**TP 1 : Identifier les Phases de Tests dans un Projet Logiciel**

**Objectif**

L'objectif de ce TP est d'identifier et de comprendre les différentes phases de tests au sein du cycle de vie d'un projet logiciel. Vous allez examiner un projet logiciel fictif, décomposer ses étapes de développement, et déterminer quelles phases de tests sont impliquées à chaque étape.

**Contexte**

Vous êtes membre de l'équipe de développement d'une application de gestion de bibliothèque. Le projet est en cours et plusieurs versions ont déjà été déployées. Vous devez analyser le projet et documenter les phases de tests mises en place pour assurer la qualité du logiciel.

**Instructions**

1. **Étude du Contexte du Projet** (5 minutes)
   * Lisez la description du projet et prenez connaissance des différentes étapes de son cycle de vie.
   * Notez les fonctionnalités principales et les composants critiques du système.
2. **Identification des Phases de Tests** (10 minutes)
   * Identifiez les phases de tests suivantes dans le cycle de vie du développement logiciel :
     + Tests unitaires
     + Tests d'intégration
     + Tests de système
     + Tests d'acceptation
   * Pour chaque phase, déterminez à quelle étape du développement elle est effectuée :
     + Conception
     + Développement
     + Tests
     + Déploiement
     + Maintenance
3. **Analyse des Pratiques de Tests** (10 minutes)
   * Décrivez les types de tests effectués à chaque phase.
   * Expliquez les outils et méthodes utilisés pour chaque type de test (par exemple, JUnit pour les tests unitaires, Selenium pour les tests d'acceptation).
   * Discutez de l'importance de chaque phase de tests et de son impact sur la qualité du logiciel.
4. **Rédaction du Rapport** (5 minutes)
   * Rédigez un rapport synthétique de votre analyse.
   * Incluez un tableau récapitulatif des phases de tests identifiées, des outils utilisés, et des impacts sur le projet.

**Livrables**

* Un rapport d'analyse (1 page maximum) avec :
  + Une description brève du projet et de ses fonctionnalités principales.
  + Un tableau récapitulatif des phases de tests identifiées et de leurs caractéristiques.
  + Une discussion sur l'importance de chaque phase de tests.

**Exemple de Tableau Récapitulatif**

| **Phase de Développement** | **Phase de Test** | **Types de Tests** | **Outils Utilisés** | **Impact sur la Qualité** |
| --- | --- | --- | --- | --- |
| Conception |  |  |  |  |
| Développement |  |  |  |  |
| Tests |  |  |  |  |
| Tests |  |  |  |  |
| Déploiement |  |  |  |  |
| Maintenance |  |  |  |  |

**Ressources**

* Documentation du projet logiciel (fourni par l'instructeur)

**TP 2 :** **Classification des Tests Existant dans les Différentes Catégories**

**Objectif**

L'objectif de ce TP est de vous familiariser avec les différentes catégories de tests et de pratiquer la classification des tests existants dans ces catégories. Vous allez examiner une liste de tests donnée et les classer en fonction de leur nature et de leur objectif.

**Instructions**

1. **Analyse de la Liste de Tests** (10 minutes)
   * Étudiez la liste de tests fournie dans le document.
   * Identifiez les caractéristiques et les objectifs de chaque test.
2. **Classification des Tests** (15 minutes)
   * Classez chaque test dans l'une des catégories suivantes :
     + Tests Unitaires
     + Tests d'Intégration
     + Tests de Système
     + Tests d'Acceptation
     + Tests de Régression
3. **Justification de la Classification** (5 minutes)
   * Pour chaque test, justifiez votre choix en expliquant pourquoi vous l'avez classé dans la catégorie sélectionnée.
   * Utilisez des critères tels que la portée, le niveau d'abstraction, et l'objectif du test pour étayer votre classification.

**Livrables**

* Un tableau récapitulatif contenant la liste des tests classés dans les différentes catégories, accompagnés d'une justification pour chaque classification.

**Exemple de Tableau Récapitulatif**

| **Test** | **Catégorie** | **Justification** |
| --- | --- | --- |
| testCalculateTotal() | Tests Unitaires | Cette méthode teste une fonction spécifique du code source, sans dépendances externes. |
| testLoginSuccess() | Tests d'Acceptation | Ce test valide le scénario de connexion réussie pour un utilisateur. |
| testIntegration() | Tests d'Intégration | Ce test évalue l'interaction entre plusieurs modules ou composants de l'application. |
| testUICompatibility() | Tests de Système | Ce test vérifie la compatibilité de l'interface utilisateur avec différents navigateurs et appareils. |
| testRegression() | Tests de Régression | Ce test vérifie qu'une modification récente n'a pas affecté le comportement existant de l'application. |

**Ressources**

Voici la liste des tests à classifier :

1. **testAddBook()**
2. **testUpdateMemberInfo()**
3. **testCalculateLateFees()**
4. **testBookBorrowingFlow()**
5. **testReturnBookFlow()**
6. **testSearchBooks()**
7. **testGenerateReports()**
8. **testMemberRegistration()**
9. **testBookDeletion()**
10. **testPreviousFeaturesAfterUpdate()**

Ces tests couvrent diverses fonctionnalités de l'application de gestion de bibliothèque, telles que la gestion des livres, des membres, des emprunts, ainsi que la génération de rapports et la vérification de l'intégrité des fonctionnalités après des mises à jour.

**TP 3 : Implémenter une Architecture de Test Simple avec des Mocks**

**Objectif**

L'objectif de ce TP est de vous familiariser avec l'utilisation des mocks dans une architecture de test. Vous allez implémenter une architecture de test simple pour une application de gestion de bibliothèque en utilisant des mocks pour isoler les composants testés.

**Instructions**

1. **Préparation de l'Environnement** (5 minutes)
   * Assurez-vous que votre environnement de développement est prêt avec les outils nécessaires (JUnit, Mockito).
   * Créez un projet Java avec une structure de répertoires appropriée pour les tests (src/main/java pour le code source et src/test/java pour les tests).
2. **Implémentation de la Classe à Tester** (10 minutes)
   * Implémentez une classe LibraryService qui a les méthodes suivantes :
     + addBook(Book book): Ajoute un livre à la bibliothèque.
     + borrowBook(String bookId, String userId): Permet à un utilisateur d'emprunter un livre.

public class LibraryService {

private BookRepository bookRepository;

private UserRepository userRepository;

public LibraryService(BookRepository bookRepository, UserRepository userRepository) {

this.bookRepository = bookRepository;

this.userRepository = userRepository;

}

public void addBook(Book book) {

bookRepository.save(book);

}

public void borrowBook(String bookId, String userId) {

Book book = bookRepository.findById(bookId);

User user = userRepository.findById(userId);

if (book != null && user != null && book.isAvailable()) {

book.setBorrower(user);

book.setAvailable(false);

bookRepository.save(book);

}

}

}

1. **Implémentation des Tests avec Mocks** (15 minutes)
   * Utilisez Mockito pour créer des mocks des dépendances BookRepository et UserRepository.
   * Implémentez des tests unitaires pour les méthodes de LibraryService.

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

public class LibraryServiceTest {

@Mock

private BookRepository bookRepository;

@Mock

private UserRepository userRepository;

@InjectMocks

private LibraryService libraryService;

@BeforeEach

public void setUp() {

MockitoAnnotations.openMocks(this);

}

@Test

public void testAddBook() {

Book book = new Book("1", "Effective Java", true);

libraryService.addBook(book);

verify(bookRepository, times(1)).save(book);

}

@Test

public void testBorrowBook() {

Book book = new Book("1", "Effective Java", true);

User user = new User("1", "John Doe");

when(bookRepository.findById("1")).thenReturn(book);

when(userRepository.findById("1")).thenReturn(user);

libraryService.borrowBook("1", "1");

assertFalse(book.isAvailable());

assertEquals(user, book.getBorrower());

verify(bookRepository, times(1)).save(book);

}

}

**Livrables**

* Code source de la classe LibraryService.
* Code source des tests unitaires de LibraryService.

**Ressources**

* IDE Java (Eclipse, IntelliJ, etc.)
* Bibliothèques JUnit et Mockito ajoutées au projet (via Maven ou Gradle).

**TP 4 : Écrire des Tests Unitaires Basiques en Utilisant TDD**

**Objectif**

L'objectif de ce TP est de pratiquer l'écriture de tests unitaires basiques en utilisant la méthodologie du Test Driven Development (TDD). Vous allez implémenter une classe simple en suivant les étapes de TDD : écrire un test, implémenter le code pour faire passer le test, puis refactorer si nécessaire.

**Instructions**

1. **Compréhension des Exigences** (5 minutes)
   * Vous allez implémenter une classe Calculator avec les fonctionnalités suivantes :
     + Addition de deux nombres (add).
     + Soustraction de deux nombres (subtract).
     + Multiplication de deux nombres (multiply).
     + Division de deux nombres (divide).
2. **Écrire les Tests Unitaires** (10 minutes)
   * Créez une classe de test CalculatorTest.
   * Écrivez un test pour chaque méthode (add, subtract, multiply, divide) en suivant les étapes de TDD :
     + Écrire un test qui échoue.
     + Implémenter juste assez de code pour faire passer le test.
     + Refactorer le code si nécessaire.
3. **Implémentation de la Classe Calculator** (10 minutes)
   * Implémentez la classe Calculator pour faire passer les tests écrits.
4. **Vérification et Refactorisation** (5 minutes)
   * Vérifiez que tous les tests passent.
   * Refactorez le code de la classe Calculator si nécessaire.

**Livrables**

* Code source de la classe Calculator.
* Code source des tests unitaires de CalculatorTest.

