Testing en Desarrollo de Software

Fundamentos de Testing

Definición y Propósito del Testing de Software

- Proceso sistemático de evaluar y verificar que un producto o aplicación de software hace lo que se supone que debe hacer. Consiste en un conjunto de actividades para detectar defectos, verificar requisitos y validar necesidades de usuarios e interesados.
- Propósito fundamental: garantizar la calidad del software y reducir el riesgo de fallos en su funcionamiento.

Importancia del Testing en el Ciclo de Desarrollo

- Garantiza la calidad del software: Cumple requerimientos y especificaciones, es confiable y seguro.
- Reduce riesgos de fallos: Evita pérdidas financieras y de reputación.
- Mejora la fiabilidad: Funciona correctamente y consistentemente.
- Reduce costos de desarrollo: Detectar errores temprano es menos costoso.
- Mejora la satisfacción del usuario: Entrega un producto de mayor calidad.

Verificación vs. Validación

- Verificación: Evalúa documentos, diseño, código; verifica si se ha creado según requisitos. No ejecuta código (revisiones, inspecciones). Encuentra errores al inicio.
- Validación: Prueba y valida el producto real; determina si satisface necesidades del cliente. Siempre ejecuta código (pruebas caja negra, blanca, no funcionales). Puede encontrar errores que verificación no detecta.
- Diferencias clave entre verificación y validación: qué se evalúa, si implica ejecución, métodos usados, objetivo, cuándo se encuentran errores, quién lo realiza, orden en el ciclo.

Testing y Reducción de Deuda Técnica

- Deuda técnica: Trabajo pospuesto intencionalmente. Lleva a baja calidad, rendimiento lento, defectos, moral negativa.
- Testing automatizado es efectivo para reducirla. Identifica y corrige problemas rápidamente. Permite depuración automatizada y escaneo continuo.

Ruta de Aprendizaje

Conceptos Fundamentales (Nivel Principiante)

- Comprender principios básicos (definición, objetivos, 7 principios, verificación vs validación).
- Aprender tipos básicos (unitarias, funcionales vs no funcionales).
- Familiarizarse con herramientas básicas (frameworks, ejecutores en IDEs).

Habilidades Intermedias

- Profundizar en metodologías (TDD, BDD).
- Expandir conocimientos a otros tipos (integración, sistema, aceptación).
- Aprender automatización de pruebas (frameworks, testing en CI/CD).

Técnicas Avanzadas

- Dominar técnicas específicas (Mocks, stubs, spies; pruebas parametrizadas, property-based).
- Testing especializado (rendimiento, seguridad, mutation testing).
- Análisis y mejora (cobertura, reducción deuda técnica, optimización suite).

Buenas Prácticas

Estructura y Organización de Pruebas

- Esencial para mantenerlas organizadas y fáciles de mantener.
- Prácticas: organización por funcionalidad, separación por tipos, código limpio, independencia de pruebas.

Naming Conventions para Pruebas

- Ayuda a identificar propósito rápidamente.
- Prácticas: Descriptivo, Formato estándar (ej: test[Funcionalidad]_[Condición]_[Resultado]), Legibilidad (nombres largos).

Cómo Escribir Pruebas Mantenibles

- Patrón AAA: Arrange, Act, Assert.
- Evitar lógica compleja, minimizar duplicación (fixtures, helpers), pruebas aisladas (una cosa por prueba), documentar propósito.

Mocks, Stubs y Spies

- Herramientas para aislar código bajo prueba.
- Mocks: simulan comportamiento para verificación de interacciones.
- Stubs: proporcionan respuestas predefinidas (datos de entrada controlados).
- Spies: registran llamadas sin modificar comportamiento (observar).

Test Coverage

- Mide qué parte del código es ejecutada por las pruebas.
- Tipos: cobertura de líneas, ramas, funciones.
- Herramientas (ej: SonarQube) ayudan a medir.
- Alta cobertura no garantiza ausencia de defectos; pruebas muestran presencia, no ausencia.

