| Professeur : H.Mcheick | Lijun ZHENG – Paul FUSSIEN | 10/12/2019 |
|---|---|---|

UQAC

**Projet : POA**

# Rapport final

## Programmation objet Avancée

Lijun ZHENG
Paul FUSSIEN

UQAC

## Informations

## Information à propos du projet

| | |
| --- | --- |
| **Nom du projet** | **Réservation de siège pour un cinéma** |
| **Type de document** | **Rapport final** |
| **Date** | **10/12/2019** |
| **Version** | **1.1** |
| **Etudiant** | **FUSSIEN Paul**<br>**ZHENG Lijun** |
| **Professeur** | **Mcheick Hamid** |

## Modifications

| Version | Date | Name | Description |
| --- | --- | --- | --- |
| 1.0 | 04/12/2019 | FUSSIEN Paul | Réalisation du design |
| 1.1 | 09/12/2019 | FUSSIEN Paul<br>ZHENG Lijun | Ecriture des différentes parties |

| Professeur : H.Mcheick | Lijun ZHENG – Paul FUSSIEN | 10/12/2019 |
|---|---|---|

UQAC

# Explication du projet

## ABSTRACT

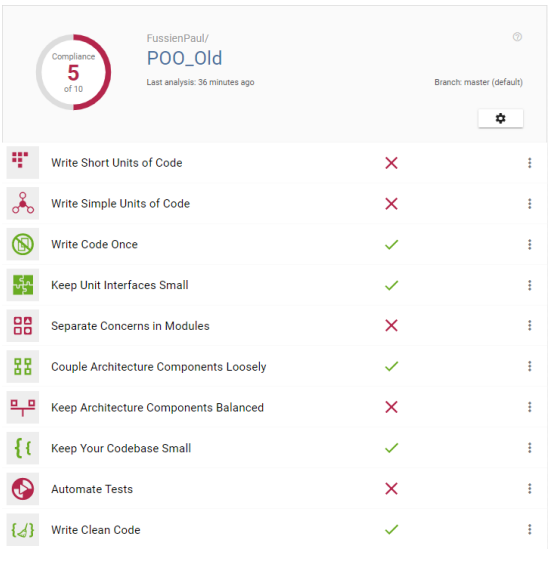
**Ce document est lié au projet du cours « Programmation objet avancée » réalisé par ZHENG Lijun et FUSSIEN Paul lors du trimestre d'automne 2019 à l'UQAC. Celui-ci concerne le refactoring d'un code existant sur Github de LyleBranzuela.**

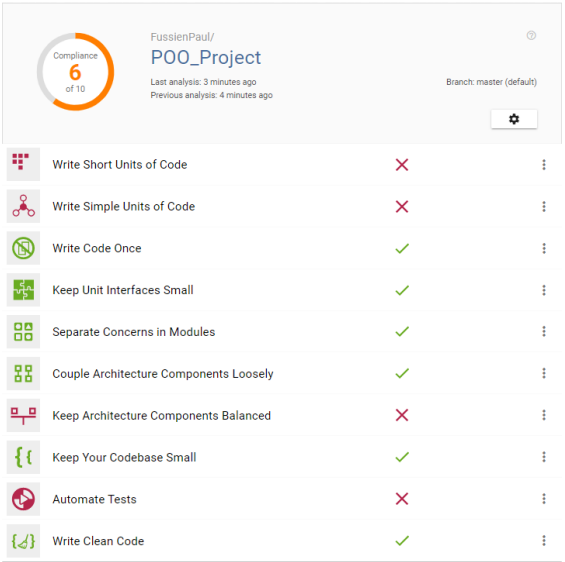
## INTRODUCTION

Dans le cadre de la validation de la maitrise, et de celui du cours de « Programmation d'objet avancée », nous avons réalisé le refactoring d'un programme de réservation de siège pour un cinéma. Lors de ce projet, les principes SOLID, JavaFx, l'expression lambda et l'architecture MVC seront utilisés afin d'améliorer le code. De plus, afin de rendre l'application utilisable pour un vrai cinéma, nous avons ajouté le NIO et la sérialisation pour stocker les données. Ces modifications à apporter, nous ont motivés pour travailler sur ce projet.

Le problème de ce code, est que celui-ci a été réalisé dans le cadre un devoir (présence de l'information question 1 à 5 sur le READ.ME) cela provoquant la réalisation point par point du projet. De plus, aucun principe que nous avons vu en POA ne sont utilisés rendant le code illisible et difficilement compréhensible. Comme indiqué avant, il n'y a pas de base de données, cela donnant une nouvelle session à chaque ouverture du programme.

