



Testing Efectivo en Java

Círculo Siete Capacitación

Clase 10 de 12
26 Agosto 2025

Métricas en Testing

- Métricas de cobertura y completitud
- Métricas de efectividad de pruebas
- Métricas de eficiencia
- Métricas de proceso
- Métricas de calidad de producto

Métricas de cobertura y completitud

- **Cobertura de código:** porcentaje de líneas, ramas, métodos o condiciones ejecutadas por las pruebas.
- **Cobertura de requerimientos:** qué porcentaje de requisitos funcionales y no funcionales están validados con pruebas.
- **Trazabilidad:** qué porcentaje de casos de prueba están ligados a historias de usuario/epics.

Métricas de efectividad de pruebas

- **Tasa de detección de defectos (Defect Detection Percentage, DDP):** defectos encontrados por pruebas / defectos totales encontrados (pruebas + producción).
- **Severidad de defectos:** distribución de defectos por criticidad (bloqueantes, mayores, menores, cosméticos).
- **Densidad de defectos:** número de defectos por KLOC (mil líneas de código) o por módulo.
- **Tasa de fuga de defectos (Defect Leakage):** defectos detectados en producción / total de defectos.

Métricas de eficiencia

- **Tiempo de ejecución de pruebas:** cuánto tarda la suite en ejecutarse (importante en CI/CD).
- **Tiempo medio para detectar un fallo (MTTD).**
- **Tiempo medio para resolver un fallo (MTTR):** desde que se reporta hasta que se corrige.
- **Automatización de pruebas:** porcentaje de pruebas automatizadas vs. manuales.

Métricas de proceso

- **Tasa de éxito de casos de prueba:** casos pasados / casos ejecutados.
- **Volumen de regresiones:** defectos reintroducidos después de cambios.
- **Estabilidad de builds:** número de builds verdes/rojos en la pipeline de CI.
- **Velocidad de feedback:** cuánto tarda en avisar la pipeline si algo se rompe.

Métricas de calidad de producto

- **Confiabilidad:** frecuencia de fallos en producción.
- **Disponibilidad:** porcentaje de uptime del sistema.
- **Rendimiento:** tiempo de respuesta promedio, percentiles (p95/p99).
- **Escalabilidad:** comportamiento del sistema bajo carga creciente.
- **Seguridad:** vulnerabilidades encontradas en pruebas (OWASP, SAST, DAST).

Reflexión

- En la práctica, no conviene medir todo, sino elegir las métricas alineadas al objetivo:
 - **Calidad interna:** cobertura, defectos, regresiones.
 - **Eficiencia del equipo:** velocidad de feedback, tiempo de ejecución.
 - **Calidad externa:** defectos en producción, NPS del usuario, rendimiento.

Nivel de Prueba	Métricas Clave	Qué Indican	Ejemplo / Herramienta
Unitarias	<ul style="list-style-type: none"> * Cobertura de código (líneas, ramas, mutación) * Tasa de éxito de tests * Tiempo de ejecución 	Qué tanto del código está validado y con qué rapidez fallan los errores	JUnit + JaCoCo / PIT Mutation
Integración	<ul style="list-style-type: none"> * Casos pasados vs. fallidos * Tiempo de feedback en CI * Defectos encontrados en integración vs unitarias 	Qué tan bien interactúan los módulos y qué tan rápido detectas fallos	Testcontainers, Spring Boot Test
End-to-End / Aceptación	<ul style="list-style-type: none"> * Cobertura de requerimientos * Densidad de defectos por historia de usuario * Tasa de fuga a producción 	Si las funcionalidades entregan valor completo y sin defectos críticos	Cucumber, Selenium, RestAssured
Regresión	<ul style="list-style-type: none"> * Volumen de regresiones * Estabilidad de builds (verdes/rojos) * Porcentaje de casos automatizados 	Si los cambios rompen lo que antes funcionaba	CI/CD + Jenkins/GitHub Actions
Performance / Carga	<ul style="list-style-type: none"> * Tiempo de respuesta (p95/p99) * Throughput (req/s) * Uso de recursos (CPU, memoria) * Escalabilidad bajo carga creciente 	Qué tan rápido, estable y escalable es el sistema	JMeter, Gatling
Seguridad	<ul style="list-style-type: none"> * Vulnerabilidades encontradas (SAST/DAST) * Cobertura de pruebas de seguridad * Tiempo medio para corregir vulnerabilidades (MTTR) 	Qué tan protegido está el sistema frente a ataques	OWASP ZAP, SonarQube, Snyk
Producción (Smoke / Canary / Shadowing)	<ul style="list-style-type: none"> * Defectos en producción * Confiabilidad (Mean Time Between Failures, MTBF) * Disponibilidad (uptime %) * Error rate 	La calidad real percibida por el usuario en operación	Observabilidad: Prometheus, Grafana, ELK, OpenTelemetry

Notas prácticas

- Para pruebas unitarias, más que la cobertura, destaca mutación (asegura que los tests realmente validen la lógica).
- Para integración, la métrica más valiosa es el tiempo de feedback en la pipeline.
- Para E2E, conviene medir fuga de defectos a producción porque es donde más duele.
- En performance, usa percentiles (p95/p99), no solo promedio.
- En producción, cruza métricas de testing con observabilidad (ej: tasa de errores después de un despliegue).

