

|  |
| --- |
| EL MOUDNI Zakaria  MR Riggio  2024-2025 |

|  |
| --- |
| Haute Ecole Libre de Bruxelles Ilya Prigogine |
| Rapport de Projet JAVA IV |
| HELB Hotel |

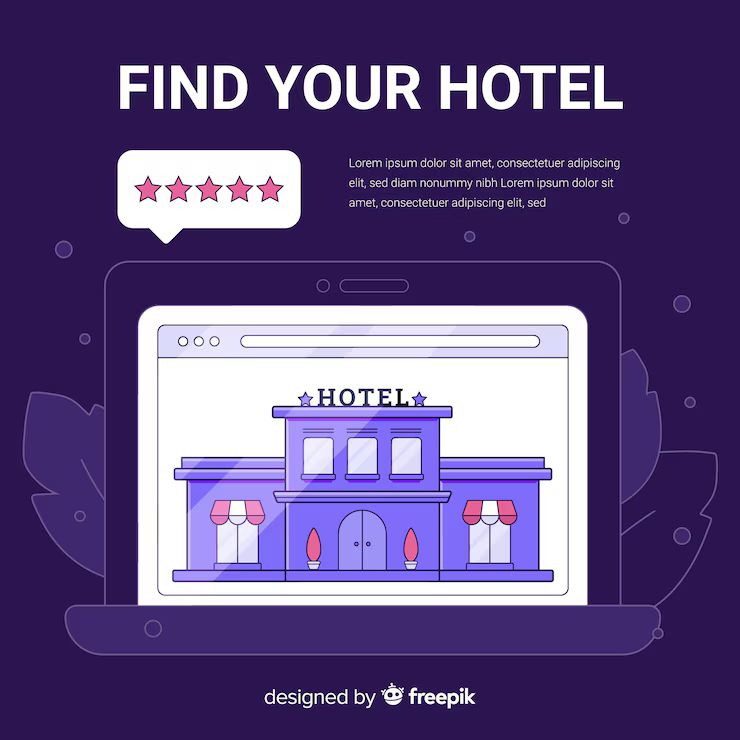


Table des matières

[1. Introduction 1](#_Toc198469662)

[2. Fonctionnalité 2](#_Toc198469663)

[3. Analyse et Application des Design Patterns 4](#_Toc198469664)

[3.1. Le modèle MVC 4](#_Toc198469665)

[3.2. Strategy – pour les assignations de chambre 4](#_Toc198469666)

[3.3. Observer – pour mettre à jour l’interface automatiquement 5](#_Toc198469667)

[3.4. Singleton – pour les objets uniques 5](#_Toc198469668)

[3.5. Factory – pour créer les tickets de loterie 6](#_Toc198469669)

[3.6. Null Object – pour éviter les null 6](#_Toc198469670)

[3.7. GRASP – pour mieux répartir les responsabilités 6](#_Toc198469671)

[3.8. Explication du diagramme UML 7](#_Toc198469672)

[Bloc Modèle (Model) 7](#_Toc198469673)

[Bloc Strategy 7](#_Toc198469674)

[Bloc Loterie 7](#_Toc198469675)

[Les flèches indiquent : 7](#_Toc198469676)

[4. Limitations et améliorations possibles 7](#_Toc198469677)

[5. Conclusion 9](#_Toc198469678)

# Introduction

Pour ce projet de fin d’année en Java IV, j’ai réalisé une application qui respecte les principes du cours, en particulier l’utilisation des design patterns vus en classe, le respect du modèle MVC, et la gestion d’une interface graphique avec JavaFX.

Le but de l’application était de simuler un hôtel avec plusieurs étages, où chaque chambre a un type bien défini (économique, business ou luxe), avec un système pour assigner automatiquement les chambres selon des stratégies différentes. J’ai aussi dû gérer les réservations, la libération des chambres, et même une sorte de mini-jeu de loterie quand un client libère sa chambre

L’interface est faite en JavaFX, et j’ai essayé de bien séparer les rôles entre les vues, les modèles et les contrôleurs. J’ai aussi fait en sorte d’éviter un maximum les null comme demandé, ce qui n’était pas simple au début. J’ai utilisé des classes comme EmptyRoom pour éviter les problèmes (on expliquera plus tard en détails cette class), et j’ai aussi mis en place une gestion de la configuration de l’hôtel en lisant un fichier. hconfig.

Au niveau des fonctionnalités, j’ai essayé de suivre les consignes. Par exemple, j’ai mis des couleurs différentes selon les types de chambres, j’ai ajouté une gestion des étages avec des lettres (A, B, C...), et j’ai essayé de faire une interface qui reste claire même quand on a beaucoup de chambres.

