Le génie pour l'industrie

Département de génie logiciel et des TI

Rapport de laboratoire

N° de laboratoire	Laboratoire 1
Étudiant	Quezada, Jonnie Klein
Code permanent	QUEJ19129809
Cours	LOG121
Session	A2021
Groupe	04
Professeur	Benoit Galarneau
Chargé de laboratoire	Bianca Popa
Date de remise	2021-10-19

1. INTRODUCTION

Il y a une certaine élégance dans la conception méticuleuse de toutes les subtilités d'un système. Certes, on peut simplement ignorer la phase de conception et passer directement au codage. Une idée fausse courante ici serait de croire que cela permettrait d'économiser beaucoup plus de temps au lieu d'allouer du temps à une planification appropriée. Cependant, comme cela a été mis en évidence au cours de ce laboratoire, il est plus coûteux de réparer les choses plus vous allez en profondeur dans le cycle de vie du développement que lors de la phase de conception. Dans le but de créer une simulation d'une ligne de production, il existe un besoin urgent d'une conception standardisée du système. L'utilisation de différents patrons de conception ainsi que l'exploitation du paradigme OOP ont été cruciales pour garantir l'intégrité de l'application et prévenir, autant que possible, les complications en cours de route. Dans ce rapport, je présenterai la structure générale résultant des décisions prises concernant la conception du système à travers l'utilisation de diagrammes UML. De plus, il y aura des détails et des justifications concernant les décisions prises pendant la phase de conception et de mise en œuvre.

2. Conception

2.1 Choix et responsabilités des classes

Classe	*Responsabilités*	*Dépendances*
Facility	Classe qui représente toute installation au sein du réseau, telle qu'une usine et un entrepôt	• IObserver • Component • FacilityConfig • IndicatorStatus • ISubject
Warehouse	Sous-classe de Facility et représente l'entrepôt où sont stockés les avions	• Facility • ISellBehavior • FacilityConfig
Factory	Sous-classe abstraite de Facility et représente les usines du réseau. Nécessite que ses sous-classes implémentent craftComponent()	• Facility • FacilityConfig
MetalFactory	Sous-classe de Factory et représente une usine qui fabrique des composants métalliques. Implémente craftComponent() pour fabriquer des composants métalliques	• Factory • FacilityConfig
MotorFactory	Sous-classe de Factory et représente une usine qui fabrique des composants de moteur. Implémente craftComponent() pour fabriquer des composants de moteur	• Factory • FacilityConfig
WingFactory	Sous-classe de Factory et représente une usine qui fabrique des composants d'aile. Implémente craftComponent() pour fabriquer des composants	• Factory • FacilityConfig

	d'aile	
PlaneFactory	Sous-classe de Factory et représente une usine qui fabrique des avions. Implémente craftComponent() pour fabriquer des avions	• Factory • FacilityConfig
IObserver	Interface qui applique le comportement permettant aux classes d'observer et d'agir par la suite sur les changements dans l'état d'une classe qui implémente IObserver.	
ISubject	Interface qui applique le comportement permettant aux classes de notifier ses observateurs chaque fois que son état change	
GlobalState	Classe singleton pour représenter l'état global du réseau et implémente IObserver	• Facility • Component • Pathing
XMLUtils	Classe d'assistance pour la lecture et l'analyse du document XML de configuration	• XMLToolKit • FacilityConfig • FacilityCoordinates • FacilityMetadata • Pathing
ComponentType	Enum qui représente les différents types de composants Component ainsi que le chemin de l'icône associé	
IndicatorStatus	Enum qui représente les différents statuts d'une instance de Facility	

XMLToolKit	Record englobant les détails pour la fonction d'assistance XMLUtils#getToolKit(String, String)	
Component	Record englobant les métadonnées d'un composant qu'une instance de Facility peut recevoir	
IconMetadata	Record englobant le type de Facility et le chemin pour ses icônes	
Pathing	Record englobant les coordonnées qu'un composant doit voyager d'un Facility à un Facility de destination.	
FacilityConfig	Record englobant FacilityCoordinates et FacilityMetadata.	
FacilityCoordinates	Record englobant les données coordonnées d'une instance de Facility	
FacilityEntryComponent	Record englobant des données sur les composants qu'une instance de Facility peut recevoir	
FacilityMetadata	Record englobant les métadonnées d'une instance de Facility	
ISellBehavior	Interface qui applique le comportement de vente sélectionné par l'utilisateur	
FixedIntervalSellStrategy	Classe qui implémente ISellBehavior et définit l'exécution de la vente comme un intervalle fixe	• ISellBehavior • Component

