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

**Rapport d'Analyse**

**Description du Projet**

Le projet concerne une application de gestion de bibliothèque qui vise à faciliter la gestion des livres, des membres et des emprunts dans une bibliothèque. Il comprend des fonctionnalités telles que l'ajout, la modification et la suppression de livres et de membres, ainsi que la gestion des emprunts et des retours.

**Phases de Tests Identifiées**

| **Phase de Développement** | **Phase de Test** | **Types de Tests** | **Outils Utilisés** | **Impact sur la Qualité** |
| --- | --- | --- | --- | --- |
| Conception | Créer des scénarios de test  Créer des critères d’acceptation | N/A | N/A | N/A |
| Développement | Tests Unitaires | Tests de fonctions unitaires | JUnit, Mockito | Identification rapide des bugs |
| Tests | Tests d'Intégration | Tests des interactions entre modules | JUnit, Spring Test | Détection des problèmes d'intégration |
| Tests | Tests de Système | Tests end-to-end | Selenium, Postman | Vérification du comportement global |
| Déploiement | Tests d'Acceptation | Tests de validation client | Selenium, Cucumber | Validation des exigences client |
| Maintenance | Régression, Patch Tests | Tests de régression | JUnit, Selenium | Maintien de la stabilité |

**Discussion sur l'Importance des Phases de Tests**

Les tests sont essentiels à chaque étape du cycle de vie du développement logiciel pour assurer la qualité du produit final. Les phases de tests identifiées dans ce projet contribuent à différents aspects de la qualité logicielle :

1. **Tests Unitaires** :
   * Effectués pendant la phase de développement, ces tests permettent de valider le comportement des fonctions et des méthodes individuelles. Ils sont cruciaux pour identifier rapidement les bugs au niveau du code source et pour garantir la robustesse des composants logiciels.
2. **Tests d'Intégration** :
   * Ces tests évaluent la façon dont les différents modules ou composants d'une application fonctionnent ensemble. Ils permettent de détecter les erreurs d'intégration entre les modules et de garantir le bon fonctionnement de l'application dans son ensemble.
3. **Tests de Système** :
   * Ces tests vérifient le comportement global de l'application dans un environnement similaire à celui de production. Ils permettent de valider les fonctionnalités clés de l'application et de s'assurer qu'elle répond aux exigences spécifiées.
4. **Tests d'Acceptation** :
   * Ces tests valident que l'application répond aux exigences spécifiques de l'utilisateur final. Ils sont essentiels pour garantir la satisfaction du client et la conformité de l'application aux spécifications fonctionnelles.
5. **Tests de Régression** :
   * Effectués pendant la phase de maintenance, ces tests vérifient que les modifications apportées à l'application n'ont pas introduit de nouveaux bugs ou affecté les fonctionnalités existantes. Ils contribuent à maintenir la stabilité et la fiabilité de l'application au fil du temps.

**Conclusion**

En conclusion, les phases de tests jouent un rôle crucial dans le développement d'un logiciel de qualité. En identifiant et en comprenant les différentes phases de tests dans le cycle de vie du développement logiciel, nous pouvons garantir la qualité, la fiabilité et la performance de l'application de gestion de bibliothèque.

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

Dans le contexte de votre application de gestion de bibliothèque, voici une liste de tests que vous pourriez réaliser, classés dans les différentes catégories. Pour chaque test, une justification est fournie :

**1. Tests Unitaires**

1. **testAddBook()** :
   * **Catégorie** : Tests Unitaires
   * **Justification** : Ce test vérifie la fonctionnalité d'ajout de livres dans la base de données en isolant la méthode concernée.
2. **testUpdateMemberInfo()** :
   * **Catégorie** : Tests Unitaires
   * **Justification** : Ce test vérifie la mise à jour des informations des membres dans la base de données.
3. **testCalculateLateFees()** :
   * **Catégorie** : Tests Unitaires
   * **Justification** : Ce test vérifie le calcul des pénalités pour retards en isolant la fonction de calcul.

**2. Tests d'Intégration**

1. **testBookBorrowingFlow()** :
   * **Catégorie** : Tests d'Intégration
   * **Justification** : Ce test vérifie l'intégration des composants liés à l'emprunt de livres (gestion des membres, des livres et des emprunts).
2. **testReturnBookFlow()** :
   * **Catégorie** : Tests d'Intégration
   * **Justification** : Ce test s'assure que le retour de livres met à jour correctement la base de données et les statuts des livres.

**3. Tests de Système**

1. **testSearchBooks()** :
   * **Catégorie** : Tests de Système
   * **Justification** : Ce test vérifie que la fonctionnalité de recherche de livres fonctionne correctement à travers l'interface utilisateur.
2. **testGenerateReports()** :
   * **Catégorie** : Tests de Système
   * **Justification** : Ce test s'assure que les rapports générés sur les livres empruntés et les membres actifs sont corrects et complets.

**4. Tests d'Acceptation**

1. **testMemberRegistration()** :
   * **Catégorie** : Tests d'Acceptation
   * **Justification** : Ce test valide que le processus d'inscription des nouveaux membres respecte les spécifications et est fonctionnel de bout en bout.
2. **testBookDeletion()** :
   * **Catégorie** : Tests d'Acceptation
   * **Justification** : Ce test s'assure que la suppression d'un livre fonctionne correctement et que toutes les dépendances sont gérées.

**5. Tests de Régression**

1. **testPreviousFeaturesAfterUpdate()** :
   * **Catégorie** : Tests de Régression
   * **Justification** : Ce test vérifie que les fonctionnalités précédemment implémentées (comme l'ajout de livres, l'emprunt, etc.) fonctionnent toujours après une mise à jour ou un changement de code.

