

Méthodologie de tests et tests unitaires

Mastère 2 - Ynov

Objectif du cours



Comprendre les usages des différents types de test



Comprendre comment mettre en place des tests efficaces



Écrire des tests en utilisant des frameworks adaptés



Apprendre à écrire du code testable et maintenable



Savoir mettre en place les outils nécessaires à l'exécution des tests

Critères d'évaluation

Test (30% de la note finale)

Vérification des acquis théoriques de la formation

TP sur un nouveau cas d'utilisation (70% de la note finale)

Mise en pratique du cours en autonomie

Capacité à reproduire les différents types de test

Plan du cours

1. Introduction aux tests logiciels
2. Tests unitaires et bonnes pratiques
3. Mise en place des tests dans un projet
4. Tests d'intégration et de composants
5. Tests non-fonctionnels
6. Tests End-To-End

Introduction aux tests logiciels

Qu'est-ce qu'un test ?

Objectifs des tests logiciels

- Garantir la qualité et la fiabilité des logiciels.
- Détecter les bugs et les problèmes avant qu'ils n'atteignent les utilisateurs.
- Assurer la conformité aux spécifications et aux exigences.

Rôle essentiel des tests dans le développement

- Les tests ne sont pas une étape optionnelle, mais un processus intégré.
- Éviter les perturbations majeures et coûteuses en cours de développement.
- Contribuer à une itération plus rapide et plus efficace du cycle de développement.

Garantie de la qualité et de la fiabilité

- Les tests renforcent la confiance des utilisateurs dans le logiciel.
- Réduisent les risques d'erreurs critiques et de dysfonctionnements.
- Permettent des mises à jour plus fluides et des corrections plus rapides.

Conséquences des bugs non détectés

Cas historiques de bugs catastrophiques

- Bug de l'an 2000 (Y2K) - Perturbations dues aux années à deux chiffres.
- Fusée Ariane 5 (1996) - Surcharge d'entiers, explosion en vol.

Pertes financières et de réputation

- Exemple : Faille de sécurité Heartbleed (2014) - Vol de données, impact sur la réputation.
- Impact sur la confiance des utilisateurs et les parts de marché.

Importance de la détection précoce

- Détecter et corriger les bugs tôt réduit les coûts de correction.
- Éviter les coûts associés aux rappels, aux remplacements et aux litiges

Avantages des tests

Fiabilité

- Identifier les faiblesses du logiciel
- Réduire les risques de dysfonctionnements en production

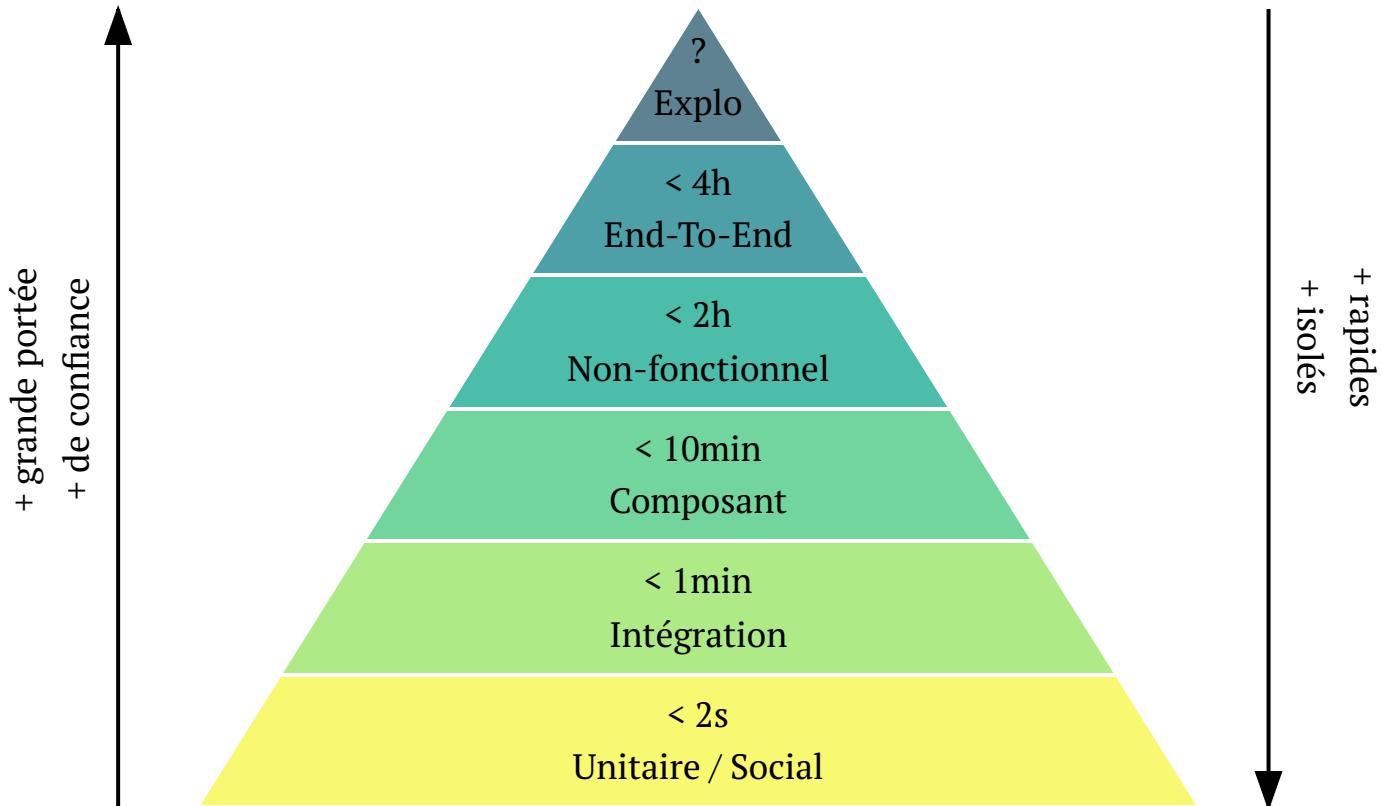
Maintenance

- Détection précoce = réduction des coûts
- Nouvelles évolutions plus simples à mettre en place

Stabilité

- Logiciel de bonne qualité = clients satisfaits
- Gain en image de marque

Pyramide des tests



Tests unitaires et bonnes pratiques

Qu'est-ce qu'un test unitaire ?