RandomizedSellStrategy	Classe qui implémente ISellBehavior et définit l'exécution de la vente comme aléatoire	• ISellBehavior • Component
Environnement	Classe qui représente l'environnement de travail du réseau et de la simulation	• GlobalState
FenetrePrincipale	Classe qui représente la fenêtre principale de l'application	• PanneauPrincipal • MenuFenetre
FenetreStrategie	Représente la fenêtre pour la sélection de la stratégie de vente	
MenuFenetre	Représente la fenêtre de menu pour sélectionner le fichier de configuration. Implémente ISubject	• ISubject
PanneauPrincipal	Classe qui représente le panneau principal de l'application	• GlobalState
PanneauStrategie	Représente le panneau pour la sélection de la stratégie de vente	• GlobalState • RandomizedSellStrategy • FixedIntervalSellStrategy
Simulation	Classe d'entrée de l'application	• Environnement • FenetrePrincipale

2.2 Diagrammes des classes

Voir class-diagram.png

2.3 Faiblesses de la conception

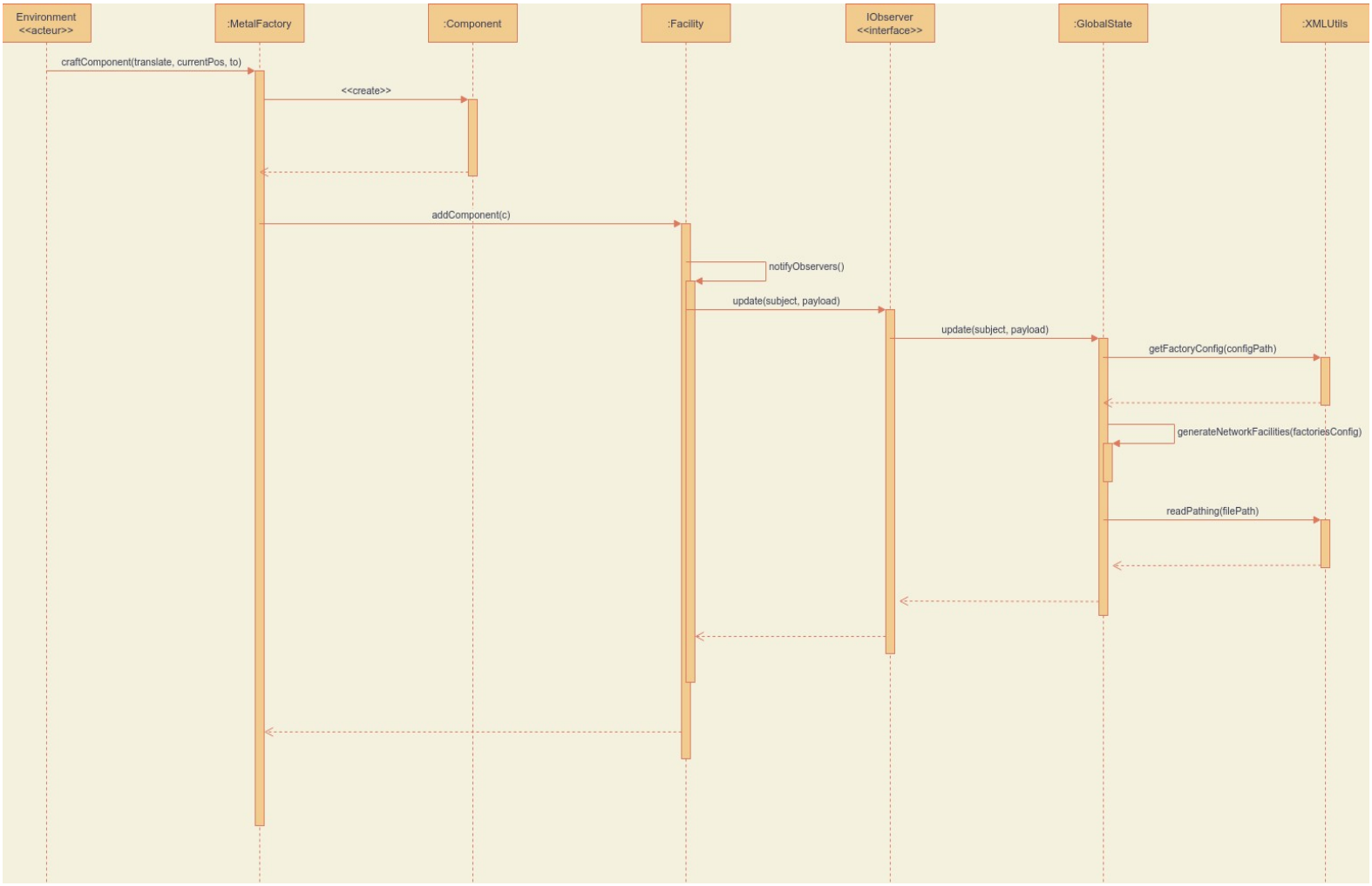
Bien que j'ai passé beaucoup de temps à réfléchir à la structure du projet et aux patrons de conception à utiliser pour maximiser l'efficacité de la gestion et du transfert des données tout en restant fidèle aux normes de l'industrie telles que l'utilisation des classes Record et de quelques modèles de programmation fonctionnels (FP), j'ai triché en revenant à un état global. J'ai décidé d'utiliser un singleton GlobalState pour stocker et muter l'état de l'application (composants en route, état du stock de chaque installation, etc.).