**Tableau Récapitulatif**

| **Test** | **Catégorie** | **Justification** |
| --- | --- | --- |
| testAddBook() | Tests Unitaires | Vérifie l'ajout de livres dans la base de données en isolant la méthode concernée. |
| testUpdateMemberInfo() | Tests Unitaires | Vérifie la mise à jour des informations des membres dans la base de données. |
| testCalculateLateFees() | Tests Unitaires | Vérifie le calcul des pénalités pour retards en isolant la fonction de calcul. |
| testBookBorrowingFlow() | Tests d'Intégration | Vérifie l'intégration des composants liés à l'emprunt de livres. |
| testReturnBookFlow() | Tests d'Intégration | Vérifie que le retour de livres met à jour correctement la base de données et les statuts des livres. |
| testSearchBooks() | Tests de Système | Vérifie que la fonctionnalité de recherche de livres fonctionne correctement à travers l'interface utilisateur. |
| testGenerateReports() | Tests de Système | S'assure que les rapports générés sont corrects et complets. |
| testMemberRegistration() | Tests d'Acceptation | Valide que le processus d'inscription des nouveaux membres respecte les spécifications et est fonctionnel. |
| testBookDeletion() | Tests d'Acceptation | S'assure que la suppression d'un livre fonctionne correctement et que toutes les dépendances sont gérées. |
| testPreviousFeaturesAfterUpdate() | Tests de Régression | Vérifie que les fonctionnalités précédemment implémentées fonctionnent toujours après une mise à jour. |

Ces tests couvrent les différentes fonctionnalités et aspects de votre application, et leur classification permet de s'assurer que chaque type de test est correctement ciblé et justifié.

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

**Étape 1 : Préparation de l'Environnement**

Assurez-vous que vous avez un projet Java configuré avec Maven ou Gradle et que les dépendances JUnit et Mockito sont ajoutées. Voici un exemple de fichier pom.xml pour Maven :

<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-service</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>3.11.2</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

**Étape 2 : Implémentation de la Classe à Tester**

Voici la classe LibraryService :

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);

}

}

}

**Dépendances**

public interface BookRepository {

void save(Book book);

Book findById(String bookId);

}

public interface UserRepository {

User findById(String userId);

}

public class Book {

private String id;

private String title;

private boolean available;

private User borrower;

public Book(String id, String title, boolean available) {

this.id = id;

this.title = title;

this.available = available;

}

public boolean isAvailable() {

return available;

}

public void setAvailable(boolean available) {

this.available = available;

}

public void setBorrower(User borrower) {

this.borrower = borrower;

}

public User getBorrower() {

return borrower;

}

}

public class User {

private String id;

private String name;

public User(String id, String name) {

this.id = id;

this.name = name;

}

}

**Étape 3 : Implémentation des Tests avec Mocks**

Voici les tests unitaires utilisant 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 {

@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);

}

}

**Explications Détaillées**

1. **Préparation de l'Environnement** :
   * Assurez-vous que JUnit et Mockito sont disponibles en tant que dépendances dans votre projet.
2. **Implémentation de la Classe LibraryService** :
   * La classe LibraryService contient deux méthodes principales : addBook et borrowBook.
   * La méthode addBook ajoute un livre en utilisant le BookRepository.
   * La méthode borrowBook permet à un utilisateur d'emprunter un livre, en vérifiant que le livre est disponible et que l'utilisateur existe, puis en mettant à jour le livre pour le marquer comme emprunté.
3. **Implémentation des Tests avec Mocks** :
   * Les tests utilisent Mockito pour créer des mocks des interfaces BookRepository et UserRepository.
   * @Mock est utilisé pour indiquer que les instances de BookRepository et UserRepository doivent être des mocks.
   * @InjectMocks indique que les mocks doivent être injectés dans l'instance de LibraryService.
   * MockitoAnnotations.openMocks(this) initialise les mocks.
   * verify est utilisé pour vérifier que certaines méthodes ont été appelées sur les mocks.
   * when et thenReturn sont utilisés pour définir les comportements des mocks.

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

**Étape 1 : Compréhension des Exigences**

La classe Calculator doit fournir les fonctionnalités suivantes :

* Addition (add)
* Soustraction (subtract)
* Multiplication (multiply)
* Division (divide)

**Étape 2 : Écrire les Tests Unitaires**

Commencez par écrire les tests unitaires pour chaque méthode.

**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);

}

@Test(expected = IllegalArgumentException.class)

public void testDivideByZero() {

Calculator calculator = new Calculator();

calculator.divide(6, 0);

}

}

**Explications**

1. **Test de l'Addition**
   * Méthode testAdd vérifie que l'addition de 2 et 3 donne 5.
2. **Test de la Soustraction**
   * Méthode testSubtract vérifie que la soustraction de 3 de 5 donne 2.
3. **Test de la Multiplication**
   * Méthode testMultiply vérifie que la multiplication de 2 et 3 donne 6.
4. **Test de la Division**
   * Méthode testDivide vérifie que la division de 6 par 3 donne 2.
5. **Test de la Division par Zéro**
   * Méthode testDivideByZero vérifie que la division par zéro lance une IllegalArgumentException.

**Étape 3 : Implémentation de la Classe Calculator**

Implémentez la classe Calculator pour faire passer les tests.

**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;

}

}

**Explications**

1. **Addition**
   * La méthode add prend deux entiers en paramètre et retourne leur somme.
2. **Soustraction**
   * La méthode subtract prend deux entiers en paramètre et retourne la différence.
3. **Multiplication**
   * La méthode multiply prend deux entiers en paramètre et retourne leur produit.
4. **Division**
   * La méthode divide prend deux entiers en paramètre et retourne le quotient.
   * Si le dénominateur est zéro, elle lance une IllegalArgumentException.

**Étape 4 : Vérification et Refactorisation**

Vérifiez que tous les tests passent.

**Vérification des Tests**

mvn test

Tous les tests doivent passer sans erreurs.

**Refactorisation**

Étant donné que le code est déjà simple et clair, il n'y a pas de refactorisation supplémentaire nécessaire dans cet exemple.

**Conclusion**

* Vous avez écrit des tests unitaires pour chaque méthode de la classe Calculator en suivant les étapes de TDD.
* Vous avez implémenté la classe Calculator pour faire passer les tests.
* Vous avez vérifié que tous les tests passent et refactorisé le code si nécessaire.

En pratiquant TDD, vous avez appris à écrire des tests avant d'implémenter le code, ce qui permet de s'assurer que le code fonctionne comme prévu dès le début.

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

**Objectif**

L'objectif est de configurer un projet Maven pour une application de gestion de bibliothèque, puis d'écrire des tests unitaires basiques pour une classe de service en utilisant JUnit et Mockito.

