Rapport de Génie logiciel

Projet de développement

Réalisé par
Loïc Filippi
Alessandro Pepegna
Meryem Boufalah
Anastasiia Kozlova
Chaeyeon Shim

Master Informatique
à Université Côte d'Azur
Premier semestre de l'année 2020 - 2021

Résumé

Dans ce rapport nous abordons les différents aspects du projet de développement et présentons nos choix de conception et d'implémentation des différents aspects du jeu. La première partie concerne les choix généraux d'organisation et les interactions entre les différentes classes. Nous réalisons ensuite une étude sur les patrons de conception. Enfin, nous effectuons une analyse sur l'évolution des métriques.

Table des matières

1 Organisation générale du code

1.1 Structure du projet

Le projet se compose de trois modules :

— java-game

— java-common

— java-server

Dans java-common, tous les éléments principaux du jeu sont présents : le joueur, l'inventaire, les âges, les cartes, les stratégies, les effets et les ressources.

Dans java-game se trouvent les éléments du moteur de jeu : le launcher, le board, le commerce, les différents *managers* et le client.

Dans java-server on trouve le serveur qui reçoit les données émises par le client, fait des calculs sur ces données et renvoie des statistiques sur les parties.

1.2 Hiérarchie d'héritages

Le jeu se déroulant en trois âges 1, il a semblé pertinent de différencier trois classes correspondant chacune à un âge du jeu.

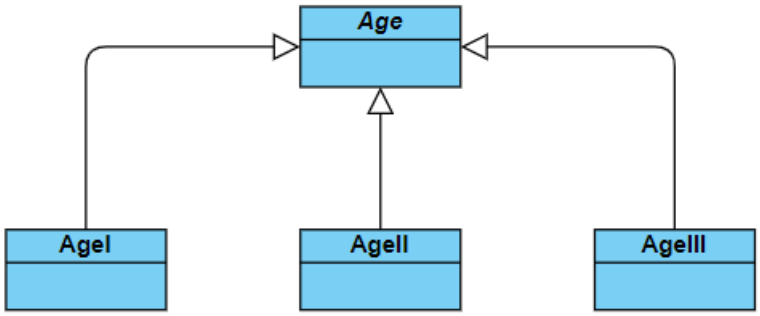

FIGURE 1 – Hiérarchie d'héritage pour la classe Age

Même chose pour les effets ici 2. Chaque effet de carte ou de plateau possède des caractéristiques bien différentes, et par conséquent sont chacun implémentés par leur classe respective.

