interfaz de componente se comporta según su especificación. Usted puede suponer que dentro del componente se completaron las pruebas de unidad sobre el objeto individual.

Esta imagen ilustra la idea de la prueba de interfaz de componente. Suponga que los componentes A, B y C se integraron para crear un componente o subsistema más grande. Los casos de prueba no se aplican a los componentes individuales, sino más bien a la interfaz del componente compuesto, creado al combinar tales componentes. Los errores de interfaz

en el componente compuesto quizá no se detecten al poner a prueba los objetos individuales, porque dichos errores resultan de interacciones entre los objetos en el componente.



Existen diferentes tipos de interfaz entre componentes de programa y, en consecuencia, distintos tipos de error de interfaz que llegan a ocurrir:

1. Interfaces de parámetro: Son interfaces en que los datos, o en ocasiones referencias de función, pasan de un componente a otro. Los métodos en un objeto tienen una interfaz de parámetro.
2. Interfaces de memoria compartida: Son interfaces en que un bloque de memoria se reparte entre componentes. Los datos se colocan en la memoria de un subsistema y otros subsistemas los recuperan de ahí. Este tipo de interfaz se usa con frecuencia en sistemas embebidos, donde los sensores crean datos que se recuperan y son procesados por otros componentes del sistema.
3. Interfaces de procedimiento: Son interfaces en que un componente encapsula un conjunto de procedimientos que pueden ser llamados por otros componentes. Los objetos y otros componentes reutilizables tienen esta forma de interfaz.
4. Interfaces que pasan mensajes: Se trata de interfaces donde, al enviar un mensaje, un componente solicita un servicio de otro componente. El mensaje de retorno incluye los resultados para ejecutar el servicio. Algunos sistemas orientados a objetos tienen esta forma de interfaz, así como los sistemas cliente-servidor

Los errores de interfaz son una de las formas más comunes de falla en los sistemas complejos. Dichos errores caen en tres clases:

- Uso incorrecto de interfaz: Un componente que llama a otro componente y comete algún error en el uso de su interfaz. Este tipo de error es común

con interfaces de parámetro, donde los parámetros pueden ser del tipo equivocado, o bien, pasar en el orden o el número equivocados de parámetros.

- Mala interpretación de interfaz: Un componente que malinterpreta la especificación de la interfaz del componente llamado y hace suposiciones sobre su comportamiento. El componente llamado no se comporta como se esperaba, lo cual entonces genera un comportamiento imprevisto en el componente que llama. Por ejemplo, un método de búsqueda binaria puede llamarse con un parámetro que es un arreglo desordenado. Entonces fallaría la búsqueda.
- Errores de temporización: Ocurren en sistemas de tiempo real que usan una memoria compartida o una interfaz que pasa mensajes. El productor de datos y el consumidor de datos operan a diferentes niveles de rapidez. A menos que se tenga cuidado particular en el diseño de interfaz, el consumidor puede acceder a información obsoleta, porque el productor de la información no actualizó la información de la interfaz compartida.

2.2.3. Pruebas de interfaz

Algunos lineamientos generales para las pruebas de interfaz son:

1. Examinar el código que se va a probar y listar explícitamente cada llamado a un componente externo. Diseñe un conjunto de pruebas donde los valores de los parámetros hacia los componentes externos estén en los extremos finales de sus rangos. Dichos valores extremos tienen más probabilidad de revelar inconsistencias de interfaz.
2. Donde los punteros pasen a través de una interfaz, pruebe siempre la interfaz con parámetros de puntero nulo.
3. Donde un componente se llame a través de una interfaz de procedimiento, diseñe pruebas que deliberadamente hagan que falle el componente. Diferir las suposiciones de falla es una de las interpretaciones de especificación equivocadas más comunes.
4. Use pruebas de esfuerzo en los sistemas que pasan mensajes. Esto significa que debe diseñar pruebas que generen muchos más mensajes

de los que probablemente ocurran en la práctica. Ésta es una forma efectiva de revelar problemas de temporización.

5. Donde algunos componentes interactúen a través de memoria compartida, diseñe pruebas que varíen el orden en que se activan estos componentes. Tales pruebas pueden revelar suposiciones implícitas hechas por el programador, sobre el orden en que se producen y consumen los datos compartidos.

En ocasiones, las inspecciones y revisiones suelen ser más efectivas en costo que las pruebas para descubrir errores de interfaz. Las inspecciones pueden concentrarse en interfaces de componente e interrogantes sobre el comportamiento supuesto de la interfaz planteada durante el proceso de inspección. Un lenguaje robusto como Java permite que muchos errores de interfaz sean descubiertos por el compilador.

2.2.4. Pruebas del sistema

Las pruebas del sistema durante el desarrollo incluyen la integración de componentes para crear una versión del sistema y, luego, poner a prueba el sistema integrado. Las pruebas de sistema demuestran que los componentes son compatibles, que interactúan correctamente y que transfieren los datos correctos en el momento adecuado a través de sus interfaces.

Evidentemente, se traslapan con las pruebas de componentes, pero existen dos importantes diferencias:

1. Durante las pruebas de sistema, los componentes reutilizables desarrollados por separado y los sistemas comerciales pueden integrarse con componentes desarrollados recientemente. Entonces se prueba el sistema completo.
2. Los componentes desarrollados por diferentes miembros del equipo o de grupos pueden integrarse en esta etapa. La prueba de sistema es un proceso colectivo más que individual. En algunas compañías, las pruebas del sistema implican un equipo de prueba independiente, sin la inclusión de diseñadores ni de programadores.

Cuando se integran componentes para crear un sistema, se obtiene un comportamiento emergente. Esto significa que algunos elementos de funcionalidad del sistema sólo se hacen evidentes cuando se reúnen los componentes. Éste podría ser un comportamiento emergente planeado que debe probarse. Por ejemplo, usted puede integrar un componente de autenticación con un componente que actualice información. De esta manera, tiene una característica de sistema que restringe la información actualizada de usuarios autorizados. Sin embargo, algunas veces, el comportamiento emergente no está planeado ni se desea. Hay que desarrollar pruebas que demuestren que el sistema sólo hace lo que se supone que debe hacer.

Por lo tanto, las pruebas del sistema deben enfocarse en poner a prueba las interacciones entre los componentes y los objetos que constituyen el sistema. También se prueban componentes o sistemas reutilizables para acreditar que al integrarse nuevos componentes funcionen como se esperaba. Esta prueba de interacción debe descubrir aquellos bugs de componente que sólo se revelan cuando lo usan otros componentes en el sistema. Las pruebas de interacción también ayudan a encontrar interpretaciones erróneas, cometidas por desarrolladores de componentes, acerca de otros componentes en el sistema.

Para la mayoría de sistemas es difícil saber cuántas pruebas de sistemas son esenciales y cuándo hay que dejar de hacer pruebas. Las pruebas exhaustivas, donde se pone a prueba cada secuencia posible de ejecución del programa, son imposibles. Por lo tanto, las pruebas deben basarse en un subconjunto de probables casos de prueba. De manera ideal, para elegir este subconjunto, las compañías de software cuentan con políticas, las cuales pueden basarse en políticas de prueba generales, como una política de que todos los enunciados del programa se ejecuten al menos una vez. Como alternativa, pueden basarse en la experiencia de uso de sistema y, a la vez, enfocarse en probar las características del sistema operativo.

Por ejemplo:

1. Tienen que probarse todas las funciones del sistema que se ingresen a través de un menú.

2. Debe experimentarse la combinación de funciones (por ejemplo, formateo de texto) que se ingrese por medio del mismo menú.
3. Donde se proporcione entrada del usuario, hay que probar todas las funciones, ya sea con entrada correcta o incorrecta.

Por experiencia con los principales productos de software, como procesadores de texto u hojas de cálculo, es claro que lineamientos similares se usan por lo general durante la prueba del producto. Usualmente funcionan cuando las características del software se usan en aislamiento. Los problemas se presentan, cuando las combinaciones de características de uso menos común no se prueban en conjunto. Un ejemplo es en un procesador de texto de uso común, el uso de notas al pie de página con una plantilla en columnas múltiples causa la distribución incorrecta del texto.

2.3. Desarrollo dirigido por pruebas

El desarrollo dirigido por pruebas (TDD, por las siglas de Test-Driven Development) es un enfoque al diseño de programas donde se entrelazan el desarrollo de pruebas y el de código. En esencia, el código se desarrolla incrementalmente, junto con una prueba para ese incremento. No se avanza hacia el siguiente incremento sino hasta que el código diseñado pasa la prueba.

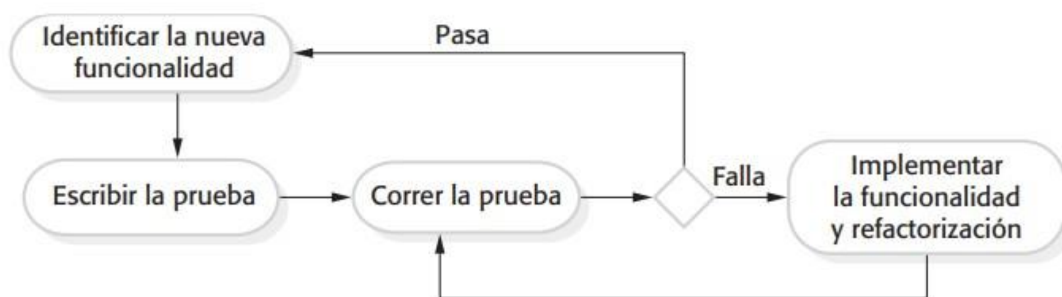
El desarrollo dirigido por pruebas se introdujo como parte de los métodos ágiles como la programación extrema. No obstante, se puede usar también en los procesos de desarrollo basados en un plan.

Los pasos en el proceso son los siguientes:

1. Se comienza por identificar el incremento de funcionalidad requerido. Éste usualmente debe ser pequeño y aplicable en pocas líneas del código.
2. Se escribe una prueba para esta funcionalidad y se implementa como una prueba automatizada. Esto significa que la prueba puede ejecutarse y reportarse, sin importar si aprueba o falla.

3. Luego se corre la prueba, junto con todas las otras pruebas que se implementaron. Inicialmente, no se aplica la funcionalidad, de modo que la nueva prueba fallará. Esto es deliberado, pues muestra que la prueba añade algo al conjunto de pruebas.
4. Luego se implementa la funcionalidad y se opera nuevamente la prueba. Esto puede incluir la refactorización del código existente, para perfeccionarlo y adicionar nuevo código a lo ya existente.
5. Una vez puestas en funcionamiento con éxito todas las pruebas, se avanza a la implementación de la siguiente funcionalidad.

Desarrollo dirigido por pruebas



Un entorno automatizado de pruebas, como el entorno JUnit que soporta pruebas del programa Java, es esencial para TDD. Conforme el código se desarrolla en incrementos muy pequeños, uno tiene la posibilidad de correr cada prueba, cada vez que se adiciona funcionalidad o se refactoriza el programa. Por consiguiente, las pruebas se incrustan en un programa independiente que corre las pruebas y apela al sistema que se prueba. Al usar este enfoque, en unos cuantos segundos se efectúan cientos de pruebas independientes.

Un argumento consistente con el desarrollo dirigido por pruebas es que ayuda a los programadores a aclarar sus ideas acerca de lo que realmente debe hacer un segmento de código. Para escribir una prueba, es preciso entender lo que se quiere, pues esta comprensión facilita la escritura del código requerido. Desde luego, si el conocimiento o la comprensión son incompletos, entonces no ayudará el desarrollo dirigido por pruebas. Por ejemplo, si su cálculo implica división, debería comprobar que no divide los números entre cero. En caso de

que olvide escribir una prueba para esto, en el programa nunca se incluirá el código a comprobar.

Además de la mejor comprensión del problema, otros beneficios del desarrollo dirigido por pruebas son:

1. Cobertura de código: En principio, cualquier segmento de código que escriba debe tener al menos una prueba asociada. Por lo tanto, puede estar seguro de que cualquier código en el sistema se ejecuta realmente. El código se prueba a medida que se escribe, de modo que los defectos se descubren con oportunidad en el proceso de desarrollo.
2. Pruebas de regresión: Un conjunto de pruebas se desarrolla incrementalmente conforme se desarrolla un programa. Siempre es posible correr pruebas de regresión para demostrar que los cambios al programa no introdujeron nuevos bugs.
3. Depuración simplificada Cuando falla una prueba, debe ser evidente dónde yace el problema. Es preciso comprobar y modificar el código recién escrito. No se requieren herramientas de depuración para localizar el problema. Los reportes del uso del desarrollo dirigido por pruebas indican que difícilmente alguna vez se necesitará usar un depurador automatizado en el desarrollo dirigido por pruebas.
4. Documentación del sistema Las pruebas en sí actúan como una forma de documentación que describen lo que debe hacer el código. Leer las pruebas suele facilitar la comprensión del código.

Uno de los beneficios más importantes del desarrollo dirigido por pruebas es que reduce los costos de las pruebas de regresión. Estas últimas implican correr los conjuntos de pruebas ejecutadas exitosamente después de realizar cambios a un sistema. La prueba de regresión verifica que dichos cambios no hayan introducido nuevos bugs en el sistema, y que el nuevo código interactúa como se esperaba con el código existente. Las pruebas de regresión son muy costosas y, por lo general, poco prácticas cuando un sistema se prueba manualmente, pues son muy elevados los costos en tiempo y esfuerzo. Ante tales situaciones, usted debe ensayar y elegir las pruebas más relevantes para volver a correrlas, y es fácil perder pruebas importantes.

Sin embargo, las pruebas automatizadas, que son fundamentales para el desarrollo de primera prueba, reducen drásticamente los costos de las pruebas de regresión. Las pruebas existentes pueden volverse a correr de manera más rápida y menos costosa. Después de realizar cambios a un sistema en el desarrollo de la primera prueba, todas las pruebas existentes deben correr con éxito antes de añadir cualquier funcionalidad accesoria. Como programador, usted podría estar seguro de que la nueva funcionalidad que agregue no causará ni revelará problemas con el código existente.

El desarrollo dirigido por pruebas se usa más en el diseño de software nuevo, donde la funcionalidad se implementa en código nuevo o usa librerías estándar perfectamente probadas. Si se reutilizan grandes componentes en código o sistemas heredados, entonces se necesita escribir pruebas para dichos sistemas como un todo.

Si se usa el desarrollo dirigido por pruebas, se necesitará de un proceso de prueba del sistema para validar el sistema; esto es, comprobar que cumple con los requerimientos de todos los participantes del sistema. Las pruebas de sistema también demuestran rendimiento, confiabilidad y evidencian que el sistema no haga aquello que no debe hacer, como producir salidas indeseadas.

El desarrollo dirigido por pruebas resulta ser un enfoque exitoso para proyectos de dimensión pequeña y mediana. Por lo general, los programadores que adoptan dicho enfoque están contentos con él y descubren que es una forma más productiva de desarrollar software. En algunos ensayos, se demostró que conduce a mejorar la calidad del código; en otros, los resultados no son concluyentes. Sin embargo, no hay evidencia de que el TDD conduzca a un código con menor calidad.

