Rapport technique QGL

Smartware

Composé de Antoine Desanti, Mathias Carasco, Nicolas Perrin, Matthieu Hebrard

Année 2020 - 2021

Sommaire

I) Description Technique

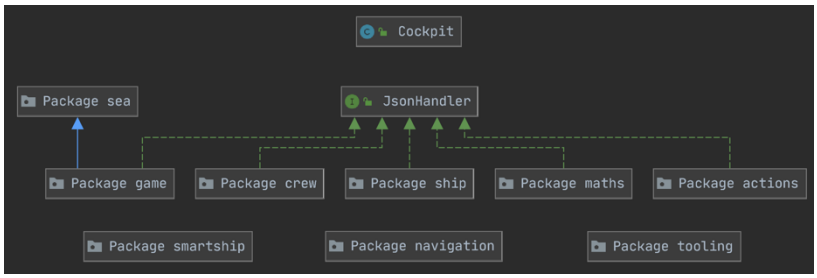
Architecture

Le *workflow* décisionnel de notre bateau à chaque tour est le suivant :

À chaque tour, notre Cockpit fait appel au RegattaHandler du Captain (quand il s'agit de ce mode de jeu) pour la prise de décisions. Le Captain va travailler avec des Advisors pour collecter des informations et ainsi savoir quelles Actions doivent être effectuées par les marins.

Notre utilisation de l'héritage rend notre architecture extensible : nous pouvons ajouter de nouvelles SeaEntities, de nouvelles formes, de nouveaux éléments sur le bateau ou encore de nouvelles actions. Même chose pour ajouter un nouveau mode de jeu.

Pour ce qui est de la sérialization/désérialization des JSON, nous effectuons ces processus sur des classes "InitGame" et "NextRound" constituants des miroirs de la structure donnée dans la documentation technique. C'est de ces instances de classes que nous allons par la suite récupérer les données requises par notre programme.



- ***Package game*** *: Implémente les classes relatives au contenu d'un jeu, à savoir le type de partie (Régate ou Bataille), ainsi que l'instance d'un jeu (données du bateau, collection de marins, shipCount).*

- ***Package crew*** *: Ce package inclut la classe "Captain". Il ordonne et gère les marins, la vigie, le gouvernail, les rames, et surveille les entités environnantes autour du bateau. Il dispose aussi de classes "Conseillères", que ce soit pour le choix de la trajectoire, des rames, de la vigie...*

- ***Package ship*** *: Ce package, en plus de contenir le modèle d'un bateau (c.à.d. sa forme, sa position, ses points de vie, son nom, ses entités) contient un sous-package "Entités". Ce dernier contient le modèle de toutes les entités possibles sur un bateau, toutes héritant d'une "Entity".*

- ***Package maths*** *: C'est ici la ToolBox mathématique de notre projet. Nous y trouvons, en plus de modèles définissant un Chemin, une Position, des classes relatives à la Géométrie, et d'autres purement relatives à une "Forme". Une classe importante est "OrthgonalCoordinateSystem". On utilise une instance de celle-ci pour se placer dans un sous-repère local, afin de réduire les problématiques d'orientation au sein du jeu (dans la mer et sur le bateau notamment).*

- ***Package actions*** *: Tous les modèles d'actions possibles y sont présents. On veille à respecter le Polymorphisme, de sorte à ce que chaque action implémente une méthode execute(), cette dernière étant abstraite dans la classe mère.*

- ***Package smartship*** *: On trouve ici des classes servant plus de modèle de données plutôt que de procédures, permettant d'enregistrer la répartition optimale des marins sur le bateau, ainsi que le chemin qu'ils doivent emprunter afin d'y accéder.*

