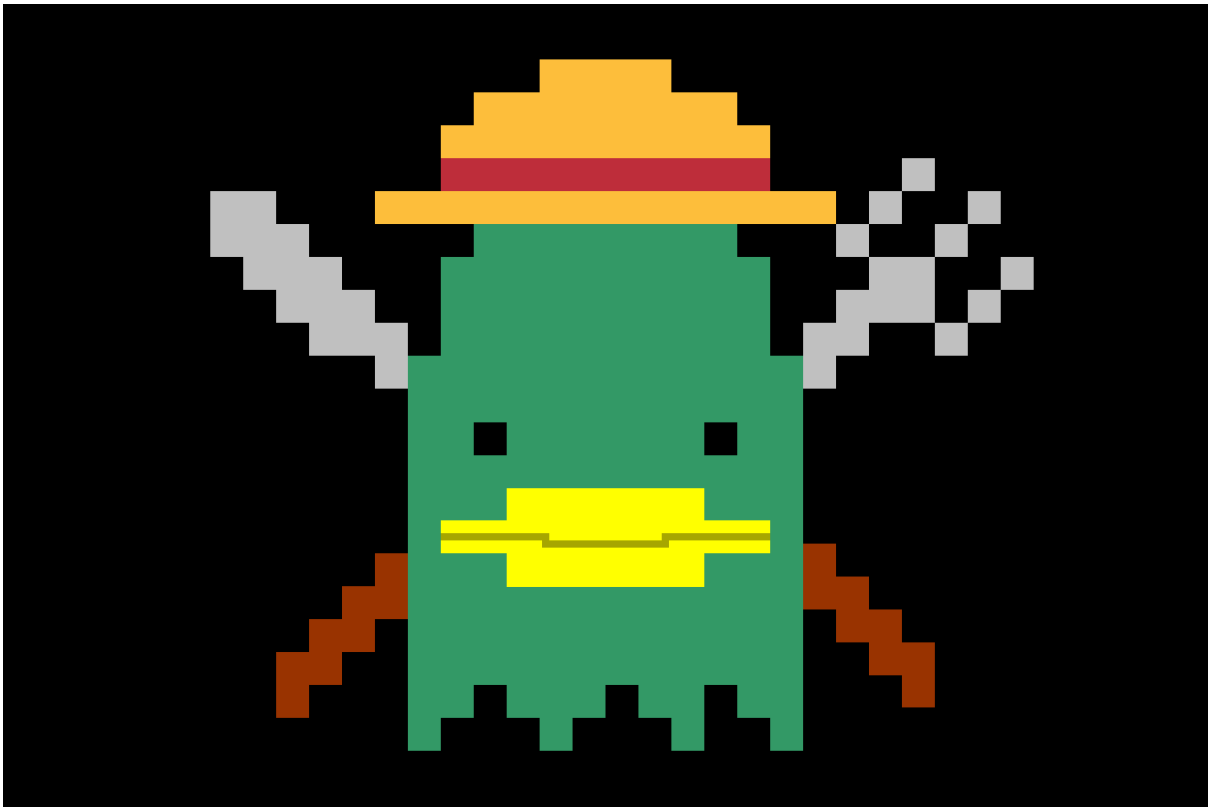
SI3 - Rapport Technique de QGL

Nom et Id: Mugiwara_Cook



2021-2022

Université Côte d'Azur - Polytech Nice – SI3-QGL
Alexandre Nunes Dias - Emile Derain - Habib Belmokhtar - Pauline Devictor

Sommaire

Table des matières

Description technique

Architecture du projet

À l'initialisation, notre programme et notamment la classe CaptainSailorMove, assigne les marins à des positions et, de ce fait, des postes (ils peuvent être rameurs, timoniers ou encore gabiers). Une fois les postes assignés les marins vont à leurs postes. En même temps, la carte représentant le monde dans lequel notre bateau va devoir naviguer se crée.

