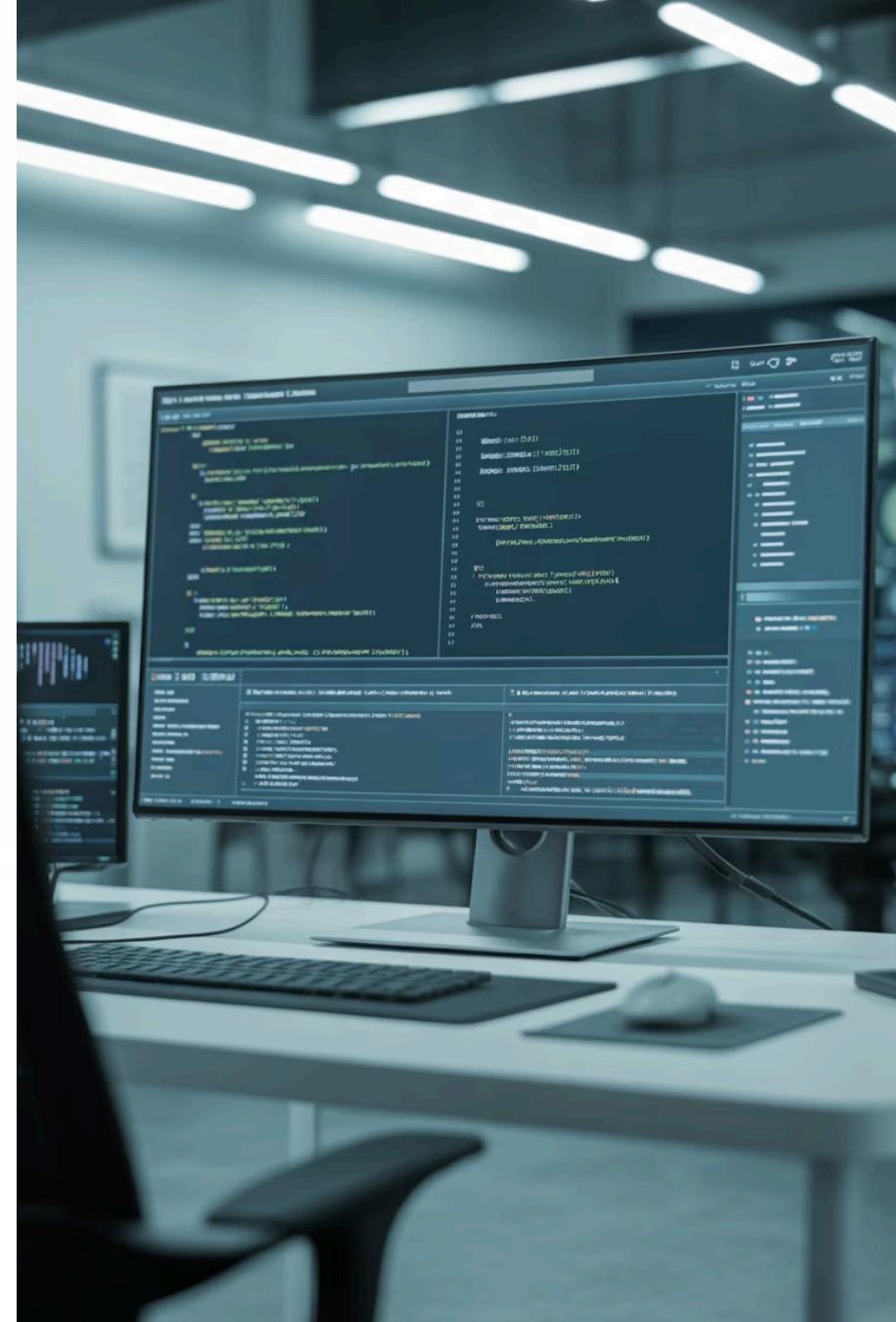


Bonnes Pratiques de Tests en Java

Guide complet sur les tests unitaires, d'intégration et l'utilisation stratégique
des bouchons





Pourquoi les tests sont essentiels

Fondement robuste

Assurance contre régressions et défauts production

Investissement stratégique

Facilite refactorisations, accélère nouvelles fonctionnalités

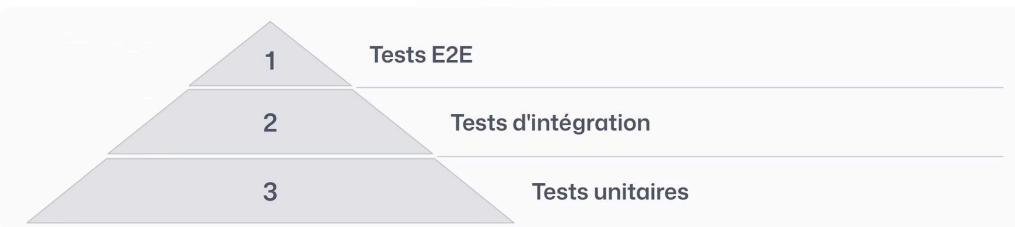
Documentation vivante

Comportements attendus, cas limites, exemples concrets

Confiance d'équipe

Modifications sereines, détection immédiate des régressions

La pyramide des tests



Distribution optimale

- **70% Tests unitaires** : rapides, focalisés, nombreux
- **20% Tests d'intégration** : interactions entre composants
- **10% Tests E2E** : parcours utilisateur complets