- ***Package navigation*** *: C'est dans ce package que nous trouvons tous les algorithmes relatifs à la navigation. Cela passe aussi par des utilitaires utilisés par l'équipage du bateau pour se repérer sur la mer, en incluant un algorithme de contournement plus complexe. De plus, une classe "Navigator" se chargera de calculer pour le bateau la vitesse optimale à fournir, ainsi que l'angle le plus petit possible sans rentrer en collision. Il peut aussi "anticiper" la position qui sera renvoyée par le Referee Engine.*

- ***Package tooling*** *: Il s'agit de toute l'artillerie de simulation : Le simulateur d'arbitre et le générateur d'images illustrant à chaque tour la position du bateau, l'état des éléments sur le bateau et le contenu de la mer.*

Notre architecture admet l'ajout d'un nouveau mode de jeu. Nous avons extrait une méthode qui sera commune à ces modes de jeu : *moveShipToTarget*() de la classe Captain, qui permet de déplacer le bateau vers un point cible. Cette méthode pourra être réutilisable par tous les modes de jeux.

Choix algorithmiques

Nous avons effectué certains choix algorithmiques. Nous avons décidé que le *workflow* d'action d'un round se déroulerait toujours dans l'ordre présenté précédemment. Pour la voile, dès que le gain de vitesse est à 1 nous l'activons. Nous avons placé en premier la gestion de la voile car c'est elle qui peut nous faire gagner le plus de vitesse en ne mobilisant un marin que pour un tour, mais aussi nous en faire perdre si elle n'est pas retirée au bon moment. En effet se déplacer avec la voile ouverte à contre vent est très coûteux.

En ce qui concerne l'évolution du bateau dans un environnement complexe, pourvu de récifs et de courants, nous avons opté pour un algorithme d'évitement. Celui-ci n'intervient que lorsqu'un obstacle se trouve sur la trajectoire immédiate entre le bateau et le prochain *checkpoint*.

Nous avons fait le choix de détecter des collisions uniquement entre deux Polygones (== un tableau de Points). Ainsi, nous devons "Polygoniser" chacune des formes. Typiquement, un cercle sera discrétisé en un grand nombre de points. Nos tests n'ont pas montré d'inconvénient en termes de taux de détection, même dans des cas un peu poussés (Côté d'un carré qui frôle un cercle). Cependant, nous aurions pu adopter un meilleur Polymorphisme au niveau des formes géométriques.

Pour la vigie, nous avons développé une première version qui utilise un marin pour accroître le champ de vision lorsque nous ne voyons aucune entité sur la mer. Une seconde version a été développée mais n'a pas passé la phase d'intégration.

Celle-ci avait pour but de fonctionner de la manière suivante pour optimiser l'usage de la vigie :

- Une structure de données dans Sea enregistre tous les récifs rencontrés de manière unique en générant un Hash avec leurs attributs. Habituellement notre liste d'entités était écrasée à chaque tour.

- Une structure de données dans l'advisor de la vigie retient toutes les zones (des cercles d'approximation) dans lesquelles la vigie a déjà été appelée auparavant.

Avec ces deux structures de données, on peut appeler la vigie si rien n'apparaît sur la mer ET que la vigie n'a pas déjà été utilisée à l'endroit où le bateau se trouve. On évite ainsi d'appeler la vigie à un endroit où elle a déjà été utilisée, et le *pathfinding* bénéficie d'une liste plus complète d'entités pour ses calculs.

Le déplacement du bateau possède certaines limites : ayant utilisé un algorithme de contournement et non de *pathfinding*, le bateau possède une vision de la situation seulement avec un tour d'avance. Il peut uniquement prédire la meilleure position au tour n+1, mais en réalité pour passer les *weeks* avec le plus d'obstacles sur la carte il serait préférable de calculer un itinéraire / un chemin, avec un vrai algorithme de *pathfinding*. Néanmoins pour plusieurs *weeks* il est tout à fait envisageable d'utiliser un simple algorithme de contournement. Typiquement, si nous revenions en arrière, nous aurions utilisé un algorithme connu, tel que "Dijkstra". La problématique algorithmique aurait ainsi été écartée, et aurait laissé celle de l'intégration simplement. Toutefois, cela imposait aussi de passer à une structure de données de type "Graphe". Pour cela, il nous aurait fallu consacrer du temps supplémentaire pour implémenter cette architecture.

