UNIVERSITÉ
CÔTE D'AZUR

Master 1 Informatique

Rapport de Projet de développement Génie logiciel

Sujet :

Réalisation d'une version informatisée du jeu « 7Wonders »

Yann MARTIN D'ESCRIENNE
Yasmine BEN FREDJ
Yohann TOGNETTI
Sébastien MARRO
Maxime JEROME

« Année universitaire 2020 - 2021 »

Table des matières

Introduction

Dans le cadre de notre premier semestre en Master 1 Informatique et Interactions à l'UNS, il nous est proposé un projet d'une durée d'1 mois nous permettant de mettre en pratique nos connaissances de la conception logicielle et de la gestion de projet sur le jeu de société 7Wonders.
Dans ce rapport nous allons vous proposer une étude globale sur notre projet sur les choix de conceptions ainsi que sur son évolution tout au long de son développement.

Organisation générale du code

I Découpage des packages

Pour structurer l'ensemble des classes et interfaces de notre projet, nous l'avons décomposé en quatre principaux modules, qui contiennent chacun les packages. Ces derniers permettent d'organiser l'ensemble des classes, interfaces et énumérations du projet.
1. Le module «client» : représente les différentes intelligences artificielles du jeu.
2. Le module «commun» : représente les données accessibles entre tous les modules.
3. Le module «gameserver» : représente le jeu et son déroulement.
4. Le module «statserver» : représente le serveur de gestion des statistiques.

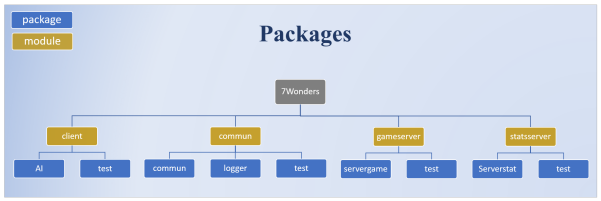

FIGURE 1 – Organisation générale des Packages

II Hiérarchie d'héritages

A. Classe abstraite :

Dans le module « client » :

- **« AI » :** une classe abstraite qui implémente l'interface «RequestToPlayer» et qui est héritée par les classes «FirstAI», «SecondAI» et «RandomAI». Elle représente un modèle d'intelligence artificielle (IA).

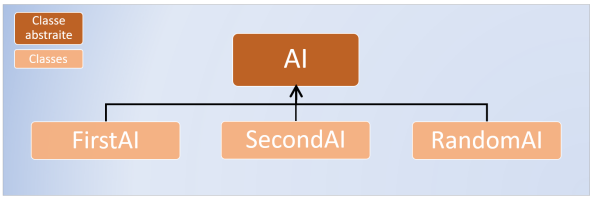

FIGURE 2 – Héritages : AI

Dans le module « commun » :

- **« AbstractAction » :** une classe abstraite héritée par les classes «BuildAction», «BuildStepAction», «DiscardAction» et «TradeAction». Elle représente un modèle de choix d'action que va effectuer une IA à chaque tour.

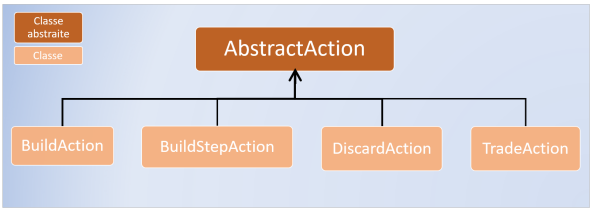

FIGURE 3 – Héritages : Action

- **« StatBase » :** une classe abstraite héritée par la classe «StatIntegerBase» qui elle-même sera héritée par toutes les classes de statistiques à base d'entiers. Elle représente un modèle de statistiques.

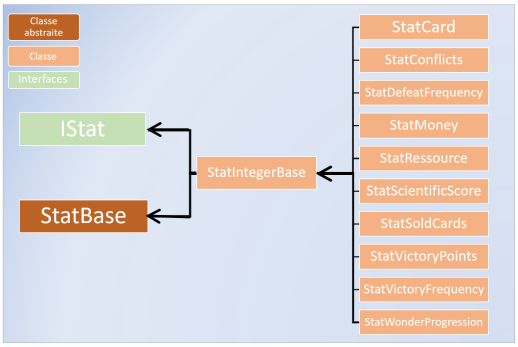

FIGURE 4 – Héritages : Statistiques

B. Interfaces :

Dans le module « commun » :

- **« ICost » :** une interface implémentée par toutes les classes «Cost», elle exige un modèle

de coût (coût des cartes ou des étapes des merveilles) que celles-ci doivent implémenter.

