Testing Efectivo en Java

Círculo Siete Capacitación

Clase 2 de 12
7 Agosto 2025

circulo siete
overcoming tech for business

3. Herramientas y Dependencias Comunes

- El ecosistema de testing en Java es amplio y maduro.
- En la siguiente lámina algunas de las herramientas más utilizadas.

Categoría	Herramienta	Descripción
Unit Testing	**JUnit 5**	Framework principal para pruebas unitarias en Java moderno
Mocking	**Mockito**	Biblioteca para simular dependencias y controlar el comportamiento de objetos
TDD/BDD	**Spock**	Framework basado en Groovy con enfoque declarativo y expresivo
Testing en Spring	**Spring Boot Test**	Anotaciones y helpers para probar aplicaciones Spring
Testing realista	**Testcontainers**	Usa contenedores Docker para pruebas de integración
Arquitectura	**ArchUnit**	Valida reglas arquitectónicas del proyecto mediante código
Cobertura	**JaCoCo**	Genera métricas de cobertura de código
Automatización	**Maven/Gradle + CI (GitHub Actions, GitLab, Jenkins)**	Ejecuta tests automáticamente en cada push

4. Tipos de Pruebas

- Unitarias
  - Prueban una sola clase o método
  - Sin dependencias externas
  - Muy rápidas
- De integración
  - Verifican la colaboración entre componentes (ej. servicio + repositorio)
  - Usan la infraestructura real o simulada
- End-to-End (E2E)
  - Simulan escenarios reales de usuario
  - Lentas, frágiles y costosas, pero necesarias en puntos críticos
- Arquitectura
  - Validan que se cumpla la separación de capas y reglas de diseño

Por nivel de abstracción o técnica

- Pruebas de contrato (Contract Testing)
  - Verifican que las integraciones entre servicios cumplan con un contrato predefinido (útil en microservicios). Ej: Pact.
- Pruebas de aceptación (Acceptance Testing)
  - Validan que el sistema cumple con los criterios de aceptación definidos por el cliente o negocio. Ej: Cucumber con Gherkin.
- Pruebas de regresión (Regression Testing)
  - Aseguran que cambios recientes no rompan funcionalidades existentes.
- Pruebas de exploración (Exploratory Testing)
  - Realizadas manualmente por testers para descubrir fallos inesperados mediante el uso libre del sistema.

Por entorno o contexto

- Pruebas de sistema (System Testing)
  - Evalúan el sistema completo como un todo, normalmente en un entorno lo más cercano posible a producción.
- Pruebas en producción (Smoke, Canary, Shadowing)
  - *Smoke testing*: Validaciones mínimas después de un despliegue.
  - *Feature flags*: Ciertas funcionalidades solo disponibles a un grupo pequeño de usuarios para validar antes de liberarlo a todos.
- Canary releases: Validan gradualmente con usuarios reales.
  - *Shadow testing*: Ejecutan las mismas peticiones de producción sobre una versión paralela no visible al usuario.

Por confiabilidad y robustez

- Pruebas de carga (Load Testing)
  - Evalúan cómo responde el sistema bajo un número creciente de usuarios o peticiones.
- Pruebas de estrés (Stress Testing)
  - Llevan el sistema más allá de sus límites para ver cómo falla (y recupera).
- Pruebas de resiliencia o caos (Chaos Engineering)
  - Se inyectan fallos intencionales para comprobar la tolerancia a fallas. Ej: Chaos Monkey.
- Pruebas de recuperación (Recovery Testing)
  - Verifican que el sistema pueda recuperarse de errores, caídas o fallos de red.

Por seguridad

- Pruebas de seguridad (Security Testing)
  - Identifican vulnerabilidades de seguridad. Incluye pruebas de inyección, autenticación, autorización, etc.
- Pruebas de penetración (Pen Testing)
  - Simulan ataques reales para evaluar la postura de seguridad.

Por dispositivo o interfaz

- Pruebas de compatibilidad
  - Aseguran que el sistema funciona correctamente en distintos dispositivos, navegadores, sistemas operativos, etc.
- Pruebas de accesibilidad (A11y Testing)
  - Evalúan que personas con discapacidades puedan utilizar el sistema conforme a estándares como *Web Content Accessibility Guidelines* (WCAG).

Por calidad no funcional

- Pruebas de usabilidad
  - Evalúan la experiencia de usuario, comprensión de la interfaz, satisfacción, etc.
- Pruebas de rendimiento (Performance Testing)
  - Engloban pruebas de carga, estrés y tiempo de respuesta en condiciones normales.
- Pruebas de consumo de recursos
  - Evalúan el uso de CPU, memoria, disco, red, para evitar cuellos de botella o consumo excesivo.