2.4. Pruebas de versión

Las pruebas de versión son el proceso de poner a prueba una versión particular de un sistema que se pretende usar fuera del equipo de desarrollo. Por lo general, la versión del sistema es para clientes y usuarios. No obstante, en un proyecto complejo, la versión podría ser para otros equipos que desarrollan

sistemas relacionados. Para productos de software, la versión sería para el gerente de producto, quien después la prepara para su venta.

Existen dos distinciones importantes entre las pruebas de versión y las pruebas del sistema durante el proceso de desarrollo:

1. Un equipo independiente que no intervino en el desarrollo del sistema debe ser el responsable de las pruebas de versión.
2. Las pruebas del sistema por parte del equipo de desarrollo deben enfocarse en el descubrimiento de bugs en el sistema (pruebas de defecto). El objetivo de las pruebas de versión es comprobar que el sistema cumpla con los requerimientos y sea suficientemente bueno para uso externo (pruebas de validación).

La principal meta del proceso de pruebas de versión es convencer al proveedor del sistema de que éste es suficientemente apto para su uso. Si es así, puede liberarse como un producto o entregarse al cliente. Por lo tanto, las pruebas de versión deben mostrar que el sistema entrega su funcionalidad, rendimiento y confiabilidad especificados, y que no falla durante el uso normal. Deben considerarse todos los requerimientos del sistema, no sólo los de los usuarios finales del sistema.

Las pruebas de versión, por lo regular, son un proceso de prueba de caja negra, donde las pruebas se derivan a partir de la especificación del sistema. El sistema se trata como una caja negra cuyo comportamiento sólo puede determinarse por el estudio de entradas y salidas relacionadas. Otro nombre para esto es “prueba funcional”, llamada así porque al examinador sólo le preocupa la funcionalidad y no la aplicación del software.

2.5. Pruebas basadas en requerimientos

Un principio general de buena práctica en la ingeniería de requerimientos es que éstos deben ser comprobables; esto es, los requerimientos tienen que escribirse de forma que pueda diseñarse una prueba para dicho requerimiento. Luego, un examinador comprueba que el requerimiento se cumpla. En consecuencia, las

pruebas basadas en requerimientos son un enfoque sistemático al diseño de casos de prueba, donde se considera cada requerimiento y se deriva un conjunto de pruebas para éste. Las pruebas basadas en requerimientos son pruebas de validación más que de defecto: se intenta demostrar que el sistema implementó adecuadamente sus requerimientos.

Por ejemplo, considere los siguientes requerimientos relacionados que se enfocan a la comprobación de alergias a medicamentos:

Si se sabe que un paciente es alérgico a algún fármaco en particular, entonces la prescripción de dicho medicamento dará como resultado un mensaje de advertencia que se emitirá al usuario del sistema. Si quien prescribe ignora una advertencia de alergia, deberá proporcionar una razón para ello.

Para comprobar si estos requerimientos se cumplen, tal vez necesite elaborar muchas pruebas relacionadas:

1. Configurar un registro de un paciente sin alergias conocidas. Prescribir medicamentos para alergias que se sabe que existen. Comprobar que el sistema no emite un mensaje de advertencia.
2. Realizar un registro de un paciente con una alergia conocida. Prescribir el medicamento al que es alérgico y comprobar que el sistema emite la advertencia.
3. Elaborar un registro de un paciente donde se reporten alergias a dos o más medicamentos. Prescribir dichos medicamentos por separado y comprobar que se emite la advertencia correcta para cada medicamento.
4. Prescribir dos medicamentos a los que sea alérgico el paciente. Comprobar que se emiten correctamente dos advertencias.
5. Prescribir un medicamento que emite una advertencia y pasar por alto dicha advertencia. Comprobar que el sistema solicita al usuario proporcionar información que explique por qué pasó por alto la advertencia.

A partir de esto se puede ver que probar un requerimiento no sólo significa escribir una prueba. Por lo general, usted deberá escribir muchas pruebas para garantizar que cubrió los requerimientos. También hay que mantener el rastreo

de los registros de sus pruebas basadas en requerimientos, que vinculan las pruebas con los requerimientos específicos que se ponen a prueba.

Pruebas de escenario

Las pruebas de escenario son un enfoque a las pruebas de versión donde se crean escenarios típicos de uso y se les utiliza en el desarrollo de casos de prueba para el sistema. Un escenario es una historia que describe una forma en que puede usarse el sistema. Los escenarios deben ser realistas, y los usuarios reales del sistema tienen que relacionarse con ellos.

Ejemplo de prueba de Escenario:

Kate es enfermera con especialidad en atención a la salud mental. Una de sus responsabilidades es visitar a domicilio a los pacientes, para comprobar la efectividad de su tratamiento y que no sufran de efectos colaterales del fármaco.

En un día de visitas domésticas, Kate ingresa al MHC-PMS y lo usa para imprimir su agenda de visitas domiciliarias para ese día, junto con información resumida sobre los pacientes por visitar. Solicita que los registros para dichos pacientes se descarguen a su laptop. Se le pide la palabra clave para cifrar los registros en la laptop.

Uno de los pacientes a quienes visita es Jim, quien es tratado con medicamentos antidepresivos. Jim siente que el medicamento le ayuda, pero considera que el efecto colateral es que se mantiene despierto durante la noche. Kate observa el registro de Jim y se le pide la palabra clave para descifrar el registro. Comprueba el medicamento prescrito y consulta sus efectos colaterales. El insomnio es un efecto colateral conocido, así que anota el problema en el registro de Jim y sugiere que visite la clínica para que cambien el medicamento. Él está de acuerdo, así que Kate ingresa un recordatorio para llamarlo en cuanto ella regrese a la clínica, para concertarle una cita con un médico. Termina la consulta y el sistema vuelve a cifrar el registro de Jim.

Más tarde, al terminar sus consultas, Kate regresa a la clínica y sube los registros de los pacientes visitados a la base de datos. El sistema genera para Kate una lista de aquellos pacientes con quienes debe comunicarse, para obtener información de seguimiento y concertar citas en la clínica.

En un breve ensayo sobre las pruebas de escenario, (Kaner) sugiere que una prueba de escenario debe ser una historia narrativa que sea creíble y bastante compleja. Tiene que motivar a los participantes; esto es, deben relacionarse con el escenario y creer que es importante que el sistema pase la prueba. También sugiere que debe ser fácil de evaluar. Si hay problemas con el sistema, entonces el equipo de pruebas de versión tiene que reconocerlos. Como ejemplo de un posible escenario para el MHC-PMS, la imagen anterior describe una forma de utilizar el sistema en una visita domiciliaria, que pone a prueba algunas características del MHC-PMS:

1. Autenticación al ingresar al sistema.
2. Descarga y carga registros de paciente específicos desde una laptop.
3. Agenda de visitas a domicilio

4. Cifrado y descifrado de registros de pacientes en un dispositivo móvil.
5. Recuperación y modificación de registros.
6. Vinculación con la base de datos de medicamentos que mantenga información acerca de efectos colaterales.
7. Sistema para recordatorio de llamadas.

Si usted es examinador de versión, opere a través de este escenario, interprete el papel de Kate y observe cómo se comporta el sistema en respuesta a las diferentes entradas. Como "Kate", usted puede cometer errores deliberados, como ingresar la palabra clave equivocada para decodificar registros. Esto comprueba la respuesta del sistema ante los errores. Tiene que anotar cuidadosamente cualquier problema que surja, incluidos problemas de rendimiento.

Si un sistema es muy lento, esto cambiará la forma en que se usa. Por ejemplo, si se tarda mucho al cifrar un registro, entonces los usuarios que tengan poco tiempo pueden saltar esta etapa. Si pierden su laptop, una persona no autorizada podría ver entonces los registros de los pacientes.

Cuando se usa un enfoque basado en escenarios, se ponen a prueba por lo general varios requerimientos dentro del mismo escenario. Por lo tanto, además de comprobar requerimientos individuales, también demuestra que las combinaciones de requerimientos no causan problemas.

2.6. Pruebas de rendimiento

Una vez integrado completamente el sistema, es posible probar propiedades emergentes, como el rendimiento y la confiabilidad. Las pruebas de rendimiento deben diseñarse para garantizar que el sistema procese su carga pretendida. Generalmente, esto implica efectuar una serie de pruebas donde se aumenta la carga, hasta que el rendimiento del sistema se vuelve inaceptable.

Como con otros tipos de pruebas, las pruebas de rendimiento se preocupan tanto por demostrar que el sistema cumple con sus requerimientos, como por descubrir

problemas y defectos en el sistema. Para probar si los requerimientos de rendimiento se logran, quizá se deba construir un perfil operativo.

Un perfil operativo es un conjunto de pruebas que reflejan la mezcla real de trabajo que manejará el sistema. Por consiguiente, si el 90% de las transacciones en un sistema son del tipo A, el 5% del tipo B, y el resto de los tipos C, D y E, entonces habrá que diseñar el perfil operativo de modo que la gran mayoría de pruebas sean del tipo A. De otra manera, no se obtendrá una prueba precisa del rendimiento operativo del sistema.

Desde luego, este enfoque no necesariamente es el mejor para pruebas de defecto. La experiencia demuestra que una forma efectiva de descubrir defectos es diseñar pruebas sobre los límites del sistema. En las pruebas de rendimiento, significa estresar el sistema al hacer demandas que estén fuera de los límites de diseño del software. Esto se conoce como “prueba de esfuerzo”. Por ejemplo, digamos que usted prueba un sistema de procesamiento de transacciones que se diseña para procesar hasta 300 transacciones por segundo.

Comienza por probar el sistema con menos de 300 transacciones por segundo. Luego aumenta gradualmente la carga del sistema más allá de 300 transacciones por segundo, hasta que está muy por arriba de la carga máxima de diseño del sistema y el sistema falla.

Este tipo de pruebas tiene dos funciones:

1. Prueba el comportamiento de falla del sistema. Pueden surgir circunstancias a través de una combinación inesperada de eventos donde la carga colocada en el sistema supere la carga máxima anticipada. Ante tales circunstancias, es importante que la falla del sistema no cause corrupción de datos o pérdida inesperada de servicios al usuario. Las pruebas de esfuerzo demuestran que la sobrecarga del sistema hace que “falle poco” en vez de colapsar bajo su carga
2. Fuerza al sistema y puede hacer que salgan a la luz defectos que no se descubrirían normalmente.

Aunque se puede argumentar que esos defectos probablemente no causen fallas en el sistema en uso normal, pudiera haber una serie de combinaciones inusuales de circunstancias normales que requieren pruebas de esfuerzo.

Las pruebas de esfuerzo son particularmente relevantes para los sistemas distribuidos basados en redes de procesadores. Dichos sistemas muestran con frecuencia degradación severa cuando se cargan en exceso. La red se empantana con la coordinación de datos que deben intercambiar los diferentes procesos.

Éstos se vuelven cada vez más lentos conforme esperan los datos requeridos de otros procesos. Las pruebas de esfuerzo ayudan a descubrir cuándo comienza la degradación, de manera que se puedan adicionar comprobaciones al sistema para rechazar transacciones más allá de este punto.

2.7. Pruebas de usuario

Las pruebas de usuario o del cliente son una etapa en el proceso de pruebas donde los usuarios o clientes proporcionan entrada y asesoría sobre las pruebas del sistema. Esto puede implicar probar de manera formal un sistema que se comisionó a un proveedor externo, o podría ser un proceso informal donde los usuarios experimentan con un nuevo producto de software, para ver si les gusta y si hace lo que necesitan.

Las pruebas de usuario son esenciales, aun cuando se hayan realizado pruebas abarcadoras de sistema y de versión. La razón de esto es que la influencia del entorno de trabajo del usuario tiene un gran efecto sobre la fiabilidad, el rendimiento, el uso y la robustez de un sistema.

Es casi imposible que un desarrollador de sistema replique el entorno de trabajo del sistema, pues las pruebas en el entorno del desarrollador forzosamente son artificiales. Por ejemplo, un sistema que se pretenda usar en un hospital se usa en un entorno clínico donde suceden otros hechos, como emergencias de pacientes, conversaciones con familiares del paciente, etcétera. Todo ello afecta

el uso de un sistema, pero los desarrolladores no pueden incluirlos en su entorno de pruebas.

En la práctica, hay tres diferentes tipos de pruebas de usuario:

1. Pruebas alfa, donde los usuarios del software trabajan con el equipo de diseño para probar el software en el sitio del desarrollador.
2. Pruebas beta, donde una versión del software se pone a disposición de los usuarios, para permitirles experimentar y descubrir problemas que encuentran con los desarrolladores del sistema.
3. Pruebas de aceptación, donde los clientes prueban un sistema para decidir si está o no listo para ser aceptado por los desarrolladores del sistema y desplegado en el entorno del cliente.

En las pruebas alfa, los usuarios y desarrolladores trabajan en conjunto para probar un sistema a medida que se desarrolla. Esto significa que los usuarios pueden identificar problemas y conflictos que no son fácilmente aparentes para el equipo de prueba de desarrollo. Los desarrolladores en realidad sólo pueden trabajar a partir de los requerimientos, pero con frecuencia esto no refleja otros factores que afectan el uso práctico del software. Por lo tanto, los usuarios brindan información sobre la práctica que ayuda con el diseño de pruebas más realistas.

Las pruebas alfa se usan a menudo cuando se desarrollan productos de software que se venden como sistemas empaquetados. Los usuarios de dichos productos quizás estén satisfechos de intervenir en el proceso de pruebas alfa porque esto les da información oportuna acerca de las características del nuevo sistema que pueden explotar. También reduce el riesgo de que cambios no anticipados al software tengan efectos perturbadores para su negocio. Sin embargo, las pruebas alfa también se utilizan cuando se desarrolla software personalizado. Los métodos ágiles, como XP, abogan por la inclusión del usuario en el proceso de desarrollo y que los usuarios tengan un papel activo en el diseño de pruebas para el sistema.

Las pruebas beta tienen lugar cuando una versión temprana de un sistema de software, en ocasiones sin terminar, se pone a disposición de clientes y usuarios para evaluación.

Los examinadores beta pueden ser un grupo selecto de clientes que sean adoptadores tempranos del sistema. De manera alternativa, el software se pone a disposición pública para uso de quienquiera que esté interesado en él. Las pruebas beta se usan sobre todo para productos de software que se emplean en entornos múltiples y diferentes (en oposición a los sistemas personalizados, que se utilizan por lo general en un entorno definido). Es imposible que los desarrolladores de producto conozcan y repliquen todos los entornos donde se usará el software.

En consecuencia, las pruebas beta son esenciales para descubrir problemas de interacción entre el software y las características del entorno donde se emplea. Las pruebas beta tambiénson una forma de comercialización: los clientes aprenden sobre su sistema y lo que puede hacer por ellos.

Las pruebas de aceptación son una parte inherente del desarrollo de sistemas personalizados. Tienen lugar después de las pruebas de versión. Implican a un cliente que prueba de manera formal un sistema, para decidir si debe o no aceptarlo del desarrollador del sistema. La aceptación implica que debe realizarse el pago por el sistema.