Une fois le code refactoré, l'interface graphique modifié et la fonction NIO installée, nous obtenons un code beaucoup plus lisible et compréhensible pour un nouveau programmeur. De plus, l'interface utilisateur s'adapte selon l'écran de l'utilisateur permettant d'éviter tous les problèmes de mise à l'échelle et avec des composants beaucoup plus optimisés. Maintenant, à chaque ouverture, nous pouvons récupérer ce qui a été réalisé lors des sessions précédentes, ce qui n'était pas le cas avant. Voici la comparaison que nous obtenons avec « Better code hub » :



Nous pouvons voir via ces deux résultats (A gauche, le projet de base et à droite, le projet refactoré) que nous avons aidé le programme à gagner un point parmi les 10 critères :

Sachant que « Write short units of code » est dû à l'interface graphique qui est difficile à couper en Java Fx et les tests automatiques qui n'y sont pas de base et est trop complexe à mettre en place et ne concerne pas vraiment le cours de POA.

Afin de présenter et couvrir l'intégralité du projet, nous allons procéder de la manière suivante :

- Un rappel des éléments pouvant aider le programme de base en utilisant les principes vus en cours. En mettant en lumière, leurs avantages et limites permettant d'affirmer nos choix
- Une présentation de notre solution avec graphiques permettant de les expliquer et possiblement ce qui pourrait être réalisé en plus afin de rendre le projet réellement viable et efficace.
- L'implémentation et la validation de ce que nous avons fais lors de ce projet

UQAC

## Approches existantes

Lors de ce projet, il était essentiel d'utiliser les notions vu lors du cours de POA afin de valiser plusieurs principes permettant un code plus clair et lisible mais également fonctionnel. Pour cela, il a fallu faire du refactoring, c'est-à-dire améliorer le code (performance et utilisation du CPU) sans changer son comportement. Nous avions donc 3 méthodes qui nous paraissaient viables à implémenter :

- La programmation orienté objet
- La programmation orienté aspect
- Les principes SOLID

Pour la programmation orienté objet, nous pouvions l'implémenter afin de rendre le code réutilisable. L'encapsulation et la maintenabilité étaient également des arguments de poids pour choisir cette méthode, mais plusieurs autres facteurs ont joué en sa défaveur :

- La performance souffre parfois en utilisation intensive du polymorphisme en temps d'exécution, patrons de conception imbriqués et autres artifices *POO*.
- Taille de la base de code conséquente, résultat de l'accumulation des niveaux d'abstraction.
- Ne convient pas à tous les problèmes : le paradigme fonctionnel si disponible, simplifie certaines complexités spécifiques.
- Pas très pratique pour des petits projets ou test de fonctionnalités non *POO*.

Or, notre projet étant petit et avec beaucoup de classes n'ayant pas beaucoup d'interactions entre-elles, la programmation orientée objet n'était donc pas la bonne méthode.

Pour la programmation orientée aspects, est très intéressante puisqu'elle permet une modularité excellente permettant de toucher le moins possible au code lors d'un ajout d'éléments. Mais celle-ci ne nous semblait pas légitime pour ce projet car :

- La lecture du code contenant les traitements ne permet pas de connaitre les aspects qui seront exécutés (sans utiliser un outil).
- Pas de normalisation : il existe plusieurs approches et implémentations, chaque implémentation proposant sa propre solution
- Notre projet, ne nous permet de réaliser de la POA où elle serait réellement utile

Les principes SOLID, sont finalement notre choix pour le refactoring de ce projet puisqu'ils permettent de combiner plusieurs aspects permettant de rendre le code lisible, réutilisable, maintenable et avec peu de dépendance.