Bien que des choses comme l'architecture Flux de Facebook et l'implémentation Flux de React.js, Redux, utilisent des états globaux, la façon dont j'ai implémenté GlobalState a été trop simpliste et muable. Les effets secondaires encourus auraient rendu le débogage un souvenir douloureux. De plus, il rompt l'encapsulation, ce qui le rend non conforme aux principes de la POO. En fait, les classes externes peuvent accéder et muter les propriétés d'un GlobalState, ce qui met en évidence la rupture de l'encapsulation et de la mutabilité. Bien que je souhaitais résoudre ces problèmes au cours du développement, je n'avais pas assez de temps pour le faire.

Néanmoins, cela aurait pu être corrigé soit en s'assurant que GlobalState reste un singleton immuable, soit en utilisant l'injection de dépendance. En tant que singleton immuable, l'état de GlobalState est garanti de ne pas muter et est donc intrinsèquement thread-safe, empêchant les bogues résultant d'un état global mutable ainsi que les problèmes de concurrence. L'alternative serait d'utiliser l'injection de dépendance. Lors de la création d'un objet, un objet reçoit toutes les informations nécessaires via son constructeur. Une autre technique serait d'utiliser des interfaces pour l'injection (Fowler, 2004).

2.4 Diagramme de séquence (uml)

2.4.1. Exemple qui illustre la dynamique du patron observateur



Voir observer-pattern.png

Pour chaque Composant qu'une installation crée et ajoute ainsi à son stock, les observateurs sont notifiés et peuvent ainsi effectuer les changements nécessaires en cas de changement d'état. Dans cette situation spécifique, MetalFactory crée une nouvelle instance de Component de type ComponentType.Metal.

