

Rapport QGL



Nom de l'équipe

Les Gorilles Bronzés

Étudiants

Florian Latapie, Loïc Le Bris, Marius Lesaulnier, Thomas Paul



21 mai 2022

Université Côte d'Azur - Polytech Nice – SI3

2021 - 2022

Table des matières

Contexte	3
Description technique	3
Architecture du projet.....	3
ICockpit, JSONs et leurs impacts sur notre travail.....	4
Application des concepts vu en cours	6
Branching strategy.....	6
Qualité du code	6
Étude fonctionnelle et outillage additionnels	9
Régate.....	9
Bataille navale	9
Outils créés.....	10
Simulateur	10
« BumpViewer », Un visualiseur de fichiers .bump.....	11
Conclusion	12
Ce que nous avons appris lors de ce projet.....	12
Connaissances venant d'autres cours qui ont pu être exploitées pour le projet.	12
Leçons tirées du projet.....	12

Contexte

Ce projet de qualité et génie logiciel a pour but de nous faire apprendre les concepts de qualité de code au travers d'une course virtuelle de bateaux. Une course est composée d'un ou plusieurs bateaux qui doivent passer par tous les checkpoints dans l'ordre en évitant les récifs et en utilisant les courants à notre avantage. À bord de notre bateau : des marins, des rames, des voiles, une vigie et un gouvernail. À l'aide de tous ces outils, nous pouvons définir et suivre un chemin que nous calculons et renvoyons au simulateur pour qu'il nous déplace notre bateau. L'objectif final est de rendre un code testé et fonctionnel chaque semaine en étant si possible les plus rapides sur la course.

Description technique

Architecture du projet

Au cours de notre projet, nous avons pris plusieurs décisions que nous allons développer dans la suite du document.

La première phase de notre programme est la récupération des données, le cockpit récupère les données JSON et les transforme en objets java grâce à la librairie Jackson.

Nous avons gardé un cockpit le plus simple possible avec la seule responsabilité de sérialiser et désérialiser les données envoyées et reçues. Cela permet une responsabilité unique pour cette classe.

Une fois transformées au format Java, ces données sont envoyées au moteur global : le GlobalEngine qui s'occupe de diviser les responsabilités deux : responsabilité du pont et responsabilité de la navigation.

L'avantage d'une telle séparation est que la partie navigation n'a pas besoin de connaître la position des marins pour pouvoir prévoir le trajet à suivre. Seules des informations élémentaires sont données à la partie navigation pour pouvoir estimer la précision : nombre d'entités de chaque catégorie, nombre de marins assignés à chaque type d'entités

Le moteur de gestion du pont, le DeckEngine place de manière unique chaque marin en fonction de chaque entité, chaque entité définie comme libre fait appel à un marin. Nous avons choisi que les entités demandent un marin et non l'inverse, car cela nous permet d'être sûr que les entités qui ne sont pas des rames soient bien utilisées par des marins. Nous avons choisi de positionner en priorité les marins sur les autres entités que les rames et de ne jamais déplacer les marins au cours d'une course une fois placés. En effet, un ou deux rameurs de moins n'ont pas un gros impact sur un grand bateau comparé la précision que peut nous apporter un gouvernail ou aux informations données par une vigie. Par sécurité, nous déplaçons tous les marins vers leur destination à chaque tour, même si cela implique de se déplacer d'aucune case pour la plupart de la course. Étant donné que nous sommes limités à 5 déplacements par tour, nous avons fait le choix de prioriser le déplacement des marins sur l'axe des ordonnées, car dans la majorité des cas, un bateau est plus long que large. Si un marin est trop loin d'une entité au premier tour, nous l'approchons le plus possible de celle-ci pour le positionner au tour suivant.

Le second moteur, appelé par le GlobalEngine, est le NavigationEngine. Comme son nom l'indique, il s'occupe de la partie calcul de la navigation et direction du bateau. Il est appelé juste après le placement des marins et connaît donc les possibilités du bateau à chaque tour. Plusieurs données sont calculées et mises en cache à chaque appel pour économiser des calculs trop lourds, ce qui pourrait nous faire perdre une course pour cause de TIMEOUT. Les valeurs calculées en début de tour sont par exemple l'angle parfait à atteindre s'il n'y a pas de récifs sur le chemin, si l'on doit lever les voiles ou non. Chaque entité possède son propre moteur lui aussi, ce qui permet de réduire un maximum les responsabilités de cette grande classe.