Test automatisé sur une unité de code individuelle (fonction, méthode, classe) afin de vérifier son comportement de manière isolée.

L'objectif est de valider que cette partie du code fonctionne correctement en testant le maximum de scénarios possibles.

Ecrits par les développeurs pour garantir la qualité du code.

Principe F.I.R.S.T.



Fast (Rapide)

0.5s/test, c'est trop lent. Des milliers de tests à exécuter



Isolated/Independent
(Isolé/Indépendant)

Tester une unique fonctionnalité à la fois

Aucun effet de bord pouvant interférer avec les autres tests



Repeatable (Répétable)

Doit toujours donner le même résultat à chaque exécution



Self-validating (Auto-validant)

Les tests doivent explicitement être en succès ou en erreur, ne doit pas être vérifié manuellement



Thorough/Timely
(Approfondi/Opportun)

Tester tous les cas possibles

Ecrire les tests avant/pendant le code

Quoi tester ?

Cas nominal

Tout se passe parfaitement

Cas limite

Relatif aux règles métiers

Relatif aux contraintes techniques et physiques

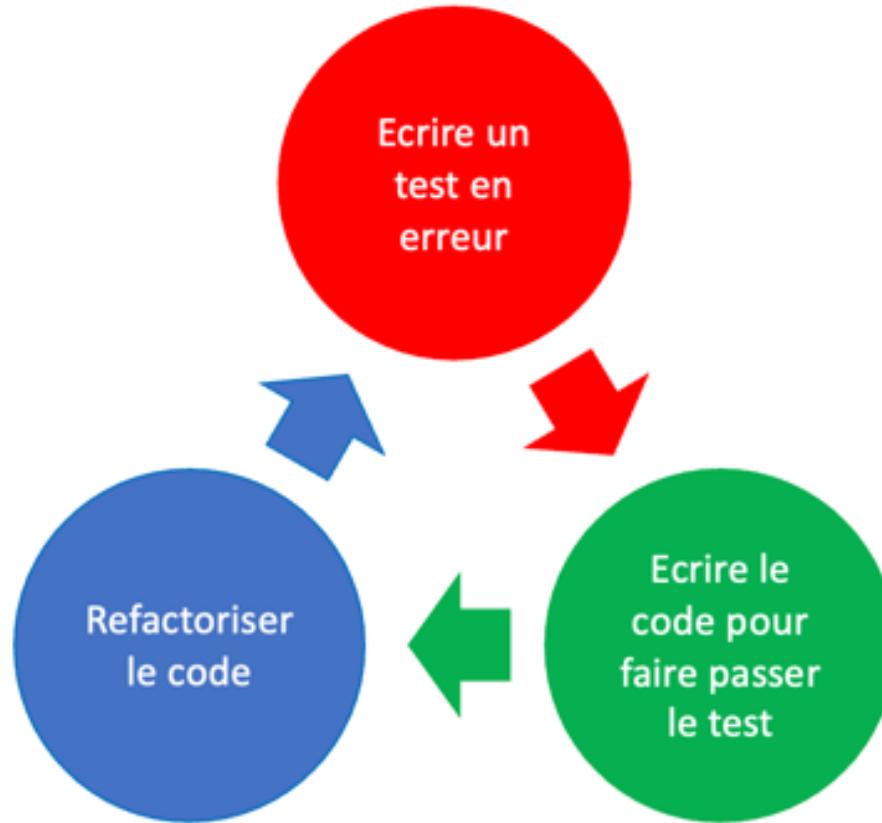
Cas pathologique

Situations improbables, souvent liés à des outils extérieurs

Structure d'un test (AAA)

Arrange: Initialisation	Créer les variables nécessaires Initialiser tout ce qui est nécessaire	<pre>test("add numbers 1 and 2 should equals to 3") { // Arrange val a = 1 val b = 2 val addOperator = AddOperator() // Act val res = addOperator.add(a, b) // Assert res shouldBe 3 }</pre>
Act: Action	Effectuer le test (appeler la fonction, cliquer sur le bouton,...) Récupérer le résultat du test	
Assert: Vérification	Comparer le résultat du test avec ce qui était attendu En cas d'échec lors de cette phase, le test est KO	

Test Driven Development



Comment écrire un test

- Framework de test : Kotest
- Bibliothèque d'assertion : Kotest Assertion

```
import io.kotest.core.spec.style.FunSpec
import io.kotest.matchers.shouldBe
import io.kotest.property.checkAll

class AdditionTest : FunSpec(){

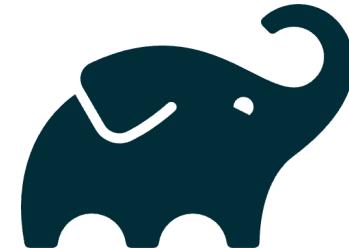
    test("add numbers 1 and 2 should equals to 3") {
        // Arrange
        val a = 1
        val b = 2
        val addOperator = AddOperator()

        // Act
        val res = addOperator.add(a, b)

        // Assert
        res shouldBe 3
    }
})
```

Exécution des tests automatisés

- Doivent être exécutés régulièrement
- Il n'existe pas "d'erreur normale"
- L'IDE permet d'exécuter les tests
- Exécution en ligne de commande en utilisant Gradle `./gradlew test`



Rapport d'exécution

Test Summary

[Failed tests](#)[Packages](#)[Classes](#)

[CesarCipherTest.cipher 'A' with key 0 should return 'A'](#)