Une fois tout ça mis en place, le bateau va se diriger vers le premier checkpoint, notre bateau ira toujours vers le checkpoint peu importe s'il rencontre un récif, c'est pour cela qu'en fonction des obstacles que notre collisionDetector détectera, nous créerons des checkpoints intermédiaires où notre bateau pourra se rendre sans danger. Une fois la nouvelle route créée avec des checkpoints intermédiaires, vient la phase d'orientation et de déplacement du bateau, dont voici le processus :

1. Calcul de tous les deltas[1] (1er Tour).
2. Calcul de toutes les distances possibles en fonction de nos deltas (1er Tour).
3. Calcul de l'angle entre le prochain checkpoint et l'orientation de notre bateau.
4. Calcul de la distance entre notre bateau et le prochain checkpoint.
5. Déterminer si le bateau doit tourner à gauche ou à droite.
6. Regarder si l'angle THÊTA(Θ) est compris entre -45 et 45 : si oui, on s'oriente grâce au gouvernail de THÊTA(Θ) et on rame pour se rapprocher du checkpoint avec : (*distancecheckpoint* - *distanceapresdeplacement* >= 0). Sinon, on cherche l'angle à droite ou à gauche le plus proche de THÊTA(Θ) en s'orientant au maximum avec les rames (les compositions sont trouvée à partir du delta) et on corrige la marge d'erreur avec le gouvernail.

Enfin, à chaque fois qu'un nouveau récif est détecté, nous actualisons la carte.

Cette architecture n'est malheureusement pas extensible. En effet, nous sommes tous encore un peu débutant et viser aussi loin et surtout viser un objectif qui ne nous était pas donné au départ fut compliqué. Cependant, ce qui rend notre architecture non extensible est l'utilisation de la classe Captain : cette classe est une sorte de classe "racine", et son fonctionnement est basé sur le mode de jeu "Regatta" proposé en QGL.

Pour jouer un autre mode, il faudrait modifier un peu la classe Captain pour que les méthodes qu'elle appelle dépendent du mode de jeu et que les appels de classes de décisions telles que ChoseAction dépendent aussi du mode de jeu. Nous pourrions imaginer une classe ChoseActionBattleRoyal qui prendrait des décisions qui dépendent du mode "BattleRoyal" et modifier notre classe actuelle ChoseAction en ChoseActionRegattaGoal,