Cette classe s'occupe aussi de générer les données pour la classe de recherche de chemin, qui est une implémentation d'un algorithme trouvé sur internet, nous sommes en qualité et génie logiciel et non en projet d'algorithmique, c'est pour cela que nous nous sommes permis de réutiliser un code préexistant pour cette partie du projet. Notre travail a été de l'adapter à notre projet.

Une fois que les deux moteurs et leurs sous-abstractions ont calculé tout ce qui est nécessaire, ils renvoient leurs actions au moteur de jeu principal, puis celui-ci s'occupe de l'envoyer au cockpit qui retransmet les informations au format JSON au simulateur.

L'architecture de notre projet est extensible après quelques modifications dans le code. Prenons le cas où nous souhaitons implémenter les canons dans le bateau avec la possibilité de tirer sur un bateau ennemi. Pour cela, nous commencerons par créer l'objet canon et l'initialiser avec les données du JSON (ses coordonnées sur le bateau par exemple) comme pour les autres entités. Il suffirait de modifier le code responsable du pont du bateau (DeckEngine) pour placer des marins sur les canons. Nous pourrions ensuite créer un nouveau moteur de jeu qui serait en charge de détecter l'ennemi, éventuellement arrêter le bateau pour viser et tirer sur l'ennemi.

Nous devons évidemment faire la distinction entre les différents modes de jeu avant de lancer la partie, car les objectifs entre le mode BATTLE et REGATTA sont différents.

ICockpit, JSONs et leurs impacts sur notre travail

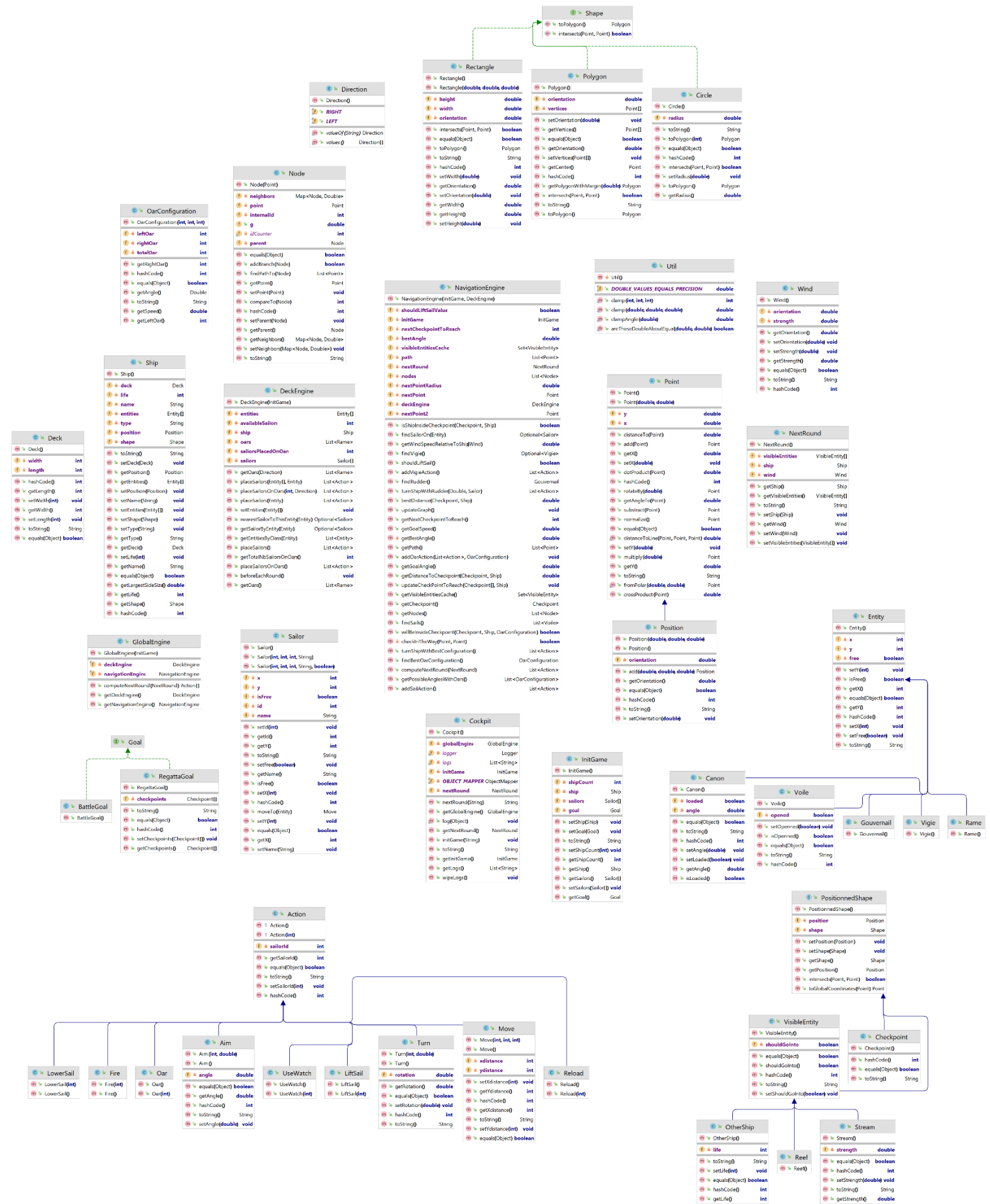
Nous avons dû travailler dans l'interface ICockpit et des données en entrée et sortie de notre programme en JSON, pour les fichiers d'initialisation (initGame), et tour à tour (nextRound).

