Projet Seven Wonders

Maya Medjad, Kilian Clark, Zaid Burkutally,
Samuele Marino, Vincent Laubry

Table des matières

1 Organisation générale du code

1.1 Découpage en package

1.1.1 Le jeu

Le projet est découpé en cinq packages qui intéragissent entre eux.
Game contient la classe Game qui va exécuter les tours de jeu, l'Enum Age qui représente les 3 âges de la partie,la classe Exchange qui représente un échange entre deux joueur, la classe Launcher qui permet de lancer une ou 1000 parties, la classe Referee qui va déterminer la validité des coups des joueurs à chaque tour.

Player contient la classe PlayerWithInventory qui lie un joueur avec un inventaire, la classe PlayerAction qui permet à un bot de choisir une des actions possible dans le jeu, la classe Player qui appelle une stratégie selon la difficulté choisie, l'interface IStrategy qui définit les méthodes que devront implémenter les stratégies, la classe StrategyFactory qui renvoie une stratégie selon le niveau de difficulté choisi.

Inventory contient l'interface IInventoryReadOnly qui sera l'inventaire seulement en lecture auquel le joueur aura accès, la classe Inventory qui contient tout ce qu'un joueur a lors d'une partie, la classe PointCalculator qui est le calcul des différents points, l'Enum Ressources avec les ressources qui pourront être ajoutées à l'inventaire.
Board contient la classe Card qui définit la structure d'une carte et les intéractions sur la carte, la classe Wonder qui fait la même chose pour les merveilles, deux classes qui sont le deck de carte et le deck de merveilles, la classe Trade qui lie une valeur avec une ressource.
Services contient la classe ClientServce qui envoie les statistiques, la classe HttpClient qui fait la requête post, la classe StatHandler qui formate les statistiques pour l'envoi.

1.1.2 Le serveur

Le serveur contient une classe Application qui lance le serveur, une classe Controller qui reçoit les statistiques envoyées par le jeu et une classe StatService qui crée le fichier CSV qui a le total des points des joueurs pour chaque partie.

1.2 Interface et intéractions entre les classes

1.2.1 Intéractions

Game utilise Deck, DeckWonders, Referee, PlayerWithInventory, IInventoryReadOnly, Resources, Player, PlayerAction qui sont les classes principales qui permettent de jouer à une partie. Il a besoin d'avoir accès aux différentes ressources, au Referee qui sera utilise à chaque tour de jeu, aux inventaires, au joueur en lui même, et aux actions qu'il effectuera à chaque tour. Le referee donnera le gagnant en fin de partie.

Referee utilise PlayerAction, PlayerWithInventory, Points Calculator, Ressources, Card, Trade et IInventoryReadOnly. Pour permettre la vérification des coups, le referee a besoin des actions que peut faire le joueur, à l'inventaire pour vérifier s'il a les ressources pour acheter quelque chose, aux fonctions de calcul de points, aux ressources, aux cartes, et aux Trade pour les échanges qui seront effectués entre les joueurs.

Card et Wonder utilisent Resources, IInventoryReadOnly, PlayerWithInventory. Ces classes ont besoin des inventaires pour vérifier si le joueur peut acheter la carte ou la merveille.
Inventory utilise Card, Trade, Resources et Wonder. Il renvoie le contenu de l'inventaire ou ajoute des ressources à l'intérieur.
PointsCalculator utilise Card, Trade, Referee et PlayerWithInventory, il accède à l'inventaire d'un joueur pour calculer ses différents total de points.
Player utilise IStrategy pour accéder à la stratégie que StrategyFactory aura donné.
PlayerAction utilise Card, Exchange, et on lui passera en paramètre une liste d'échanges pour enregistrer quels échange seront fait pour l'action actuelle.

1.2.2 Détail sur les interfaces

IInventoryReadOnly utilise Card et Wonder,cette interface permet de bloquer l'accès à certaines fonctions au joueur, ainsi il ne pourra pas tricher en modifiant les informations de son inventaire.
ICard est une simplification de Card et Wonder, ce sont deux classes qui font la même choses mais qui sont traitées légèrement différemment lors du jeu. On les différencies mais elles gardent la même base.
IStrategy est l'implémentation du pattern Strategy et permet de faire plusieurs stratégies pour un seul joueur.

