Projet Java IV
2023

Réalisé par Popadiuc Claudiu

Supervisé par Monsieur Riggio, Jonathan

Table des matières

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.

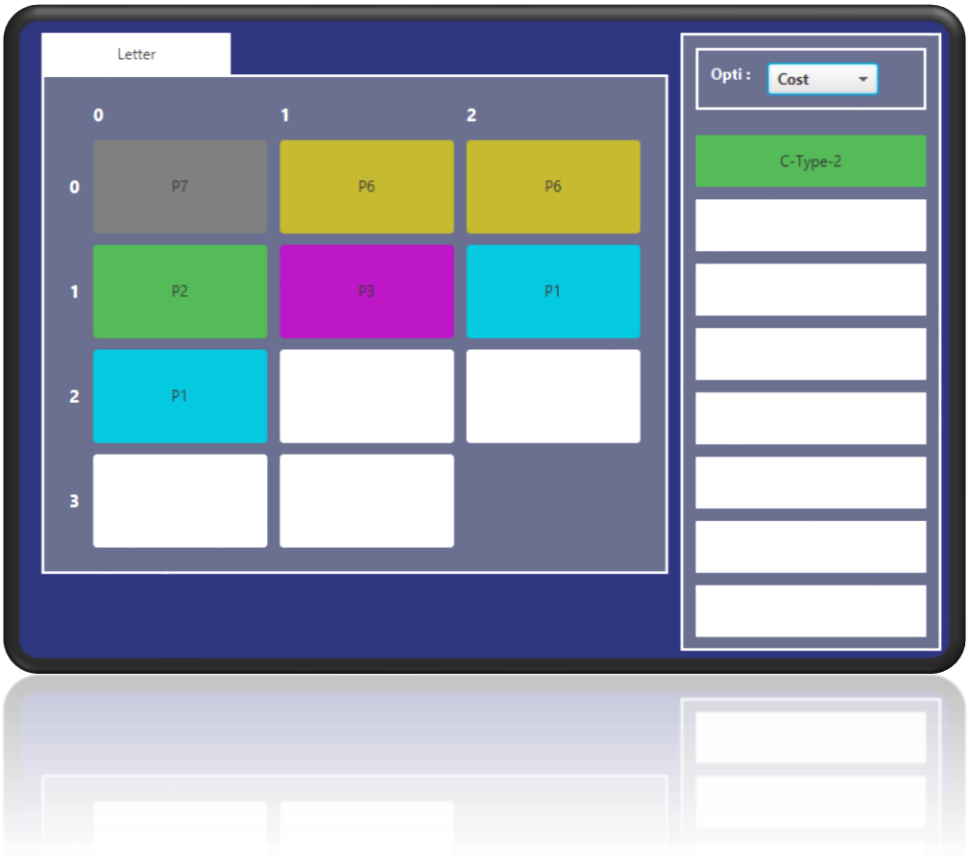
Ensuite, j'aborderai les limitations de mon application et me poserai des questions telles que dans quels cas d'utilisation l'application pourrait-elle ne pas fonctionner comme prévu ? Y a-t-il des aspects techniques qui n'ont pas été pris en compte ? Si j'avais disposé de plus de temps, qu'aurais-je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



Voici l'interface graphique du logiciel de gestion d'entrepôt, qui offre une vue de mon système de stockage de composants et de produits. Comme vous pouvez le constater, l’entrepôt est équipé de 11 emplacements dédiés au stockage des produits. Ces emplacements sont utilisés pour la production et le stockage de divers produits et composants.

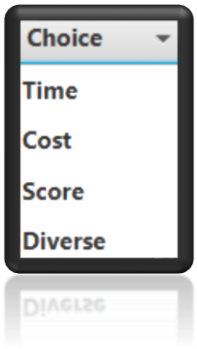
Dans la partie gauche de l'interface, vous trouverez la zone de stockage des produits. Il y a déjà plusieurs types de produits. Par exemple, nous avons le P7, un drone de surveillance, le P6, un robot suiveur, le P1, une batterie, le P2, un capteur de mouvement, et le P3, un moteur