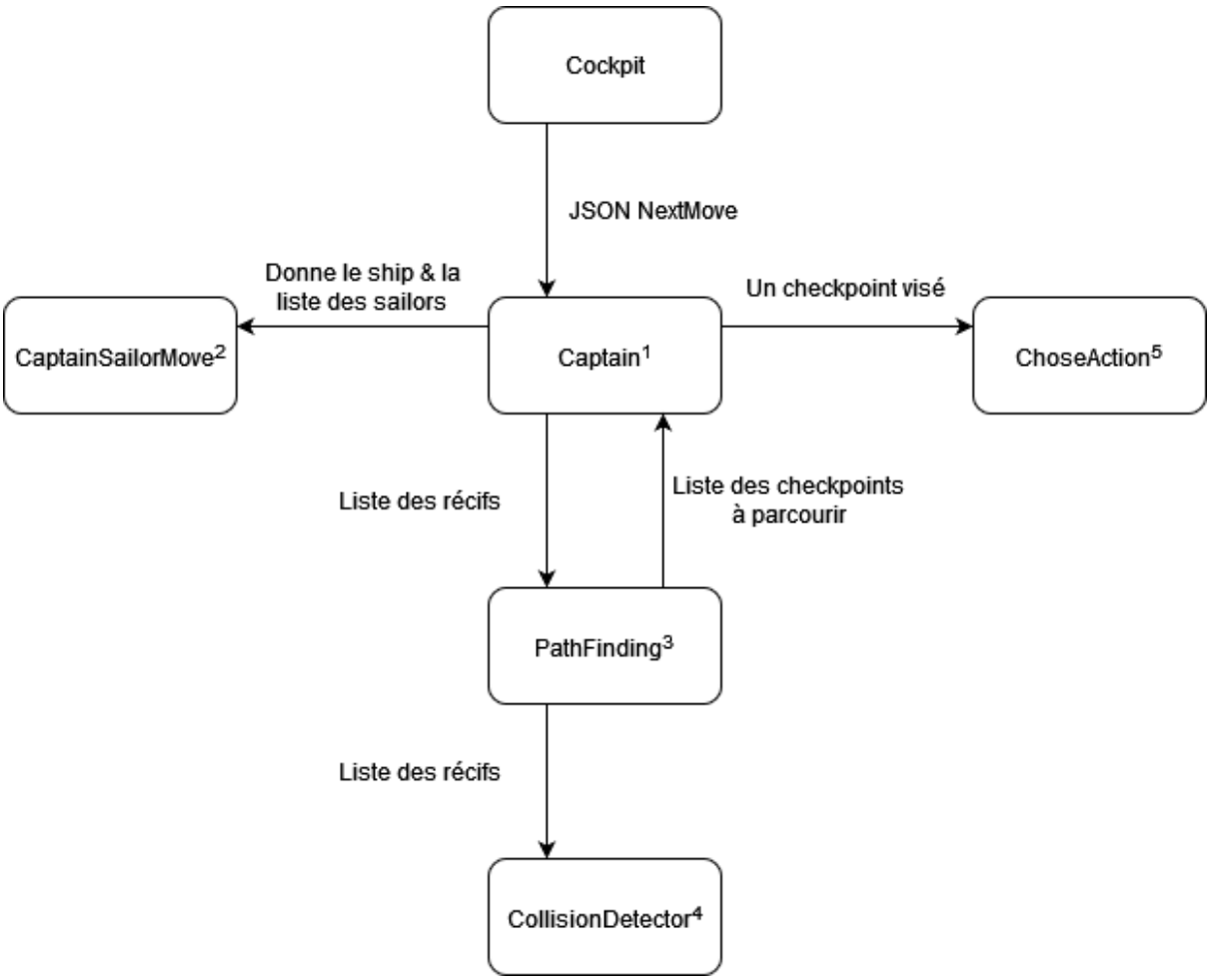
[1]deltas = ensemble des différences du nombre de rames entre la droite et la gauche du bateau, le delta 1 par exemple représente une différence de 1 rame à droite, libre à nous ensuite de choisir le nombre de rames à droite et à gauche pour faire ce delta.

ces deux classes héritant d'une classe mère ChoseAction qui serait appelée en permanence par Captain, mais dont les effets seraient différents en fonction du mode.

Nous avons implémenté l'algorithme de recherche A* pour notre pathfinding. L'intégration fut plutôt simple puisqu'il suffisait de diviser la carte en plein de petites cases puis de donner en entrée de notre algorithme de recherche la case où l'on se trouvait et la case vers laquelle on voulait aller. L'algorithme nous renvoie alors une liste de checkpoints qu'il fallait suivre pour arriver à bon port. L'algorithme est assez performant même si le nombre de cases est élevé, néanmoins nous avons rencontré un problème : notre bateau était trop lent pour finir la course dans les temps à cause du nombre trop élevé de checkpoints à traverser avant de finir la course.

Voici les 5 classes qui joue un rôle important pour faire avancer notre bateau:

1. Captain est la classe qui centralise toutes les données en faisant appelle aux différentes classes suivantes.
2. CaptainSailorMove a la responsabilité du déplacement des marins sur le bateau.
3. PathFinding a la responsabilité de créer un chemin, en évitant les récifs jusqu'à un checkpoint.
4. CollisionDetector a la responsabilité d'actualiser la carte avec les nouveaux récifs visibles pour que le pathfinding puisse les éviter.
5. ChoseAction a la responsabilité de choisir les actions à faire (nombre de rames, voile, …) pour arriver à un checkpoint.



Impacts des contraintes imposées

Les informations délivrées par nextRound et par initGame sont tellement importantes que cela nous a malheureusement porté préjudice au début du projet. En effet, avant notre premier refactor, quasiment toutes nos méthodes de décision dépendaient fortement de nextRound et initGame, mais principalement de nextRound.

