Rapport de Projet
Vamos

Introduction

Le but de ce projet est de modéliser une implémentation du jeu de plateaux 7 Wonders, réalisée en Java en équipe de 5, et en utilisant des outils de travail collaboratif (GitHub entre autres), de tests unitaires et automatisés (JUnit, Cucumber, Mockito), et d'analyse de code à l'aide de métriques également automatisés (avec SonarQube).

Ce projet amène plusieurs difficultés : 7 Wonders est un jeu disposant d'une multitude de règles, qui ne seront pas complexes à implémenter individuellement, mais qui rendront l'implémentation longue; et la durée assez courte pour réaliser un

projet de cette envergure (5 semaines) sera un vrai challenge pour mener ce projet à bien.

Organisation du code

Découpage des packages

Le code est découpé en deux modules nommés client et server, qui sont eux mêmes découpés en plusieurs packages.
Le client se décompose en plusieurs packages majeurs, comme core qui contient le moteur de jeu de base, ainsi que card qui contient que effects qui contient tous les effets du jeu.
Il y en ensuite wonder qui est également un gros package du client.

Hiérarchie d'héritages

Effect:
La classe abstraite Effect oblige ses 12 filles à implémenter la méthode abstraite applyEffect(…), comme ça lorsqu'on a un objet de type Effect, on est sûre de pouvoir appeler cette dernière avec les bons paramètres.

Strategy:
DumbStrategy, MilitaryStrategy, ScienceStrategy héritent de la classe abstraite Strategy, obligeant l'implémentation de chooseAction(Player player).

Interaction entre les classes pour la modélisation et son déroulement

(diagramme de classes en annexe et en plus clair dans le dossier DOCUMENTATION dans notre repository sur GITHUB , fichier UML.png)

Répartition des responsabilité des différentes classes

- Player : choix des actions, exécution des actions, gestion des points
- Card: instancier toutes les cartes des fichiers de JSON
- Wonder: instancier toutes les wonders des fichiers de JSON
- Game: initialisation des mains des joueurs/du deck, procéder les tournées des joueurs/changements des ages/changements du sens, calcul des points victoires/militaire/sciences, exécution des combats .

Les patrons

Patron utilisée(explication)

Stratégie

Le Design Pattern Strategy, implémenté par un attribut de type Strategy, classe abstraite définissant le coup à jouer en fonction des informations que possède le joueur, dont hériteront alors DumbStrategy, MilitaryStrategy, ou ScienceStrategy. Strategy propose plusieurs avantages par rapport à Template : Il serait possible de changer de stratégie en cours de jeu, le code serait plus simple à lire (et donc mieux maintenable), et l'instanciation d'un joueur ne diffèrerait pas en fonction de la stratégie qu'il devrait appliquer. Ainsi, nous choisissons d'utiliser le Design Pattern Strategy pour implémenter l'IA primitive de nos joueurs dans notre soumission du projet, dans le package player.

Patron potentielle

Template

Métrique

GitHub

Nous avons utilisé GitHub pour la gestion du projet, des User Stories, des Issues, et des releases. Cet outil nous a permis de tout travailler de manière collaborative sur le code du projet, et de récupérer les dernières mises à jour faites par les membres du groupe, ainsi que de partager nos mises à jour avec ces derniers. GitHub fournit également quelques fonctionnalités pour analyser le travail effectué par l'équipe.

Nous trouverons notamment en *annexe II* et en *annexe III* le nombre de *commits* faits en fonction du temps par les contributeurs du projet, et le nombre d'additions/deletions faits en fonction du temps. Nous pouvons ainsi voir grâce à ces graphes que l'équipe a fourni une quantité de travail plus conséquente en début qu'en fin de projet. Ceci s'explique par le fait que plus nous avancions dans le projet, plus nous souffrions du manque d'organisation initiale du code; ainsi nous passions beaucoup plus de temps à réfléchir à l'implémentation des fonctionnalités que sur l'implémentation et les tests en soi.

Pour le nombre d'additions et de délétions dramatiquement élevé en fin de projet, ceci s'explique par l'ajout sur la master de la Javadoc (décision tout à fait discutable). Ce pic d'activité ne représente donc pas une donnée exploitable.

SonarQube

Comme présenté en cours pour l'analyse du code par des métriques et la lecture de ces derniers, nous avons utilisé SonarQube, utilisé conjointement avec un GitHub Action envoyant les données au Sonar (hébergé sur le serveur personnel d'un des membres du groupe) lors de chaque commit sur la branche *master*.

En *annexe IV*, nous pouvons voir une vue de l'accueil de SonarQube, nous montrant quelques métriques, comme par exemple le *coverage*, le nombre de *code smells*, ou le nombre de failles de sécurités. Globalement les métriques sont bonnes (aucun bug, aucune faille de sécurité); le nombre de *code smells* (c'est à dire de mauvaises habitudes de programmation) est bas, et ceux présents ne sont pas critiques; mais le coverage (pourcentage du code testé) reste faible.

