Projet Java IV

2023

Une image contenant logo

Description générée automatiquement

Réalisé par Popadiuc Claudiu

Supervisé par Monsieur Riggio, Jonathan

Table des matières

[1)Introduction 3](#_Toc135004460)

[2)Présentation de l’interface graphique 4](#_Toc135004461)

[2.1) Interface graphique 4](#_Toc135004462)

[2.2) Optimisation 5](#_Toc135004463)

[2.3) Lettres et nombres de la zone des produits 6](#_Toc135004464)

[2.4) Alerte zone produit complet 7](#_Toc135004465)

[2.5) Alerte zone composant complet 7](#_Toc135004466)

[2.6) Information Emplacements innocupé 8](#_Toc135004467)

[2.7) Information Emplacements occupé 9](#_Toc135004468)

[2.9) Statistiques de l’emplacement 10](#_Toc135004469)

[4)Limitations 12](#_Toc135004470)

[5)Conclusion 13](#_Toc135004471)

# **1)Introduction**

Dans le cadre du cours de JAVA Q IV, nous avons été chargés de développer un logiciel de gestion d'entrepôt. Ce logiciel est conçu pour gérer plusieurs entrepôts automatisés qui s'occupent à la fois de la fabrication et du stockage de produits électroniques, le tout supervisé par un employé qualifié.

Dans ce rapport, je vais commencer par présenter l'interface graphique de manière visuelle et explicative afin de faciliter la compréhension du fonctionnement de l'application.

Ensuite, j'introduirai l'analyse et les design patterns utilisés dans l'application, en expliquant la structure de ma mise en œuvre à l'aide d'outils d'analyse. Je détaillerai également les raisons pour lesquelles j'ai choisi certains design patterns spécifiques pour mon application.

Ensuite, j'aborderai les limitations de mon application et me poserai des questions telles que dans quels cas d'utilisation l'application pourrait-elle ne pas fonctionner comme prévu ? Y a-t-il des aspects techniques qui n'ont pas été pris en compte ? Si j'avais disposé de plus de temps, qu'aurais-je pu améliorer ?

Enfin, je conclurai ce rapport en résumant les principales conclusions et en soulignant les points clés du développement du logiciel de gestion d'entrepôt.

# **2)Présentation de l’interface graphique**

## **2.1) Interface graphique**

Je tiens à préciser que cette interface graphique n'aurait jamais été possible sans l'utilisation du FXML. En effet, j'ai commencé le projet en utilisant le FXML car c'est beaucoup plus simple. Avec le Scene Builder, il était possible de modifier directement le style de l'interface sans avoir à exécuter l'application, comme dans Android Studio. J'ai beaucoup apprécié cette fonctionnalité de JavaFX. Cependant, pour le fichier run.sh, j'ai dû traduire mon code FXML en code Java afin de pouvoir exécuter mon projet.

Une image contenant capture d’écran, Rectangle, carré, Caractère coloré

Description générée automatiquement

Permettez-moi de vous présenter l'interface graphique du logiciel de gestion d'entrepôt, qui offre une vue complète et intuitive de notre système de stockage et de production. Comme vous pouvez le constater, notre entrepôt est équipé de 11 emplacements dédiés au stockage des produits. Ces emplacements sont actuellement utilisés pour la production et le stockage de divers produits et composants.

Dans la partie gauche de l'interface, vous trouverez la zone de stockage des produits. Nous avons déjà enregistré plusieurs types de produits dans notre système. Par exemple, nous avons le P7, un drone de surveillance, le P6, un robot suiveur, le P1, une batterie, le P2, un capteur de mouvement, et le P3, un moteur électrique. Il est également possible d'introduire d'autres produits, tels que le P4, une alarme de sécurité de couleur rouge foncé, ou encore le P5, une voiture télécommandée de couleur orange.

Dans la partie droite de l'interface, vous pouvez observer la zone de stockage des composants. Par défaut, nous avons prévu 8 emplacements pour les composants nécessaires à la fabrication de nos produits. Actuellement, nous disposons d'un composant disponible appelé le "C-Type-2". Ce composant polyvalent nous permet de fabriquer différents produits, ce qui explique la présence des produits déjà fabriqués dans la partie gauche de l'interface. Cependant d’autre types de composants peuvent aussi être dans cette zone de composants tel que le "C-Type-1" et le "C-Type-3" qui sont la batterie et le moteur électrique

Cette visualisation détaillée de l'interface graphique nous offre une vue d'ensemble claire et précise de l'état actuel de notre entrepôt. Elle nous permet de mieux comprendre le flux de production et de suivre facilement le stockage des produits électroniques. En examinant attentivement les différentes zones de stockage et les produits déjà présents, nous pouvons prendre des décisions éclairées concernant la gestion de l'entrepôt et l'optimisation de nos processus de production.