Recomendación general

- Para un proyecto sólido de calidad, un enfoque balanceado podría incluir:
  - Unitarias - JUnit, Spock
  - Integración - Testcontainers, Spring Boot Test
  - End-to-end - Selenium, Cypress, Playwright
  - Arquitectura - ArchUnit
  - Contratos - Pact
  - Rendimiento - JMeter, Gatling
  - Seguridad - OWASP ZAP, Snyk
  - Aceptación - Cucumber

5. Tema Adicional: Test-Driven Development (TDD)

- ¿Por qué incluirlo? Es una práctica fundamental para calidad del código.
- Ciclo Red-Green-Refactor:
  - Red: Escribir prueba que falle (define el requerimiento).
  - Green: Implementar mínima funcionalidad que pase la prueba.
  - Refactor: Mejorar código manteniendo pruebas en verde.
- Beneficios en Java:
  - Código más modular y mantenible.
  - Diseño guiado por responsabilidades claras.
  - Documentación viva mediante pruebas.

Desarrollo Guiado por Pruebas (TDD Test-Driven Development)

- Objetivo
  - Entender los fundamentos, beneficios, ciclo de vida y buenas prácticas del Desarrollo Guiado por Pruebas (TDD)
  - Cómo aplicarlo eficazmente en proyectos Java para mejorar la calidad del software y fomentar un diseño más claro.

¿Qué es TDD?

- TDD es una práctica de desarrollo en la cual escribimos primero los tests antes del código de producción.
- Es tanto una técnica de diseño como de verificación, y nos obliga a pensar en el comportamiento esperado antes de escribir la implementación.
- “Write the test first, then make it pass, then refactor.”

Ciclo de TDD: RED – GREEN – REFACTOR

- RED
  - Escribir una prueba que falla.
  - Define qué comportamiento se espera.
  - Obliga a pensar en la API antes del código.
- GREEN
  - Escribir la mínima cantidad de código para que la prueba pase.
  - No se busca perfección, solo funcionalidad.
- REFACTOR
  - Mejorar el diseño del código manteniendo los tests verdes.
  - El test sirve como red de seguridad durante la refactorización.

Beneficios del TDD

- Diseño centrado en el comportamiento (BDD-lite)
- Menor acoplamiento y mayor cohesión
- Más confianza para refactorizar
- Documentación viva (los tests explican el comportamiento esperado)
- Menor cantidad de bugs en producción

Errores comunes al aplicar TDD

- Escribir mucho código sin tests previos
- No ejecutar el ciclo completo (omitir refactor)
- Tests acoplados a la implementación en lugar del comportamiento
- Pruebas muy genéricas o triviales

Herramientas para aplicar TDD en Java

- JUnit 5 – framework principal de testing
- IDE con soporte para ejecución rápida de tests (IntelliJ, Eclipse, VS Code)
- Mockito / Spock – para pruebas con dependencias o especificaciones
- Maven / Gradle – para ejecutar tests desde la consola o CI/CD
- CI Tools (GitHub Actions, Jenkins, GitLab CI) – para automatizar validaciones TDD en cada push

TDD en colaboración

- Se combina muy bien con pair programming.
- Ideal para entornos ágiles y de mejora continua.
- Puede utilizarse junto con DDD para guiar el diseño de objetos.

Cuándo usar TDD (y cuándo no)

- Ideal para:
  - Lógica de negocio crítica
  - Componentes reutilizables
  - Algoritmos con reglas bien definidas
- No siempre necesario en:
  - Prototipos rápidos
  - Código de infraestructura sin lógica (ej. configuración)

Notas finales sobre TDD

- TDD es una práctica poderosa que transforma la forma en que diseñamos y verificamos software.
- Aunque requiere disciplina y práctica, sus beneficios en términos de calidad, diseño y confianza en el código lo convierten en una herramienta esencial para todo desarrollador Java profesional.
- Escribir pruebas no es un paso posterior, es una forma de pensar antes de codificar.

6. Conclusión

- Las pruebas en Java son un pilar de calidad, no un accesorio.
- Combinar principios sólidos, la pirámide de testing y herramientas modernas permite crear software resiliente.
- La pirámide no es dogma, sino guía. Adapta las proporciones según tu dominio (ej: apps móviles pueden necesitar más E2E).
- TDD lleva estas prácticas al siguiente nivel integrando testing en el corazón del desarrollo.
- No se trata solo de encontrar bugs, sino de garantizar comportamiento esperado.
- Las pruebas deben ser rápidas, repetibles y automatizadas.
- Aplicar una estrategia piramidal asegura balance entre velocidad, cobertura y mantenimiento.
- Java cuenta con un ecosistema maduro y profesional de herramientas que permiten aplicar testing desde la unidad hasta la arquitectura.

Tarea

- Instalar:
  - Java 21.
    - Deseable 24
  - Maven 3.9.11