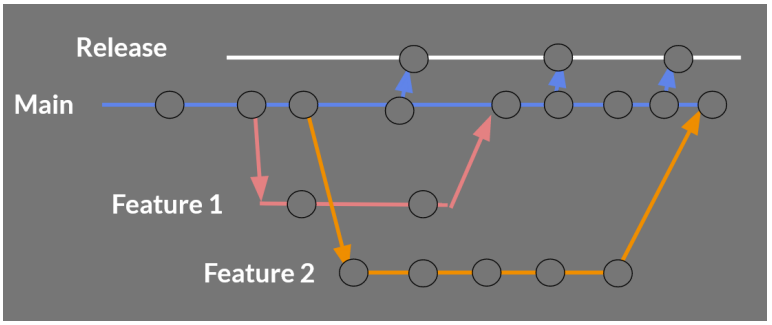
Après notre premier refactor, nous avons réussi à alléger fortement la dépendance de nos classes vis-à-vis de nextRound. Cependant, concernant initGame, certaines de nos classes dépendent beaucoup des informations délivrées par celle-çi, une erreur de choix de conception a été faite, celle d'attribuer des sailors à des équipements particuliers, par exemple, initGame est capable de nous donner la liste des sailors assignés aux rames. Il est clair que ce choix de conception est une erreur qui nous coûte cher, et qui crée, pour nos classes de décision notamment, une grosse dépendance vis-à-vis de celle-ci.

Concernant ICockpit, nous n'avons rencontré aucun problème. Elle a été utilisée uniquement pour son rôle. De plus, aucune classe ne dépend d'elle (elle est utilisée uniquement dans notre simulateur qui n'est pas compris dans notre rendu).

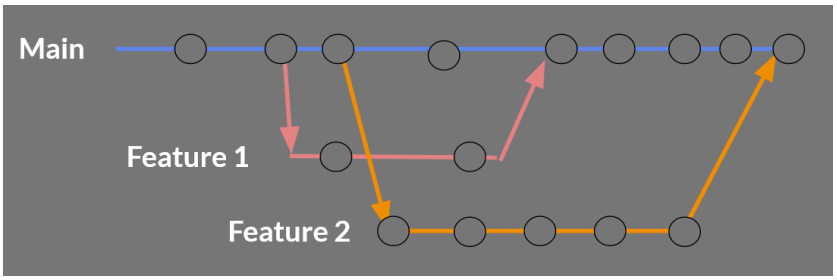
Application des concepts vu en cours

Git Branching Strategy

Initialement, nous avions prévu pour notre branching strategy, de faire une branche Release pour le code stable à délivrer de façon hebdomadaire. Le reste du temps, le code est sur la branche Main pour corriger les bugs et améliorer sa couverture de test. Enfin pour toute feature pouvant entrer en conflit avec le code existant, nous créons une nouvelle branche associée au nom de la feature. Nous avons gardé cette stratégie les premières semaines puis nous avons abandonné la branche Release après avoir eu de nombreux problèmes : branche existante mais qui n'apparaissait pas dans la liste et erreurs de commandes lors des tentatives de cherry-picks.

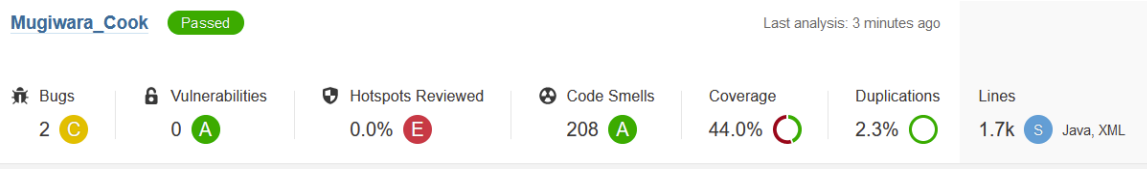


Finalement, nous avons gardé le même modèle sans la partie Release. Pour les Features, nous détruisons régulièrement les branches qui ne sont plus utilisées. Ce qui correspond à un Github Flow. Cette branching strategy est plus simple que celle prévue originellement, le code peut parfois se retrouver instable lors de merges avec les branches de feature mais nous n'avons pas eu de réel problème sur ce point là.



