



Génie logiciel et projet de développement

Auteurs:

Juliette Sabatier
Raphaël Pietrzak
Dylann Batisse
Margaux Schmied

Professeurs:

Philippe Renevier
Philippe Collet

2021/2022

Contents

1	Introduction	1
2	Organisation générale du code	1
2.1	Découpage en modules et packages	1
2.1.1	Modules	1
2.1.2	Packages	1
2.2	Hierarchie d'héritages	2
2.3	Interfaces et interactions entre les classes	2
2.4	Répartition des responsabilités	3
2.4.1	Design GRASP SOLID	3
2.4.2	Responsabilités	3
3	Patrons de conception utilisés	4
3.1	Factory	4
3.2	Strategy	4
3.3	GameLoop	4
3.4	Façade	4
4	Patrons de conception envisagés	4
4.1	Observer	4
5	Métriques	5
5.1	Couverture	5
5.2	Duplication de code	5
5.3	Qualité du code	5
5.4	Complexité cyclomatique	6

1 Introduction

Nous avons travaillé sur la création du jeu Seasons dans le but de faire jouer des IA, le tout par le biais du langage de programmation JAVA et du gestionnaire de dépendance MAVEN.

Nous aborderons les différents éléments et concepts qui composent le projet avec une courte description pour chacun d'entre eux afin de garantir une meilleure compréhension globale.

2 Organisation générale du code

2.1 Découpage en modules et packages

2.1.1 Modules

app : Lancement du jeu avec le client/serveur lancé en simultané pour récupérer les statistiques de la partie courante.

server : Utilisation d'un serveur pour la récupération de statistiques envoyé par un client, elles pourront plus tard être utilisées pour une meilleure intégration des IA.

report : Permet la récupération des métriques de SonarQube pour l'analyse de code.

common : Module qui nous sert de dépendance commune pour pouvoir utiliser les différentes classes dans deux modules différents.

2.1.2 Packages

Afin de faciliter la recherche des classes dans les packages nous avons essayé d'appliquer un découpage efficace des différents composants du jeu.

board : Composants du plateau de jeu (gestion de la progression de la partie, plateau du jeu...).

cards : Outil de création des cartes et des effets appliquées à celles-ci.

dice : Représentation et création des jeux de dés.

exception : Différentes exceptions propres au projet pouvant être lancées lors de l'exécution du code.

game : Classes essentielles au fonctionnement du jeu notamment pour l'initialisation de la partie et de son évolution.

ia : Représentation d'un joueur.

- `inventory`: Inventaire du joueur et ses différents composants.
- `strategy.choose`: Regroupement de stratégies pour le joueur.

util : Éléments n'ayant pas d'appartenance particulière et étant utilisés dans tout le projet.

2.2 Hiérarchie d'héritages

Pour chaque tâche importante nous en avons abstrait une partie afin d'en faciliter l'implémentation. Par exemple: `AbstractCard` permet aux cartes d'avoir une base commune facilement modifiable et utilisable, simplifiant grandement leur création.

2.3 Interfaces et interactions entre les classes

Au lancement du jeu, nous avons la possibilité de lancer plusieurs parties à la suite. Une partie commence la création des joueurs structuré par l'interface `Player` à l'aide de `PlayerFactory`. Ces derniers se distinguent par des stratégies pouvant être différentes pour chaque action.

S'ensuit la création du `Board`. Ce dernier centralise les informations liées à la partie comme la gestion temporelle en charge de l'évolution des années et des saisons.

Vient ensuite la création d'un deck à l'aide de la factory chargée de produire des cartes basées sur l'interface `Card` elle-même implémentée par la classe abstraite `AbstractCard`. Ces cartes seront différenciées par leur effet sur le jeu. Elles seront toutes ajoutées au `Deck` qui sera stocké dans le `Board` et utilisé tout au long de la partie.

Chaque joueur est associé à un inventaire stocké dans le board. L'inventaire d'un joueur contient sa main qui évolue en fonction du déroulement du jeu. De plus, il contient également les bonus, le stock d'énergie et le gestionnaire des invocations du joueur.

Au cours de la partie, chaque changement de saison provoque la création d'un nouveau jeu de dés définie par la nouvelle saison et le nombre de joueurs.

Une fois la partie initialisée, les joueurs vont jouer chacun leur tour grâce à la `TurnBasedGameLoop` qui va créer une `PlayerTurnLoop` par joueur. Une `PlayerTurnLoop` définit le déroulement du tour d'un joueur, lui permettant de faire un choix entre plusieurs actions en fonction de sa stratégie. Ainsi, un joueur peut invoquer ou activer une ou plusieurs cartes, cristalliser ses énergies, utiliser des bonus, chaque action exigeant une ou plusieurs conditions spécifiques. Il a également l'option de simplement terminer son tour. Un joueur n'est pas limité à un choix d'action par tour, il peut utiliser toutes celles qui lui sont disponibles.

2.4 Répartition des responsabilités

2.4.1 Design GRASP SOLID

Nous avons essayé de suivre au mieux les principes du design SOLID et GRASP pour faire en sorte que chaque classe ait une responsabilité.

2.4.2 Responsabilités

Inventory : une classe `PlayerEnergyStock` y est stockée pour baisser sa responsabilité et donc permet à l'inventaire de ne pas se préoccuper des énergies du joueur. Finalement l'inventaire va tout de même gérer beaucoup de facettes comme:

- l'invocation des cartes
- le coût des cartes en énergies
- le choix d'une action possible pour le tour courant
- l'activation de certaines cartes

Cette classe devrait retravaillée afin de respecter le SRP (Single Responsibility Principle) et de réduire sa complexité (cf. Complexité cyclomatique).