Pour la partie graphique, nous avions le choix entre deux méthodes : Swing/Awt et JavaFx. Ces deux méthodes peuvent réaliser une IU très efficace et peu couteuse. Nous avons donc réalisé un tableau permettant de les comparer sur ce que nous voulions faire :

| | **Swing/Awt** | **JavaFx** |
|---|---|---|
| **Component** | **X** | |
| **Interface Graphique** | | **X**<br>**Scene Builder \| CSS** |
| **Développement** | | **X**<br>**WPF** |
| **Architecture** | | **X**<br>**MVC** |
| **Support** | | **X** |

Comme peut le faire sous-entendre ce tableau, nous avons choisi de réaliser le projet sous une interface graphique JavaFx, car, celui-ci possède beaucoup d'avantage et nous l'avions déjà utilisé lors d'un TP de cours. Ensuite, L'interface graphique JavaFx peut être construite par le logiciel ***Scene Builder***, le view est compatible avec CSS et l'architecture est compatible avec MVC. De plus, le développement de JavaFx est ressemblé au développement de WPF en C# (Voir Annexe). D'ailleurs, il permet d'utiliser moins de ressource ce qui représente un réel avantage.

Enfin, nous avions rajouté une fonction qui n'était pas de base dans le projet étant une base de données. Dans le projet de base, a chaque ouverture de l'application, une nouvelle session est créée et tout ce qui a été réalisé en amont est perdu. Normalement, pour la partie de la base de données, il faut créer un système de serveur-client. C'est le serveur qui communiquer avec la vraie base de données (par exemple : MarioDB) afin d'écrire ou de lire les données, et ensuite il peut envoyer les informations au client. Pour simplifier la base de données, nous créons un fichier .txt au lieu de la base de données afin de simuler le système de serveur-client. Pour cela, nous avions deux choix : IO et NIO. De la même manière, nous avons réalisé un tableau permettant de les comparer.

| **IO** | **NIO** |
|---|---|
| **Stream** | **Buffer** |
| **Bloqué** | **Non-bloqué** |

Comparé avec IO, NIO est plus flexible, on peut modifier les données dans le ***Buffer.*** D'ailleurs, dans le monde réel, il y a plusieurs clients et un seul serveur. Ce n'est pas pratique pour le mode ***Bloqué.*** Nous avons donc choisi NIO pour le projet.

UQAC

## Nos solutions

Afin de résumé ce qui a été dis précédemment, nous avons choisis les options suivantes :

- Principes SOLID et POO
- Java Fx
- NIO
- L'architecture MVC

Afin d'expliquer ce qui a été réalisé, nous devons d'abord rappeler ce que sont les principes SOLID :

Le S correspond à "Single Responsibility" qui se traduit par une seule responsabilité. Ce principe implique que votre code au sein d'une classe ne doit avoir qu'une seule responsabilité, qu'un seul type de tâche à effectuer. Si vous prenez conscience que 2 tâches différentes sont effectuées, posez-vous la question de savoir si vous devez scinder votre classe en deux ou non.

Le O correspond à "Open / Close" qui se traduit par... Ouvrir / Fermer ! La raison de ce principe est très simple, votre classe doit-être ouverte à une extension/mise à jour, mais doit-être fermée pour toutes modifications. Cette méthode permet d'éviter la régression suite à l'ajout de nouvelles fonctionnalités. Bien sûr qu'il peut y avoir des cas ou la modification est indispensable comme la résolution d'un bug, etc. Mais ce principe invite à le faire le moins possible.

Le L correspond à "Liskov Substitution" qui se traduit par substitution de Liskov. Juste pour information complémentaire, Liskov, de son prénom Barbara est une informaticienne de renom (prix Turing, équivalent du Nobel) qui a évoqué ce principe au début des années 90. Ce principe stipule d'une manière simplifiée, qu'un objet S d'un sous type T doit pouvoir remplacer un objet de type T sans avoir de conséquences dans le comportement du programme. L'exemple concret le plus souvent utilisé pour évoquer ce principe est le rectangle ainsi que le carré. En effet, prenons le rectangle comme Type principal et le Carré comme sous type du rectangle (le Type carré est un rectangle particulier et donc potentiellement héritable du Type rectangle), la question est alors, le carré peut-il remplacer le rectangle dans le programme sans avoir de conséquences ? Eh bien non ! Si vous remplacez l'objet de type rectangle et que vous mettez un objet de type carré, avec quel côté du rectangle peut-on calculer l'air de cet objet vu que le carré ne possède qu'un seul côté, qu'une seule propriété, et que le type rectangle en possède deux ? Pour respecter ce principe, la technique consistera à créer une classe dont héritera le carré et le rectangle et par exemple de créer une méthode abstraite que chacune des figures implémentera afin de pouvoir calculer l'air.