Qualité du code

La qualité de notre code a grandement évolué tout au long du projet, nous avons commencé le projet en étant emportés par la compétition et nous en avons oublié le sujet principal de ce cours : la qualité. Notre couverture de tests était presque inexistante et nous avons donc dû délaisser la scène compétitive pour nous concentrer sur l'amélioration de notre code.



Nous avons ainsi passé les semaines suivantes à améliorer la qualité de notre code et en une semaine nous avons ainsi réussi à remonter jusqu'à 67%, notre objectif initial étant de remonter à 70% durant la semaine.

Avec la précipitation nous avons réussi à remonter la couverture de tests, mais cela a généré de nouveaux bugs et ces nouveaux tests n'étaient pas tous pertinents comme nous le montrait les Pitests.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
47	69%	550/792	47%	249/534	70%	249/355

Ce manque de qualité est ce qui nous a conduit à par la suite échouer plusieurs semaines d'affilées car nous n'avions pas le temps d'à la fois corriger nos tests, améliorer la qualité du code existant et du nouveau code ainsi qu'ajouter toutes les features nécessaires pour réussir la semaine suivante.

Finalement nous avons réussi à faire remonter la qualité de notre code à notre objectif pour cette fin de projet : 80% de mutants tués, tout en réussissant à diminuer notre dette technique qui hélas contient toujours 9h liée au nom du package. Nous sommes satisfait de la qualité du code final même si nous sommes conscients que notre qualité a été variable tout au long du projet. Mais les erreurs que nous avons faites nous ont démontré l'importance d'avoir un code qualitatif et les impacts qu'un manque de qualité peut avoir sur un projet.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
44	94%	941/998	81%	577/710	84%	577/687

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
fr.unice.polytech.si3.qgl.Mugiwara_Cook	2	78%	46/59	81%	39/48	93%	39/42
fr.unice.polytech.si3.qgl.Mugiwara_Cook.actions	4	92%	34/37	85%	11/13	100%	11/11
fr.unice.polytech.si3.qgl.Mugiwara_Cook.game	3	97%	69/71	77%	36/47	78%	36/46
fr.unice.polytech.si3.qgl.Mugiwara_Cook.geometry	2	84%	21/25	100%	10/10	100%	10/10
fr.unice.polytech.si3.qgl.Mugiwara_Cook.geometry.shapes	4	91%	30/33	75%	6/8	86%	6/7
fr.unice.polytech.si3.qgl.Mugiwara_Cook.goal	3	91%	20/22	80%	4/5	100%	4/4
fr.unice.polytech.si3.qgl.Mugiwara_Cook.pathfinding	1	97%	73/75	92%	33/36	94%	33/35
fr.unice.polytech.si3.qgl.Mugiwara_Cook.pathfinding.cartography	3	99%	232/234	80%	196/244	81%	196/243
fr.unice.polytech.si3.qgl.Mugiwara_Cook.sea	6	92%	54/59	79%	11/14	100%	11/11
fr.unice.polytech.si3.qgl.Mugiwara_Cook.ship	2	94%	63/67	84%	36/43	92%	36/39
fr.unice.polytech.si3.qgl.Mugiwara_Cook.ship.equipment	3	96%	43/45	100%	18/18	100%	18/18
fr.unice.polytech.si3.qgl.Mugiwara_Cook.shipmaster	2	98%	100/102	73%	45/62	75%	45/60
fr.unice.polytech.si3.qgl.Mugiwara_Cook.shipmaster.captainNextMove	3	91%	41/45	77%	44/57	77%	44/57
fr.unice.polytech.si3.qgl.Mugiwara_Cook.shipmaster.captainNextMove.choice	2	88%	35/40	75%	18/24	78%	18/23
fr.unice.polytech.si3.qgl.Mugiwara_Cook.shipmaster.captainNextMove.possible.possibleAngle	1	93%	13/14	100%	16/16	100%	16/16
fr.unice.polytech.si3.qgl.Mugiwara_Cook.shipmaster.captainNextMove.possible.possibleDistance	2	88%	23/26	100%	17/17	100%	17/17
fr.unice.polytech.si3.qgl.Mugiwara_Cook.shipmaster.captainSailorMoves	1	100%	44/44	77%	37/48	77%	37/48