General

⌚ Test Results	494 ms
⌚ AdditionTest	75 ms
✅ add numbers 1 and 2 should equals to 3	9 ms
⌚ add numbers 1 and 2 should equals to 4	8 ms
✅ identity of addition	37 ms
✅ associativity of addition	11 ms
✅ cipher should be circular	10 ms
> ✅ CesarCipherPropertyBasedTest	36 ms
> ✅ CesarCipherTest	2 ms
> ✅ UnitTestsApplicationTests	381ms

⌚ Tests failed: 1, passed: 14 of 15 tests – 494 ms

Expected :4
Actual :3
[Click to see difference](#)

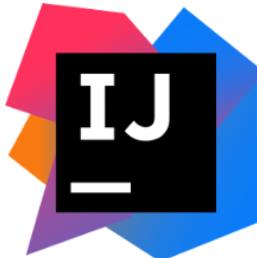
```
io.kotest.assertions.AssertionFailedError: expected:<4> but was:<3>
    at com.jicay.unittests.AdditionTest$1$2.invokeSuspend(AdditionTest.kt:32)
    at com.jicay.unittests.AdditionTest$1$2.invoke(AdditionTest.kt)
    >   at com.jicay.unittests.AdditionTest$1$2.invoke(AdditionTest.kt) <103 internal lines>
    >   at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:33)
```

Installation de l'environnement

- Installer le Java Development Kit (JDK) 21+ : [SDKMan](#) ou [OpenJDK](#)

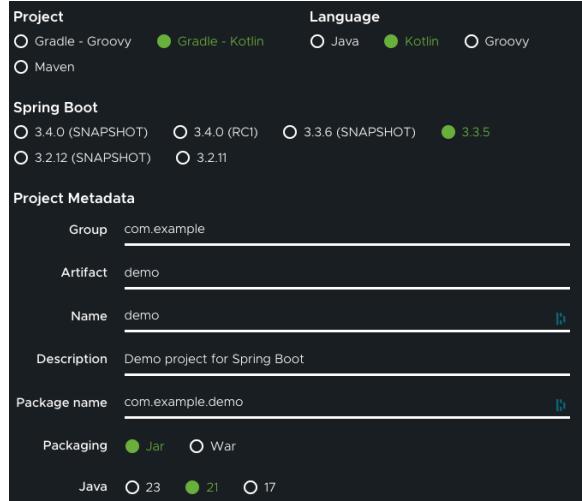


- Installer IntelliJ IDEA : [JetBrains](#)



Création du projet Spring Boot

- Créer un projet Spring Boot sur [Spring Initializr](#)

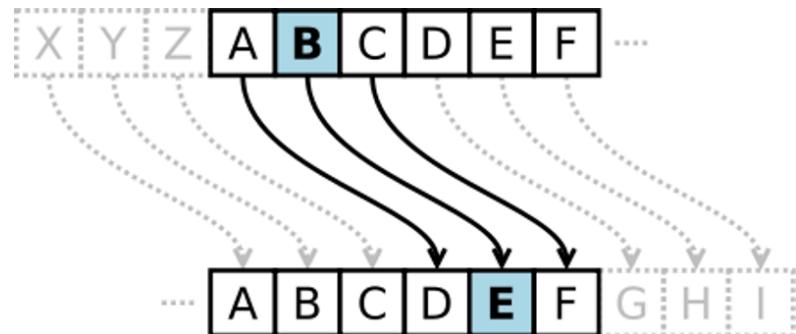


- Dans `build.gradle.kts`, ajouter les dépendances Kotest

```
dependencies {  
    implementation("org.springframework.boot:spring-boot-starter")  
    implementation("org.jetbrains.kotlin:kotlin-reflect")  
    testImplementation("org.springframework.boot:spring-boot-starter-test")  
    testImplementation("io.kotest:kotest-runner-junit5:5.9.1")  
    testImplementation("io.kotest:kotest-assertions-core:5.9.1")  
}
```

TP : algorithme de César

- En TDD, implémenter l'algorithme du chiffrement de César :
 - Entrée : un caractère `char` et un entier `key`
 - Sortie : un caractère "décalé" de la valeur de l'entier
 - Exemple : `cypher('A', 2) = 'C'`
- Quelques règles :
 - Seules les lettres majuscules sont autorisées
 - Lorsqu'on dépasse `'Z'`, on revient à `'A'`
 - Si `key > 26`, on recommence le cycle
 - Si `key < 0`, c'est une erreur



TP : étapes

Écrire un test qui appelle une méthode cipher pour 'A' et 2

Écrire le code
MINIMUM
pour que le test passe

Ajouter un second test pour
'A' et 5

Modifier le code pour que les 2 tests fonctionnent en implémentant le minimum

Améliorer le code

Ajouter un test pour une des règles, modifier le code pour que tous les tests passent, améliorer le code et recommencer

Property-based test : définitions

- Un test classique teste des exemples
- Teste des propriétés toujours valables, aussi appelées invariants
 - Propriété définies sur un ensemble de données possibles (entier positif, liste de 3 éléments, ...)
 - À chaque exécution du test, de nouvelles valeurs seront testées
- Ne remplace pas les tests basés sur les exemples, mais les complète

Property-based test : exemple de propriétés

- Cas de l'addition d'entiers :
 - Identité : $x + 0 = x$ (0 neutre de l'addition)
 - Associativité : $(x + y) + z = x + (y + z)$
 - Commutativité : $x + y = y + x$

Property-based test : mise en place

- Bibliothèque Kotest Property-based Testing

```
test("identity of addition") {
    checkAll<Int> { a ->
        // Arrange
        val op = AddOperator()

        // Act
        val res = op.add(a, 0)

        // Assert
        res shouldBe a
    }
}
```

```
test("associativity of addition") {
    checkAll<Int, Int> { a, b ->
        // Arrange
        val op = AddOperator()

        // Act
        val res1 = op.add(a, b)
        val res2 = op.add(b, a)

        // Assert
        res1 shouldBe res2
    }
}
```

```
test("commutativity of addition") {
    checkAll<Int, Int, Int> { a, b, c ->
        // Arrange
        val op = AddOperator()

        // Act
        val res1 = op.add(a, op.add(b, c))
        val res2 = op.add(op.add(a, b), c)

        // Assert
        res1 shouldBe res2
    }
}
```