**Exemple de Classe de Test et de Classe à Implémenter**

**1. Classe de Test : CalculatorTest**

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

import org.junit.jupiter.api.Test;

public class CalculatorTest {

@Test

public void testAdd() {

Calculator calculator = new Calculator();

int result = calculator.add(2, 3);

assertEquals(5, result);

}

@Test

public void testSubtract() {

Calculator calculator = new Calculator();

int result = calculator.subtract(5, 3);

assertEquals(2, result);

}

@Test

public void testMultiply() {

Calculator calculator = new Calculator();

int result = calculator.multiply(2, 3);

assertEquals(6, result);

}

@Test

public void testDivide() {

Calculator calculator = new Calculator();

int result = calculator.divide(6, 3);

assertEquals(2, result);

}

}

**2. Classe à Implémenter : Calculator**

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int subtract(int a, int b) {

return a - b;

}

public int multiply(int a, int b) {

return a \* b;

}

public int divide(int a, int b) {

if (b == 0) {

throw new IllegalArgumentException("Division by zero is not allowed.");

}

return a / b;

}

}

**Étapes Détaillées du TDD**

1. **Écrire un Test Qui Échoue**
   * Pour chaque méthode (add, subtract, multiply, divide), commencez par écrire un test unitaire dans CalculatorTest.
   * Initialement, la classe Calculator n'aura pas les méthodes nécessaires, donc les tests échoueront.
2. **Implémenter le Code pour Faire Passer le Test**
   * Ajoutez les méthodes dans la classe Calculator une par une pour faire passer les tests.
   * Chaque méthode doit juste faire assez pour que le test passe.
3. **Refactorer le Code**
   * Une fois que tous les tests passent, vérifiez si le code peut être amélioré sans casser les tests.

**Ressources**

* IDE Java (Eclipse, IntelliJ, etc.)
* Bibliothèque JUnit ajoutée au projet (via Maven ou Gradle).

**TP 5 : Configuration d’un Projet Maven avec JUnit et Mockito, Écriture des Premiers Tests pour une Gestion de Bibliothèque**

**Objectif**

L'objectif de ce TP est de configurer un projet Maven pour une application de gestion de bibliothèque en ajoutant les dépendances nécessaires pour JUnit et Mockito, puis d'écrire des tests unitaires basiques pour une classe de service en utilisant ces frameworks.

**Instructions**

1. **Création du Projet Maven** (10 minutes)
   * Créez un nouveau projet Maven dans votre IDE (Eclipse, IntelliJ, etc.).
   * Configurez le fichier pom.xml pour inclure les dépendances de JUnit et Mockito.
2. **Configuration du fichier pom.xml** (5 minutes)
   * Ajoutez les dépendances suivantes au fichier pom.xml :
     + JUnit
     + Mockito
3. **Création des Classes à Tester** (10 minutes)
   * Créez une classe Book qui représente un livre avec les attributs id, title, et author.
   * Créez une interface BookRepository avec une méthode findById(int id).
   * Créez une classe LibraryService qui utilise BookRepository pour récupérer des livres.
4. **Écriture des Tests Unitaires** (20 minutes)
   * Créez une classe de test LibraryServiceTest.
   * Écrivez des tests unitaires en utilisant JUnit pour vérifier le comportement de la méthode getBookById de LibraryService.
   * Utilisez Mockito pour simuler le comportement de BookRepository.

**Détails du Projet**

1. **Créer le Projet Maven**
   * Utilisez votre IDE pour créer un nouveau projet Maven.
2. **Configuration du fichier pom.xml**

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>library-management</artifactId>

<version>1.0-SNAPSHOT</version>

<properties>

<maven.compiler.source>17</maven.compiler.source>

<maven.compiler.target>17</maven.compiler.target>

<junit.jupiter.version>5.8.1</junit.jupiter.version>

<mockito.version>4.0.0</mockito.version>

</properties>

<dependencies>

<!-- JUnit 5 Dependencies -->

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-api</artifactId>

<version>${junit.jupiter.version}</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>${junit.jupiter.version}</version>

<scope>test</scope>

</dependency>

<!-- Mockito Dependencies -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>${mockito.version}</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-junit-jupiter</artifactId>

<version>${mockito.version}</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

1. **Création des Classes Book, BookRepository, et LibraryService**

public class Book {

private int id;

private String title;

private String author;

// Constructor, getters, and setters

public Book(int id, String title, String author) {

this.id = id;

this.title = title;

this.author = author;

}

public int getId() {

return id;

}

public String getTitle() {

return title;

}

public String getAuthor() {

return author;

}

}

public interface BookRepository {

Book findById(int id);

}

public class LibraryService {

private BookRepository bookRepository;

public LibraryService(BookRepository bookRepository) {

this.bookRepository = bookRepository;

}

public Book getBookById(int id) {

return bookRepository.findById(id);

}

}

1. **Écriture des Tests Unitaires avec JUnit et Mockito**

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import static org.mockito.\*;

public class LibraryServiceTest {

private LibraryService libraryService;

@Mock

private BookRepository bookRepository;

@BeforeEach

public void setUp() {

MockitoAnnotations.initMocks(this);

libraryService = new LibraryService(bookRepository);

}

@Test

public void testGetBookById() {

Book book = new Book(1, "1984", "George Orwell");

when(bookRepository.findById(1)).thenReturn(book);

Book result = libraryService.getBookById(1);

assertNotNull(result);

assertEquals(1, result.getId());

assertEquals("1984", result.getTitle());

assertEquals("George Orwell", result.getAuthor());

verify(bookRepository).findById(1);

}

@Test

public void testGetBookById\_NotFound() {

when(bookRepository.findById(2)).thenReturn(null);

Book result = libraryService.getBookById(2);

assertNull(result);

verify(bookRepository).findById(2);

}

}

**Explications**

1. **Test de getBookById**
   * Test vérifiant que la méthode retourne bien un livre avec les propriétés attendues.
   * Utilisation de Mockito pour simuler BookRepository et vérifier que la méthode findById est appelée avec le bon argument.
2. **Test de getBookById quand le livre n'est pas trouvé**
   * Test vérifiant que la méthode retourne null quand le livre n'est pas trouvé.
   * Utilisation de Mockito pour simuler BookRepository et vérifier que la méthode findById est appelée avec le bon argument.

**TP 6 : Réorganisation d’un Ensemble de Tests Existants selon les Bonnes Pratiques**

**Objectif**

L'objectif de ce TP est d'améliorer la qualité et la maintenabilité d'un ensemble de tests existants en appliquant les bonnes pratiques de structuration et d'organisation des tests. Vous devrez réorganiser les tests pour qu'ils soient plus clairs, plus concis, et suivent les standards de l'industrie.

**Contexte**

Vous travaillez sur un projet de gestion de bibliothèque. Le projet contient déjà plusieurs tests unitaires, mais ceux-ci sont mal organisés et difficiles à maintenir. Votre tâche consiste à réorganiser ces tests en suivant les bonnes pratiques.

**Étapes à Suivre**

1. **Analyse des Tests Existants**
   * Ouvrez les fichiers de test fournis.
   * Analysez la structure actuelle des tests.
2. **Identification des Problèmes**
   * Identifiez les problèmes de structure et d'organisation dans les tests existants. Notez les problèmes tels que :
     + Duplicated Code (Code dupliqué)
     + Long Methods (Méthodes longues)
     + Poor Naming Conventions (Mauvaises conventions de nommage)
     + Lack of Proper Assertions (Manque d'assertions appropriées)
     + Inconsistent Setup and Teardown (Configuration et démontage incohérents)
3. **Réorganisation des Tests**
   * Réorganisez les tests en suivant les bonnes pratiques. Assurez-vous d'inclure :
     + Des noms de méthodes de test descriptifs et cohérents
     + La suppression des duplications de code
     + La simplification des méthodes de test longues
     + L'utilisation appropriée de Setup et Teardown pour initialiser et nettoyer les ressources nécessaires pour les tests
4. **Validation**
   * Exécutez les tests pour vous assurer qu'ils passent toujours après la réorganisation.
   * Validez que les tests sont plus faciles à comprendre et à maintenir.

**Tests Existant à Réorganiser (Exemple)**

**Avant Réorganisation :**

public class LibraryServiceTest {

private LibraryService libraryService;

private BookRepository bookRepository;

@BeforeEach

public void setUp() {

bookRepository = mock(BookRepository.class);

libraryService = new LibraryService(bookRepository);

}

@Test

public void testBookFound() {

Book book = new Book(1, "1984", "George Orwell");

when(bookRepository.findById(1)).thenReturn(book);

Book result = libraryService.getBookById(1);

assertNotNull(result);

assertEquals("1984", result.getTitle());

assertEquals("George Orwell", result.getAuthor());

}

@Test

public void testBookNotFound() {

when(bookRepository.findById(2)).thenReturn(null);

Book result = libraryService.getBookById(2);

assertNull(result);

}

// Duplicated setup code

@Test

public void testBookFoundAgain() {

Book book = new Book(1, "1984", "George Orwell");

when(bookRepository.findById(1)).thenReturn(book);

Book result = libraryService.getBookById(1);

assertNotNull(result);

assertEquals("1984", result.getTitle());

assertEquals("George Orwell", result.getAuthor());

}

}

**Livrables**

* Une classe de test réorganisée, suivant les bonnes pratiques.
* Un rapport expliquant les changements effectués et les raisons de ces changements.