Dans ce rapport, je vais expliquer les fonctionnalités que j’ai mises en place, comment j’ai structuré mon code, quels patterns j’ai utilisés et pourquoi, puis je parlerai des limites de mon projet et de ce que j’aurais voulu faire en plus si j’avais eu plus de temps. Je ne suis pas un expert mais j’ai donné mon maximum, et je pense que j’ai beaucoup appris en le faisant.

# Fonctionnalité

Pour commencer nous allons présenter l’affichage principal de l’application, comme expliquer c’est une application qui permet la gestion d’un hôtel avec des réservations qui sont faites dans un site et traités par le logiciel.

Une image contenant texte, capture d’écran, logiciel, Icône d’ordinateur

Le contenu généré par l’IA peut être incorrect.

Quand on lance le programme, l’hôtel est automatiquement généré en fonction d’un fichier. hconfig. Ce fichier représente la structure d’un étage type (comme une matrice), et l’hôtel entier est basé dessus.

Une image contenant texte, Police, capture d’écran, nombre

Le contenu généré par l’IA peut être incorrect.

Le premier chiffre dans le fichier indique combien d’étages il y a. Chaque ligne après ça correspond à un étage type, et chaque lettre dans la ligne représente un type de chambre. Par exemple, E pour économique, B pour business, L pour luxe, et Z pour zone vide. Donc si une case contient un Z, n’y aura rien à cet endroit dans l’interface, ça sera juste vide.

L'affichage principal est une grille par étage, chaque étage est séparé, et chaque case correspond à une chambre (sauf celles marquées Z). Chaque bouton de chambre porte un nom qui indique sa position : la lettre de l’étage (A, B, C...) + un chiffre pour sa position dans la ligne + une lettre pour le type. Exemple : D3E = étage D, 3ème chambre, économique.

Quand on clique sur une chambre vide, on peut y faire une réservation. On doit remplir un formulaire avec le nom, prénom, nombre de jours, fumeur ou pas, enfants ou pas, etc. Ensuite, l’application assigne une chambre automatiquement selon la stratégie choisie. Il y a plusieurs stratégies dispos dans un menu déroulant : on peut choisir aléatoire, en fonction du calme, en fonction de l’objectif (vacances ou travail), ou simplement dans l’ordre (séquentiel). Si aucune chambre n’est dispo qui correspond à la demande, un message s’affiche.

Une image contenant texte, capture d’écran, logiciel, Police

Le contenu généré par l’IA peut être incorrect.

Quand on clique sur une chambre occupée, on peut la libérer. Mais avant de vraiment libérer, y a une petite fenêtre qui demande de noter le séjour (de 1 à 5), puis on lance un petit jeu de loterie qui permet de gagner une réduction pour une prochaine fois. Ce jeu change selon le type de ticket du client (Bronze, Silver ou Gold). Le ticket Gold par exemple fonctionne comme un jeu de boîtes, le client choisit une boîte et à chaque tour il en élimine une autre, et il peut décider de garder ou changer jusqu’à la fin. Le Silver est plus simple, c’est juste un mot mélangé que le client doit deviner. S’il trouve le bon, il gagne une réduction, sinon tant pis, et le Bronze c’est carrément un bouton au hasard.

J’ai aussi ajouté des couleurs pour bien distinguer les chambres. Chaque type a une couleur différente, comme ça visuellement on comprend vite ce qu’on voit. Les chambres occupées ont un style différent, donc c’est facile de voir ce qui est libre ou pas.

J’ai aussi mis en place un affichage par étage, donc on peut naviguer étage par étage pour voir les chambres. Y a un système pour afficher les infos d’une réservation en cliquant dessus, comme le nom de la personne, combien de jours il reste, etc. Et aussi un bouton pour modifier la réservation si besoin.

Toutes les réservations sont enregistrées dans un fichier CSV, donc si on lance l’appli, on peut retrouver les réservations qui sont stocker dans le CSV. Pareil pour la config de l’hôtel, elle reste la même tant qu’on ne modifie pas le fichier .hconfig.

# Analyse et Application des Design Patterns

Dans cette partie je vais expliquer comment j’ai structuré mon projet en utilisant plusieurs design patterns qu’on a vus en cours, et que j’ai appliqués pour organiser l’architecture. J’ai essayé de faire ça proprement pour que mon code soit lisible, réutilisable et plus facile à maintenir. Je vais présenter les patterns un par un, en expliquant comment je m’en suis servi.