Métricas relevantes para Unit Testing

1. Cobertura de código

- Definición: porcentaje del código ejecutado por las pruebas unitarias.
- Subtipos:
 - Cobertura de líneas
 - Cobertura de ramas/condiciones
 - Cobertura de métodos/clases
 - Herramientas: JaCoCo, IntelliJ Coverage.
- Limitación: alta cobertura no garantiza calidad de tests (por eso complementa con mutación).

2. Cobertura por mutación

- Definición: mide la capacidad de las pruebas para detectar cambios artificiales (“mutantes”) en el código.
- Indicador: % de mutantes detectados por los tests.
- Herramientas: PIT Mutation Testing.
- Valor: más confiable que la cobertura de líneas, porque verifica si los tests realmente validan lógica.

3. Tasa de éxito de casos de prueba

- Definición: proporción de pruebas que pasan vs. ejecutadas.
- Indicador: $(\text{tests pasados} / \text{tests ejecutados}) * 100$.
- Valor: útil para ver estabilidad del código y detectar flakiness.

4. Tiempo de ejecución

- Definición: cuánto tardan en ejecutarse las pruebas unitarias.
- Objetivo: mantenerlas rápidas ($<1s$ cada una, segundos en total).
- Valor: feedback rápido en CI/CD.

5. Volumen de pruebas por módulo

- Definición: cantidad de pruebas unitarias vs. complejidad del módulo.
- Correlación: módulos más críticos o complejos deberían tener más pruebas.
- Herramientas: SonarQube (código + tests por paquete).

6. Defectos detectados en producción atribuibles a unit testing

- Definición: cuántos defectos pudieron haberse detectado con pruebas unitarias y no lo fueron.
- Indicador: % de defectos funcionales simples escapados a integración o producción.
- Valor: mide efectividad real de los unit tests.

7. Mantenibilidad de tests

- (No siempre es cuantitativa, pero sí medible con heurísticas.)
- Complejidad de los tests (ej: demasiados mocks → difícil mantener).
- Duplicación en el código de pruebas.
- Legibilidad (tests con nombres claros vs. genéricos).
- Herramientas: SonarQube, PMD, revisión manual.

Resumen

- Cobertura de código (líneas, ramas, métodos).
- Cobertura por mutación (calidad real de los tests).
- Tasa de éxito / estabilidad.
- Tiempo de ejecución (rápidos = útiles).
- Volumen de tests vs. complejidad del módulo.
- Defectos escapados atribuibles a unit testing.
- Mantenibilidad de tests (legibilidad, duplicación, mocks).


Cobertura de código: qué es, cómo usarla bien, y dónde se rompe

1) Qué es (y tipos que sí importan)

- Líneas: % de líneas ejecutadas. Fácil de “inflar”.
- Ramas/decisiones: % de ramas de if/else/switch/?: ejecutadas. Mucho más informativa.
- Condiciones: evalúa cada condición booleana dentro de una decisión (a && b). Útil en lógica compleja.
- Mutación (complemento, no cobertura): mide si tus tests fallarían ante cambios sutiles. Es la prueba de fuego de que la cobertura no es “humo”.
- Regla: usa líneas para monitoreo global, ramas para calidad real, y mutación para validar que los tests tienen aserciones útiles.

2) Umbrales recomendados (basados en riesgo)

- No impongas un número uniforme; ajusta por criticidad:

Componente	Líneas 	Ramas	Mutación
Dominio/crítico	80-90%	70-80%	60-70%
Servicios/Aplicación	70-80%	60-70%	40-60%
Adaptadores/infra (DTOs, mappers)	50-60%	40-50%	opcional

3) Buenas prácticas - 1/2

- Mide por módulo (no solo global). Un agregado alto oculta “islas” sin pruebas.
- Protege ramas críticas: reglas de negocio, autorizaciones, cálculos monetarios, manejo de errores.
- Cubrir bordes y equivalencias: límites (min/max), nulos, colecciones vacías, valores repetidos.
- Evita “tests de turismo” (que solo pasean por el código): cada test debe afirmar el comportamiento.
- Diseña para testear: funciones puras en el dominio, puertos/mocks en los límites; reduce estáticos/singletons.
- Excluye generado/ceremonial (clases Lombok, DTOs simples, mapeos triviales) del objetivo de cobertura, pero no del reporte.