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

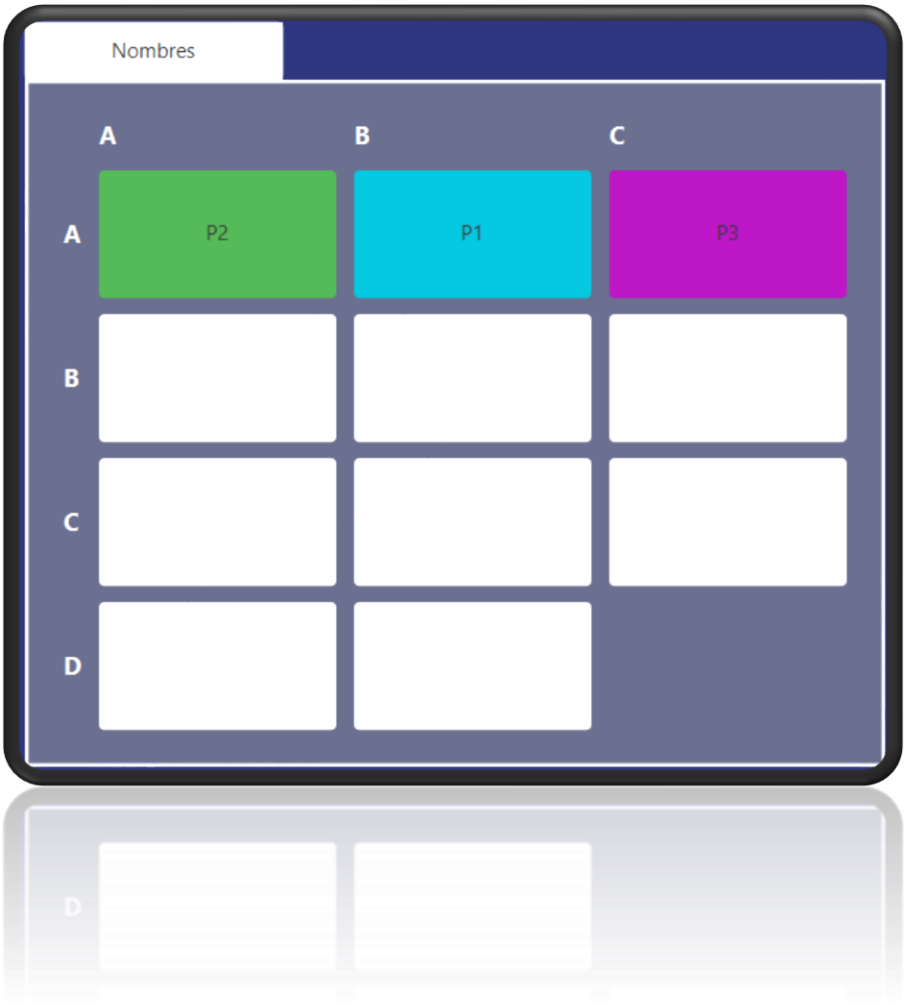
Cette visualisation détaillée de l'interface graphique nous offre une vue d'ensemble claire et précise de l'état actuel de notre entrepôt. Elle nous permet de mieux comprendre le flux de production et de suivre facilement le stockage des produits électroniques. En examinant attentivement les différentes zones de stockage et les produits déjà présents.

2.2) Optimisation



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.

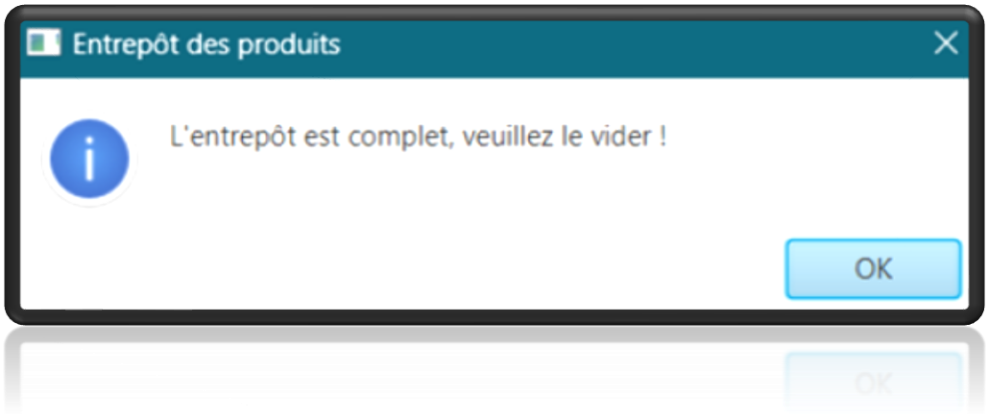
2.3) Lettres et nombres de la zone des produits



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

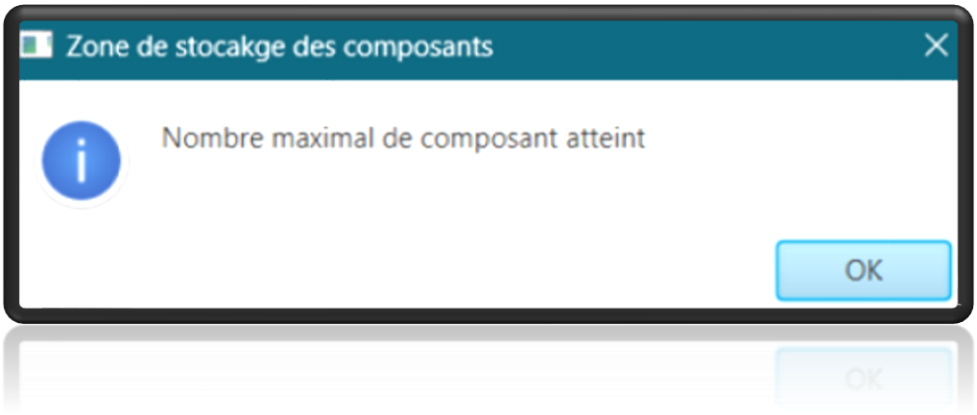
Lorsque vous appuyez sur le bouton « Nombres », les lignes et les colonnes de l'interface se transforment en numéros, facilitant ainsi l'identification et la localisation des emplacements de stockage, le nom du bouton correspondant se transforme alors en « Lettres ». Cependant, si vous appuyez à nouveau 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