Équilibre entre couverture, rapidité et coût de maintenance

Principes fondamentaux



Isolation

Chaque test indépendant, une seule unité de code testée



Rapidité

Exécution en millisecondes pour usage fréquent



Répétabilité

Même résultat à chaque exécution, conditions identiques



Clarté

Code test lisible et maintenable comme code production



JUnit 5 : Framework de référence

Architecture modulaire

JUnit Platform, Jupiter, Vintage - flexibilité exceptionnelle

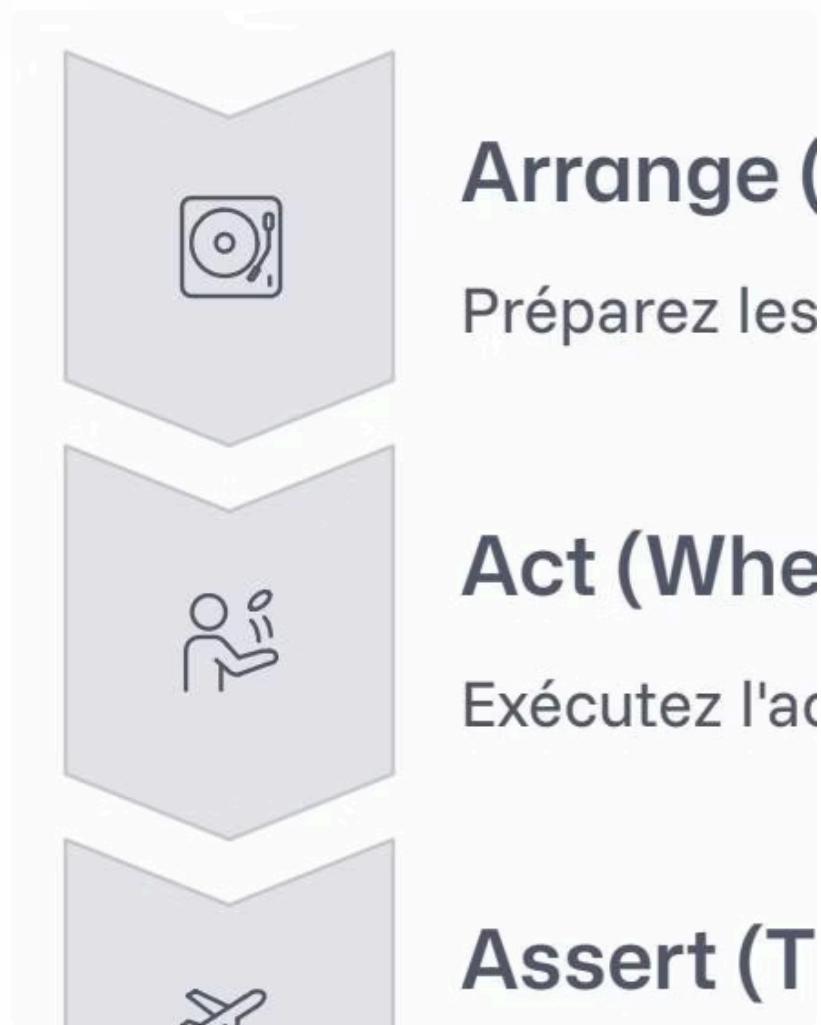
Annotations expressives

`@DisplayName`,
`@ParameterizedTest`, `@Nested`
 pour clarté

Assertions évoluées

`assertAll()`, `assertThrows()`, `assertTimeout()` modernes

Pattern AAA : Structure claire



01

Arrange (Given)

Préparer données et environnement

02

Act (When)

Exécuter action ou méthode testée

03

Assert (Then)

Vérifier résultat correspond aux attentes

Organisation cohérente et prévisible pour lisibilité maximale



Nommage expressif des tests



Trois éléments clés

Méthode testée, scénario spécifique,
comportement attendu



Convention recommandée

methodName_scenarioDescription_expectedBehavior()



@DisplayName

Phrases complètes en langage naturel
pour rapports lisibles

Exemple : calculateVAT_with100EurosAmount_returns20Euros()

Tests paramétrés

Maximiser la couverture

Valider comportement avec plusieurs jeux de données
sans duplication

- @ValueSource : valeurs simples
- @CsvSource : paramètres multiples
- @MethodSource : cas complexes



Réduit duplication, augmente couverture, simplifie diagnostic



```
package com.example.test;

import static org.junit.jupiter.api.Assertions.*;

import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.CsvFileSource;
import org.junit.jupiter.params.provider.CsvSource;
import org.junit.jupiter.params.provider.MethodSource;
import org.junit.jupiter.params.provider.ValueSource;

public class ExampleTest {
    @Test
    void test() {
        List<String> list = Stream.of("a", "b", "c").collect(Collectors.toList());
        assertEquals(3, list.size());
        assertEquals("a", list.get(0));
        assertEquals("b", list.get(1));
        assertEquals("c", list.get(2));
    }

    @ParameterizedTest
    @CsvFileSource(resources = "data.csv")
    void testWithCsvFile(String expected, List<String> list) {
        assertEquals(expected, list.get(0));
    }

    @ParameterizedTest
    @CsvSource(value = {"a,b,c"}, delimiter = ',')
    void testWithCsvString(String expected, List<String> list) {
        assertEquals(expected, list.get(0));
    }

    @ParameterizedTest
    @ValueSource(strings = {"a", "b", "c"})
    void testWithValueSource(String expected, List<String> list) {
        assertEquals(expected, list.get(0));
    }

    @ParameterizedTest
    @MethodSource("provideList")
    void testWithMethodSource(List<String> list) {
        assertEquals("a", list.get(0));
    }

    @ParameterizedTest
    @ArgumentsSource(CsvFileSource.class)
    void testWithArgumentsSource(String expected, List<String> list) {
        assertEquals(expected, list.get(0));
    }
}

class CsvFileSource implements ArgumentsSource {
    private final String resource;
    private final String encoding;
    private final String separator;
    private final String quote;

    public CsvFileSource(String resource, String encoding, String separator, String quote) {
        this.resource = resource;
        this.encoding = encoding;
        this.separator = separator;
        this.quote = quote;
    }

    @Override
    public Stream<Arguments> getArgumentsForTest() {
        return null;
    }
}
```

AssertJ : Assertions fluides

API intuitive

Style code ressemble langage naturel, autocomplétion IDE

Messages clairs

Erreurs détaillées avec valeurs attendues et réelles formatées

Chaînage naturel

Vérifications sophistiquées en quelques lignes élégantes

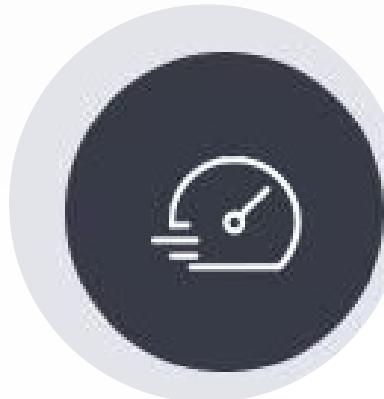
```
assertThat(list)
    .hasSize(5)
    .contains("item")
    .doesNotContainNull();
```