**Instructions et Correction**

1. **Création du Projet Maven**
   * Utilisez votre IDE (Eclipse, IntelliJ, etc.) pour créer un nouveau projet Maven.
   * Donnez-lui un nom, par exemple, library-management.
2. **Configuration du fichier pom.xml**

Ajoutez les dépendances suivantes au fichier pom.xml pour inclure JUnit et Mockito :

<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**

**Classe Book :**

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;

}

}

**Interface BookRepository :**

public interface BookRepository {

Book findById(int id);

}

Classe LibraryService :

java

Copier le code

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**

**Classe de test LibraryServiceTest :**

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 org.mockito.MockitoAnnotations;

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**
   * **Objectif** : Vérifier que la méthode retourne bien un livre avec les propriétés attendues.
   * **Mise en œuvre** : Utilisation de Mockito pour simuler BookRepository. Le mock retourne un objet Book lorsque la méthode findById(1) est appelée.
   * **Assertions** : Vérifiez que l'objet retourné n'est pas nul, que son id, title et author sont corrects. Utilisez verify pour confirmer que findById(1) a été appelée.

@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);

}

1. **Test de getBookById quand le livre n'est pas trouvé**
   * **Objectif** : Vérifier que la méthode retourne null quand le livre n'est pas trouvé.
   * **Mise en œuvre** : Utilisation de Mockito pour simuler BookRepository. Le mock retourne null lorsque la méthode findById(2) est appelée.
   * **Assertions** : Vérifiez que l'objet retourné est null. Utilisez verify pour confirmer que findById(2) a été appelée.

@Test

public void testGetBookById\_NotFound() {

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

Book result = libraryService.getBookById(2);

assertNull(result);

verify(bookRepository).findById(2);

}

**Correction 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.

**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);

}

@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());

}

}

**Étapes de Réorganisation des Tests**

1. **Analyse des Tests Existants**
   * Le test testBookFound et testBookFoundAgain contiennent des duplications.
   * Les noms des tests ne sont pas suffisamment descriptifs.
   * Les interactions avec les mocks ne sont pas vérifiées.
2. **Identification des Problèmes**
   * Duplicated Code (Code dupliqué)
   * Poor Naming Conventions (Mauvaises conventions de nommage)
   * Lack of Proper Assertions (Manque de vérification des interactions avec les mocks)
3. **Réorganisation des Tests**
   * Suppression du code dupliqué.
   * Renommage des tests pour des noms plus descriptifs.
   * Ajout de vérifications des interactions avec les mocks.

**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() {

// Arrange

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

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

// Act

Book result = libraryService.getBookById(1);

// Assert

assertNotNull(result);

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

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

verify(bookRepository).findById(1);

}

@Test

public void testGetBookById\_BookNotFound() {

// Arrange

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

// Act

Book result = libraryService.getBookById(2);

// Assert

assertNull(result);

verify(bookRepository).findById(2);

}

}

**Explication des Changements**

1. **Suppression du Code Dupliqué**
   * Les tests testBookFound et testBookFoundAgain ont été fusionnés en un seul test testGetBookById\_BookFound pour éviter la duplication de code.
2. **Noms de Méthodes Descriptifs**
   * Les noms des méthodes de test ont été modifiés pour être plus descriptifs : testGetBookById\_BookFound et testGetBookById\_BookNotFound. Cela permet de comprendre immédiatement ce que chaque test vérifie.
3. **Vérification des Interactions avec les Mocks**
   * Ajout de vérifications avec verify(bookRepository).findById(1) et verify(bookRepository).findById(2) pour s'assurer que les mocks sont appelés comme prévu. Cela garantit que les interactions avec les dépendances sont correctes.
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())). Cela permet de s'assurer que l'objet retourné est correct et que les modifications n'ont pas cassé le code.
5. **Organisation du Code de Test**
   * Utilisation d'un schéma Arrange-Act-Assert (AAA) dans les tests pour une meilleure clarté et lisibilité. Cela aide à structurer les tests de manière logique et cohérente.

En suivant ces étapes de réorganisation, les tests sont désormais mieux structurés, plus clairs, et plus faciles à maintenir. Ils répondent aux bonnes pratiques de tests unitaires, améliorant ainsi la qualité et la maintenabilité du code.

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

**Écrire un Test qui Échoue**

**Code Initial des Tests :**

@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());

}

**Implémenter la Fonctionnalité**

**Code de la Méthode searchBooksByTitle dans LibraryService :**

public class LibraryService {

private BookRepository bookRepository;

public LibraryService(BookRepository bookRepository) {

this.bookRepository = bookRepository;

}

public List<Book> searchBooksByTitle(String title) {

List<Book> allBooks = bookRepository.findAll();

return allBooks.stream()

.filter(book -> book.getTitle().toLowerCase().contains(title.toLowerCase()))

.collect(Collectors.toList());

}

}

**Refactoriser le Code**

* Vérifiez si le code peut être simplifié ou optimisé.
* Assurez-vous que le nommage des variables et des méthodes est clair et conforme aux conventions.
* Ajoutez des commentaires si nécessaire pour expliquer des parties complexes du code.

**Rapport**

**Étapes Suivies :**

1. Écrit des tests qui échouent pour valider le comportement attendu de la méthode searchBooksByTitle.
2. Implémenté la méthode searchBooksByTitle dans la classe LibraryService.
3. Validé que les tests passent après l'implémentation.
4. Refactorisé le code pour améliorer sa lisibilité et sa maintenabilité.

**Choix de Conception :**

* Utilisation de stream() et filter() pour une recherche efficace et lisible.
* Conversion du titre en minuscules pour rendre la recherche insensible à la casse.
* Tests exhaustifs pour vérifier les cas où des livres sont trouvés et non trouvés.

En suivant ce processus, vous aurez appliqué le cycle TDD de manière pratique et acquis une meilleure compréhension de son utilité dans le développement logiciel.

**Correction 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.

Exemple de code pour un test unitaire avec JUnit :

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());

}

}

1. **Implémentation des Tests (10 minutes)**
   * Les participants implémentent les tests unitaires dans le projet de l'application de gestion de bibliothèque en utilisant Maven.

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. **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.
2. **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.

**Correction 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. Les participants écriront des tests unitaires qui échouent d'abord, puis implémenteront 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.