Le I correspond à "Interface segregation" qui se traduit par ségrégation des interfaces. C'est une technique qui consiste à préférer la création de plusieurs petites interfaces qui concentre vraiment le domaine dans lequel elle a été produite plutôt qu'une seule qui contiendrait beaucoup trop de déclarations. Cela facilite beaucoup de choses comme les tests unitaires, une meilleure décomposition du code et donc une meilleure compréhension générale.

Le D correspond à "Dependency Inversion" qui se traduit par inversion des dépendances. Ce principe permet d'inverser les dépendances entre les modules de vos applications. Concrètement, si vous avez un projet avec du code métier (Business Layer - BL) et un projet qui s'occupe de la persistance de vos données (Data Access Layer - DAL ), généralement votre projet BL référence votre projet DAL. Et bien avec ce principe, c'est justement l'inverse qui va se produire, vous n'allez plus faire référence à DAL dans votre projet BL et vous allez créer une abstraction et c'est à partir de là qu'intervient ce que l'on appelle une injection de dépendance. Ceci à des avantages à plus d'un titre : Faciliter les tests unitaires, gérer vos objets d'une manière responsable, remplacer votre DAL par une autre (changement d'une base de données), etc.

Ces principes utilisés, nous obtenons les modifications suivantes :

```java
public class MovieSession implements
Comparable<MovieSession>
{
    private String movieName;
    private char rating;
    private Time sessionTime;
    private SeatReservation[][] sessionSeats;
    public static int NUM_ROWS = 8; //How
many Rows of Seats
    public static int NUM_COLS = 6; //How many
Columns of Seats
```

```java
public class Movie {

	private String name;
	private char rating;
	public Movie(String name, char rating)
{
		this.name = name;
		this.rating = rating;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public char getRating() {
		return rating;
	}
	public void setRating(char rating) {
		this.rating = rating;
	}
}
```

```java
public static int convertRowToIndex(char rowLetter)
{
    return (rowLetter - '0') - 17;
}

/**
 * A method that converts an integer of index to a char of row to be used in a seat reservation.
 * @param rowIndex the index of the character of the row.
 * @return the character of the row based on the index.
 */
public static char convertIndexToRow(int rowIndex)
{
    return (char) ((rowIndex + '0') + 17);
}
```

```java
public interface Conversion {

      /**
     * A method that converts an char of row to a integer of index to be used in an array.
     * @param rowLetter the character of the row.
     * @return the index of the character of the row based on the characters.
     */
    public static int convertRowToIndex(char rowLetter)
    {
        return (rowLetter - '0') - 17;
    }

    /**
     * A method that converts an integer of index to a char of row to be used in a seat reservation.
     * @param rowIndex the index of the character of the row.
     * @return the character of the row based on the index.
     */
    public static char convertIndexToRow(int rowIndex)
    {
        return (char) ((rowIndex + '0') + 17);
    }
}
```

Ensuite, nous avons utilisé la méthode NIO permettant de réaliser un buffer de données non bloqués. La base de données étant un fichier txt (pouvant être remplacé par un serveur en ligne). Le socketchannel a été remplacé par filechannel pour le rendre utilisable. Voici, un exemple de code que nous avons pour la méthode NIO :

```
private void createFiles() {
	try {
	for(Session session : movies)
	{
		String uri = "src/" + session.getId() + ".txt";

		System.out.println(uri);
		RandomAccessFile aFile = new RandomAccessFile(uri, "rw");
		FileChannel      channel = aFile.getChannel();

		ByteArrayOutputStream baos = new ByteArrayOutputStream();
		ObjectOutputStream oos = new ObjectOutputStream (baos);
		oos.writeObject(session);
		oos.flush();
		channel.write (ByteBuffer.wrap (baos.toByteArray()));
	}
	} catch (IOException e) {
		e.printStackTrace();
	}
}
```

La sérialisation a également été utilisé car dans l'architecture MVC, le client sérialise l'objet qui est stocké dans le buffer pour être, par la suite, envoyé au serveur. Le serveur désérialise l'objet afin de le stocker dans une base de données (Mysql).

UQAC

## Implémentations et validation

Maintenant que le code était mis en place et les fonctions rajoutés, il nous fallait de réaliser les diagrammes des classes des deux projets mais également la comparaison des deux notes de better code hub et prendre en compte les remarques sur chaque critère non respecté.

