



Polytech Nice Sophia

# RAPPORT

## PS5 - Takenoko

réalisé par  
la StonksDev Team



composée de  
Dan NAKACHE, Léo POURCELOT, Thomas DI GRANDE  
& Vincent TUREL

Année universitaire 2020/2021

# Sommaire

<b>1</b>	<b>Synthèse du projet</b>	<b>1</b>
<b>2</b>	<b>Qualité du code</b>	<b>1</b>
2.1	Code de bonne qualité . . . . .	1
2.2	Code de moins bonne qualité . . . . .	2
<b>3</b>	<b>Qualité du projet</b>	<b>2</b>
3.1	Points positifs du projet . . . . .	2
3.2	Points négatifs du projet . . . . .	3
<b>4</b>	<b>Rétrospective</b>	<b>3</b>
	<b>Références bibliographiques</b>	<b>4</b>



## 1 Synthèse du projet

Les fonctionnalités suivantes ont été implémentées, conformément aux règles du jeu :

- La règle pour joueurs avertis a été ajoutée :  
Si un joueur pioche un objectif déjà réalisé sur le plateau de jeu, il doit défausser la carte et en piocher une nouvelle de la catégorie de son choix...
- Plateau de jeu totalement terminé (parcelles, pions panda et jardinier, irrigations),
- 5 actions disponibles (poser parcelle, bouger panda, bouger jardinier, piocher irrigation, piocher objectif),
- 2 actions non principales (poser une irrigation, poser un aménagement),
- 6 faces du dé météo possibles (Soleil, Pluie, Vent, Orage, Nuages, et ?),
- 3 types d'aménagements intégrés directement sur les tuiles et rajoutés par les joueurs (Enclos, Engrais, Bassin),
- 2 pions qui se comportent comme dans le vrai jeu (panda et jardinier),
- 1 IA aléatoire (RandomPlayer),
- 1 IA rudimentaire (DumbPlayer),
- 4 IA « intelligentes » :
  - EquivalentObjectivePlayer (a pour but de valider les 3 types d'objectifs équitablement),
  - RushPandaPlayer (a pour but de valider les objectifs panda en priorité),
  - RushGardenerPlayer (a pour but de valider les objectifs jardinier en priorité),
  - RushPatternPlayer (a pour but de valider les objectifs parcelles en priorité),
- 2 à 4 joueurs possibles,
- 3 types d'objectifs : parcelles, panda, jardinier,
- Condition d'arrêt (arrêt au bout de 300 actions effectuées),
- Condition d'égalité (si les joueurs ont le même score et même nombre d'objectifs panda accomplis),
- Mode histoire (affichage pas à pas de la partie),
- Mode compte-rendu avec barre de chargement et tableau récapitulatif des statistiques sur un ensemble de parties.

## 2 Qualité du code

### 2.1 Code de bonne qualité

Les bots « communiquent » avec le `Game` grâce à un système de question-réponse. Cela permet d'éviter qu'un bot malicieux puisse modifier la `Map` ou d'autres objets. Lorsqu'une `Map` est passée à un bot, c'est toujours un clone et pas la `Map` initiale. `Game` est chargé de modifier



la `Map`.

Pour les enums, on a au départ implémenté des méthodes comme `fromInt` et `toInt`, avant de découvrir `ordinal` et `values`. Cela nous a permis de supprimer beaucoup de dette technique et de simplifier le fonctionnement de beaucoup de mécanismes.

Dans certaines situations, on a besoin de connaître certaines variantes d'une enum, mais pas toutes (comme pour la météo). Pour résoudre ce problème, on a ajouté des attributs statiques, contenant uniquement les variantes valides pour un cas précis (cela peut se faire car on a un très petit nombre de cas).

Pour les différents joueurs, le polymorphisme nous a permis de développer de nouveaux bots sans jamais interférer avec les autres parties du projet.

## 2.2 Code de moins bonne qualité

L'approche pour l'API des patterns est beaucoup trop générale. Elle aurait pu être simplifiée tout en répondant aux consignes. De la même façon, l'implémentation initiale de la `Map` stockait l'ensemble des tuiles dans un tableau de `Optional<Tile>`, ce qui permet d'accéder à l'ensemble des tuiles en un temps constant ( $O(1)$ ). Cela nous forçait à stocker environ 12000 slots (pour les tuiles et pour les irrigations) en très grande partie inutilisés. Cela ralentissait énormément la copie de la `Map`.

