Testing de Software

1. Fundamentos del Testing de Software

¿Qué es el testing? Es el proceso de ejecutar un programa con el objetivo de encontrar errores. Implica la verificación (comprobación de que se cumplen los requisitos especificados) y la validación (asegurar que se construyó el sistema correcto).

Objetivos del testing:

- Detectar fallas antes del despliegue.
- Asegurar la calidad funcional y no funcional del software.
- Validar el cumplimiento de los requisitos.
- Aumentar la confiabilidad y mantenibilidad del producto.

2. Clasificación de Pruebas

2.1. Según su objetivo:

- **Pruebas funcionales:** Validan las funcionalidades de acuerdo a los requerimientos.
- **Pruebas no funcionales:** Evalúan aspectos como rendimiento, seguridad, usabilidad, escalabilidad, etc.

2.2. Según el nivel:

- **Pruebas unitarias:** Evalúan componentes individuales (funciones, clases).
- **Pruebas de integración:** Verifican la interacción entre módulos o servicios.
- **Pruebas de sistema:** Prueban el sistema completo en un entorno de prueba.
- **Pruebas de aceptación:** Determinan si el software cumple con los criterios del cliente.

2.3. Según la técnica:

- **Pruebas estáticas:** Revisión de código, análisis estático, sin ejecución.
- **Pruebas dinámicas:** Requieren la ejecución del software.

2.4. Caja negra vs. Caja blanca

- **Caja negra:** Se centra en los resultados esperados sin conocer la implementación interna. Útil para pruebas funcionales y de aceptación.
- **Caja blanca:** Evalúa la lógica interna del código, incluyendo flujos de control y condiciones. Es común en pruebas unitarias y de integración técnica.

Comparación:

- **Caja negra:** Simula el punto de vista del usuario. Rápida de ejecutar y fácil de automatizar.
- **Caja blanca:** Detecta errores internos más complejos, asegura mayor cobertura de código.

3. Tipos de Pruebas Especializadas

- **Pruebas de regresión:** Validan que cambios en el código no introduzcan nuevos errores.
- **Pruebas de estrés:** Evaluación bajo condiciones extremas de carga.
- **Pruebas de carga:** Verifican el comportamiento del sistema bajo carga esperada.
- **Pruebas exploratorias:** El tester investiga libremente el sistema en busca de defectos no anticipados.

4. Enfoques Basados en Desarrollo

4.1. TDD (Test-Driven Development)

Definición: Técnica en la cual se escriben primero los tests antes de codificar la funcionalidad.

Ciclo:

1. Escribir un test que falle.
2. Escribir el código mínimo para que pase el test.
3. Refactorizar.

Ventajas:

- Código mejor estructurado.
- Menor tasa de errores.
- Mayor cobertura de pruebas.

Aplicación práctica en PHP (Login):

- Test de validación de credenciales.
- Test de redirección en caso de éxito/falla.
- Simulación con PHPUnit de usuarios válidos/erróneos.

4.2. BDD (Behavior-Driven Development)

Definición: Extiende TDD incorporando lenguaje natural para describir el comportamiento deseado desde el punto de vista del usuario.
Sintaxis Gherkin:

- Given (Dado)
- When (Cuando)
- Then (Entonces)

Ejemplo:

Given que el usuario no está autenticado
When intenta acceder al panel de control
Then debería ser redirigido al login

Herramientas: Behat (PHP), Cucumber, SpecFlow.

4.3. ATDD (Acceptance Test-Driven Development)

Definición: Desarrollo guiado por las pruebas de aceptación escritas antes de la implementación. Implica colaboración entre cliente, desarrollador y tester.

5. Herramientas Focalizadas en PHP

5.1. Unitarias e Integración

- **PHPUnit:** Framework de pruebas unitarias más usado en PHP.
- **Codeception:** Framework de testing completo (unitario, integración, funcional, aceptación).
- **Pest:** Framework con sintaxis elegante y centrado en la legibilidad.

5.2. Funcionales y de aceptación

- **Behat:** Ideal para BDD y pruebas de aceptación en lenguaje natural.
- **Laravel Dusk:** Automatización de pruebas funcionales del frontend (navegación con navegador).

5.3. Cobertura y calidad

- **Xdebug + PHPUnit:** Para medir cobertura.
- **PHPStan, Psalm:** Análisis estático del código.

5.4. Rendimiento, estrés y carga

- **Artillery (Node.js):** Puede ser integrado con APIs PHP para pruebas de carga.
- **JMeter:** Soporta testing de rendimiento contra endpoints web.

6. Estrategia de Testing en un Proyecto PHP

Nivel de prueba	Herramienta	Frecuencia
Unitarias	PHPUnit / Pest	En cada push
Integración	Codeception	Antes de deploy
Aceptación/BDD	Behat	Por historia de usuario
Funcionales	Laravel Dusk	Antes de staging
Carga y estrés	Artillery / JMeter	Semanal o por release

7. Métricas y Reportes de Calidad

- **Cobertura de código:** % de líneas ejecutadas por tests.
- **Ratio de defectos:** Número de bugs encontrados vs. esperados.
- **Tiempo de respuesta bajo carga:** ms/req bajo estrés.
- **MTTR:** Tiempo medio de resolución de errores.

8. Caso Práctico Resumido: Sistema de Autenticación

Funcionalidades a testear:

- Registro.
- Inicio de sesión.
- Recuperación de contraseña.

Plan de pruebas sugerido:

- Pruebas unitarias para validaciones de input.
- Pruebas funcionales para el flujo completo.
- Pruebas de carga simulando 1000 usuarios concurrentes.
- Pruebas BDD con Behat para escenarios comunes.

9. Beneficios del Uso de Herramientas de Ingeniería y Control de Versiones

- **Gestión eficiente de pruebas automatizadas.**
- **Historial claro de fallas, correcciones y regresiones.**
- **Integración continua (CI/CD) con GitHub Actions y otras plataformas.**
- **Colaboración entre desarrolladores y testers mediante pull requests y revisiones.**

Conclusión: El testing efectivo requiere planificación estratégica, aplicación adecuada de metodologías (TDD, BDD, ATDD), y el aprovechamiento de herramientas modernas que potencien la calidad del producto, la colaboración y la trazabilidad en el desarrollo.