## Le modèle MVC

C’est la base de mon architecture. J’ai séparé le programme en trois couches :  
- Vue : elle contient uniquement l’affichage (grille des chambres, formulaires de réservation, jeux, etc.)  
- Contrôleur : il gère toutes les actions quand on interagit avec l’interface  
- Modèle : il regroupe les classes qui contiennent les vraies données comme Hotel, Room, Reservation, etc.

Par exemple, quand on clique sur une chambre, la vue transmet l’action au contrôleur, qui consulte le modèle pour savoir si la chambre est libre, puis décide d’ouvrir une fenêtre ou pas.

En plus, mes vues ne font que de l'affichage. Y a zéro logique dedans, tout passe par le contrôleur. Ça rend l’interface beaucoup plus simple à modifier, et ça évite que le code parte dans tous les sens.

## Strategy – pour les assignations de chambre

J’ai utilisé le pattern Strategy pour gérer les différentes manières d’assigner une chambre. J’ai 4 classes qui implémentent l’interface AssignmentStrategy :  
- RandomAssignment

- Sequential

- PurposeBased

- QuietZone

Elles ont chacune une logique différente pour choisir une chambre. Le contrôleur ne connaît que l’interface, et appelle la méthode assignRoom() sans savoir ce qu’il y a dedans. J’ai aussi une classe Strategy (dans mon modèle) qui agit comme un sélecteur de stratégie, en stockant les différentes instances et en choisissant la bonne en fonction du nom.

## Observer – pour mettre à jour l’interface automatiquement

J’ai mis en place le pattern Observer pour relier les vues au modèle, sans avoir à tout rafraîchir manuellement. Par exemple, quand une chambre est assignée ou libérée, elle notifie les objets RoomObserver (comme la vue), qui se mettent à jour automatiquement.

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

J’ai fait la même chose pour les réservations (ReservationObserver) et les jeux de loterie (LotteryObserver), pour afficher les messages de réussite ou d’échec.

## Singleton – pour les objets uniques

J’ai utilisé le Singleton pour toutes les classes qui ne doivent exister qu’une seule fois :  
- EmptyRoom et EmptyReservation (pattern Null Object)- les 4 stratégies d’assignation (RandomAssignment, etc.)

- Au début, j’avais fait de l’hôtel un Singleton avec getInstance(), mais je l’ai supprimé plus tard pour mieux respecter MVC et GRASP. Maintenant, c’est le contrôleur qui crée l’hôtel directement à partir du fichier .hconfig

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

Ça permet d’avoir un seul point d’accès global, et d’éviter d’avoir plusieurs instances incohérentes.

## Factory – pour créer les tickets de loterie

Le pattern Factory est utilisé dans LotteryTicketFactory. Quand un client libère une chambre, on génère un ticket (Bronze, Silver ou Gold) en fonction de la note qu’il a donnée et du type de chambre.

Une image contenant texte, capture d’écran, Police, ligne

Le contenu généré par l’IA peut être incorrect.

Le contrôleur appelle simplement la factory, qui choisit le bon type de ticket à créer. Ça évite les if partout dans le code et rend l'ajout de nouveaux tickets plus simple.

## Null Object – pour éviter les null

J’ai remplacé les null par des objets vides :

- EmptyRoom remplace les chambres absentes dans la matrice

- EmptyReservation remplace les chambres sans réservation

Ces objets implémentent les mêmes méthodes, mais ne font rien. Ça m’évite d’avoir des NullPointerException et de devoir vérifier à chaque fois si un objet est null.

C’est un pattern utilisé en cours mais non demandé, il reste très intéressant pour ne pas avoir de null dans le code

## GRASP – pour mieux répartir les responsabilités

J’ai essayé de suivre les principes GRASP qu’on a vus en cours, comme :

- Contrôleur : la classe HELBHotelController relie les vues et le modèle

- Créateur : le Hotel est construit par un parser à partir du fichier .hconfig

-Couplage faible : chaque vue ou classe connaît juste ce qu’elle doit, et rien de plus. Ça évite de tout casser dès qu’on change un bout de code.