- **« IEffect » :** une interface implémentée par toutes les classes «Effect», elle exige un modèle d'effet ou gains (gains des différentes cartes et étapes des merveilles ainsi que les effets des guildes) que celles-ci doivent implémenter.
- **« PlayerRequestGame » :** une interface implémentée par la classe «PlayerController» dans le module « commun » et utilisée par la classe abstraite «AI» du module client pour permettre aux intelligences artificielles d'accéder à toutes les informations liées au joueur et à ses voisins.
- **« RequestToPlayer » :** une interface implémenter par la classe abstraite «AI» du module « client », elle permet donc d'exiger un modèle de tous les choix que devra faire les intelligences artificielles.
- **« IStat » :** une interface implémenter par la classe «StatIntegerBase», elle représente l'addition des statistiques.

Dans le module « gameserver » :

- **« PlayerManager » :** une interface implémentée par la classe « PlayerManagerImpl », elle exige un modèle de direction de l'objet «player» durant la partie. Par exemple :
  - *Au début de la partie :* assigner le plateau et les voisins au joueur.
  - *Au début d'un tour :* assigner une main au joueur.
  - *Durant le tour :* demander à l'IA de faire un choix puis l'exécute.
- **« PlayerView » :** une interface implémentée par la classe «PlayerManagerImpl», elle permet au joueur d'avoir une vue globale du jeu, donc de voir ses voisins ainsi que l'ensemble des joueurs.

Dans le module « statserver » :

- **« IDealer » :** une interface implémentée par la classe abstraite «DealerBase», ainsi elle exige que toutes les classes«Dealer» possèdent une méthode pour traiter les données et permettre de calculer les statistiques.

III Répartition des responsabilités et interaction entre les classes

A. Le Module client :

Dans ce module, nous pouvons trouver trois intelligences artificielles (IAs) différentes. Ces trois IAs utilisent toutes une classe abstraite « AI », qui représente les diverses fonctionnalités d'une intelligence artificielle. Parmi les trois IA, chacune d'elle possède une stratégie unique.

- **« RandomAI » :** Fait des choix aléatoires.
- **« FirstAI » :** Fait des choix par priorité (La priorité 1 (P1) étant la plus haute et la priorité 4 (P4) la plus basse) :
  - P1 : Points de victoire
  - P2 : Matières premières
  - P3 : Puissance militaires
  - P4 : Défausse de cartes
- **« SecondAI » :** Fait des choix par priorité mais à chaque changement d'Âge, les priorités changent.
  - **Âge 1 :**
    - P1 : Matières premières,
    - P2 : Bâtiments commerciaux
    - P3 : Produits manufacturés.
  - **Âge 2 :**
    - P1 : Puissance militaire
    - P2 : Construire une étape de merveille
    - P3 : Points de victoire
  - **Âge 3 :**
    - P1 : Points de victoire
    - P2 : Guilde
    - P3 : Bâtiments scientifiques

B. Le Module commun :

Dans ce module nous pouvons retrouver :

- Le package « action » contient les classes qui correspondent aux actions qu'un joueur peut faire.
  - Les actions «BuildAction», «BuildStepAction», «DiscardAction» et «TradeAction», permettent respectivement de gérer la construction des bâtiments, construire une étape de merveille, défausser une carte et faire des échanges avec les autres joueurs. Elles sont toutes basées sur la classe abstraite «AbstractAction».

- Le package « card » contient la représentation d'une carte (classe «Card») et d'un deck (classe «Deck»), qui est une liste de cartes. Chaque carte possède un type, représenté par l'enum «CardType».

- Le package « communication » envoie les statistiques du jeu au serveur. Il contient les classes «CommunicationMessages» (représente la communication entre le jeu et le serveur), «JsonUtils» (sert à désérialiser et sérialiser les paquets reçus par le serveur, «StatsObjects» (ensemble des statistiques envoyées au serveur). Il contient aussi un sous-package «statsobjects» permettant de calculer les différentes statistiques dans le jeu (exemple : la classe «StatConflicts» permet de faire des statistiques sur les conflits militaires).

- Le package « cost » représente le prix des cartes en pièces (classe «CoinCost») ou bien en matériaux (classe «MaterialCost»). La classe «MaterialCost»

utilise deux autres classes : «MaterialCostSolver» qui permet d'obtenir les combinaisons possibles et valides d'acheter chez le joueur voisin et «MaterialCostArray» qui représente le prix des matériaux.