Existen seis etapas en el proceso de pruebas de aceptación

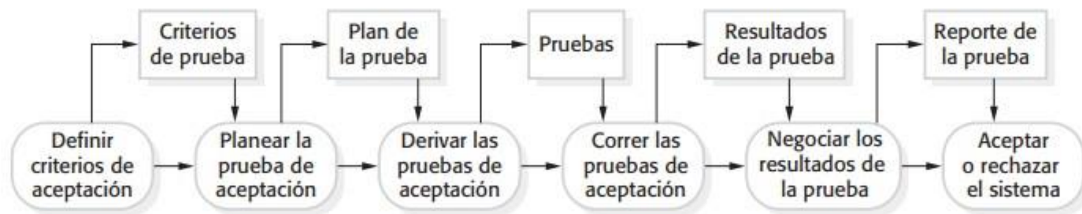
1. Definir los criterios de aceptación: Esta etapa debe, de manera ideal, anticiparse en el proceso, antes de firmar el contrato por el sistema. Los criterios de aceptación forman parte del contrato del sistema y tienen que convenirse entre el cliente y el desarrollador. Sin embargo, en la práctica suele ser difícil definir los criterios de manera tan anticipada en el proceso. Es posible que no estén disponibles requerimientos detallados y que haya cambios significativos en los requerimientos durante el proceso de desarrollo.
2. Plan de pruebas de aceptación: Esto incluye decidir sobre los recursos, el tiempo y el presupuesto para las pruebas de aceptación, así como

establecer un calendario de pruebas. El plan de pruebas de aceptación debe incluir también la cobertura requerida de los requerimientos y el orden en que se prueban las características del sistema. Tiene que definir riesgos al proceso de prueba, como caídas del sistema y rendimiento inadecuado, y resolver cómo mitigar dichos riesgos.

3. Derivar pruebas de aceptación: Una vez establecidos los criterios de aceptación, tienen que diseñarse pruebas para comprobar si un sistema es aceptable o no. Las pruebas de aceptación deben dirigirse a probar tanto las características funcionales como las no funcionales del sistema (por ejemplo, el rendimiento). Lo ideal sería que dieran cobertura completa a los requerimientos del sistema. En la práctica, es difícil establecer criterios de aceptación completamente objetivos. Con frecuencia hay espacio para argumentar sobre si las pruebas deben mostrar o no que un criterio se cubre de manera definitiva.
4. Correr pruebas de aceptación: Las pruebas de aceptación acordadas se ejecutan sobre el sistema. De manera ideal, esto debería ocurrir en el entorno real donde se usará el sistema, pero esto podría ser perturbador y poco práctico. En consecuencia, quizá deba establecerse un entorno de pruebas de usuario para efectuar dichas pruebas. Es difícil automatizar este proceso, ya que parte de las pruebas de aceptación podría necesitar poner a prueba las interacciones entre usuarios finales y el sistema. Es posible que se requiera cierta capacitación de los usuarios finales.
5. Negociar los resultados de las pruebas: Es poco probable que se pasen todas las pruebas de aceptación definidas y que no haya problemas con el sistema. Si éste es el caso, entonces las pruebas de aceptación están completas y el sistema está listo para entregarse. Con mayor regularidad se descubrirán algunos problemas. En tales casos, el desarrollador y el cliente tienen que negociar para decidir si el sistema es suficientemente adecuado para ponerse en uso. También deben acordar sobre la respuesta del desarrollador para identificar problemas.
6. Rechazo/aceptación del sistema Esta etapa incluye una reunión entre los desarrolladores y el cliente para decidir si el sistema debe aceptarse o no. Si el sistema no es suficientemente bueno para usarse, entonces se

requiere mayor desarrollo para corregir los problemas identificados. Una vez completo, se repite la fase de pruebas de aceptación.

Proceso de prueba de aceptación



Ejemplos de pruebas de usuario

Tabla 33
Plantilla Historia de usuario- registro de calificaciones
Historia de Usuario – Registro de calificaciones

Historia de usuario	
Número: 15	**Nombre:** Registro de calificaciones
Usuario: Docente	
Prioridad en negocio: Media	**Iteración asignada:** 3
Riesgo en desarrollo: Alta	**Puntos estimados:** 3
Descripción: COMO docente QUIERO poder ingresar las calificaciones de los estudiantes PARA tener un control académico de cada una de las asignaturas que imparte un profesor.	
Escenarios de prueba: Dado el ingreso de calificaciones de un estudiante CUANDO se pulse el botón de guardar, ENTONCES se presenta un mensaje indicando que la operación se realizó con éxito. Dado el ingreso de una calificación superior a 20 o inferior a 0 CUANDO se pulse el botón de guardar, ENTONCES se presenta un mensaje de alerta.	

Nota: Fuente: Freire, C. & Eras, S. (2017). PUCE SD.

2.8. Pruebas de seguridad

La valoración de la seguridad del sistema es cada vez más importante, pues más y más sistemas críticos se habilitan en Internet, por lo que cualquier persona con una conexión de red podría ingresar a ellos. Todos los días se presentan

historias de ataques en sistemas basados en Web, y virus y gusanos se distribuyen regularmente mediante protocolos de Internet.

Todo esto significa que los procesos de verificación y validación para sistemas basados en Web deben enfocarse en la evaluación de la seguridad, en la que se pone a prueba la habilidad del sistema para resistir diferentes tipos de ataques. Sin embargo, este tipo de

valoración de la seguridad es muy difícil de realizar. Por ende, los sistemas se implementan a menudo con vacíos en la seguridad. Los atacantes aprovechan esos vacíos para lograr el acceso al sistema, o causar daño al sistema o a sus datos.

En esencia, existen dos razones que hacen muy difíciles las pruebas de seguridad:

1. Los requerimientos de seguridad, como algunos requerimientos de protección, se enuncian como “no debe”. Esto es, especifican lo que no debe ocurrir, ya que es algo que se opone a la funcionalidad o al comportamiento requerido del sistema. Por regla general, no es posible definir este comportamiento no deseado como simples restricciones a comprobar por el sistema.

   Si hay recursos disponibles, es posible demostrar, al menos en principio, que un sistema cumple con sus requerimientos funcionales. Sin embargo, es imposible probar que un sistema no hace algo. Sin importar la cantidad de pruebas, las vulnerabilidades en la seguridad pueden permanecer en un sistema después de que se implementa. Desde luego, es posible generar requerimientos funcionales diseñados para proteger al sistema contra algunos tipos conocidos de ataques. No obstante, es imposible derivar requerimientos para tipos de ataques desconocidos o no anticipados. Incluso en sistemas que se hayan usado durante años, un atacante ingenioso es capaz de descubrir nuevas formas de ataque y penetrar a lo que se consideraba un sistema seguro.
2. Las personas que atacan un sistema son inteligentes y buscan activamente vulnerabilidades que puedan aprovechar. Quieren experimentar con el sistema y utilizan artilugios que se alejan mucho de

la actividad y el uso normales del sistema. Por ejemplo, en un campo de apellido podrían ingresar 1,000 caracteres con una mezcla de letras, signos de puntuación y números. Más aún, una vez que encuentran una vulnerabilidad, podrían intercambiar información sobre ésta y aumentar el número de atacantes potenciales.

Los atacantes a menudo intentan descubrir las suposiciones que hacen los desarrolladores del sistema, para entonces contradecir dichas suposiciones y observar lo que sucede. Están en una posición para usar y explorar un sistema durante cierto periodo y analizarlo mediante herramientas de software para descubrir vulnerabilidades que puedan aprovechar. De hecho, es muy probable que tengan más tiempo para buscar vulnerabilidades que los ingenieros de pruebas del sistema, pues estos últimos también deben enfocarse en realizar las pruebas del sistema.

Por esta razón, el análisis estático es particularmente útil como herramienta de prueba de seguridad. Un análisis estático de un programa puede guiar rápidamente al equipo de prueba hacia áreas de un programa que incluyen errores y vulnerabilidades. Las anomalías reveladas en el análisis estático pueden corregirse directamente, o bien, ayudan a identificar pruebas necesarias para revelar si dichas anomalías representan en realidad un riesgo para el sistema.

Para comprobar la seguridad de un sistema, puede usarse una combinación de pruebas, basado en herramientas y verificación formal:

1. Pruebas basadas en la experiencia: En este caso, el sistema se analiza contra tipos de ataque que conoce el equipo de validación. Esto implica el desarrollo de casos de prueba o el examen del código fuente de un sistema. Por ejemplo, para comprobar que el sistema no es susceptible al bien conocido ataque de envenenamiento SQL, se prueba el sistema usando
2. entradas que incluyan comandos SQL. Para comprobar que no ocurrirán errores de desbordamiento de buffer, se examinan todos los buffers de entrada para ver si el programa comprueba que las asignaciones a los elementos del buffer están dentro de los límites.

3. Este tipo de validación se realiza habitualmente en conjunto con la validación basada en herramientas, donde estas últimas brindan información que ayuda a enfocar las pruebas del sistema. Pueden crearse listas de verificación de conocidos problemas de seguridad para auxiliar en el proceso. La siguiente imagen brinda algunos ejemplos de preguntas que ayudan a impulsar las pruebas basadas en la experiencia. En una lista de verificación de problemas de seguridad también deberían incluirse las comprobaciones acerca de si se siguieron los lineamientos de diseño y programación para seguridad.
4. Equipos tigre: Ésta es una forma de pruebas basadas en la experiencia en las que es posible apoyarse en experiencia externa al equipo de desarrollo para probar un sistema de aplicación. Se establece un “equipo tigre”, al que se le impone el objetivo de violar la seguridad del sistema. Ellos simulan ataques al sistema y usan su ingenio para descubrir nuevas formas de comprometer la seguridad del sistema. Los miembros del equipo tigre deben tener experiencia previa con pruebas de seguridad y descubrir debilidades de seguridad en los sistemas.
5. Pruebas basadas en herramientas Para este método se usan varias herramientas de seguridad, tales como verificadores de contraseña que permiten analizar el sistema. Los verificadores de contraseñas detectan contraseñas inseguras, por ejemplo, los nombres comunes o las cadenas de letras consecutivas. Este enfoque en realidad es una extensión de la validación basada en la experiencia, donde la experiencia con las fallas de seguridad se concentra en las herramientas utilizadas. Desde luego, el análisis estático es otro tipo de prueba basada en herramientas.
6. Verificación formal Un sistema puede verificarse contra una especificación de seguridad formal. Sin embargo, como en otras áreas, la verificación formal de la seguridad no se usa de manera amplia.

Inevitablemente, las pruebas de la seguridad están limitadas por el tiempo y los recursos disponibles del equipo de pruebas. Esto significa que, por lo regular, es conveniente adoptar

un enfoque basado en el riesgo para las pruebas de seguridad y enfocarse en lo que se consideran los riesgos más significativos que enfrenta el sistema. Si se

dispone de un análisis de los riesgos de seguridad para el sistema, puede usarse para impulsar el proceso de pruebas. Así como las pruebas del sistema contra los requerimientos de seguridad se derivan de dichos riesgos, el equipo de pruebas también debería tratar de romper el sistema al adoptar enfoques alternativos que amenacen los activos del sistema.

2.9. Pruebas de despliegue

En muchos casos, el software debe ejecutarse en varias plataformas y bajo más de un entorno de sistema operativo. La prueba de despliegue, en ocasiones llamada prueba de configuración, ejercita el software en cada entorno en el que debe operar. Además, examina todos los procedimientos de instalación y el software de instalación especializado (por ejemplo, “instaladores”) que usarán los clientes, así como toda la documentación que se usará para introducir el software a los usuarios finales.

Para asegurar el correcto funcionamiento del sistema, resulta esencial que tengamos en cuenta las dependencias que pueden existir entre los distintos componentes del sistema y sus versiones. Una aplicación puede que sólo funcione con una versión concreta de una biblioteca auxiliar. Un disco duro puede que sólo rinda al nivel deseado si instalamos un controlador concreto. Componentes que por separado funcionarían correctamente, combinados causan problemas, por lo que deberemos utilizar sólo combinaciones conocidas que no presenten problemas de compatibilidad.

Si nuestro sistema reemplaza a un sistema anterior o se despliega paulatinamente en distintas fases, también hemos de planificar cuidadosamente la transición del sistema antiguo al nuevo de forma que sus usuarios no sufran una disrupción en el funcionamiento del sistema. En ocasiones, el sistema se instala físicamente en un entorno duplicado y la transición se hace de forma instantánea una vez que la nueva configuración funciona correctamente. Cuando el presupuesto no da para tanto, tal vez haya que buscar un momento de baja utilización del sistema para realizar la actualización (por las noches o en fin de semana, por ejemplo).

Como ejemplo de una prueba de despliegue más profunda puede abarcar combinaciones de navegadores web con varios sistemas operativos (por ejemplo, Linux, Mac OS, Windows).

El despliegue tiene por objetivo asegurar que una aplicación se podrá implantar en un entorno que cuente con la infraestructura y aplicaciones base requeridas por ésta, atendiendo estrictamente a las instrucciones descritas en la documentación de soporte a la instalación. En general, los despliegues se realizarán en el entorno de pruebas, sin embargo, teniendo en

cuenta el objetivo del servicio, este factor no es condicionante y se pueden llegar a realizar los despliegues sobre otros entornos.

2.10. Pruebas de usabilidad

Las pruebas de usabilidad son procedimientos con los que se puede probar la usabilidad de una web. Utilizando métodos empíricos, una prueba de usabilidad ofrece posibilidades para optimizar la experiencia del usuario. El resultado de tal prueba puede llevar a cambios en el diseño o en la redacción de la web, lo que debería llevar a un aumento de las conversiones. Este es el caso, en particular, del comercio electrónico. Las pruebas de usabilidad también pueden ser un aspecto del desarrollo ágil de un software.

El éxito final de una aplicación radica en que funcione impecablemente y su diseño sea amigable. Esto hace que, junto a las pruebas funcionales, las pruebas de usabilidad sean una herramienta fundamental para corregir y mejorar las aplicaciones, entender cómo interactúan los usuarios con ellas y qué tan fácil les resulta usarlas.

La característica de usabilidad tiene una serie de sub características tales como:

- Capacidad para reconocer su adecuación: Capacidad del producto que permite al usuario entender si el software es adecuado para sus necesidades.
- Capacidad de aprendizaje: Capacidad del producto que permite al usuario aprender su aplicación.

- Capacidad para ser usado: Capacidad del producto que permite al usuario operarlo y controlarlo con facilidad.
- Protección contra errores de usuario: Capacidad del sistema para proteger a los usuarios de cometer errores.
- Estética de la interfaz de usuario: Capacidad de la interfaz de usuario de ser agradable visualmente, y satisfacer la interacción con el usuario.
- Accesibilidad: Capacidad del producto que permite que sea utilizado por usuarios con determinadas características y discapacidades.
- De esta manera, las aplicaciones fáciles de usar proporcionan múltiples beneficios, tales como:
- Usuarios más satisfechos: La satisfacción de los usuarios es un resultado directo de las posibilidades que tengan estos de conseguir sus objetivos, con el mínimo esfuerzo posible.
- Usuarios más fieles: La facilidad de uso produce una utilización mayor de funcionalidades tanto en frecuencia como en amplitud. Provoca en los usuarios el deseo de volver a utilizar la aplicación y de seguir indagando en sus funcionalidades.
- Menor costo de soporte: Una aplicación más fácil de usar genera menos problemas a los usuarios y por tanto se reducen las necesidades de soporte y ayuda.
- Menor costo de mantenimiento: Los problemas de usabilidad se reflejan por las quejas de los usuarios a través de las llamadas a soporte, comentarios negativos tanto en redes sociales como en tiendas de aplicaciones, lo que genera un ciclo permanente de modificaciones. Por esta razón, es mejor hacer las aplicaciones más usables al momento de construirlas.