Principes SOLID

Tout au long du projet, nous avons essayé d'appliquer au mieux les principes SOLID. Ils sont en partie respectés mais nous avons malgré tout des bouts de code qui ne respectent pas certains principes. Certaines méthodes sont trop longues et peuvent contenir plusieurs responsabilités, et nous avons par moment modifié certaines classes au lieu de les étendre ce qui a induit de la duplicité de code durant le projet. Par exemple, initGame a beaucoup trop de responsabilité au vu du rôle que cette classe est censée : fournir les données de début de partie.Finalement nous nous retrouvons avec une classe, qui fait le tri parmis les marins disponible en fonction de leur rôle, ce qui n'a absolument rien à voir.

Refractors

Nous avons effectué un seul réel refactor, ce refactor n'était pas seulement pour modifier la structure mais aussi pour changer le fonctionnement de notre algorithme. En effet, juste un avant l'implémentation du gouvernail, nous avons remarqué que notre façon de déterminer le déplacement du bateau était à la fois trop coûteuse en temps mais aussi non optimisée au vu des éléments qui allaient être ajoutés plus tard telle que la voile ou le gouvernail.

Soyons plus explicite sur l'ancien fonctionnement pour comprendre la nécessité de ce refactor. Avant, nous prenions en compte toutes les possibilités de déplacement, c'est-à-dire, nous calculions la position du bateau en fonction des différentes options. Initialement, il y avait uniquement des rames, donc plus il y avait de rame et de marin plus il y avait d'options de déplacement ce qui faisait beaucoup de calcul. Cependant, si nous avions ajouté le gouvernail dans cet état, le nombre de possibilité aurait été fortement augmenté, puis l'ajout de la voile n'aurait fait qu'empirer les choses. Pour résumer, nous aurions eu un nombre de possibilités beaucoup trop importantes et le temps de calcul aurait était trop conséquent au moment d'implémenter le pathfinding, nous risquions aussi d'avoir un TIME-LIMIT lors des courses.

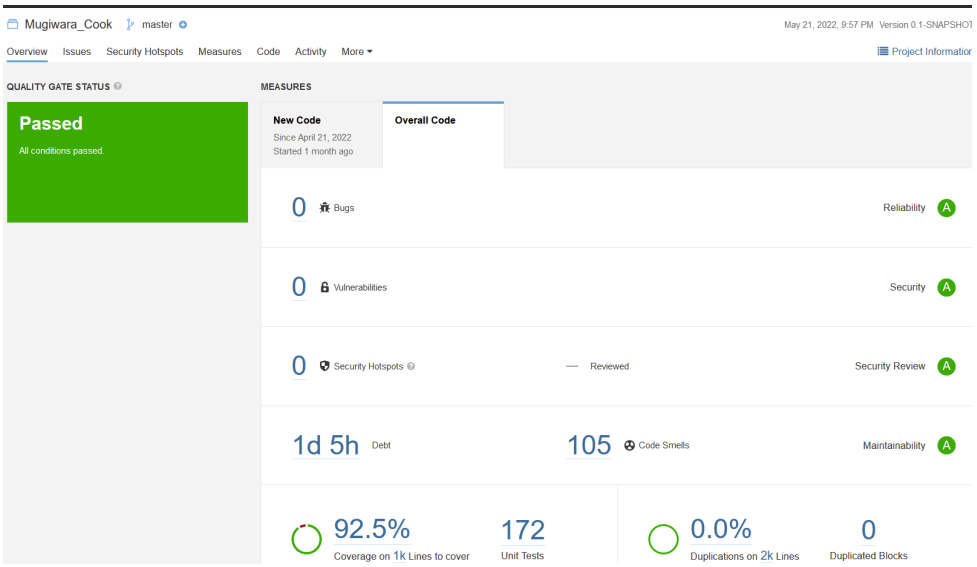
Nous avons donc préféré faire un refactor pour opter pour une méthode de déplacement qui ne calcule pas à chaque fois la position du bateau mais qui fonctionne plus à "l'instinct", comme expliqué dans la partie "Architecture" du projet. Ce refactor nous a permis d'implémenter par la suite la voile sans trop de difficulté.