Pour résumer, le choix de notre architecture s'est fait dans la mesure du possible (en termes de temps, expérience, compétences et capacité de correction) dans le respect du Polymorphisme, de l'Abstraction des types, des responsabilités données aux classes, et de la faisabilité à tester leurs méthodes. Nous aurions aimé, si nous avions commencé le projet plus tard, implémenter des Design-pattern connus, tel que le Observer/Observable, le Singleton, les Factory.

II) Application des concepts vu en cours

Branching Strategy

Notre *branching strategy* fonctionne actuellement de la manière suivante : nous avons trois types de branches :

- La branche **master** dans laquelle nous avons notre version stable du projet et quelques rares *hotfixes*.

- La branche **development** qui permet de faire les *merges* des différentes *features* que chacun de nous met en place sur une branche à part. Cette branche est moins stable que la branche master.

- Les branches **feature/<feature_name>** propre à chaque *feature* dans laquelle une ou plusieurs personnes travaillent. Ces branches restent instables jusqu'à ce que la *feature* soit entièrement développée. Alors, nous « *mergeons* » notre branche dans *development* pour peaufiner notre *feature* et faire en sorte que l'intégration soit faite le plus proprement possible.

Notre *branching strategy* a été mise en place dès lors que nous avons vu en cours les façons de faire existantes. La *branching strategy* adoptée est alors venue assez naturellement et nous semble efficace. Nous avons ajouté au contributing.md de notre *repository* Github un fichier décrivant ce système de branches.

Qualité du code

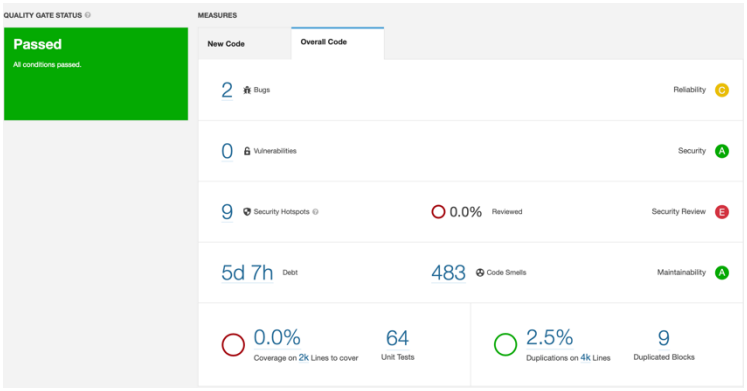
Tout au long du projet, nous nous sommes aidés le plus possible des outils présentés en cours tels que Pitest ou encore SonarQube. A chaque nouveau push sur *development* ou à chaque *pull request* sur *master*, un nouveau Build Maven ainsi qu'un rapport Pitest sont générés. Le rapport PiTest le plus récent est consultable sur notre Github Page.

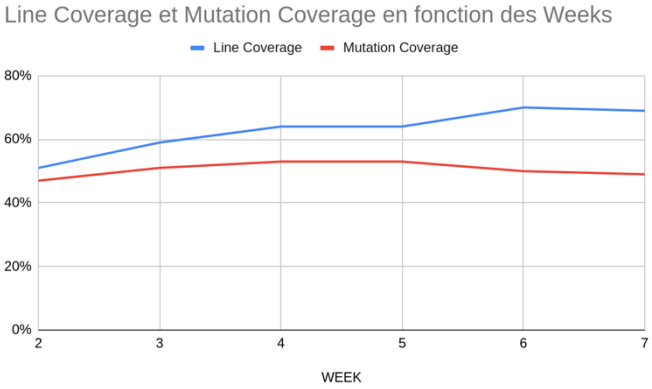
Nous avons ainsi pu nous servir des métriques des tests de mutations pour connaître la valeur réelle de nos tests unitaires, mais aussi de notre code.