Introduction aux bouchons



Isolation

Code testé isolé de dépendances externes



Performance

Élimination appels lents : BD, API, fichiers



Contrôle

Simulation scénarios difficiles : erreurs, timeouts

iMOCKITO

Mockito : Framework standard

1

Création simple

@Mock ou mock(MyClass.class) en une ligne

2

Configuration élégante

when().thenReturn() pour comportement souhaité

3

Vérification puissante

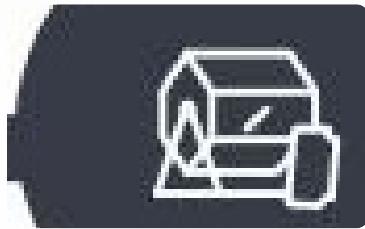
verify() pour valider interactions et appels

Types de doublures de test



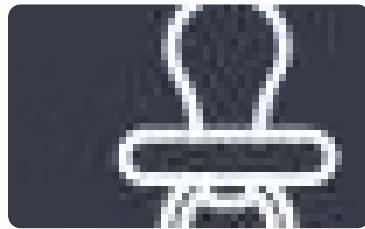
Fake

Implémentation simplifiée
fonctionnelle (BD mémoire)



Stub

Réponses prédefinies, valeurs
en dur



Mock

Vérifie interactions, enregistre
appels



Spy

Objet réel partiellement
substitué

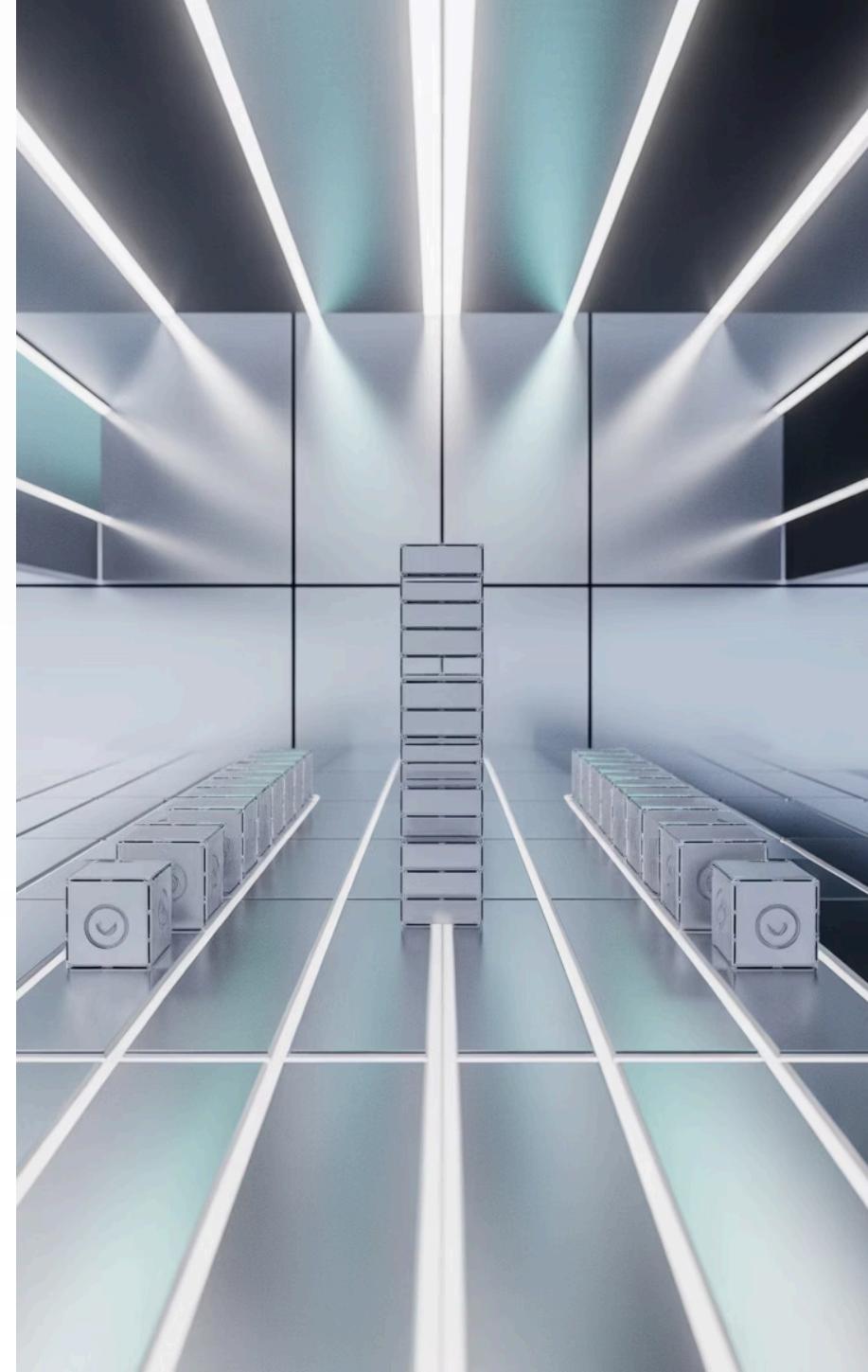