Hand : la main du joueur ainsi que les actions basiques telles qu'enlever une carte de sa main.

Invocation : gestion et contrôle des cartes sur le plateau d'un joueur.

EnergyStock : permet de gérer un stock d'énergies.

Seasons : se préoccupe des saisons du jeu, connaît toutes les informations liées à la saison courante et permet d'utiliser les cartes s'activant lors d'un changement de saison.

Deck : charge toutes les cartes du jeu et gère la pioche ainsi que la défausse.

CardFactory : charge le json et créer les cartes s'y trouvant pour le Deck.

PlayerFactory : création de joueurs de types différents.

PlayerTurnLoop : permet au joueur d'exécuter son tour en lui laissant le choix des actions à faire en fonction de celles qui sont disponibles.

TurnBasedGameLoop : envoie les statistiques au serveur, active des cartes de fin de partie, fait choisir à chaque joueur un dé, exécute les tours des joueurs, réinitialise les cartes activées à la fin d'une manche, utilise les cartes activables à la fin d'un tour.

Board : chargé de l'initialisation des inventaires, des mains des joueurs et peut connaître le gagnant de la partie.

GameController : initialise la partie.

App : configure et lance le jeu, et initialise les joueurs.

Serveur/Client/Data : gère les statistiques du jeu.

3 Patrons de conception utilisés

3.1 Factory

Création d'un objet dérivé d'un type abstrait. Il permet la création de toutes les cartes et des différents joueurs facilement.

3.2 Strategy

Choix d'une stratégie à adopter suivant la situation du joueur dans le jeu. Pour l'instant chaque stratégie est basique et ne permet pas le changement dynamique suivant ce qu'il se passe dans le jeu. Le patron de conception Strategy est à approfondir dans le TER.

3.3 GameLoop

Fait progresser le cours du jeu en fonction des choix des joueurs. Elle est utilisée pour le tour unique d'un joueur (PlayerTurnLoop) et pour le déroulement du jeu (une ou plusieurs parties).

3.4 Façade

Cache aux IA la conception du jeu et du plateau tout en leur permettant d'accéder aux informations qui leur seront nécessaires pour interagir avec. Permet aussi de simplifier l'accès à ces dites informations, elle regroupe en un appel de fonction, une recherche fastidieuse dans le board.









4 Patrons de conception envisagés

4.1 Observer

Permet d'envoyer des signaux aux observateurs quand un changement est observé dans le jeu. Ce patron aurait pu nous être utile pour le changement de stratégies des joueurs au cours du jeu mais nous avons préféré implémenter une façade pour les IA afin qu'elles voient le jeu quand elles en ont besoin sans pouvoir le modifier.

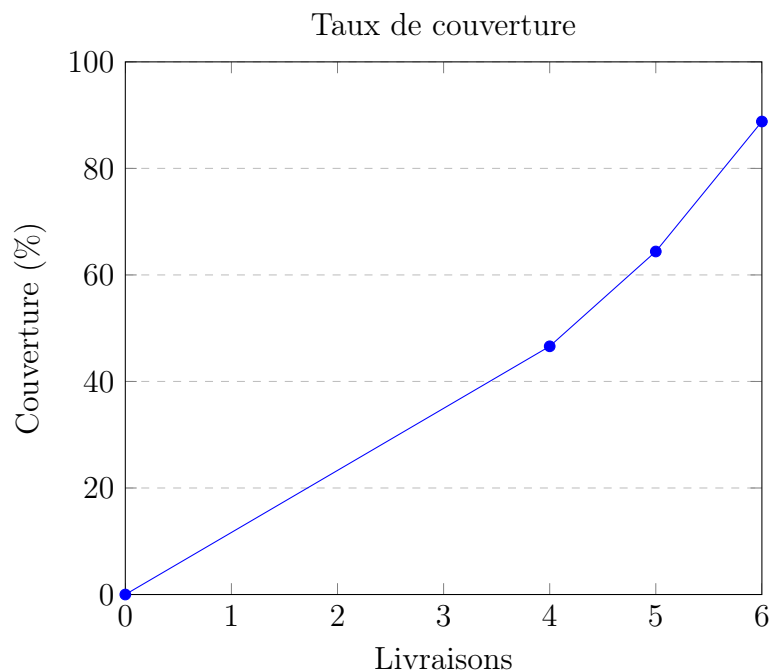
pdfTeXcmds atveryend

5 Métriques

 Vulnerabilities	 Bugs	 Code Smells	Coverage	Duplications
0 	0 	42 	88.4% 	1.4% 

5.1 Couverture

Avec 88% de couverture, nous sommes confiant quant à la fiabilité de ce projet. Les 12% non couverts font principalement partie du serveur et du client qui sont difficilement testables. Comme le graphique ci-dessous le montre, nous avons considérablement augmenté notre taux de couverture au cours des dernières livraisons.



5.2 Duplication de code

Notre code contient 1.4% de code dupliqué. Il s'agit en réalité d'un choix volontaire afin d'ajouter une fonctionnalité concernant App. Ainsi nous pouvons choisir de lancer une partie avec le serveur en simultané et d'envoyer les statistiques de la partie courante au serveur, ou de lancer le serveur séparément et d'y connecter des clients qui lui enverront leurs statistiques.

5.3 Qualité du code

De plus, SonarQube ne détecte aucune vulnérabilité ni aucun bug renforçant notre confiance dans ce projet. Enfin, nous n'avons que 42 smell code ce qui correspond à un dette négligeable de 6h07.

5.4 Complexité cyclomatique

La ville 3D ci-dessous représente la complexité cyclomatique de chacune des classes du projet. Nous pouvons remarquer un bâtiment prédominant rouge représentant la classe Inventory. Cette dernière n'a pas été totalement optimisée par manque de temps.