Las pruebas de usabilidad evalúan el grado en que el sistema puede ser utilizado por usuarios específicos con efectividad, eficiencia y satisfacción en un contexto de uso específico.

Si bien existen varias técnicas para analizar la usabilidad de una web, donde nos enfocaremos será en la evaluación heurística y en las pruebas con usuarios.

Las pruebas de usabilidad también conocido como User Experience Test (UX), es un método de prueba para medir qué tan fácil y fácil es una aplicación de software. Un pequeño grupo de usuarios finales de destino usa una aplicación de software para exponer fallas de usabilidad. Las pruebas de usabilidad se enfocan principalmente en la facilidad de uso de un usuario de la aplicación, la flexibilidad de la aplicación para manejar los controles y la capacidad de la aplicación para lograr sus objetivos.

Por qué las pruebas de usabilidad

La estética y el diseño son importantes. El aspecto de un producto suele determinar qué tan bien funciona.

Hay muchas aplicaciones de software / sitios web que fallan gravemente cuando se inician, por las siguientes razones:

- ¿Dónde hago clic en Siguiente?
- ¿Qué página necesita ser navegada?
- ¿Qué icono o jerga representa qué?
- Los mensajes de error no son consistentes o no se muestran de manera efectiva
- El tiempo de la sesión no es suficiente.

En ingeniería de software, las pruebas de usabilidad identifican errores de usabilidad en el sistema al principio del ciclo de desarrollo y pueden salvar un producto de fallas.

El objetivo de esta prueba es satisfacer a los usuarios y se centra principalmente en los siguientes parámetros de un sistema:

Eficiencia del sistema

- ¿Es el sistema fácil de aprender?
- ¿Es útil el sistema y agrega valor a la audiencia objetivo?
- ¿Son el material, el color, los iconos, las imágenes estéticamente agradables?

Eficiencia

- Lograr la pantalla o página web deseada no debería requerir mucha navegación, y la barra de desplazamiento debería usarse con frecuencia.
- Uniformidad en el formato de pantalla / páginas en su aplicación / sitio web.
- Opción de buscar dentro de una aplicación de software o sitio web.

Precisión

- No deben estar presentes detalles desactualizados o incorrectos, como información de contacto / dirección.
- No debe haber enlaces rotos.

Amistad del usuario

- Los controles utilizados deben ser auto explicativos y no requieren capacitación para operar.
- Se debe brindar asistencia a los usuarios para comprender la aplicación / sitio web
- La alineación con los objetivos anteriores ayuda a realizar pruebas de usabilidad efectivas

Ventajas de las pruebas de usabilidad

- Ayuda a descubrir problemas de usabilidad antes de que se comercialice el producto.
- Ayuda a mejorar la satisfacción del usuario final.
- Hace que su sistema sea muy eficiente y efectivo
- Ayuda a recopilar comentarios reales de su público objetivo que realmente usa su sistema durante las pruebas de usabilidad. No tiene que depender de «opiniones» de personas al azar.

Pruebas de usabilidad en desventaja

El costo es una consideración importante en las pruebas de usabilidad. Se necesitan muchos recursos para configurar un laboratorio de pruebas de usabilidad. Reclutar y administrar probadores de usabilidad también puede ser costoso

En informática la heurística consiste en encontrar o construir algoritmos con buena velocidad para ser ejecutados como los juegos informáticos o los programas que detectan si un correo electrónico es un spam o no.

2.11. Pruebas de compatibilidad

La compatibilidad es la capacidad de vivir y trabajar juntos sin ninguna discrepancia. Las aplicaciones de software compatibles también funcionan en la misma configuración. Por ejemplo, si el sitio Google.com es compatible, debería abrirse en todos los navegadores y sistemas operativos.

La compatibilidad es una prueba no funcional para garantizar la satisfacción del cliente. Sirve para determinar si su aplicación de software o producto es lo suficientemente competente para ejecutarse en diferentes navegadores, bases de datos, hardware, sistema operativo, dispositivos móviles y redes.

La aplicación también podría afectar debido a las diferentes versiones, resolución, velocidad de Internet y configuración, etc. Por lo tanto, es importante probar la aplicación de todas las formas posibles para reducir las fallas y superar las vergüenzas de la fuga de errores. Como pruebas no funcionales, las pruebas de compatibilidad son para respaldar que la aplicación se ejecuta correctamente en diferentes navegadores, versiones, sistemas operativos y redes con éxito.

Tipos de pruebas de compatibilidad de software

- Prueba de compatibilidad del navegador
- Hardware
- Redes
- Dispositivos móviles
- Sistema operativo
- Versiones

Hay dos tipos de comprobaciones de versión en la prueba de compatibilidad: Prueba de compatibilidad con versiones anteriores

Es una técnica para verificar el comportamiento y la compatibilidad del hardware o software desarrollado con sus versiones anteriores del hardware o software. Las pruebas de compatibilidad con versiones anteriores son mucho más predecibles, ya que se conocen todos los cambios de las versiones anteriores.

Prueba de compatibilidad hacia adelante

Es un proceso para verificar el comportamiento y la compatibilidad del hardware o software desarrollado con las versiones más recientes del hardware o software. Las pruebas de compatibilidad con versiones posteriores son difíciles de predecir porque se desconocen los cambios en las versiones más recientes.

Herramientas para pruebas de compatibilidad

BrowserStack: Esta herramienta ayuda al ingeniero de software a verificar la aplicación en diferentes navegadores.

Escritorio virtual: Se utiliza para ejecutar las aplicaciones en varios sistemas operativos, como máquinas virtuales. Se puede vincular el número de sistemas y comparar los resultados.

¿Por qué realizamos pruebas de compatibilidad?

La prueba de compatibilidad consiste en verificar que la aplicación funcione de la misma manera para todas las plataformas.

Por lo general, el equipo de desarrollo y el equipo de pruebas prueban la aplicación en una única plataforma. Pero una vez que la aplicación se lanza en producción, el cliente puede probar nuestro producto en una plataforma diferente y puede encontrar errores en la aplicación que no son dignos en términos de calidad.

Para reducir estos problemas y no molestar a sus clientes, es importante probar la aplicación en todas las plataformas.

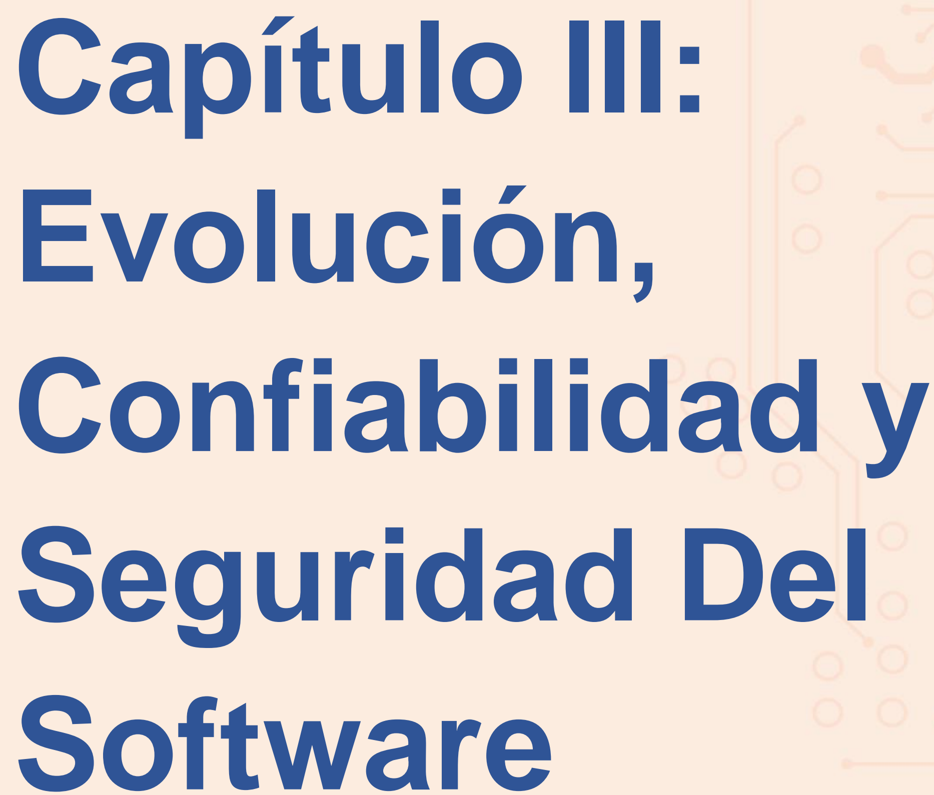
Defectos comunes de las pruebas de compatibilidad

- Cambios en la interfaz de usuario (apariencia)

- Cambio de tamaño de fuente
- Problemas relacionados con la alineación
- Cambio de estilo y color CSS
- Problemas relacionados con la barra de desplazamiento
- Superposición de contenido o etiqueta
- Mesas o marcos rotos

Cómo hacer una prueba de compatibilidad

1. El primer paso en las pruebas de compatibilidad es definir el conjunto de entornos o plataformas en las que se espera que funcione la aplicación.
2. El evaluador debe tener suficiente conocimiento de las plataformas / software / hardware para comprender el comportamiento esperado de la aplicación en varias configuraciones.
3. El entorno debe configurarse para realizar pruebas con diferentes plataformas, dispositivos y redes para verificar si su aplicación se ejecuta bien en diferentes configuraciones.
4. Informe los errores. Arregle las fallas.



Capítulo III: Evolución, Confiabilidad y Seguridad Del Software

Evolución, Confiabilidad y Seguridad Del Software

El desarrollo del software no se detiene cuando un sistema se entrega, sino que continúa a lo largo de la vida de éste. Después de distribuir un sistema, inevitablemente debe modificarse, con la finalidad de mantenerlo útil.

Tanto los cambios empresariales como los de las expectativas del usuario generan nuevos requerimientos para el software existente.

Es posible que tengan que modificarse partes del software para corregir errores encontrados durante su operación, para adaptarlo a los cambios en su plataforma de software y hardware, y para mejorar su rendimiento u otras características no funcionales.

La evolución del software es importante porque las organizaciones invierten grandes cantidades de dinero en él y en la actualidad son completamente dependientes de dichos sistemas.

Sus sistemas se consideran activos empresariales críticos, por lo que tienen que invertir en el cambio del sistema para mantener el valor de estos activos. En consecuencia, las compañías más grandes gastan más en conservar los sistemas existentes que en el desarrollo de sistemas nuevos.

La evolución del software puede potenciarse al cambiar los requerimientos empresariales, con reportes de defectos del software o por cambios a otros sistemas en un entorno del sistema de software. Hopkins y Jenkins (2008) acuñaron el término “desarrollo de software abandonado” (subutilizado) para describir situaciones en que los sistemas de software tienen que desarrollarse y gestionarse en un ambiente donde dependen de muchos otros sistemas de software.

Por consiguiente, la evolución de un sistema rara vez puede considerarse en aislamiento. Los cambios al entorno conducen a cambios en el sistema que, a la vez, pueden generar más cambios en el entorno.

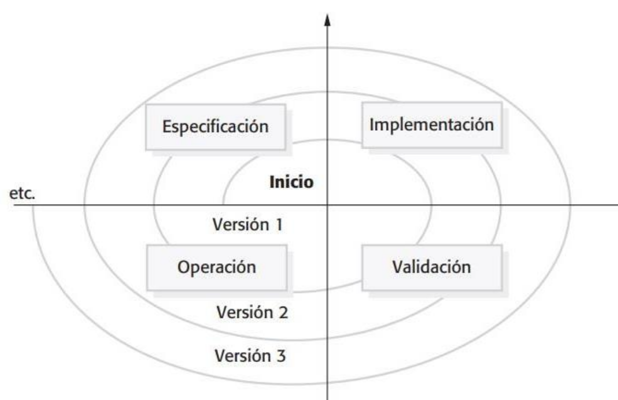
Desde luego, el hecho de que los sistemas tengan que evolucionar en un ambiente "rico en sistemas" con frecuencia aumenta las dificultades y los costos de la evolución.

Además de comprender y analizar el impacto de un cambio propuesto sobre el sistema en sí, también es probable que se deba valorar cómo esto afectaría a otros sistemas en el entorno operacional.

Por lo general, los sistemas de software útiles tienen una vida muy larga. Por ejemplo, los grandes sistemas militares o de infraestructura, como los de control de tráfico aéreo, llegan a durar 30 años o más; en tanto que los sistemas empresariales con frecuencia superan los 10 años. Puesto que el costo del software es elevado, una compañía debe usar un sistema de software durante muchos años para recuperar su inversión.

Evidentemente, los requerimientos de los sistemas instalados cambian conforme lo hacen el negocio y su entorno. Por consiguiente, se crean a intervalos regulares nuevas versiones de los sistemas, las cuales incorporan cambios y actualizaciones.

Por ende, la ingeniería de software se debe considerar como un proceso en espiral, con requerimientos, diseño, implementación y pruebas continuas, a lo largo de la vida del sistema



Esto comienza por crear la versión 1 del sistema. Una vez entregada, se proponen cambios y casi de inmediato comienza el desarrollo de la versión 2. De hecho, la necesidad de evolución puede volverse evidente incluso antes de que el sistema se distribuya, de manera que las futuras versiones del software estarían en desarrollo antes de que se libere la versión actual.

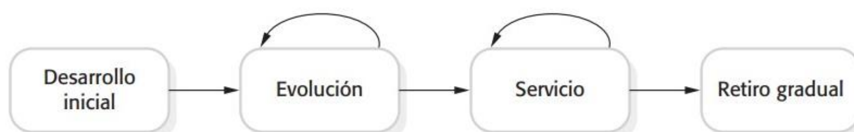
Este modelo de evolución de software implica que una sola organización es responsable tanto del desarrollo del software inicial como de la evolución del software. La mayoría de los productos de software empacados se desarrollan siguiendo este enfoque. Para el software personalizado, por lo general se utiliza un enfoque diferente.

Una compañía de software lo desarrolla para un cliente y, luego, el personal de desarrollo del propio cliente se hace cargo del sistema. Ellos son los responsables de la evolución del software. De forma alternativa, el cliente del software puede otorgar por separado un contrato a una compañía diferente, con la finalidad de que dé soporte al sistema y continuar su evolución.

En este caso, es probable que existan discontinuidades en el proceso espiral. Los documentos de requerimientos y diseño quizá no se compartan entre una compañía y otra. Éstas podrían fusionarse o reorganizarse y heredar el software de otras compañías, para luego descubrir que este último tiene que cambiarse.

Cuando la transición del desarrollo a la evolución no es uniforme, el proceso de cambiar el software después de la entrega se conoce como “mantenimiento de software”, el mantenimiento incluye actividades de proceso adicionales, como la comprensión del programa, además de las actividades normales de desarrollo del software.

