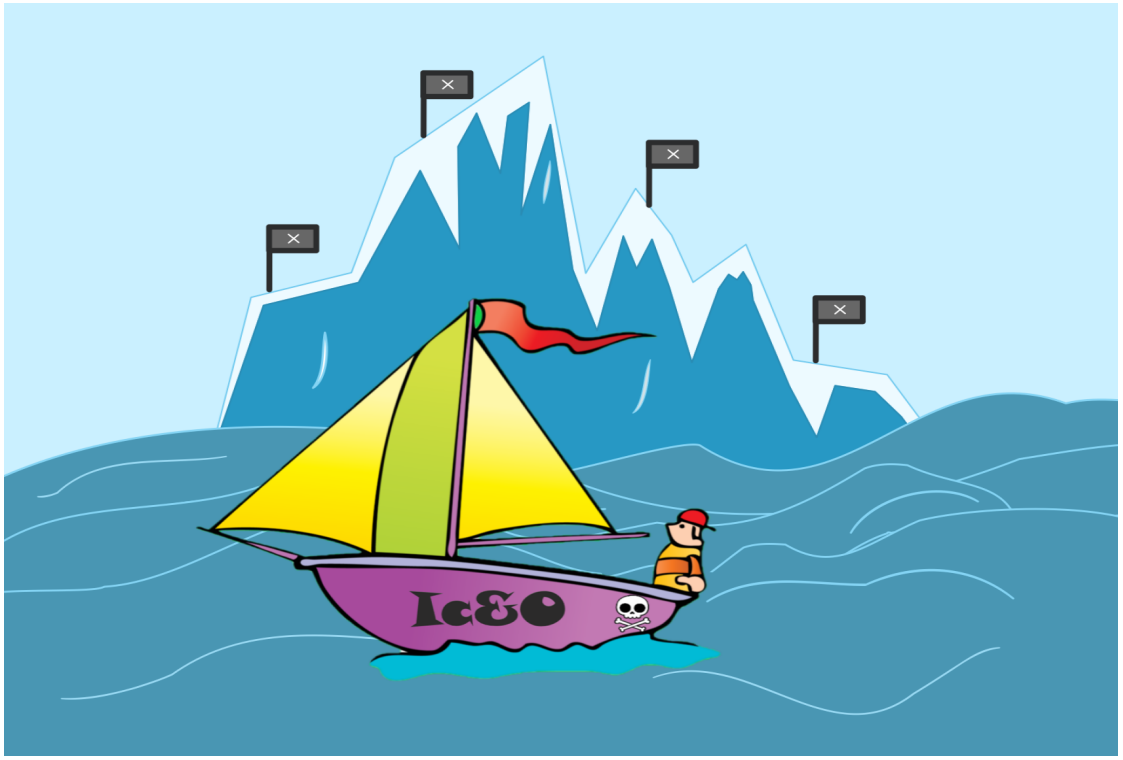
Rapport de Qualité et Génie Logiciel



Equipe AJIL

Alexis ROCHE
Jordan BENAVENT
Igor MELNYK
Louis HATTIGER
Tobias BONIFAY (matelot repêché)

Sommaire

Description technique

1.1 Architecture du projet

1.1.1 Différentes phases de notre programme dans la prise de décision

Afin de réaliser les différentes actions pour gagner la course, notre bateau passe par différentes étapes.
Lors du premier tour notre bateau est dans la phase d'initialisation. Nous utilisons un ObjectMapper afin de pouvoir créer toutes les informations de notre bateau. Une fois cette étape effectuée, nous poursuivons sur les autres tours.

Pour les autres tours, la première étape est de mettre à jour les données du jeu contenues dans la classe *Game* (les nouveaux récifs découverts, la position du bateau…) à partir du nouveau JSON reçu. Nous regardons si notre bateau intersecte un autre checkpoint, pour savoir si nous l'avons validé ou pas..
Ensuite, nous vérifions que nos marins se sont bien placés et prévoyons le déplacement des marins si besoin. La prochaine étape consiste à vérifier si un récif se trouve sur notre chemin ou non, et si oui calculer un chemin avec l'algorithme A* que nous avons implémenté pour notre pathFinding. Si on lance A*, nous récupérons une liste de checkpoints intermédiaires et si nous n'avons pas besoin de lancer le pathFinding, nous calculons tout simplement le déplacement à réaliser pour le tour actuel.