TP : Property-based tests

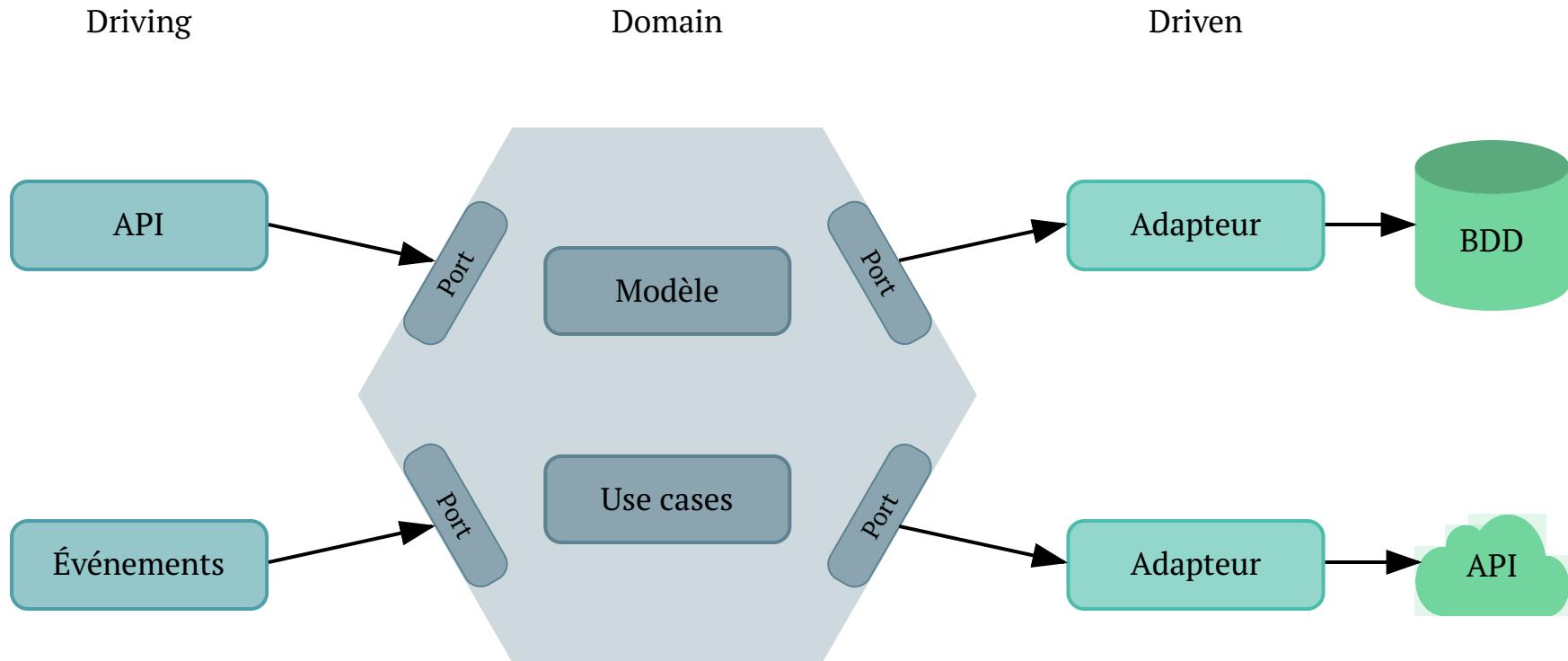
- Ajouter la dépendance

```
dependencies {  
    // ...  
    testImplementation("io.kotest:kotest-property:5.9.1")  
    // ...  
}
```

- Trouver des invariants pour l'algorithme de César et implémenter des property-based tests correspondants

Mise en place des tests dans un projet

Architecture Hexagonale



Architecture Hexagonale : Domaine

- Contient toute la partie métier de l'application
 - Modèle = les objets métiers du domaine qui contiennent le maximum de règles métiers relatives à ces objets
 - Use case = le point d'entrée dans le domaine et l'ordonnanceur pour répondre au besoin métier (appels extérieurs entre autres)
- Contient uniquement du code natif : pas de framework ni de bibliothèques externes
- Communique avec l'extérieur via l'utilisation d'interfaces, appelée ports, qui sont implémentées hors du domaine
- Développé en TDD, couvert au maximum par les tests unitaires

Architecture Hexagonale : Driving

- Partie de l'infrastructure qui s'interface avec les frameworks et la bibliothques
- Pilote et appelle le domaine : instancie les objets du modèle et appelle les méthodes des use cases
- Possède des Data Transfer Object (DTO) mappés en objets du domaine
- Couvert par les tests d'intégration, plus rarement par les tests unitaires
- Par exemple, les contrôleurs REST

Architecture Hexagonale : Driven

- Partie de l'infrastructure qui s'interface avec les frameworks et les bibliothèques
- Piloté et appelé par le domaine via des ports : utilise et retourne des objets du domaine
- Sert souvent à récupérer ou stocker des données
- Couvert par les tests d'intégration, plus rarement par les tests unitaires
- Par exemple, les bases de données, les appels à une API tierce

Test double ou comment gérer les dépendances ?