Rajlich y Bennett propusieron una visión alternativa del ciclo de vida de la evolución del software. En ese modelo, distinguen entre evolución y servicio. La evolución es la fase donde es posible hacer cambios significativos a la arquitectura y la funcionalidad del software. Durante el servicio, los únicos cambios que se realizan son relativamente pequeños.



Durante la evolución, el software se usa con éxito y hay un flujo constante de propuestas de cambios a los requerimientos. Sin embargo, conforme el software se modifica, su estructura tiende a degradarse y los cambios se vuelven más y

más costosos. Esto sucede con frecuencia después de algunos años de uso, cuando también se requieren otros cambios ambientales como el hardware y los sistemas operativos.

En alguna etapa de su ciclo de vida, el software alcanza un punto de transición donde los cambios significativos que implementan nuevos requerimientos se vuelven cada vez menos rentables.

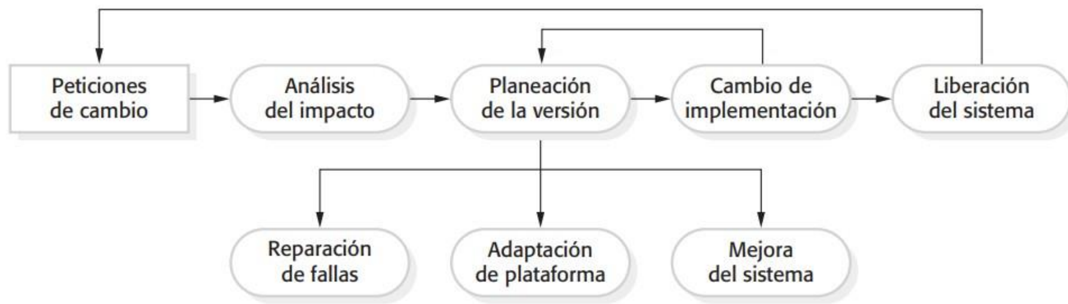
En dicha fase, el software avanza de la evolución al servicio. Durante la fase de servicio, el software todavía es útil y se utiliza, pero sólo se le realizan pequeños cambios tácticos. Durante esta fase, la compañía normalmente considera cómo reemplazar el software. En la fase final, de retiro gradual (Phase-out), el software todavía puede usarse, aunque no se implementan más cambios. Los usuarios tienen que sobrellevar cualquier problema que descubran.

3.1. Proceso de evolución

Los procesos de evolución del software varían dependiendo del tipo de software que se mantiene, de los procesos de desarrollo usados en la organización y de las habilidades de las personas que intervienen.

En algunas organizaciones, la evolución es un proceso informal, donde las solicitudes de cambios provienen sobre todo de conversaciones entre los usuarios del sistema y los desarrolladores. En otras compañías, se trata de un proceso formalizado con documentación estructurada generada en cada etapa del proceso.

La siguiente imagen muestra un panorama del proceso de evolución, el cual incluye actividades fundamentales de análisis del cambio, planeación de la versión, implementación del sistema y su liberación a los clientes.



El costo y el impacto de dichos cambios se valoran para saber qué tanto resultará afectado el sistema por el cambio y cuánto costaría implementarlo. Si los cambios propuestos se aceptan, se planea una nueva versión del sistema. Durante la planeación de la versión se consideran todos los cambios propuestos (reparación de fallas, adaptación y nueva funcionalidad).

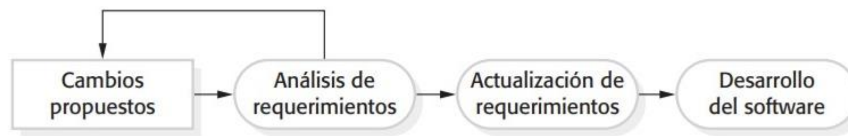
Entonces se toma una decisión acerca de cuáles cambios implementar en la siguiente versión del sistema. Después de implementarse, se valida y se libera una nueva versión del sistema. Luego, el proceso se repite con un conjunto nuevo de cambios propuestos para la siguiente liberación.

Es posible considerar la implementación del cambio como una iteración del proceso de desarrollo, donde las revisiones al sistema se diseñan, se aplican y se ponen a prueba. Sin embargo, una diferencia crítica es que la primera etapa de implementación del cambio puede involucrar la comprensión del programa, sobre todo si los desarrolladores del sistema original no son los responsables de implementar el cambio.

Durante esta fase de comprensión del programa, hay que entender cómo está estructurado, cómo entrega funcionalidad y cómo lo afectaría el cambio propuesto. Se necesita de esta comprensión para asegurarse de que el cambio implementado no cause nuevos problemas, cuando se introduzca al sistema existente.

De manera ideal, la etapa de implementación del cambio de este proceso debe modificar la especificación, el diseño y la implementación del sistema para reflejar los cambios al mismo. Se proponen, analizan y validan los nuevos requerimientos que reflejan los cambios al sistema. Los componentes del sistema se rediseñan e implementan, y el sistema vuelve a probarse. Si es

adecuado, como parte del proceso de análisis de cambio, podrían elaborarse prototipos con los cambios propuestos.



3.2. Evolución dinámica del programa

La dinámica de evolución del programa es el estudio del cambio al sistema. En las décadas de 1970 y 1980, Lehman y Belady (1985) realizaron varios estudios empíricos acerca del cambio al sistema, con una visión para entender más sobre las características de la evolución del software. El trabajo continuó en la década de 1990, conforme Lehman y sus colegas investigaron la importancia de la retroalimentación en los procesos de evolución (Lehman, 1996; Lehman et al., 1998; Lehman et al., 2001). A partir de estos estudios, propusieron las "leyes de Lehman" relacionadas al cambio del sistema.

Lehman y Belady afirman que dichas leyes suelen ser verdaderas para todos los tipos de sistemas de software organizacional grandes. Se trata de sistemas en los cuales los requerimientos se modifican para reflejar las necesidades cambiantes de la empresa.

Las nuevas versiones del sistema son esenciales para que éste proporcione valor al negocio.

Ley	Descripción
Cambio continuo	Un programa usado en un entorno real debe cambiar; de otro modo, en dicho entorno se volvería progresivamente inútil.
Complejidad creciente	A medida que cambia un programa en evolución, su estructura tiende a volverse más compleja. Deben dedicarse recursos adicionales para conservar y simplificar su estructura.
Evolución de programa grande	La evolución del programa es un proceso autorregulador. Los atributos del sistema, como tamaño, tiempo entre versiones y número de errores reportados, son casi invariantes para cada versión del sistema.
Estabilidad organizacional	Durante la vida de un programa, su tasa de desarrollo es aproximadamente constante e independiente de los recursos dedicados al desarrollo del sistema.
Conservación de familiaridad	A lo largo de la existencia de un sistema, el cambio incremental en cada liberación es casi constante.
Crecimiento continuo	La funcionalidad ofrecida por los sistemas tiene que aumentar continuamente para mantener la satisfacción del usuario.
Declive de calidad	La calidad de los sistemas declinará, a menos que se modifiquen para reflejar los cambios en su entorno operacional.
Sistema de retroalimentación	Los procesos de evolución incorporan sistemas de retroalimentación multiagente y multiciclo. Además, deben tratarse como sistemas de retroalimentación para lograr una mejora significativa del producto.

La primera ley afirma que el mantenimiento del sistema es un proceso inevitable. A medida que cambia el entorno del sistema, surgen nuevos requerimientos y el sistema debe modificarse. Cuando el sistema modificado se reintroduce al entorno, promueve más cambios ambientales, de manera que el proceso de evolución comienza de nuevo.

La segunda ley afirma que, conforme cambia un sistema, su estructura se degrada. La única manera de evitar que esto ocurra es invertir en mantenimiento preventivo. Se invierte tiempo mejorando la estructura del software sin agregar nada a su funcionalidad. Evidentemente, esto significa costos adicionales, por encima de los asignados para implementar los cambios requeridos al sistema.

La tercera ley es, quizá, la más interesante y polémica de las leyes de Lehman. Sugiere que los sistemas grandes tienen una dinámica propia que se establece en una etapa temprana del proceso de desarrollo. Esto determina las grandes tendencias del proceso de mantenimiento del sistema y limita el número de cambios posibles al sistema. Lehman y Belady sugieren que esta ley es consecuencia de factores estructurales que influyen en el cambio al sistema y lo restringen, así como de factores organizacionales que afectan el proceso de evolución.

La cuarta ley de Lehman sugiere que la mayoría de los grandes proyectos de programación funcionan en un estado “saturado”. Es decir, un cambio en los recursos o en el personal tiene efectos imperceptibles en la evolución a largo plazo del sistema. Esto es congruente con la tercera ley, que sugiere que la evolución del programa es en gran medida independiente de las decisiones administrativas. Esta ley confirma que los grandes equipos de desarrollo de software con frecuencia son improductivos, porque los gastos en comunicación dominan el trabajo del equipo.

La quinta ley de Lehman se relaciona con los incrementos de cambio en cada versión del sistema. Agregar nueva funcionalidad a un sistema introduce inevitablemente nuevas fallas al mismo. Cuanta más funcionalidad se agregue en cada versión, más fallas habrá. Por consiguiente, un gran incremento en funcionalidad en una versión del sistema significa que esto tendrá que seguir en una versión ulterior, donde se reparen las fallas del nuevo sistema. A dicha versión debe agregarse relativamente poca funcionalidad. Esta ley sugiere que no se deben presupuestar grandes incrementos de funcionalidad en cada versión, sin tomar en cuenta la necesidad de reparación de las fallas.

Las primeras cinco leyes fueron las propuestas iniciales de Lehman; las leyes restantes se agregaron después de un trabajo posterior. Las leyes sexta y séptima son similares y, en esencia, indican que los usuarios de software se volverán cada vez más infortunados con el sistema, a menos que se le mantenga y se le agregue nueva funcionalidad. La ley final refleja el trabajo más reciente sobre los procesos de retroalimentación, aunque todavía no está claro cómo se aplica en el desarrollo de software práctico.

En general, las observaciones de Lehman parecen sensatas. Hay que tomarlas en cuenta cuando se planee el proceso de mantenimiento. Podría suceder que las consideraciones empresariales requieran ignorarlas en algún momento.

Por ejemplo, por razones de marketing, quizá sea necesario realizar muchos cambios significativos al sistema en una sola versión. La consecuencia probable de esto es que tal vez se requieran una o más versiones dedicadas a la reparación del error. A menudo esto se observa en el software de computadoras

personales, cuando una nueva gran versión de alguna aplicación con frecuencia viene seguida por una actualización para reparar un bug.

3.3. Mantenimiento del software

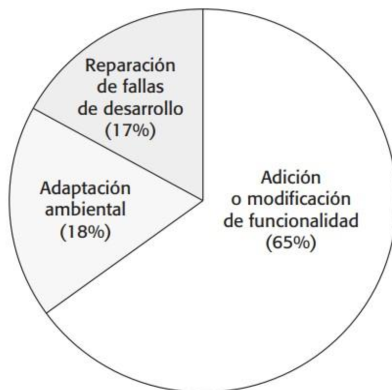
El mantenimiento del software es el proceso general de cambiar un sistema después de que éste se entregó. El término usualmente se aplica a software personalizado, en el que grupos de desarrollo separados intervienen antes y después de la entrega.

Los cambios realizados al software van desde los simples para corregir errores de codificación, los más extensos para corregir errores de diseño, hasta mejorías significativas para corregir errores de especificación o incorporar nuevos requerimientos. Los cambios se implementan modificando los componentes del sistema existentes y agregándole nuevos componentes donde sea necesario.

Existen tres tipos de mantenimiento de software:

1. Reparaciones de fallas: Los errores de codificación por lo general son relativamente baratos de corregir; los errores de diseño son más costosos, ya que quizás impliquen la reescritura de muchos componentes del programa. Los errores de requerimientos son los más costosos de reparar debido a que podría ser necesario un extenso rediseño del sistema.
2. Adaptación ambiental: Este tipo de mantenimiento se requiere cuando algún aspecto del entorno del sistema, como el hardware, la plataforma operativa del sistema u otro soporte, cambia el software. El sistema de aplicación tiene que modificarse para lidiar con dichos cambios ambientales.
3. Adición de funcionalidad: Este tipo de mantenimiento es necesario cuando varían los requerimientos del sistema, en respuesta a un cambio organizacional o empresarial. La escala de los cambios requeridos en el software suele ser mucho mayor que en los otros tipos de mantenimiento.

Distribución del esfuerzo de mantenimiento



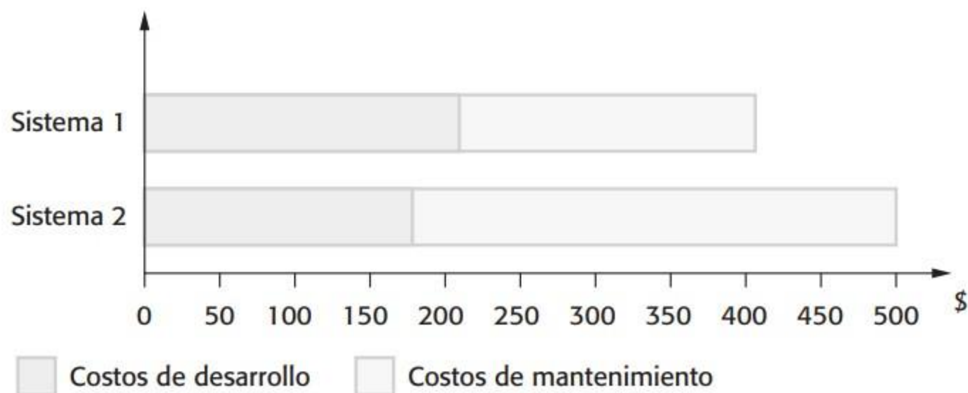
Los costos relativos de mantenimiento y del nuevo desarrollo varían de un dominio de aplicación a otro. Los costos de mantenimiento para sistemas de aplicación empresarial son ampliamente comparables con los costos de desarrollo del sistema. Para sistemas embebidos de tiempo real, los costos de mantenimiento son hasta cuatro veces mayores que los costos de desarrollo.

Los requerimientos de alta fiabilidad y rendimiento de dichos sistemas significan que los módulos deben estar estrechamente ligados y, por lo tanto, son difíciles de cambiar. Aunque estas estimaciones tienen más de 25 años de antigüedad, es improbable que las distribuciones de costos para diferentes tipos de sistema hayan cambiado significativamente.

Por lo general, resulta efectivo en costo invertir esfuerzo en el diseño y la implementación de un sistema, con la finalidad de reducir los costos de cambios futuros. Agregar nueva funcionalidad después de la entrega es costoso porque toma tiempo aprender cómo funciona el sistema y analizar el impacto de los cambios propuestos.

Por lo tanto, es posible que el trabajo realizado durante el desarrollo para hacer que el software sea más fácil de entender, y de cambiar, reduzca los costos de evolución. Las buenas técnicas de ingeniería de software, como la especificación precisa, el uso de desarrollo orientado a objetos y la administración de la configuración, contribuyen a reducir los costos de mantenimiento.