Une réimplémentation en passant par une `HashMap<Coordinate, Tile>` et des `ArrayList<Irrigation>` nous a permis de multiplier la vitesse d'exécution par neuf dans des tests grandeur nature. Notre implémentation est toujours plus lente que celle des autres équipes.

Il y a aussi un couplage entre les classes `Map` et `Gardener`. En effet, `Map` contient un attribut de type `Gardener`, et `Gardener::moveToAndAct` prend en argument une `Map`. On aurait pu refactor cela, mais par manque de temps et de consensus, nous ne l'avons pas fait.

## 3 Qualité du projet

### 3.1 Points positifs du projet

Une des plus grandes réussites de ce projet est l'organisation que nous avons mise en place.

La première différence par rapport au projet précédent est d'utiliser des pull request pour proposer nos changements, plutôt que de toujours commit sur `master`. Cela nous a permis de nous coordonner davantage et d'éviter de nombreux conflits au moment de la fusion.

Nous avons également mis en place une validation systématique des pull request par l'ensemble des membres du projet. On a d'abord utilisé l'outil « réagir avec l'emoji  » sur Github,



puis l'outil de review intégré. Cela nous a permis de beaucoup réduire la dette technique avant qu'elle ne soit fusionnée dans la branche principale.

Nous avons aussi utilisé la continuous integration pour compiler et tester notre code à chaque modification. Cela nous a permis d'avoir davantage confiance en notre code et de ne pas avoir à exécuter les tests sur notre machine à chaque review.

Nous avons fait en sorte de répartir les responsabilités de façon à ce que le joueur prenne des décisions mais sans modifier directement l'état du jeu, plutôt que de lui laisser les pleins pouvoirs sur ce dernier. Cela nous permet de garantir que les joueurs ne peuvent pas tricher au cours de leur tour.

Grâce à un système d'exceptions assez complet, nous avons réussi à déterminer un maximum de cas limites et donc gérer ces différents cas en conséquence.

### 3.2 Points négatifs du projet

Nous avons eu beaucoup de mal à découper nos premières milestones. Elles étaient très conséquentes et couvraient beaucoup trop de fonctionnalités. Cela s'est traduit par une irrégularité dans leur vitesse de complétion.

Nous avons également eu des difficultés à tester l'intégralité de notre projet. Tester les briques élémentaires du projet comme le package `map` était simple, mais la difficulté s'accroissait à mesure que nous testions des objets plus complexes. L'utilisation de mocks nous a permis de contourner partiellement ces difficultés.

Pour ce qui est de l'organisation, nous aurions pu améliorer la distribution des tâches. En effet, chacun d'entre nous s'est cantonné à son propre domaine tout au long du développement. La conséquence d'une telle spécialisation est que certains travaillaient toujours au début de la semaine et d'autres à la fin.

## 4 Rétrospective

Ce projet a été pour nous l'occasion de développer depuis zéro un programme de taille conséquente. Nous nous sommes rendus compte que la plupart des problèmes rencontrés sont en réalité des problèmes de communication entre les développeurs.

Les méthodes que nous avons appliquées concernant la validation du code nous ont été très prolifiques. Plus généralement, nous avons réussi à prendre en main les différents outils de Github, ce qui nous a permis de mieux gérer le développement.



## Références bibliographiques

- [1] Antoine Bauza. Règles du jeu takenoko, Mar 2018. <https://www.matagot.com/IMG/pdf/takenokoregle-2.pdf>.
- [2] JeuxStrategie. Liste des pièces du jeu. [http://jeuxstrategieter.free.fr/Takenoko\\_complet.php](http://jeuxstrategieter.free.fr/Takenoko_complet.php).
- [3] Red Blob Games. Hexagonal grids, May 2020. <https://www.redblobgames.com/grids/hexagons/>.
- [4] WIKIPÉDIA. Takenoko, Mar 2016. <https://fr.wikipedia.org/>.