Github Action

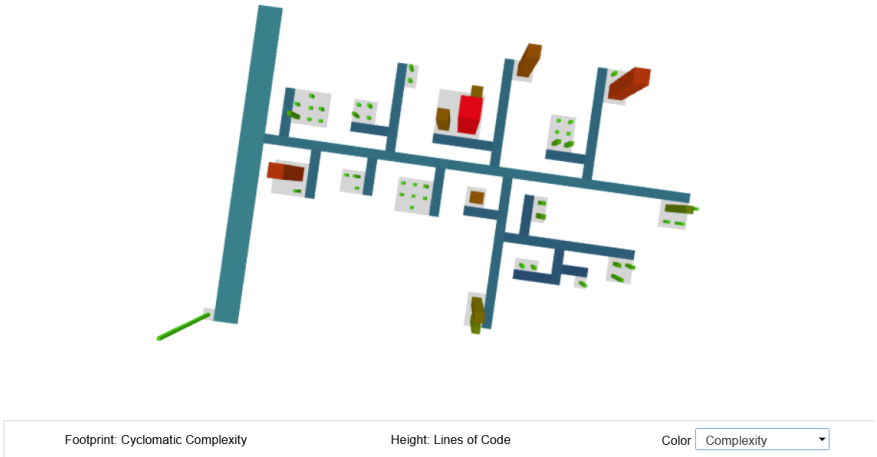
Github Action nous a permis de gagner en efficacité et en qualité, en effet au début du projet presque tous les membres du groupe avaient des ordinateurs peu performants et ne pouvait donc pas lancer les PITest sans devoir attendre environ 40 minutes. Nous avons ensuite tous eu des problèmes d'ordinateurs en même temps, ce qui a rendu Github Action particulièrement précieux car nous l'avons brièvement utilisé pour exécuter nos PITest ce qui nous a fait gagner énormément de temps.

Métriques et outils d'analyse

Dans le cadre du projet QGL que nous avons été introduit aux métriques ainsi qu'aux divers outils pour les analyser. Nous avons utilisé deux outils pour traiter les métriques de notre code : SonarQube et PITest. SonarQube est un outil qui nous a servi à visualiser la complexité de notre code ainsi qu'à déterminer précisément quelles parties du code nuisaient à la qualité de celui-ci et sur quels critères. Cet outil nous permet aussi de pouvoir mesurer la dette technique. PITest est un outil de test très performant qui nous a permis de tester extensivement notre projet y compris nos tests eux-mêmes. En effet, les métriques de suivi des tests nous ont aidé à produire des tests unitaires robustes et efficaces sur la majorité du projet qui ont couvert 92.5% des classes de code. Ces deux outils ont donc fortement contribué à l'application des concepts vus en cours puisqu'ils nous ont aidé à appliquer les principes SOLID et améliorer la qualité du code.



Néanmoins, nous devons prendre du recul sur les résultats fournis par ces outils, par exemple SonarQube permet d'évaluer la dette technique mais dans notre cas nous avons nécessairement environ 9 heures de dette technique liée au nom de notre package : "Mugiwara_Cook" car il comporte des majuscules.



Etude fonctionnelle et outillage additionnels

Stratégie pour la victoire

Pour obtenir la victoire, la stratégie que nous avons adopté un algorithme A* pour la création de la map, pour le pathfinding nous avons ajouté de très légères marges pour esquiver les récifs. Pour déterminer la trajectoire, nous avons ajouté des checkpoints dans la liste avec un attribut précisant s'ils doivent nécessairement être traversés ou s'ils sont facultatifs.