Ce faible pourcentage s'explique par une mauvaise habitude organisationnelle : au début du projet, beaucoup plus de temps était consacré à l'implémentation des fonctionnalités qu'à leurs tests. Nous avons choisi plus tard de passer autant de temps à tester qu'à implémenter de nouvelles fonctionnalités, mais le retard du début du projet avait alors déjà été accumulé, ce qui explique le faible *coverage* final du projet.

Conclusion

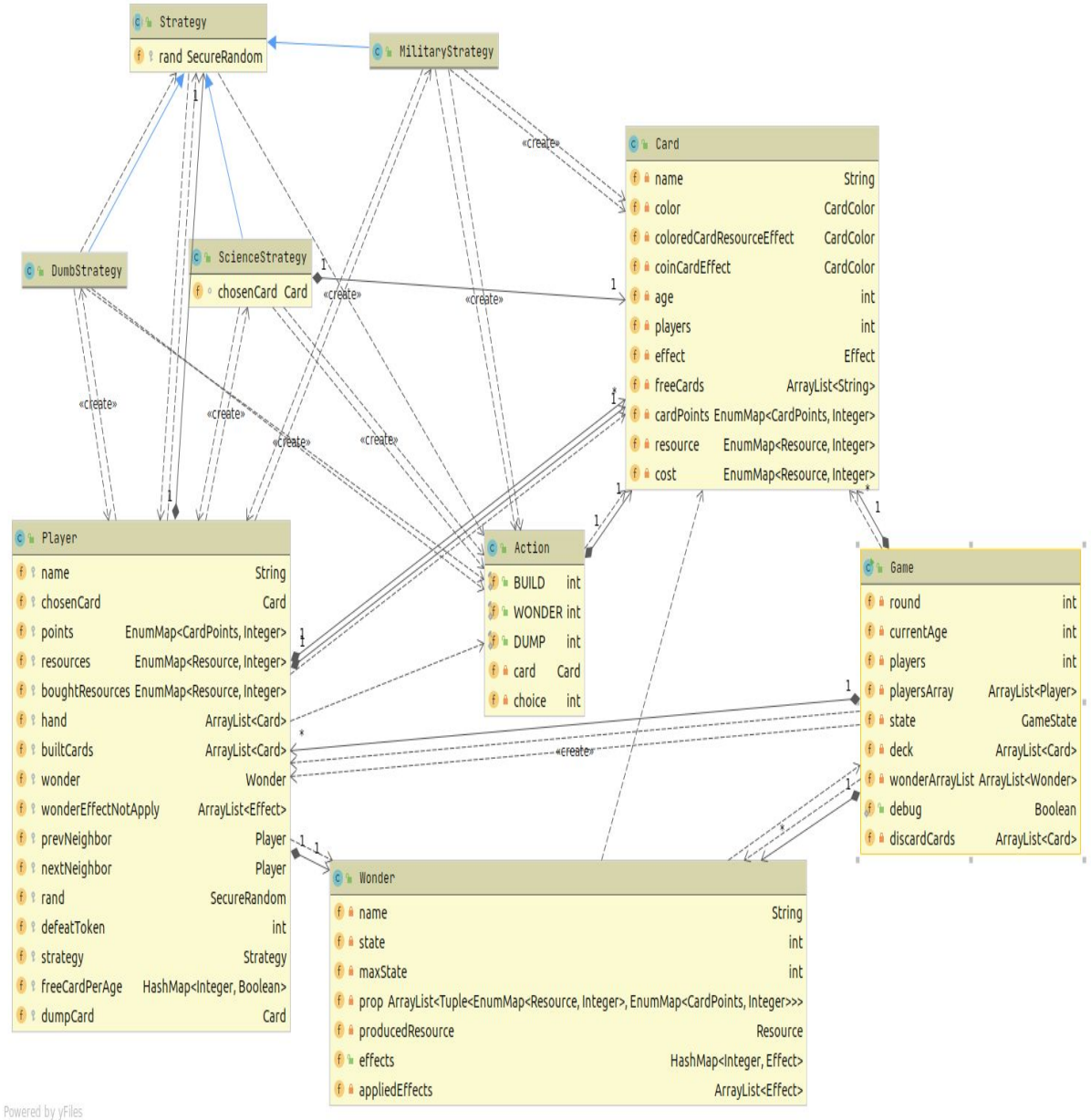
Pour conclure, ce projet nous a apporté beaucoup, en particulier sur les techniques de programmation agile en équipe. Nous aurions dû passer plus de temps au début du projet à lister les User Stories que nous avons implémenté tout au long du projet, et à les classer par ordre de priorité et de complexité, en faisant du T-Shirt Sizing par exemple.

Réaliser les tests au fur et à mesure des implémentations nous permettra à l'avenir de travailler plus efficacement et de gagner du temps. En effet, même si nous avions l'impression de gagner du temps en implémentant plus rapidement des fonctionnalités au début du projet sans faire les tests pour aller avec, le retard que nous avons accumulé à la fin du projet au niveau des tests unitaires nous a fait finalement perdre un temps précieux que nous aurions pu passer à implémenter les quelques règles manquantes à notre implémentation du 7 Wonders.