Voici, les deux diagrammes des classes :

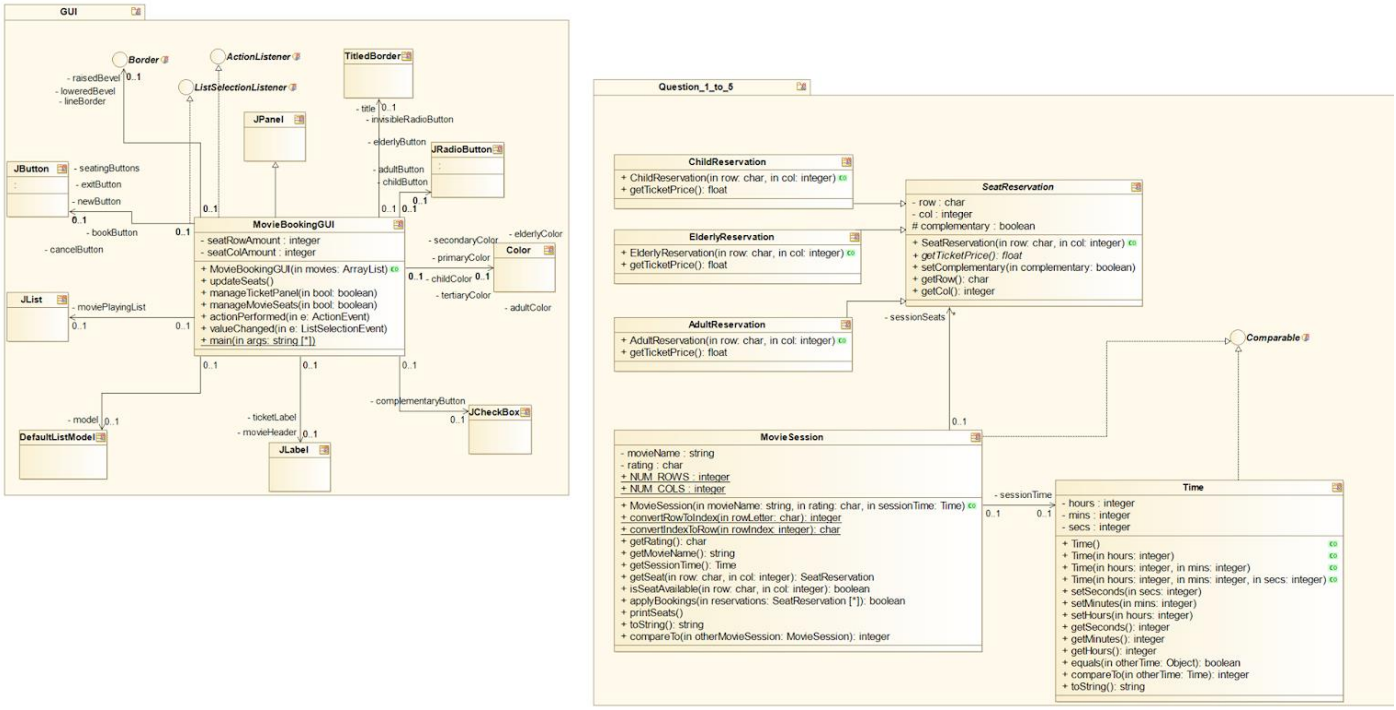



**Figure 1: Diagramme des classes projet de base**

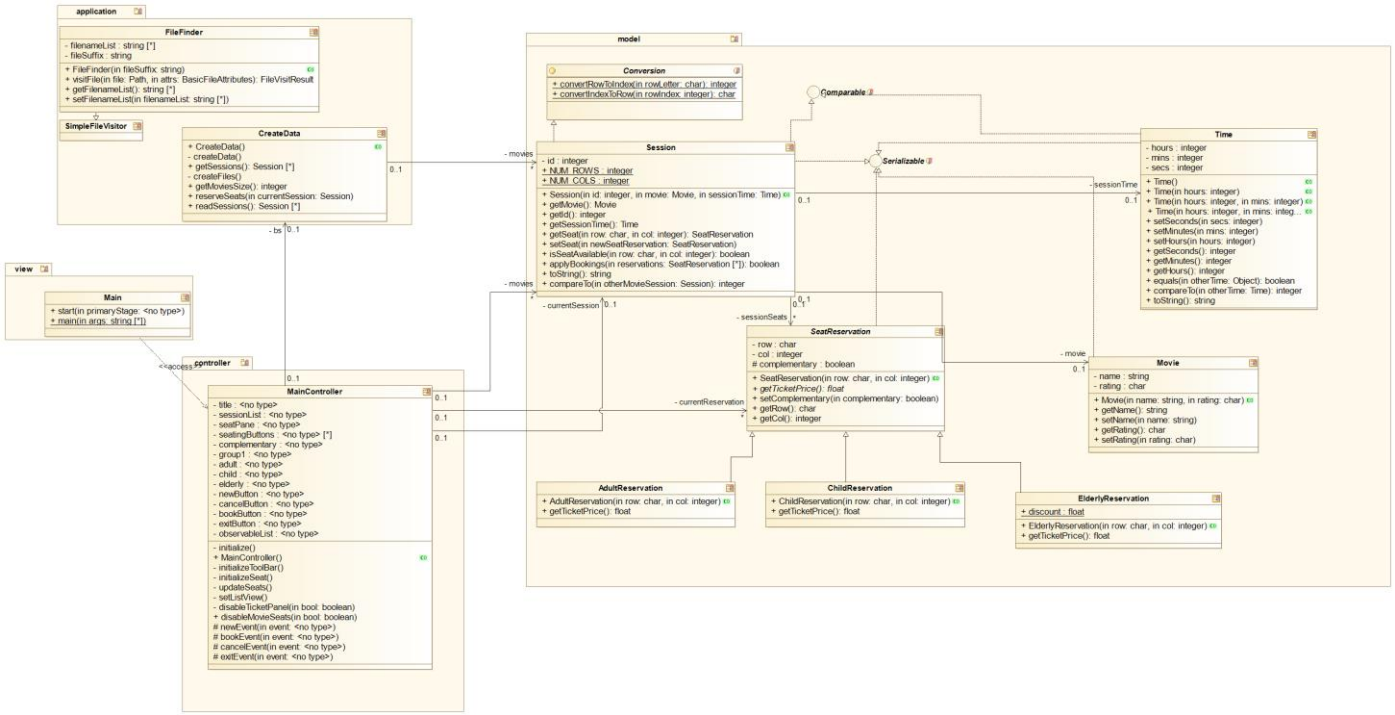
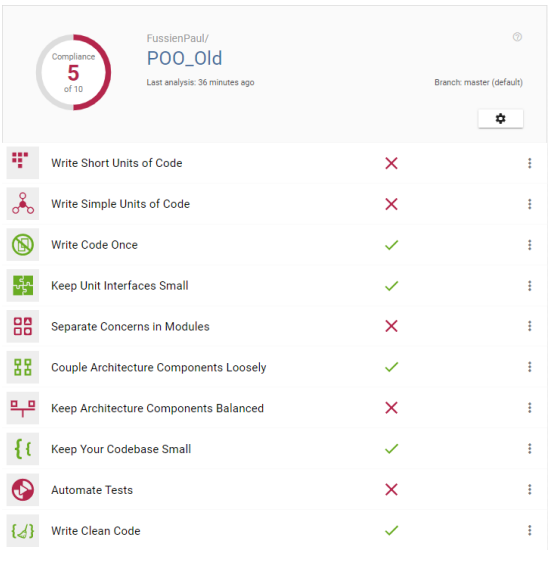
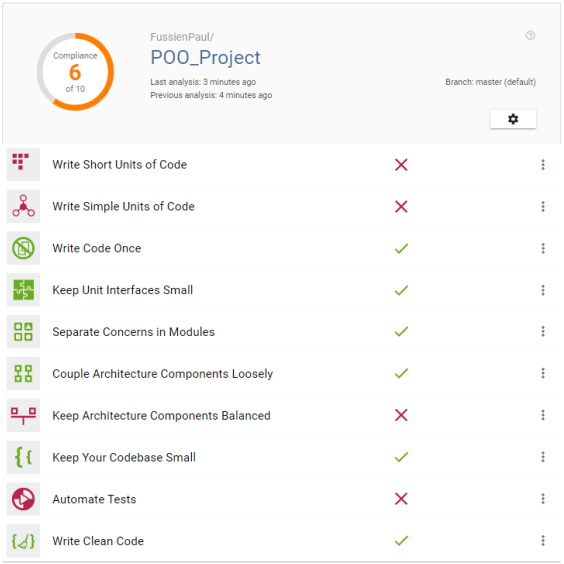



**Figure 2: Diagramme des classes projet refactoré**

Dans le deuxième cas, on voit que les classes ont été cassées et réparties afin de respecter les principes SOLID.