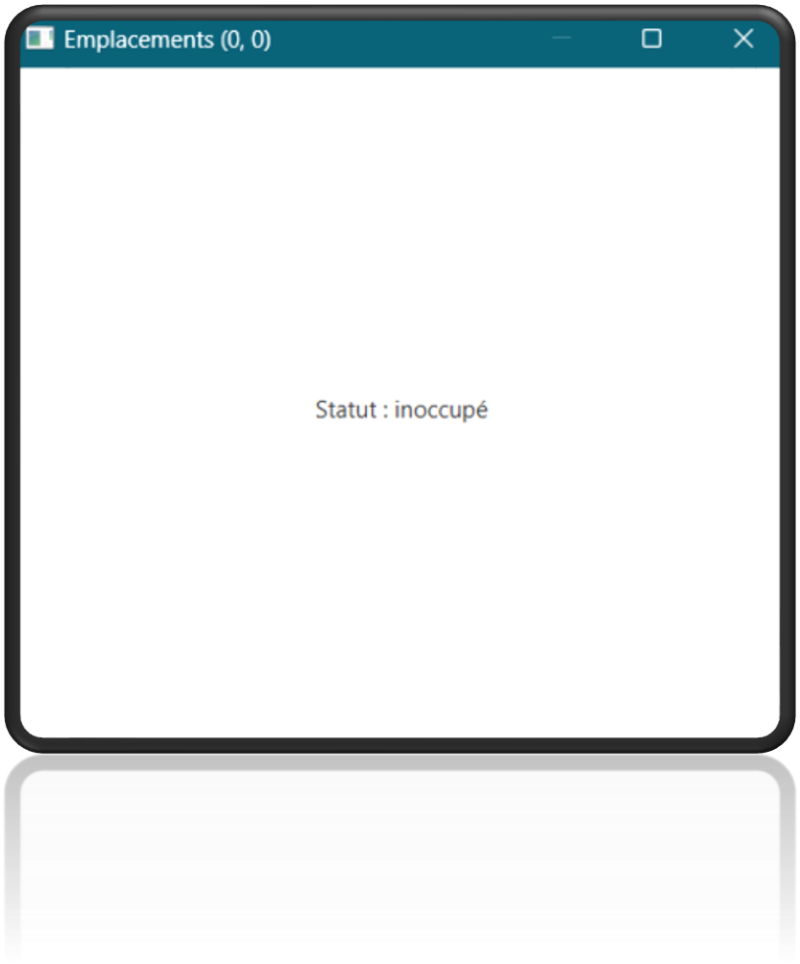
Si la partie de gauche, la zone de stockage des produits, se remplit, nous recevons une alerte. Il suffit ensuite d'appuyer sur le bouton "OK" pour vider l'entrepôt en une seule fois.

2.5) Alerte zone composant complet



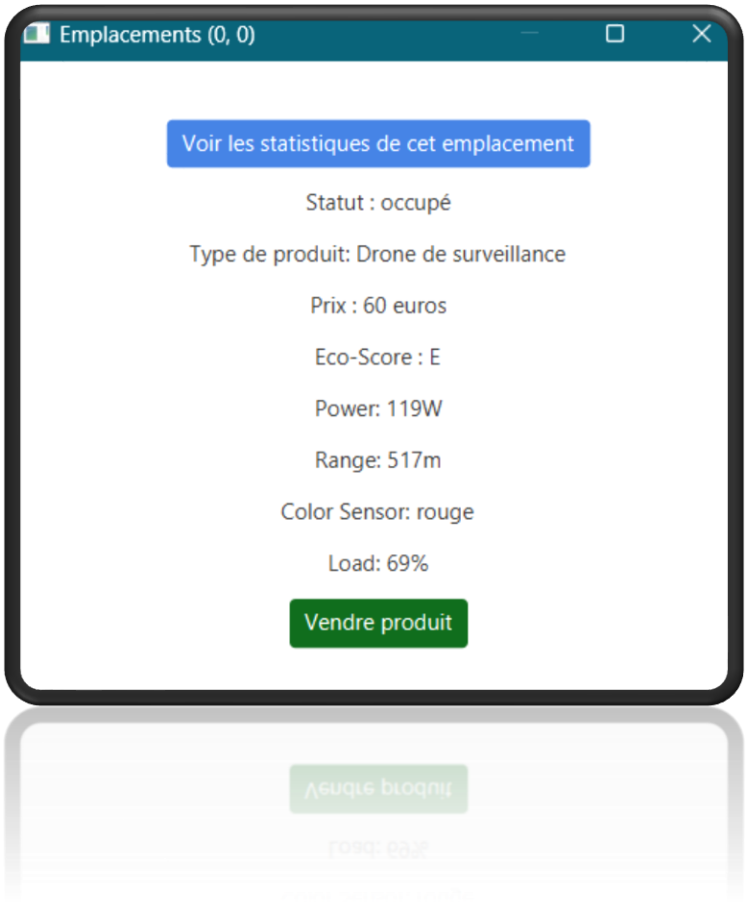
Une action similaire ce produit lorsque la zone des composants est pleine, à ce moment-là, en revanche la zone n'est pas vidée, mais les produits qui arrive sont tout simplement ignoré.