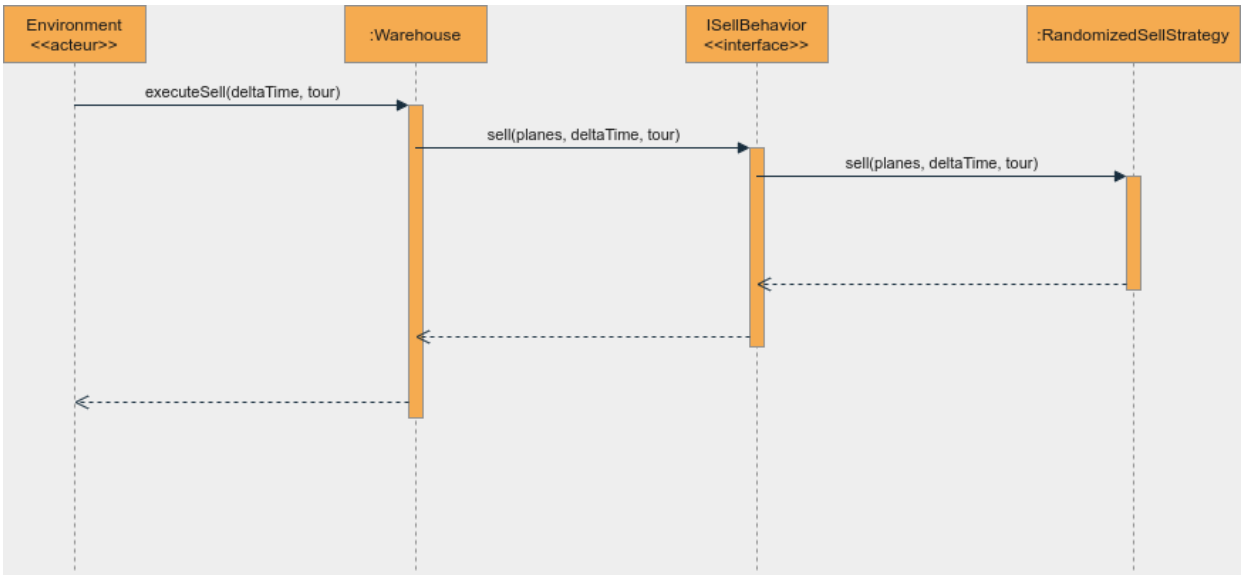
À chaque tournée, le statut de production de MetalFactory augmente. Une fois l'état de production au plus haut, Environnement commande un nouveau Component de type ComponentType.Metal à créer par MetalFactory. Environnement appellerait la méthode craftComponent(translate, currentPos, to) de MetalFactory, en fournissant le facteur de translation (quantité d'unités x et y qu'un composant se déplace dans un plan), currentPos (les coordonnées de la MetalFactory qui l'a créé) et à (les coordonnées de l'Usine de destination).

À la suite de l'appel de craftComponent(Point, Point, Point), MetalFactory <réera un nouveau composant, puis ajoutera ce composant à son stock héritée {stock: ArrayList<Component>} étant une sous-classe de Factory qui est une sous-classe de Facility.

Pour ce faire, MetalFactory appellera addComponent(c: Component). Après avoir ajouté le composant au stock, dans addComponent(c: Component), il y aura un appel de notifyObservers() qui appellera ensuite la fonction IObserver update(subject: ISubject, payload: Object) de chaque observateur.

Dans ce cas, GlobalState est l'un des observateurs. Suite à l'appel de sa fonction update(subject : ISubject, payload : Object), il recrée une partie de son état {facilities : Map<Facility, ArrayList<Component>>} qui est un Map d'un Facility en association de sa stock {stock : ArrayList< Component>} via XMLUtils.

2.4.2. Autre diagramme de séquence



Voir randomized-sell-strategy.png

À chaque TOUR, Environnement appellera la méthode executeSell(deltaTime: long, tour: int) du Warehouse. En fonction de l'ISellStrategy sélectionnée par l'utilisateur, Warehouse vendra soit un avion au cours de ce TOUR sur la base d'une chance prédéfinie, soit à intervalles fixes.

Dans cette situation spécifique, nous explorons ce qui se passera si une RandomizedSellStrategy est sélectionnée. Lors de l'appel Warehouse executeSell(deltaTime: long, tour: int), l'implémentation par RandomizedSellStrategy de ISellBehavior sell(planes: ArrayList<Component>, deltaTime: long, tour: int) sera appelée. La fonction de vente de RandomizedSellStrategy déterminera (la chance par défaut est 1/20) si un avion sera vendu dans ce TOUR.

3 Décisions de conception / d'implémentation

3.1 Décision 1: Utilisation de Record (Java 14) pour représenter les données lues à partir de configuration.xml

L'un des défis de la lecture de données à partir de fichiers externes est de savoir comment les modéliser au sein de l'application. Accorder peu d'attention à cette partie du code conduit souvent à des requêtes internes très complexes. Par exemple, on peut opter pour une seule classe POJO pour représenter les données dans configuration.xml. Bien que cela puisse fonctionner, cela augmente encore la difficulté à comprendre la structure de l'application et conduit souvent à une base de code très désordonnée. Cette classe POJO unique devra ensuite être transmise à tous les objets nécessitant ses données, ce qui corrobore davantage le couplage entre les classes, ou l'on peut choisir de faire de la classe POJO un singleton auquel chaque classe a accès. Mais cela introduit des problèmes de concurrence possibles et rend les tests imprévisibles.