L'impact dans notre projet est le suivant :

Nous avons choisi de garder les mêmes noms de classes et tout trier dans des sous packages, cela nous a permis de bien séparer les responsabilités et d'appliquer les bonnes abstractions en reprenant l'architecture déduite depuis le JSON. Nous avons pu identifier de l'héritage entre les différentes données, ce qui nous a permis d'éviter la duplication de code.

Rapport QGL : Les Gorilles Bronzés

Voici une représentation graphique de l'architecture de notre projet



Application des concepts vu en cours

Branching strategy

Au cours du projet, nous avons travaillé chaque semaine sur une branche propre à chaque nouvelle fonctionnalité (branche de feature) que nous devons implémenter. Cette “branching strategy” est communément appelée Git flow. Cette stratégie nous permet de toujours avoir un code sur master qui soit fonctionnel et prêt à la livraison chaque semaine. Nous avons choisi cette branching strategy car elle nous paraissait pertinente pour ce projet : chaque semaine, les demandes pour la livraison sont des features précises à implémenter, de manière incrémentale.

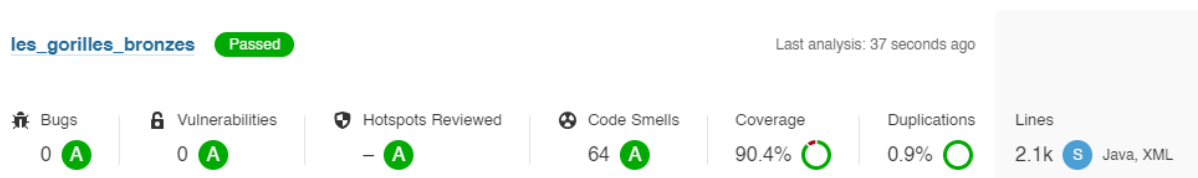
À la fin de la semaine, nous fusionnons notre branche de feature dans la branche développement. Cette branche nous permet de s’assurer que tout fonctionne correctement, puis on merge dans la branche master pour y ajouter le tag de la semaine.

Qualité du code

Maintenant, nous allons analyser la qualité de notre code et comment nous avons mesuré celle-ci en utilisant des plugins comme SonarQube et PITest.

Nous avons gardé comme objectif tout au long du développement de fournir un code de qualité. Malgré tout, certaines classes pourraient être ré-agencées et divisées en sous classes afin d’avoir un code plus propre et plus lisible. Par soucis de temps et étant donné que nous devons rendre un code qui passe les courses chaque semaine et qui nous permettait de rester compétitif, nous avons allié du mieux possible cet aspect de compétitivité tout en produisant du code le plus qualitatif possible.