## **2.2) Optimisation**

Une action similaire ce produit lorsque la zone des composants est pleine, à ce moment-là, en revanche la zone n’est pas vidée, mais les produits qui arrive sont tout simplement ignoré.

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

Description générée automatiquement

Si j’appuie sur l’optimisation, une liste d’optimisation se mets à disposition, je peux ensuite choisir l’optimisation de production que je souhaite, ainsi que mettre la production de produits en pause.

## **2.3) Lettres et nombres de la zone des produits**

Une image contenant capture d’écran, Rectangle, carré, Caractère coloré

Description générée automatiquement

Vous pouvez facilement personnaliser l'affichage des colonnes et des lignes en utilisant les boutons « Lettres » et « Nombres ».

Lorsque vous appuyez sur le bouton « Nombres », les lignes et les colonnes de l'interface se transforment en numéros, facilitant ainsi l'identification et la localisation des emplacements de stockage, le nom du bouton correspondant se transforme alors en « Lettres ». Cependant, si vous appuyez à nouveau sur le bouton, les lignes et les colonnes redeviennent des lettres, offrant une représentation plus conventionnelle de l'emplacement des produits dans l'entrepôt. Le bouton retrouve alors son libellé original, soit « Nombres ».

## **2.4) Alerte zone produit complet**

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

Si la partie de gauche, la zone de stockage des produits, se remplit, nous recevons une alerte. Il suffit ensuite d'appuyer sur le bouton "OK" pour vider l'entrepôt en une seule fois.

## **2.5) Alerte zone composant complet**

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

## **2.6) Information Emplacements innocupé**

Une image contenant texte, capture d’écran

Description générée automatiquement

Voici comment l’affichage des informations se présente lorsque l’on appuie sur l’un des produits dans la zone de stockage. Si le produit est vide, l’alerte affichera simplement "Statut inoccupé ".

## **2.7) Information Emplacements occupé**

Une image contenant texte, capture d’écran, Police, Page web

Description générée automatiquement

Si un produit est présent, l'affichage de l'alerte sera différent. On y trouvera le statut "occupé", ainsi que tous les attributs nécessaires à sa création. De plus, deux actions seront disponibles : la possibilité de consulter les statistiques de l'emplacement du produit et la possibilité de vendre le produit. Si on appuie sur vendre produit, un fichier txt sera générer avec les informations du produit.

**2.8) Ticket**

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Cependant si on appuie sur le bouton « voir les statistiques de cet emplacement", une autre fenêtre s’affiche :

## **2.9) Statistiques de l’emplacement**

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Ici, nous pourrons consulter toutes les statistiques de l'emplacement, y compris le nombre de ventes réalisées, le nombre total de produits présents dans cet emplacement, ainsi que le type de produits spécifique à cet emplacement. Comme vous pouvez le voir, il s'agit actuellement d'un drone, mais si je le vends et qu'une batterie arrive à cet emplacement, nous verrons qu'il y aura un total de 2 produits, comprenant les deux types de produits.

**3)Analyse et applications des Design Patterns**

## **3.1) Analyse**

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

Description générée automatiquement

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

Description générée automatiquement

La classe "HELBElectroController" joue le rôle de contrôleur dans mon architecture MVC (Modèle-Vue-Contrôleur). Le contrôleur agit comme un intermédiaire entre le modèle et la vue. Il reçoit les actions de l'utilisateur provenant de la vue, les traite et effectue les opérations nécessaires sur le modèle. Dans mon application, cette classe occupe une place centrale où toute la logique métier s'articule. C'est là que les emplacements pour les zones de stockage de produits ou de composants sont créés, où l'optimisation est gérée, où les alertes sont traitées, et bien d'autres fonctionnalités principales de l'application.

La classe "HELBElectroView" représente la vue dans mon architecture MVC. Elle est responsable de l'affichage des données de l'application et représente l'interface utilisateur avec laquelle l'utilisateur interagit. Elle joue un rôle essentiel dans la présentation des informations.

La classe "Factory" utilise le pattern de conception Fabrique. C'est dans cette classe que toutes les fabrications de produits et de composants se déroulent.

