



Haute Ecole
Libre de
Bruxelles

Q4 2022

Programmation IV : JAVA

HELBPark : Projet de JAVA IV

Jessy ADAM

Professeur :

M. Riggio



2022

Table des matières

1	Introduction	1
2	Présentation de l'interface graphique	2
3	Analyse et Designs Patterns^{TMTM}	5
3.1	Designs Patterns ^{TMTMTMTM}	6
3.1.1	MVC	6
3.1.2	Factory	6
3.1.3	Singleton	6
3.1.4	Stratégie	6
3.1.5	Observer	7
4	Limitations	8
5	Conclusion	9

Dans le cadre du cours de JAVA IV, il nous a été demandé de réaliser une application JavaFX utilisant les concepts vus en classe.

Le projet, dénommé HELBPark, est un système de gestion de parking ajoutant automatiquement les véhicules qui s'y présente et permettant de libérer et corriger les informations de ces même véhicules, en plus d'autres contraintes.

Ce rapport concernant ma version du projet, expliquera le contenu et la processus de développement de celle-ci.

Ce rapport parlera donc de ces points :

- Une présentation de l'interface graphique;
- Une analyse ainsi qu'une explication des Design Patterns™ utilisés;
- Des limitations de cette application;
- Terminant par une simple conclusion.



FIGURE 1.1: Logo et Titre de l'application

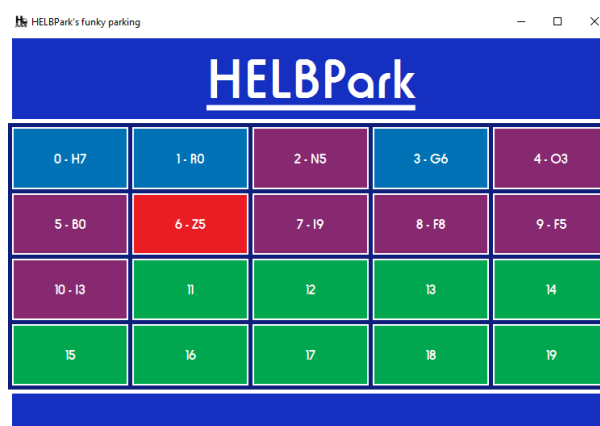
Présentation de l'interface graphique

Au lancement de l'application ressemble à ceci :



L'esthétique est inspirée de panneau de circulation, car le contexte paraissait approprié.

Une fois que des véhicules sont arrivés ils sont affichés ainsi :

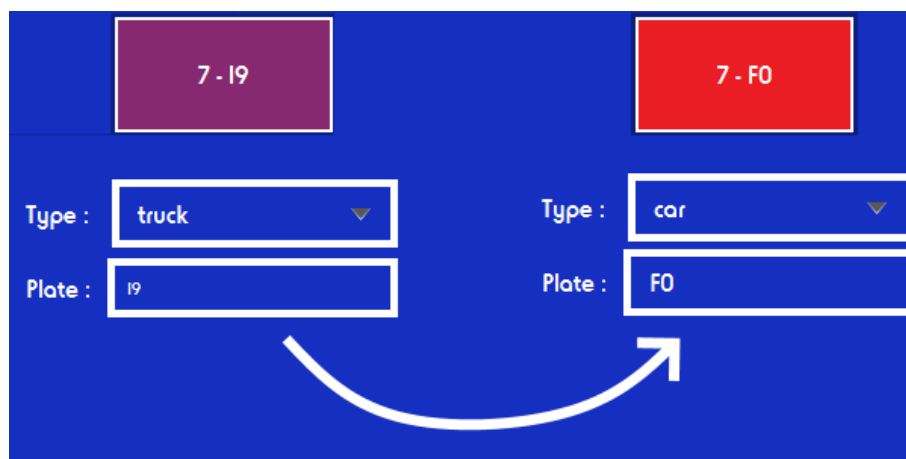


Chaque véhicule ayant une couleur différente comme demandé. Pour rappel, les couleurs sont : Bleu pour les motos, rouge pour les voitures, mauve pour les camionnettes, et finalement vert si la place est vide.

Quand on appuie sur un bouton et que la place est occupée, un menu apparaît qui permettra de modifier et libérer cette place. Elle ressembla à ceci :



On peut ensuite changer les paramètres d'un véhicules et appuyer sur le bouton "Apply" pour effectuer ces changement. Ceci mettra aussi le véhicule à jour dans l'interface principale, comme vous pouvez le voir ci-dessous :



L'employé en sera aussi notifié sur l'interface principale, via ce message :

Changes applied !

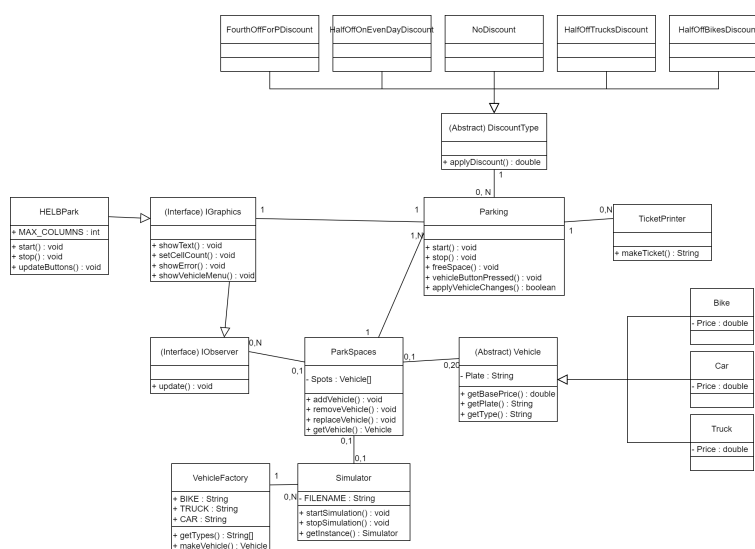
On peut aussi libérer la place, en appuyant sur le bouton à gauche. Ceci générera le ticket qui sera notifié sur l'interface principale comme ci-dessous :