1.3 Répartition des responsabilités entre les classes

Les hiérarchies au niveau du code ont été faites pour découpler tous les objets afin que les principes SOLID et GRASP soient respectés et de façon que les classes soient extensibles.

Principes GRASP :
Les responsabilités ont été affectées à des classes spécialistes, par exemple la classe Referee possède les informations nécessaires pour désigner un vainqueur à la partie.
Le code est faiblement couplé puisqu'on peut faire évoluer le code et il n'y aura pas besoin de changer toutes les classes qui sont liées à ces objets changés, par exemple de nouvelles stratégies pour les joueurs peuvent être rajoutées sans changer le code.
Toutes nos classes sont fortement cohésives, elles sont faciles à comprendre, facile à maintenir et facile à réutiliser, par exemple la classe Inventory ne contient que les responsabilités qui sont liées à un Inventory, les ressources.
Le contrôleur dans notre code est la classe Launcher qui délègue les opérations et responsabilités aux classes concernées, par exemple la classe Launcher délègue la responsabilité du jeu à la classe Game.
La responsabilité des objets a un objet intermédiaire pour éviter le couplage, par exemple les Exchange sont rattachés avec la classe Referee afin d'éviter le couplage entre les Joueurs.
Polymorphisme : Les fonctions varient par rapport à des Types, par exemple la stratégie d'un joueur dépend du Type qu'il a choisi. Une stratégie type Random sera différent de celle de Easy.
Une classe B doit être responsable de la création des instances de la classe A si les instances de B sont en étroite collaboration avec A, par exemple la classe Game instancie la classe PlayerWithInventory puisqu'elle est utilisée étroitement et les données nécessaires à instancier la classe PlayerWithInventory sont dans Game.

Principes SOLID :
Nos classes ont une et une seule responsabilité puisque nous avons une forte cohésion et un faible

couplage dans notre code, la classe Trade contient toutes les responsabilités nécessaires pour effectuer un Trade (La ressource qui va être échangée et aussi sa quantité)
Nos classes sont ouvertes à l'extension mais fermées à la modification, par exemple la classe StrategyFactory est ouverte à l'extension mais fermée à la modification.
Le principe de substitution de Liskov : Les objets peuvent être remplacés par des instances de leurs sous-types sans casser le programme, par exemple IInventoryReadOnly peut être remplacé par Inventory sans causer de problème dans le code.
Nos interfaces sont spécifiques au besoin des objets, par exemple ICard est spécifique à la classe Card pour faire payer une carte par exemple.
Dans tout notre code la dépendance est faite au niveau des abstractions et pas des réalisations concrètes, par exemple la fonction launchGame() de la classe Game dépend de IInventoryReadOnly et non pas de la classe Inventory directement.

La répartition des responsabilités a été mise dans des packages pour mieux organiser le code. Par exemple le package Board contient tous les objets qui sont dépendant sur le Board pour lancer une partie. Les packages découpent le code selon la responsabilité de chaque partie et ils interagissent entre eux pour mieux gérer ce qu'ils traitent. Par exemple le Package Game qui contient les objets Exchange est responsable des échanges de Ressources encapsulées dans des objets Trade. La classe Referee quant à elle s'assure que les Choix des joueurs soient valides et c'est aussi cette classe qui renvoi le gagnant de la partie grâce aux Resource (points de victoires et points militaires). Board contient toutes les responsabilités liées au jeu, Game les responsabilités liées aux règles du jeu, Inventory les responsabilités des Ressources d'un joueur et Player les responsabilités des joueurs. Cette hiérarchie a été implémentée de manière qu'un joueur ne puisse pas accéder à son inventaire pour modifier ses ressources et ses cartes. Elle découpe aussi le Referee pour qu'il ne n'interagisse pas directement avec les joueurs mais avec leur inventaire.