La classe "ProductDetail" est une classe spécifique qui regroupe les fonctionnalités liées aux détails des produits. En divisant mon code de cette manière, je peux mieux organiser ma logique et la rendre plus maintenable. Par exemple, lorsque l'utilisateur clique sur un emplacement, c’est cette classe qui gère la logique des détails du produit correspondant, lui permettant ainsi de le vendre ou de consulter des statistiques supplémentaires.

La classe "Ticket" est une autre classe spécifique qui regroupe la logique de génération d'un fichier .txt. Cette classe facilite la lecture du code en isolant la logique de génération de tickets dans une entité distincte.

L'interface "Optimisation" fournit une interface pour choisir parmi les cinq options d'optimisation disponibles. Elle permet à l'utilisateur de sélectionner une méthode d'optimisation adaptée à ses besoins spécifiques.

L'interface "Component" permet de normaliser les attributs communs à ces composants et de faciliter leur utilisation dans l'application. L'interface "Component" se fait implémentée par les trois types de composants suivants : "ComponentBattery", "ComponentElectricMotor" et "ComponentMotionSensor". Cette interface fournit simplement un nom et une couleur pour l'interface graphique. Chacun de ces trois composants possède des attributs spécifiques à leur création, par exemple, une batterie nécessite une charge spécifique.

Enfin, la classe "Product" est la classe mère des sept types de produits existants. Chaque produit possède les mêmes attributs, mais avec des valeurs différentes, telles que le prix ou l’eco-score. De plus, chaque produit possède une liste des composants nécessaires à sa création, avec leurs propres attributs spécifiques.

## **3.2) Application des Design Patterns**

Factory (Fabrique) :

Le design pattern Factory permet de centraliser la création d'objets dans une classe dédiée appelée "Factory". Au lieu d'instancier directement des objets, on fait appel à la Factory qui se charge de créer et de retourner l'objet approprié en fonction des paramètres ou de la logique définie. Cela permet de simplifier la création d'objets et de rendre le code plus flexible en évitant les dépendances directes entre les classes.

Singleton :

Le design pattern Singleton vise à garantir qu'une classe n'a qu'une seule instance dans toute l'application. Cela se fait en utilisant une méthode statique qui permet d'accéder à cette unique instance. Le Singleton est utile dans les cas où il est important d'avoir une seule et unique instance d'une classe, par exemple pour gérer des ressources partagées ou des configurations globales.

MVC (Modèle-Vue-Contrôleur) :

Le design pattern MVC est un modèle architectural qui sépare la logique de présentation des données et des actions de l'utilisateur. Il se compose de trois composants principaux :

Le Modèle (Model) représente les données et la logique métier de l'application.

La Vue (View) est responsable de l'affichage des données et de l'interface utilisateur.

Le Contrôleur (Controller) agit comme un intermédiaire entre le modèle et la vue, il traite les actions de l'utilisateur et met à jour le modèle en conséquence, ce qui entraîne une mise à jour de la vue.

Créateur (Builder) :

Le design pattern Créateur permet de créer des objets complexes étape par étape. Il fournit une interface fluide et flexible pour créer des objets en permettant de spécifier différentes configurations et options de manière séquentielle. Cela permet de simplifier la création d'objets complexes en évitant une surcharge de constructeurs et en rendant le code plus lisible.

Expert en information (Information Expert) :

Le design pattern Expert en information (ou Expert) consiste à assigner une responsabilité à la classe qui possède le plus d'informations nécessaires pour l'exécuter. L'idée est de regrouper la responsabilité avec les données associées, ce qui permet de réduire les dépendances et de rendre le code plus cohérent. Cette approche favorise le couplage faible entre les classes et facilite la maintenance et l'évolutivité du code.

Principe de faible couplage :

Le principe de faible couplage vise à réduire les dépendances entre les classes et les modules d'un système. Il encourage la conception de classes qui interagissent de manière minimale, en se concentrant sur des interfaces bien définies plutôt que sur des dépendances directes. Un faible couplage permet une plus grande flexibilité, car les modifications apportées à une classe ont moins d'impact sur les autres parties du système.