Afin d'augmenter nos chances de victoire en mode régate et bataille navale, nous aurions pu réduire le nombre de checkpoints intermédiaires qui sont créés par l'algorithme de pathfinding pour guider le bateau. En effet, un nombre important de checkpoints sont créés afin d'éviter les obstacles mais cette mécanique empêche au bateau d'aller à sa vitesse maximale car celui-ci doit s'assurer d'être dans le checkpoint à chaque fois.

Un autre problème majeur est la gestion des marins. Tous les marins sur notre bateau ont un équipement qui leur est attribué et, par conséquent, ne bougent pas. C'est évidemment un problème qui empêche l'optimisation de la course mais surtout qui empêche les marins d'utiliser les canons pour se protéger et de naviguer en même temps dans le cas d'une bataille navale.

Pour la bataille navale, nous avions plusieurs idées de stratégies : une idée de stratégie agressive et une stratégie pacifique. La première consistant à tirer sur tous les bateaux à porter afin d'être le dernier survivant, ce qui est un critère pour gagner. La stratégie pacifique consistait à fuir tous les ennemis et à tout miser sur le temps, car un autre critère pour obtenir la victoire en bataille navale est de tenir jusqu'à la fin du temps imparti, cette stratégie est plus simple et ne requiert pas l'utilisation de canon, mais elle requiert d'encore plus anticiper le déplacement des bateaux et d'être capable d'esquiver à la fois les bateaux, les récifs et les boulets de canons.

Outils utilisés

Au début du projet, nous avons créé un simulateur en python que nous avons ensuite adapté en java, ce simulateur nous permettait de tester notre code à l'avance et de savoir s'il allait passer la semaine. Dans le cas contraire, il nous aidait à déterminer l'origine de nos problèmes. Mais bien que ce simulateur nous aidait à débugger, il a contribué à la mauvaise qualité de notre code, sans ce simulateur nous aurions plus testé le code pour être certains de son fonctionnement. Nous avons arrêté de mettre à jour le simulateur lors de l'arrivée du pathfinding par faute de temps. Nous avons ensuite utilisé le WebRunner pour simuler nos parcours et nous aider tout en veillant à conserver une certaine qualité de code.

Nous avons aussi utilisé des logiciels liés à Git tels que Github Desktop, ce logiciel nous permet de visualiser facilement les derniers commits, d'en annuler et de pouvoir facilement changer de branche. Github Desktop affiche les derniers commits avec les tags associés s'il y en a, ce qui permet de vérifier rapidement si un tag a été mis. On peut aussi

s’en servir pour trouver rapidement le commit à partir duquel on veut créer une branche.

Conclusion

Ce que nous avons appris

Tout au long de ce projet, nous nous sommes appliqués à appliquer les bonnes pratiques d'écriture de code. Grâce à toutes les erreurs et corrections qui ont suivi, nous possédons désormais plus de recul et avons une compréhension plus poussée des règles à respecter pour produire du code qualitatif. Nous avons également une connaissance plus avancée du fonctionnement de Maven, du choix d'une architecture pour un projet et de comment modifier une structure . Néanmoins, nous avons aussi découvert de nouvelles notions et de nouveaux outils extrêmement importants tels que l'automatisation qui nous ont beaucoup servi pour ce projet pour accomplir diverses tâches souvent désagréables et chronophages. Nous avons aussi appris la notion de métrique : leur rôle, comment les analyser et avec quels outils afin de rendre notre code de la meilleure qualité possible. Enfin, nous avons appris ce que sont les branching strategies, nos erreurs de départ sur celles-ci nous ont permis de bien comprendre à quoi elles servent, quels sont les modèles classiques, comment s'en servir, comment créer la nôtre et pourquoi en créer une, ainsi nous sommes prêts à faire nos prochains projets sans commettre à nouveau les même erreurs.

Connaissance des autres cours

Puisque le projet a été écrit en Java, les notions abordées dans les cours de POO et de PS5 au premier semestre ont servi dans l'avancée du projet. L'introspection acquise grâce au cours d'ASD et de PCP sur la complexité des algorithmes ont également pu être exploitées au cours de ce projet, en particulier sur la partie de pathfinding, qui constitue la partie la plus complexe de notre projet.