Costos de desarrollo y mantenimiento



En general, resulta más costoso agregar funcionalidad después de que un sistema está en operación, que implementar la misma funcionalidad durante el desarrollo. Las razones son:

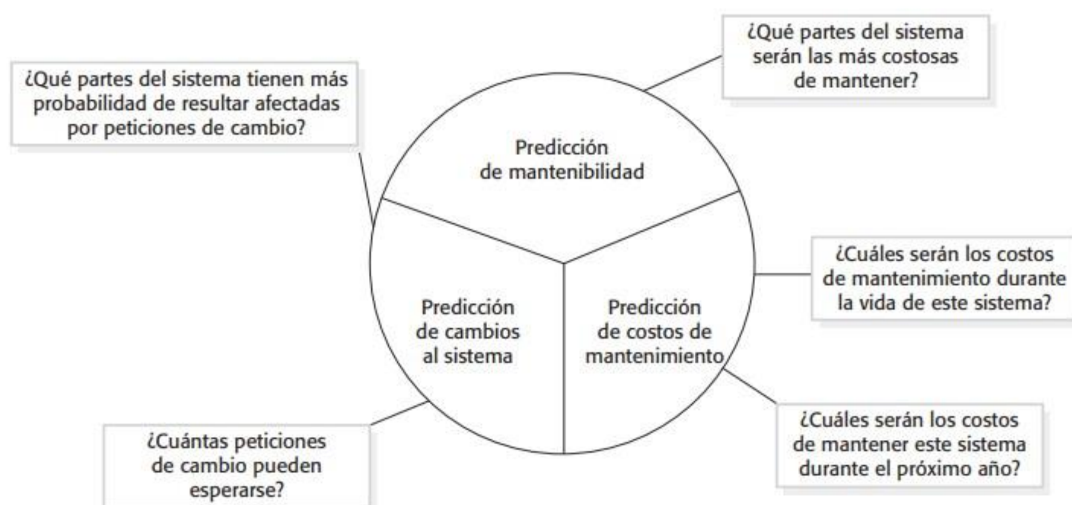
1. Estabilidad del equipo: Después de que un sistema se entrega, es normal que el equipo de desarrollo se separe y que los individuos trabajen en nuevos proyectos. El nuevo equipo o los individuos responsables del mantenimiento del sistema no entienden el sistema o los antecedentes de las decisiones de diseño del mismo. Necesitan emplear tiempo para comprender el sistema existente, antes de implementar cambios en él.
2. Práctica de desarrollo deficiente: El contrato para mantener un sistema por lo general está separado del contrato de desarrollo del sistema. El contrato de mantenimiento puede otorgarse a una compañía diferente, y no al desarrollador original del sistema. Este factor, junto con la falta de estabilidad del equipo, indica que no hay incentivo para que un equipo de desarrollo escriba software mantenible. Si el equipo de desarrollo puede buscar atajos para ahorrar esfuerzo durante el desarrollo, para ellos vale la pena hacerlo, incluso si esto significa que el software sea más difícil de cambiar en el futuro.
3. Habilidades del personal: El personal de mantenimiento con frecuencia es relativamente inexperto y no está familiarizado con el dominio de aplicación. El mantenimiento tiene una mala imagen entre los ingenieros de software. Lo ven como un proceso que requiere menos habilidades que el desarrollo de sistemas y se asigna a menudo al personal más

novato. Más aún, los sistemas antiguos pueden estar escritos en lenguajes de programación obsoletos. Es posible que el personal de mantenimiento no tenga mucha experiencia de desarrollo en estos lenguajes y debe aprenderlos para mantener el sistema.

4. Antigüedad y estructura del programa: Conforme se realizan cambios al programa, su estructura tiende a degradarse. En consecuencia, a medida que los programas envejecen, se vuelven más difíciles de entender y cambiar. Algunos sistemas se desarrollaron sin técnicas modernas de ingeniería de software. Es posible que nunca hayan estado bien estructurados y tal vez estuvieron optimizados para eficiencia y no para comprensibilidad. La documentación del sistema puede estar perdida o ser inconsistente. Es posible que los sistemas antiguos no se hayan sujetado a una gestión rigurosa de configuración, de modo que se desperdicia tiempo para encontrar las versiones correctas de los componentes del sistema a cambiar.

Predicción de mantenimiento

Los gerentes aborrecen las sorpresas, sobre todo si derivan en costos inesperadamente elevados. Por consiguiente, se debe tratar de predecir qué cambios deben proponerse al sistema y qué partes del sistema es probable que sean las más difíciles de mantener. También hay que tratar de estimar los costos de mantenimiento globales para un sistema durante cierto lapso de tiempo.



Predecir el número de peticiones de cambio para un sistema requiere un entendimiento de la relación entre el sistema y su ambiente externo. Algunos sistemas tienen una relación muy compleja con su ambiente externo, y los cambios a dicho entorno inevitablemente derivarán en cambios al sistema.

Para evaluar las relaciones entre un sistema y su ambiente, se debe valorar:

1. El número y la complejidad de las interfaces del sistema: Cuanto más grande sea el número de interfaces y más complejas sean dichas interfaces, más probable será que se requieran cambios de interfaz conforme se propongan nuevos requerimientos.
2. El número de requerimientos de sistema inherentemente inestables: Es más probable que los requerimientos que reflejan políticas y procedimientos de la organización sean más inestables que los requerimientos que se basan en características de un dominio estable.
3. Los procesos empresariales donde se usa el sistema: A medida que evolucionan los procesos empresariales, generan peticiones de cambio del sistema. Cuantos más procesos use un sistema, habrá más demandas de cambio del mismo

Reingeniería de software

El proceso de evolución del sistema incluye comprender el programa que debe cambiarse y, luego, implementar dichos cambios. Sin embargo, muchos sistemas, especialmente los sistemas heredados más antiguos, son difíciles de entender y de cambiar. Es posible que los programas se hayan optimizado para rendimiento o utilización de espacio a expensas de la claridad o, con el tiempo, la estructura inicial del programa quizá se corrompió debido a una serie de cambios.

Para hacer que los sistemas de software heredados sean más sencillos de mantener, se pueden someter a reingeniería para mejorar su estructura y entendimiento. La reingeniería puede implicar volver a documentar el sistema, refactorizar su arquitectura, traducir los programas a un lenguaje de programación moderno, y modificar y actualizar la estructura y los valores de los datos del sistema.

La funcionalidad del software no cambia y, normalmente, conviene tratar de evitar grandes cambios a la arquitectura de sistema.

Hay dos beneficios importantes de la reingeniería respecto de la sustitución:

1. Reducción del riesgo: Hay un alto riesgo en el desarrollo de software empresarial crítico. Pueden cometerse errores en la especificación del sistema o tal vez haya problemas de desarrollo. Las demoras en la introducción del nuevo software podrían significar que la empresa está perdida y que se incurrirá en costos adicionales.
2. Reducción de costos: El costo de la reingeniería puede ser significativamente menor que el costo de desarrollar software nuevo. Ulrich (1990) cita un ejemplo de un sistema comercial para el cual los costos de reimplementación se estimaron en $50 millones. El sistema tuvo reingeniería exitosa por $12 millones. Con la tecnología de software moderna, el costo relativo de reimplementación quizá sea menor que esto, pero todavía superará considerablemente los costos de la reingeniería.

En el proceso de reingeniería la entrada al proceso es un sistema heredado, en tanto que la salida es una versión mejorada y reestructurada del mismo programa.

Las actividades en este proceso de reingeniería son las siguientes:

1. Traducción del código fuente: Con una herramienta de traducción, el programa se convierte de un lenguaje de programación antiguo a una versión más moderna del mismo lenguaje, o a un lenguaje diferente.
2. Ingeniería inversa: El programa se analiza y se extrae información de él. Esto ayuda a documentar su organización y funcionalidad. De nuevo, este proceso es, por lo general, completamente automatizado.
3. Mejoramiento de la estructura del programa: La estructura de control del programa se analiza y modifica para facilitar su lectura y comprensión, lo cual suele estar parcialmente automatizado, pero se requiere regularmente alguna intervención manual.

4. Modularización del programa: Las partes relacionadas del programa se agrupan y, donde es adecuado, se elimina la redundancia. En algunos casos, esta etapa implicará refactorización arquitectónica (por ejemplo, un sistema que use muchos almacenes de datos diferentes puede refactorizarse para usar un solo depósito). Éste es un proceso manual.
5. Reingeniería de datos: Los datos procesados por el programa cambian para reflejar cambios al programa. Esto puede significar la redefinición de los esquemas de bases de datos y convertir las bases de datos existentes a la nueva estructura. Por lo general, hay que limpiar los datos. Esto implica encontrar y corregir errores, eliminar registros duplicados, etcétera. Hay herramientas disponibles para auxiliar en la reingeniería de datos.

El proceso de reingeniería

3.4. Administración de sistemas heredados

Cada vez con mayor frecuencia, las compañías empiezan a entender que el proceso de desarrollo del sistema es un proceso de todo el ciclo de vida y que no es útil una separación artificial entre el desarrollo del software y su mantenimiento. Sin embargo, todavía existen muchos sistemas heredados que son sistemas empresariales críticos. Éstos tienen que extenderse y adaptarse a las cambiantes prácticas del comercio electrónico.

La mayoría de las organizaciones, por lo general, tienen un portafolio de sistemas heredados, que se usan con un presupuesto limitado para mantenimiento y actualización. Deben decidir cómo obtener el mejor retorno de la inversión. Esto requiere hacer una valoración realista de sus sistemas heredados y, luego, decidir acerca de la estrategia más adecuada para hacer evolucionar dichos sistemas.

Existen cuatro opciones estratégicas:

1. Desechar completamente el sistema: Esta opción debe elegirse cuando el sistema no vaya a realizar una aportación efectiva a los procesos empresariales. Por lo general, esto ocurre cuando los procesos empresariales cambiaron desde la instalación del sistema y no se apoyan más en el sistema heredado.
2. Dejar sin cambios el sistema y continuar el mantenimiento regular: Esta opción debe elegirse cuando el sistema todavía se requiera, pero sea bastante estable y los usuarios del sistema hagan relativamente pocas peticiones de cambio.
3. Someter el sistema a reingeniería para mejorar su mantenibilidad: Esta opción debe elegirse cuando la calidad del sistema se haya degradado por el cambio y todavía se propone un nuevo cambio al sistema. Este proceso podría incluir el desarrollo de nuevos componentes de interfaz, de modo que el sistema original logre trabajar con otros sistemas más recientes.
4. Sustituir todo o parte del sistema con un nuevo sistema: Esta opción tiene que elegirse cuando factores como hardware nuevo signifiquen que el viejo sistema no pueda continuar en operación o donde sistemas comerciales (off-the-shelf) permitirían al nuevo sistema desarrollarse a un costo razonable. En muchos casos se adopta una estrategia de sustitución evolutiva, en la cual grandes componentes del sistema se sustituyen con sistemas comerciales, y otros componentes se reutilizan siempre que sea posible.

Cuando se valora un sistema heredado, tiene que observarse desde las perspectivas empresarial y técnica.

Desde la perspectiva empresarial, se debe decidir si la compañía necesita realmente o no el sistema. Desde una perspectiva técnica, hay que valorar la calidad del software de aplicación, así como el hardware y software de soporte del sistema. Luego, se usa una combinación del valor empresarial y la calidad del sistema, para informar la decisión acerca de qué hacer con el sistema heredado.

Factores usados en la valoración del entorno

Factor	Preguntas
Estabilidad del proveedor	¿El proveedor todavía está en operación? ¿El proveedor es financieramente estable y es probable que continúe en operación? Si el proveedor ya no está en el negocio, ¿alguien más mantiene los sistemas?
Tasa de falla	¿El hardware tiene una alta tasa de fallas reportadas? ¿El software de soporte se cae y fuerza el reinicio del sistema?
Edad	¿Qué antigüedad tienen el hardware y el software? Cuanto más viejos sean el hardware y el software de soporte, más obsoletos serán. Quizá todavía funcionen correctamente, pero podría haber beneficios económicos y empresariales significativos al moverse hacia un sistema más moderno.
Rendimiento	¿El rendimiento del sistema es adecuado? ¿Los problemas de rendimiento tienen un efecto relevante sobre los usuarios del sistema?
Requerimientos de soporte	¿Qué apoyo local requieren el hardware y el software? Si existen altos costos asociados con este apoyo, valdría la pena considerar la sustitución del sistema.
Costos de mantenimiento	¿Cuáles son los costos del mantenimiento de hardware y de las licencias del software de soporte? El hardware más antiguo puede tener costos de mantenimiento más altos que los sistemas modernos. El software de soporte quizá tenga costos altos por licencia anual.
Interoperabilidad	¿Hay problemas de interfaz entre el sistema y otros sistemas? ¿Se pueden usar los compiladores, por ejemplo, con las versiones actuales del sistema operativo? ¿Se requiere emulación de hardware?

Factor	Preguntas
Entendimiento	¿Cuán difícil es entender el código fuente del sistema actual? ¿Cuán complejas son las estructuras de control que se utilizan? ¿Las variables tienen nombres significativos que reflejan su función?
Documentación	¿Qué documentación del sistema está disponible? ¿La documentación está completa, y es consistente y actualizada?
Datos	¿Existe algún modelo de datos explícito para el sistema? ¿En qué medida los datos se duplican a través de los archivos? ¿Los datos usados por el sistema están actualizados y son consistentes?
Rendimiento	¿El rendimiento de la aplicación es adecuado? ¿Los problemas de rendimiento tienen un efecto significativo sobre los usuarios del sistema?
Lenguaje de programación	¿Hay compiladores modernos disponibles para el lenguaje de programación usado para desarrollar el sistema? ¿El lenguaje de programación todavía se usa para el desarrollo de nuevos sistemas?
Administración de la configuración	¿Todas las versiones de la totalidad de las partes del sistema se administran mediante un sistema de administración de la configuración? ¿Existe una descripción explícita de las versiones de componentes que se usan en el sistema actual?
Datos de prueba	¿Existen datos de prueba para el sistema? ¿Hay un registro de pruebas de regresión realizadas cuando se agregaron nuevas características al sistema?
Habilidades del personal	¿Hay personal disponible que tenga las habilidades para mantener la aplicación? ¿Hay personal disponible que tenga experiencia con el sistema?

3.5. Propiedades de confiabilidad

Conforme los sistemas de cómputo se insertan profundamente en las vidas empresariales y personales, se incrementan los problemas que derivan de las fallas del sistema y del software. Una falla del software del servidor en una empresa de comercio electrónico podría conduc ir a dicha compañía hacia una gran pérdida de ingresos y, posiblemente, también de clientes. Un error de software en un sistema de control embebido en un automóvil provocaría costosas devoluciones de ese modelo por reparación y, en los peores casos, sería un factor que contribuya a los accidentes.

Hay cuatro dimensiones principales de la confiabilidad, como se indica en la figura

1. **Disponibilidad**: De manera informal, la disponibilidad de un sistema es la probabilidad de que en un momento dado éste funcionará, ejecutará y ofrecerá servicios útiles a los usuarios.

2. **Fiabilidad**: De manera informal, la fiabilidad de un sistema es la probabilidad, durante un tiempo determinado, de que el sistema brindará correctamente servicios como espera el usuario.
3. **Protección**: De modo no convencional, la protección de un sistema es un juicio de cuán probable es que el sistema causará daños a las personas o su ambiente.
4. **Seguridad**: Informalmente, la seguridad de un sistema es un juicio de cuán probable es que el sistema pueda resistir intrusiones accidentales o deliberadas.