Stratégie :

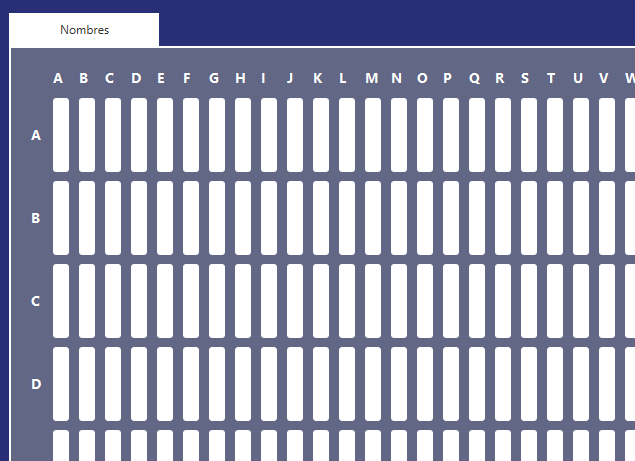
Le design pattern Stratégie permet de définir une famille d'algorithmes interchangeables et encapsulés dans des classes distinctes, tout en permettant aux objets de les utiliser de manière transparente. Il favorise la modularité et la flexibilité du code en permettant de changer dynamiquement le comportement d'un objet

* **Factory** : Permet de centraliser la création de mes objets. J’utilise ce Design Pattern pour créer mes composants et mes produits.
* **Singleton** : Permet de créer une seule instance de la classe via la methode getInstance (). Je l’utilise dans la factory car.
* **MVC** : J’ai utilisé la classe **« HELBElectroView »** qui représente la vue, la classe « **HELBElectroController** » qui sert de Controller et les autres classes servent pour le modèle.
* **Créateur** : Qui détient la responsabilité de créer la vue de mon interface graphique avec tout ce qui se tient de la Grid, Vbox, Boutton, etc... La classe **« HELBElectroView »** a la responsabilité de création de tout cela car c’est cette classe qui s’occupe de l’affichage.
* Créateur : Qui détient la responsabilité de créer les boutons ? La classe « InterfaceGraphic » a la responsabilité de création des boutons car c’est cette classe qui s’occupe de l’affichage.
* Expert en information : principe d’attribuer la responsabilité aux objets. Le fait que les emplacements remplissent le parking. Le parking contient toutes les informations des emplacements donc c’est logique qu’elles soient créées dans celui-ci (ControllerParking).
* Principe de faible couplage : si on effectue une modification, cela ne devrait pas impacter d’autres classes. Il faut réduire au minimum le nombre de modification à effectuer.
* MVC : j’ai utilisé la classe « IntrefaceGraphic » qui représente la vue, la classe « ControllerParking » qui sert de Controller et les autres classes servent pour le modèle.
* Contrôleur : la classe « ControllerParking ». Elle représente le système dans sa globalité. Par exemple, le fait que lorsqu’on ajoute un véhicule et que l’emplacement change de couleur, cela crée des changements dans les données.
* Singleton : permet de créer une seule instance de classe.
* Fabrique : permet de centraliser la création de mes objets. J’utilise ce pattern pour créer mes véhicules. S’il y a d’autres objets à créer, c’est dans cette classe que ça se passe.
* Stratégie : le comportement des prix varie selon les jours. Ils évoluent donc dynamiquement à l’exécution.

# **4)Limitations**

## **4.1) Dans quels cas l'application pourrait-elle ne pas fonctionner comme prévu ?**

* Le nombre d'emplacements de produits est par défaut de 11. Mon programme permet d'en ajouter autant que désiré, mais si l'on dépasse déjà 50 colonnes et lignes, cela devient chaotique et il devient difficile d'utiliser le programme. Il n'est donc pas adapté pour dépasser une vingtaine de produits en termes de partie graphique.



* Le comportement de la simulation est normalement bien fonctionnel mais ça n’a pas été vérifier d’a à z, il se peut donc qu’il y a des incohérences dans le remplissage des composants
* Lorsque la zone de stockage de composants est pleine, nous recevons une alerte pour le prochain composant qui essaie d'entrer dans la zone mais qui est ignoré. Je reçois cette alerte pour chaque composant, donc si je ne produis rien pendant quelques minutes, je vais recevoir autant d'alertes que de composants ignorés.
* Si une erreur se produit dans le fichier "helbelectro.data", le composant est simplement ignoré, même s'il s'agit d'une simple erreur d'espace ou de virgule. Cela peut poser un problème si nous recevons un fichier avec des erreurs qui ne sont pas intentionnelles, car aucun composant ne pourra être créé.

