

Tabla 2.7 Pruebas de caja blanca en diferentes niveles de prueba

Nivel de prueba	Ejemplo de estructura a cubrir Código
Componente	fuelle (p. ej., declaraciones, ramas, decisiones, rutas)
Integración de componentes	Gráfico de llamadas (es decir, eventos de llamada de una función por otra), un conjunto de funciones API
Sistema	modelo de proceso de negocio
Integración de sistema	Gráfico de llamadas a nivel de servicios ofrecidos por los sistemas comunicantes.
Aceptación	Modelo de árbol de la estructura del menú.

el número de declaraciones ejecutables cubiertas por las pruebas dividido por todas las declaraciones ejecutables en un código determinado.

La métrica de cobertura definida anteriormente toma valores entre 0 y 1 (porque tanto A como B son positivos y A siempre es menor o igual a B). La forma más común de presentación de esta métrica es su forma porcentual:

$$\text{cobertura} = \frac{A}{B} \times 100\%$$

Por ejemplo, si cubrimos 5 de 20 declaraciones de código ejecutable con nuestras pruebas, entonces logramos la cobertura $\frac{5}{20} = 0,25$ o, en otras palabras, $(5/20) \times 100\% = 25\%$.

En el nivel de prueba de componentes, la cobertura del código está determinada por el porcentaje del código de un componente que ha sido probado. Este valor se puede medir en términos de varios aspectos del código (elementos de cobertura), como el porcentaje de declaraciones ejecutables o el porcentaje de resultados de decisiones probados en un módulo determinado. Los tipos de cobertura anteriores se denominan colectivamente cobertura de código. En el nivel de prueba de integración de componentes, las pruebas de caja blanca se pueden realizar sobre la base de la arquitectura del sistema (por ejemplo, interfaces entre módulos) y la cobertura estructural se puede medir en términos del porcentaje de interfaces probadas. La Tabla 2.7 muestra ejemplos de estructuras que se pueden utilizar para realizar pruebas de caja blanca en diferentes niveles de prueba.

Es posible que se necesiten habilidades o conocimientos especiales para diseñar y ejecutar pruebas de caja blanca, como conocimiento de la construcción de código (que permite, por ejemplo, el uso de herramientas para medir la cobertura del código), conocimiento de cómo almacenar datos (que permite, por ejemplo, la evaluación de consultas a bases de datos), o cómo utilizar herramientas para medir la cobertura e interpretar correctamente los resultados que generan.

Es fundamental resaltar que lograr un nivel de cobertura arbitrariamente alto no garantiza una calidad arbitrariamente alta. Incluso una cobertura del 100% de todas las declaraciones ejecutables no garantiza que cada declaración ejecutable realmente funcione como se espera. Es un medio para medir la integridad de la prueba, en lugar de la calidad del sistema bajo prueba.

2.2.2.4 Pruebas de caja negra

Las pruebas de caja negra ³ se basan en especificaciones (es decir, información externa al objeto de prueba). El objetivo principal de las pruebas de caja negra es verificar que el comportamiento del sistema se ajuste al comportamiento descrito en la especificación.

Para obtener más información sobre las pruebas de caja negra, consulte la Sección. 4.2, donde analizamos en detalle las técnicas de prueba de caja negra.

2.2.2.5 Niveles de prueba versus tipos de prueba

Los niveles y tipos de pruebas son independientes entre sí. Aunque en la práctica será cierto que ciertos tipos de pruebas tienden a ocurrir en ciertos niveles de prueba específicos, en general, cualquier tipo de prueba se puede ejecutar en cualquier nivel de prueba. Esto se muestra en el siguiente ejemplo.

Ejemplo Estamos probando una aplicación web de una tienda electrónica de venta de libros. Los usuarios registrados, después de iniciar sesión, tienen acceso a un catálogo con función de búsqueda (que incluye búsqueda por título, autor, categoría de libro, rango de precios). Pueden agregar artículos seleccionados al carrito de compras y luego proceder al pago. A continuación se describen algunos ejemplos de pruebas funcionales, no funcionales, de caja negra y de caja blanca (en aras de la simplicidad, solo damos descripciones breves de las pruebas, no describimos los casos de prueba en detalle).

Ejemplos de pruebas funcionales (y de caja negra):

- Validar el registro del usuario en el sistema. • Verificar si es posible registrar un usuario con un nombre existente en el sistema. • Compruebe si es posible registrar un usuario con una dirección de correo electrónico existente en el sistema.
- Consultar compra estándar y pago correcto. • Comprobar si al poner un libro en el carrito de compras, esa copia queda bloqueada en el sistema y otros usuarios no pueden agregarlo a sus carritos. • Comprobar que transcurrido un tiempo determinado y fijo, si no se realiza el pago, la sesión caduca y todos los libros del carrito vuelven a la tienda y quedan disponibles para el resto de usuarios.

Ejemplos de pruebas no funcionales:

- Consultar el tiempo de respuesta tras consultar un catálogo de 1 millón de artículos. • Prueba de vulnerabilidad de inyección SQL³ mediante el cuadro de búsqueda. • Comprobar el tiempo de respuesta del sistema en la situación de ejecución simultánea de una consulta realizada por mil usuarios.

³ La inyección SQL es un ataque a un sitio web o aplicación web en el que se agrega código de lenguaje SQL a un campo en un formulario para obtener acceso a una cuenta o cambiar datos.

Ejemplos de pruebas de caja blanca:

- Verificar la cobertura del extracto del componente de registro de clientes (intención: verificar todas las rutas y posibles excepciones y errores que debe manejar el sistema).
- Comprobar el funcionamiento de todos los elementos del menú principal (abarcando la estructura de la página web).
- Probar todas las funciones de la interfaz relacionadas con el sistema de pago electrónico.

Cada uno de los tipos de pruebas enumerados anteriormente se puede realizar en cualquier nivel de prueba. Para ilustrar esto, los siguientes son ejemplos de pruebas funcionales, no funcionales, de caja negra y de caja blanca que se realizan en todos los niveles de prueba para una aplicación bancaria. El primer grupo son las pruebas funcionales (y de caja negra):

- Para las pruebas de componentes, las pruebas están diseñadas para reflejar cómo un módulo debe calcular el interés compuesto.
- Para las pruebas de integración de componentes, las pruebas están diseñadas para reflejar cómo la información de la cuenta capturada en la interfaz de usuario se transfiere a la capa de lógica empresarial.
- Para las pruebas del sistema, se están diseñando pruebas que reflejen cómo los titulares de cuentas pueden solicitar una línea de crédito para sobregiros.
- Para las pruebas de integración de sistemas, las pruebas están diseñadas para reflejar cómo un sistema determinado verifica la solvencia de un titular de cuenta utilizando un microservicio externo.
- Para las pruebas de aceptación, las pruebas están diseñadas para reflejar cómo un empleado del banco aprueba o rechaza una solicitud de préstamo.

El siguiente grupo contiene ejemplos de pruebas no funcionales:

- Para las pruebas de componentes, se diseñan pruebas de rendimiento que evalúan el número de ciclos de CPU necesarios para realizar cálculos complejos de la cantidad total de interés.
- Para las pruebas de integración de componentes, las pruebas de seguridad están diseñadas para detectar vulnerabilidades de seguridad relacionadas con desbordamientos del búfer de datos pasados desde la interfaz de usuario a la capa de lógica empresarial.
- Para las pruebas del sistema, las pruebas de portabilidad están diseñadas para verificar que la presentación La capa funciona en todos los navegadores y dispositivos móviles compatibles.
- Para las pruebas de integración del sistema, las pruebas de confiabilidad están diseñadas para evaluar la resiliencia del sistema en caso de una falta de respuesta del microservicio utilizado para verificar la solvencia.
- Para las pruebas de aceptación, se diseñan pruebas de usabilidad para evaluar las características de accesibilidad para personas con discapacidad, aplicadas a la interfaz de procesamiento de préstamos por parte del banco.