**Correction Détaillée**

**Après Réorganisation :**

public class LibraryServiceTest {

private LibraryService libraryService;

private BookRepository bookRepository;

@BeforeEach

public void setUp() {

bookRepository = mock(BookRepository.class);

libraryService = new LibraryService(bookRepository);

}

@Test

public void testGetBookById\_BookFound() {

Book book = new Book(1, "1984", "George Orwell");

when(bookRepository.findById(1)).thenReturn(book);

Book result = libraryService.getBookById(1);

assertNotNull(result);

assertEquals("1984", result.getTitle());

assertEquals("George Orwell", result.getAuthor());

verify(bookRepository).findById(1);

}

@Test

public void testGetBookById\_BookNotFound() {

when(bookRepository.findById(2)).thenReturn(null);

Book result = libraryService.getBookById(2);

assertNull(result);

verify(bookRepository).findById(2);

}

}

**Changements Apportés et Raisons**

1. **Suppression du Code Dupliqué**
   * Le code dupliqué dans testBookFound et testBookFoundAgain a été fusionné en un seul test testGetBookById\_BookFound.
2. **Noms de Méthodes Descriptifs**
   * Les noms des méthodes de test ont été changés pour être plus descriptifs (testBookFound -> testGetBookById\_BookFound et testBookNotFound -> testGetBookById\_BookNotFound).
3. **Vérification d'Appels Mock**
   * Ajout de la vérification des appels aux mocks (verify(bookRepository).findById(1) et verify(bookRepository).findById(2)), assurant que les interactions avec les mocks sont comme attendu.
4. **Assertions Complètes**
   * Les assertions sont complètes et valident toutes les propriétés pertinentes de l'objet retourné (assertEquals("1984", result.getTitle()); et assertEquals("George Orwell", result.getAuthor());).

En suivant cette approche, les tests sont désormais mieux structurés, plus clairs, et plus faciles à maintenir, répondant aux bonnes pratiques de tests unitaires.

**TP 7 : Implémentation d’une Fonctionnalité Simple en Utilisant le Cycle TDD**

**Objectif**

L'objectif de ce TP est de mettre en pratique le développement piloté par les tests (TDD) en implémentant une fonctionnalité simple pour un projet de gestion de bibliothèque. Vous suivrez les étapes du cycle TDD : écrire un test qui échoue, implémenter la fonctionnalité pour faire passer le test, et refactoriser le code.

**Contexte**

Vous travaillez sur un projet de gestion de bibliothèque. Une nouvelle fonctionnalité doit être ajoutée : la possibilité de rechercher des livres par titre. Vous allez implémenter cette fonctionnalité en utilisant le cycle TDD.

**Fonctionnalité à Implémenter**

Ajouter une méthode à la classe LibraryService qui permet de rechercher des livres par leur titre. La méthode doit retourner une liste de livres dont le titre contient la chaîne de caractères recherchée (recherche partielle).

**Étapes à Suivre**

1. **Écrire un Test qui Échoue**
   * Créez un test dans LibraryServiceTest pour vérifier que la recherche par titre retourne les résultats attendus.
   * Le test doit vérifier que la méthode retourne une liste contenant les livres correspondants.
2. **Implémenter la Fonctionnalité**
   * Ajoutez la méthode searchBooksByTitle dans LibraryService.
   * Faites en sorte que le test passe en implémentant la logique de recherche.
3. **Refactoriser le Code**
   * Améliorez la qualité du code si nécessaire sans casser les tests existants.
   * Assurez-vous que le code respecte les bonnes pratiques de développement.

**Exemples de Tests (Initialement Échoués)**

public class LibraryServiceTest {

private LibraryService libraryService;

private BookRepository bookRepository;

@BeforeEach

public void setUp() {

bookRepository = mock(BookRepository.class);

libraryService = new LibraryService(bookRepository);

}

@Test

public void testSearchBooksByTitle\_Found() {

// Arrange

Book book1 = new Book(1, "Effective Java", "Joshua Bloch");

Book book2 = new Book(2, "Java Concurrency in Practice", "Brian Goetz");

List<Book> books = Arrays.asList(book1, book2);

when(bookRepository.findAll()).thenReturn(books);

// Act

List<Book> result = libraryService.searchBooksByTitle("Java");

// Assert

assertEquals(2, result.size());

assertTrue(result.contains(book1));

assertTrue(result.contains(book2));

}

@Test

public void testSearchBooksByTitle\_NotFound() {

// Arrange

Book book1 = new Book(1, "Effective Java", "Joshua Bloch");

Book book2 = new Book(2, "Java Concurrency in Practice", "Brian Goetz");

List<Book> books = Arrays.asList(book1, book2);

when(bookRepository.findAll()).thenReturn(books);

// Act

List<Book> result = libraryService.searchBooksByTitle("Python");

// Assert

assertTrue(result.isEmpty());

}

}

**Livrables**

* La classe LibraryService avec la méthode searchBooksByTitle implémentée.
* Les tests dans LibraryServiceTest qui valident la fonctionnalité.
* Un rapport expliquant les étapes suivies et les choix de conception.

**TP 8 : Écriture de Tests avant l'Implémentation dans une Application de Gestion de Bibliothèque**

**Objectif**

L'objectif de ce TP est de comprendre l'importance des tests unitaires dans le processus de développement logiciel en écrivant des tests pour une nouvelle fonctionnalité avant d'implémenter le code correspondant dans une application de gestion de bibliothèque. Vous allez vous exercer à penser aux comportements attendus de la fonctionnalité et à les traduire en tests unitaires.

**Contexte**

Vous êtes membre de l'équipe de développement d'une application de gestion de bibliothèque. Cette application permet aux utilisateurs de rechercher des livres, d'effectuer des emprunts, de gérer leur profil utilisateur, etc. Une nouvelle fonctionnalité doit être ajoutée à l'application : la possibilité pour les utilisateurs de marquer un livre comme "favori".

**Tâches**

1. **Introduction (5 minutes)**
   * Introduction à l'importance des tests unitaires dans le processus de développement logiciel.
   * Explication du concept d'écriture de tests avant l'implémentation du code.
2. **Choix de la Fonctionnalité (5 minutes)**
   * Chaque participant choisit la fonctionnalité de marquer un livre comme "favori" à implémenter et à tester.
3. **Configuration de l'Environnement (5 minutes)**
   * Les participants configurent leur environnement de développement avec les outils nécessaires pour écrire des tests unitaires en Java (par exemple, IDE, Maven, JUnit).
4. **Écriture des Tests (10 minutes)**
   * Les participants travaillent individuellement.
   * Pour la fonctionnalité choisie, chaque participant écrit les tests unitaires en décrivant les comportements attendus de la fonctionnalité.
   * Les tests doivent être écrits en utilisant JUnit.
5. **Implémentation des Tests (10 minutes)**
   * Les participants implémentent les tests unitaires dans le projet de l'application de gestion de bibliothèque en utilisant Maven.
6. **Discussion (5 minutes)**
   * Les participants partagent leurs expériences avec l'écriture des tests avant l'implémentation du code.
   * Discussion sur les avantages de cette approche, comme la clarification des exigences et la détection précoce des erreurs de conception.
7. **Conclusion (5 minutes)**
   * Récapitulation des principaux points abordés pendant le TP.
   * Encouragement à l'intégration de cette pratique dans les futurs projets de développement logiciel.

**Livrables**

Chaque participant doit fournir :

* Les tests unitaires écrits pour la nouvelle fonctionnalité, intégrés dans le projet de l'application de gestion de bibliothèque.
* Une brève justification des choix de tests réalisés.

**Ressources**

* Environnement de développement (IDE compatible avec Java, Maven, JUnit).
* Documentation sur les bonnes pratiques d'écriture de tests unitaires avec JUnit.

Ce TP permettra aux participants de comprendre l'importance des tests unitaires dans le processus de développement logiciel et de développer leurs compétences dans l'écriture de tests avant l'implémentation du code, dans le contexte spécifique d'une application de gestion de bibliothèque.

**TP 9 : Pratique de Faire Échouer, Faire Passer et Refactoriser des Tests dans une Application de Gestion de Bibliothèque**

**Objectif**

L'objectif de ce TP est de comprendre le cycle de développement itératif en pratiquant la technique de "Faire Échouer, Faire Passer, Refactoriser" pour une fonctionnalité donnée dans une application de gestion de bibliothèque. Vous allez vous exercer à écrire des tests qui échouent d'abord, puis à implémenter le code pour les faire passer, suivi d'une phase de refactorisation pour améliorer la qualité du code.

**Contexte**

Vous êtes membre de l'équipe de développement d'une application de gestion de bibliothèque. Cette application permet aux utilisateurs de rechercher des livres, d'effectuer des emprunts, de gérer leur profil utilisateur, etc. Une nouvelle fonctionnalité doit être ajoutée à l'application : la possibilité pour les utilisateurs de marquer un livre comme "favori".

**Tâches**

1. **Introduction (5 minutes)**
   * Introduction à la technique de "Faire Échouer, Faire Passer, Refactoriser".
   * Explication du cycle de développement itératif et de son importance dans la conception logicielle.
2. **Choix de la Fonctionnalité (5 minutes)**
   * Chaque participant choisit la fonctionnalité de marquer un livre comme "favori" à implémenter et à tester.
3. **Écriture des Tests (10 minutes)**
   * Les participants travaillent individuellement.
   * Pour la fonctionnalité choisie, chaque participant écrit les tests unitaires en décrivant les comportements attendus de la fonctionnalité.
   * Les tests doivent être écrits de manière à échouer lors de la première exécution.

Exemple de code pour un test unitaire qui échoue :

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

import org.junit.jupiter.api.Test;

public class LibraryTest {

Library library = new Library();

@Test

public void testMarkBookAsFavorite() {

Book book = new Book("1234567890", "Le Petit Prince", "Antoine de Saint-Exupéry");

library.addBook(book);

library.markBookAsFavorite("1234567890");

assertTrue(book.isFavorite()); // Cette assertion échouera car la fonctionnalité n'a pas encore été implémentée

}

}

1. **Implémentation du Code (10 minutes)**
   * Les participants implémentent le code nécessaire pour faire passer les tests écrits précédemment.
   * L'objectif est de faire en sorte que tous les tests unitaires passent avec succès.

Exemple de code pour l'implémentation de la fonctionnalité "marquer un livre comme favori" dans la classe Library :

public class Library {

private List<Book> books = new ArrayList<>();

public void addBook(Book book) {

books.add(book);

}

public void markBookAsFavorite(String isbn) {

for (Book book : books) {

if (book.getIsbn().equals(isbn)) {

book.setFavorite(true);

break;

}

}

}

}

1. **Refactorisation du Code (5 minutes)**
   * Les participants examinent le code implémenté et identifient les améliorations possibles.
   * Ils refactorisent le code pour améliorer sa lisibilité, sa maintenabilité et sa performance.
2. **Discussion (5 minutes)**
   * Les participants partagent leurs expériences avec la pratique de "Faire Échouer, Faire Passer, Refactoriser".
   * Discussion sur les avantages de cette approche, comme la réduction des erreurs et la facilitation de la maintenance du code.