Voici les captures d’écran du rapport SonarQube à la fin du projet



La plupart de nos “code smells” sont basés sur le fait que nous avons dû tester les .equals de nos objets pour avoir de bons tests de mutation, cela est repéré comme un code smell par l’outil d’analyse de code.

Code Smell +1

1 An equals check is performed here, which is better expressed with `assertEquals`.

Use `assertNotEquals` Instead.

Code Smell +1

Use `assertNotEquals` Instead.

Code Smell +1

Use `assertEquals` Instead.

Code Smell +1

Use `assertNotEquals` Instead.

Code Smell +1

Use `assertNotEquals` Instead.

Code Smell +1

Use `assertEquals` Instead.

Code Smell +1

Use `assertNotEquals` Instead.

Code Smell +1

Use `assertNotEquals` Instead.

Code Smell +1

```

13         assertEquals(2.0, aim.getAngle());
14     florin... assertEquals(1, aim.getSailorId());
15     assertEquals(3.0);
16     assertEquals(3.0, aim.getAngle());
17     aim.setSailorId(2);
18     assertEquals(2, aim.getSailorId());
19     assertEquals("Aim sailorId=2(angle=3.0)", aim.toString());
20
21 70631...
22 florin... assertEquals(1 aim.equals(aim));
23
24 florin... assertEquals(aim.equals(null));
25
26 florin... assertEquals(aim, new Aim(2, 3.0));
27 70631... assertEquals(aim.hashCode(), new Aim(2, 3.0).hashCode());
28 70631... assertNotEquals(aim, new Aim(2, 4.0));
29 70631... assertNotEquals(aim.hashCode(), new Aim(2, 4.0).hashCode());
30 florin... }
31
32 @Test
33 void fireTest() {
34     Fire fire = new Fire(1);

```

Use `assertEquals` Instead. Why is this an issue?

Code Smell Major Open Not assigned 2min effort

17 minutes ago • L22 1 🔗

🔗 junit, tests

Use `assertNotEquals` Instead. Why is this an issue?

Code Smell Major Open Not assigned 2min effort

17 minutes ago • L23 1 🔗

🔗 junit, tests

Use `assertNotEquals` Instead. Why is this an issue?

Code Smell Major Open Not assigned 2min effort

17 minutes ago • L24 1 🔗

🔗 junit, tests

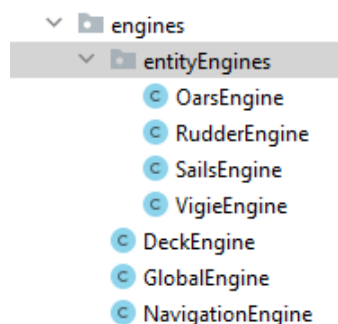
Nous avons dû, durant le projet : fournir un code de qualité, nous allons donc voir ci-dessous les impacts de la qualité de ce code tout d'abord sur notre projet et ensuite sur nos livraisons ainsi que sur l'organisation du travail dans l'équipe.

Par moments, nous nous sommes retrouvés avec une qualité de code moindre et cela peut avoir des impacts sur le projet. Une qualité moindre du code signifie que nos méthodes ne fonctionnent pas complètement comme on le souhaiterait ou qu'elles ne détectent pas les éventuels problèmes. Si la qualité de notre code était moindre, nous aurions un contrôle moins précis sur les comportements de chaque méthode et classes. Ce qui pourrait entraîner à terme des courses qui ne passent plus ou de façon moins optimisée. Avec un code de meilleure qualité, notre code serait plus lisible et plus facilement testable. Ce qui rendrait les phases de tests et de débogage moins longues et moins complexes. On aurait alors un code qui respecte bien les principes SOLID et qui est correctement structuré.

Les tests unitaires nous permettent de vérifier que le code que nous avons produit fonctionne bien comme on le souhaite : par exemple, au début du projet, si on demandait au marin de bouger de plus de 5 cases, il ne se déplaçait pas, car il y avait un oubli dans le code. Grâce à un test unitaire qui vérifiait l'efficacité de notre fonction, on a pu repérer une dysfonction. Cela a finalement permis de résoudre ce problème.