2 Les patrons de conception et leur utilisation au sein du projet

2.1 Les patrons réalisés

Nous avons, à ce stade du projet, implémenté deux patrons de conception : *Le pattern Strategy.* À la suite de notre soutenance de projet, nous avons mis en place ce Pattern qui permet de rendre interchangeables des algorithmes en fonction de l'état dans lequel se trouve le contexte qui les utilise. Autrement dit, grâce à cet algorithme notre classe Player va pouvoir posséder une stratégie de jeu, et la changer en fonction de l'état dans lequel lui ainsi que le jeu qui l'entoure se trouvent.

Le pattern Factory. Ce Design Pattern permet d'encapsuler la création d'objets instanciés de classes qui héritent toutes d'une même classe mère. Plus clairement, une classe Factory va permettre d'instancier dynamiquement des sous-classes et par conséquent de mettre à profit l'utilisation du polymorphisme. Lorsque nous avons présenté le projet, nous avions une classe PlayerFactory qui nous permettait de créer des sous-objets de Player en fonction de la difficulté envoyée par l'appelant. Depuis nous avons supprimé ce PlayerFactory pour permettre l'utilisation du pattern Strategy, et nous avons créé une classe StrategyFactory qui, dans le même principe que la classe supprimée, créée une stratégie en fonction du niveau de difficulté fournie.

2.2 Les patrons annulés

Nous voulions aussi implémenter d'autres patrons de conception avant de renoncer car nous avons découvert qu'il ne répondait pas à nos besoins. Parmi ces Design Patterns se trouvent :

Le pattern Facade. Il a pour but de cacher derrière une interface simple un système complexe reliant des objets métier avec de nombreuses interactions. Ce Design Pattern nous a paru tout d'abord intéressant, mais n'ayant pas de sous-système suffisamment complexe et suffisamment indépendant du reste du système qui nécessiterait l'utilisation d'une Facade, il nous a paru finalement inutile.

Le pattern Proxy. Nous avons dans notre projet la contrainte suivante : "La classe représentant le joueur ne doit, ni être composé de son inventaire, ni pouvoir le modifier directement." afin d'éviter toute triche. Or nous avons besoin que le joueur puisse accéder au contenu de son inventaire afin de pouvoir jouer. Nous avons donc cru que le Design Pattern Proxy, qui permet de substituer un objet par un autre, répondrait à nos besoins et que nous pourrions grâce à cela fournir à l'utilisateur un proxy de son inventaire. Il s'avère que l'utilisation que nous en aurions fait ne correspond pas à la définition de ce pattern. Nous avons donc décidé, en nous inspirant du pattern proxy, de finalement créer une interface, implémentée par l'inventaire, qui ne contiendrait que les signatures des méthodes permettant d'accéder à son contenu. Ainsi nous donnerions au joueur l'accès à son inventaire, mais seulement à travers cette interface.

3 Les Métriques

Metric	Value
LOC (Lines of Code)	1890
Code Smells	78
Technical Debt	1 day and 5 hours
Technical Debt Ratio	1.5 %
Rating	A
Tests	44
Success rate of tests	100 %
NOS (Number of Statements)	615
NOM (Number of Methods)	137
Number of Classes	34
CYCLO (Cyclomatic Complexity)	251