- Les classes et méthodes s'appellent entre elles
- Les tests doubles prennent la place des dépendances et servent à isoler le code de la méthode à tester
- Plusieurs types de test double, le choix dépend du test à effectuer
- Utilisation de bibliothèques

Test double : Dummy

- Objet qui ne sert qu'à combler un paramètre, utilisé lorsque les valeurs ou comportements de l'objet ne sont pas importants

```
interface User {  
    fun getName(): String  
}  
  
fun userExists(user: User?): Boolean {  
    return user != null  
}
```

```
class DummyUser : User {  
    override fun getName(): String {  
        TODO("Not yet implemented")  
    }  
  
    test("exist") {  
        // Arrange  
        val dummy = DummyUser()  
  
        // Act  
        val res = userExists(dummy)  
  
        // Assert  
        res shouldBe true  
    }  
}
```

Test double : Stub

- Objet qui retourne des valeurs prédéfinies, utilisé pour simuler un comportement ou une valeur de retour connue

```
interface User {  
    fun getName(): String  
}  
  
fun nameInfo(user: User): String {  
    return "${user.getName()} has length ${user.getName().length}"  
}
```

```
class StubUser : User {  
    override fun getName(): String {  
        return "toto"  
    }  
}  
  
test("name length") {  
    // Arrange  
    val stub = StubUser()  
  
    // Act  
    val res = nameInfo(stub)  
  
    // Assert  
    res shouldBe "toto has length 4"  
}
```

Test double : Fake

- Implémentation simplifiée de la dépendance, utilisée pour simuler un comportement plus complexe que le stub

```
interface DBDao {  
    fun findAll(): List<String>  
    fun insert(value: String)  
}  
  
class Service(private val dbDao: DBDao) {  
    fun insertAll(values: List<String>) {  
        values.forEach { dbDao.insert(it) }  
    }  
}
```

```
class FakeDbDao : DBDao {  
    private val db = mutableListOf<String>()  
  
    override fun findAll(): List<String> {  
        return db  
    }  
  
    override fun insert(value: String) {  
        db.add(value)  
    }  
}  
  
test("insert all in db") {  
    val fake = FakeDbDao()  
    val service = Service(fake)  
  
    service.insertAll(listOf("toto", "tata"))  
  
    val stored = fake.findAll()
```

Test double : Spy

- Objet qui enregistre les appels, utilisé pour vérifier les appels de méthodes

```
interface User {  
    fun getName(): String  
}  
  
fun nameInfo(user: User): String {  
    return "${user.getName()} has length ${user.getName().length}"  
}
```

```
class SpyUser : User {  
    var isNameCalled = false  
  
    override fun getName(): String {  
        isNameCalled = true  
        return "toto"  
    }  
}  
  
test("name length and name has been called") {  
    // Arrange  
    val spy = SpyUser()  
  
    // Act  
    val res = nameInfo(spy)  
  
    // Assert  
    res shouldBe "toto has length 4"  
    spy.isNameCalled shouldBe true  
}
```

Test double : Mock

- Objet qui vérifie les arguments envoyés pour simuler un comportement

```
interface User {  
    fun getName(isUpperCase: Boolean): String  
}  
  
fun nameInfo(user: User): String {  
    val userName = user.getName(true)  
    return "$userName has length ${userName.length}"  
}
```

```
class MockUser : User {  
    var isNameCalled = false  
  
    override fun getName(isUpperCase: Boolean): String {  
        isNameCalled = true  
        return when (isUpperCase) {  
            true -> "TOTO"  
            false -> "toto"  
        }  
    }  
  
    test("name length and name has been called with upper case") {  
        val mock = MockUser()  
  
        val res = nameInfo(mock)  
  
        res shouldBe "TOTO has length 4"  
        mock.isNameCalled shouldBe true  
    }  
}
```

Bibliothèque de mock

- Bibliothèque utilisée pour Kotlin : MockK

```
class MockKUserTest : FunSpec(){

    val mock = mockk<User>()

    test("name length and name has been called with upper case") {
        // Arrange
        every { mock.getName(true) } returns "TOTO"

        // Act
        val res = nameInfo(mock)

        // Assert
        res shouldBe "TOTO has length 4"
        verify(exactly = 1) { mock.getName(true) }
    }
})
```

Test double

	Dummy	Stub	Spy	Mock	Fake
Instance	✓	✓	✓	✓	✓
Contenu	✗	✓	✓	✓	✗
Vérification	✗	✗	✓	✓	✗
Contrôle des paramètres	✗	✗	✗	✓	✗
Simule l'implémentation	✗	✗	✗	✗	✓

Tests sociaux

- Teste plusieurs modules (méthodes, classes, ...) étroitement liés
- Permet de valider une logique fonctionnelle
- Autorise des refactorisations plus profondes
- À utiliser pour tester la partie purement fonctionnelle du code
- N'exclut pas les tests unitaires solitaires dans le cas de règles spécifiques complexes

CI/CD

CI = Continuous Integration

- Ensemble des commandes exécutées à chaque publication sur le repository de code
 - Exécutions des tests isolés
 - Exécutions de job de validation de qualité

CD = Continuous Deployment

- Ensemble de commandes pour déployer l'application sur différents environnements
 - Déploiements
 - Exécutions des tests nécessitant un environnement réel

GitHub Actions

- CI/CD intégrée à GitHub
- Définition des pipelines à partir de fichier YAML intégrés au projet
- Possibilité d'afficher des rapports générés au format Markdown

The screenshot shows the GitHub Actions interface. On the left, there's a sidebar with 'Summary' selected, followed by 'Jobs' (Build is green), 'Run details' (Usage and Workflow file), and a '+' button. The main area shows a workflow named 'gradle.yml' triggered by 'on: push'. A 'Build' step is shown with a green checkmark and a duration of '6m 37s'. Below this, a 'Build summary' section for 'detekt' contains a 'Metrics' table:

Metrics	Value
number of properties	4
number of functions	14
number of classes	9
number of packages	8

Couverture de code

- Mesure le pourcentage de lignes utilisées lors de l'exécution des tests
- Les rapports comprennent :
 - Le pourcentage de couverture de ligne
 - Le pourcentage de couverture de méthodes/fonctions
 - Le pourcentage de couverture de branches
- On définit souvent un niveau de couverture de code minimum sur un projet
-  Une couverture à 100% n'est pas synonyme d'un code correctement testé

JaCoCo

- Plugin gradle utilisé pour mesurer la couverture de code
- Fournit un rapport HTML détaillé
- Intégrable facile dans GitHub Action pour afficher son rapport