- Responsabilité : j’ai déplacé toute la logique des réservations dans ReservationManager pour ne pas surcharger le contrôleur

## Une image contenant texte, diagramme, ligne, Police Le contenu généré par l’IA peut être incorrect.Explication du diagramme UML

Le diagramme UML montre la structure principale de mon application. On y voit trois grands blocs :

### Bloc Modèle (Model)

* Room, EmptyRoom : les chambres
* Reservation, EmptyReservation : les réservations
* ReservationManager : gère les assignations/propositions
* Hotel : contient toutes les chambres, construit à partir d’un fichier .hconfig
* Strategy : sélectionne dynamiquement une stratégie selon un nom

### Bloc Strategy

* AssignmentStrategy est une interface
* Assignment est une classe abstraite
* RandomAssignment, QuietZone, PurposeBased, Sequential sont des implémentations concrètes
* Toutes les stratégies sont des Singletons et utilisent Hotel + ReservationManager

### Bloc Loterie

* LotteryTicket est une classe abstraite
* GoldTicket, SilverTicket, BronzeTicket en héritent et ont chacun leur propre jeu
* LotteryTicketFactory décide quel type de ticket créer
* DiscountCode est généré automatiquement selon le ticket

### Les flèches indiquent :

* Pleine = héritage (extends)
* Pointillé = implémentation (implements)

# Limitations et améliorations possibles

Même si j’ai essayé de rendre l’application la plus complète et stable possible, y a encore quelques trucs que j’aurais aimé pousser plus loin si j’avais eu plus de temps ou plus d’expérience. Le projet est globalement fonctionnel, mais voici ce que je considère comme des limites ou des pistes d’amélioration.

D’abord, même si l’affichage des boutons s’adapte à la taille de la fenêtre celle des réservations ne le fait pas, c’est une oublie de ma part.

Ensuite, le jeu de loterie fonctionne pour les trois types de tickets, mais dans le cas du Gold (avec les portes façon Monty Hall), je comprenais pas réellement ce qu’il fallait faire mais j’ai compris par la suite en faisant des recherches sur youtube et j’ai trouvé également sur stackoverflow des gens qui avaient créer ce jeux ainsi que des discussions sur ça

Un autre point, c’est qu’on n’a pas encore mis de vraie base de données. Tout passe par un fichier .csv pour les réservations. Ça suffit pour ce projet, mais à l’avenir, avoir une base de données un vrai hotel devrait travailler avec une bdd mais combiner une base de donnée avec du javafx devrait être un peu plus compliqué qu’une lecture de fichier csv.

Également je pense que j’aurais pu faire une amélioration sur l’utilisation de certains patterns surtout celui de l’Observer, qui pour moi était le plus compliqué dans la compréhension au début, avec le temps j’ai pu comprendre mais je n’ai pas eu l’occasion de l’appliquer parfaitement comme il se doit dans mon code je pense. Pareil pour la view je pense que j’aurais été capable de mieux la faire car après réflexion je me pose la question de l’état actuel est-il suffisant ou j’en ai fais trop.

Enfin, je sais que le projet respecte bien les contraintes (pas de null, pas de enum, pas de .stream()), toujours appliquer vos conseils, mais à certains moments j’ai trouvé que ça m’obligeait à faire des détours ou à alourdir le code vu qu’on avait pas le droit à certaine utilisation . J’ai essayé de garder tout propre, mais je pense qu’avec un peu plus de recul j’aurais pu simplifier certains blocs de logique, ou mieux factoriser certaines parties du contrôleur.

# Conclusion

Ce projet m’a vraiment aidé à mieux comprendre comment organiser un vrai programme en Java. Au début, j’étais un peu perdu avec tous les patterns, les contraintes comme “pas de null”, “pas de stream”, ou encore comment séparer correctement le modèle, la vue et le contrôleur. Mais à force de tester, de corriger, de refaire certaines parties, j’ai commencé à prendre le coup de main et à mieux structurer mon code.

Je suis content parce que j’ai réussi à mettre en place une application complète, avec plusieurs étages, différents types de chambres, des stratégies d’assignation, un système de réservation, une gestion correcte des erreurs, et même un mini-jeu de loterie avec trois types de tickets. J’ai respecté les consignes qu’on m’a données, que ce soit pour les patterns (MVC, Strategy, Observer, Factory, Singleton, Null Object…), ou pour l’affichage (UI propre, légende visible, boutons adaptatifs...).

Bien sûr, je sais que tout n’est pas parfait. Il y a encore des choses à améliorer, et je pourrais faire certains trucs plus proprement ou plus intelligemment. Mais pour mon niveau actuel, je trouve que c’est un bon projet, j’ai appris énormément en le faisant, et surtout j’ai commencé à avoir une vraie logique de développeur, pas juste coder pour que ça fonctionne, mais coder en pensant à comment le projet tient debout.

Je sais maintenant comment mieux répartir les responsabilités entre les classes, comment éviter les erreurs grâce à des patterns, et comment faire une application qui tient la route. J’ai encore du chemin à faire, mais je repars de ce projet avec beaucoup plus de bases qu’avant.