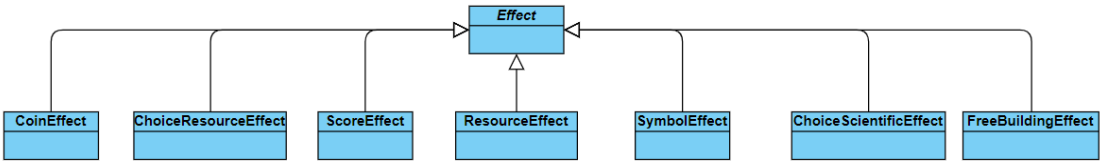

FIGURE 2 – Hiérarchie d'héritage pour la classe Effect

1.3 Structure du programme

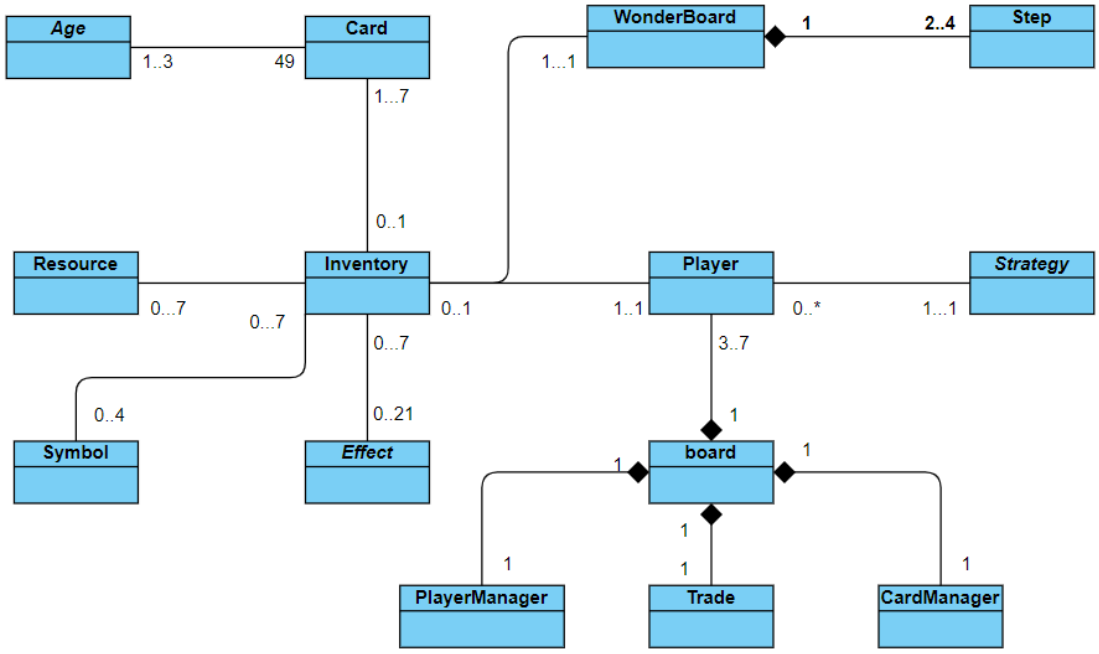

FIGURE 3 – Les principales classes constituant le jeu et son déroulement

Le programme peut se décomposer en plusieurs parties 3

- Le moteur de jeu qui s'organise autour de *Board*, avec la classe *Trade* pour gérer le commerce entre les joueurs, et deux utilitaires pour la gestion des cartes et des joueurs. Il va être chargé du déroulement de la partie, de résoudre les conflits entre les joueurs à chaque fin d'âge.

- L'inventaire, avec tous les éléments de jeu qu'il doit conserver au long de la partie pour son déroulement : les cartes des joueurs (jouées ou non), les effets que peuvent attribuer celles-ci ou les plateaux, les ressources et matières que les joueurs accumulent durant la partie, et le plateau qui est attribué au début de la partie à chaque joueur avec l'état d'avancement des étapes de la merveille.

- Le joueur et sa stratégie pour choisir la carte et l'action à effectuer avec celle-ci. Nous avons développé deux "stratégies" : la première permet de choisir la première carte et de la construire si c'est possible ou sinon de la défausser. La seconde se concentre sur la merveille, le joueur essayant autant que possible d'avancer sur la construction de ses étapes, ou de jouer une carte qu'il peut construire si construire son étape de merveille n'est pas possible.

2 Design patterns

2.1 Pattern Strategy

Dans les premières itérations du projet, les éléments et les mécaniques de jeu restaient relativement réduites, ce qui limitait le champ d'action des joueurs. Cela a conduit à peu considérer le concept de "choix d'action" des joueurs.
Lorsque le jeu a gagné en complexité au niveau des règles et des possibilités qui s'offrent au joueur, nous avons doté les joueurs d'une intelligence et d'une stratégie de jeu.

Pour cela, nous avons opté pour le pattern *Strategy* 4. Ce choix s'explique en grande partie par le fait que, bien souvent, les joueurs ne jouent pas tous de la même façon pour atteindre les objectifs de victoire, et cela nous a permis de représenter les différents comportements et stratégies de jeu que peut adopter un joueur.

Un premier avantage d'un tel choix est que cela rend facile l'attribution de stratégies aux bots et leur éventuel changement au cours de la partie.
Ensuite, cela permet d'établir des comparaisons (et une hiérarchisation) des différentes stratégies grâce aux données récoltées au fil des parties qui sont ensuite compilées pour afficher différentes statistiques concernant les joueurs.

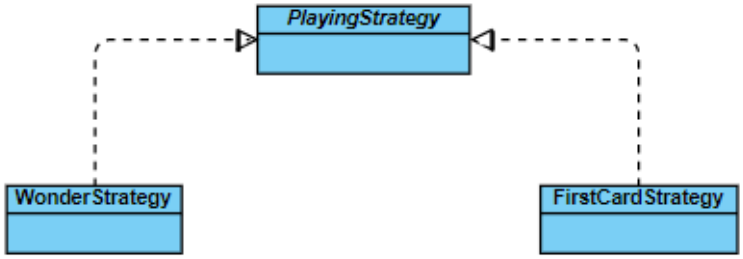

FIGURE 4 – L'interface et les deux implémentations de celle-ci

2.2 Patterns considérés

Nous traitons dans cette partie les patrons de conception possibles, qui auraient pu être appliqués dans le projet, mais qui n'ont pas été retenus.

2.2.1 Pattern Command

Nous aurions pu appliquer le pattern *Command* pour gérer les actions des joueurs, mais le faible nombre d'actions réalisables par un joueur (jouer la carte, construire une étape de merveille, défausser la carte, jouer une des cartes défaussées) et le fait qu'elles ne sont utilisées que dans la méthode qui sert à exécuter les actions d'un joueur dans la class *Board*, nous sommes arrivés à la conclusion que ce pattern n'était pas nécessaire et avons donc opté pour un *switch*.

3 Analyse par les métriques

3.1 SonarQube

Pour cette partie, nous allons nous concentrer sur les quatre derniers bilans 5.

Il est utile de noter que *SonarQube* a été branché sur notre projet juste après la cinquième release, et de fait, nous pouvons très clairement observer l'effet bénéfique de Sonar.