El siguiente grupo contiene ejemplos de pruebas de caja blanca:

- Para las pruebas de componentes, las pruebas están diseñadas con el objetivo de garantizar una cobertura completa de las instrucciones y decisiones del código en todos los módulos que realizan cálculos financieros.

- Para las pruebas de integración de componentes, las pruebas están diseñadas para ver cómo cada pantalla de la interfaz de visualización del navegador pasa datos a la siguiente pantalla y a la capa de lógica empresarial.
- Para las pruebas del sistema, las pruebas están diseñadas para garantizar la cobertura de posibles secuencias de páginas web que se muestran al solicitar una línea de crédito.
- Para las pruebas de integración de sistemas, se diseñan pruebas que verifican todos los tipos posibles de consultas enviadas al microservicio utilizado para la verificación de crédito.
- Para las pruebas de aceptación, las pruebas están diseñadas para cubrir todas las estructuras soportadas y rangos de valores para archivos de datos financieros utilizados en transferencias interbancarias.

Esta sección proporciona ejemplos de todos los tipos de pruebas realizadas en todos los niveles, pero no todo el software necesita incluir todos los tipos de pruebas en todos los niveles. Lo importante es que se ejecuten las pruebas adecuadas en cada nivel; esto es especialmente cierto para el nivel más temprano en el que se produce un tipo de prueba.

2.2.3 Pruebas de confirmación y pruebas de regresión

Una vez que se han realizado cambios en el sistema para corregir defectos o agregar o modificar funcionalidades, se deben realizar pruebas para confirmar que los cambios realmente solucionaron el defecto o implementaron correctamente la funcionalidad relevante, sin causar consecuencias adversas imprevistas. El programa de estudios de Foundation Level distingue entre dos tipos de pruebas de este tipo: pruebas de confirmación (también llamadas reevaluaciones) y pruebas de regresión.

Pruebas de confirmación Las

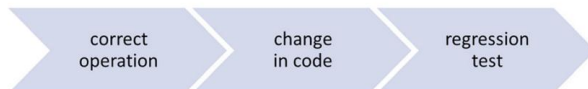
pruebas de confirmación (también conocidas como repuebas) se realizan en relación con la ocurrencia de una falla y la reparación del defecto asociado. La función de las pruebas de confirmación es verificar que el defecto efectivamente ha sido reparado (ver Fig. 2.17). Normalmente, una nueva prueba es la misma prueba que reveló la falla, realizada nuevamente, después de que se haya reparado la falla, pero a veces la prueba de confirmación puede ser una prueba diferente, por ejemplo, si el defecto informado fue una falta de funcionalidad. El mínimo absoluto al realizar pruebas de confirmación es ejecutar los pasos que causaron la falla anteriormente. Si se supera la prueba, podemos asumir que el defecto se ha solucionado.

Como nunca se sabe cuándo ocurrirá una falla y cuándo se reparará, la ejecución de nuevas pruebas no se puede planificar con anticipación. Sin embargo, se debe asignar tiempo suficiente para las pruebas de confirmación durante la planificación (ver Capítulo 5).

Fig. 2.17 Prueba de confirmación



Fig. 2.18 Prueba de regresión



Pruebas de regresión Las

pruebas de regresión verifican si un cambio (arreglo o modificación) realizado en un componente del código afectará inadvertidamente el comportamiento de otras partes del código en el mismo componente, en otros componentes del mismo sistema o incluso en otros sistemas (ver figura 2.18). Además, se deben tener en cuenta los cambios en el entorno, como la introducción de una nueva versión del sistema operativo o del sistema de gestión de bases de datos. Los efectos secundarios no deseados mencionados anteriormente se denominan regresiones (de la palabra "regresión", que significa deterioro de algo), y las pruebas de regresión implican volver a ejecutar pruebas para detectar estas regresiones.

Dado que las pruebas de regresión se realizan con mayor frecuencia cuando se agrega otra funcionalidad o al final de cada iteración, estas pruebas generalmente, a diferencia de las pruebas de confirmación, se pueden programar perfectamente. Dado que las pruebas de regresión deben realizarse con frecuencia y además evolucionan con bastante lentitud, son candidatas naturales para la automatización. La automatización de las pruebas de regresión ahorra tiempo y esfuerzo, ya que los evaluadores pueden dedicar el tiempo ahorrado a otras actividades, como crear y ejecutar nuevas pruebas. La automatización de este tipo de pruebas debería comenzar temprano en el proyecto.

Un problema práctico común con las pruebas de regresión es que su número crece muy rápidamente. Después de cierto punto, tardan bastante en ejecutarse durante la noche y los resultados esperan a los evaluadores a la mañana siguiente. A veces, sin embargo, un conjunto de pruebas de regresión ni siquiera se puede ejecutar de la noche a la mañana. En este caso, se debe tomar una decisión para optimizar la ejecución de la prueba de regresión. Las posibles soluciones incluyen:

- **Priorización:** realizar primero las pruebas más importantes y relevantes; aquellos que no se ejecutan son menos importantes, por lo que hay poco riesgo de que un fallo importante pase desapercibido.
- **Reducción:** elimina algunas pruebas del conjunto de regresión; El criterio para la eliminación puede basarse, por ejemplo, en la capacidad de la prueba para detectar fallas: si una prueba detecta fallas con bastante frecuencia, definitivamente debe permanecer, mientras que si una prueba nunca ha detectado ninguna falla, puede eliminarse del conjunto de pruebas.
- **Estrategia mixta:** las pruebas más importantes se ejecutan de forma programada, mientras que la parte que no se puede realizar debido a consideraciones de tiempo se ejecuta con menos frecuencia, como cada dos iteraciones.

Las pruebas de confirmación y las pruebas de regresión se pueden realizar en todos los niveles de prueba, especialmente en modelos SDLC iterativos e incrementales; las nuevas funciones, los cambios en las funciones existentes y la refactorización dan como resultado frecuentes cambios de código que implican pruebas adecuadas. Dada la continua evolución del sistema, las pruebas de confirmación y las pruebas de regresión son muy importantes, especialmente para los sistemas relacionados con IoT, donde los objetos individuales (por ejemplo, dispositivos) se actualizan o reemplazan con frecuencia.

Ejemplo Al probar una tienda electrónica de libros (consulte la sección anterior), un evaluador provocó una falla: el sistema permitió que un usuario se registrara con la misma dirección de correo electrónico que ya tiene otro usuario previamente registrado. Llamemos "prueba A" a la prueba que detectó este problema. El evaluador presentó un informe de defecto y se encargó de otras actividades. Específicamente, realizó la prueba B (intentar pagar un pedido estándar), que pasó. En la siguiente iteración, se implementó un nuevo componente para informar las operaciones de pago. Después de estos cambios, el evaluador volvió a realizar la prueba B para ver si el componente afectaba la funcionalidad de pago en sí. Momentos después, el evaluador recibió un mensaje de que el defecto que había informado anteriormente se había solucionado. El evaluador volvió a ejecutar la prueba A, que esta vez pasó. El probador cerró el defecto.

En el ejemplo anterior, la prueba A y la prueba B se ejecutaron dos veces, en orden: A, B, B, A.

- La primera ejecución de la prueba A y la primera ejecución de la prueba B no se aplican ni a las pruebas de regresión ni a las pruebas de confirmación, ya que estas pruebas se realizaron por primera vez.
- La segunda ejecución de la prueba B es un ejemplo de prueba de regresión, ya que la intención del evaluador es comprobar si la implementación de un nuevo componente ha roto algo en el módulo de pago.
- La segunda ejecución de la prueba A es un ejemplo de prueba de confirmación, ya que la intención del evaluador es verificar que la prueba A, que causó una falla anteriormente, pasará ahora, después de la corrección.