BookManagement

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed %	Lines	Missed %	Methods	Missed %	Classes
com.jicay.bookmanagement.infrastructure.driven.adapter	██████████	0 %	n/a	4	4	12	12	4	4	1	1	
com.jicay.bookmanagement.infrastructure.driver.web.dto	██████████	0 %	n/a	5	5	8	8	5	5	2	2	
com.jicay.bookmanagement.infrastructure.driver.web	██████████	0 %	n/a	3	3	7	7	3	3	1	1	
com.jicay.bookmanagement	████	0 %	n/a	2	2	4	4	2	2	2	2	
com.jicay.bookmanagement.infrastructure.application	████	0 %	n/a	2	2	3	3	2	2	1	1	
com.jicay.bookmanagement.domain.model	██████	78 %	██████████	50 %	3	5	1	5	1	3	0	1
com.jicay.bookmanagement.domain.usecase	██████	100 %	n/a	0	4	0	6	0	4	0	2	
Total	176 of 246	28 %	2 of 4	50 %	19	25	35	45	17	23	7	10

Coverage Report: JaCoCo		
• BookManagement		
Outcome	Value	
Code Coverage %	95.12%	
✓ Number of Lines Covered	39	
✗ Number of Lines Missed	2	
Total Number of Lines	41	

Mutation tests

- Change le code source (= mutations) et exécute les tests :
 - Si les tests passent toujours, on parle de mutant survivant -> tests insuffisants
 - Si les tests échouent, on parle de mutant tué -> tests suffisants
- De nombreuses mutations possibles :
 - Modifier la valeur d'une constante
 - Remplacer un opérateur (changer un > en < ou >= par exemple)
 - Supprimer une instruction
 - ...
- Assez gourmand en ressources et plus long

Mutation tests : cas d'usage

```
class UserMutation(private val age: Int) {  
    fun isLawfulAge(): Boolean {  
        return age >= LAWFUL_AGE  
    }  
  
    companion object {  
        const val LAWFUL_AGE = 18  
    }  
}
```

- Mutant survivant : Si on change le `>=` en `==`, les tests sont toujours OK
- Mutant tué : Si on change `LAWFUL_AGE = 18` en `LAWFUL_AGE = 17`, le second test devient KO

```
class UserMutationTest : FunSpec({  
  
    test("is lawful age") {  
        val userMutation = UserMutation(18)  
  
        val isLawfulAge = userMutation.isLawfulAge()  
  
        isLawfulAge shouldBe true  
    }  
  
    test("is under age") {  
        val userMutation = UserMutation(17)  
  
        val isLawfulAge = userMutation.isLawfulAge()  
  
        isLawfulAge shouldBe false  
    }  
})
```

Mutation tests : PI Test

- Plugin gradle utilisé pour exécuter les tests de mutation
- Permet de paramétriser quels types de mutation on souhaite pour notre projet
- Fournit un rapport HTML détaillé

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
6	100% <div style="background-color: #a9f5d0; border: 1px solid #a9f5d0; padding: 2px;">36/36</div>	100% <div style="background-color: #a9f5d0; border: 1px solid #a9f5d0; padding: 2px;">11/11</div>	100% <div style="background-color: #a9f5d0; border: 1px solid #a9f5d0; padding: 2px;">11/11</div>

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
com.jicay.bookmanagement.domain.model	1	100% <div style="background-color: #a9f5d0; border: 1px solid #a9f5d0; padding: 2px;">1/1</div>	100% <div style="background-color: #a9f5d0; border: 1px solid #a9f5d0; padding: 2px;">2/2</div>	100% <div style="background-color: #a9f5d0; border: 1px solid #a9f5d0; padding: 2px;">2/2</div>
com.jicay.bookmanagement.domain.usecase	2	100% <div style="background-color: #a9f5d0; border: 1px solid #a9f5d0; padding: 2px;">8/8</div>	100% <div style="background-color: #a9f5d0; border: 1px solid #a9f5d0; padding: 2px;">3/3</div>	100% <div style="background-color: #a9f5d0; border: 1px solid #a9f5d0; padding: 2px;">3/3</div>
com.jicay.bookmanagement.infrastructure.driven.adapter	1	100% <div style="background-color: #a9f5d0; border: 1px solid #a9f5d0; padding: 2px;">12/12</div>	100% <div style="background-color: #a9f5d0; border: 1px solid #a9f5d0; padding: 2px;">2/2</div>	100% <div style="background-color: #a9f5d0; border: 1px solid #a9f5d0; padding: 2px;">2/2</div>
com.jicay.bookmanagement.infrastructure.driver.web	1	100% <div style="background-color: #a9f5d0; border: 1px solid #a9f5d0; padding: 2px;">11/11</div>	100% <div style="background-color: #a9f5d0; border: 1px solid #a9f5d0; padding: 2px;">2/2</div>	100% <div style="background-color: #a9f5d0; border: 1px solid #a9f5d0; padding: 2px;">2/2</div>
com.jicay.bookmanagement.infrastructure.driver.web.dto	1	100% <div style="background-color: #a9f5d0; border: 1px solid #a9f5d0; padding: 2px;">4/4</div>	100% <div style="background-color: #a9f5d0; border: 1px solid #a9f5d0; padding: 2px;">2/2</div>	100% <div style="background-color: #a9f5d0; border: 1px solid #a9f5d0; padding: 2px;">2/2</div>

TP : Gestion de livre : Partie métier (1/6)

- Application de gestion de liste de livres
- Un livre est caractérisé par son titre et son auteur
- Actions possibles :
 - Ajouter un livre
 - Lister tous les livres par ordre alphabétique

TP : Gestion de livre : Partie métier (2/6)

- En TDD, implémentation du domaine, c'est-à-dire toutes les règles métiers :
 - Création du modèle : la classe `Book`
 - Création du use case : une classe contenant les méthodes de création et de liste
 - Création des ports : une interface qui écrira en base de données par la suite
- Implémenter les différentes règles :
 - Ajout d'un livre
 - Un livre doit avoir un titre et un auteur non vide
 - La récupération de tous les livres doit retourner la liste complète triée par ordre alphabétique selon le titre du livre
- Utiliser les outils et exemples présentés précédemment (Kotest, MockK)