- Le package « effect » représente tous les effets que l'on peut retrouver sur les cartes du jeu.

- Le package « material » représente les différents matériaux que l'on retrouve dans le jeu. Un matériau (classe «Material») a un nombre de ressources et un type (classe «MaterialType»). Ce package contient aussi la classe «ChoiceMaterial» qui permet de choisir entre plusieurs materiau.

- Le package « Player » correspond à la représentation d'un joueur par la classe Player.

- Le package « request » gère les requêtes données par les IA. Il permet de vérifier si l'action choisie par le joueur est valide grâce à la classe «RequestPlayerActionCheck».

- Le package « wonderboard » gère les points de conflits militaires par la classe «BattlePoint». Il contient aussi la représentation d'une merveille avec la classe «WonderBoard» ainsi que les étapes de merveilles qui correspondent à la classe «WonderStep».

C. Le Module « gameserver » :

Dans le module « gameserver » on retrouve :

- Le package «card» contient les classes «CardFactory» et «CardManager». Elles permettent respectivement de : générer les decks de cartes suivant l'Âge et gérer les decks dans le jeu.

- Le package «clientstats» contient la classe «SocketManager» qui permet de s'occuper de la gestion des paquets reçus par le serveur et renvoie les choix du joueur (IA) au serveur.

- Le package «engine» qui permet de gérer le déroulement d'une partie par le biais de la classe «GameEngine» et gérer l'effet Guilde des Scientifiques grâce à la classe «ScientistsGuildAction».

- Le package «player» contient les interfaces «PlayerManager» et «PlayerView» et la classe «PlayerController» qui permet de contrôler les actions du joueur ainsi que la classe «PlayerManagerImpl» qui assigne les voisins, le plateau de jeu et crée la vue des joueurs .

- Le package «score» contient les classes «BattleScore» et «ScoreCalculator». Elles permettent respectivement le calcul de score des conflits militaires et le score total des joueurs.

- La classe «WonderBoardFactory» qui permet la création des plateaux des merveilles du jeu avec la classe.

- La classe «GameInitializer» qui permet d'initialiser une partie.

- La classe « App» qui correspond au moteur de jeu. C'est ici qu'on lance les parties.

D. Le Module « statsserver » :

Ce module correspond au serveur de statistiques. Dans ce module, on peut retrouver :

- La classe « FileManager » qui permet la gestion d'un fichier pour les statistiques.

- Le package « listerners » : contient les classes « StatsListener » et « FinishListener » qui permettent respectivement d'additionner les statistiques et mettre fin aux additions de statistiques.

- Le package « stats » qui contient la classe « StatsObjectOrchestrer ». Cette classe permet d'afficher les statistiques sur la console et d'enregistrer celles-ci sous format CSV.
Cette classe utilise les classes contenues dans le package «dealers». Ces classes permettent de traiter les données et les renvoyer sous forme de listes de chaînes de caractères.

- La classe « Server » qui est une représentation du serveur.

- La classe « App » qui permet de lancer le serveur.

Étude des patrons de conception

Les patrons de conception sont des solutions génériques pour des problèmes de conceptions logiciel. L'utilisation de ces derniers ont permis à travers notre projet de diminuer le couplage et de renforcer la cohésion entre classes/modules.

I Patrons utilisés et justifications

A. Patron Singleton

Le patron Singleton permet l'utilisation d'une seule instance d'un objet, ce qui évite sa réinstanciation. Cela a été utilisé avec les classes «GameLogger», qui permet d'avoir les informations en temps réel, et «StatModule», pour «StatModule» cela révèle l'utilisation d'un unique objet représentant les statistiques pour éviter son dispersement lourd dans des fonctions à besoin statistiques.

B. Patron Factory

Le patron Factory nous a permis la génération des cartes en fonction du nombre de joueurs et des plateaux de merveille avec face aléatoire.

C. Patron Template

Le patron Template a été utilisé sur la gestion des données statistiques, en effet même si la majorité des statistiques sont représentées des entiers, une crainte a été posée sur le possible changement de représentation de certaines statistiques (progression de la merveille, ...) et même l'ajout de statistiques avec une différente représentation. Donc l'ajout de statistiques et la conversion de celles-ci sont des patrons Template.