Además de estas cuatro propiedades básicas de confiabilidad, también se pueden considerar las siguientes:

1. **Reparabilidad**: Las fallas del sistema son inevitables; no obstante, el desajuste causado por la falla podría minimizarse siempre que el sistema logre repararse rápidamente. Para que ello ocurra, se puede diagnosticar el problema, acceder al componente que falló y hacer cambios para corregir dicho componente. La reparabilidad en software se mejora cuando la organización que usa el sistema tiene acceso al código fuente y cuenta con las habilidades para cambiarlo. El software de fuente abierta facilita esta labor, aunque la reutilización de componentes suele dificultarlo más.
2. **Mantenibilidad**: Mientras se usan los sistemas, surgen nuevos requerimientos y es importante mantener la utilidad de un sistema al cambiarlo para acomodar estos nuevos requerimientos. El software mantenible es aquel que económicamente se adapta para lidiar con los nuevos requerimientos, y donde existe una baja probabilidad de que los cambios insertarán nuevos errores en el sistema.
3. **Supervivencia**: Un atributo muy importante para sistemas basados en Internet es la supervivencia. La supervivencia es la habilidad de un sistema para continuar entregando servicio en tanto está bajo ataque y mientras que, potencialmente, parte del sistema se deshabilita. El trabajo sobre supervivencia se enfoca en la identificación de los componentes clave del sistema y en la garantía de que ellos puedan entregar un servicio mínimo. Para mejorar la supervivencia se usan tres estrategias:

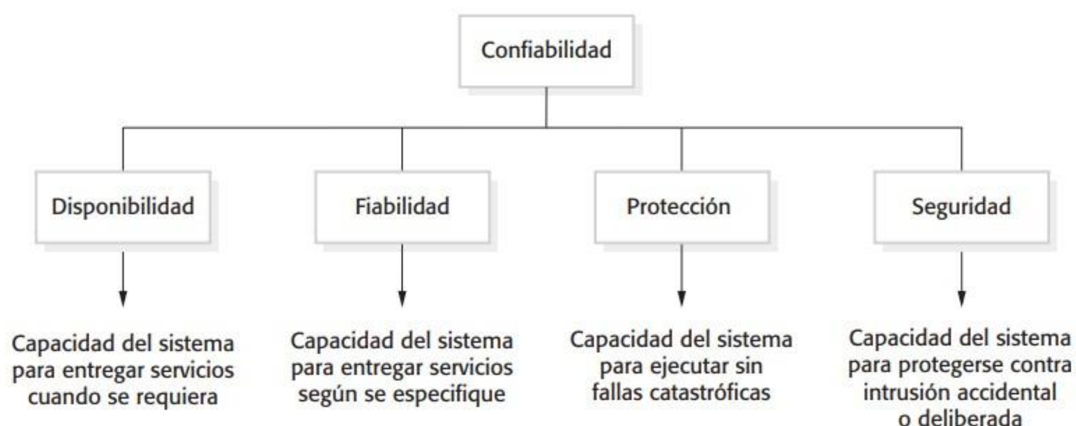
resistencia al ataque, reconocimiento del ataque y recuperación del daño causado por un ataque.

La noción de confiabilidad del sistema como propiedad que abarca la disponibilidad, seguridad, fiabilidad y protección, se introdujo porque estas propiedades están estrechamente relacionadas. La operación segura del sistema depende, por lo general, de que éste se halle disponible y opere de manera fiable. Un sistema se volvería no fiable cuando un curioso corrompa sus datos.

Los ataques de negación de servicio sobre un sistema tienen la intención de comprometer la disponibilidad de un sistema. Si un sistema se infecta con un virus, entonces no se estaría seguro de su fiabilidad o su protección, dado que el virus podría cambiar su comportamiento.

Por lo tanto, para desarrollar software confiable, es necesario garantizar que:

- Se evite la entrada de errores accidentales en el sistema durante la especificación y el desarrollo del software.
- Se diseñen procesos de verificación y validación que sean efectivos en descubrimiento de errores residuales que afecten la confiabilidad del sistema.
- Se desarrollen mecanismos de protección contra ataques externos que comprometan la disponibilidad o la seguridad del sistema.
- Se configuren correctamente el sistema utilizado y el software de apoyo para su entorno operacional.



3.5.1. Disponibilidad y fiabilidad

Con una mayor demanda de infraestructuras confiables y de alto rendimiento diseñadas para servir a sistemas críticos, los términos disponibilidad y fiabilidad no podrían ser más populares. Si bien manejar una mayor carga del sistema es una preocupación común, la disminución del tiempo de inactividad y la eliminación de puntos únicos de falla son igual de importantes.

En informática, el término disponibilidad se utiliza para describir el período de tiempo en que un servicio está disponible, así como el tiempo requerido por un sistema para responder a una solicitud realizada por un usuario.

Al configurar sistemas de producción robustos, minimizar el tiempo de inactividad y las interrupciones del servicio a menudo es una alta prioridad.

Independientemente de cómo tus sistemas y software son confiables, pueden ocurrir problemas que pueden derribar tus aplicaciones o tus servidores. La implementación de alta disponibilidad de la infraestructura es una estrategia útil para reducir el impacto de este tipo de eventos.

Los sistemas de alta disponibilidad pueden recuperarse de la falla del servidor o componente automáticamente.

En conclusión:

La disponibilidad se trata de la capacidad de un servicio, de unos datos o de un sistema, a ser accesible y utilizable por los usuarios (o procesos) autorizados cuando estos lo requieran. Supone que la información pueda ser recuperada en el momento en que se necesite, evitando su pérdida o bloqueo.

Fiabilidad

No hay duda que la fiabilidad de un programa de computadora es un elemento importante de su calidad general. Si un programa falla frecuentemente en su funcionamiento, no importa si el resto de los factores de calidad son aceptables. La fiabilidad del software, a diferencia de otros factores de calidad, puede ser medida o estimada mediante datos históricos o de desarrollo.

La fiabilidad del software se define en términos estadísticos como la probabilidad de operación libre de fallos de un programa de computadora es un entorno determinado y durante un tiempo específico Siempre que se habla de fiabilidad, surge una pregunta fundamental.

3.5.2. Protección

Los sistemas críticos para la protección son aquellos en los que resulta esencial que la operación del sistema sea segura en todo momento; esto es, el sistema nunca debe dañar a las personas o a su entorno, incluso cuando falle. Los ejemplos de sistemas críticos para la protección incluyen los sistemas de control y monitorización en las aeronaves, los sistemas de control de procesos en plantas químicas y farmacéuticas, y los sistemas de control de automotores.

El control de hardware de los sistemas críticos para la protección es más fácil de implementar y analizar que el control del software. A pesar de ello, ahora se construyen sistemas de tal complejidad que no pueden controlarse tan sólo con el hardware.

El control del software es esencial debido a la necesidad de manejar gran cantidad de sensores y actuadores con leyes de control complejas. Por ejemplo, una aeronave militar avanzada, aerodinámicamente inestable, requiere ajuste continuo, controlado por software, de sus superficies de vuelo para garantizar que no se desplome.

La fiabilidad y protección del sistema se relacionan, pero un sistema fiable puede ser inseguro y viceversa. El software todavía suele comportarse de tal forma que el comportamiento resultante del sistema conduzca a un accidente.

Hay cuatro razones por las que los sistemas de software que son fiables no necesariamente son seguros:

1. Nunca se puede tener una total certeza de que un sistema de software esté libre de fallas en el desarrollo y sea tolerante a los mismos. Las fallas en el desarrollo no detectadas pueden estar inactivas durante mucho

tiempo y las fallas en la operación del software pueden ocurrir después de muchos años de operación infalible.

2. La especificación podría estar incompleta en cuanto a que no describe el comportamiento requerido del sistema en algunas situaciones críticas.
3. El mal funcionamiento del hardware origina que el sistema se comporte de forma impredecible, y presente al software con un entorno no anticipado. Cuando los componentes se hallan cerca de la falla física, éstos pueden comportarse de manera errática y generar señales fuera de los rangos para el manejo del software.
4. Los operadores del sistema pueden generar entradas que no son individualmente incorrectas, pero que, en ciertas situaciones, conducirían a un mal funcionamiento del sistema. Un ejemplo anecdótico de esto sucedió cuando el tren de aterrizaje de una aeronave colapsó mientras la nave estaba en tierra. Al parecer, un técnico presionó un botón que ordenó al software de gestión de utilidades elevar el tren de aterrizaje. El software realizó la instrucción del mecánico a la perfección. Sin embargo, el sistema debía desactivar el comando a menos que el avión estuviera en el aire.

3.5.3. Seguridad

La seguridad es un atributo del sistema que refleja la habilidad de éste para protegerse a sí mismo de ataques externos, que podrían ser accidentales o deliberados. Estos ataques externos son posibles puesto que la mayoría de las computadoras de propósito general ahora están en red y, en consecuencia, son accesibles a personas externas.

Ejemplos de ataques pueden ser la instalación de virus y caballos de Troya, el uso sin autorización de servicios del sistema o la modificación no aprobada de un sistema o de sus datos. Si real mente se quiere un sistema seguro, es mejor no conectarlo a Internet.

Siendo así, sus problemas de seguridad estarán limitados a garantizar que usuarios autorizados no abusen del sistema. Sin embargo, en la práctica, existen

enormes beneficios del acceso en red para los sistemas más grandes, de modo que desconectarlos de Internet no es efectivo en costo. Para algunos sistemas, la seguridad es la dimensión más importante de confiabilidad del sistema.

Los sistemas militares, los de comercio electrónico y los que requieren procesamiento e intercambio de información confidencial deben diseñarse de modo que logren un alto nivel de seguridad. Por ejemplo, si un sistema de reservaciones de una aerolínea no está disponible, causará inconvenientes y ciertas demoras en la emisión de boletos. Sin embargo, si el sistema es inseguro, entonces un atacante podría borrar todos los libros y sería prácticamente imposible que continuaran las operaciones normales de la aerolínea.

Desde luego, dichas amenazas son interdependientes. Si un ataque provoca que el sistema no esté disponible, entonces no podrá actualizar la información que cambia con el tiempo. Esto significa que la integridad del sistema puede estar comprometida; entonces, tal vez deba desmantelarse para reparar el problema. Por ello, se reduce la disponibilidad del sistema.

En la práctica, la mayoría de las vulnerabilidades en los sistemas sociotécnicos obedecen a fallas humanas más que a problemas técnicos. Las personas eligen contraseñas fáciles de descifrar o escriben sus contraseñas en lugares sencillos de encontrar. Los administradores de sistemas cometen errores al establecer control de acceso o archivos de configuración, en tanto que los usuarios no instalan o usan software de protección. Sin embargo, se debe tener mucho cuidado cuando se clasifique un problema como un error de usuario.

Los problemas humanos reflejan con frecuencia malas decisiones de diseño de sistemas que requieren, por ejemplo, cambios de contraseñas a menudo (de modo que los usuarios escriben sus contraseñas) o complejos mecanismos de configuración.

Los controles que se deben implementar para mejorar la seguridad del sistema son comparables con los de la fiabilidad y protección:

1. Evitar la vulnerabilidad: Controles cuya intención sea garantizar que los ataques no tengan éxito. Aquí, la estrategia es diseñar el sistema de modo que se eviten los problemas de seguridad. Por ejemplo, los sistemas

militares sensibles no están conectados a redes públicas, de forma que es imposible el acceso externo. También hay que pensar en la encriptación como un control basado en la evitación. Cualquier acceso no autorizado a datos encriptados significa que el atacante no puede leerlos. En la práctica, es muy costoso y consume mucho tiempo romper una encriptación fuerte.

2. Detectar y neutralizar ataques: Controles cuya intención sea detectar y repeler ataques. Dichos controles implican la inclusión de funcionalidad en un sistema que monitoriza su operación y verifica patrones de actividad inusuales. Si se detectan, entonces se toman acciones, como desactivar partes del sistema, restringir el acceso a ciertos usuarios, etcétera.
3. Limitar la exposición y recuperación: Controles que soportan la recuperación de los problemas. Éstos varían desde estrategias de respaldo automatizadas y “réplica” de la información, hasta pólizas de seguros que cubran los costos asociados con un ataque exitoso al sistema.

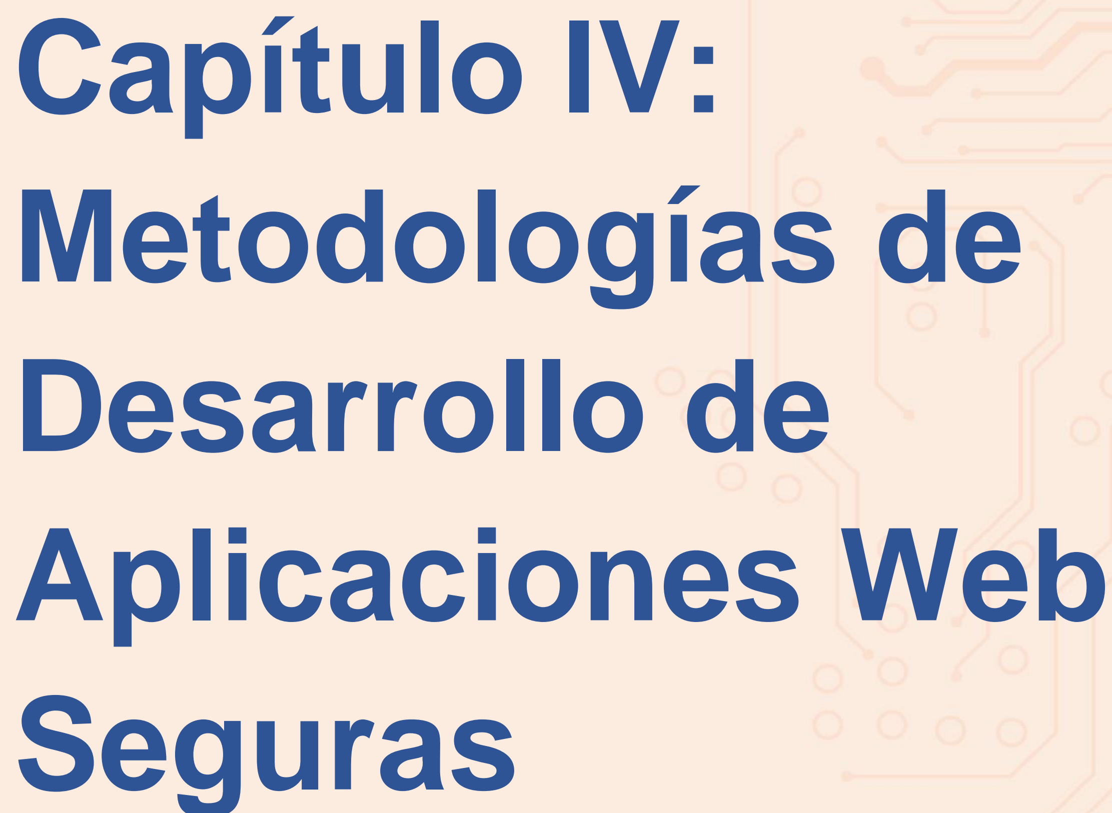
Sin un nivel razonable de seguridad, no se puede confiar en la disponibilidad, fiabilidad y protección de un sistema. Los métodos para certificar la disponibilidad, fiabilidad y seguridad suponen que el software en operación es el mismo que el software que se instaló originalmente.