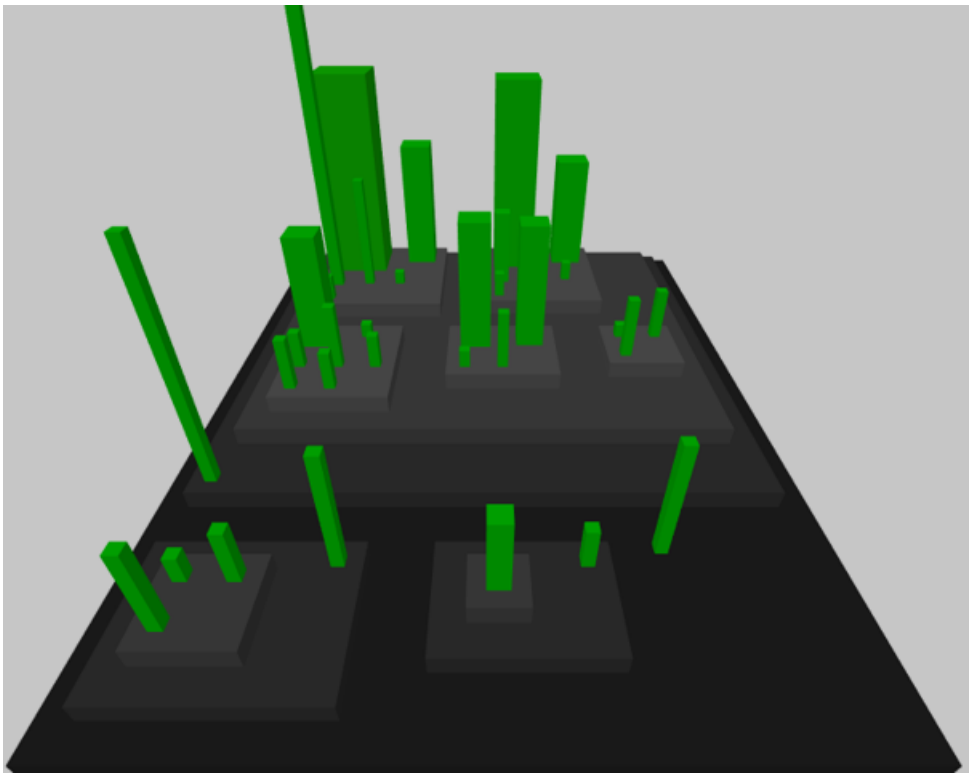

Figure 1: Le code city du projet

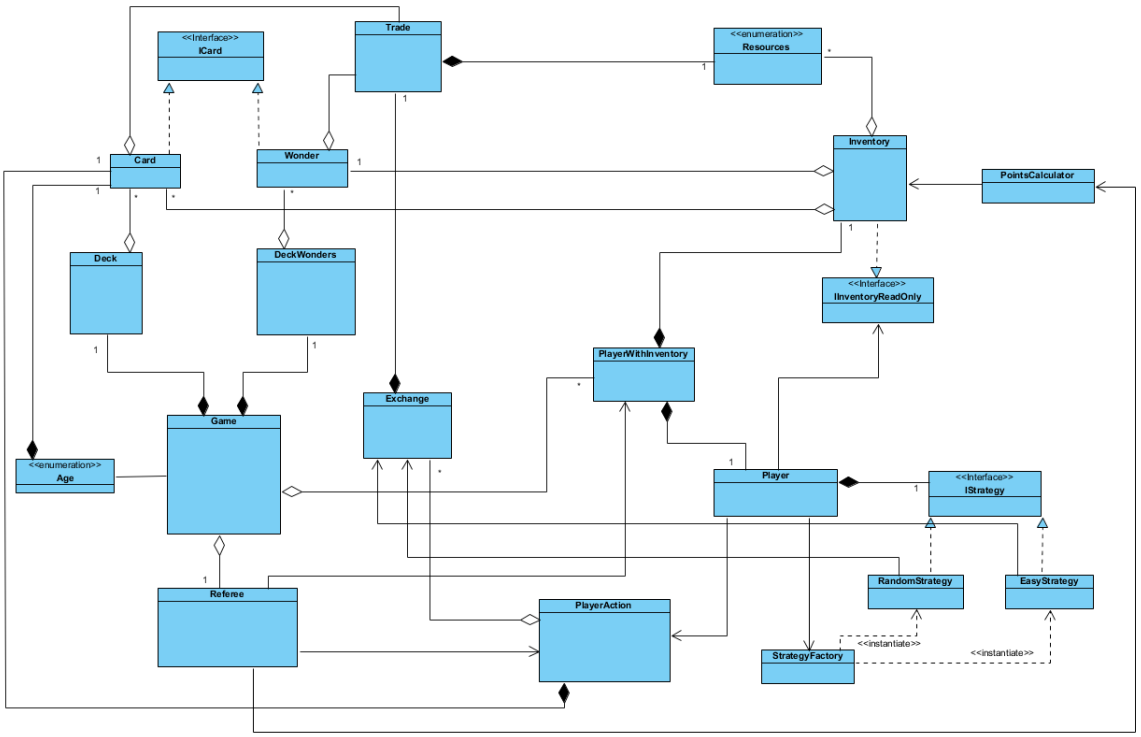

Figure 2: UML du projet