Une alternative serait de diviser chaque groupe de données pertinente dans sa propre classe POJO, comme au Figure 1. Le problème avec ceci est la nature verbeuse de Java. Il serait tout simplement trop fastidieux d'écrire manuellement des comparateurs, des getters et des setters, etc. pour chaque POJO différent.

```java
public class ComponentPOJO {
    String iconPath;
    ComponentType type;

    Point translate;
    Point currentPos;
    Point to;

    public ComponentPOJO(String iconPath, ComponentType type, Point translate, Point currentPos, Point to) {
        // [...] constructeur
    }

    public String getIconPath() {
        return iconPath;
    }

    public void setIconPath(String iconPath) {
        this.iconPath = iconPath;
    }

    // [...] reste des getters et setters


    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        ComponentPOJO that = (ComponentPOJO) o;

        if (getIconPath() != null ? !getIconPath().equals(that.getIconPath()) : that.getIconPath() != null)
            return false;
        if (type != that.type) return false;
        if (!Objects.equals(translate, that.translate)) return false;
        if (!Objects.equals(currentPos, that.currentPos)) return false;
        return Objects.equals(to, that.to);
    }

    @Override
    public int hashCode() {
        int result = getIconPath() != null ? getIconPath().hashCode() : 0;
        result = 31 * result + (type != null ? type.hashCode() : 0);
        result = 31 * result + (translate != null ? translate.hashCode() : 0);
```

Figure 1

Une autre solution serait d'utiliser Java 14 Record. C'est similaire à avoir plusieurs POJO pour modéliser des groupes de données pertinents, mais sans sa verbosité inhérente, car la plupart des méthodes nécessaires seront déjà pré-construites. Si nous comparons la Figure 1 qui n'est qu'un POJO avec la Figure 2 qui est un Record, il y a une nette différence dans les lignes de code nécessaires pour représenter un Component.

Figure 2

J'ai décidé d'utiliser Record comme conteneur de données. Depuis le début, j'avais déjà pensé à regrouper les données pertinentes dans leurs propres classes, mais emprunter la voie POJO m'aurait pris beaucoup de temps. L'inconvénient que cela pose est la personnalisation de hascode(), equals(), toString() car je dépends de l'implémentation par défaut. Cependant, il est toujours possible de remplacer ces méthodes en cas de besoin.

3.2 Décision 2: Utilisation de l'héritage et du polymorphisme pour structurer la hiérarchie des différentes usines du réseau

J'ai donc utilisé Record pour représenter les données de base de configuration.xml. Cependant, des groupes de données plus complexes tels que les usines nécessitent un modèle beaucoup plus complexe. Supposons que nous fassions une classe pour les usines, comment cela affecte-t-il l'interaction de la classe qui l'utilise (Environnement) ? Comment alors modéliserions-nous notre classe de manière à ce que la classe cliente puisse facilement utiliser la classe modèle sans nécessairement connaître les subtilités de son implémentation ? Comment allons-nous nous assurer que cette conception est évolutive?

La solution la plus simple, bien que certainement pas la meilleure, consiste simplement à avoir une seule classe Facility qui représente toutes les usines et tous les entrepôts, comme au Figure 3. Cependant, cela crée inutilement de la complexité, soit pour la classe cliente qui l'utilise, soit au sein de la classe Facility à la place. Comment différencierions-

nous quel composant sort de craftComponent() ? Devrions-nous créer un tas d'instructions if-else telles que pour une installation de {type : String}, un certain composant est créé ? Bien que cela puisse très bien fonctionner, ce n'est pas une solution évolutive, et chaque évolution de cette fonction spécifique entraîne une dette technique encore plus importante.