Injection de dépendances

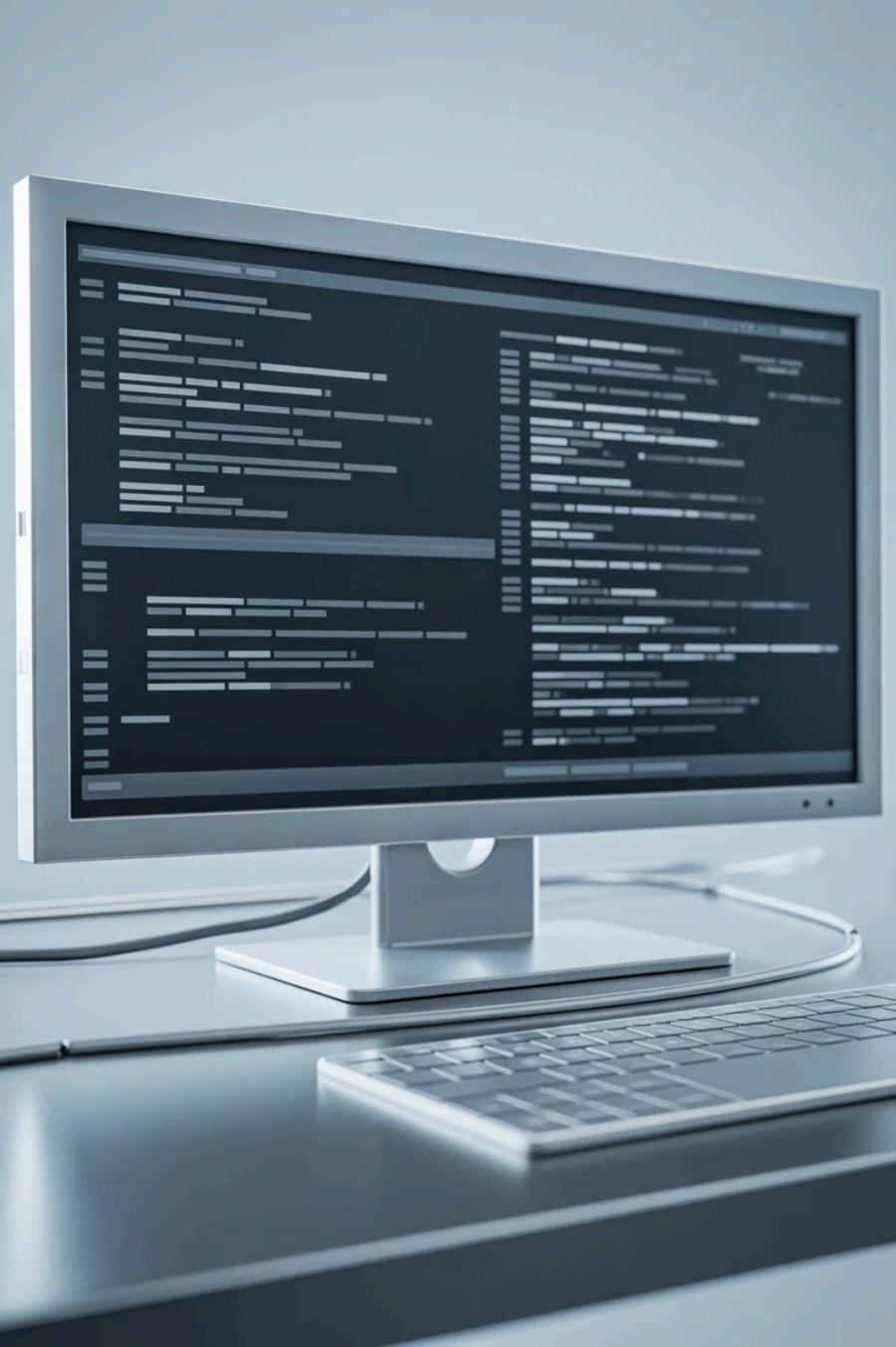
X Code non testable

```
public class OrderService {  
    private PaymentGateway  
    gateway =  
        new PaymentGateway();  
  
    public void process() {  
        gateway.charge();  
    }  
}
```

✓ Code testable

```
public class OrderService {  
    private final PaymentGateway  
    gateway;  
  
    public  
    OrderService(PaymentGatewa  
y g){  
        this.gateway = g;  
    }  
  
    public void process() {  
        gateway.charge();  
    }  
}
```





Patterns de stubbing

Retour valeur

```
when(mock.method()).thenReturn(value)
```

Lancer exception

```
thenThrow(new Exception())
```

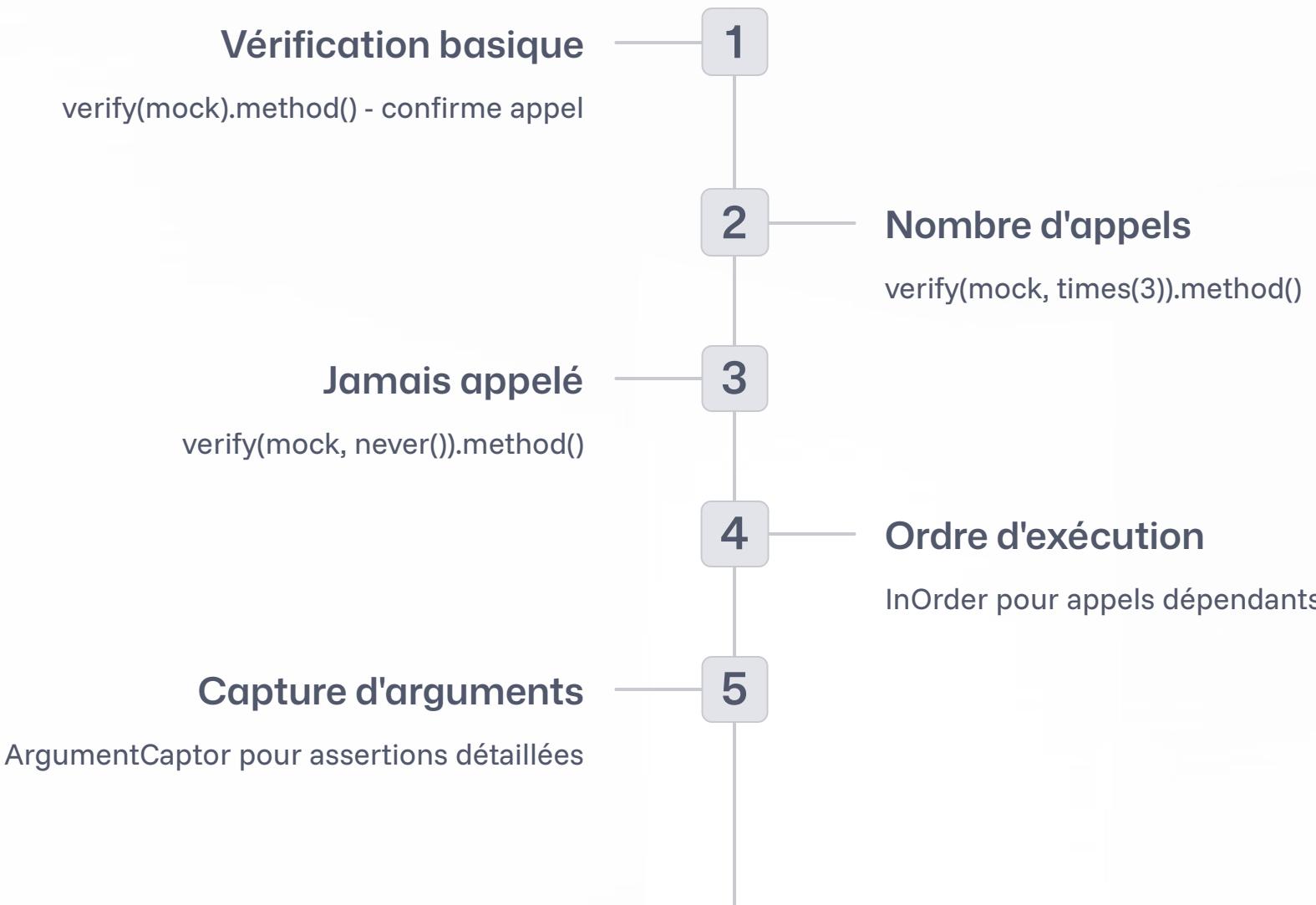
Réponse dynamique

```
thenAnswer() pour calculs
```