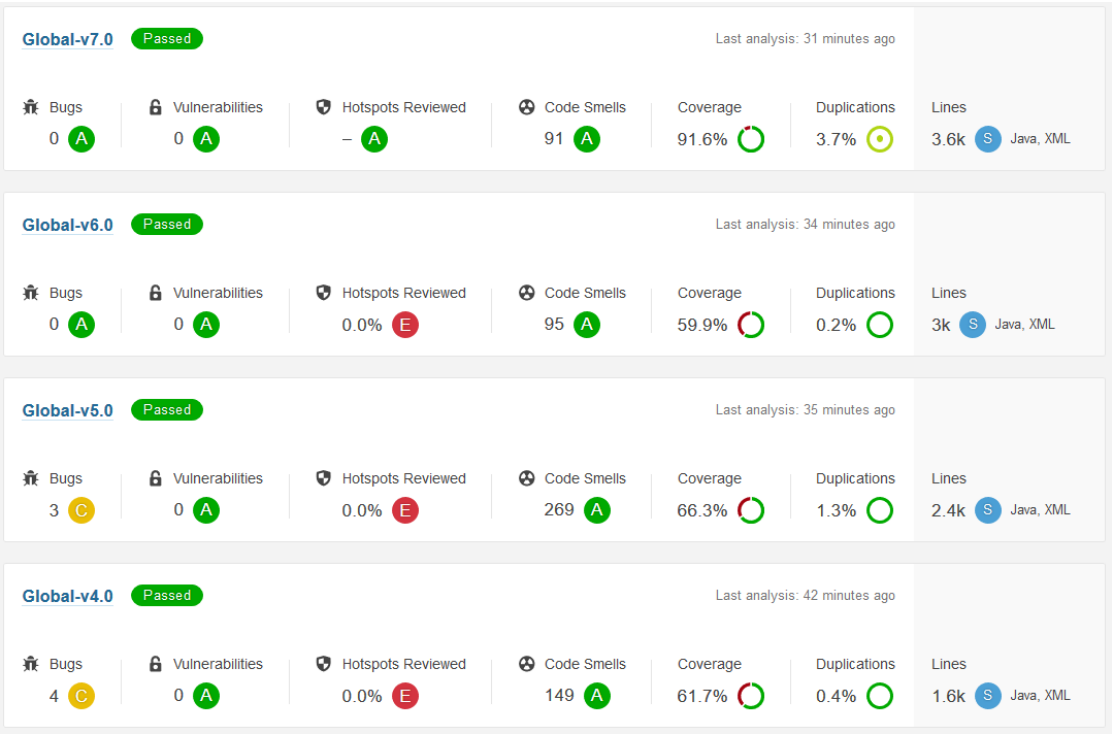

FIGURE 5 – Les bilans pour les quatre dernières *releases*

3.1.1 Code Smells et Bugs

Entre la $v5.0$ et la $v6.0$, on observe une très nette diminution des *Code Smells* et une forte diminution des sources potentielles de bugs dans notre code.
La plupart des *Code Smells* éliminés ont été introduits durant le développement de la $v4.0$ alors que *SonarQube* n'était pas encore branché. L'apport de ce dernier est facilement visible pour la version $v6.0$ où, avec l'ajout de 600 lignes au projet, le nombre de *Code Smells* a, lui, diminué.

3.1.2 Duplication de code

On observe instantanément après avoir branché *SonarQube* que la duplication de code a chuté de 1.3% à 0.2% de la $v5.0$ à la $v6.0$.
Il est cependant intéressant de toujours prendre du recul sur les métriques que nous apportent ces outils. Ici, on nous présente une duplication de code de 0.2%, et lorsque nous regardons plus

précisément, cela concernait l'initialisation d'un plateau Merveille et généraliser l'utilisation de celle-ci dans le jeu au lieu d'introduire les autres merveilles incomplètes.
En ce qui concerne le pourcentage de duplications dans la dernière itération, nous avons ajouté une fonctionnalité pour le *TER* et laissé deux méthodes identiques avec des type de retour différents. On peut voir ici un exemple de dette qui a été créée afin de ne pas affecter la fonctionnalité de base existante tout en ajoutant un élément important de l'IA.

3.1.3 Tests

Malgré le nombre grandissant de méthodes à tester, nous avons tenté de ne pas oublier certaines branches conditionnelles. À noter que, selon le principe de Pareto, les états conditionnels ne représentant en moyenne que 20% du code total sont responsables de 80% des bugs, d'où l'importance de couvrir toutes les voies d'un switch lors d'un test, par exemple. Grâce à *SonarQube*, nous pouvons facilement observer quels sont les lignes qui ont été couvertes par un test au sein d'une méthode, et donc plus précisément au sein d'un branchement conditionnel.
Une fois passée la phase d'apprentissage et de familiarisation avec cet outil, nous observons que la couverture des tests atteint les 91.6% dans la dernière version.