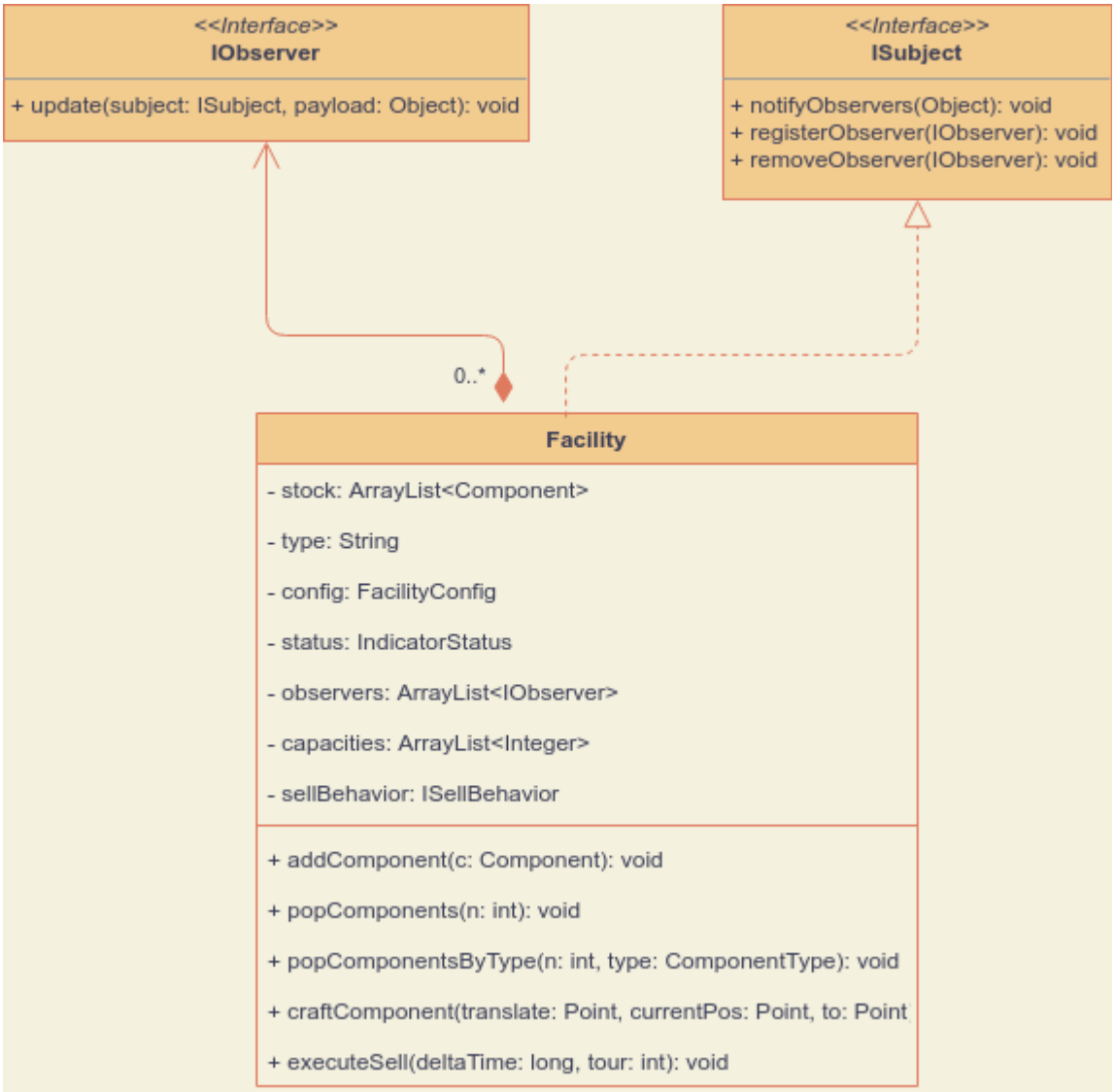


Figure 3

Une meilleure solution serait d'utiliser l'héritage et le polymorphisme pour séparer les responsabilités entre de nombreuses classes différentes comme au Figure 4. De plus, cela

devient en fait plus facile à utiliser pour la classe cliente grâce au polymorphisme. Il est également beaucoup plus évolutif, car l'ajout d'un autre type de Factory, par exemple, ne nécessite que la création d'une nouvelle classe qui étend Factory. Il n'y aurait pas besoin de bricoler une fonction craftComponent() pour étendre sa logique.