Matchers flexibles

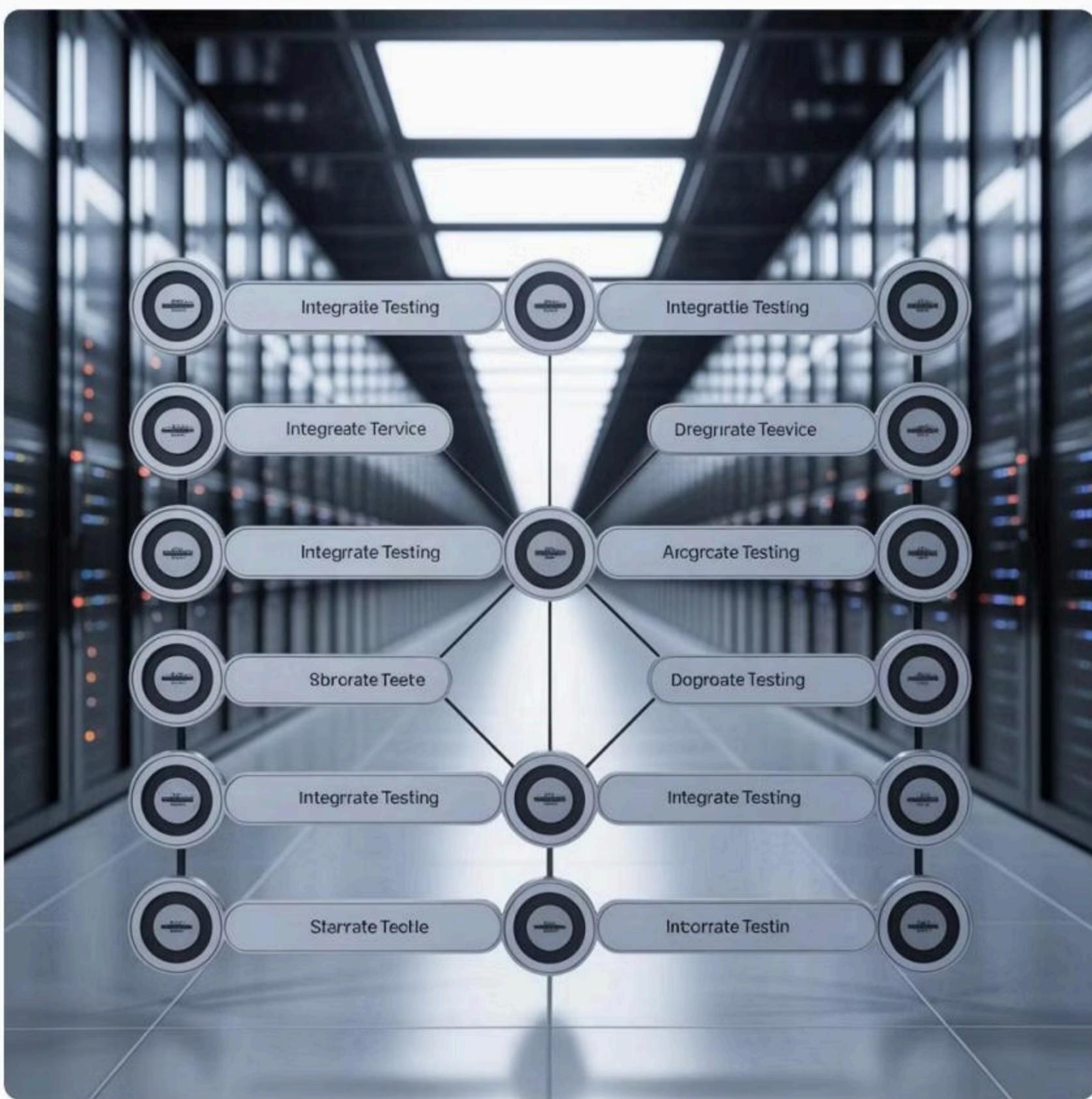
```
any(), eq(), argThat()
```

Vérification des interactions





Tests d'intégration



Spring Boot Test

Écosystème complet

- `@SpringBootTest` : contexte complet
- `@WebMvcTest` : couche web uniquement
- `@DataJpaTest` : JPA et BD mémoire
- `@JsonTest` : sérialisation JSON