2.6) Information Emplacements innocupé



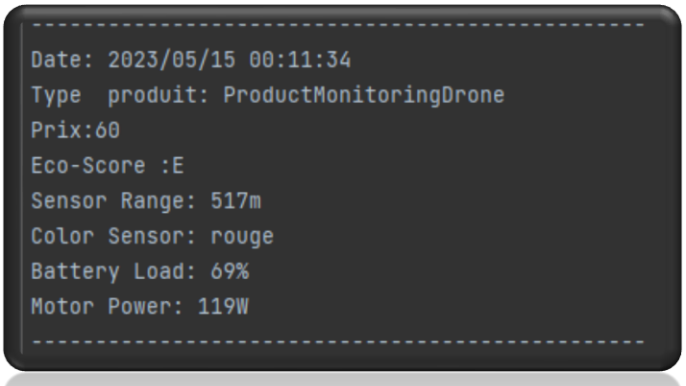
Voici comment l'affichage des informations se présente lorsque l'on appuie sur l'un des produits dans la zone de stockage. Si le produit est vide, l'alerte affichera simplement "Statut inoccupé ".

2.7) Information Emplacements occupé



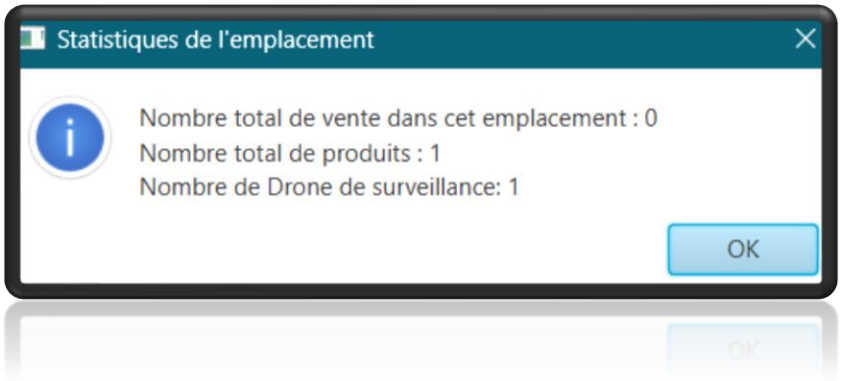
Si un produit est présent, l'affichage de l'alerte sera différent. On y trouvera le statut "occupé", ainsi que tous les attributs nécessaires à sa création. De plus, deux actions seront disponibles : la possibilité de consulter les statistiques de l'emplacement du produit et la possibilité de vendre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