Si el sistema es atacado y el software se compromete de alguna forma (por ejemplo, si el software se modifica al incluir un gusano), entonces ya son insostenibles los argumentos de fiabilidad y protección.

Los errores en el desarrollo de un sistema pueden conducir a lagunas de seguridad. Si un sistema no responde a entradas inesperadas o si no se verifican los límites vectoriales, entonces los atacantes aprovecharían tales debilidades para conseguir acceso al sistema.

Terminología de seguridad

Término	Definición
Activo	Algo de valor que debe protegerse. El activo sería el sistema de software en sí o los datos usados por dicho sistema.
Exposición	Posible pérdida o daño a un sistema de cómputo. Esto sería la pérdida o el daño a los datos, o bien, una pérdida de tiempo y esfuerzo si es necesaria la recuperación después de una violación a la seguridad.
Vulnerabilidad	Una debilidad en un sistema basado en computadora que puede aprovecharse para causar pérdida o daño.
Ataque	Aprovechamiento de una vulnerabilidad del sistema. Por lo general, es desde afuera del sistema y es un intento deliberado por causar cierto daño.
Amenaza	Circunstancias que tienen potencial para causar pérdida o daño. Se puede pensar en ellas como una vulnerabilidad de sistema que está sujeta a un ataque.
Control	Medida de protección que reduce la vulnerabilidad de un sistema. La encriptación es un ejemplo de control que reduce la vulnerabilidad de un sistema de control de acceso débil.



Capítulo IV: Metodologías de Desarrollo de Aplicaciones Web Seguras

Metodologías de Desarrollo de Aplicaciones Web Seguras

4.1. OWASP: Open Web Application Security Project

El Open Web Application Security Project (OWASP) está dedicado a la búsqueda y la lucha contra las vulnerabilidades en el software.

La OWASP Foundation es una organización sin ánimo de lucro que proporciona la infraestructura y apoya a este trabajo.

- La participación es gratuita y abierta para todos
- Aquí todo es gratuito y de código abierto
- Objetivos: crear herramientas, documentación y estándares relacionados con la seguridad en aplicaciones

El Proyecto de seguridad de aplicaciones web abiertas (OWASP) es una fundación sin fines de lucro que brinda orientación sobre cómo desarrollar, comprar y mantener aplicaciones de software confiables y seguras.

OWASP se destaca por su popular lista Top 10 de vulnerabilidades de seguridad de aplicaciones web.

La lista OWASP Top 10 de problemas de seguridad se basa en el consenso entre la comunidad de desarrolladores sobre los principales riesgos de seguridad.

Se actualiza cada pocos años a medida que cambian los riesgos y surgen otros nuevos. La lista explica las fallas de seguridad de aplicaciones web más peligrosas y brinda recomendaciones para tratarlas.

OWASP busca educar a los desarrolladores, diseñadores, arquitectos y dueños de negocios sobre los riesgos asociados con las vulnerabilidades de seguridad de aplicaciones web más comunes. OWASP admite productos de seguridad comerciales y de código abierto. Es conocido como un foro en el que los expertos en seguridad y los profesionales de la tecnología de la información pueden establecer contactos y desarrollar su experiencia.

El OWASP Top 10 es una lista de los 10 riesgos de seguridad más importantes que afectan a las aplicaciones web. Se revisa cada pocos años para reflejar los cambios en la industria y el riesgo. La lista tiene descripciones de cada categoría de riesgos de seguridad de aplicaciones y métodos para remediarlos.

OWASP compila la lista a partir de encuestas comunitarias, datos aportados sobre vulnerabilidades y exploits comunes y bases de datos de vulnerabilidades.

La primera versión de la lista OWASP Top 10 se publicó en 2003. Las actualizaciones siguieron en 2004, 2007, 2010, 2013 y 2017. La actualización más reciente se publicó en 2021.

Los riesgos que forman parte de la lista en cualquier momento se identifican por su rango en la lista y el año de la lista. Entonces, por ejemplo, el principal riesgo de seguridad en la lista más reciente es el control de acceso roto. Se le asigna el identificador de A01:2021. "A" es para AppSec, seguido de su rango en la lista, el símbolo ":" y el año.

El sistema de numeración ayuda a hacer referencia a versiones anteriores de riesgos, especialmente cuando el nombre de una categoría ha cambiado o las categorías se han fusionado o ampliado.

OWASP top 10

	2003	2021
A1	Unvalidated parameters	Broken access control
A2	Broken access control	Cryptographic failures
A3	Broken account and session management	Injection
A4	Cross-site scripting flaws	Insecure design
A5	Buffer overflows	Security misconfiguration
A6	Command injection flaws	Vulnerable and outdated components
A7	Error-handling problems	Identification and authentication failures
A8	Insecure use of cryptography	Software and data integrity failures
A9	N/A	Security logging and monitoring failures
A10	Web application server misconfiguration	Server-side request forgery

A01:2021 Control de acceso roto. Con estas vulnerabilidades, los atacantes pueden eludir los controles de acceso elevando sus propios permisos o de alguna otra manera. Este enfoque brinda a los usuarios no autorizados acceso

a datos o sistemas. Los controles de acceso rotos se han convertido en la principal categoría de riesgo de seguridad para las aplicaciones web, pasando del quinto puesto en 2017. Esta categoría se creó en 2017 mediante la fusión de otras dos categorías: control de acceso a funciones faltantes y referencias directas a objetos inseguras.

A02:2021 Fallos criptográficos. Estos riesgos ocurren cuando los métodos criptográficos no se utilizan adecuadamente para proteger los datos. Estas vulnerabilidades incluyen el uso de cifrados criptográficos obsoletos, protocolos criptográficos que no se implementan correctamente y otros problemas relacionados con los controles criptográficos. Esta categoría se conocía anteriormente como exposición de datos confidenciales. OWASP cambió el nombre para reflejar la importancia de las fallas criptográficas para permitir la exposición de información confidencial.

A03:2021 Inyección. Estas vulnerabilidades permiten a los atacantes insertar datos en una aplicación que incluye comandos maliciosos, redirige los datos a un sitio web malicioso o cambia la propia aplicación. El tipo de falla más común, la inyección de lenguaje de consulta estructurado todavía representa un vector importante para los ataques. La corrección de los ataques de inyección consiste en autenticar explícitamente todos los datos que no son de confianza, especialmente los datos enviados por los usuarios finales. Esta categoría se amplió para incluir la categoría de secuencias de comandos entre sitios en la lista de los 10 principales de 2021.

A04:2021 Diseño inseguro. Los riesgos en esta categoría provienen de fallas en el diseño de la arquitectura del sistema. Estos problemas ocurren cuando una aplicación está diseñada en torno a procesos inseguros. Por ejemplo, estos problemas surgen cuando una aplicación se desarrolla utilizando un proceso de autenticación que no es seguro o un sitio web no está diseñado para evitar los bots.

A05:2021 Configuración incorrecta de seguridad. En esta categoría, son los problemas con la configuración de seguridad de una aplicación los que facilitan los ataques. Por ejemplo, es posible que una aplicación no filtre correctamente los paquetes entrantes y permita el uso de un ID de usuario, una contraseña o

una autorización predeterminados. Esta categoría se amplió en 2021 para incluir la categoría de entidades externas Extensible Markup Language.

A06:2021 Componentes vulnerables y obsoletos. Estos riesgos surgen cuando los desarrolladores utilizan componentes de software con vulnerabilidades en las aplicaciones. También aparecen cuando el software no tiene parches, está desactualizado o está comprometido de manera similar. Los componentes vulnerables incluyen bibliotecas, marcos, interfaces de programación de aplicaciones (API) u otros módulos. Si el sistema operativo subyacente o un intérprete de programa no está parcheado, podría causar estos problemas. Las API y las bibliotecas de software desactualizadas también pueden crear estos problemas para la aplicación.

A07:2021 Fallos de identificación y autenticación. Estas vulnerabilidades incluyen problemas de autenticación que permiten el relleno de credenciales y ataques de fuerza bruta. La categoría también incluye aplicaciones que no usan la autenticación multifactor y no invalidan las sesiones de usuario que han expirado o no están activas. En 2017, estos riesgos se denominaron autenticación rota. La categoría cambió de nombre en 2021 para incluir autenticación rota y administración de sesión rota.

A08:2021 Fallas de integridad de software y datos. Esta categoría abarca el código de la aplicación y la infraestructura que no protege completamente el software o la integridad de los datos. Por ejemplo, pueden surgir problemas si no se utilizan firmas digitales cuando se instalan actualizaciones de software. Esta categoría se amplió en 2021 para incluir la categoría de deserialización insegura.

A09:2021 Registro de seguridad y fallas de monitoreo. Estas vulnerabilidades ocurren cuando un sistema no se supervisa adecuadamente para detectar y responder a los ataques y se mantienen registros de estos eventos. Antes de 2021, esta categoría se denominaba registro y seguimiento insuficientes. El cambio de nombre refleja la expansión de la categoría para incluir más tipos de fallas de monitoreo y registro.

A10:2021 Falsificación de solicitud del lado del servidor. Las aplicaciones deben realizar una validación adecuada de los recursos proporcionados por el

usuario para evitar estos ataques. Los actores de amenazas pueden usar estas vulnerabilidades para hacer que las aplicaciones accedan a sitios web maliciosos.

4.2. Open-Source Security Testing Methodology Manual (OSSTMM)

El Manual de la Metodología Abierta de Comprobación de la Seguridad (OSSTMM, Open Source Security Testing Methodology Manual) es uno de los estándares profesionales más completos y comúnmente utilizados en Auditorías de Seguridad para revisar la Seguridad de los Sistemas desde Internet. Incluye un marco de trabajo que describe las fases que habría que realizar para la ejecución de la auditoría. Se ha logrado gracias a un consenso entre más de 150 expertos internacionales sobre el tema, que colaboran entre sí mediante Internet. Se encuentra en constante evolución y actualmente se compone de las siguientes fases:

Sección A -Seguridad de la Información

- Revisión de la Inteligencia Competitiva
- Revision de Privacidad
- Recolección de Documentos

Sección B - Seguridad de los Procesos

- Testeo de Solicitud
- Testeo de Sugerencia Dirigida
- Testeo de las Personas Confiables

Sección C - Seguridad en las tecnologías de Internet

- Logística y Controles
- Exploración de Red
- Identificación de los Servicios del Sistema
- Búsqueda de Información Competitiva
- Revisión de Privacidad
- Obtención de Documentos

- Búsqueda y Verificación de Vulnerabilidades
- Testeo de Aplicaciones de Internet
- Enrutamiento
- Testeo de Sistemas Confiados
- Testeo de Control de Acceso
- Testeo de Sistema de Detección de Intrusos
- Testeo de Medidas de Contingencia
- Descifrado de Contraseñas
- Testeo de Denegación de Servicios
- Evaluación de Políticas de Seguridad

Sección D - Seguridad en las Comunicaciones

- Testeo de PBX
- Testeo del Correo de Voz
- Revisión del FAX
- Testeo del Modem

Sección E - Seguridad Inalámbrica

- Verificación de Radiación Electromagnética (EMR)
- Verificación de Redes Inalámbricas [802.11]
- Verificación de Redes Bluetooth
- Verificación de Dispositivos de Entrada Inalámbricos
- Verificación de Dispositivos de Mano Inalámbricos
- Verificación de Comunicaciones sin Cable
- Verificación de Dispositivos de Vigilancia Inalámbricos
- Verificación de Dispositivos de Transacción Inalámbricos
- Verficación de RFID
- Verificación de Sistemas Infrarrojos
- Revisión de Privacidad

Sección F - Seguridad Física

- Revisión de Perímetro
- Revisión de monitoreo
- Evaluación de Controles de Acceso

- Revisión de Respuesta de Alarmas
- Revisión de Ubicación
- Revisión de Entorno

Referencias
Bibliográficas

Referencias Bibliográficas

Álvarez, A. (2012). Manual Imprescindible de Métodos ágiles y Scrum. España: Anaya Multimedia.

Boehm, B. W. (1979). "Software engineering; R & D Trends and defense needs." In Research Directions in Software Technology. Wegner, P. (ed.). Cambridge, Mass.: MIT Press. 1–9

Hopkins, R. y Jenkins, K. (2008). Eating the IT Elephant: Moving from Greenfield Development to Brownfield. Boston, Mass.: IBM Press.

Kaner, C. (2003). "The power of 'What If . . .' and nine ways to fuel your imagination: Cem Kaner on scenario testing". Software Testing and Quality Engineering, 5 (5), 16–22.

Kendall, E. (2013). Análisis y Diseño de Sistemas. México: Pearson Educación.

Lehman, M. M. (1996). "Laws of Software Evolution Revisited". Proc. European Workshop on Software Process Technology (EWSPT'96), Springer-Verlag. 108–24.

Lehman, M. M. y Belady, L. (1985). Program Evolution: Processes of Software Change. London: Academic Press

Massol, V. y Husted, T. (2003). JUnit in Action. Greenwich, Conn.: Manning Publications Co.

Piattini, M. (2012). Calidad de Sistemas de Información. México: Alfaomega.

Pressman, R. (2010). Ingeniería del Software: Un enfoque práctico. México: Mc Graw Hill.

Raymond, E. S. (2001). The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. Sebastopol. Calif.: O'Reilly Media, Inc.

Sánchez, S. (2012). Ingeniería del Software: Un enfoque desde la guía Swebok. México: Alfaomega.

Sommerville, I. (2014). Ingeniería de Software. México: Pearson Educación.

St. Laurent, A. (2004). Understanding Open Source and Free Software Licensing. Sebastopol, Calif.: O'Reilly Media Inc.

Ulrich, W. M. (1990). "The Evolutionary Growth of Software Reengineering and the Decade Ahead". American Programmer, 3 (10).

Resumen

En este libro se investigó los procesos de implementación, pruebas y mantenimiento de la ingeniería del software y se documentó los diferentes tipos de pruebas de software para validar el correcto funcionamiento de un software, se documentó también los procesos de evolución de un software para lograr un correcto mantenimiento del software. Este libro tiene como objetivo aplicar los conceptos y teorías de validación del software con el fin de crear de software de calidad, confiable, que se encuentre disponible, protegido y seguro, para que el profesional de sistemas informáticos sea capaz de gestionar con eficiencia y en ambientes éticos y de responsabilidad proyectos inmersos en la creación software. Se investigó varios libros llegando a realizar un compendio de las partes más importantes y se llegó a la conclusión de que la ingeniería del software proporciona varias técnicas para la evaluación y validación del software y varias metodologías para el desarrollo de software confiable y seguro que el profesional puede utilizar dependiendo las diferentes necesidades, situaciones o proyectos a desarrollar

Editorial Grupo AEA

www.grupo-aea.com

www.editorialgrupo-aea.com

Grupo de Asesoría Empresarial & Académica

Grupoaea.ecuador

Editorial Grupo AEA

ISBN: 978-9942-7014-8-0