**Ressources**

* Environnement de développement (IDE compatible avec Java, Maven, JUnit).
* Documentation sur les bonnes pratiques d'écriture de tests unitaires avec JUnit.

Ce TP permettra aux participants de comprendre et de mettre en pratique le cycle de développement itératif en écrivant des tests qui échouent d'abord, en implémentant le code pour les faire passer, et en refactorisant ensuite le code pour améliorer sa qualité dans le contexte spécifique d'une application de gestion de bibliothèque.

**TP 10 : Refactorisation pour Utiliser la Composition plutôt que l'Héritage**

**Objectif**

L'objectif de ce TP est de comprendre les principes de conception SOLID en pratiquant la refactorisation d'une classe pour remplacer l'héritage par la composition dans le contexte d'une application de gestion de bibliothèque. Vous allez vous exercer à identifier les relations de type "est-un" et "a-un" entre les classes et à appliquer la composition pour améliorer la flexibilité et la maintenabilité du code.

**Contexte**

Vous êtes membre de l'équipe de développement d'une application de gestion de bibliothèque. L'application comporte une classe Book qui représente un livre avec ses attributs et ses méthodes. Actuellement, la classe Book étend une classe Item, qui contient des attributs et des méthodes communs à tous les types d'items de la bibliothèque.

**Tâches**

1. **Introduction (5 minutes)**
   * Introduction aux principes de conception SOLID, en mettant l'accent sur le principe de substitution de Liskov (LSP) et le principe d'inversion de dépendance (DIP).
   * Explication des avantages de l'utilisation de la composition par rapport à l'héritage dans certaines situations.
2. **Analyse de la Classe Book (5 minutes)**
   * Les participants examinent la classe Book et identifient les fonctionnalités spécifiques aux livres ainsi que les fonctionnalités partagées avec d'autres types d'items de la bibliothèque.
3. **Refactorisation de la Classe Book (15 minutes)**
   * Les participants travaillent individuellement.
   * Pour chaque fonctionnalité partagée avec d'autres types d'items, les participants extraient cette fonctionnalité dans une classe séparée et l'injectent dans la classe Book via la composition.
   * Les participants doivent s'assurer que la classe Book ne dépend plus de la classe Item pour son fonctionnement.

Exemple de refactorisation de la classe Book en utilisant la composition :

public class Book {

private String title;

private String author;

private String isbn;

private ItemDetails itemDetails;

public Book(String title, String author, String isbn) {

this.title = title;

this.author = author;

this.isbn = isbn;

this.itemDetails = new ItemDetails();

}

// Getters and setters for title, author, isbn

public String getDetails() {

return itemDetails.getDetails();

}

}

1. **Discussion (5 minutes)**
   * Les participants partagent leurs expériences avec la refactorisation de la classe Book en utilisant la composition.
   * Discussion sur les avantages de cette approche, notamment en termes de flexibilité, de maintenabilité et de réutilisabilité du code.

**Livrables**

Chaque participant doit fournir :

* La version refactorisée de la classe Book utilisant la composition plutôt que l'héritage.
* Une brève justification des choix de refactorisation réalisés.

**Ressources**

* Environnement de développement (IDE compatible avec Java).
* Documentation sur les principes de conception SOLID et sur les avantages de l'utilisation de la composition.

Ce TP permettra aux participants de comprendre et de mettre en pratique les principes de conception SOLID en refactorisant une classe pour utiliser la composition plutôt que l'héritage dans le contexte spécifique d'une application de gestion de bibliothèque.

**TP 11 : Refactorisation de Code Statique pour le Rendre plus Testable dans une Application de Gestion de Bibliothèque**

**Objectif**

L'objectif de ce TP est de comprendre les problèmes liés à l'utilisation de méthodes et de classes statiques dans le code, et d'apprendre à les refactoriser pour améliorer la testabilité. Vous allez identifier du code statique existant dans une application de gestion de bibliothèque, puis le refactoriser pour le rendre plus modulaire et plus facile à tester.

**Contexte**

Vous êtes membre de l'équipe de développement d'une application de gestion de bibliothèque. L'application comporte plusieurs méthodes statiques utilisées pour effectuer diverses opérations, telles que la gestion des dates de retour des livres, la génération de rapports, etc. Ces méthodes statiques rendent le code difficile à tester et à maintenir.

**Tâches**

1. **Introduction (5 minutes)**
   * Introduction aux problèmes liés à l'utilisation de méthodes et de classes statiques.
   * Explication des techniques de refactorisation pour rendre le code plus testable, notamment l'injection de dépendances et l'utilisation d'interfaces.
2. **Analyse du Code Statique (5 minutes)**
   * Les participants examinent une classe LibraryUtils qui contient plusieurs méthodes statiques utilisées dans l'application de gestion de bibliothèque.
3. **Refactorisation du Code Statique (15 minutes)**
   * Les participants travaillent individuellement.
   * Ils refactorisent les méthodes statiques de LibraryUtils pour les transformer en méthodes d'instance, en injectant les dépendances nécessaires via des constructeurs ou des setters.
   * Les participants doivent s'assurer que le code refactorisé est modulaire et plus facile à tester.

Exemple de code avant refactorisation :

public class LibraryUtils {

public static Date calculateReturnDate(Date borrowDate, int loanPeriodDays) {

Calendar calendar = Calendar.getInstance();

calendar.setTime(borrowDate);

calendar.add(Calendar.DAY\_OF\_MONTH, loanPeriodDays);

return calendar.getTime();

}

public static void printReport(List<Book> books) {

for (Book book : books) {

System.out.println(book.getTitle() + " by " + book.getAuthor());

}

}

}

Exemple de code après refactorisation :

public class LibraryUtils {

private DateProvider dateProvider;

private ReportPrinter reportPrinter;

public LibraryUtils(DateProvider dateProvider, ReportPrinter reportPrinter) {

this.dateProvider = dateProvider;

this.reportPrinter = reportPrinter;

}

public Date calculateReturnDate(Date borrowDate, int loanPeriodDays) {

Calendar calendar = Calendar.getInstance();

calendar.setTime(borrowDate);

calendar.add(Calendar.DAY\_OF\_MONTH, loanPeriodDays);

return calendar.getTime();

}

public void printReport(List<Book> books) {

reportPrinter.print(books);

}

}

public class DateProvider {

public Date getCurrentDate() {

return new Date();

}

}

public class ReportPrinter {

public void print(List<Book> books) {

for (Book book : books) {

System.out.println(book.getTitle() + " by " + book.getAuthor());

}

}

}

1. **Écriture de Tests Unitaires (5 minutes)**
   * Les participants écrivent des tests unitaires pour les méthodes refactorisées en utilisant des frameworks de test comme JUnit et Mockito.
   * Ils s'assurent que les méthodes refactorisées fonctionnent correctement et sont faciles à tester.

Exemple de test unitaire :

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.\*;

public class LibraryUtilsTest {

private LibraryUtils libraryUtils;

private DateProvider mockDateProvider;

private ReportPrinter mockReportPrinter;

@BeforeEach

public void setUp() {

mockDateProvider = mock(DateProvider.class);

mockReportPrinter = mock(ReportPrinter.class);

libraryUtils = new LibraryUtils(mockDateProvider, mockReportPrinter);

}

@Test

public void testCalculateReturnDate() {

Date borrowDate = new Date();

int loanPeriodDays = 14;

Date expectedReturnDate = new Date(borrowDate.getTime() + loanPeriodDays \* 24 \* 60 \* 60 \* 1000L);

Date actualReturnDate = libraryUtils.calculateReturnDate(borrowDate, loanPeriodDays);

assertEquals(expectedReturnDate, actualReturnDate);

}

@Test

public void testPrintReport() {

List<Book> books = Arrays.asList(new Book("1984", "George Orwell", "1234567890"));

libraryUtils.printReport(books);

verify(mockReportPrinter).print(books);

}

}

1. **Discussion (5 minutes)**
   * Les participants partagent leurs expériences avec la refactorisation du code statique.
   * Discussion sur les avantages de rendre le code plus modulaire et testable, ainsi que sur les meilleures pratiques pour éviter les méthodes statiques non testables à l'avenir.

**Livrables**

Chaque participant doit fournir :

* La version refactorisée de la classe LibraryUtils en utilisant des dépendances injectées.
* Les tests unitaires couvrant les méthodes refactorisées.
* Une brève justification des choix de refactorisation réalisés.

**Ressources**

* Environnement de développement (IDE compatible avec Java).
* Documentation sur les principes de conception orientée objet et les techniques de refactorisation.
* Documentation sur les frameworks de test (JUnit, Mockito).

Ce TP permettra aux participants de comprendre les problèmes liés à l'utilisation de méthodes et de classes statiques, et d'apprendre à les refactoriser pour améliorer la testabilité, la modularité et la maintenabilité du code dans le contexte spécifique d'une application de gestion de bibliothèque.

**TP 12 : Implémentation de l'Inversion de Dépendances dans un Projet de Gestion de Bibliothèque**

**Objectif**

L'objectif de ce TP est de comprendre et d'implémenter le principe de l'inversion de dépendances (DIP) dans un projet existant. Vous allez refactoriser des parties du code pour inverser les dépendances et rendre le système plus modulaire et testable.

**Contexte**

Vous travaillez sur une application de gestion de bibliothèque. Actuellement, certaines classes dépendent directement de classes concrètes, ce qui rend le code difficile à tester et à maintenir. Votre tâche est de refactoriser le code pour appliquer le principe de l'inversion de dépendances en utilisant des interfaces et l'injection de dépendances.