TestRestTemplate et MockMvc pour API REST

`@ActiveProfiles("test")` pour configurations spécifiques tests



Bases de données de test

H2 en mémoire

BD SQL embarquée, tests rapides, compatible dialectes SQL

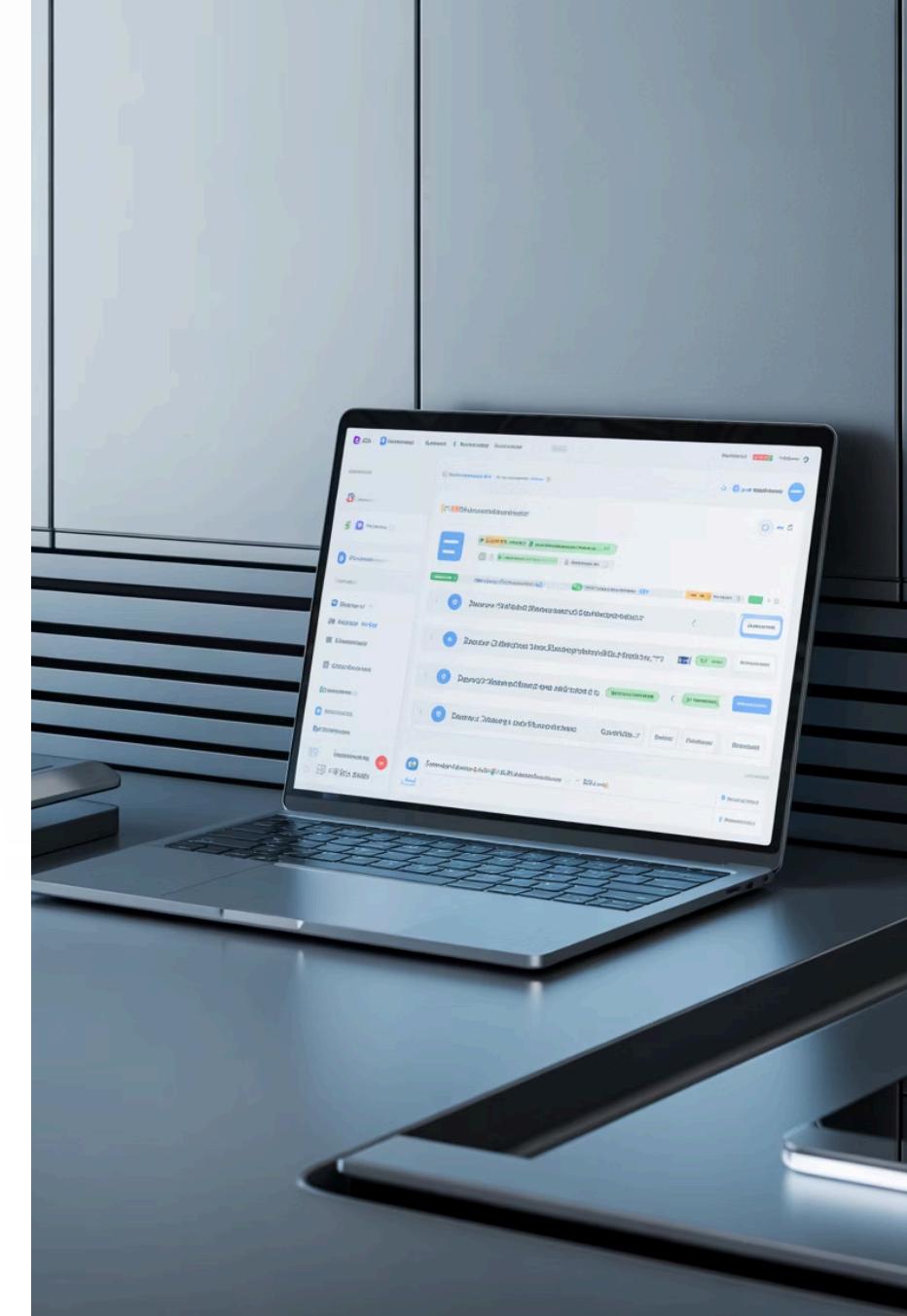
Testcontainers

Conteneurs Docker réels, même BD que production

Base dédiée

Instance persistante pour tests intégration longs

Approche hybride : H2 pour tests unitaires, Testcontainers pour intégration critique



Testcontainers : Docker pour tests

```
@Testcontainers  
class RepositoryTest {  
    @Container  
    static  
    PostgreSQLContainer  
    postgres = new  
  
    PostgreSQLContainer(  
        "postgres:15");  
  
    @DynamicPropertySo  
urce  
    static void props(  
  
        DynamicPropertyRegi  
stry r) {  
  
        r.add("spring.datasour  
ce.url",  
  
            postgres::getJdbcUrl);  
    }  
}
```

Avantages majeurs

- Même technologies que production
- Élimine "ça marche sur ma machine"
- Gestion automatique démarrage/arrêt
- Ports aléatoires évitent conflits





