Nycola Plaisance

Conception et Développement Orientés Objet

420-C46-JO

Groupe : 01

**Rapport Simulateur de vol**

Travail remis à :

Charles Jacob

Cégep Régional de Lanaudière à Joliette

Date de remise : 31 mai 2021

# Application des patrons GOF

## MVC

Afin d’appliquer le patron MVC j’ai séparé mes classes en fonction de leur rôle, c’est-à-dire Modèle, Vue et Contrôleur. Le modèle représente l’ensemble des classes qui sont chargé de représenter de réelle chose tel que des aéroports, des aéronefs et ainsi de suite ou encore des choses un peu plus surréelles tel que les états des aéronefs. Le contrôleur lui se charge d’effectuer les traitements ordonner par la vue sur le modèle. La vue, elle, se charge principalement de fournir une interface pour l’utilisateur et de fournir au contrôleur le « input » de l’utilisateur.

## Patron « Factory » et le singleton

Afin d’appliquer le patron usine ainsi que le patron singleton j’ai écrit une classe qui sera chargé de créer tous types de clients et celle-ci sera un singleton par souci d’optimisation. En gros, la présence de mon usine est justifiée par le besoin d’instancier plusieurs classes (les clients) relativement semblable. Donc cette tache devait bien être exécuter par une classe et c’est ici que l’usine devient utile. Ensuite, j’ai pris la décision de faire de cette usine un singleton puisqu’il semblait logique de le faire ainsi vue que dans le cas contraire, un nouvel objet d’usine à client serait instancié à chaque fois que nous aurions besoin de créer un client et je vois ceci comme une perte de temps (et d’opérations), si on ne détruit simplement pas cet objet on peut alors le réutiliser !

## Patron Facade

Afin d’appliquer le patron façade j’ai fait bien attention à ne faire aucune communication entre les classes derrières les façades (c’est-à-dire les classes Scenario, Controller\_Simulation et View\_Simulation) et tout autre classe qui n’appartienne pas aux mêmes parties du programme (respectivement, le modèle, la vue et le contrôleur). Ainsi, les seules classes qui communique aux classes appartenant aux autres parties du programme ne sont que les façades.

## Patron Stratétat

Afin d’appliquer le patron statétat, j’ai codé des classes qui corresponde à chaque état d’un aéronef (respectivement, l’attente, la maintenance, le vol, l’embarquement et le désembarquement). Ensuite, ces états ont été garni de méthode différente ce qui les faisait faire des actions différentes dépendamment de leur état.

# Décision de design et Patrons GRASP

## La classe Scénario

L’existence de la classe scénario est un exemple parfait du patron GRASP de fabrication pure, dans le sens qu’en réalité, le scénario n’est pas vraiment une chose qui existe. C’est une classe qui ne provient pas d’un élément faisant partie de la réalité ce qui fait de cette classe une classe de fabrication pure.

## La forte cohésion

Lors de la conception de toute classe j’ai fait très attention à faire en sorte que chaque classe aille des responsabilités qui était en lien directe avec son rôle et ses données. De plus, chaque fonction a été séparer le plus possible l’une de l’autre. Par exemple, La méthode LookForMatch de la classe Airport ne fait que ce que son nom indique, elle regarde si un match est possible avec les données de l’aéroport et si c’est le cas, celle-ci déclenche la méthode Match et celle-ci ne fait que ce que son nom indique elle lie le client et l’aéronef ensemble et débute la mise en marche de l’embarquement ou de la mise en vol (dépendant du type d’aéronef).

## Le placement des clients spéciaux dans le Scenario

En théorie, vue que les clients normaux (tel que les passagers et les cargos) se retrouvent dans un aéroport (donc dans la classe représentant un aéroport) il serait logique d'avoir une classe qui regroupe les clients spéciaux. Cependant, je trouve qu'avoir une classe qui a pour seul but de réunir des Clients est très inutile et une perte de temps pour ce qui est de la gestion des données de la classe. Donc, j'ai pris la décision de mettre ces Clients spéciaux directement dans la classe "Scenario".

## Le placement des clients normaux dans les aéroports

Ce choix de disposition des clients Normaux est une extension de la logique appliquer dans le point précédent. En gros, il faisait du sens que les clients spéciaux soient dans une liste du scénario vue qu'il étaient éparpillé partout sur la carte cependant les clients normaux eux ne sont pas éparpillés. Cette différence apporte la décision de les placer dans les aéroports (non seulement cela fait énormément de sens puisque dans la vie réelle les clients sont bel et bien dans l'aéroport).

## Création de la classe Position

"Position" est un « wrapper » avec un merveilleuse petite fonctionnalité (la conversion des coordonnées pixels en coordonnées d'une carte) qui est présente dans plusieurs autres classes tel que "Airport" et "Client\_Special". Il faisait beaucoup de sens de faire cette classe puisque non seulement elle contenait des données mais en plus elle contenait des fonctions spéciales à elle et donc le fait de réunir le tout sous une classe se voit justifié.

## Les états dans une file

Le fait de placer les états dans une file permet facilement de passer d'un état au suivant (suffit de dépiler). De plus, cette structure de données nous permet de planifier chaque état à travers lesquels nos aéronefs vont faire dès que ce dernier débute son embarquement ou son décollage.