À l'aide des outils que nous avons utilisés tout au long du projet, et notamment SonarQube, nous nous sommes facilement rendus compte de la qualité générale de notre projet. En effet, le module SoftVis3D (CodeCity) nous a permis de visualiser simplement la qualité de notre code. À partir de cela, on a pu se concentrer sur les tests lorsque la qualité qui ressortait du rapport SonarQube ne nous convenait pas suffisamment. On pouvait alors livrer un code plus qualitatif.

Nous avons effectué un refactoring global que nous n'avons pas merge dans main, car celui-ci provoque un timeout dans le QGL Web Runner. Dans notre projet, la logique du déplacement du bateau est gérée par le navigation engine cependant nous avons remarqué que cette classe était trop longue et avait trop de responsabilité. Nous avons donc décidé de la séparer en différentes sous-classes permettant de gérer l'action sur les différentes entités dans des engines différents. Navigation engine calcul une direction, un angle, une vitesse à adopter et appelle ensuite successivement les différents entity engines qui lui retourneront une liste d'action des différents marins.

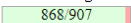
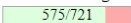
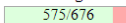


La génération automatique de PIT-Reports nous a permis de gagner du temps et d'être tous informés sur l'évolution de la qualité du code chaque semaine. En effet, nous n'avons plus à générer

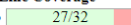
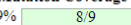
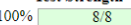
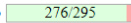
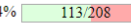
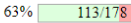
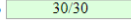
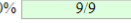
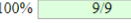
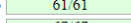
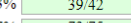
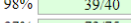
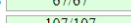
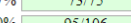
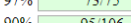
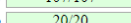
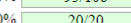
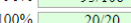
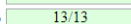
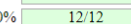
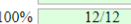
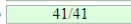
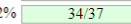
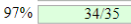
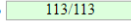
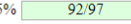
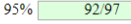
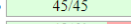
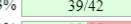
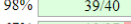
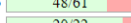
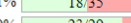
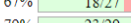
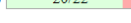
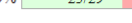
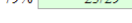



des PIT-Reports par nous même à l'aide de la commande Maven. Nous avons délégué cette tâche dans une GitHub Action qui génère un PIT-Report dès qu'on merge notre branche de feature dans la branche de développement ainsi que dans celle de master. Ce document est ensuite envoyé sur notre canal Slack pour nous permettre de le consulter.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
31	96%  868/907	80%  575/721	85%  575/676

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
fr.unice.polytech.si3.qgl.les_gorilles_bronzes	1	84%  27/32	89%  8/9	100%  8/8
fr.unice.polytech.si3.qgl.les_gorilles_bronzes.engines	3	94%  276/295	54%  113/208	63%  113/178
fr.unice.polytech.si3.qgl.les_gorilles_bronzes.objects	2	100%  30/30	100%  9/9	100%  9/9
fr.unice.polytech.si3.qgl.les_gorilles_bronzes.objects.actions	4	100%  61/61	93%  39/42	98%  39/40
fr.unice.polytech.si3.qgl.les_gorilles_bronzes.objects.geometry	3	100%  67/67	97%  73/75	97%  73/75
fr.unice.polytech.si3.qgl.les_gorilles_bronzes.objects.geometry.shapes	3	100%  107/107	90%  95/106	90%  95/106
fr.unice.polytech.si3.qgl.les_gorilles_bronzes.objects.goals	2	100%  20/20	100%  20/20	100%  20/20
fr.unice.polytech.si3.qgl.les_gorilles_bronzes.objects.obstacles	1	100%  13/13	100%  12/12	100%  12/12
fr.unice.polytech.si3.qgl.les_gorilles_bronzes.objects.obstacles.visible_entities	3	100%  41/41	92%  34/37	97%  34/35
fr.unice.polytech.si3.qgl.les_gorilles_bronzes.objects.ship	4	100%  113/113	95%  92/97	95%  92/97
fr.unice.polytech.si3.qgl.les_gorilles_bronzes.objects.ship.entity	3	100%  45/45	93%  39/42	98%  39/40
fr.unice.polytech.si3.qgl.les_gorilles_bronzes.pathfinding	1	79%  48/61	51%  18/35	67%  18/27
fr.unice.polytech.si3.qgl.les_gorilles_bronzes.util	1	91%  20/22	79%  23/29	79%  23/29

Étude fonctionnelle et outillage additionnels

Nous avons étudié le sujet, et nous pouvons donner nos deux stratégies, la première implémentée dans le projet, la seconde est celle que l'on aurait mis en place pour tenter de gagner ce mode de jeu.