3) Buenas prácticas - 2/2

- Observa ramas “faltantes”: en el reporte de Jacoco, prioriza las decisiones no cubiertas (suelen revelar reglas no probadas).
- Refactoriza tests lentos: unitarias deben correr en segundos; si tardan, quizá son de integración.
- Combina con mutación (PITEST): te protege contra aserciones laxas o tests que no fallan cuando deberían.
- Usa quality gates por riesgo: distintos mínimos para :domain, :app, :infra.
- Monitorea tendencias (no solo el número actual): una caída sostenida es señal de deuda.
- Revisa PRs por huecos específicos: “Esta rama de error no tiene test”, “este catch nunca se ejecutó”.

4) Anti-patrones (alerta roja)

- Cobertura-driven development: escribir tests para “pintar verde” sin validar reglas reales.
- Aserciones vacías: tests que no verifican nada sustantivo.
- Mockear todo: tantos dobles que ya no pruebas nada útil (pierdes regresión semántica).
- Perseguir 100%: genera costo altísimo, especialmente en infraestructura; enfoca ese esfuerzo en dominio.
- Excluir lo incómodo: sacar del reporte justo donde duele (manejo de errores, fallbacks). Mejor probar o refactorizar.

5) Limitaciones (por qué sola no basta)

- Falso sentido de seguridad: puedes tener 90% y aún así no detectar un bug si las aserciones son débiles.
- No mide oráculos: ignora si estás validando lo correcto (de ahí el valor de mutación).
- Reflexión/Proxies: parte del flujo real puede no contarse (AOP, proxys de Spring).
- Concurrencia/tiempo: condiciones de carrera, timeouts y retrasos no se revelan con cobertura.
- Código muerto: aumenta el denominador; la mejor cobertura ahí es borrarlo.



pitest.org

PIT Mutation Testing




<https://pitest.org/>

1) ¿Qué es PIT Mutation Testing?

- PIT inyecta mutantes (cambios pequeños en tu bytecode: `>`, `>=`, `+`, `-`, `return`, etc.) y verifica si tus tests fallan.
- Mutante “asesinado” \Rightarrow el test falló: bien.
- Mutante “sobreviviente” \Rightarrow el test no detectó el cambio: hay un hueco en las aserciones o en los casos.
- Por qué usarlo: la cobertura de líneas puede mentir; la mutación valida que tus aserciones realmente protegen la lógica.

2) Umbrales y estrategia por riesgo

- No persigas 100% de mutación. Usa objetivos por criticidad:
- Consejo: si introduces mutación en dominio, puedes relajar levemente cobertura de líneas sin perder calidad.

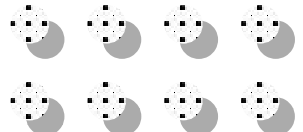
Módulo	Umbral mutación 	Notas 
Dominio/reglas de negocio	60-70%	Apunta a matar mutantes en cálculos, validaciones y autorizaciones. 
Servicios/aplicación	40-60%	Cobertura razonable, prioriza rutas de error. •
Infra / mappers / DTO	opcional	Ejecuta muestreo solo si el costo es bajo. •

3) Cómo escribir tests “mata-mutantes”

- Aserciones específicas: valida salidas concretas, no “no es null” genérico.
- Casos por particiones y bordes: (min, max, cero, negativos, colecciones vacías).
- Ramas de error/exception: espera excepciones, mensajes o estados.
- Property-based (donde aplique): invariantes (con lib rápida o aleatoria propia).
- Evita mocks innecesarios en dominio: los mocks difuminan señales.
- Datos representativos: usa Test Data Builders / Object Mother claros.

Laboratorio

- Revisar labs - PIT



Lectura del reporte (qué priorizar)

- Survived: casos faltantes o aseveraciones débiles. Revisa línea y mutador que sobrevivió.
- No Coverage: la línea nunca se ejecutó (agrega test o elimina código muerto).
- Timed Out: la prueba cuelga; revisa timeouts o dependencias externas.
- Run Error: test inestable; corrige flakiness.
- Enfócate en survivors de módulos críticos y en no coverage de decisiones importantes.

Equivalentes y falsos positivos (inevitables pero manejables)

- Algunos mutantes no cambian el comportamiento observable (equivalentes).
- Marca rutas no observables para exclusión fina (método/clase) o agrega observabilidad (p.ej., expón estado/resultado).
- No “silencios” sobrevivientes sin analizar: verifica primero si falta una aserción o caso de borde.

Integración con Spring/Testcontainers/Mockito

- Dominio puro: sin Spring, sin contenedores → es donde PIT brilla.
- Servicios con Spring: considera tests de slice (@DataJpaTest, @WebMvcTest) y evita levantar el contexto completo si no aporta.
- Testcontainers: úsalo en integración; evita mutación allí (lento).
- Mockito: ok, pero no maquilles dominio con mocks: mejor tests contra objetos reales del core.