L'outil Sonarqube a également été utilisé au cours du projet. Son utilisation est un peu plus lourde puisqu'en mode *standalone* il faut lancer une image *docker* sur son ordinateur avant d'effectuer l'analyse. Cet outil, bien que très puissant si utilisé correctement, n'a pas été suffisamment employé à cause d'une gestion de projet un peu compliquée sur certaines périodes. En effet, nous avons loué un VPS pour intégrer l'analyse Sonar à notre Github Actions, mais faute de bug qui se sont enchaînés à des moments cruciaux du projet, nous avons complètement mis de côté cette façon d'utiliser Sonar. Ainsi, nous nous référions souvent aux rapports que l'on générait mais pas aussi souvent que si nous avions pu implémenter cette solution.

Nous estimons que la qualité de notre code est correcte, avec une note globale de 6,5/10. Nous savons qu'il y a un certain nombre de faiblesses sur le découpage de code et la répartition des

responsabilités. De plus, la couverture de test n'est pas suffisante. Nous avons un bon indice de *maintainability* sur Sonarqube, en revanche nous avons une dette technique de 5 jours que nous avons accumulé. En effet nous avons été derrière les rendus à partir de la WEEK7 pour réussir à ajouter toutes les fonctionnalités à temps.





Avec une qualité inférieure, nous aurions certainement passé moins de WEEKs, et le fonctionnement du bateau aurait été trop instable. Avec une qualité supérieure, nous aurions pu passer moins de temps sur du débogage et sans doute passer plus de WEEKs.

La mesure de la qualité de notre projet avec Sonarqube et PiTest nous a permis d'obtenir des indicateurs sur notre progression, mais aussi d'identifier des faiblesses dans notre code.

Refactoring

Dans le package player, nous n'avions initialement pas d'organisation se rapprochant de la hiérarchie d'un navire. Nous avons donc fait un *refactor* avec un package *Crew* qui rassemble les membres d'équipage qui ont chacun un rôle précis, permettant de respecter le principe de *Single Responsibility*. Nous avons donc conçu une classe *Captain* qui est en haut de la chaîne de commandement. Le Captain se réfère à divers *Advisors*, des membres de l'équipages qui effectuent des calculs et conseils le fonctionnement du bateau. Le Captain compile ces informations pour les traduire en Actions à distribuer aux marins.

Nous avons automatisé le processus de *build* Maven et de lancement des tests de mutation comme décrit précédemment. Ce système nous a permis d'être confiant sur nos push sur Git pour être certain que notre code soit compilable, notamment pour les rendus hebdomadaires. Nous n'avons donc jamais eu de soucis de Build Maven. Les tests de mutation font partie du même workflow, ainsi nous n'avions qu'à nous rendre sur notre Github Page pour consulter les métriques PiTest.
Ces automatisations ont été un réel gain de temps, et nous aurions aimé réussir à intégrer SonarQube dans ce système.

L’objectif de plus long terme aurait été d’adopter une approche la plus “Dev-Ops” possible. Avoir un *workflow* complet sur chacune des phases, *le Building, Testing, Packaging, Releasing, Configuration* et *Monitoring*.
Nous sommes conscients qu’utiliser le maximum d’automatisation aurait réduit le taux d’échec sur les nouvelles *releases*, permis d’avoir une livraison plus continue et testée automatiquement, un temps de *fix* des bugs plus court entre les *releases*.
Typiquement, nous aurions aimée qu’à chaque *pull request* sur master, des tests de non-régression soient lancés avec en entrée le JSON gameConfig de l’ensemble des *weeks* précédentes, ce dernier étant récupéré automatiquement sur le WebRunner.