Un objet Déplacement est alors créé, contenant les informations sur le déplacement à faire. Notre système de stratégie prévoit alors si l'usage d'une voile est nécessaire, et ensuite coordonne les actions (utilisation du gouvernail ou non, quels marins doivent ramer). Toutes ses actions sont ajoutées dans une liste, qui sera ensuite convertie en JSON et transmise au simulateur.

1.1.2 Extensibilité de notre architecture

Dans le cas de l'introduction de nouveaux modes de jeu, il y aura probablement de nouvelles formes, de nouvelles actions comme FIGHT ou APPROACH (aborder un navire) pour les marins ou encore de nouvelles entités dans la mer comme les requins.
Notre architecture est facilement extensible à ces modifications, en effet nous possédons une classe mère *Shape*, *Actions*, *Entities* et *VisibleEntities* permettant de créer très facilement des classes filles correspondant aux nouvelles formes, actions, entités dans le bateau et des entités visibles dans la mer.
En ce qui concerne l'ajout de nouvelles actions nous avons simplement à créer une nouvelle classe qui extends de *Actions* pour la nouvelle action et ajouter son type (son nom) dans notre enum Actions. On procède de même pour l'ajout de nouvelles formes, entités dans le bateau et entités dans la mer.
De nouveaux modes de jeu impliquent de nouvelles stratégies. Il faut donc pouvoir changer de stratégie en fonction du mode de jeu sélectionné. Cela ne pose pas de problème dans

notre projet. En effet, nous changeons déjà notre stratégie en fonction de si notre bateau possède un gouvernail ou non mais aussi lorsqu'il y a des récifs dans la mer (on déploie AStar si des récifs sont détectés entre nous et le checkpoint). Oui notre architecture est extensible pour l'introduction d'un nouveau mode mais aussi en général.

1.1.3 Choix de conception et d'algorithme

Dans un premier temps nous avons structuré notre projet dans différents packages contenant des objets de sens similaires. Nous avons regroupé toutes les formes, les entités du bateau, les entités de la mer et de même pour les actions disponibles. Tous ces regroupements sont utiles afin de pouvoir créer une classe mère et de pouvoir respecter au maximum le critère "Open/Closed" des principes SOLID.

Durant les premières semaines nous avons créé une classe de stratégie *Strategy* permettant de prendre les décisions de déplacement de notre bateau. Par la suite, nous nous sommes aperçus à l'aide de SonarCube que cette classe possédait beaucoup de responsabilités. Nous avons ainsi divisé ses responsabilités dans différentes classes s'occupant de fonctionnalités plus précises. Nous avons donc des classes s'occupant de la validation des checkpoints, du placement des marins, du calcul du déplacement des marins et de la gestion de la voile. En application au principe du "Single Responsability" de SOLID. Nous avons également centralisé les calculs mathématiques des intersections pour les utiliser dans toutes classes nécessitant cette fonctionnalité.

Pour notre PathFinding, nous avons utilisé l'algorithme A*. Un simple évitement n'aurait pas permis de trouver la sortie de labyrinthe. Pour éviter des calculs inutiles, nous ne lançons cet algorithme que si un récif est présent entre notre bateau et le checkpoint ciblé. L'avantage du A* par rapport à Djikstra est qu'il utilise une heuristique pour accélérer et optimiser les calculs lors de la recherche du plus court chemin.

1.2 Impact des contraintes sur le projet

Lors du projet nous avons eu la contrainte de récupérer des String avec le format JSON. Nous avons dû créer des objets contenant les attributs du même nom et de même type afin de pouvoir utiliser un *ObjectMapper* et de créer tous les objets reçus dans le JSON.
L'interface *ICockpit* n'a pas été une grande contrainte pour nous. Elle a uniquement le rôle de récupérer toutes les informations qui nous sont passées en paramètre.

Nous avons eu la contrainte de créer un objet *Game* récupérant toutes les informations lors de l'initialisation du bateau et de pouvoir stocker toutes les informations. De même, nous avons créé un objet *NextRound* afin de récupérer l'ensemble des informations pour le tour et qui permet de mettre à jour notre jeu et ses données.