**Instructions**

**Étape 1 : Analyse du Code Existants (5 minutes)**

1. Identifiez les dépendances directes dans le code existant.
2. Notez les classes qui dépendent directement de classes concrètes.

**Étape 2 : Création d'Interfaces (5 minutes)**

1. Créez des interfaces pour les classes concrètes identifiées.
2. Modifiez les classes concrètes pour qu'elles implémentent ces interfaces.

**Étape 3 : Refactorisation pour l'Inversion de Dépendances (15 minutes)**

1. Modifiez les classes dépendantes pour qu'elles dépendent des interfaces plutôt que des classes concrètes.
2. Utilisez l'injection de dépendances pour fournir les implémentations concrètes aux classes dépendantes.

**Étape 4 : Écriture de Tests Unitaires (5 minutes)**

1. Écrivez des tests unitaires pour vérifier que le code refactorisé fonctionne correctement.
2. Utilisez des mocks pour tester les interactions entre les classes dépendantes et leurs dépendances.

**Exemple de Code Avant Refactorisation**

public class LibraryService {

private BookRepository bookRepository = new BookRepository();

public List<Book> getAllBooks() {

return bookRepository.findAllBooks();

}

}

public class BookRepository {

public List<Book> findAllBooks() {

// Logique pour récupérer les livres de la base de données

}

}

**Exemple de Code Après Refactorisation**

**Création d'Interfaces**

public interface BookRepository {

List<Book> findAllBooks();

}

**Implémentation de l'Interface**

public class BookRepositoryImpl implements BookRepository {

public List<Book> findAllBooks() {

// Logique pour récupérer les livres de la base de données

}

}

**Refactorisation de la Classe Dépendante**

public class LibraryService {

private BookRepository bookRepository;

public LibraryService(BookRepository bookRepository) {

this.bookRepository = bookRepository;

}

public List<Book> getAllBooks() {

return bookRepository.findAllBooks();

}

}

**Tests Unitaires**

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

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.\*;

import java.util.Arrays;

import java.util.List;

public class LibraryServiceTest {

private LibraryService libraryService;

private BookRepository mockBookRepository;

@Before

public void setUp() {

mockBookRepository = mock(BookRepository.class);

libraryService = new LibraryService(mockBookRepository);

}

@Test

public void testGetAllBooks() {

List<Book> books = Arrays.asList(new Book("1984", "George Orwell", "1234567890"));

when(mockBookRepository.findAllBooks()).thenReturn(books);

List<Book> result = libraryService.getAllBooks();

assertEquals(books, result);

verify(mockBookRepository).findAllBooks();

}

}

**Conclusion**

Ce TP vous a permis de comprendre et d'implémenter le principe de l'inversion de dépendances dans une application de gestion de bibliothèque. En refactorisant le code pour dépendre des interfaces plutôt que des classes concrètes, vous avez rendu le système plus modulaire et testable. L'injection de dépendances permet de découpler les classes et de faciliter les tests unitaires en utilisant des mocks.

**TP 13 : Écrire des Tests en Utilisant Différentes Assertions**

**Objectif**

L'objectif de ce TP est d'apprendre à utiliser différentes assertions pour écrire des tests unitaires complets et robustes. Vous allez écrire des tests pour une application de gestion de bibliothèque en utilisant diverses assertions pour valider le comportement des méthodes.

**Contexte**

Vous travaillez sur une application de gestion de bibliothèque. Cette application possède plusieurs fonctionnalités telles que la gestion des livres, des membres et des emprunts. Nous allons écrire des tests unitaires pour certaines de ces fonctionnalités en utilisant différentes assertions disponibles dans JUnit.

**Instructions**

**Étape 1 : Préparation du Projet (5 minutes)**

1. Assurez-vous que votre projet est configuré pour utiliser JUnit.
2. Créez une classe de test pour les fonctionnalités suivantes :
   * Ajout de nouveaux livres
   * Inscription de nouveaux membres
   * Enregistrement des emprunts de livres

**Étape 2 : Écrire des Tests pour l'Ajout de Nouveaux Livres (10 minutes)**

1. Écrivez des tests pour la méthode addBook qui ajoute un nouveau livre à la bibliothèque.
2. Utilisez différentes assertions pour valider :
   * Que le livre a été ajouté correctement
   * Que les informations du livre sont correctes
   * Que le nombre de livres a augmenté

**Étape 3 : Écrire des Tests pour l'Inscription de Nouveaux Membres (5 minutes)**

1. Écrivez des tests pour la méthode registerMember qui inscrit un nouveau membre à la bibliothèque.
2. Utilisez différentes assertions pour valider :
   * Que le membre a été inscrit correctement
   * Que les informations du membre sont correctes
   * Que le nombre de membres a augmenté

**Étape 4 : Écrire des Tests pour l'Enregistrement des Emprunts de Livres (5 minutes)**

1. Écrivez des tests pour la méthode borrowBook qui enregistre l'emprunt d'un livre par un membre.
2. Utilisez différentes assertions pour valider :
   * Que l'emprunt a été enregistré correctement
   * Que le livre n'est plus disponible
   * Que le nombre d'emprunts actifs a augmenté

**Exemple de Code Avant de Commencer**

**LibraryService.java**

public class LibraryService {

private List<Book> books = new ArrayList<>();

private List<Member> members = new ArrayList<>();

private List<BorrowRecord> borrowRecords = new ArrayList<>();

public void addBook(Book book) {

books.add(book);

}

public void registerMember(Member member) {

members.add(member);

}

public void borrowBook(Member member, Book book) {

if (books.contains(book) && !isBookBorrowed(book)) {

borrowRecords.add(new BorrowRecord(member, book, new Date()));

}

}

private boolean isBookBorrowed(Book book) {

return borrowRecords.stream().anyMatch(record -> record.getBook().equals(book));

}

// Getters for testing

public List<Book> getBooks() {

return books;

}

public List<Member> getMembers() {

return members;

}

public List<BorrowRecord> getBorrowRecords() {

return borrowRecords;

}

}

**LibraryServiceTest.java**

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

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import java.util.Date;

public class LibraryServiceTest {

private LibraryService libraryService;

@BeforeEach

public void setUp() {

libraryService = new LibraryService();

}

@Test

public void testAddBook() {

Book book = new Book("1984", "George Orwell", "1234567890");

libraryService.addBook(book);

assertFalse(libraryService.getBooks().isEmpty());

assertEquals(1, libraryService.getBooks().size());

assertEquals("1984", libraryService.getBooks().get(0).getTitle());

assertEquals("George Orwell", libraryService.getBooks().get(0).getAuthor());

}

@Test

public void testRegisterMember() {

Member member = new Member("John Doe", "12345");

libraryService.registerMember(member);

assertFalse(libraryService.getMembers().isEmpty());

assertEquals(1, libraryService.getMembers().size());

assertEquals("John Doe", libraryService.getMembers().get(0).getName());

}

@Test

public void testBorrowBook() {

Book book = new Book("1984", "George Orwell", "1234567890");

Member member = new Member("John Doe", "12345");

libraryService.addBook(book);

libraryService.registerMember(member);

libraryService.borrowBook(member, book);

assertFalse(libraryService.getBorrowRecords().isEmpty());

assertEquals(1, libraryService.getBorrowRecords().size());

assertEquals("1984", libraryService.getBorrowRecords().get(0).getBook().getTitle());

assertEquals("John Doe", libraryService.getBorrowRecords().get(0).getMember().getName());

}

}

**Conclusion**

Ce TP vous permet de pratiquer l'écriture de tests en utilisant différentes assertions pour valider le comportement de votre code. En utilisant des assertions variées, vous pouvez vérifier que votre code fonctionne correctement sous différents aspects, ce qui améliore la robustesse et la fiabilité de votre application.

**TP 14 : Écrire des Tests pour des Scénarios de Panne (Tests d'Échec)**

**Objectif**

L'objectif de ce TP est d'apprendre à écrire des tests unitaires pour des scénarios de panne, afin de vérifier que l'application gère correctement les erreurs et les situations exceptionnelles. Vous allez écrire des tests pour une application de gestion de bibliothèque qui simulent des pannes et valider leur gestion.

**Contexte**

Vous travaillez sur une application de gestion de bibliothèque. Cette application possède plusieurs fonctionnalités telles que la gestion des livres, des membres et des emprunts. Nous allons écrire des tests unitaires pour vérifier la gestion des erreurs et des pannes dans certaines de ces fonctionnalités.

**Instructions**

**Étape 1 : Préparation du Projet (5 minutes)**

1. Assurez-vous que votre projet est configuré pour utiliser JUnit.
2. Créez une classe de test pour les fonctionnalités suivantes :
   * Ajout de nouveaux livres
   * Inscription de nouveaux membres
   * Enregistrement des emprunts de livres

**Étape 2 : Écrire des Tests pour l'Ajout de Nouveaux Livres (10 minutes)**

1. Écrivez des tests pour la méthode addBook qui simulent des erreurs telles que :
   * L'ajout d'un livre avec un ISBN déjà existant
   * L'ajout d'un livre avec des informations manquantes
2. Utilisez des assertions pour valider que les exceptions appropriées sont lancées et que les messages d'erreur sont corrects.

**Étape 3 : Écrire des Tests pour l'Inscription de Nouveaux Membres (5 minutes)**

1. Écrivez des tests pour la méthode registerMember qui simulent des erreurs telles que :
   * L'inscription d'un membre avec un identifiant déjà existant
   * L'inscription d'un membre avec des informations manquantes
2. Utilisez des assertions pour valider que les exceptions appropriées sont lancées et que les messages d'erreur sont corrects.

**Étape 4 : Écrire des Tests pour l'Enregistrement des Emprunts de Livres (5 minutes)**

1. Écrivez des tests pour la méthode borrowBook qui simulent des erreurs telles que :
   * L'emprunt d'un livre déjà emprunté
   * L'emprunt d'un livre par un membre non inscrit
2. Utilisez des assertions pour valider que les exceptions appropriées sont lancées et que les messages d'erreur sont corrects.