Las pruebas relacionadas con cambios se pueden realizar en todos los niveles de prueba. Refiriéndose al ejemplo de aplicación bancaria de la Sección. 2.2.2.4, podemos distinguir los siguientes ejemplos de pruebas relacionadas con cambios:

- Para las pruebas de componentes, las pruebas de regresión automatizadas se diseñan por componente (estas pruebas luego se incluirán en el marco de integración continua).
- Para las pruebas de integración de componentes, las pruebas están diseñadas para confirmar la efectividad de las correcciones relacionadas con defectos en las interfaces a medida que dichas correcciones se colocan en el repositorio de código.
- Para las pruebas del sistema, todas las pruebas para un flujo de trabajo determinado se realizan nuevamente si alguna de las pantallas cubiertas por ese flujo de trabajo ha cambiado.
- Para las pruebas de integración del sistema, se realizan diariamente pruebas de la aplicación que funciona con el microservicio para verificación de crédito (como parte de la implementación continua de este microservicio).
- Para las pruebas de aceptación, todas las pruebas fallidas anteriormente se realizan nuevamente después de la solución del defecto detectado en las pruebas de aceptación.

2.3 Pruebas de mantenimiento

FL-2.3.1 (K2) Resumir las pruebas de mantenimiento y sus desencadenantes

El momento en que se lanza un sistema a un cliente no es el final, sino sólo un estado intermedio del desarrollo del producto. Aunque muchos administradores de TI no se dan cuenta, muchos estudios empíricos de ingeniería de software muestran que el costo de mantenimiento del sistema suele ser significativamente mayor que el costo de desarrollarlo. Esto se debe a un hecho simple: normalmente lleva mucho más tiempo utilizar un software que desarrollarlo, y durante su uso pueden ocurrir muchas situaciones diferentes que requieren cambios o correcciones en el software. Ejemplos de tales situaciones son:

- La necesidad de un parche para el sistema debido al descubrimiento de un ataque relacionado con vulnerabilidad de seguridad
- Agregar, eliminar o modificar características del programa
- Reparar el defecto que causó la falla reportada por el usuario
- La necesidad de archivar datos debido al desmantelamiento del software
- La necesidad de mover el sistema a un nuevo entorno, causado, por ejemplo, por actualizar la versión del sistema operativo

Una vez implementado en un entorno de producción, el software o sistema requiere mayor mantenimiento. Varios tipos de cambios, como los descritos anteriormente, son prácticamente inevitables. Además, el mantenimiento es necesario para mantener o mejorar las características de calidad no funcionales requeridas del software o sistema a lo largo de su ciclo de vida, especialmente en términos de rendimiento, compatibilidad, confiabilidad, seguridad y portabilidad.

Es bueno recordar que las pruebas de mantenimiento se realizan después del lanzamiento del software, durante su uso operativo. Las pruebas realizadas antes de que el software se entregue al cliente no se pueden considerar pruebas de mantenimiento. El mantenimiento, por definición, se aplica a un producto que ya está en uso.

Después de realizar cualquier cambio en la fase de mantenimiento, se deben realizar pruebas de mantenimiento. El propósito de las pruebas de mantenimiento es tanto verificar que el cambio se ha implementado exitosamente como detectar posibles efectos secundarios (p. ej., regresiones) en las partes sin cambios del sistema (es decir, generalmente en la mayoría de las áreas del sistema). Por lo tanto, las pruebas de mantenimiento incluyen tanto las partes del sistema que han sido cambiadas como las partes no modificadas que pueden haber sido afectadas por los cambios. El mantenimiento se puede realizar tanto de forma programada (en relación con las nuevas versiones) como de forma no programada (en relación con las correcciones urgentes).

A la luz de las consideraciones anteriores, está claro que el tipo más común de prueba de mantenimiento será la prueba de regresión. Sin embargo, este no es el único tipo de prueba de mantenimiento. Por ejemplo, si se va a reemplazar el software por otro, se deben realizar pruebas de archivado o de migración de datos. Estas pueden ser pruebas completamente nuevas que realizaremos solo por primera vez en toda la historia, tal vez de años, de uso de la aplicación.

Una versión de mantenimiento puede requerir la ejecución de pruebas de mantenimiento en múltiples niveles de prueba y utilizando diferentes tipos de pruebas, según el alcance de los cambios realizados.

El alcance de las pruebas de mantenimiento incluye:

- El nivel de riesgo asociado con el cambio (por ejemplo, la medida en que el cambio el área de software se comunica con otros componentes o sistemas) • El tamaño del sistema existente • El tamaño del cambio realizado

Hay varias razones para realizar el mantenimiento del software y, por lo tanto, realizar pruebas de mantenimiento. Esto incluye tanto cambios planificados como no planificados. Los eventos que desencadenan el mantenimiento se pueden dividir en las siguientes categorías:

- **Modificación.** Esta categoría incluye, entre otras, mejoras planificadas (p. ej., en forma de nuevas versiones de software), cambios correctivos y de emergencia, cambios en el entorno operativo (p. ej., actualizaciones planificadas del sistema operativo o de la base de datos), actualizaciones de software para Software COTS y parches que solucionan defectos y vulnerabilidades de seguridad.
- **Actualización o migración.** Esta categoría incluye, entre otras, la transición de una plataforma a otra, lo que puede implicar pruebas del nuevo entorno y el software modificado o pruebas de conversión de datos (al migrar datos de otra aplicación a un sistema mantenido).
- **Jubilación.** Esta categoría se refiere a la situación en la que una aplicación se retira del uso. Esto puede requerir probar la migración o el archivado de datos si es necesario almacenarlos durante un período prolongado. También puede ser necesario probar los procedimientos de recuperación después de archivar durante un período prolongado.

Además, es posible que sea necesario incluir pruebas de regresión para garantizar que las funciones que permanecen en uso sigan funcionando correctamente.

En el caso de sistemas relacionados con IoT, pueden ser necesarias pruebas de mantenimiento tras la introducción de elementos completamente nuevos o modificados en el sistema, como dispositivos de hardware o servicios de software. Durante las pruebas de mantenimiento de dichos sistemas, se pone especial énfasis en las pruebas de integración en varios niveles (por ejemplo, en los niveles de red y de aplicación) y en los aspectos de seguridad, especialmente con respecto a los datos personales.

Ejemplo Una empresa produce un sistema CRM (Customer Relationship Management). Este sistema se basa en una base de datos relacional, que no fue muy bien diseñada y contiene muchos datos redundantes. La aplicación hace referencia a la base de datos a través de un conjunto de funciones bien definidas, en lugar de directamente mediante la construcción de consultas.

El arquitecto decidió que se debía mejorar la estructura de la base de datos: el esquema de la base de datos debía transformarse a la llamada tercera forma normal. El cambio requerirá instalación en cada terminal de usuario.

Este es un evento desencadenante de mantenimiento, relacionado con modificación (cambio correctivo) y, en cierto sentido, migración (ya que se modificará la estructura de la base de datos). Por lo tanto, deberás realizar las siguientes pruebas de mantenimiento relacionadas con la instalación del parche (antes de su implementación):

- Pruebas de regresión directamente relacionadas con el uso de funciones que se comunican con el base de datos
- Pruebas de migración y recuperación de datos y retorno a la antigua estructura de la base de datos en caso de que por alguna razón la nueva estructura de la base de datos en el ordenador del cliente no funcione como debería.

Análisis de impacto

El análisis de impacto nos permite evaluar los cambios realizados en la versión mantenida en términos tanto de los efectos previstos como de los efectos secundarios esperados o potenciales y nos permite identificar áreas del sistema que se verán afectadas por los cambios. Además, puede ayudar a identificar el impacto del cambio en las pruebas existentes. Los efectos secundarios del cambio y las áreas del sistema que pueden verse afectadas por él deben probarse para detectar regresión, una actividad que puede estar precedida por una actualización de las pruebas existentes afectadas por el cambio.

Se puede realizar un análisis de impacto antes de realizar un cambio para determinar si el cambio realmente debería realizarse (dadas las posibles consecuencias para otras áreas del sistema).

Realizar un análisis de impacto puede resultar difícil si:

- Las especificaciones (p. ej., requisitos comerciales, historias de usuarios, arquitectura) están desactualizadas o no están disponibles
- Los casos de prueba no se han documentado o están desactualizados
- La trazabilidad bidireccional entre las pruebas y la base de la prueba no se ha establecido

El soporte de herramientas es inexistente o inadecuado

- Las personas involucradas no tienen conocimiento del campo y/o del sistema en cuestión
- Se ha prestado muy poca atención a las características de calidad del sistema en términos de mantenibilidad durante el desarrollo del software.