2.8) Ticket



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

2.9) Statistiques de l'emplacement

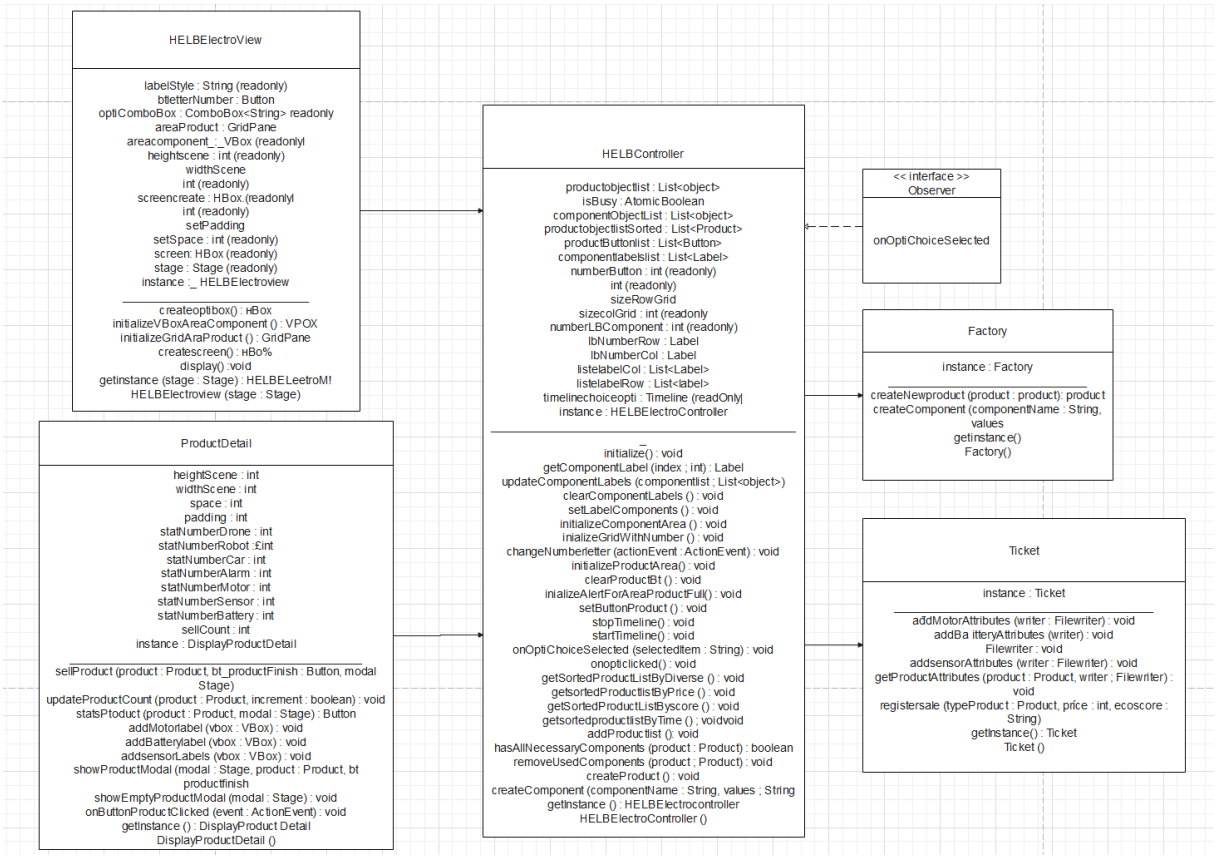


Ici, nous pourrons consulter toutes les statistiques de l'emplacement, y compris le nombre de ventes réalisées, le nombre total de produits présents dans cet emplacement, ainsi que le type de produits spécifique à cet emplacement. Comme vous pouvez le voir, il s'agit actuellement d'un drone, mais si 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

1)



1) 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 gère la logique métier. 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 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.

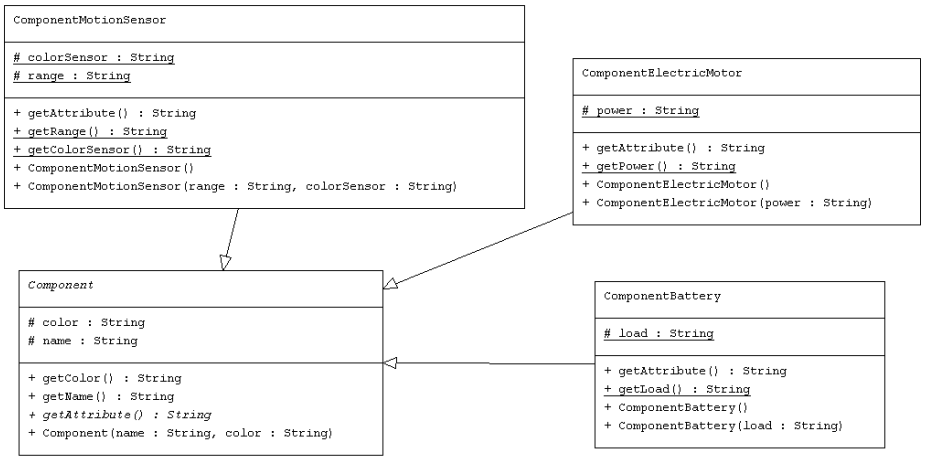
   La classe "Factory" utilise le pattern de conception Fabrique. C'est dans cette classe que toutes les fabrications de produits et de composants se déroulent.

   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 "Observer" fournit une interface pour choisir parmi les quatres options d'optimisation disponibles. Elle permet à l'utilisateur de sélectionner une méthode d'optimisation adaptée à ses besoins spécifiques. Elle fait partie de mon Design Pattern « Observer ».