Tests d'API REST



MockMvc

API fluide pour requêtes et vérifications réponses HTTP



@WebMvcTest

Charge uniquement couche web, dépendances mockées



TestRestTemplate

Vraies requêtes HTTP, serveur complet démarré



RestAssured

Syntaxe BDD élégante, validations JSON puissantes

Gestion des transactions

1

@Transactional

Rollback automatique après chaque test, isolation garantie

2

@Commit

Vraies transactions committées pour cas spécifiques

3

@Sql

Scripts SQL pour setup et cleanup complets

I test

automatiquement démarré

n

pérations en base se font d

st

tomatique : la base reste pr

WireMock : Simuler API externes

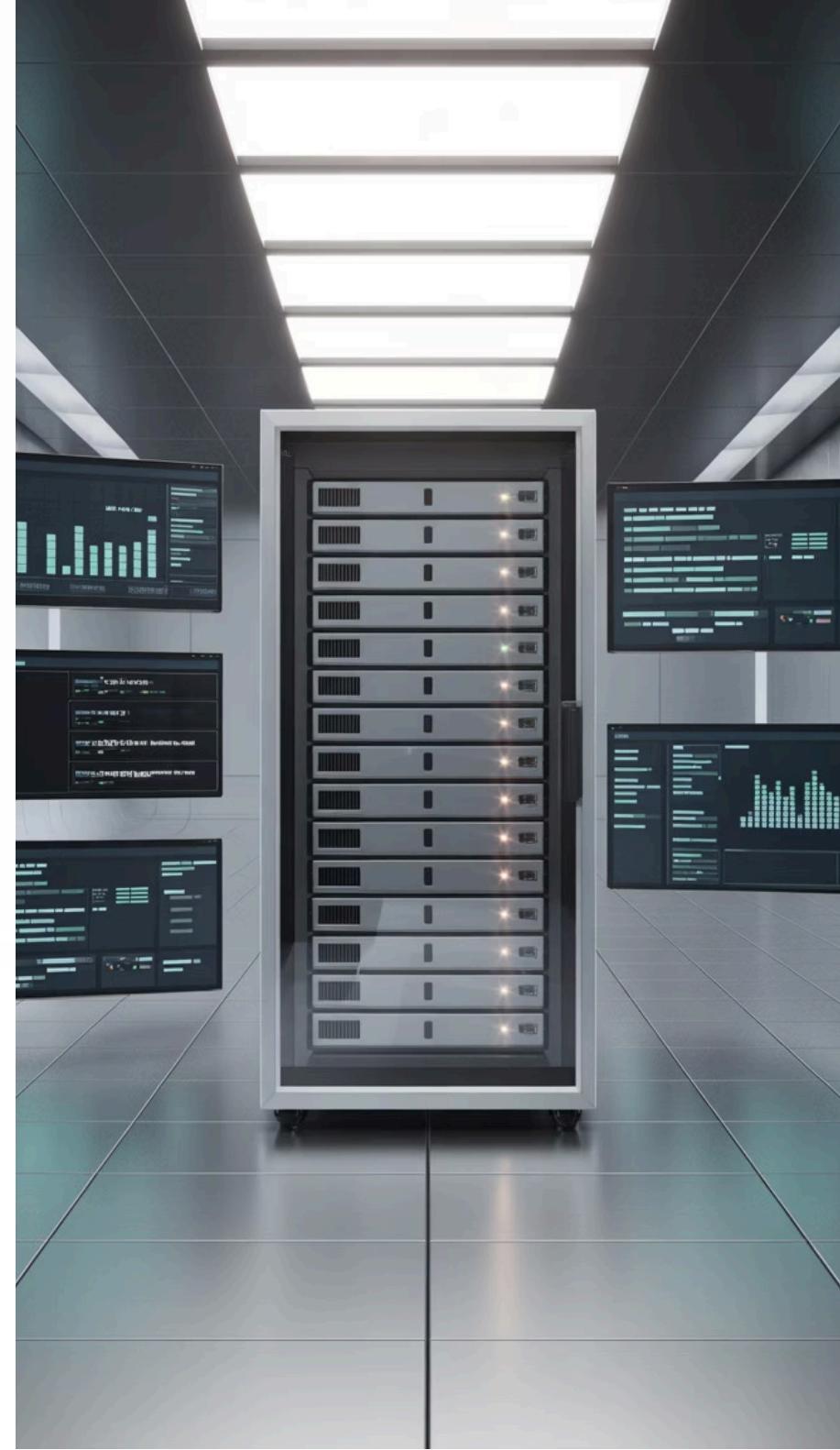
Serveurs HTTP simulés

Vrai serveur répondant selon configurations

- Patterns URL sophistiqués
- Regex et JsonPath
- Simulation latences et erreurs

```
@WireMockTest
class ApiClientTest {
    @Test
    void test GetUser() {
        stubFor(get("/api/users/1")
            .willReturn(ok()
                .withHeader("Content-Type",
                    "application/json")
                .withBody(
                    "{\"name\":\"John\"}")));
    }

    User user = client.getUser(1);
    assertThat(user.getName())
        .isEqualTo("John");
    }
}
```