Las dificultades para realizar análisis de impacto son bastante comunes. Esto se debe a que el mantenimiento suele realizarse meses o años después del lanzamiento del software y, a menudo, las personas que crearon la versión original del sistema ya no están en el equipo u organización y la documentación está desactualizada y obsoleta.

Ejemplo. Es necesario implementar un parche de software, relacionado con un cambio en la ley sobre cálculo de impuestos. El cambio afecta al componente X. Utilizando la trazabilidad bidireccional, resulta que el módulo X está asociado a los riesgos R1, R3, R6 y R8, todos los cuales tienen un nivel de riesgo muy alto (es decir, son críticos).

(continuado)

Los riesgos restantes (R2, R4, R5 y R7) tienen un nivel bajo (es decir, son insignificantes). Además, el componente X se comunica con otros cuatro componentes entre los ocho módulos del sistema. Este simple análisis de riesgo muestra que el cambio tiene un alto riesgo y su impacto en el sistema es significativo.

Preguntas de muestra

Pregunta 2.1

(FL-2.1.1, K2)

Eres tester en un proyecto para desarrollar software de piloto automático de aviones. El proyecto se lleva a cabo según el modelo V.

¿Cuál de las siguientes oraciones describe MEJOR las consecuencias asociadas con elegir este modelo?

- R. El énfasis en las pruebas estará en técnicas de prueba basadas en la experiencia.
- B. El equipo de prueba no realizará pruebas estáticas.
- C. Al finalizar la primera iteración, se creará un prototipo funcional del sistema. disponible.
- D. En las fases iniciales, los evaluadores participan en revisiones de requisitos, análisis de pruebas y diseño de prueba.

Elija una respuesta.

Pregunta 2.2

(FL-2.1.2, K1)

Una buena práctica de prueba dice que para cada actividad de desarrollo, debe haber una actividad de prueba asociada. ¿Qué modelo SDLC muestra este principio explícitamente?

- A. Modelo iterativo.
- B. Modelo espiral de Boehm.
- C. Melé.
- D. Modelo V.

Elija una respuesta.

Pregunta 2.3

(FL-2.1.3, K1)

¿Qué enfoque de prueba primero utiliza criterios de aceptación para historias de usuarios como base para derivar casos de prueba?

- R. TDD.
- B. ATDD.
- C. FDD.
- D. BDD.

Elija una respuesta.

Pregunta 2.4

(FL-2.1.4, K2)

¿Cuál de las siguientes afirmaciones describe cómo el enfoque DevOps respalda las PRUEBAS?

- R. DevOps permite acortar el ciclo de lanzamiento del software mediante el uso de un Generación automatizada de datos de prueba para pruebas de componentes.
- B. DevOps permite comentarios rápidos al desarrollador sobre la calidad del código que comprometerse con el repositorio.
- C. DevOps permite la generación automática de casos de prueba para el nuevo código enviado por un desarrollador al repositorio.
- D. DevOps permite reducir el tiempo necesario para la planificación de lanzamientos y la planificación de iteraciones.

Elija una respuesta.

Pregunta 2.5

(FL-2.1.5, K2)

¿Cuál de los siguientes es un ejemplo del uso del enfoque de desplazamiento a la izquierda?

- A. Aplicación del desarrollo impulsado por pruebas de aceptación (ATDD).
- B. Basar las pruebas en pruebas exploratorias.
- C. Creación de prototipos de GUI durante la fase de obtención de requisitos.
- D. Monitoreo continuo de la calidad del producto después de su entrega al cliente.

Elija una respuesta.

Pregunta 2.6

(FL-2.1.6, K2)

¿Cuál de las siguientes oraciones describe mejor las actividades de un evaluador que asiste a una reunión retrospectiva?

- R. El evaluador solo plantea problemas relacionados con la prueba. Todos los demás temas serán planteados por cualquiera de los demás participantes.
- B. El evaluador es un observador, asegurándose de que la reunión siga los principios de retrospectivas y los valores del enfoque ágil.
- C. El evaluador proporciona retroalimentación y otra información sobre todas las actividades realizadas por el equipo durante la iteración completa.
- D. El evaluador recopila la información proporcionada por otros participantes de la reunión para diseñar pruebas para la siguiente iteración basada en esta información.

Elija una respuesta.

Pregunta 2.7

(FL-2.2.1, K2)

Estás probando el sistema de piloto automático del avión. Quiere probar la exactitud de la comunicación entre el componente de geolocalización y el componente del controlador del motor.

¿Cuál de los siguientes es el MEJOR ejemplo de una base de prueba para diseñar estas pruebas?

- A. Diseño detallado del componente de geolocalización.
- B. Diseño arquitectónico.
- C. Informe de análisis de riesgos.
- D. Normatividad del área de aviónica/derecho aeronáutico.

Elija una respuesta.

Pregunta 2.8

(FL-2.2.2, K2)

¿Cuál de los siguientes es un ejemplo de una prueba no funcional?

- A. Cubrir una determinada combinación de condiciones y observar las acciones ejecutadas para comprobar la correcta implementación de una determinada regla de negocio.
- B. Cubriendo el resultado "VERDADERO" en la decisión "SI ($x > 5$) ENTONCES..." en el código.
- C. Verificar que el sistema valida correctamente la sintaxis de la dirección de correo electrónico ingresada por el usuario en el formulario de registro de usuario.
- D. Verificar que el tiempo de inicio de sesión del sistema sea inferior a 5 ms cuando ya hay 1000 usuarios conectados al mismo tiempo.

Elija una respuesta.

Pregunta 2.9

(FL-2.2.3, K2)

¿Cuál de las siguientes pruebas normalmente no se puede programar con anticipación?

- A. Pruebas de regresión.
- B. Pruebas de aceptación operativa (OAT).
- C. Pruebas de aceptación de usuario (UAT).
- D. Pruebas de confirmación.

Elija una respuesta.

Pregunta 2.10

(FL-2.3.1, K2)

Durante las pruebas del software IoT (Internet de las cosas), se descubrió un defecto, pero no se solucionó debido a lo cercano del lanzamiento del software. Después del lanzamiento, hasta ahora no ha causado ningún fallo al cliente. Un mes después del lanzamiento, el equipo decidió solucionar el defecto.

¿Con cuál de los eventos desencadenantes de mantenibilidad estamos lidiando en esta situación?

- A. Actualización de software.
- B. Migración de software.
- C. Modificación del software.
- D. Agregar nuevas funciones al sistema.

Elija una respuesta.

Capítulo 3 Pruebas estáticas



Palabras clave

Anomalía Una condición que se desvía de las expectativas. Después de ISO 24765 Pruebas dinámicas Pruebas que implican la ejecución del elemento de prueba. Según ISO 29119-1 Revisión formal Un tipo de revisión que sigue un proceso definido con un resultado documentado formalmente. Después de ISO 20246 Revisión informal Un tipo de revisión que no sigue un proceso definido y no tiene un resultado documentado formalmente. Un tipo de revisión formal que utiliza roles de equipo definidos y mediciones para identificar defectos en un producto de trabajo y mejorar el proceso de revisión y el desarrollo de software. proceso. Después de ISO 20246 Un tipo de prueba estática en la que un producto o proceso de trabajo es evaluado

Inspección por una o más personas para detectar defectos o proporcionar mejoras El proceso de evaluar un componente o sistema sin ejecutarlo, en función de su forma, estructura, contenido, o documentación. Después de ISO 24765 Pruebas estáticas Pruebas que no implican la ejecución de un elemento de prueba Revisión técnica Una revisión formal realizada por expertos técnicos que examinan la calidad de un producto de trabajo e identifican discrepancias con las especificaciones y estándares Tutorial Un tipo de revisión en la que un autor dirige a los miembros de la revisión a través de un producto de trabajo y los miembros hacen preguntas y comentarios sobre posibles problemas. Después de ISO 20246.