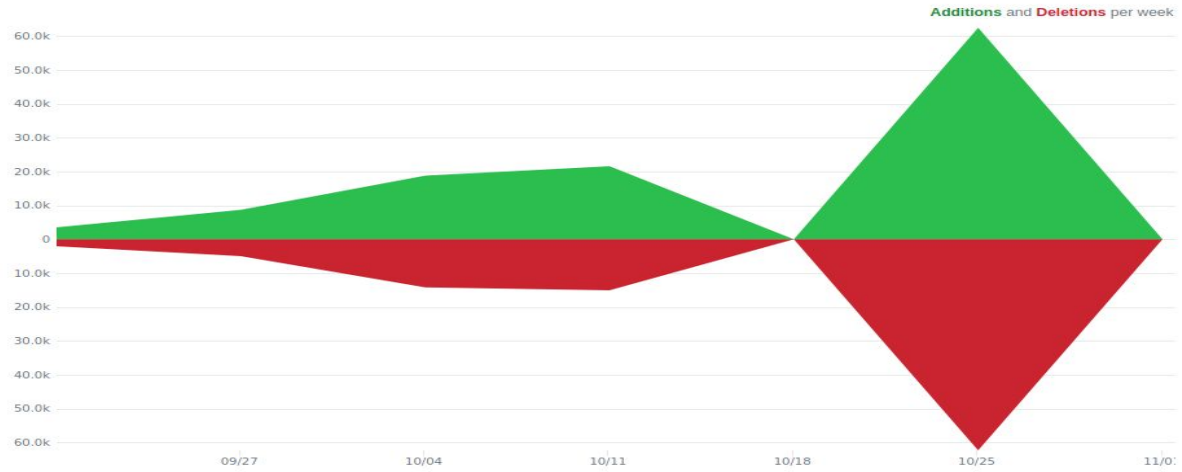
Enfin, programmer de manière agile ne veut pas dire ne pas concevoir au préalable l'architecture logicielle à l'avance, au contraire ! Cette incompréhension nous a coûté cher par la suite, notamment sur l'implémentation longue et difficile des *Effects*, et sur le manque d'utilisation de *Design Patterns* dans notre projet.

Annexes

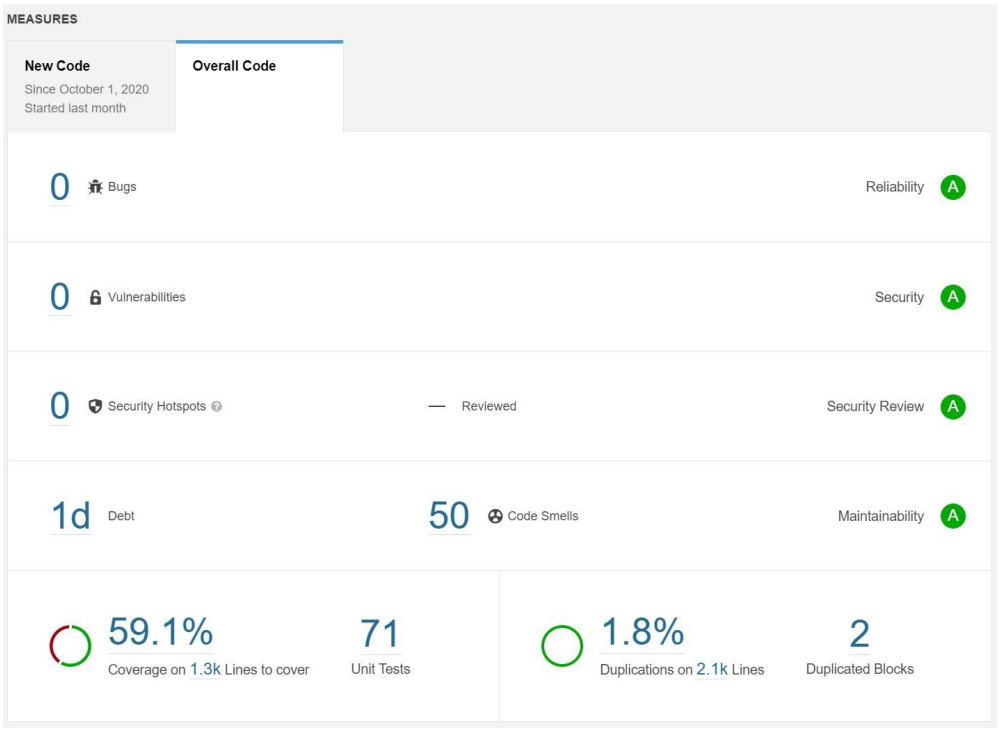
Diagramme de classe



Commits GitHub



Additions/Deletions de Lignes de Code GitHub



Accueil SonarQube