Exemple de refactorisation du code pour utiliser une boucle forEach :

public class Library {

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

public void addBook(Book book) {

books.add(book);

}

public void markBookAsFavorite(String isbn) {

books.stream()

.filter(book -> book.getIsbn().equals(isbn))

.findFirst()

.ifPresent(book -> book.setFavorite(true));

}

}

1. **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.

**Correction TP 10 : Refactorisation pour Utiliser la Composition plutôt que l'Héritage dans une Application de Gestion de Bibliothèque**

**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. Les participants apprendront 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.

**Analyse de la Classe Book**

**Code avant refactorisation**

Voici un exemple de ce à quoi la classe Book et la classe Item pourraient ressembler avant la refactorisation :

public class Item {

private String title;

private String description;

// Getters and setters for title and description

public String getTitle() {

return title;

}

public void setTitle(String title) {

this.title = title;

}

public String getDescription() {

return description;

}

public void setDescription(String description) {

this.description = description;

}

}

public class Book extends Item {

private String author;

private String isbn;

// Getters and setters for author and isbn

public String getAuthor() {

return author;

}

public void setAuthor(String author) {

this.author = author;

}

public String getIsbn() {

return isbn;

}

public void setIsbn(String isbn) {

this.isbn = isbn;

}

}

**Refactorisation de la Classe Book**

**Étape 1 : Extraire les attributs et méthodes communs dans une classe ItemDetails**

Nous allons extraire les attributs et méthodes communs à une nouvelle classe ItemDetails et utiliser la composition pour inclure ItemDetails dans Book.

public class ItemDetails {

private String title;

private String description;

// Getters and setters for title and description

public String getTitle() {

return title;

}

public void setTitle(String title) {

this.title = title;

}

public String getDescription() {

return description;

}

public void setDescription(String description) {

this.description = description;

}

}

**Étape 2 : Modifier la classe Book pour utiliser la composition**

Voici la classe Book refactorisée pour utiliser la composition :

public class Book {

private String author;

private String isbn;

private ItemDetails itemDetails;

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

this.itemDetails = new ItemDetails();

this.itemDetails.setTitle(title);

this.itemDetails.setDescription(description);

this.author = author;

this.isbn = isbn;

}

// Getters and setters for author and isbn

public String getAuthor() {

return author;

}

public void setAuthor(String author) {

this.author = author;

}

public String getIsbn() {

return isbn;

}

public void setIsbn(String isbn) {

this.isbn = isbn;

}

// Delegating methods to itemDetails

public String getTitle() {

return itemDetails.getTitle();

}

public void setTitle(String title) {

itemDetails.setTitle(title);

}

public String getDescription() {

return itemDetails.getDescription();

}

public void setDescription(String description) {

itemDetails.setDescription(description);

}

}

**Justification des Choix de Refactorisation**

1. **Respect du principe de substitution de Liskov (LSP) :** En séparant les attributs et méthodes communs dans une classe distincte (ItemDetails), nous garantissons que la classe Book n'étend pas inutilement une classe parent. Cela permet de réduire les contraintes sur les relations d'héritage et de respecter le principe LSP.
2. **Flexibilité accrue :** En utilisant la composition, nous pouvons facilement modifier ou étendre les fonctionnalités de ItemDetails sans affecter directement les classes qui les utilisent (Book). Cela rend le code plus flexible et maintenable.
3. **Responsabilité unique :** Chaque classe a maintenant une responsabilité claire et unique. ItemDetails est responsable de gérer les attributs communs, tandis que Book gère les attributs spécifiques aux livres.
4. **Réutilisabilité :** La classe ItemDetails peut être réutilisée par d'autres types d'items dans la bibliothèque, évitant ainsi la duplication de code.

**Conclusion**

Ce TP a permis de comprendre et de mettre en pratique les principes de conception SOLID en refactorisant une classe pour utiliser la composition plutôt que l'héritage. Cette approche améliore la flexibilité, la maintenabilité et la réutilisabilité du code dans le contexte spécifique d'une application de gestion de bibliothèque.

**Correction TP 11 : Refactorisation de Code Statique pour le Rendre plus Testable**

**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. Nous allons refactoriser la classe LibraryUtils pour éliminer les méthodes statiques et les rendre plus modulaires et testables.

**Contexte**

Nous avons une classe LibraryUtils qui contient plusieurs méthodes statiques utilisées pour effectuer diverses opérations, telles que la gestion des dates de retour des livres et la génération de rapports. Nous allons refactoriser cette classe pour utiliser des objets et l'injection de dépendances, afin de rendre le code plus testable.

**Étape 1 : Analyse du Code Statique**

Voici le code de la classe LibraryUtils avant la 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());

}

}

}

**Étape 2 : Refactorisation de la Classe LibraryUtils**

**Création de Classes Non Statiques**

Nous allons créer deux nouvelles classes, DateProvider et ReportPrinter, et modifier LibraryUtils pour utiliser ces classes via la composition.

**DateProvider.java**

public class DateProvider {

public Date calculateReturnDate(Date borrowDate, int loanPeriodDays) {

Calendar calendar = Calendar.getInstance();

calendar.setTime(borrowDate);

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

return calendar.getTime();

}

}

**ReportPrinter.java**

import java.util.List;

public class ReportPrinter {

public void print(List<Book> books) {

for (Book book : books) {

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

}

}

}

**LibraryUtils.java**

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) {

return dateProvider.calculateReturnDate(borrowDate, loanPeriodDays);

}

public void printReport(List<Book> books) {

reportPrinter.print(books);

}

}

**Étape 3 : Écriture de Tests Unitaires**

Nous allons écrire des tests unitaires pour les méthodes refactorisées en utilisant JUnit et Mockito.

**LibraryUtilsTest.java**

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 java.util.Arrays;

import java.util.Date;

import java.util.List;

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 \* 24L \* 60 \* 60 \* 1000);

when(mockDateProvider.calculateReturnDate(borrowDate, loanPeriodDays)).thenReturn(expectedReturnDate);

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);

}

}

**Justification des Choix de Refactorisation**

1. **Réduction de la Couplage** : En utilisant des objets pour gérer les opérations, nous réduisons le couplage entre les différentes parties de l'application. Les classes ne dépendent plus directement de méthodes statiques, ce qui facilite les tests et la maintenance.
2. **Injection de Dépendances** : En injectant des dépendances via des constructeurs, nous permettons aux classes d'être facilement testées en isolant leurs dépendances. Cela facilite l'utilisation de mocks et de stubs dans les tests unitaires.
3. **Responsabilité Unique** : Chaque classe a une responsabilité claire et unique. DateProvider gère les calculs de dates, ReportPrinter gère l'impression des rapports, et LibraryUtils coordonne les deux.