Régate

Notre stratégie est en deux parties. Dans le premier cas, il n'y a pas de récifs en chemin sur les deux prochains checkpoints, nous définissons le point à atteindre le plus proche du suivant pour gagner un maximum de tours sur les courtes courses. Le second cas, il y a au moins un récif en chemin, on envoie toutes les données qui ont été observées à la classe de recherche de chemin, celle-ci nous retourne des checkpoints intermédiaires à atteindre pour arriver à destination, l'avantage est que nous avons pu réutiliser le code de déplacement du bateau vers un checkpoint réel pour atteindre nos checkpoints fictifs. Cela nous a donc permis d'éviter une duplication de code, ce qui aurait été une mauvaise pratique en termes de qualité de code.

Bataille navale

Pour cette partie, notre stratégie serait d'implémenter un évitement actif des bateaux : prévoir leur trajectoire, au départ en mode MVC en prévoyant juste une ligne droite, mais plus tard éventuellement en faisant une prédiction de leurs virages.

L'objectif serait de se cacher derrière des récifs au début de la partie en attendant que les premiers bateaux s'entre-tuent entre eux puis une fois les ennemis affaiblis nous arriverions avec nos canons pour les tuer et ainsi gagner la partie.

Outils créés

Nous avons créé 2 outils qui sont disponibles dans la version finale du projet :