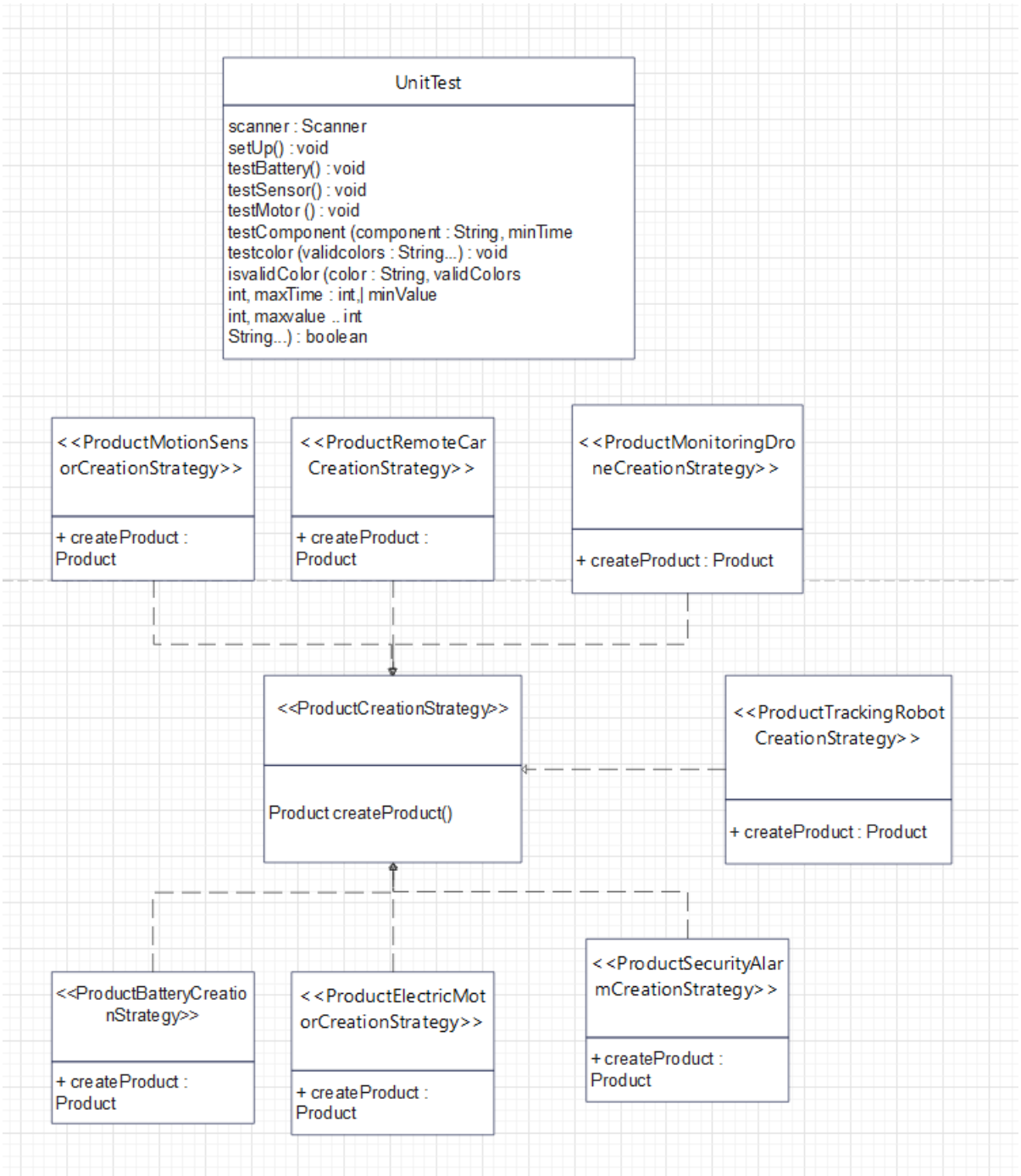
2)



Component -> Factory

2) La class "Component" se fait implémentée par les trois types de composants suivants : "ComponentBattery", "ComponentElectricMotor" et "ComponentMotionSensor". Cette classe 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. Ces composants sont créés depuis ma Factory.

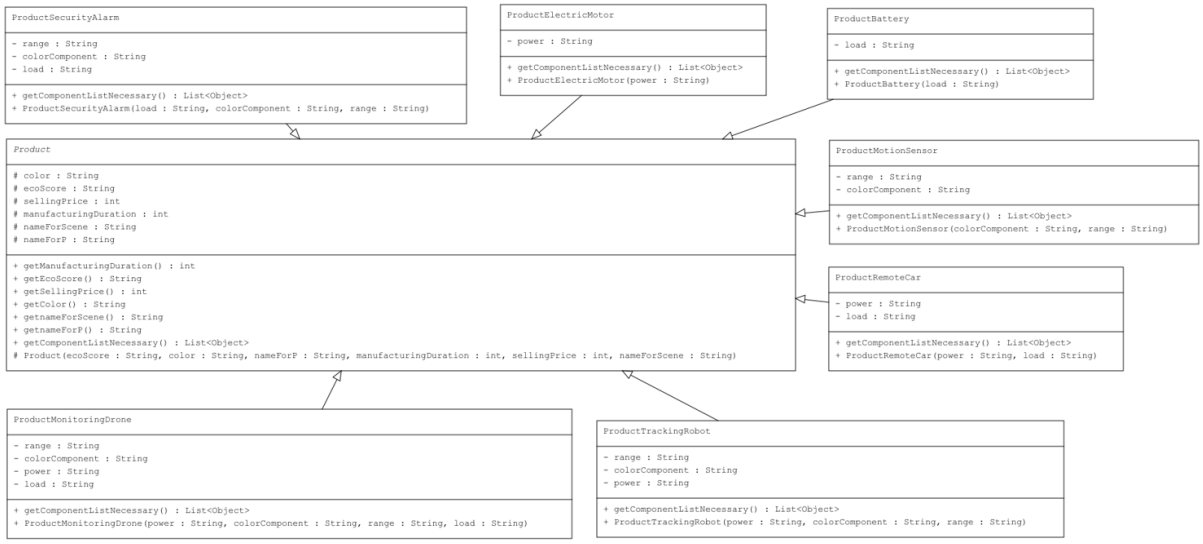
3)