**Exemple de Code Avant de Commencer**

**LibraryService.java**

import java.util.ArrayList;

import java.util.Date;

import java.util.List;

public class LibraryService {

private List<Book> books = new ArrayList<>();

private List<Member> members = new ArrayList<>();

private List<BorrowRecord> borrowRecords = new ArrayList<>();

public void addBook(Book book) throws IllegalArgumentException {

if (book.getIsbn() == null || book.getTitle() == null || book.getAuthor() == null) {

throw new IllegalArgumentException("Missing book information");

}

for (Book b : books) {

if (b.getIsbn().equals(book.getIsbn())) {

throw new IllegalArgumentException("Book with this ISBN already exists");

}

}

books.add(book);

}

public void registerMember(Member member) throws IllegalArgumentException {

if (member.getMemberId() == null || member.getName() == null) {

throw new IllegalArgumentException("Missing member information");

}

for (Member m : members) {

if (m.getMemberId().equals(member.getMemberId())) {

throw new IllegalArgumentException("Member with this ID already exists");

}

}

members.add(member);

}

public void borrowBook(Member member, Book book) throws IllegalArgumentException {

if (!members.contains(member)) {

throw new IllegalArgumentException("Member not registered");

}

if (!books.contains(book)) {

throw new IllegalArgumentException("Book not available");

}

if (isBookBorrowed(book)) {

throw new IllegalArgumentException("Book already borrowed");

}

borrowRecords.add(new BorrowRecord(member, book, new Date()));

}

private boolean isBookBorrowed(Book book) {

return borrowRecords.stream().anyMatch(record -> record.getBook().equals(book));

}

// Getters for testing

public List<Book> getBooks() {

return books;

}

public List<Member> getMembers() {

return members;

}

public List<BorrowRecord> getBorrowRecords() {

return borrowRecords;

}

}

**Exemple de Tests**

**LibraryServiceTest.java**

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

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

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import java.util.Date;

public class LibraryServiceTest {

private LibraryService libraryService;

@BeforeEach

public void setUp() {

libraryService = new LibraryService();

}

@Test

public void testAddBookWithMissingInformation() {

Book book = new Book(null, "George Orwell", "1234567890");

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

libraryService.addBook(book);

});

assertEquals("Missing book information", exception.getMessage());

}

@Test

public void testAddBookWithDuplicateISBN() {

Book book1 = new Book("1984", "George Orwell", "1234567890");

Book book2 = new Book("Animal Farm", "George Orwell", "1234567890");

libraryService.addBook(book1);

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

libraryService.addBook(book2);

});

assertEquals("Book with this ISBN already exists", exception.getMessage());

}

@Test

public void testRegisterMemberWithMissingInformation() {

Member member = new Member(null, "John Doe");

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

libraryService.registerMember(member);

});

assertEquals("Missing member information", exception.getMessage());

}

@Test

public void testRegisterMemberWithDuplicateID() {

Member member1 = new Member("John Doe", "12345");

Member member2 = new Member("Jane Doe", "12345");

libraryService.registerMember(member1);

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

libraryService.registerMember(member2);

});

assertEquals("Member with this ID already exists", exception.getMessage());

}

@Test

public void testBorrowBookAlreadyBorrowed() {

Book book = new Book("1984", "George Orwell", "1234567890");

Member member = new Member("John Doe", "12345");

libraryService.addBook(book);

libraryService.registerMember(member);

libraryService.borrowBook(member, book);

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

libraryService.borrowBook(member, book);

});

assertEquals("Book already borrowed", exception.getMessage());

}

@Test

public void testBorrowBookByNonRegisteredMember() {

Book book = new Book("1984", "George Orwell", "1234567890");

Member member = new Member("John Doe", "12345");

libraryService.addBook(book);

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

libraryService.borrowBook(member, book);

});

assertEquals("Member not registered", exception.getMessage());

}

}

**Conclusion**

Ce TP vous a permis de pratiquer l'écriture de tests pour des scénarios de panne. En simulant des erreurs et en utilisant des assertions pour vérifier que les exceptions appropriées sont lancées et que les messages d'erreur sont corrects, vous vous assurez que votre application gère correctement les situations exceptionnelles et les erreurs. Cela améliore la robustesse et la fiabilité de votre application.

**TP 15 : Implémenter des Tests Paramétrés sur les Types et les Valeurs avec JUnit**

**Objectif**

L'objectif de ce TP est d'apprendre à écrire des tests paramétrés en utilisant JUnit. Vous allez écrire des tests pour une application de gestion de bibliothèque afin de vérifier diverses combinaisons de types et de valeurs pour certaines fonctionnalités.

**Contexte**

Vous travaillez sur une application de gestion de bibliothèque. Cette application possède plusieurs fonctionnalités telles que la gestion des livres, des membres et des emprunts. Nous allons écrire des tests paramétrés pour vérifier les différentes combinaisons de données d'entrée pour certaines fonctionnalités.

**Instructions**

**Étape 1 : Préparation du Projet (5 minutes)**

1. Assurez-vous que votre projet est configuré pour utiliser JUnit 5.
2. Ajoutez la dépendance junit-jupiter-params à votre fichier pom.xml si ce n'est pas déjà fait.

**Exemple de dépendance Maven :**

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-params</artifactId>

<version>5.8.1</version>

<scope>test</scope>

</dependency>

**Étape 2 : Écrire des Tests Paramétrés pour l'Ajout de Livres (10 minutes)**

1. Écrivez des tests paramétrés pour la méthode addBook qui vérifient différentes combinaisons de titres, auteurs et ISBN.
2. Utilisez des assertions pour valider que les livres sont ajoutés correctement ou que les exceptions appropriées sont lancées en cas d'erreurs.

**Étape 3 : Écrire des Tests Paramétrés pour l'Inscription de Membres (10 minutes)**

1. Écrivez des tests paramétrés pour la méthode registerMember qui vérifient différentes combinaisons de noms et identifiants de membres.
2. Utilisez des assertions pour valider que les membres sont inscrits correctement ou que les exceptions appropriées sont lancées en cas d'erreurs.

**Exemple de Code Avant de Commencer**

**LibraryService.java**

import java.util.ArrayList;

import java.util.List;

public class LibraryService {

private List<Book> books = new ArrayList<>();

private List<Member> members = new ArrayList<>();

public void addBook(Book book) throws IllegalArgumentException {

if (book.getIsbn() == null || book.getTitle() == null || book.getAuthor() == null) {

throw new IllegalArgumentException("Missing book information");

}

for (Book b : books) {

if (b.getIsbn().equals(book.getIsbn())) {

throw new IllegalArgumentException("Book with this ISBN already exists");

}

}

books.add(book);

}

public void registerMember(Member member) throws IllegalArgumentException {

if (member.getMemberId() == null || member.getName() == null) {

throw new IllegalArgumentException("Missing member information");

}

for (Member m : members) {

if (m.getMemberId().equals(member.getMemberId())) {

throw new IllegalArgumentException("Member with this ID already exists");

}

}

members.add(member);

}

}

**Exemple de Tests Paramétrés**

**LibraryServiceTest.java**

import org.junit.jupiter.api.BeforeEach;

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.MethodSource;

import java.util.stream.Stream;

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

public class LibraryServiceTest {

private LibraryService libraryService;

@BeforeEach

public void setUp() {

libraryService = new LibraryService();

}

@ParameterizedTest

@MethodSource("provideBooksForAddBook")

public void testAddBookParametrized(String title, String author, String isbn, boolean shouldThrowException) {

Book book = new Book(title, author, isbn);

if (shouldThrowException) {

assertThrows(IllegalArgumentException.class, () -> libraryService.addBook(book));

} else {

assertDoesNotThrow(() -> libraryService.addBook(book));

}

}

private static Stream<Arguments> provideBooksForAddBook() {

return Stream.of(

Arguments.of("1984", "George Orwell", "1234567890", false),

Arguments.of(null, "George Orwell", "1234567890", true),

Arguments.of("1984", null, "1234567890", true),

Arguments.of("1984", "George Orwell", null, true),

Arguments.of("1984", "George Orwell", "1234567890", false) // Duplicate ISBN to test duplicate addition

);

}

@ParameterizedTest

@MethodSource("provideMembersForRegisterMember")

public void testRegisterMemberParametrized(String name, String memberId, boolean shouldThrowException) {

Member member = new Member(name, memberId);

if (shouldThrowException) {

assertThrows(IllegalArgumentException.class, () -> libraryService.registerMember(member));

} else {

assertDoesNotThrow(() -> libraryService.registerMember(member));

}

}

private static Stream<Arguments> provideMembersForRegisterMember() {

return Stream.of(

Arguments.of("John Doe", "12345", false),

Arguments.of(null, "12345", true),

Arguments.of("John Doe", null, true),

Arguments.of("Jane Doe", "12345", true), // Duplicate ID to test duplicate addition

Arguments.of("John Doe", "67890", false)

);

}

}

**Conclusion**

Ce TP vous a permis de pratiquer l'écriture de tests paramétrés en utilisant JUnit. Les tests paramétrés sont un outil puissant pour vérifier différentes combinaisons de données d'entrée et assurer que votre application fonctionne correctement dans un large éventail de scénarios. En utilisant des sources de méthodes pour fournir des arguments de test, vous pouvez facilement étendre et maintenir vos tests.

**TP 16 : Utiliser les Matchers pour Écrire des Tests Plus Lisibles**

**Objectif**

L'objectif de ce TP est d'apprendre à utiliser les matchers fournis par les frameworks de test (comme Hamcrest) pour écrire des tests plus lisibles et expressifs. Vous allez écrire des tests pour une application de gestion de bibliothèque en utilisant des matchers pour vérifier les résultats attendus de manière plus claire.