D. Patron Strategy

Le patron Strategy a été utilisé notamment sur les intelligences artificielles (IAs), en effet cela nous permet de gérer plus aisément leurs stratégies et leur interaction avec le jeu. Ce patron a aussi été utilisé sur les coûts et sur les effets. On peut manipuler les coûts et les effets en passant par une même interface, ce qui permet la manipulation de ces fonctionnalités plus simple.

E. Patron Proxy

Le patron Proxy a été utilisé pour gérer les requêtes données par les IAs, notre proxy a fait office de vérifieur de données afin que les IAs ne puissent pas envoyer de données faussées ou tenter de tricher.

F. Patron Command

Ce patron définit la relation IA - jeu, l'IA passe par un patron Proxy pour effectuer une Action (il y a plusieurs actions possibles), et selon l'action et ses paramètres, nous modifions l'instance du jeu.

II Patrons non retenus et justifications

A. Patrons Observer et State

Notre jeu fonctionne de façon séquentielle, il donne l'illusion au joueur (IA) qu'il se déroule de façon asynchrone, par conséquent il n'y a pas besoin de vérifier si un état change, puisque de façon séquentiel nous traitons les informations une à une, donc nous savons ce qu'il se passe étape par étape.

B. Patron Façade

Nous pensons que notre code est assez bien découpé pour ne pas avoir à utiliser ce patron de conception, nous n'avons pas de sous-système assez complexe qui requiert ce patron.

Analyse de l'évolution métrique

Dans cette partie nous allons voir itération par itération l'évolution des métriques de SonarQ et expliquer les éventuels changements entre celles-ci.

I Itération 1

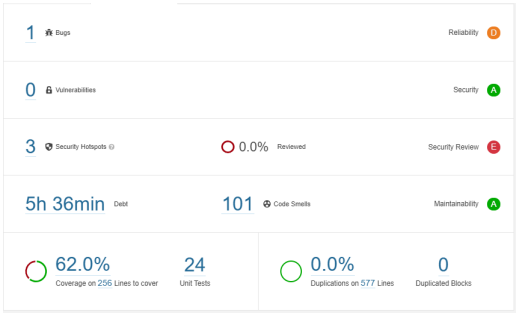

FIGURE 5 – SonarQ : Itération 1

L'itération 1 démarre avec deux classes non testées : «GameEngine» et «PlayerController» ce qui explique un coverage moyen. Les Security Hotspots sont dûs au new Random() de même pour le bug préconisant d'utiliser le même random et non d'en recréer un. La dette technique est majoritairement augmentée par une classe inutile (commun.java) et des méthodes de test en public.

II Itération 2

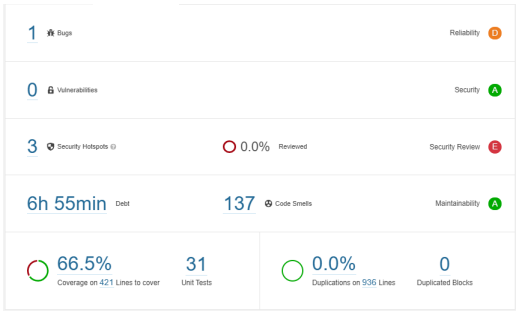

FIGURE 6 – SonarQ : Itération 2

L'itération 2 n'a aucune évolution majeure des métriques, les tests de «GameEngine» reste moyen (50% testé).

III Itération 3

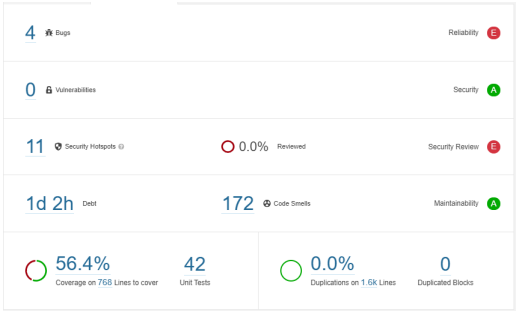

FIGURE 7 – SonarQ : Itération 3

L'itération 3 subit une baisse de toutes les métriques de SonarQ, les Security Hotspots sont dûs à des printStackTrace hors du logger. Les bugs quant à eux, à l'oublie de close() lors de la lecture de fichier. La dette augmente fortement à cause du renvoie d'ArrayList par certaines fonctions et non l'interface «List». Également à cause des méthodes par défaut dans les interfaces.