Revisar

Análisis estático

Sinónimos: recorrido estructurado

3.1 Conceptos básicos de las pruebas estáticas

FL-3.1.1 (K1) Reconocer tipos de productos que pueden ser examinados mediante las diferentes técnicas de prueba

estática FL-3.1.2 (K2) Explicar el valor de las pruebas

estáticas FL-3.1.3 (K2) Comparar y contrastar estática y dinámica pruebas


Las pruebas estáticas son un conjunto de métodos y técnicas de prueba en las que el componente o sistema bajo prueba no se ejecuta (no se ejecuta). Las pruebas estáticas también pueden aplicarse a productos de trabajo no ejecutables distintos del software, como diseño, documentación, especificaciones, etc.

Los objetivos de las pruebas estáticas incluyen, en particular, la mejora de la calidad, la detección de defectos y la evaluación de características tales como legibilidad, integridad, corrección, comprobabilidad o consistencia del producto de trabajo bajo revisión. Por tanto, las pruebas estáticas se pueden utilizar tanto para la verificación como para la validación de un producto de trabajo.

En el desarrollo ágil de software, los evaluadores, los representantes comerciales (clientes, usuarios) y los desarrolladores trabajan juntos durante el desarrollo de requisitos (por ejemplo, escribiendo historias de usuarios, creando ejemplos o participando en sesiones de refinamiento del trabajo pendiente). El objetivo de este trabajo colaborativo es garantizar que los requisitos del usuario y los productos de trabajo relacionados cumplan ciertos criterios, como la definición de listo (ver Sección 5.1.3). Se pueden utilizar técnicas de revisión para garantizar que los requisitos sean completos, comprensibles y comprobables y, en el caso de historias de usuarios, contengan criterios de aceptación comprobables. Al hacer las preguntas correctas, los evaluadores examinan, cuestionan y ayudan a mejorar los requisitos propuestos.

Las pruebas estáticas se pueden dividir en dos grupos principales de técnicas:

- Análisis estático •
- Revisiones

Análisis estático  Implica evaluar el producto de trabajo bajo prueba (generalmente código, requisitos o documentos de diseño) utilizando herramientas. El programa de estudios de Foundation Level no describe los métodos de análisis estático en detalle. Encontrarás más detalles en el programa de estudios de Nivel Avanzado “Analista de Pruebas Técnicas” [29]. Ejemplos de técnicas de análisis estático incluyen:

- Medición de código (p. ej., medición de su tamaño o complejidad ciclomática) • Análisis de flujo de control • Análisis de flujo de datos •

Comprobación de la compatibilidad de tipos de variables, verificación de la correcta aplicación de estándares de escritura de código (p. ej., denominación de variables), etc.

El análisis estático puede identificar problemas antes que las pruebas dinámicas, lo que requiere menos esfuerzo porque no se requieren casos de prueba y el análisis generalmente se realiza con herramientas. El análisis estático a menudo se incluye en los marcos de integración continua como un paso del proceso de implementación automatizada (consulte la Sección 2.1.4). Aunque se utiliza principalmente para detectar defectos de código específicos, el análisis estático también se utiliza ampliamente para evaluar la capacidad de mantenimiento, el rendimiento y la vulnerabilidad del código ante ataques de seguridad.

Las revisiones son técnicas de prueba estáticas mucho más utilizadas. Por lo tanto, los discutiremos con más detalle en la Sección 3.2.

3.1.1 Productos de trabajo examinables mediante pruebas estáticas

Las técnicas de prueba estáticas se pueden aplicar a prácticamente cualquier producto de trabajo. El programa de estudios enumera muchos ejemplos de productos de trabajo sujetos a pruebas estáticas. A continuación se proporciona una lista algo ampliada:

- Todo tipo de especificaciones (negocios, requisitos funcionales, requisitos no funcionales, etc.) • Epics, historias de usuarios, criterios de aceptación y otros tipos de documentación utilizados en proyectos ágiles

- Diseño de arquitectura •

Código fuente: quizás excluyendo el código fuente escrito por terceros a los que no tenemos acceso (por ejemplo, en el caso de proyectos comerciales) • Todo tipo de software de prueba, incluidos planes de prueba, procedimientos de prueba, casos de prueba, automatizado guiones de prueba, datos de prueba, documentos de análisis de riesgos y cartas de prueba

- Manuales de usuario, incluida ayuda en línea integrada, manuales del operador del sistema, instrucciones de instalación y notas de la versión.
- Sitios web (en términos de su contenido, estructura, usabilidad, etc.) • Documentos del proyecto (por ejemplo, contratos, planes de proyecto, cronogramas, presupuestos)

Si bien la revisión puede aplicarse básicamente a cualquier producto de trabajo, para que tenga sentido, los participantes en la revisión deben poder leer y analizar el producto con comprensión. Además, en el caso del análisis estático, los productos de trabajo revisados mediante esta técnica deben tener una estructura formal contra la cual se realizan las pruebas. Ejemplos de tales estructuras formales son:

- Código fuente (ya que cada programa está escrito en un lenguaje de programación definido basado en una gramática formal)
- Modelos (por ejemplo, diagramas UML que tienen una sintaxis específica y reglas formales para creándolos)
- Documentos de texto (porque el texto se puede comparar, por ejemplo, con las reglas gramaticales del idioma)

El análisis estático se aplica con mayor frecuencia a productos de trabajo formalizados, como modelos o requisitos formales de arquitectura de sistemas (por ejemplo, escritos en lenguajes de especificación como Z o UML). Para documentos escritos en lenguaje natural, el análisis estático puede

implican, por ejemplo, comprobar la legibilidad, la sintaxis, la gramática, la puntuación o la ortografía.

3.1.2 Valor de las pruebas estáticas

Las pruebas estáticas no suelen ser baratas (requieren una comprobación manual del producto), pero, si se realizan correctamente, son efectivas y eficientes. Esto se debe a que permite encontrar defectos y problemas muy temprano en el SDLC que serían difíciles de detectar en fases posteriores. Esto generalmente se refiere a defectos de diseño. Este tipo de defecto suele ser muy insidioso, porque un defecto de diseño no detectado en las fases iniciales se propaga muy fácilmente a las fases posteriores y se crea una implementación defectuosa basada en un diseño defectuoso. El problema generalmente se hace evidente en la etapa de prueba de aceptación o del sistema, donde puede resultar muy costoso corregir dicho defecto de diseño.

Las pruebas estáticas bien realizadas antes de la ejecución de las pruebas dinámicas dan como resultado una eliminación rápida (y razonablemente económica) de problemas que podrían ser la fuente de otros defectos. Como resultado, se detectan menos problemas en las pruebas dinámicas, por lo que el costo total de las pruebas dinámicas es menor que si no se realizaran pruebas estáticas. Este uso de pruebas estáticas está en línea con el principio de pruebas tempranas y desplazamiento a la izquierda (ver Sección 1.3).

Las pruebas estáticas brindan la oportunidad de evaluar la calidad y generar confianza en el producto del trabajo revisado. Las partes interesadas pueden evaluar si los requisitos documentados describen sus necesidades reales. Debido a que las pruebas estáticas se pueden realizar en una etapa temprana del SDLC, se crea una comprensión común del producto y sus requisitos entre las partes interesadas involucradas en las pruebas estáticas. Este entendimiento compartido también mejora la comunicación. Por esta razón, se recomienda que las revisiones involucren a varias partes interesadas que representen las perspectivas más amplias y diversas sobre el producto del trabajo bajo revisión.

Los defectos en el código se pueden detectar y corregir usando pruebas estáticas de manera más eficiente que cuando se usan pruebas dinámicas, porque en las pruebas dinámicas, la ocurrencia de una falla generalmente requiere un análisis tedioso de la causa de la falla y dedicar una gran cantidad de tiempo a identificar el defecto. causándolo. Esta eficiencia de las pruebas estáticas generalmente da como resultado menos defectos restantes en el código y una menor carga de trabajo general.