TP : Gestion de livre : Partie métier (3/6)

- Définir des tests de propriétés
- Exemple de propriété :
 - La liste des livres retournés contient tous les éléments de la liste stockée

TP : Gestion de livre : Partie métier (4/6)

- Définir la CI/CD pour GitHub Actions

- Définir les étapes suivantes :
 - Build de l'application
 - Lancement des tests
 - Publier les résultats des tests dans la pipeline (plugin)

TP : Gestion de livre : Partie métier (5/6)

- Mise en place des tests de couverture de code avec JaCoCo
- Ajouter l'étape dans la CI/CD et publier le rapport dans le pipeline

TP : Gestion de livre : Partie métier (6/6)

- Mise en place des tests de mutation avec PI Test
- Ajout de l'étape dans la CI/CD

Tests d'intégration et de composants

Test d'intégration

- Test automatisé écrit par le développeur
- Valide unitairement l'intégration et l'utilisation des bibliothèques et frameworks
- Doit valider que la bibliothèque ou le framework est utilisé correctement et non la bibliothèque elle-même

Fonctionnement

- Contrairement aux tests unitaires, l'application est démarrée
- On ne teste que la classe s'interfaisant avec la bibliothèque ou le framework :
 - On mocke les autres classes dont on dépend
 - On appelle directement la classe que l'on souhaite

Comment les mettre en place (1/2)

- Ajout d'un dossier `testIntegration` dans le dossier `src` du projet (même niveau que `test`) contenant deux dossiers `kotlin` et `resources`
- Ajout des informations dans `build.gradle.kts` :
 - Création d'une testing suite dédiée : indique quel dossier contient les tests

```
testing {
    suites {
        val testIntegration by registering(JvmTestSuite::class) {
            sources {
                kotlin {
                    setSrcDirs(listOf("src/testIntegration/kotlin"))
                }
                compileClasspath += sourceSets.main.get().output
                runtimeClasspath += sourceSets.main.get().output
            }
        }
    }
}
```

Comment les mettre en place (2/2)

- Crédit de la configuration dédiée : permet d'ajouter les dépendances spécifiques

```
val testIntegrationImplementation: Configuration by configurations.getting {  
    extendsFrom(configurations.implementation.get())  
}
```

- Ajouter les dépendances

```
testIntegrationImplementation("io.mockk:mockk:1.13.8")  
testIntegrationImplementation("io.kotest:kotest-assertions-core:5.9.1")  
testIntegrationImplementation("io.kotest:kotest-runner-junit5:5.9.1")  
testIntegrationImplementation("com.ninja-squad:springmockk:4.0.2")  
testIntegrationImplementation("org.springframework.boot:spring-boot-starter-test") {  
    exclude(module = "mockito-core")  
}
```

Spring et l'injection de dépendances

- L'injection de dépendances permet de déléguer au framework la création et l'injection des dépendances d'une classe
- En Spring, un objet créé et injectable s'appelle un **Bean**
- L'injection se fait selon le type et/ou le nom de la dépendance

Spring : Création d'un bean

- Crédit par annotation sur une classe : `@Component` , `@Service`

```
@Service  
class BeanService() {  
    fun a() {}  
}
```

- Crédit par annotation sur une fonction : `@Bean`

```
@Bean  
fun beanService(): BeanService {  
    return BeanService()  
}
```

Spring : Injection d'un bean

- Déclaration dans le constructeur d'un autre bean (ou de la fonction de déclaration)

```
@Service
class DependantObject(private val beanService: BeanService) {}

// Équivalent à

@Bean
fun dependantObject(beanService: BeanService): DependantObject {
    return DependantObject(beanService)
}
```

- Autowire manuel

```
@Service
class DependantObject() {
    @Autowired
    private lateinit var beanService: BeanService
}
```

Mocker les dépendances

- Il faut mockr les dépendances des classes que l'on utilise
- Utilisation de la notion de mock de bean permettant de substituer l'implémentation d'un bean Spring par un mock.
- Utilisation de la librairie `springmockk` pour mocker les beans avec l'annotation `@MockkBean`

Créer un controller REST

```
@RestController
@RequestMapping("/demo")
class DemoController {
    // GET localhost:8080/demo/123?language=fr returns {"a":"demo 123 fr","b":123}
    // GET localhost:8080/demo/123 returns {"a":"demo 123 null","b":123}
    @GetMapping("/{id}")
    fun getDemo(
        @PathVariable("id") id: Int,
        @RequestParam(value = "language", required = false) language: String?
    ): DemoExampleDTO {
        return DemoExampleDTO("demo $id ${language}", id)
    }

    // POST localhost:8080/demo with body {"a":"demo","b":123} returns 201 Created with no content
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    fun createDemo(@RequestBody demo: DemoExampleDTO) {
        // Do something
    }
}

data class DemoExampleDTO(val a: String, val b: Int)
```

Tester un appel REST

- Il faut simuler une commande REST avec :
 - Le bon verbe
 - La bonne URL
 - Les bons paramètres
 - Les bons headers
- Puis valider :
 - Le code de retour
 - Le contenu
- Mocker l'appel au domaine (avec `MockkBean`)

Tester un appel REST : MockMVC

- Ajout de l'annotation `@WebMvcTest` sur la classe de test et récupération de l'objet `MockMVC`

```
mockMvc
    .get("/url")
    .andExpect {
        status { isOk() }
        content { content { APPLICATION_JSON } }
        content {
            json(
                """
                    [
                        {
                            "name": "A"
                        },
                        {
                            "name": "B"
                        }
                    ]
                """
            ).trimIndent()
        }
    }
```

S'interfacer avec une BDD (1/2)

- Dépendances dans `build.gradle.kts`

```
implementation("org.springframework.boot:spring-boot-starter-jdbc")
implementation("org.postgresql:postgresql")
```

- Paramétrage dans `application.yaml`

```
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/postgres
    username: postgres
    password: password
    driver-class-name: org.postgresql.Driver
```

S'interfacer avec une BDD (2/2)