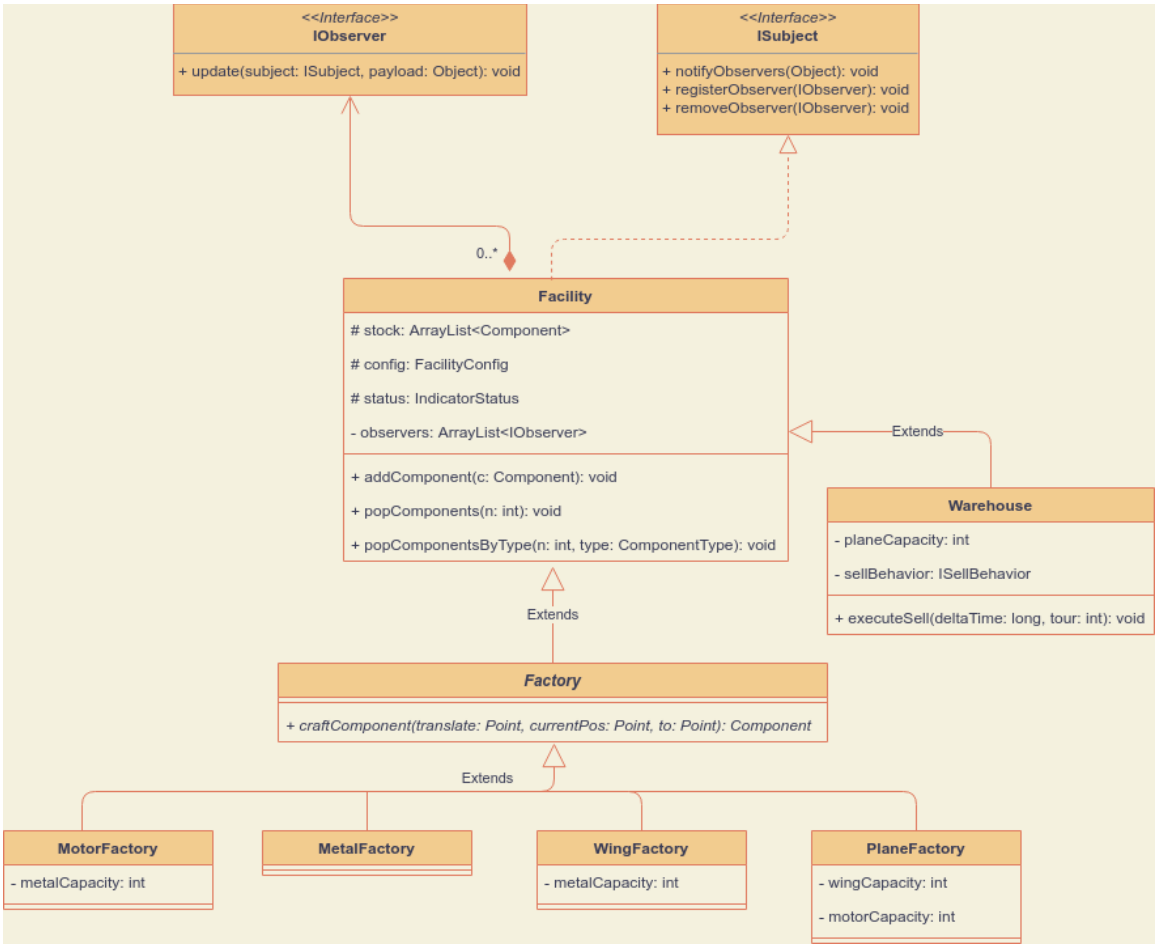


Figure 4

J'ai décidé d'utiliser la conception de la figure 4. Elle est beaucoup plus maintenable et évolutive. Il est également beaucoup plus facile à gérer pour la classe cliente car elle peut utiliser des classes parentes comme types de paramètres rendant leurs fonctions plus génériques et modulaires.

4 Conclusion

Au bout du compte, la phase de conception est une phase très importante dans le cycle de vie du développement logiciel. Elle permet de repérer les erreurs et de réduire ainsi l'accumulation de dette technique. Au cours de ce laboratoire, l'utilisation réussie de l'héritage, du polymorphisme, des modèles de conception ont tous été des outils qui m'ont aidé à créer des classes et des fonctions plus modulaires et ayant une hiérarchie claire entre elles. Bien que l'utilisation d'un GlobalState mutable ne soit pas souhaitable, c'est certainement quelque chose qui peut être amélioré plus tard, ou quelque chose que je peux corriger pour de futurs projets. Néanmoins, je pense que la facilité d'utilisation d'autres classes/objets au sein de la classe client (Environnement) est quelque chose dont je peux être fier. Passer du temps à concevoir méticuleusement le système de telle sorte que les interactions entre les classes ne soient pas complexes est un défi que j'ai apprécié de relever.

5 Références

- Fowler, Martin. 2004. «Inversion of Control Containers and the Dependency Injection pattern». <https://martinfowler.com/articles/injection.html#InterfaceInjection>.