La eficacia de las técnicas estáticas está demostrada empíricamente. Jones y Bonsignour [31] proporcionan una gran cantidad de datos que muestran que las técnicas estáticas contribuyen significativamente a la calidad general del producto y a reducir los costos de prueba y mantenimiento. En el caso de la inspección (un tipo de revisión que se describe en la siguiente sección), ¡la eficiencia de eliminación de defectos aumenta en un promedio del 85%!

La siguiente es una lista de beneficios de las pruebas estáticas:

- Detección temprana y eficaz de defectos (y posible eliminación), incluso antes de que comiencen las pruebas dinámicas: la capacidad de realizar pruebas incluso antes de que se cree un prototipo de software funcional.

- Identificar defectos difíciles de detectar en pruebas dinámicas posteriores. Esto es especialmente cierto en el caso de defectos arquitectónicos y de diseño.
- Identificar defectos que son imposibles de detectar en pruebas dinámicas, como localizar código inviable (inalcanzable),¹ código no utilizado, uso incorrecto o falta de uso de patrones de diseño en el código, o todo tipo de defectos en productos de trabajo no ejecutables. como por ejemplo documentación.
- Prevenir la aparición de defectos en el diseño y el código detectando ambigüedades, contradicciones, omisiones, descuidos, elementos redundantes o inconsistencias en documentos como la especificación de requisitos o el diseño arquitectónico.
- Incrementar la eficiencia del trabajo de programación. Se puede mejorar el diseño y la mantenibilidad del código, por ejemplo, imponiendo un estándar uniforme para escribirlos. Esto hace que el código sea más fácil de modificar no sólo por el autor sino también por otros desarrolladores, y disminuye la posibilidad de introducir un defecto al modificar el código.

- Reducir el costo y el tiempo del desarrollo de software, incluidas las pruebas, especialmente las pruebas dinámicas.
- Reducir el coste de la calidad en todo el ciclo de desarrollo de software reduciendo costes en la fase de mantenimiento, así como reduciendo fallos en la fase de explotación del software.

- Mejorar la comunicación entre los miembros del equipo mediante la participación en revisiones, incluidas las reuniones de revisión.

Al evaluar los costos incurridos por las pruebas, se introduce el concepto de "costo de calidad". Es el costo total incurrido por las actividades de calidad, el cual consiste en los costos de:

- Actividades preventivas (por ejemplo, costos de capacitación)
- Detección (por ejemplo, costo de pruebas)
- Fallas internas (por ejemplo, costo de reparar defectos encontrados en producción)
- Fallas externas (por ejemplo, costo de arreglar defectos en campo encontrados por los usuarios)

Aunque las revisiones pueden ser costosas de implementar, los costos generales de calidad suelen ser mucho más bajos que si no se realizan, porque se debe dedicar menos tiempo y esfuerzo a la eliminación de defectos más adelante en el proyecto. Los participantes en el proceso de revisión también se benefician de una mejor comprensión compartida del producto que se revisa.

¹ La identificación de un código inviable sólo es posible en algunos casos, porque en general el problema de la accesibilidad a un determinado lugar en el código es el llamado problema indecidible. Esto significa que no existe ningún algoritmo que, para cualquier programa y cualquier lugar en su código, responda a la pregunta de si hay entradas al programa para las cuales el control llega a ese lugar en el código.

Ejemplo de análisis económico de inspecciones Considere una simulación económica simplificada de si vale la pena implementar pruebas estáticas en una organización. En el escenario A, el equipo realizará únicamente pruebas dinámicas. En el Escenario B, las pruebas dinámicas estarán precedidas por pruebas estáticas. El siguiente modelo es muy simple y solo pretende ilustrar cómo las pruebas estáticas pueden reducir el costo total del desarrollo y mantenimiento del software.

Supuestos del modelo:

- El equipo de probadores está formado por seis personas. • El costo mensual del salario de un probador es de \$7000. • Las pruebas estáticas y las pruebas dinámicas tienen una duración de 4 meses cada una y participa todo el equipo de pruebas. Las pruebas estáticas (si las hay) se realizan antes de las pruebas dinámicas.
- El número total de defectos de software es 130. • Las pruebas estáticas encuentran el 50% de todos los defectos existentes (ver, por ejemplo, [31] para datos de la industria).
- Las pruebas dinámicas encuentran el 80% de todos los defectos existentes. El 20% restante son defectos de campo, descubiertos por el cliente después del lanzamiento del software (estos valores se derivan de datos históricos). • Arreglar un defecto encontrado en pruebas estáticas cuesta \$500 en promedio (datos de la industria). • Arreglar un defecto encontrado en las pruebas dinámicas cuesta \$1800 en promedio (datos históricos). • Arreglar un defecto de campo (encontrado por el usuario después de que se lanza el software) cuesta \$12,600 (datos históricos).

Escenario A: sin pruebas estáticas

Número de defectos detectados en pruebas dinámicas: $80\% \times 130 = 104$

Número de defectos de campo: $20\% \times 130 = 26$

Costo de las pruebas dinámicas: $6 \times \$7000 \times 4 = \$168,000$

Costo de eliminación de defectos antes del lanzamiento: $104 \times \$1800 = \$187,200$

Costo de eliminar defectos después del lanzamiento: $26 \times \$12,600 = \$327,600$

Costo total de calidad: $\$168,000 + \$187,200 + \$327,600 = \$682,800$

Escenario B: con pruebas estáticas

Número de defectos detectados en pruebas estáticas: $50\% \times 130 = 65$

Número de defectos detectados en pruebas dinámicas: $80\% \times (130 - 65) = 52$

Número de defectos de campo: $130 - (65 + 52) = 13$

Costo de las pruebas estáticas: $6 \times \$7000 \times 4 = \$168,000$

Costo de las pruebas dinámicas: $6 \times \$7000 \times 4 = \$168,000$

(continuado)

Costo de eliminar defectos encontrados en pruebas estáticas: $65 * \$500 = \$32,500$
Costo de eliminación de defectos antes del lanzamiento: $52 * \$1800 = \$93,600$
Costo de eliminar defectos después del lanzamiento: $13 * \$12,600 = \$163,800$

Costo total de calidad: $\$168.000 + \$168.000 + \$32.500 + \$93.600 + \$163.800$
 $= \$625,900$

Esquemáticamente, la comparación de estos dos escenarios se muestra en la Fig. 3.1.

Los costos de calidad previos al lanzamiento son más altos en el escenario B (\$462,100 versus \$355,200), debido a los altos costos de inspección. Sin embargo, la ganancia se produce después del lanzamiento, debido a la mitad del número de defectos de campo en el Escenario B, gracias al uso de revisiones. Por lo tanto, el costo total resulta ser menor en el escenario B. La diferencia de costo exacta entre los escenarios A y B es $\$682\,800 - \$625\,900 = \$56\,900$.

Por supuesto, en el análisis debemos tener en cuenta los errores de estimación, así como el hecho de que el equipo de pruebas necesitó cuatro meses adicionales para realizar pruebas estáticas. Sin embargo, el análisis muestra que el uso de pruebas estáticas no sólo puede mejorar significativamente la calidad final del producto (26 defectos de campo frente a 13) sino también reducir significativamente los costos de desarrollo. En la simulación anterior, ahorramos alrededor de \$57 000, por lo que los costos en el Escenario B son aproximadamente un 8,3 % menos que los del Escenario A.