**Conclusion**

Ce TP a permis 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. En utilisant la composition et l'injection de dépendances, nous avons rendu le code plus flexible et plus facile à tester.

**Correction TP 12 : Implémentation de l'Inversion de Dépendances**

**Objectif**

L'objectif de ce TP est de comprendre et d'implémenter le principe de l'inversion de dépendances (DIP) dans un projet existant. Nous allons refactoriser le code pour inverser les dépendances, rendant ainsi le système plus modulaire et plus facile à tester.

**Contexte**

Nous travaillons sur une application de gestion de bibliothèque. Actuellement, certaines classes dépendent directement de classes concrètes, ce qui complique les tests et la maintenance du code. Nous allons refactoriser ces classes pour appliquer le principe de l'inversion de dépendances en utilisant des interfaces et l'injection de dépendances.

**Étape 1 : Analyse du Code Existant**

Nous avons identifié que la classe LibraryService dépend directement de la classe concrète BookRepository. Voici le code avant refactorisation :

**LibraryService.java**

public class LibraryService {

private BookRepository bookRepository = new BookRepository();

public List<Book> getAllBooks() {

return bookRepository.findAllBooks();

}

}

**BookRepository.java**

public class BookRepository {

public List<Book> findAllBooks() {

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

}

}

**Étape 2 : Création d'Interfaces**

Nous créons une interface pour BookRepository.

**BookRepository.java**

public interface BookRepository {

List<Book> findAllBooks();

}

**Étape 3 : Implémentation de l'Interface**

Nous modifions BookRepository pour implémenter l'interface BookRepository.

**BookRepositoryImpl.java**

public class BookRepositoryImpl implements BookRepository {

public List<Book> findAllBooks() {

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

}

}

**Étape 4 : Refactorisation de la Classe Dépendante**

Nous modifions LibraryService pour qu'elle dépende de l'interface BookRepository plutôt que de la classe concrète.

**LibraryService.java**

public class LibraryService {

private BookRepository bookRepository;

public LibraryService(BookRepository bookRepository) {

this.bookRepository = bookRepository;

}

public List<Book> getAllBooks() {

return bookRepository.findAllBooks();

}

}

**Étape 5 : Écriture de Tests Unitaires**

Nous écrivons des tests unitaires pour LibraryService en utilisant des mocks pour tester les interactions avec BookRepository.

**LibraryServiceTest.java**

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 java.util.Arrays;

import java.util.List;

public class LibraryServiceTest {

private LibraryService libraryService;

private BookRepository mockBookRepository;

@BeforeEach

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();

}

}

**Justification des Choix de Refactorisation**

1. **Réduction du Couplage** : En utilisant des interfaces, nous réduisons le couplage entre les différentes parties de l'application. Les classes dépendent des abstractions plutôt que des implémentations concrètes, ce qui facilite les changements et les tests.
2. **Injection de Dépendances** : En injectant les dépendances via des constructeurs, nous permettons aux classes d'être facilement testées en isolant leurs dépendances. Cela facilite l'utilisation de mocks et de stubs dans les tests unitaires.
3. **Flexibilité et Extensibilité** : En utilisant des interfaces, nous pouvons facilement remplacer les implémentations concrètes par d'autres versions, par exemple, pour les tests ou pour changer de source de données (par exemple, de MySQL à MongoDB).

**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, nous avons 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.

**Correction 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. Nous allons écrire des tests pour une application de gestion de bibliothèque en utilisant diverses assertions pour valider le comportement des méthodes.

**Contexte**

Nous travaillons 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.

**Étape 1 : Préparation du Projet**

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**

Nous allons écrire des tests pour la méthode addBook qui ajoute un nouveau livre à la bibliothèque. Nous allons utiliser différentes assertions pour valider :

1. Que le livre a été ajouté correctement
2. Que les informations du livre sont correctes
3. Que le nombre de livres a augmenté

**LibraryServiceTest.java**

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

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import java.util.ArrayList;

import java.util.Date;

import java.util.List;

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);

// Assertion to check that the list is not empty

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

// Assertion to check the size of the list

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

// Assertions to check the details of the book

Book addedBook = libraryService.getBooks().get(0);

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

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

assertEquals("1234567890", addedBook.getIsbn());

}

}

**Étape 3 : Écrire des Tests pour l'Inscription de Nouveaux Membres**

Nous allons écrire des tests pour la méthode registerMember qui inscrit un nouveau membre à la bibliothèque. Nous allons utiliser différentes assertions pour valider :

1. Que le membre a été inscrit correctement
2. Que les informations du membre sont correctes
3. Que le nombre de membres a augmenté

**LibraryServiceTest.java**

@Test

public void testRegisterMember() {

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

libraryService.registerMember(member);

// Assertion to check that the list is not empty

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

// Assertion to check the size of the list

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

// Assertions to check the details of the member

Member registeredMember = libraryService.getMembers().get(0);

assertEquals("John Doe", registeredMember.getName());

assertEquals("12345", registeredMember.getMemberId());

}

}

**Étape 4 : Écrire des Tests pour l'Enregistrement des Emprunts de Livres**

Nous allons écrire des tests pour la méthode borrowBook qui enregistre l'emprunt d'un livre par un membre. Nous allons utiliser différentes assertions pour valider :

1. Que l'emprunt a été enregistré correctement
2. Que le livre n'est plus disponible
3. Que le nombre d'emprunts actifs a augmenté

**LibraryServiceTest.java**

@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);

// Assertion to check that the borrow records list is not empty

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

// Assertion to check the size of the borrow records list

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

// Assertions to check the details of the borrow record

BorrowRecord borrowRecord = libraryService.getBorrowRecords().get(0);

assertEquals("1984", borrowRecord.getBook().getTitle());

assertEquals("John Doe", borrowRecord.getMember().getName());

// Additional assertion to check that the book is considered borrowed

assertTrue(libraryService.getBorrowRecords().stream()

.anyMatch(record -> record.getBook().equals(book)));

}

}

**Étape 5 : Mise à Jour de la Classe LibraryService**

Assurez-vous que la classe LibraryService est correctement définie pour supporter les tests. Voici un exemple de ce à quoi la classe pourrait ressembler :

**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) {

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;

}

}

**Conclusion**

Ce TP vous a permis 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. Vous avez appris à utiliser des assertions pour vérifier l'état des objets, la taille des collections et la correspondance des propriétés spécifiques.