## **4.2) Y a-t-il des aspects techniques qui n'ont pas été pris en compte ?**

* Je n’ai pas du tout implémenté les tests unitaires.
* Lorsque je vends un produit, j’ai un compteur qui est incrémenter pour les statistiques sauf que tous les emplacements prennent ce compteur en compte, donc s’il y a une vente en 0,0 et une vente en 2,3, les 2 emplacements afficheront 2 ventes.
* Mon interface graphique n’est pas compatible avec les noms des lignes et colonnes en lettres pour plus de 26 car mon système ne permet pas d’afficher [AA, A] ça ira seulement jusque-là lettre [Z, Z] équivalent de [26, 26].
* L’optimisation diverse devrait fonctionner mais ca ne fonctionne pas comme prévu, et il est difficile de savoir cela avec certitude.

## **4.3) Si j'avais disposé de plus de temps, qu'aurais-je pu améliorer ?**

* J’aurais pu grandement améliorer l’interface graphique car c’est ce que j’aime plus dans le code.

# **5)Conclusion**

Pour conclure ce rapport, après plus d'un mois et demi de travail et environ 100 heures de travail, j'ai enfin terminé le projet de Java IV. Ce projet m'a demandé énormément de temps, mais il m'a permis de comprendre tant de choses. En effet, il y avait des concepts que je n'avais pas encore bien saisis lors du projet de l'aquarium dans le domaine de la programmation orientée objet. Grâce à ce projet, j'ai beaucoup mieux compris la programmation orientée objet, l'utilisation des classes mères et filles, les liens statiques, les getters et les setters, ainsi que tous les aspects du Q3. J'ai pu implémenter de nombreuses fonctionnalités dans ce projet, ce qui m'a ouvert les yeux sur ce que j'avais réalisé lors du Q3.

Beaucoup d'élèves se plaignent de la pression de ce cours, mais personnellement, je trouve que ça va si on s'y prend à temps, comme je l'ai fait. Cependant, si j'ai une chose à critiquer, c'est celle-ci : pourquoi avoir ajouté les tests unitaires une semaine et demie avant les examens de révision ? Sachant que plus de la moitié des étudiants n'ont toujours pas commencé, cela les décourage encore plus de ne pas commencer. Cela peut avoir un impact sur le moral, car parfois ce sont les petits détails qui jouent. Donc, je préfère le dire, ce serait mieux si vous voulez ajouter les tests, de les ajouter au moment où vous nous donnez l'énoncé, ou alors de ne pas les inclure du tout. Ajouter une charge de travail supplémentaire une semaine et demie avant les examens de révision, alors que certains projets doivent déjà être finalisés et rendus, ce n'est pas une très bonne idée. Je comprends que nous devrions les implémenter dans le projet car c'est un concept abordé dans le cours, et c'est une bonne chose. Cependant, ne le faites pas aussi tard.

Sinon il y a de très bons points positifs, j'ai également appris à utiliser les interfaces et les design patterns, qui peuvent sembler compliqués au premier abord, mais qui deviennent plus simples dès qu'on les comprend bien. Personnellement, j'avais peur de ce projet au début, car je ne suis pas du genre à être très attentif aux cours théoriques. Je n'avais donc presque rien compris aux design patterns, je ne savais même pas ce qu'était un singleton avant de le mettre en œuvre dans mon projet. Finalement, ce n'était pas si compliqué à comprendre et à implémenter. Certains design patterns, comme le MVC, sont vraiment utiles et simplifient la vie des programmeurs. Ils permettent d'avoir un code très maintenable et propre, ce sont des pratiques que j'apprécie, donc je vais probablement continuer à coder de cette manière lorsque l'occasion se présentera.

J'ai également appris à utiliser JavaFX, qui, à mon avis, est bien meilleur que Swing. JavaFX offre beaucoup plus de fonctionnalités, son interface graphique est nettement meilleure et surtout, le lien entre le contrôleur et le fichier FXML est vraiment très simple. Malheureusement, pour ce projet, en particulier pour le fichier run.sh, j'ai dû supprimer le fichier FXML. Cependant, ce que j'aimais le plus avec JavaFX, c'était quand même le FXML. C'était tellement simple à utiliser. Comme je l'ai mentionné précédemment, je n'aurais jamais réussi à créer l'interface graphique que vous voyez sans le FXML. Heureusement que j'ai commencé avec cette approche, sinon j'aurais eu une interface assez simple et peu attrayante. Tester chaque ligne de code peut parfois être très compliqué, tandis qu'avec le FXML, cela se faisait directement dans le Scene Builder.

Pour terminer cette conclusion, je tiens à vous remercier de m'avoir donné l'opportunité d'apprendre autant de nouvelles choses dans ce projet.