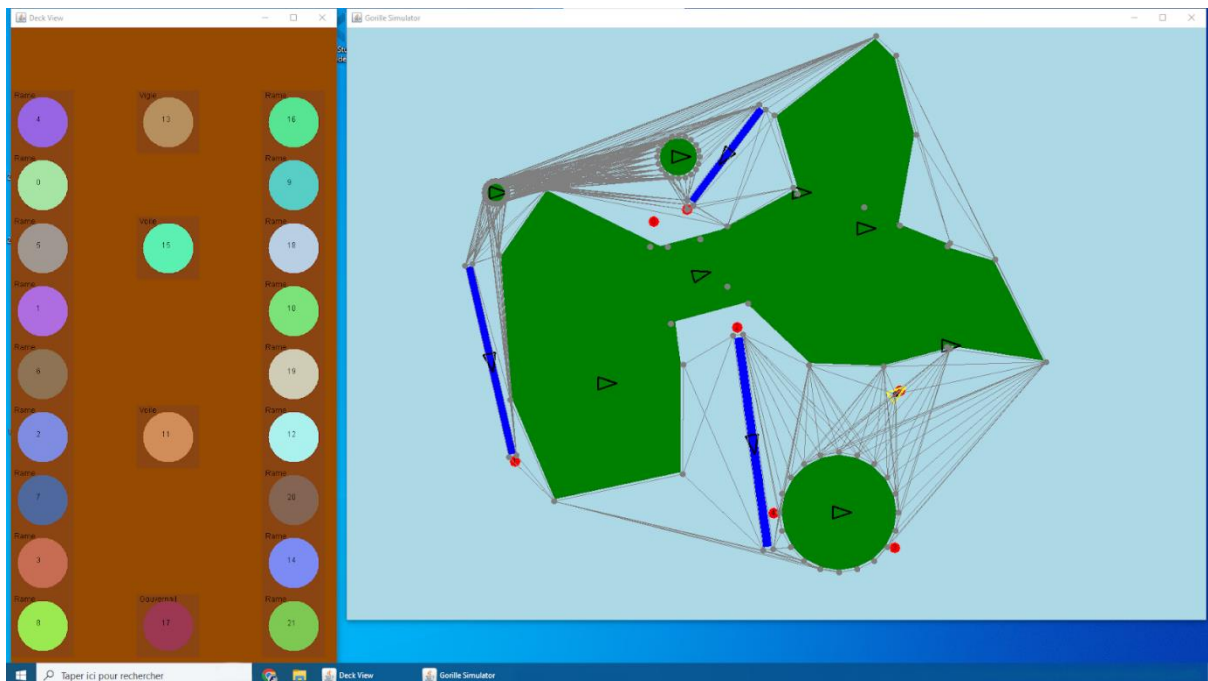
Simulateur

Nous avons créé un simulateur est une version simplifiée du QGL Web Runner donné pour tester et évaluer notre code, il reprend les actions les plus importantes : déplacement et actions des marins, déplacement du bateau, il ne gère pas les courants ni les collisions , mais est suffisant pour tester si notre code marche. Dans le but de pouvoir tester avant d'utiliser des crédits sur le QGL Web Runner, l'avantage de notre simulateur, c'est qu'il permet de visualiser les arêtes du graphe de recherche de chemin. Nous affichons aussi en temps réel sur une fenêtre différente le pont du bateau et les marins, ce qui permet de vérifier qu'ils sont bien placés pendant la course.

Notre simulateur est limité en 2 points, il envoie tous les récifs à chaque tour, l'utilisation de la vigie est inutile sur notre simulateur. Le second point est que les tours sont calculés en une fois et non subdivisés comme sur le réel simulateur. Les subdivisions n'ont pas été calculées dans une optique de MVC. La fonction vigie n'est pas gérée, on donne l'entièreté des récifs et courants de la carte au bateau à chaque tour.

```
recherche de fichiers dans : C:\Users\flori\Documents\GitHub\qgl-21-22-les_gorilles_bronzes\tooling\src\main\java\simulator\weeks/
choisissez le fichier en écrivant le numéro :
0 : InitGame.json
1 : WEEK10.json
2 : WEEK8-PREVIEW.json
3 : WEEK9-PREVIEW.json
```

Ici, il est demandé à l'utilisateur la course qu'il souhaite visualiser, il entre la valeur au clavier (le numéro avant le nom du fichier).