**Correction 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**

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**

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.

**Code : 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());

}

}

**Étape 3 : Écrire des Tests pour l'Inscription de Nouveaux Membres**

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.

**Code : LibraryServiceTest.java**

@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());

}

}

**Étape 4 : Écrire des Tests pour l'Enregistrement des Emprunts de Livres**

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.

**Code : LibraryServiceTest.java**

@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());

}

}

**Classe LibraryService**

Assurez-vous que la classe LibraryService est correctement définie pour supporter les tests. Voici un exemple de ce à quoi la classe pourrait ressembler :

**Code : 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;

}

}

**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.

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

**Préparation du Projet**

Avant de commencer à écrire des tests paramétrés, assurez-vous que votre projet est configuré pour utiliser JUnit 5 et que la dépendance junit-jupiter-params est ajoutée à votre fichier pom.xml.

**Étape 1 : Écriture des Tests Paramétrés pour l'Ajout de Livres**

Dans cette étape, nous allons écrire des tests paramétrés pour la méthode addBook afin de vérifier différentes combinaisons de titres, auteurs et ISBN.

**Code de Test Paramétré pour l'Ajout de Livres**

import org.junit.jupiter.api.BeforeEach;

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

);

}

}

**Étape 2 : Écriture des Tests Paramétrés pour l'Inscription de Membres**

Dans cette étape, nous allons écrire des tests paramétrés pour la méthode registerMember afin de vérifier différentes combinaisons de noms et d'identifiants de membres.

**Code de Test Paramétré pour l'Inscription de Membres**

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 {

// ...

@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 moyen puissant de vérifier différentes combinaisons de données d'entrée et de s'assurer que votre application fonctionne correctement dans une variété de scénarios. En utilisant des sources de méthodes pour fournir des arguments de test, vous pouvez facilement étendre et maintenir vos tests.

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

**Configuration du Projet**

Avant de commencer à écrire des tests utilisant des matchers, assurez-vous que votre projet est configuré pour utiliser un framework de test compatible avec Hamcrest, comme JUnit Jupiter avec Hamcrest. Assurez-vous que les dépendances nécessaires sont ajoutées à votre fichier pom.xml si vous utilisez Maven, ou à votre fichier de configuration de dépendances si vous utilisez un autre gestionnaire de dépendances.

**Écriture des Tests Utilisant des Matchers**

Dans cette partie, nous allons écrire des tests pour les fonctionnalités de l'application de gestion de bibliothèque en utilisant des matchers pour les assertions.

**Test d'Ajout de Nouveaux Livres**

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() {

// Création d'un nouveau livre

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

// Ajout du livre à la bibliothèque

libraryService.addBook(book);

// Vérification que le livre a bien été ajouté à la liste des livres de la bibliothèque

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

}

}

**Test d'Inscription de Nouveaux Membres**

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 testRegisterMember() {

// Création d'un nouveau membre

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

// Inscription du membre à la bibliothèque

libraryService.registerMember(member);

// Vérification que le membre a bien été ajouté à la liste des membres de la bibliothèque

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

}

}

**Test d'Enregistrement d'Emprunts de Livres**

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 testBorrowBook() {

// Création d'un nouveau livre

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

// Création d'un nouveau membre

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

// Ajout du livre à la bibliothèque

libraryService.addBook(book);

// Inscription du membre à la bibliothèque

libraryService.registerMember(member);

// Emprunt du livre par le membre

libraryService.borrowBook(member, book);

// Vérification que l'emprunt a bien été enregistré dans les emprunts de la bibliothèque

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.

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

**Étape 1 : Examen du Fichier de Tests Existant**

Dans cette étape, vous devez examiner le fichier de tests existant pour l'application de gestion de bibliothèque. Identifiez les différentes fonctionnalités testées dans ce fichier et les tests correspondants.

**Étape 2 : Création de Classes de Tests**

Pour chaque fonctionnalité identifiée, créez une classe de tests distincte. Par exemple, vous pouvez créer des classes de tests pour la gestion des livres, la gestion des membres, les emprunts, etc.

**Étape 3 : Déplacement des Tests Pertinents**

Déplacez les tests correspondants dans les classes de tests nouvellement créées. 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

Dans cet exemple, nous avons créé quatre classes de tests distinctes pour chaque fonctionnalité de l'application de gestion de bibliothèque.

**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 d'ajout d'un nouveau livre

}

@Test

public void testUpdateBook() {

// Test de mise à jour des informations d'un livre

}

@Test

public void testRemoveBook() {

// Test de suppression d'un livre

}

@Test

public void testSearchBook() {

// Test de recherche d'un livre

}

}

De manière similaire, vous pouvez créer des classes de tests pour d'autres fonctionnalités telles que la gestion des membres, les emprunts, la génération de rapports, etc.

**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.

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

**Étape 1 : Examen du Fichier de Tests**

Dans cette étape, examinez attentivement le fichier de tests existant pour l'application de gestion de bibliothèque. Identifiez les parties du code qui peuvent être améliorées en termes de lisibilité, de duplication de code ou de complexité excessive.

**Étape 2 : Application des Techniques de Refactoring**

**Extraction de Méthodes**

Identifiez les blocs de code répétitifs dans les tests et extrayez-les dans des méthodes réutilisables.

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 :

java

Copier le code

@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));

}

**Réduction de la Duplication de Code**

Identifiez les parties de code similaires dans différents tests et les facteurs communs pour les rendre plus génériques.