**Contexte**

Vous travaillez sur une application de gestion de bibliothèque. Cette application possède plusieurs fonctionnalités telles que la gestion des livres, des membres et des emprunts. Vous allez écrire des tests pour vérifier le comportement de ces fonctionnalités en utilisant des matchers pour les assertions.

**Instructions**

1. Configurez votre projet pour utiliser un framework de test compatible avec Hamcrest (comme JUnit Jupiter avec Hamcrest).
2. Écrivez des tests pour les fonctionnalités suivantes de l'application de gestion de bibliothèque en utilisant des matchers pour les assertions :
   * Ajout de nouveaux livres
   * Inscription de nouveaux membres
   * Enregistrement des emprunts de livres

**Exemple de Test Utilisant des Matchers**

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import static org.hamcrest.MatcherAssert.assertThat;

import static org.hamcrest.Matchers.\*;

public class LibraryServiceTest {

private LibraryService libraryService;

@BeforeEach

public void setUp() {

libraryService = new LibraryService();

}

@Test

public void testAddBook() {

Book book = new Book("1984", "George Orwell", "1234567890");

libraryService.addBook(book);

assertThat(libraryService.getBooks(), hasItem(book));

}

@Test

public void testRegisterMember() {

Member member = new Member("John Doe", "12345");

libraryService.registerMember(member);

assertThat(libraryService.getMembers(), hasItem(member));

}

@Test

public void testBorrowBook() {

Book book = new Book("1984", "George Orwell", "1234567890");

Member member = new Member("John Doe", "12345");

libraryService.addBook(book);

libraryService.registerMember(member);

libraryService.borrowBook(member, book);

assertThat(libraryService.getBorrowRecords(), hasItem(

allOf(

hasProperty("book", equalTo(book)),

hasProperty("member", equalTo(member))

)

));

}

}

**Conclusion**

Ce TP vous a permis de pratiquer l'utilisation des matchers pour écrire des tests plus lisibles et expressifs. En utilisant des matchers comme hasItem, equalTo, hasProperty, etc., vous pouvez rendre vos assertions plus claires et plus facilement compréhensibles. Cela améliore la lisibilité de vos tests et facilite la détection des erreurs lors de l'exécution.

**TP 17 : Réorganiser un Projet de Tests en Classes de Tests Structurées**

**Objectif**

L'objectif de ce TP est d'apprendre à organiser un projet de tests en classes de tests structurées. Vous allez réorganiser les tests pour une application de gestion de bibliothèque en regroupant les tests pertinents dans des classes de tests distinctes selon leur fonctionnalité.

**Contexte**

Vous travaillez sur une application de gestion de bibliothèque qui possède plusieurs fonctionnalités telles que la gestion des livres, des membres et des emprunts. Actuellement, les tests pour ces fonctionnalités sont dispersés dans un seul fichier de tests. Vous allez réorganiser ces tests en classes de tests distinctes pour une meilleure lisibilité et maintenabilité.

**Instructions**

1. Examinez le fichier de tests existant pour l'application de gestion de bibliothèque.
2. Identifiez les différentes fonctionnalités testées dans ce fichier.
3. Créez des classes de tests distinctes pour chaque fonctionnalité, en regroupant les tests pertinents.
4. Déplacez les tests correspondants dans les classes de tests nouvellement créées.
5. Assurez-vous que chaque classe de tests est nommée de manière significative pour refléter la fonctionnalité qu'elle teste.

**Exemple de Structure de Classes de Tests**

src

└── test

└── java

└── com

└── example

└── library

├── BookManagementTest.java

├── MemberManagementTest.java

├── BorrowingTest.java

└── ReportGenerationTest.java

**Exemple de Classe de Tests**

**BookManagementTest.java**

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

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

public class BookManagementTest {

private LibraryService libraryService;

@BeforeEach

public void setUp() {

libraryService = new LibraryService();

}

@Test

public void testAddBook() {

// Test adding a new book

}

@Test

public void testUpdateBook() {

// Test updating book information

}

@Test

public void testRemoveBook() {

// Test removing a book

}

@Test

public void testSearchBook() {

// Test searching for a book

}

}

**MemberManagementTest.java**

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

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

public class MemberManagementTest {

private LibraryService libraryService;

@BeforeEach

public void setUp() {

libraryService = new LibraryService();

}

@Test

public void testRegisterMember() {

// Test registering a new member

}

@Test

public void testUpdateMember() {

// Test updating member information

}

@Test

public void testRemoveMember() {

// Test removing a member

}

@Test

public void testSearchMember() {

// Test searching for a member

}

}

**BorrowingTest.java**

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

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

public class BorrowingTest {

private LibraryService libraryService;

@BeforeEach

public void setUp() {

libraryService = new LibraryService();

}

@Test

public void testBorrowBook() {

// Test borrowing a book

}

@Test

public void testReturnBook() {

// Test returning a book

}

@Test

public void testPenaltiesForLateReturn() {

// Test penalties for late return

}

}

**Conclusion**

Ce TP vous a permis de pratiquer la réorganisation d'un projet de tests en classes de tests structurées. En regroupant les tests pertinents dans des classes dédiées selon leur fonctionnalité, vous améliorez la lisibilité et la maintenabilité de votre suite de tests. Cette organisation facilite également l'ajout de nouveaux tests et la compréhension des fonctionnalités testées dans l'application.

**TP 18 : Refactoriser le Code de Test pour le Rendre Plus Maintenable**

**Objectif**

L'objectif de ce TP est de pratiquer le refactoring du code de test pour le rendre plus maintenable. Vous allez examiner un fichier de tests existant pour une application de gestion de bibliothèque, identifier les opportunités de refactorisation et apporter des modifications pour améliorer la lisibilité, la maintenabilité et la robustesse des tests.

**Contexte**

Vous travaillez sur une application de gestion de bibliothèque qui possède plusieurs fonctionnalités telles que la gestion des livres, des membres et des emprunts. Actuellement, le fichier de tests pour cette application contient du code qui peut être amélioré en termes de lisibilité et de maintenabilité.

**Instructions**

1. Examinez le fichier de tests existant pour l'application de gestion de bibliothèque.
2. Identifiez les parties du code qui peuvent être améliorées en termes de lisibilité, de duplication de code ou de complexité excessive.
3. Appliquez des techniques de refactoring telles que l'extraction de méthodes, la réduction de la duplication de code et la simplification des assertions pour améliorer le code de test.
4. Assurez-vous que les tests refactorisés restent fonctionnellement équivalents aux tests initiaux et qu'ils couvrent toujours les mêmes cas de test.

**Exemple de Techniques de Refactoring à Appliquer**

1. **Extraction de Méthodes** : Identifiez des blocs de code répétitifs dans les tests et extrayez-les dans des méthodes réutilisables.
2. **Réduction de la Duplication de Code** : Identifiez les parties de code similaires dans différents tests et les facteurs communs pour les rendre plus génériques.
3. **Simplification des Assertions** : Simplifiez les assertions complexes en utilisant des matchers ou des méthodes d'assertion plus expressifs.
4. **Séparation des Responsabilités** : Assurez-vous que chaque test se concentre sur un aspect spécifique de la fonctionnalité testée et évitez de mélanger les préoccupations dans un seul test.

**Exemple de Refactoring**

**Avant le refactoring :**

@Test

public void testAddBook() {

// Initialisation des données

LibraryService libraryService = new LibraryService();

Book book = new Book("1984", "George Orwell", "1234567890");

// Ajout d'un livre

libraryService.addBook(book);

// Vérification que le livre a bien été ajouté

assertTrue(libraryService.getBooks().contains(book));

}

**Après le refactoring :**

@Test

public void testAddBook() {

// Arrange

LibraryService libraryService = new LibraryService();

Book book = new Book("1984", "George Orwell", "1234567890");

// Act

libraryService.addBook(book);

// Assert

assertTrue(libraryService.containsBook(book));

}

**Conclusion**

Ce TP vous permettra de pratiquer le refactoring du code de test pour améliorer sa lisibilité, sa maintenabilité et sa robustesse. En appliquant des techniques de refactoring telles que l'extraction de méthodes, la réduction de la duplication de code et la simplification des assertions, vous pouvez rendre votre suite de tests plus efficace et plus facile à maintenir.

**TP 19 : Identifier et Corriger les Mauvaises Odeurs dans un Projet Donné**

**Objectif**

L'objectif de ce TP est de pratiquer l'identification et la correction des mauvaises odeurs (code smells) dans un projet de tests pour une application de gestion de bibliothèque. Vous allez examiner un fichier de tests existant, identifier les mauvaises odeurs présentes, et proposer des corrections pour améliorer la qualité du code de test.

**Contexte**

Vous travaillez sur un projet de tests pour une application de gestion de bibliothèque. Le fichier de tests contient plusieurs mauvaises odeurs telles que des duplications de code, des assertions complexes, des noms de tests peu descriptifs, etc. Ces mauvaises odeurs rendent le code de test difficile à maintenir et à comprendre.

**Instructions**

1. Examinez attentivement le fichier de tests existant pour l'application de gestion de bibliothèque.
2. Identifiez les mauvaises odeurs présentes dans le code de test. Certaines mauvaises odeurs courantes à rechercher peuvent inclure :
   * Duplications de code
   * Assertions complexes ou redondantes
   * Noms de tests peu descriptifs ou mal choisis
   * Utilisation excessive de commentaires
   * Dépendances entre les tests
   * Tests fragiles ou sensibles aux données
3. Proposez des corrections pour chaque mauvaise odeur identifiée. Vous pouvez utiliser des techniques de refactoring telles que l'extraction de méthodes, la réduction de la duplication de code, la simplification des assertions, etc.
4. Assurez-vous que les corrections proposées améliorent la lisibilité, la maintenabilité et la robustesse du code de test.