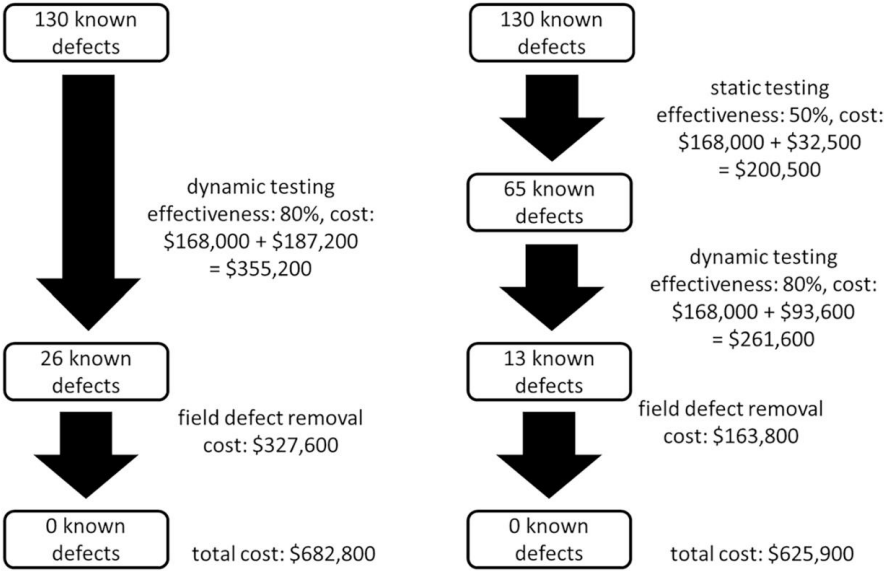


Fig. 3.1 Comparación de procesos de eliminación de defectos sin y con pruebas estáticas

3.1.3 Diferencias entre pruebas estáticas y pruebas dinámicas

Tanto las pruebas estáticas como las dinámicas tienen el mismo objetivo: evaluar la calidad del producto e identificar defectos lo antes posible. Estas actividades son complementarias, ya que permiten detectar diferentes tipos de defectos.

La Figura 3.2 representa simbólicamente la diferencia fundamental entre las pruebas estáticas (parte superior de la figura) y las pruebas dinámicas (parte inferior de la figura). Con las pruebas estáticas, podemos encontrar un defecto directamente en el producto de trabajo. No encontramos fallos, porque con las técnicas estáticas, por definición, no estamos ejecutando el software y un fallo sólo puede ocurrir como consecuencia del sistema que está ejecutando.

En el caso de las pruebas dinámicas, la mayoría de las veces el primer signo de un mal funcionamiento del software es, a su vez, una falla (el resultado de no pasar una prueba). Cuando se observa una falla, se inicia un proceso de depuración, durante el cual se analiza el código fuente para encontrar el defecto responsable de causar esta falla.

A veces, un defecto en un producto de trabajo puede permanecer oculto durante mucho tiempo porque no causa una falla. Además, es posible que la ruta en la que se encuentra rara vez se pruebe o sea de difícil acceso, lo que dificulta el diseño y la ejecución de pruebas dinámicas que lo detecten. Las pruebas estáticas pueden encontrar un defecto con mucho menos esfuerzo. Por otro lado, hay defectos que son mucho más fáciles de detectar con pruebas dinámicas que con pruebas estáticas, como las pérdidas de memoria.

Otra diferencia es que las pruebas estáticas se pueden utilizar para aumentar la coherencia y la calidad interna de los productos de trabajo, mientras que las pruebas dinámicas se centran principalmente en el comportamiento visible externamente.

Además, sólo se pueden aplicar pruebas estáticas a productos de trabajo no ejecutables, como código fuente o documentación. Las pruebas dinámicas, por otro lado, sólo se pueden realizar contra un producto de trabajo en ejecución, es decir, software en ejecución. A

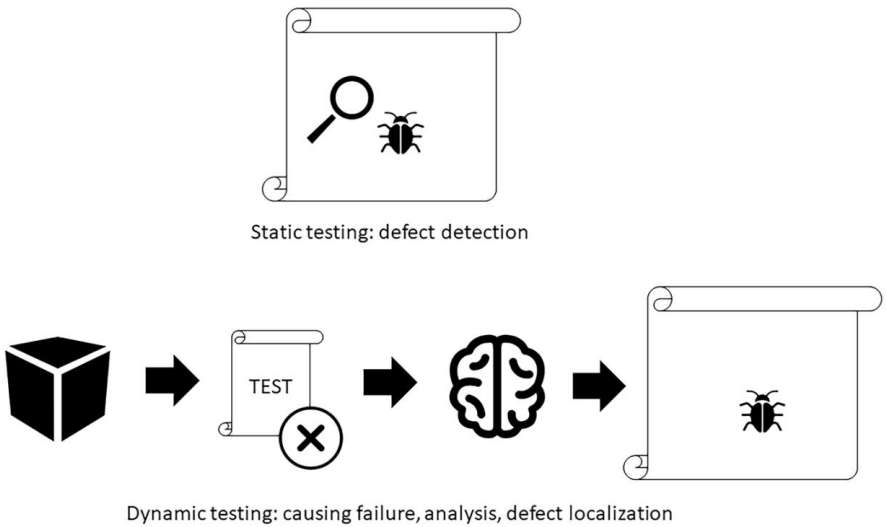


Fig. 3.2 Diferencia entre pruebas estáticas y pruebas dinámicas

La consecuencia de este hecho es que las pruebas dinámicas se pueden utilizar para medir algunas características de calidad que son imposibles de medir en las pruebas estáticas, porque dependen del programa en ejecución. Un ejemplo serían las pruebas de rendimiento que miden el tiempo de respuesta del sistema para una solicitud de usuario específica.

Mencionamos anteriormente que las pruebas estáticas generalmente detectan defectos diferentes a las pruebas dinámicas. Ejemplos de defectos típicos que son más fáciles y económicos de detectar y solucionar con pruebas estáticas que con pruebas dinámicas son:

- Defectos en los requisitos (como inconsistencias, ambigüedades, omisiones, contradicciones, inexactitudes, omisiones, repeticiones, elementos redundantes) • Defectos de diseño (por ejemplo, algoritmos o estructuras de bases de datos ineficientes, alto acoplamiento, modularización deficiente del código) • Tipos específicos de código defectos (p. ej., variables con valores indefinidos, variables declaradas pero nunca utilizadas, código inaccesible, código duplicado, algoritmos implementados de manera ineficiente con demasiado tiempo o complejidad de memoria) • Desviaciones de los estándares (p. ej., falta de cumplimiento con el desarrollo del código estándares)
- Especificaciones de interfaz incorrectas (p. ej., uso de diferentes unidades de medida en los sistemas llamante y llamado, tipo incorrecto u orden incorrecto de parámetros pasados a una llamada de función API) • Vulnerabilidades de seguridad (p. ej., susceptibilidad a ataques de desbordamiento de búfer, SQL (transferencia cruzada) inyección, XSS secuencias de comandos del sitio), ataque DDoS • Brechas o imprecisiones en la trazabilidad o cobertura (por ejemplo, no hay pruebas que coincidan con los criterios de aceptación para una historia de usuario determinada)

² El acoplamiento se refiere al grado en que un componente de software depende de otro y expresa el nivel de interdependencia entre módulos. Un alto acoplamiento significa que un cambio en un módulo puede requerir un cambio en otros módulos, lo que puede generar mayores costos de mantenimiento y desarrollo y dificultar la modificación y expansión del sistema. Un bajo acoplamiento significa que los componentes son más independientes entre sí, lo que los hace más fáciles de gestionar.

³ La cohesión se refiere al grado en que los componentes de un módulo están interrelacionados y persiguen un objetivo común. Alta cohesión significa que los componentes dentro de un módulo se centran en lograr el mismo objetivo único (tarea o funcionalidad), lo que los hace fáciles de entender, mantener y desarrollar. Una baja cohesión significa que los componentes dentro de un módulo persiguen objetivos diferentes, lo que los hace difíciles de entender, introduce dependencias innecesarias entre ellos y puede conducir a mayores costos de desarrollo y mantenimiento del sistema y a una legibilidad y escalabilidad deficientes.

⁴ XSS es una forma de ciberataque a sitios web para piratear a sus usuarios. El pirata informático coloca un script malicioso en un sitio web aparentemente amigable y seguro, lo que hace que cualquier persona que lo utilice sea vulnerable a ataques (según home.co.uk).