III) Étude fonctionnelle et outillages additionnels

Côté Player

Afin de maximiser les chances de victoire, nous avons privilégié l'efficacité d'exécution des tâches à bord du bateau.

Ainsi, au cours de la gestion des actions, nous opérons dans l'ordre suivant :

1) La vigie est activée lorsque nous n'avons aucune visibilité

2) Gestion de la voile : Celle-ci est hissée dans le cas où elle apporte un gain significatif de vitesse. Dans le cas contraire elle est retirée.

3) Le marin le plus proche rejoint la barre

4) Les marins se déplacent vers les rames selon les besoins.

5) Les marins se mettent à ramer.

Cet ordre d'exécution permet de traiter d'abord les phénomènes les plus importants, c'est à dire ceux ayant un fort impact sur la vitesse ou la trajectoire du bateau. Seulement dans un second temps, on vise la précision afin que le bateau se dirige encore plus rapidement vers sa destination.

Ces différentes catégories d'actions sont guidées par les *Advisors*. Un premier choix est celui d'activer la vigie ou non pour le tour selon notre visibilité actuelle des éléments. S'en suit le *SailAdvisor*, qui dit au Captain si la voile est utile. Le *Captain* choisit ensuite de la lever ou non. Le *TrajectoryAdvisor*, appelé en réalité en premier, calcule les combinaisons de rameurs à gauche et à droite avec les vitesses et orientations résultantes. Après la voile, les résultats fournis par le TrajectoryAdvisor sont analysés avec le nombre de marins restants. C'est là qu'est actionnée la gestion du *Rudder* et les déplacements des marins avec le QuarterMaster.

L'ensemble de ces opérations se déroule dans la méthode de haut niveau moveShipToTarget(Actions, Position). L'étape suivante est donc de définir quelle est le point cible *(target)* que le bateau doit rejoindre. C'est ici qu'intervient le PathFinding.

Path Finding

Nous ne parlerons pas à proprement parler de *Pathfinding* puisque nous avons mis en place un algorithme de contournement, qui aurait pu effectivement être étendu à un *pathfinding* si nous avions su résoudre l'ensemble de nos problèmes pour se diriger. Néanmoins, nous sommes parvenus à implémenter un algorithme de contournement valide d'un point de vue *proof of concept*, que voici:

Sur le principe, cet algorithme consiste simplement à observer l'environnement dans un rayon de 1000m (quand la vigie est inactive, sinon 5000m). Il s'agit, durant le balayage de ce périmètre, de regarder si un obstacle se situe sur la trajectoire du bateau à ce point.

Chaque point est ainsi affecté à un score qui tient compte du fait que le point soit dans l'axe du *checkpoint*, que la déviation de trajectoire ne soit pas trop importante, et enfin du fait qu'un obstacle se situe sur la trajectoire. Évidemment, si un récif se situe sur la trajectoire entre le bateau et le point testé, ce point est automatiquement éliminé (avec un score en conséquence) car une collision peut être fatale.

A la dernière étape, l'algorithme de contournement récupère le point qui a le meilleur score. Dès lors, le capitaine ordonne à ses marins de viser ce point et de le rejoindre. En revanche, dès lors que le bateau ne détecte plus aucun obstacle sur sa trajectoire jusqu'au prochain *checkpoint,* c'est justement ce *checkpoint* qui est pris pour cible à rejoindre. Ainsi, l'algorithme de contournement (et toutes ses étapes : balayage autour puis sur le long de la trajectoire) n'est actif que lorsqu'un contournement est nécessaire pour rejoindre le *checkpoint*.

Par manque de temps, nous n'avons pas réussi à résoudre les bugs que nous avions sur l'algorithme de contournement, donc nous l'avons désactivé pour le rendu final afin de passer la week 6 et la week 8. Avec le *pathfinding*, la week 10 passait correctement mais les chemins avec obstacles restaient assez hasardeux pour notre algorithme.