**Exemple de Mauvaise Odeur et Correction**

**Mauvaise Odeur : Duplications de Code**

@Test

public void testAddBook() {

// Initialisation des données

LibraryService libraryService = new LibraryService();

Book book = new Book("1984", "George Orwell", "1234567890");

// Ajout d'un livre

libraryService.addBook(book);

// Vérification que le livre a bien été ajouté

assertTrue(libraryService.getBooks().contains(book));

}

@Test

public void testRemoveBook() {

// Initialisation des données

LibraryService libraryService = new LibraryService();

Book book = new Book("1984", "George Orwell", "1234567890");

libraryService.addBook(book);

// Suppression d'un livre

libraryService.removeBook(book);

// Vérification que le livre a bien été supprimé

assertFalse(libraryService.getBooks().contains(book));

}

**Correction : Réduction de la Duplication de Code en Utilisant une Méthode d'Utilitaire**

@BeforeEach

public void setUp() {

libraryService = new LibraryService();

book = new Book("1984", "George Orwell", "1234567890");

}

@Test

public void testAddBook() {

// Ajout d'un livre

libraryService.addBook(book);

// Vérification que le livre a bien été ajouté

assertTrue(libraryService.containsBook(book));

}

@Test

public void testRemoveBook() {

// Préparation : Ajout d'un livre

libraryService.addBook(book);

// Suppression d'un livre

libraryService.removeBook(book);

// Vérification que le livre a bien été supprimé

assertFalse(libraryService.containsBook(book));

}

**Conclusion**

Ce TP vous permettra de pratiquer l'identification et la correction des mauvaises odeurs dans un projet de tests pour une application de gestion de bibliothèque. En repérant et en corrigeant les mauvaises odeurs telles que les duplications de code, les assertions complexes et les noms de tests peu descriptifs, vous pouvez améliorer la qualité et la maintenabilité du code de test.

**TP 20 : Appliquer les Techniques de Refactoring sur un Code Existant**

**Objectif**

L'objectif de ce TP est de pratiquer l'application des techniques de refactoring sur un code de test existant pour une application de gestion de bibliothèque. Vous allez examiner un fichier de tests existant, identifier les opportunités de refactoring, et appliquer des modifications pour améliorer la lisibilité, la maintenabilité et la robustesse du code.

**Contexte**

Vous travaillez sur un projet de tests pour une application de gestion de bibliothèque. Le fichier de tests contient du code qui peut être amélioré en termes de lisibilité, de duplication de code ou de complexité excessive. Vous allez appliquer des techniques de refactoring pour rendre le code de test plus efficace et plus facile à maintenir.

Voici quelques exemples de codes de test qui pourraient nécessiter du refactoring :

**1. Duplications de Code**

@Test

public void testAddBook() {

// Initialisation des données

LibraryService libraryService = new LibraryService();

Book book = new Book("1984", "George Orwell", "1234567890");

// Ajout d'un livre

libraryService.addBook(book);

// Vérification que le livre a bien été ajouté

assertTrue(libraryService.getBooks().contains(book));

}

@Test

public void testRemoveBook() {

// Initialisation des données

LibraryService libraryService = new LibraryService();

Book book = new Book("1984", "George Orwell", "1234567890");

libraryService.addBook(book);

// Suppression d'un livre

libraryService.removeBook(book);

// Vérification que le livre a bien été supprimé

assertFalse(libraryService.getBooks().contains(book));

}

**2. Assertions Complexes**

@Test

public void testCalculateTotal() {

ShoppingCart cart = new ShoppingCart();

cart.addItem(new Item("Product1", 10));

cart.addItem(new Item("Product2", 20));

double total = cart.calculateTotal();

// Assertion complexe

assertEquals(30, total - 0.01);

}

**3. Noms de Tests Peu Descriptifs**

@Test

public void testMethod1() {

// Test code...

}

@Test

public void testMethod2() {

// Test code...

}

**4. Utilisation Excessive de Commentaires**

@Test

public void testAddBook() {

// Création d'une instance de la classe de service

LibraryService libraryService = new LibraryService();

// Création d'un livre

Book book = new Book("1984", "George Orwell", "1234567890");

// Appel de la méthode à tester

libraryService.addBook(book);

// Vérification que le livre a été ajouté à la bibliothèque

assertTrue(libraryService.getBooks().contains(book));

}

**5. Tests Fragiles ou Sensibles aux Données**

@Test

public void testCalculateDiscount() {

ShoppingCart cart = new ShoppingCart();

cart.addItem(new Item("Product1", 100));

// Remise de 10% attendue

double discount = cart.calculateDiscount();

// Assertion sensible aux données

assertEquals(10, discount);

}

Ces exemples montrent des opportunités de refactoring qui peuvent améliorer la qualité, la lisibilité et la maintenabilité du code de test.

**Instructions**

1. Examinez attentivement le fichier de tests existant pour l'application de gestion de bibliothèque.
2. Identifiez les parties du code qui peuvent être améliorées en termes de lisibilité, de duplication de code ou de complexité excessive.
3. Appliquez des techniques de refactoring telles que l'extraction de méthodes, la réduction de la duplication de code, la simplification des assertions, etc.
4. Assurez-vous que les modifications apportées améliorent la lisibilité, la maintenabilité et la robustesse du code de test.
5. Vérifiez que les tests refactorisés restent fonctionnellement équivalents aux tests initiaux et qu'ils couvrent toujours les mêmes cas de test.

**Conclusion**

Ce TP vous permettra de pratiquer l'application des techniques de refactoring sur un code de test existant pour une application de gestion de bibliothèque. En repérant et en corrigeant les problèmes de lisibilité, de duplication de code et de complexité excessive, vous pouvez améliorer la qualité et la maintenabilité du code de test.

**TP 21 : Discussion sur des Exemples Concrets de Design Patterns**

**Objectif**

L'objectif de ce TP est de discuter des exemples concrets de design patterns applicables dans le contexte de développement logiciel, en mettant particulièrement l'accent sur leur utilisation dans le développement de tests unitaires et de tests d'intégration pour une application de gestion de bibliothèque.

**Instructions**

1. Présentez brièvement ce qu'est un design pattern et son utilité dans le développement logiciel.
2. Choisissez quelques design patterns couramment utilisés dans le développement de logiciels et discutez de leur pertinence dans le contexte des tests unitaires et des tests d'intégration.
3. Pour chaque design pattern choisi, fournissez un exemple concret de son utilisation dans le développement de tests pour une application de gestion de bibliothèque.
4. Encouragez les participants à poser des questions et à partager leurs expériences concernant l'utilisation de design patterns dans le développement de tests.

**Exemples de Design Patterns à Discuter**

1. **Pattern de Fabrique (Factory Pattern)** : Utilisé pour créer des objets sans spécifier la classe exacte de l'objet qui sera créé. Par exemple, une fabrique de livres pourrait être utilisée pour créer différents types de livres (roman, essai, biographie, etc.) en fonction des paramètres fournis.
2. **Pattern Observateur (Observer Pattern)** : Utilisé pour mettre en œuvre la communication entre les objets dans un modèle de publication-abonnement. Par exemple, un observateur pourrait être utilisé pour notifier les utilisateurs lorsque de nouveaux livres sont ajoutés à la bibliothèque.
3. **Pattern Singleton (Singleton Pattern)** : Utilisé pour garantir qu'une classe n'a qu'une seule instance et fournir un point d'accès global à cette instance. Par exemple, un gestionnaire de connexion à la base de données pourrait être implémenté en tant que singleton pour s'assurer qu'une seule connexion à la base de données est utilisée par toute l'application.
4. **Pattern Stratégie (Strategy Pattern)** : Utilisé pour définir une famille d'algorithmes, encapsuler chacun d'eux et les rendre interchangeables. Par exemple, une stratégie pourrait être utilisée pour définir différentes méthodes de calcul de frais de retard pour les livres empruntés.

**Conclusion**

Ce TP vous permettra d'explorer et de discuter des exemples concrets de design patterns applicables dans le contexte du développement de tests pour une application de gestion de bibliothèque. En comprenant comment ces design patterns peuvent être utilisés pour résoudre des problèmes spécifiques de conception et de développement, vous serez mieux équipé pour écrire des tests robustes et maintenables.

**TP 22 : Implémenter et Refactoriser une Fonctionnalité en Utilisant des Design Patterns**

**Objectif**

L'objectif de ce TP est de mettre en pratique l'implémentation et le refactoring d'une fonctionnalité dans une application de gestion de bibliothèque en utilisant des design patterns. Vous allez choisir une fonctionnalité spécifique, l'implémenter en utilisant un design pattern approprié, puis refactoriser le code pour améliorer sa qualité et sa maintenabilité.

**Instructions**

1. Choisissez une fonctionnalité spécifique de l'application de gestion de bibliothèque à implémenter ou à améliorer à l'aide d'un design pattern.
2. Identifiez le design pattern le plus approprié pour cette fonctionnalité en fonction des besoins de conception et des contraintes du projet.
3. Implémentez la fonctionnalité en utilisant le design pattern choisi. Assurez-vous que le code est fonctionnel et respecte les principes de conception.
4. Une fois la fonctionnalité implémentée, identifiez les opportunités de refactoring pour améliorer la qualité du code. Cela peut inclure la réduction de la duplication de code, l'amélioration de la lisibilité du code, etc.
5. Refactorisez le code en utilisant des techniques appropriées pour rendre le code plus propre, plus modulaire et plus maintenable.
6. Assurez-vous que les tests unitaires couvrent correctement la fonctionnalité implémentée et refactorisée.
7. Documentez les changements apportés et les décisions de conception prises pendant le processus d'implémentation et de refactoring.

**Exemple de Fonctionnalité et Design Pattern**

**Fonctionnalité** : Gérer les emprunts de livres dans la bibliothèque.

**Design Pattern** : Pattern Stratégie (Strategy Pattern)

**Conclusion**

Ce TP vous permettra de mettre en pratique l'utilisation des design patterns dans le développement d'une application de gestion de bibliothèque. En choisissant et en appliquant judicieusement les design patterns, vous pourrez améliorer la conception et la qualité de votre code, ce qui conduira à des applications plus robustes et plus faciles à maintenir.