⁵ Un ataque DDoS implica lanzar un ataque simultáneamente desde múltiples ubicaciones al mismo tiempo (desde múltiples computadoras). Un ataque de este tipo se lleva a cabo principalmente desde ordenadores sobre los que se ha tomado el control mediante software especial (p. ej., bots y troyanos) (según home.pl).

Tabla 3.1 Análisis estático
resultados para el código
mantenibilidad

Componente	LOC	COMENTARIO	CC
principal	129	0%	7
dividir	206	1%	12
en_inferior	62	1%	6
a_superior	63	3%	6
fusionar	70	0%	15
configuración	42	15%	8
stdio	243	2%	21

Ejemplo El equipo de pruebas quiere evaluar la facilidad y el costo de mantener el software después de su entrega al cliente. Para ello, pretende utilizar el análisis estático. técnicas. Se han identificado tres métricas que se utilizarán para medir la código fuente:

- LOC: el número de líneas de código ejecutables en un componente
- COMENTARIO: el porcentaje de líneas de código anotadas en un componente.
- CC: complejidad ciclomática⁶ de un componente

Los resultados se muestran en la Tabla 3.1.

El análisis estático llevó a las siguientes conclusiones:

- Los componentes son bastante pequeños. El mayor de ellos, stdio, tiene 243 líneas de código. Por lo tanto, el tamaño de los componentes no debería ser un problema importante. en cuanto a mantenibilidad.
- El código no está suficientemente comentado. Excepto por el módulo de configuración, en el que El 15% del código está comentado, en todos los demás módulos, el porcentaje de líneas comentadas está entre 0 y 3%. Los módulos principal y de combinación no contener comentarios en absoluto. Este hecho definitivamente debería informarse a los desarrolladores. atención.
- Tres componentes, split, merge y stdio, tienen valores altos (superiores a 10). Complejidad ciclomática. Deberían ser analizados. Esto es especialmente cierto para fusión en la que la densidad de declaraciones de decisión es muy alta: una decisión para aproximadamente cinco líneas de código, lo que puede sugerir una legibilidad muy pobre de la fuente código. No refactorizar estos componentes podría dificultar su mantenimiento. ellos en el futuro.

⁶ La complejidad ciclomática de un componente se define como el número de declaraciones de decisión en el código de ese módulo más uno. Esta es una medida muy simple de la complejidad de la estructura del código.

3.2 Proceso de retroalimentación y revisión

FL-3.2.1 (K1) Identificar los beneficios de la retroalimentación temprana y frecuente de las partes

interesadas FL-3.2.2 (K2) Resumir las actividades del proceso de revisión FL-3.2.3

(K1) Recordar qué responsabilidades se asignan a los roles principales cuando realizar revisiones FL-3.2.4

(K2) Comparar y contrastar los


diferentes tipos de revisión FL-3.2.5 (K1) Recordar los factores que contribuyen a una revisión exitosa

En los siguientes párrafos, discutiremos en detalle:

- Ventajas de la retroalimentación temprana y frecuente de los clientes
- Proceso de revisión del producto de trabajo
- Roles y responsabilidades en las revisiones formales
- Tipos de revisiones
- Factores de éxito en las revisiones

En la última subsección de este capítulo (Sección [3.2.6](#)), analizamos técnicas para revisar los productos del trabajo. Esta sección estaba presente en el programa de estudios Foundation Level v3.1 pero se eliminó en el nuevo (v4.0). No obstante, decidimos dejarlo en el libro como una sección adicional, porque analiza la importante cuestión de cómo abordar de manera práctica la actividad misma de revisar un producto de trabajo. Esta etapa es un paso central en el proceso de revisión (ver Sección [3.2.2](#)), por lo que es útil conocer las técnicas básicas de revisión. El material presentado en la Sección. [3.2.6](#) no es examinable.

3.2.1 Beneficios de la retroalimentación temprana y frecuente de las partes interesadas

Las revisiones  son una forma de proporcionar retroalimentación temprana al equipo porque se pueden realizar en una etapa temprana del ciclo de desarrollo. Sin embargo, la retroalimentación no debe limitarse únicamente a revisiones, sino que debe ser una práctica común durante todo el ciclo de desarrollo del proyecto. Los comentarios pueden provenir de evaluadores o desarrolladores, pero igualmente importante para el equipo será el del propio cliente.

La retroalimentación temprana y frecuente permite una notificación rápida de posibles problemas de calidad y permite al equipo responder rápidamente al problema. Si hay poca participación de las partes interesadas durante el desarrollo de software, es posible que el producto que se está desarrollando no cumpla con su visión original o actual. No cumplir con lo que las partes interesadas esperan puede resultar en costosas reelaboraciones, incumplimiento de plazos y juegos de culpas e incluso llevar al fracaso total del proyecto.

La retroalimentación frecuente de las partes interesadas durante el desarrollo de software puede evitar malentendidos sobre los requisitos y garantizar que los cambios en los requisitos se realicen correctamente.

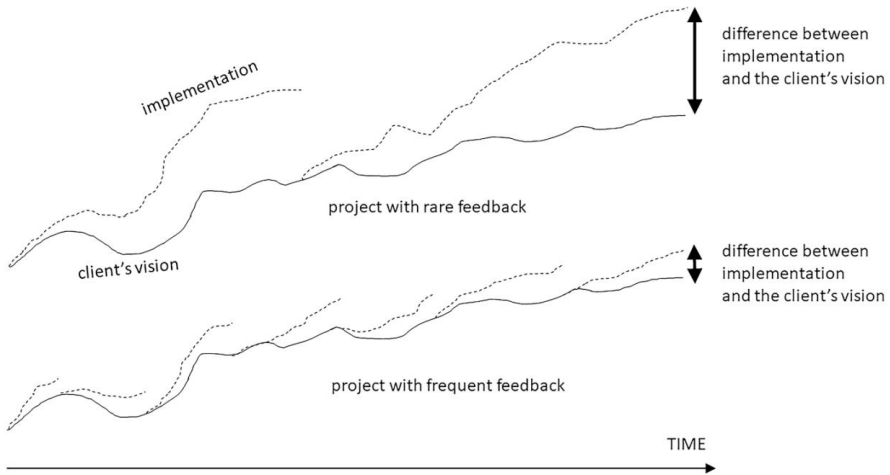


Fig. 3.3 Ilustración del beneficio de la retroalimentación frecuente

entendido e implementado antes. Esto ayuda al equipo a comprender mejor lo que están construyendo. Les permite centrarse en aquellas funciones que ofrecen el mayor valor a las partes interesadas y que tienen el impacto más positivo en los riesgos acordados.

La figura 3.3 muestra simbólicamente el beneficio de la retroalimentación frecuente. La línea continua representa la visión del cliente y la línea discontinua representa cómo se ve realmente el producto en su implementación. Estas dos visiones divergen cada vez más con el tiempo. En los momentos de retroalimentación, la implementación se ajusta a la visión del cliente. El gráfico superior muestra una situación en la que el contacto con el cliente es esporádico y se produce sólo dos veces. Se puede ver que el resultado es una implementación que difiere significativamente de lo que era la visión del cliente. El gráfico inferior representa una situación en la que el cliente frecuentemente proporciona comentarios al equipo. Esto da como resultado menos diferencias entre la visión del cliente y la implementación real, porque transcurre mucho menos tiempo entre las reuniones con el cliente que en la situación del gráfico superior. Como resultado, el producto final está mucho más alineado con la visión del cliente. Además, cabe señalar que el coste de “ajustar” el producto a la visión del cliente será mucho mayor en el caso de feedback poco frecuente, debido a que habrá que modificar y ajustar el producto en mucha mayor medida que en la situación de retroalimentación frecuente.

3.2.2 Actividades del proceso de revisión

Las revisiones varían en tipo, nivel de formalización u objetivos, pero todos los tipos de revisiones tienden a tener fases o grupos de actividades similares. La norma ISO 20246 Ingeniería de software y sistemas: revisiones de productos de trabajo [6] define un proceso de revisión genérico que proporciona un marco estructurado pero flexible dentro del cual adaptar un