Maintenant que le projet est terminé, nous avons réalisé deux tests sur better code hub donnant les résultats suivants :



UQAC

# Conclusion

A travers ce rapport, nous avons présenté notre travail final de POA qui a pour but d'améliorer un code existant.

Comment dit lors de l'introduction, nous avons mit en pratique les méthodes apprises en POA afin de rendre un projet de classe en un projet respectant plusieurs principes de codage et utilisant des méthodes nécessaires pour l'optimisation et la réutilisation de celui-ci.

Dans le but de respecter le refactoring, le comportement du projet n'a pas changé mais ses performances ont été augmentées. Vous pouvez trouver ce code en utilisant cet URL :

https://github.com/20minute/POO

UQAC

## Référence

Site stackoverFlow, Octobre 2017, « How does JavaFX compare to WPF? [closed] »

- https://stackoverflow.com/questions/2016470/how-does-javafx-compare-to-wpf

Site educba, 2019, « Java Swing vs Java FX »

- https://www.educba.com/java-swing-vs-java-fx/

Site Jenkov, 2014, « Java NIO vs IO »

- http://tutorials.jenkov.com/java-nio/nio-vs-io.html

Site GitHUB, Mars 2019, « Movie Seat Reservation App » par Lyle Branzuela

- https://github.com/LyleBranzuela/Movie-Seat-Reservation-App

UQAC

## Annexes

Code FXML :

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.ListView?>
<?import javafx.scene.control.RadioButton?>
<?import javafx.scene.control.ScrollPane?>
<?import javafx.scene.control.ToggleGroup?>
<?import javafx.scene.control.ToolBar?>
<?import javafx.scene.layout.BorderPane?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.layout.Pane?>
<?import javafx.scene.control.CheckBox?>
<fx:root type="javafx.scene.layout.BorderPane" maxHeight="-Infinity" maxWidth="-Infinity"
minHeight="-Infinity" minWidth="-Infinity" prefHeight="600.0" prefWidth="800.0"
                         xmlns="http://javafx.com/javafx/8.0.171" xmlns:fx="http://javafx.com/fxml/1" >
  <top>
    <Pane prefHeight="52.0" prefWidth="600.0" BorderPane.alignment="CENTER">
      <children>
        <Label fx:id="title" alignment="CENTER" layoutX="1.0" prefHeight="57.0" prefWidth="800.0"
text="Label" />
      </children>
    </Pane>
  </top>
  <bottom>
  <GridPane BorderPane.alignment="CENTER">
                  <Label text="Ticket Panel:" GridPane.columnIndex="0" GridPane.rowIndex="0"/>
     <RadioButton fx:id= "adult" mnemonicParsing="false" text="Adult" textFill="#999999"
GridPane.columnIndex="1" GridPane.rowIndex="0">
       <toggleGroup>
```

```
            <ToggleGroup fx:id="group1" />
          </toggleGroup>
        </RadioButton>
        <RadioButton fx:id= "child" mnemonicParsing="false" text="Child" toggleGroup="$group1"
GridPane.columnIndex="2" GridPane.rowIndex="0"/>
        <RadioButton fx:id= "elderly" mnemonicParsing="false" text="Elderly" toggleGroup="$group1"
GridPane.columnIndex="3" GridPane.rowIndex="0"/>
        <CheckBox fx:id="complementary" text="Complementary" GridPane.columnIndex="4"
GridPane.rowIndex="0"/>
        <Button fx:id="newButton" mnemonicParsing="false" text="New" GridPane.columnIndex="5"
GridPane.rowIndex="0" onMouseClicked="#newEvent"/>
        <Button fx:id="bookButton" mnemonicParsing="false" text="Book" GridPane.columnIndex="6"
GridPane.rowIndex="0" onAction="#bookEvent"/>
        <Button fx:id="cancelButton" mnemonicParsing="false" text="Cancel"
GridPane.columnIndex="7" GridPane.rowIndex="0" onMouseClicked="#cancelEvent"/>
        <Button fx:id="exitButton" mnemonicParsing="false" text="Exit" GridPane.columnIndex="8"
GridPane.rowIndex="0" onMouseClicked="#exitEvent"/>
    </GridPane>


    </bottom>
    <center>
      <GridPane fx:id="seatPane" prefHeight="150.0" prefWidth="200.0"
BorderPane.alignment="CENTER" />
    </center>
    <right>
                    <ListView prefHeight="150" prefWidth="300.0" fx:id="sessionList" />
    </right>
</fx:root>
```