Enfin, la dernière stratégie que nous avions tentée de mettre en place pour augmenter nos chances de victoire était la gestion intelligente de la vigie, décrite dans la partie Architecture de ce rapport. Pour rappel elle devait permettre d'avoir une liste de toutes les entités rencontrées, et de savoir où est ce que nous avions déjà utilisé la vigie. Ainsi, le *pathfinding* aurait bénéficié d'une meilleure connaissance de la mer pour effectuer un trajet optimal, et nous éviterions de mobiliser un marin lorsque ce n'est pas nécessaire pour la vigie.

Côté Referee

Pour réussir le projet il nous est rapidement apparu comme une évidence que nous devions concevoir un système de simulation dans notre package Tooling. Nous avons alors développé un *RefereeEngine* respectant les contraintes techniques du projet pour pouvoir comprendre et observer le comportement de notre navire. Nous avons ensuite ajouté à ce RefereeEngine un générateur d'image pour obtenir à chaque tour une image de notre bateau sur la mer en tenant compte de sa position, son orientation, et des éléments disposés sur son environnement. Il y a également une vue de l'intérieur du bateau pour observer les déplacements des marins. Enfin, en plus de la génération d'une image par tour, un gif est généré pour avoir une animation complète de la partie. Typiquement, son appel est analogue à celui fait par le WebRunner. Au lancement, le RefereeEngine va retourner un fichier JSON au format *initGame* (en se basant sur un GameConfig.json donné dans le *WebRunner*). Puis, tant que le jeu n'est pas fini, notre programme appelle le nextRound() du Referee en lui donnant les actions choisies. Voici le déroulé :