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));

}

@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));

}

Après le refactoring :

java

Copier le code

@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));

}

@Test

public void testRemoveBook() {

// Arrange

LibraryService libraryService = new LibraryService();

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

libraryService.addBook(book);

// Act

libraryService.removeBook(book);

// Assert

assertFalse(libraryService.containsBook(book));

}

**Simplification des Assertions**

Simplifiez les assertions complexes en utilisant des matchers ou des méthodes d'assertion plus expressifs.

**Conclusion**

Ce TP vous a permis 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.

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

**Mauvaise Odeur : Duplications de Code**

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));

}

@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));

}

Après le refactoring :

// Déclaration de variables

private LibraryService libraryService;

private Book book;

@BeforeEach

public void setUp() {

libraryService = new LibraryService();

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

}

@Test

public void testAddAndRemoveBook() {

// Ajout d'un livre

libraryService.addBook(book);

// Vérification que le livre a bien été ajouté

assertTrue(libraryService.containsBook(book));

// Suppression d'un livre

libraryService.removeBook(book);

// Vérification que le livre a bien été supprimé

assertFalse(libraryService.containsBook(book));

}

**Explication :**

* Nous avons extrait les initialisations de données et les étapes de vérification communes dans une méthode setUp() annotée par @BeforeEach, ce qui permet de les réutiliser pour tous les tests de cette classe.
* Nous avons combiné les tests testAddBook() et testRemoveBook() en un seul test testAddAndRemoveBook(), ce qui réduit la duplication de code et simplifie la structure des tests.
* En utilisant une seule méthode de test pour les opérations d'ajout et de suppression de livre, nous réduisons également la dépendance entre les tests et rendons le code plus lisible.

**Conclusion**

En identifiant et en corrigeant les mauvaises odeurs telles que les duplications de code, nous améliorons la lisibilité, la maintenabilité et la robustesse du code de test. Cela rend également la suite de tests plus efficace et plus facile à maintenir à long terme.

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

**1. Duplications de Code**

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));

}

@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));

}

Après le refactoring :

private LibraryService libraryService;

private Book book;

@BeforeEach

public void setUp() {

libraryService = new LibraryService();

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

}

@Test

public void testAddAndRemoveBook() {

// Ajout d'un livre

libraryService.addBook(book);

// Vérification que le livre a bien été ajouté

assertTrue(libraryService.containsBook(book));

// Suppression d'un livre

libraryService.removeBook(book);

// Vérification que le livre a bien été supprimé

assertFalse(libraryService.containsBook(book));

}

**2. Assertions Complexes**

Avant le refactoring :

@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);

}

Après le refactoring :

@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 simplifiée avec un delta acceptable

assertEquals(30, total, 0.01);

}

**3. Noms de Tests Peu Descriptifs**

Avant le refactoring :

@Test

public void testMethod1() {

// Test code...

}

@Test

public void testMethod2() {

// Test code...

}

Après le refactoring :

java

Copier le code

@Test

public void testAddingBookToLibrary() {

// Test code...

}

@Test

public void testRemovingBookFromLibrary() {

// Test code...

}

**4. Utilisation Excessive de Commentaires**

Avant le refactoring :

@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));

}

Après le refactoring :

@Test

public void testAddBookToLibrary() {

// Arrange

LibraryService libraryService = new LibraryService();

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

// Act

libraryService.addBook(book);

// Assert

assertTrue(libraryService.containsBook(book));

}

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

Avant le refactoring :

@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);

}

Après le refactoring :

@Test

public void testCalculateDiscount() {

ShoppingCart cart = new ShoppingCart();

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

// Act

double discount = cart.calculateDiscount();

// Assert

assertEquals(10, discount);

}

Ces exemples montrent comment les techniques de refactoring peuvent améliorer la lisibilité, la maintenabilité et la robustesse du code de test.

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

**Introduction**

Dans cette discussion, nous allons explorer plusieurs design patterns couramment utilisés dans le développement logiciel et discuter de leur application dans le contexte du développement de tests pour une application de gestion de bibliothèque.

**1. Pattern de Fabrique (Factory Pattern)**

Le pattern de fabrique est utilisé pour créer des objets sans spécifier la classe exacte de l'objet qui sera créé. Dans le contexte des tests pour une bibliothèque, un exemple concret de l'utilisation de ce pattern pourrait être une fabrique de livres. Cette fabrique pourrait être utilisée pour créer différents types de livres (romans, essais, biographies, etc.) en fonction des paramètres fournis. Cela permet une création d'objets flexible et modulaire, ce qui peut être utile lors de l'écriture de tests pour différents scénarios de livres.

**2. Pattern Observateur (Observer Pattern)**

Le pattern observateur est utilisé pour mettre en œuvre la communication entre les objets dans un modèle de publication-abonnement. Dans le contexte des tests pour une bibliothèque, un exemple concret de l'utilisation de ce pattern pourrait être un observateur utilisé pour notifier les utilisateurs lorsque de nouveaux livres sont ajoutés à la bibliothèque. Par exemple, un service d'envoi de notifications pourrait être un observateur qui reçoit des notifications chaque fois qu'un nouveau livre est ajouté, et ensuite envoie des notifications aux abonnés.

**3. Pattern Singleton (Singleton Pattern)**

Le pattern singleton est utilisé pour garantir qu'une classe n'a qu'une seule instance et fournir un point d'accès global à cette instance. Dans le contexte des tests pour une bibliothèque, un exemple concret de l'utilisation de ce pattern pourrait être un gestionnaire de connexion à la base de données. En utilisant le pattern singleton, nous nous assurons qu'une seule connexion à la base de données est utilisée par toute l'application, ce qui peut être crucial pour assurer la cohérence des tests et éviter les problèmes de concurrence.

**4. Pattern Stratégie (Strategy Pattern)**

Le pattern stratégie est utilisé pour définir une famille d'algorithmes, encapsuler chacun d'eux et les rendre interchangeables. Dans le contexte des tests pour une bibliothèque, un exemple concret de l'utilisation de ce pattern pourrait être une stratégie utilisée pour définir différentes méthodes de calcul de frais de retard pour les livres empruntés. Par exemple, différentes stratégies pourraient être utilisées pour calculer les frais de retard en fonction de la durée de l'emprunt, du type de livre, etc. En utilisant le pattern stratégie, nous pouvons facilement changer le comportement de calcul des frais de retard sans modifier le code client.