Recursos Adicionales

- Para continuar aprendiendo: Libros (ej: TDD by Example, Clean Code), Cursos online, Documentación (frameworks, herramientas CI), Comunidades, Herramientas (cobertura, análisis estático SonarQube, automatización).
- Testing es una habilidad que se desarrolla con práctica continua y aplicación consciente de principios/buenas prácticas.

Metodologías de Testing

Test-Driven Development (TDD)

- Pruebas se escriben antes del código de implementación.
- Ciclo: Escribir prueba que falle, desarrollar código mínimo para pasarla, refactorizar.
- Ayuda a: clarificar requisitos, asegurar cobertura de pruebas, reducir bugs, mejorar diseño.

Behavior-Driven Development (BDD)

- Extensión de TDD, enfatiza colaboración (dev, QA, stakeholders). Usa lenguaje común para describir comportamiento esperado.
- Características: lenguaje natural (Gherkin 'Given-When-Then'), enfoque en comportamiento, facilita comunicación, produce documentación viva.

Manual vs. Automated Testing

- Testing Manual: Tester humano ejecuta paso a paso. Útil para exploratorias, intuición humana.
- Testing Automatizado: Herramientas y scripts ejecutan automáticamente. Ideal para repetitivas, regresiones, velocidad, consistencia.
- Ventajas automatizado: ahorra tiempo, resultados consistentes, más pruebas en menos tiempo, reduce errores humanos, facilita CI/CD.

CI/CD y Testing

- Testing automatizado es fundamental en pipelines de CI/CD para entrega frecuente y confiable.
- En CI: verifica que cambios no rompan funcionalidad.
- En CD: garantiza calidad para despliegue automático.
- Pipeline típico incluye: pruebas unitarias, integración, funcionales automatizadas, análisis de calidad/cobertura, despliegue condicional.

Errores Comunes

- Errores que cometen desarrolladores junior: No seguir principios, pruebas frágiles, pruebas complicadas, ignorar verificación temprana, falta de independencia, pruebas incompletas (casos límite/error), confundir verificación/validación, ignorar deuda técnica.
- Evitar errores: Seguir buenas prácticas, entender principios fundamentales, enfoque disciplinado.

Tipos de Testing

Unit Testing (Pruebas Unitarias)

- Nivel más básico, se centra en componentes individuales o funciones aisladas.
- Características: enfocadas en lógica interna, rápidas, identifican problemas temprano, facilitan refactoring.

Integration Testing (Pruebas de Integración)

- Verifican que diferentes módulos o servicios funcionan bien juntos (ej: base de datos, microservicios, APIs).
- Esenciales para verificar comunicación entre componentes, sistemas integrados y flujos de datos.

Functional Testing (Pruebas Funcionales)

- Verifican que el software hace lo que se supone desde la perspectiva del usuario (contra requisitos funcionales).
- Verifican características según requisitos, se centran en comportamiento externo, pueden ser manuales o automatizadas, no se preocupan por código interno.

System Testing (Pruebas de Sistema)

- Evalúa el sistema completo e integrado contra requisitos especificados. Se realiza en entorno simulado, abarca pruebas funcionales y no funcionales.

Acceptance Testing (Pruebas de Aceptación)

- Verifica si el sistema cumple requisitos del negocio y es aceptable para entrega. Realizadas por usuarios finales o clientes.
- Cruciales: confirman necesidades del usuario, validación final, descubren problemas desde perspectiva del usuario real.

Performance Testing (Pruebas de Rendimiento)

- Evalúa velocidad, escalabilidad y estabilidad bajo carga (pruebas de carga, estrés, estabilidad).

Security Testing (Pruebas de Seguridad)

- Identifica vulnerabilidades y asegura protección de datos/recursos. Esencial para datos sensibles o acceso protegido.

Ejercicios Prácticos

- Escribir Mi Primera Prueba Unitaria (ejemplo con Jest).
- Implementar TDD en un Pequeño Proyecto (pasos con ejemplo calculadora).
- Configurar un Pipeline de CI con Tests Automatizados (ejemplo con GitHub Actions).