3) Je dispose également d'une interface appelée "ProductCreationStrategy". Cette interface sera utilisée pour le modèle de conception "Strategy" et permettra de renvoyer une méthode de création de produit. Les 7 types de produits implémenteront cette interface, et chaque produit pourra avoir sa propre façon de création.

Enfin, la classe "UnitTest", qui se trouvera dans un package différent. Cette classe permettra de tester le fichier de simulation avec 3 tests distincts, chacun testant les 3 types de composants.

4)



Product -> Factory

4) Pour poursuivre, 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. Ces produits sont créés depuis ma Factory.

3.2) Application des Design Patterns

Factory :

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.

Le design pattern "Factory" est utilisé pour créer des objets sans exposer la logique de création directement dans le code client. Dans mon code, la méthode "createComponent" de la classe "Factory" est une méthode de création de composants. Elle utilise un paramètre "componentName" pour déterminer quel composant doit être créé et renvoie une instance du composant correspondant. En utilisant ce pattern, j'encapsule la logique de création des composants dans la classe "Factory", ce qui facilite la création de nouveaux composants ou la modification de la logique de création sans affecter le reste du code.

Strategy :

Le design pattern" Strategy" est utilisé pour définir une famille d'algorithmes, encapsuler chacun d'eux et les rendre interchangeables. Dans mon code, j'utilise ce pattern pour créer des produits à l'aide de différentes stratégies de création. La classe "Factory" contient la méthode " createNewProduct" qui prend un objet Product en argument et utilise des stratégies de création spécifiques en fonction du type de produit. Chaque stratégie de création implémente l'interface "ProductCreationStrategy" et fournit une implémentation de la méthode" createProduct". En utilisant ce pattern, je peux ajouter de nouveaux types de produits et de nouvelles stratégies de création sans modifier directement la classe "Factory" et en respectant le principe d'ouverture/fermeture.

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.

Dans mon code, j'ai implémenté la classe "Factory" en tant que Singleton en utilisant la méthode "getInstance". Cela permet d'assurer qu'il n'y aura qu'une seule instance de la classe "Factory" dans mon application. En déclarant la méthode " getInstance" comme statique, je peux y accéder directement à partir de n'importe quel endroit de mon code, sans avoir besoin de créer une instance de la classe " Factory". Cela facilite l'accès à l'instance unique de la classe.

J’ai aussi utilisé ce pattern pour mes classes "HELBElectroController","HELBElectroView", "ProductDetail".

Observer :

Le design pattern "Observer" est un modèle de conception comportemental qui permet d'établir une relation de dépendance entre des objets, de sorte que lorsqu'un objet change d'état, tous les objets qui en dépendent sont notifiés et mis à jour automatiquement.

Dans mon application, l’interface "Observer " définit une méthode "onOptiChoiceSelected "qui sera appelée lorsque le choix d'optimisation est sélectionné dans la vue. La classe "HELBElectroController" implémente l'interface Observer et définit la méthode. Cette méthode est responsable de traiter le choix d'optimisation sélectionné et d'effectuer les actions appropriées. Lorsque le choix d'optimisation est modifié dans la vue, la méthode "notifyOptiComboBoxObservers " est appelée. Cette méthode itère sur tous les observateurs enregistrés et appelle leur méthode "onOptiChoiceSelected " avec le choix d'optimisation sélectionné en tant que paramètre.

MVC (Modèle-Vue-Contrôleur) :

Le modèle MVC (Modèle-Vue-Contrôleur) est un modèle de conception utilisé pour organiser et structurer le code d'une application.

Le modèle MVC est utilisé pour séparer les préoccupations et assurer une meilleure organisation et une meilleure maintenabilité du code. Il permet de clarifier les responsabilités de chaque composant et facilite les modifications ultérieures. Par exemple, si je souhaite modifier l'interface utilisateur, vous pouvez le faire sans avoir à modifier la logique métier du modèle. Le modèle MVC divise l'application en trois composants principaux :

- Le modèle : Il représente les données de l’application. Dans mon code, les classes telles que "Product" et toutes ces classes filles, font partie du modèle.
- La vue : Elle est responsable de l'affichage des données au travers de l'interface utilisateur. Dans mon code, la classe "HELBElectroView" est la vue. Elle crée et affiche les composants et les produits dans l'interface graphique.
- Le contrôleur : Il agit comme un intermédiaire entre le modèle et la vue. Il reçoit les actions de l'utilisateur depuis la vue, interagit avec le modèle en conséquence, puis met à jour la vue en conséquence. Dans mon code, la classe "HELBElectroController" agit en tant que contrôleur. Elle reçoit les actions de l'utilisateur, telles que le choix de création de composants ou de produits, et interagit avec le modèle en conséquence.