1) On vérifie la validité des Actions

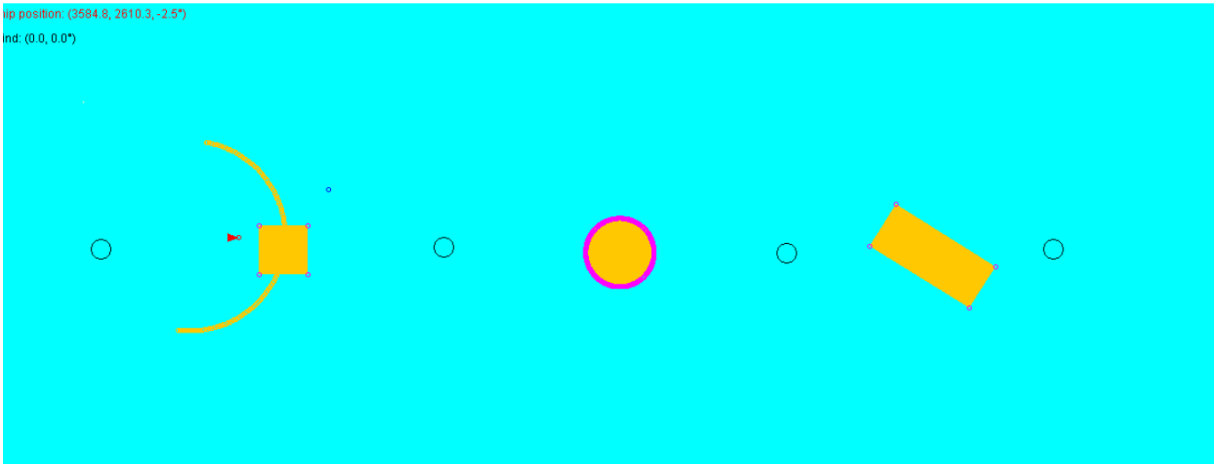
2) On les trie par ordre d'exécution

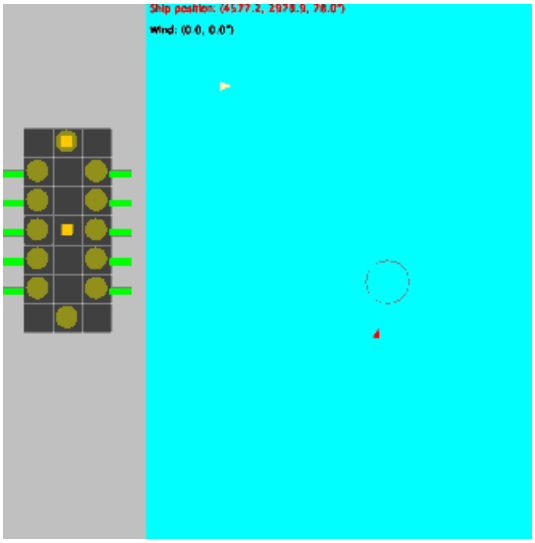
3) On les exécute avec un simple action.execute()

4) Envoi d'un NextRound avec la nouvelle position, et les entités proche d'un certain rayon autour du bateau, et simultanément génération de l'image PNG dans un dossier "output"

5) Si le jeu est fini, on envoie le signal de fin.

6) Un GIF concatène les PNG et affiche une animation du bateau qui se déplace





Un des outils que nous aurions aimé développer est un système pour effectuer des tests de régression. En effet, avec le *Referee* nous pouvons savoir si une course est terminée. Nous aurions alors pu avoir un système qui fasse passer les fichiers *initGame* et *GamConfig* de toutes les *WEEKs* précédentes pour constater qu'aucune modification ne fasse régresser notre code.
Nous aurions également voulu réussir à faire fonctionner le Sonarqube dans le workflow Github Actions. Cela nous aurait d'une grande aide pour maîtriser la qualité du code, d'autant plus si cela avait pu être mis en place tôt dans le projet.

Conclusion

Pour conclure, nous avons tiré divers enseignements de ce projet notamment grâce à des bases que nous avons d'autres matières.

La matière Algorithmique et Structure de Données nous a permis d'avoir une vision rapide des différentes manières d'implémenter notre code mais aussi et surtout de mieux comprendre quel bout de code est plus efficace qu'un autre. La complexité algorithmique est un point central à tout projet, il est donc essentiel de mettre en place les concepts qui y sont associés. Cependant sans les concepts fondamentaux et élémentaires étudiés en PS5, il nous aurait-été impossible de produire ce qu'est actuellement notre projet. Ainsi, les tests unitaires, les patterns GRASP et principes SOLID ont été des piliers pour la conception du code du projet.

Lors de ce projet, nous avons continué à améliorer notre façon de coder et surtout de coder proprement. C'est via ce projet que nous avons pu mettre en place des outils extrêmement utiles et approfondir nos connaissances sur des sujets tels que Maven ou Git. L'automatisation est sûrement le mot qui décrit le mieux notre travail ce semestre.

Comme nous l'avons déjà vu lors de PS5, les tests sont de plus en plus primordiaux à tout projet durable. Ce projet nous a ouvert les yeux sur cette qualité qui fera dorénavant partie de notre façon de travailler. Nous avons également été confrontés à la difficulté des rendus hebdomadaires avec une équipe composée de 3 alternants.

L'intérêt des tests d'intégration poussés et surtout de régression nous est apparu comme une illumination en fin de projet, lorsque nous nous sommes aperçus que des anciennes *weeks* ne passaient plus. Désormais nous savons que cela sera obligatoire pour nos prochains projets.

A travers les nombreux changements qu'a vécu le projet, il nous a été essentiel de tester plus que nous ne l'avions jamais fait pour pouvoir éviter de se noyer sous le flux d'améliorations. L'erreur qui nous a justement coutée cher et de ne pas l'avoir fait suffisamment durant la partie de mise place du *pathfinding* et des collisions. C'est sûrement cela qui nous a coûté le podium …

Ce projet nous a également permis de mieux comprendre comment fonctionne un projet bien mené avec des outils d'analyse.