- Utilisation du JDBC Template pour écrire du SQL

```
@Service
class DemoDAO(private val namedParameterJdbcTemplate: NamedParameterJdbcTemplate) {
    fun getDemo(): List<DemoExample> {
        return namedParameterJdbcTemplate
            .query("SELECT * FROM demo", MapSqlParameterSource()) { rs, _ ->
                DemoExample(
                    a = rs.getString("a")
                )
            }
    }

    fun createDemo(demo: DemoEntity) {
        namedParameterJdbcTemplate
            .update(
                "INSERT INTO demo (a) values (:a)", mapOf(
                    "a" to demo.a,
                )
            )
    }
}

data class DemoExample(val a: String)
}
```

Tester une intégration en BDD

- Permet de tester les requêtes SQL sur une base de données
- Deux choix principaux :
 - Utiliser une BDD embarquée (H2, HSQLDB)
 - Utiliser la BDD réelle et réussir à l'embarquer dans nos tests
- 3 étapes pour chaque tests :
 - Préparation de la BDD (création de données, ...)
 - Appel de la méthode à tester
 - Vérification du résultat et du contenu de la BDD après l'appel

Testcontainers (1/2)

- Testcontainers est une bibliothèque permettant de démarrer un container Docker pour notre test
- Ajout des dépendances :

```
testIntegrationImplementation("org.testcontainers:postgresql:1.19.1")
testIntegrationImplementation("org.testcontainers:jdbc-test:1.12.0")
testIntegrationImplementation("org.testcontainers:testcontainers:1.19.1")
testIntegrationImplementation("io.kotest.extensions:kotest-extensions-testcontainers:2.0.2")
```

Testcontainers (2/2)

```
@SpringBootTest
@ActiveProfiles("testIntegration")
class BookDAOIT() : FunSpec() {
    init {
        extension(SpringExtension)

        afterSpec {
            container.stop()
        }
    }

    companion object {
        private val container = PostgreSQLContainer<Nothing>("postgres:13-alpine")

        init {
            container.start()
            System.setProperty("spring.datasource.url", container.jdbcUrl)
            System.setProperty("spring.datasource.username", container.username)
            System.setProperty("spring.datasource.password", container.password)
        }
    }
}
```

Liquibase

- Liquibase est un outil de gestion de version et de migrations de base de données
- À chaque fois que l'application est démarée, on vérifie si la base de données est dans la bonne version, si non, on exécute les scripts manquants
- Fonctionne à partir d'un fichier listant plusieurs changelogs à appliquer. Chaque changelog est décrit dans un fichier contenant toutes les actions à effectuer pour la migration
- Outil en XML simple à mettre en place et compatible avec un très grand nombre de BDD

Intégration de Liquibase (1/2)

- Ajout de la dépendance dans `build.gradle.kts`

```
implementation("org.liquibase:liquibase-core")
```

- Ajout de la clé de configuration dans `application.yaml` pour indiquer le chemin du fichier de changelog parent

```
spring:  
    liquibase:  
        change-log: classpath:/db/changelog.xml
```

- Création du fichier `db/changelog.xml` parent dans le dossier `src/main/resources/db`

```
<?xml version="1.0" encoding="utf-8"?>  
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog"  
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
                      xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog http://www.liquibase.org/xml/ns/dbchangelog.xsd">  
    <include file="/db/changelogs/20231009000500_initial_schema.xml"/>  
</databaseChangeLog>
```

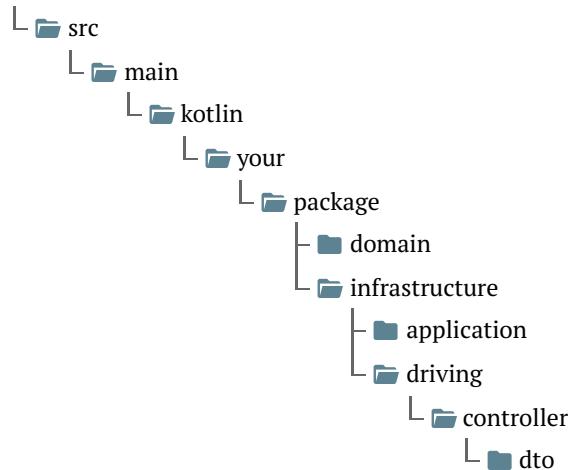
Intégration de Liquibase (2/2)

- Création du fichier de changelog dans le dossier `src/main/resources/db/changelogs`

```
<?xml version="1.0" encoding="utf-8"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog http://www.liquibase.org/xml/ns/dbchangelog.xsd">
    <changeSet id="20231009000500" author="jicay">
        <preConditions onFail="MARK_RAN">
            <not>
                <tableExists tableName="demo"/>
            </not>
        </preConditions>
        <createTable tableName="demo">
            <column name="id" type="int">
                <constraints nullable="false"/>
            </column>
            <column name="a" type="varchar">
                <constraints nullable="false"/>
            </column>
        </createTable>
        <addPrimaryKey columnNames="id" tableName="demo"/>
        <addAutoIncrement tableName="demo" columnName="id"/>
    </changeSet>
</databaseChangeLog>
```

TP : Gestion de livre : Partie web (1/3)

- Ajouter les dossiers manquants de l'arborescences :



- Ajout de la déclaration de la suite de test dans `build.gradle.kts`

TP : Gestion de livre : Partie web (2/3)

- Création de la partie driving de l'architecture :
 - Création du DTO -> nouvelle classe `BookDTO` (package `dto`)
 - Création d'un controller -> nouvelle classe `BookController` (package `controller`)
 - Deux routes :
 - `GET /books` : retourne la liste des livres
 - `POST /books` : crée un livre
- Appel du domaine :
 - Déclaration des objets du domaine en tant que bean Spring (package `application`)

```
@Configuration
class UseCasesConfiguration {
    @Bean
    fun myUseCase(dependency: MyDependency): MyUseCase {
        return MyUseCase(dependency)
    }
}
```

Tests non-fonctionnels

Tests End-To-End

Merci de votre attention