Expert en information :

Le design pattern Expert en information 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.

Dans mon code on peut identifier des éléments qui respectent ce principe, dans la méthode "hasAllNecessaryComponents" la responsabilité de vérifier si tous les composants nécessaires à la création d'un produit sont disponibles est attribuée à " HELBElectroController". Il possède les informations sur la liste des composants disponibles "componentObjectList", et il utilise ces informations pour vérifier si chaque composant nécessaire est présent. Ainsi, "HELBElectroController" est l'expert en matière d'informations sur les composants et est responsable de la vérification.

En attribuant ces responsabilités à "HELBElectroController", le code respecte le principe de l'expertise, car chaque objet est responsable de la manipulation des informations dont il dispose. Cela favorise un découplage entre les objets et une meilleure encapsulation des responsabilités.

Principe de faible couplage :

Le principe de faible couplage est un concept de conception qui vise à réduire les dépendances entre les différents modules d'un système. Il s'agit de minimiser les interactions et les interdépendances entre les différentes parties d'un système, de sorte que chaque module puisse fonctionner de manière indépendante et être modifié sans avoir un impact significatif sur les autres modules.

Dans mon code, le principe de faible couplage peut être observé dans plusieurs aspects :

- Utilisation de l'interface Component : mon code utilise une interface Component pour représenter les composants. Cela permet de réduire le couplage en permettant à la classe "HELBElectroController" d'interagir avec les composants de manière générique, sans avoir à connaître les détails spécifiques de chaque implémentation de composant.
- Utilisation de la classe "Factory" : La classe "HELBElectroController" utilise la classe "Factory" pour créer des instances de composants et de produits. Cette approche permet d'encapsuler la création d'objets et de réduire le couplage en évitant la dépendance directe entre "HELBElectroController" et les classes concrètes de composants et de produits.

Principe de forte cohésion :

La forte cohésion signifie que les éléments d'un module sont étroitement liés et travaillent ensemble pour accomplir une tâche spécifique. Les responsabilités sont clairement définies et chaque module est spécialisé dans une fonction particulière. Voici quelque exemple de comment dans mon code, la forte cohésion est appliquée :

- La méthode "createComponent" dans la classe "HELBElectroController" crée un composant en utilisant la classe "Factory" et ajoute ce composant à une liste interne.

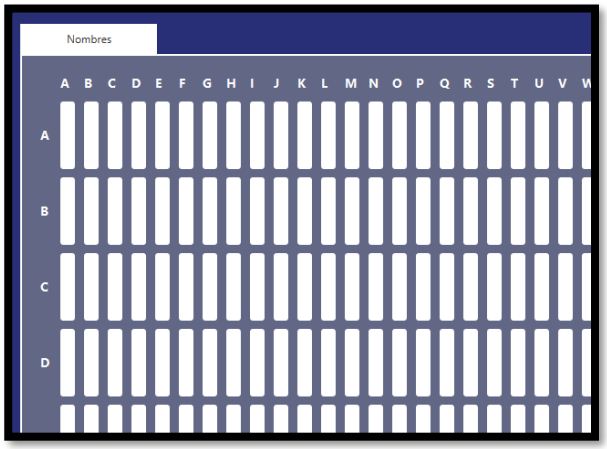
Cette méthode est spécifiquement responsable de la création et de l'ajout des composants, ce qui démontre une cohésion élevée.

- La méthode "registerSale" dans la classe Ticket génère un ticket de vente en fonction du type de produit spécifié. Chaque type de produit est traité individuellement, ce qui démontre une cohésion élevée.

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, ce serait dommage de ne pas pouvoir crée de composant pour si peu, et cela peut poser un problème si nous recevons un fichier avec des erreurs qui ne sont pas intentionnelles, aucun composant ne pourra être créé.
- Si on vend des milliers de produits, ce sera compliqué à maintenir car tous les fichier txt serait dans le dossier du projet et on ne s'y retrouvera plus.

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

- Je n'ai pas réussi à implémenter correctement 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].
- Lorsqu’il faut vider complètement l’entrepôt, toutes les données des emplacements de chaque emplacement est supprimé, et j’ai un problème avec cela, c’est que certaines statistiques du nombre de composant et de quel type se supprime aussi.

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 le blocus ? Sachant que plus de la moitié des étudiants n'ont toujours pas commencé le projet, cela les décourage encore plus de ne pas commencer. Cela peut avoir un impact sur le moral, car parfois ce sont des 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 le blocus, 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 Scène 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, et c'est vraiment maintenant que je me suis débloqué sur pleins de termes que j'avais du mal à comprendre auparavant.