Tests de performance

1

@Timeout JUnit

Échec si opération dépasse durée limite

2

JMH Benchmarks

Mesures précises avec précision statistique

3

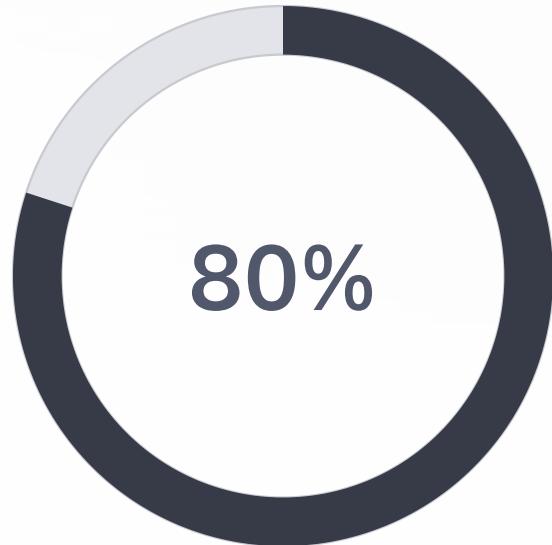
Gatling/JMeter

Tests charge, nombreux utilisateurs concurrents

- ❑ Environnement stable requis : isolez tests, collectez plusieurs échantillons

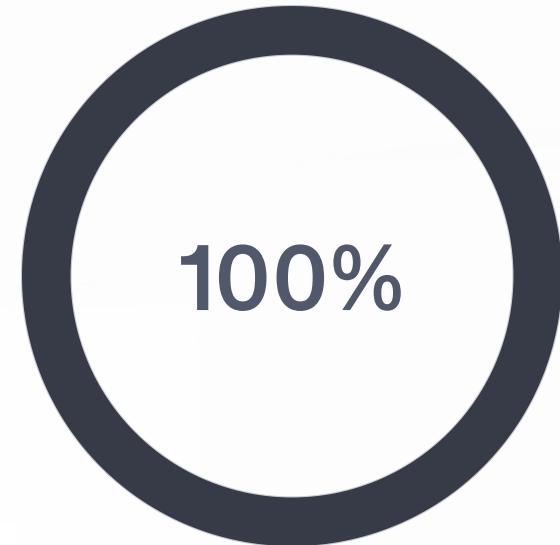


Couverture de code



Objectif réaliste

Couverture code métier critique



Faux objectif

Ni nécessaire ni suffisant pour qualité

JaCoCo génère rapports détaillés - focalisez zones critiques, pas 100% mécanique

Branch coverage plus révélateur que line coverage

Mutation testing



Qualité des tests

PITest introduit bugs délibérés, vérifie détection

- Inverse conditions
- Change opérateurs
- Supprime appels méthode

Révèle tests faibles avec assertions insuffisantes

Coûteux en temps - réservez code critique ou exécution périodique



Refactoring en confiance

Tests comportements publics

Résistent refactorings, cassent uniquement si comportement change

Petites étapes incrémentales

Changez une chose, testez, committez - minimise risques

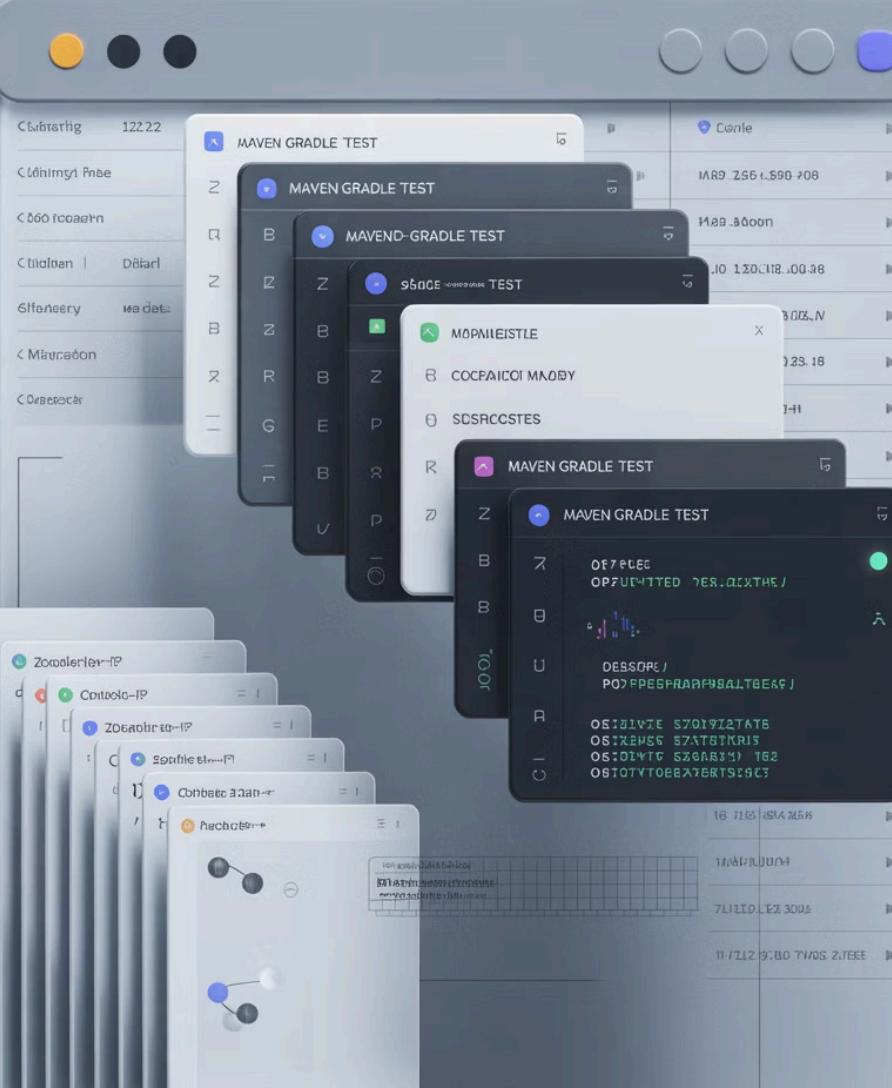
Refactorings IDE automatisés

Combinaisons audacieuses grâce aux tests

TDD : Test-Driven Development



Project structure organization



Organisation projet

Structure Maven/Gradle

```
src/  
  main/java/  
  test/java/  
  integration-test/  
  java/
```

Tests mirrorrent arborescence code source

Conventions nommage

- Tests unitaires :
`ClassNameTest.java`
- Tests intégration :
`ClassNameIntegrationTest.java`
- Tests E2E :
`FeatureNameE2ETest.java`



Intégration Continue



Tests unitaires

Feedback rapide, quelques minutes maximum



Tests intégration

Si unitaires passent, validation interactions



Tests E2E/performance

Branche spéciale ou calendrier défini

Parallélisation pour accélération, surveillance tests instables, blocage merges si échec

Vers l'excellence des tests

Investissement stratégique

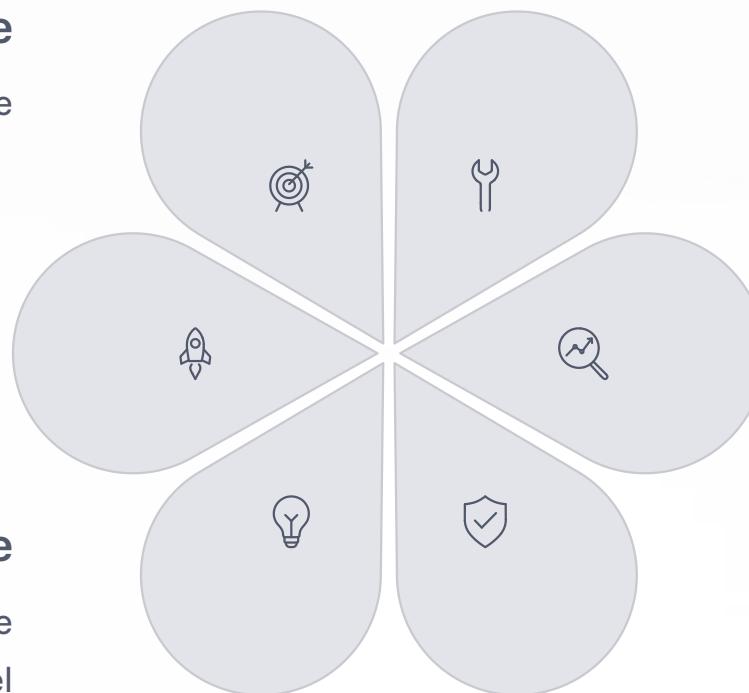
Qualité, maintenabilité, vélocité équipe

Avantage compétitif

Excellence tests = voyage continu, pas destination

Priorisation intelligente

80% couverture bien conçue surpassé 100% superficiel



Écosystème mature

JUnit 5, Mockito, AssertJ, Spring Boot Test

Pratique constante

Standards clairs, discipline équipe, amélioration continue

Confiance totale

Livraison rapide, stabilité code garantie