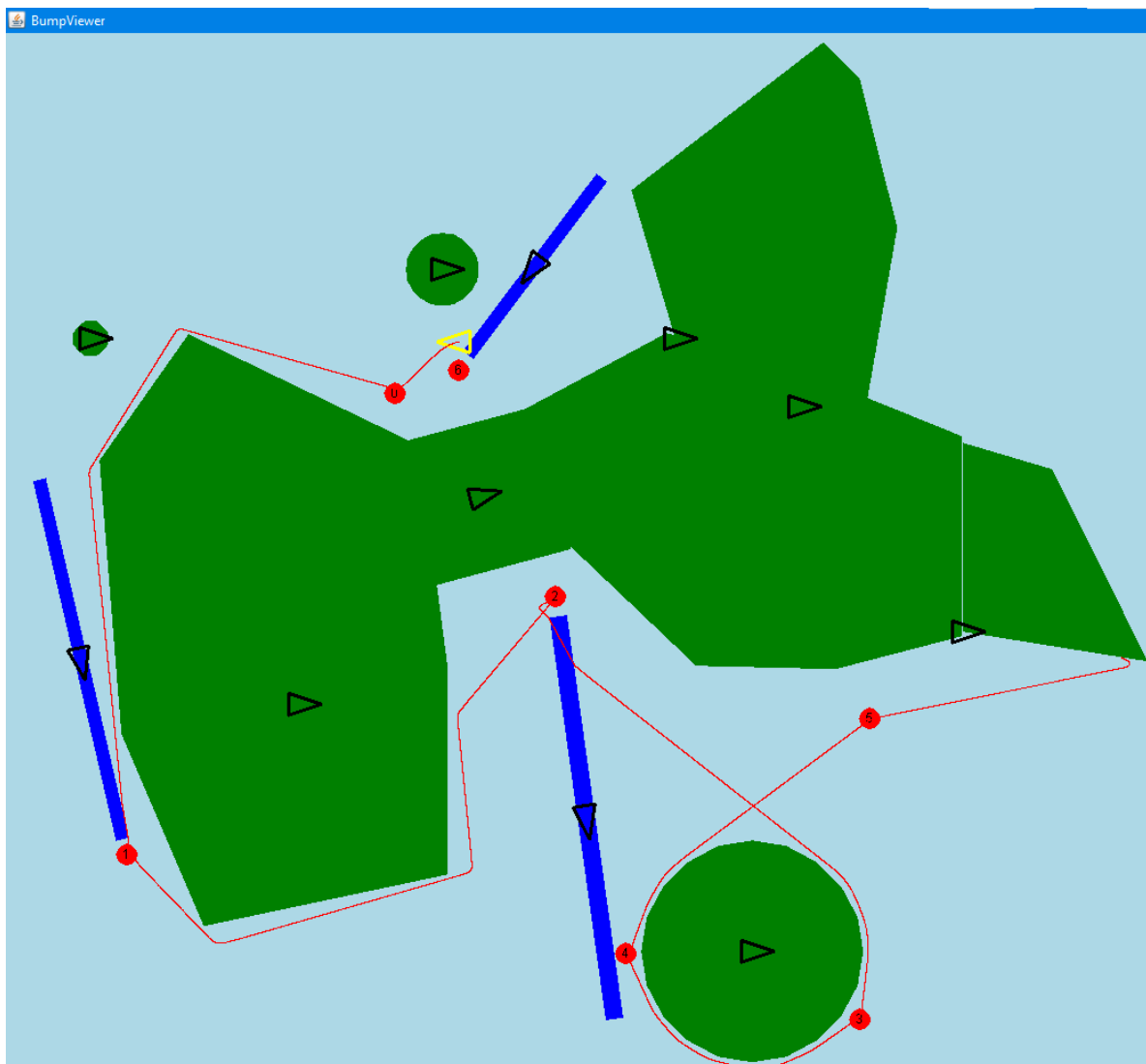


Ici il y a deux fenêtres qui sont lancées : à gauche la vue du deck “deck view” et à droite l’affichage du simulateur en lui-même. Le deck view permet de voir le positionnement des marins et des entités pendant la course, cela est utile pour pouvoir voir si nos marins sont bien positionnés.

« BumpViewer », Un visualiseur de fichiers .bump

Après une courte discussion avec M. Cousté nous avons décidé de créer un programme qui permet d’afficher correctement les données de fichiers .bump renvoyés par le QGL Web Runner. En effet, nous avons constaté que l’échelle entre le chemin parcouru par le bateau et les récifs n’étant pas les mêmes, il pouvait y avoir un doute. Afin de clarifier ce doute, le BumpViewer a été créé.

Ce programme réutilise une grosse partie du simulateur précédent, les contrôles sont donc identiques.



Conclusion

Ce que nous avons appris lors de ce projet

Nous avons appris à utiliser des nouveaux outils découverts en cours, notamment PITest, SonarQube, Maven et GitHub Actions. PITest et SonarQube nous étaient le plus utile pour avoir des métriques sur la qualité du code que nous étions en train de développer. L'utilisation des données JSON était une découverte pour nous tous mais facile à prendre en main une fois les spécificités de la dépendance comprises. La notion de compétition au cours de ce module nous a vraiment motivé à rendre un code fonctionnel et régulièrement.

Nous avons également appris lors de ce projet à s'organiser et travailler en groupe en passant par un Trello et une branching strategy.

La mise en place de certains principes SOLID nous permettent de rendre un code de qualité, et plus facilement maintenable.

Connaissances venant d'autres cours qui ont pu être exploitées pour le projet.

Pour réaliser le pathfinding, nous avons utilisé les cours sur les graphes qui sont enseignés en ASD et en PCP. Nous sommes également repartis des connaissances enseignées lors du projet du semestre S5. En effet, nous avons mis en pratique le MVP (minimum viable product) pour implémenter des fonctionnalités chaque semaine.

Leçons tirées du projet

La leçon que nous tirons dans ce projet est que nous devons plus prévoir notre code en amont pour pouvoir coder plus efficacement. Cependant, nous avons remarqué que lorsqu'on ne nous met pas de contraintes pour utiliser les milestones et les issues comme le PS5 nous avons tendance à ne pas les utiliser. Néanmoins, nous devrions prendre le temps de rédiger ces issues, car elles nous permettraient de mieux découper les tâches à faire au sein du groupe et éviter que plusieurs personnes travaillent sur la même chose. Pour un projet avec seulement 4 personnes, cela ne nous semble pas indispensable, mais en entreprise, ce sera nécessaire pour avoir rapidement un aperçu de ce qui a été fait par d'autres contributeurs.