Ticket S7_std.txt created

Et la place sera mise à jour sur l'interface principale, comme vous pouvez le voir ci-dessous :



Afin de présenter la structure de l'application, j'ai fait un diagramme de classe :



Comme vous pouvez le voir le principal de l'application est divisé en 3 parties, selon le Design Pattern^{TMTM} Modèle-Vue-Contrôleur.

L'interface de la vue IGraphics contient toutes les méthodes nécessaires au fonctionnement avec le contrôleur et donc toute classe qui en hérite, dont HELBPark, peut être la vue. Ici, due à la manière dont fonctionne JavaFX, HELBPark est aussi le point d'entrée du programme.

La classe Parking, étant le contrôleur, prend en charge tous les événements de l'interface graphique et modifie le modèle si besoin. Celui-ci utilise TicketPrinter pour enregistrer les tickets et les diverses versions de DiscountType pour faire les réductions.

Le modèle ParkSpaces stocke des véhicules dans un tableau et est l'expert en informations des véhicules. Il garde une liste de IObserver, dont IGraphics hérite, pour les notifier de tout changement. La classe Simulator se charge de fournir les informations par le biais de VehicleFactory qui générer les véhicules à stocker.

3.1 Designs Patterns^{TMTMTMTM}

3.1.1 MVC

Comme discuté plus haut, j'utilise MVC pour la structure principale de l'application. Ce pattern consiste à séparer les différentes couches de l'application en des classes séparées afin de pouvoir changer de vue si nécessaire, ainsi que pour appliquer le principe de forte cohésion et ne pas donner toutes les responsabilités à une seule classe.

3.1.2 Factory

Factory est un Design Pattern^{TMTMTMTMTM} consistant à délégué la complexité de l'instanciation d'une classe à une autre dans le but de généraliser la création de ces instances. Ici je l'ai utiliser pour les véhicules, comme j'utilise de l'héritage pour différencier entre les types, j'utilise une fabrique, `VehicleFactory`, pour directement instancier la bonne classe via une variable de type `String` définit comme constantes dans cette classe.

3.1.3 Singleton

Singleton permet de s'assurer qu'il y a toujours au maximum une et une seule instance d'une classe en rendant le constructeur privé et en gardant l'instance dans une variable statique. Je n'ai trouvé pertinent de l'utiliser que dans la classe `Simulator` pour m'assurer qu'il ne peut y avoir qu'une simulation à la fois. Mais pour tout dire je ne vois pas encore les avantages offerts par ce Design Pattern^{TMTMTMTMTMTM} par rapport à une classe statique, ce qui fait que je ne l'ai pas utilisé pour la fabrique `VehicleFactory` que j'ai créé après y avoir réfléchi.

3.1.4 Stratégie

Stratégie consiste à mettre la logique d'une méthode qui doit beaucoup varier dans différentes classes héritant du même parent afin de pouvoir facilement les interchanger. Ici, je l'ai utilisé pour les diverses réductions disponibles, toutes étant basés sur la classe `DiscountType`. Ceux-ci sont choisi au lancement de l'application en fonction du jour. Je l'ai choisi car il fallait facilement pouvoir en choisir une en fonction du jour et le fonctionnement m'avait l'air de bien coller.

3.1.5 Observer

Observer consiste à utiliser une interface pour regrouper des classes "d'observateurs" sur lesquelles il faut appeler une méthode commune, depuis une classe "observable". Ici je l'ai utilisé pour directement notifier les vues des changements dans le modèle via l'interface `IObserver`, héritée par `IGraphics` et donc `HELBPark`, afin d'éviter de devoir passer par le contrôleur alors qu'il ne fait pas grand chose à ce niveau là.

Pour ce qui est des limitations, une d'entre elles est qu'on ne peut pas corriger le fait qu'on ne détecte pas un véhicule étant donné que la fenêtre ne s'affiche que si on clique sur une place occupée.

Ensuite, bien que j'ai essayé d'appliquer le faible couplage, je pense encore que trop de classes se connaissent entre elles, comme `VehiculeFactory` contenant les constantes de type, la vue gardant l'instance du contrôleur et vice-versa.

Aussi l'application se retrouve difficile à traduire car toutes les variables `String` ne sont pas regroupés. Les commentaires sont aussi en français alors que l'interface ne l'est pas, ce qui n'est probablement pas le meilleur de mes choix.

Finalement, il est difficile de créer une autre vue car le point d'entrée de l'application est dans la vue car je n'arrivait pas à garder une instance de l'application `JavaFX` en l'appelant d'autre part.

Pour conclure je pense avoir créer une application assez robuste permettant de faire toutes les fonctionnalités demandés en appliquant les Design Patterns^{TMTMTMTMTMTMTMTM} applicables à la situation.

Ce rapport aura parlé du développement de cette application dont une présentation de l'interface graphique, un analyse et explication des Design Patterns^{TMTMTMTMTMTMTMTM} incorporés ainsi que du raisonnement derrière et quelques limitations de l'application.

Sur ce je vous remercie d'avoir lu et je suis officiellement en burnout après 4 rapport.