IV Itération 4

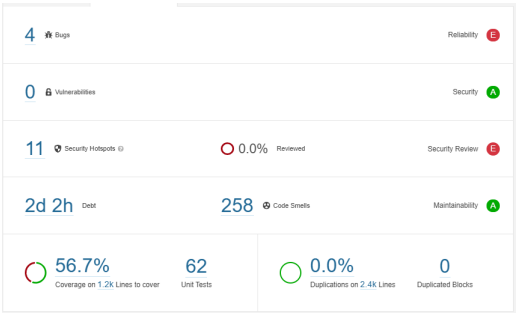

FIGURE 8 – SonarQ : Itération 4

L'itération 4 double presque la dette précédente. Cela se produit pour les mêmes raisons que les itérations précédentes, mais le code augmentant, le nombre de code smell augmente également.

V Itération 5

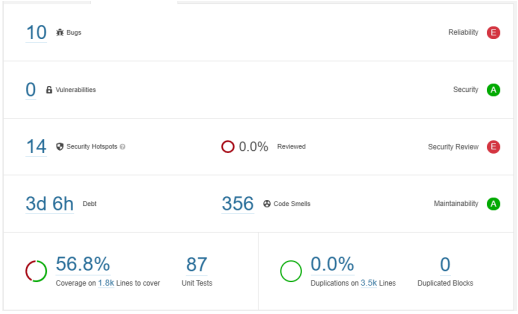

FIGURE 9 – SonarQ : Itération 5

L'itération 5 voit son nombre de bugs augmenté à cause d'objets potentiellements null (surtout dans la partie des statistiques). Les trois Security Hotspots ajoutés sont dûs au même raison ceux dit plus haut concernant les random(). La dette augmente d'un jour surtout à cause d'une fusion des classes entre «playerController» et les différentes actions (3h de dette à lui seul) et de nombreux switch case dans le code.

VI Itération 6

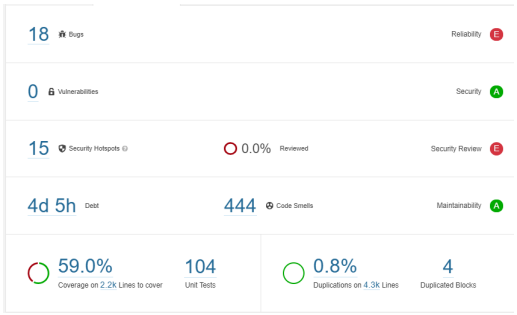

FIGURE 10 – SonarQ : Itération 6

L'itération 6 augmente de 8 le nombre de nouveaux bugs, un d'entre eux vient d'une boucle for qui ne s'arrête jamais (i = 4, i > 3, i++), et le reste de assertNotEquals entre deux types d'objet différent. La dette augmente encore d'une journée, «CardFactory» l'augmentant de 4h à cause de nombreuses string écrites en dur dans le code au lieu d'utiliser des constantes.

VII Itération 7

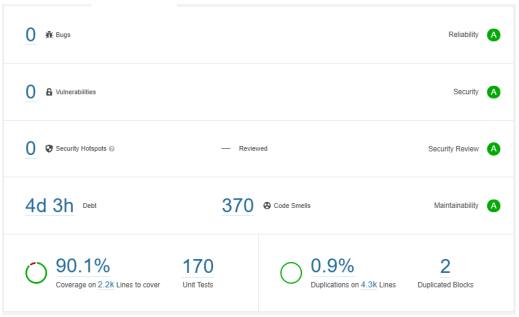

FIGURE 11 – SonarQ : Itération 7

Sur la 7ème et dernière itération, un effort a été effectué afin de corriger tous les bugs et problèmes de sécurité cités plus haut. De même, le coverage s'est nettement amélioré grâce aux tests de toutes les classes non triviales. La limite est atteinte à 90% car de nombreuse classes composées de getters et setter ne sont alors pas testées.

Tout au long des itération un coverage de plus de 50% à essayer d'être maintenu, ainsi qu'une bonne appréciation pour la maintenabilité et duplication du code.

Conclusion

Pour conclure, ce projet nous a permis de nous remettre à niveau avec tous nos nouveaux camarades. Malgré quelques légers retards notre projet est fini dans le temps imparti, il a fallu faire preuve d'ingéniosité et de créativité pour respecter les délais ainsi que d'implémenter correctement nos fonctionnalités tout en respectant nos contraintes technologiques.
En perspective, nous aurions voulu améliorer notre hygiène/nos pratiques de programmation en ce qui concerne le couvrage de nos tests mais aussi de la pertinence de nos *commits*, ainsi nous aurions voulu séparer complètement les joueurs du jeu, et de sauvegarder dans une base de données nos statistiques.