**Conclusion**

En discutant de ces exemples concrets de design patterns et de leur application dans le contexte du développement de tests pour une application de gestion de bibliothèque, nous avons vu comment les design patterns peuvent être des outils puissants pour résoudre des problèmes spécifiques de conception et de développement. En comprenant comment et quand utiliser ces patterns, les développeurs peuvent écrire des tests plus robustes, plus modulaires et plus faciles à maintenir.

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

Dans cet exemple, nous allons implémenter et refactoriser la fonctionnalité de gestion des emprunts de livres dans une bibliothèque en utilisant le design pattern Strategy.

**1. Implémentation de la Fonctionnalité avec le Pattern Stratégie**

Tout d'abord, nous allons implémenter la gestion des emprunts en utilisant différentes stratégies pour le calcul des frais de retard en fonction de la durée de l'emprunt.

**Interface de la Stratégie :**

// Interface pour la stratégie de calcul des frais de retard

public interface LateFeeCalculationStrategy {

double calculateLateFee(int daysLate);

}

**Implémentations des Stratégies :**

// Implémentation de la stratégie de calcul des frais de retard pour les emprunts de courte durée

public class ShortTermLateFeeCalculationStrategy implements LateFeeCalculationStrategy {

@Override

public double calculateLateFee(int daysLate) {

return daysLate \* 0.50; // 0.50$ par jour de retard

}

}

// Implémentation de la stratégie de calcul des frais de retard pour les emprunts de longue durée

public class LongTermLateFeeCalculationStrategy implements LateFeeCalculationStrategy {

@Override

public double calculateLateFee(int daysLate) {

return daysLate \* 1.00; // 1.00$ par jour de retard

}

}

**Classe Utilisant le Pattern Stratégie :**

// Classe de gestion des emprunts

public class LoanManager {

private LateFeeCalculationStrategy lateFeeCalculationStrategy;

// Setter pour injecter la stratégie

public void setLateFeeCalculationStrategy(LateFeeCalculationStrategy lateFeeCalculationStrategy) {

this.lateFeeCalculationStrategy = lateFeeCalculationStrategy;

}

// Méthode pour calculer les frais de retard en fonction de la durée de l'emprunt

public double calculateLateFee(int daysLate) {

return lateFeeCalculationStrategy.calculateLateFee(daysLate);

}

}

**2. Refactoring du Code pour Améliorer la Qualité**

Maintenant, nous allons refactoriser le code pour améliorer sa qualité, en utilisant par exemple l'injection de dépendances pour rendre la classe LoanManager plus flexible et facile à tester.

**Refactoring avec Injection de Dépendances :**

// Classe de gestion des emprunts avec injection de dépendances

public class LoanManager {

private final LateFeeCalculationStrategy lateFeeCalculationStrategy;

// Constructeur prenant la stratégie en paramètre

public LoanManager(LateFeeCalculationStrategy lateFeeCalculationStrategy) {

this.lateFeeCalculationStrategy = lateFeeCalculationStrategy;

}

// Méthode pour calculer les frais de retard en fonction de la durée de l'emprunt

public double calculateLateFee(int daysLate) {

return lateFeeCalculationStrategy.calculateLateFee(daysLate);

}

}

**Conclusion**

En utilisant le pattern Strategy, nous avons pu implémenter la fonctionnalité de gestion des emprunts de livres de manière flexible et modulaire. En refactorisant le code pour utiliser l'injection de dépendances, nous avons rendu la classe LoanManager plus testable et plus facile à utiliser dans différentes situations. Cela démontre l'importance des design patterns dans l'écriture de code propre, maintenable et extensible.

**1. Singleton**

**Objectif** : Assurer qu'une seule instance d'un service de gestion des livres est utilisée dans toute l'application.

**Exemple de Code** :

public class BookManagementService {

private static final BookManagementService instance = new BookManagementService();

private BookManagementService() {

// Private constructor to prevent instantiation

}

public static BookManagementService getInstance() {

return instance;

}

// Other methods for book management...

}

**2. Factory**

**Objectif** : Créer des instances de différentes classes de livres sans spécifier la classe exacte.

**Exemple de Code** :

public interface BookFactory {

Book createBook(String title, String author, String isbn);

}

public class NovelFactory implements BookFactory {

@Override

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

return new Novel(title, author, isbn);

}

}

public class EssayFactory implements BookFactory {

@Override

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

return new Essay(title, author, isbn);

}

}

// Usage:

BookFactory factory = new NovelFactory();

Book novel = factory.createBook("Title", "Author", "ISBN");

**3. Observer**

**Objectif** : Notifier les abonnés lorsqu'un nouveau livre est ajouté à la bibliothèque.

**Exemple de Code** :

public interface BookAdditionObserver {

void update(Book book);

}

public class NotificationService implements BookAdditionObserver {

@Override

public void update(Book book) {

System.out.println("Notification: New book added - " + book.getTitle());

// Send notification to subscribers...

}

}

public class Library {

private List<BookAdditionObserver> observers = new ArrayList<>();

public void addObserver(BookAdditionObserver observer) {

observers.add(observer);

}

public void addBook(Book book) {

// Add book to library...

notifyObservers(book);

}

private void notifyObservers(Book book) {

for (BookAdditionObserver observer : observers) {

observer.update(book);

}

}

}

**4. Decorator**

**Objectif** : Ajouter des fonctionnalités supplémentaires aux livres, telles que l'option de réservation.

**Exemple de Code** :

public abstract class BookDecorator implements Book {

protected Book decoratedBook;

public BookDecorator(Book decoratedBook) {

this.decoratedBook = decoratedBook;

}

@Override

public String getTitle() {

return decoratedBook.getTitle();

}

@Override

public String getAuthor() {

return decoratedBook.getAuthor();

}

@Override

public String getISBN() {

return decoratedBook.getISBN();

}

}

public class ReservationDecorator extends BookDecorator {

public ReservationDecorator(Book decoratedBook) {

super(decoratedBook);

}

@Override

public void reserve(User user) {

// Reserve book...

}

}

// Usage:

Book novel = new Novel("Title", "Author", "ISBN");

novel = new ReservationDecorator(novel);

Ces exemples illustrent comment utiliser les design patterns Singleton, Factory, Observer et Decorator dans le contexte de gestion de bibliothèque pour résoudre différents problèmes de conception.