Applications des concepts vu en cours

2.1 Justification de notre branching strategy

Nous avons choisi Github Flow. Ce choix nous paraît être le plus adapté à notre organisation. En effet, cette stratégie permet à chaque fonctionnalité d'être développée de manière indépendante des autres, et sans être trop contraignante.
Lorsqu'une fonctionnalité est développée et testée, on *merge* la branche sur le *master*. Nous ne passons pas par des branches intermédiaires (comme en Git Flow), ceci rend cette stratégie plus simple à mettre en place. Cette simplicité permet d'être en livraison continue avec des cycles courts, sans avoir à réaliser des potentiels *merges* sur différentes branches intermédiaires.
Le seul problème a été que nous avions passé beaucoup de temps sur la branche *PathFinding*, et nous n'avons pas régulièrement *rebase* cette branche. Nous avons donc eu une *merge* assez difficile à réaliser.
Malgré cet accident, cette stratégie a très bien fonctionné, car elle était adaptée pour notre projet (petite équipe, avec une livraison par semaine).

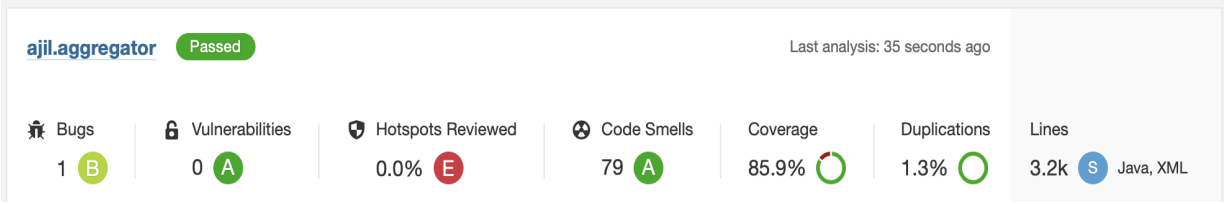
2.2 Qualité du code

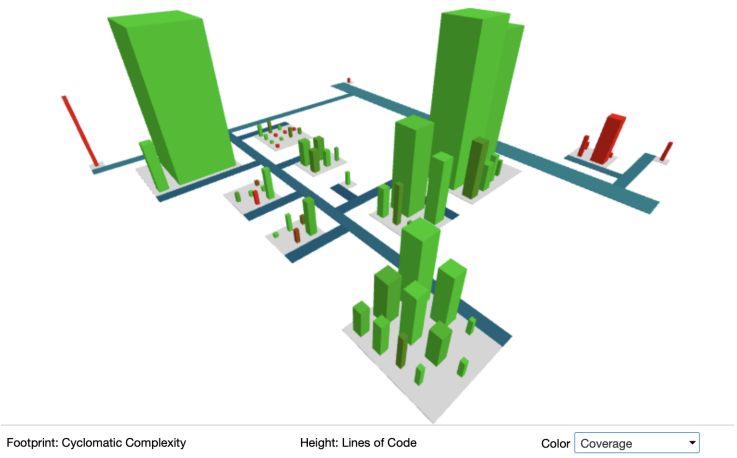
2.2.1 Auto-critique de la qualité de notre code (avec tests et métriques sonar)

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
48	91%	1428/1566	81%	1012/1256	85%	1012/1197





Notre principal défaut est d'avoir certaines classes trop complexes, notamment les classes du package Strategy, car ce sont elles le noyau de notre système de prise de décision. Avec Sonar 3D Viewer, nous pouvons observer que ces classes en question sont trop complexes et grandes, mais nous pouvons aussi apercevoir qu'elles sont bien testées. Nous avons effectué des refactors, mais avec du temps supplémentaire nous aurions voulu effectuer un autre refactor du package Strategy.

Dans certains cas, des mutants étaient très difficiles à tuer, et on ne pouvait pas réussir à les tuer. Mais nous avons réfléchi et analysé pourquoi ce n'était pas un problème sur le comportement voulu de notre code.

Nous avons pu utiliser *Sonar* et *PITest* pour améliorer la qualité de notre code au fur et à mesure du projet. *Sonar* nous a permis d'éliminer des bugs/codes smells et *PITest* nous a permis de vérifier la pertinence de nos tests, et de rajouter des tests supplémentaires dans les endroits nécessaires (comme dans certains calculs d'intersection, on nous avons rajouté des tests supplémentaires pour s'assurer de la fiabilité des méthodes).

2.2.2 Impact de la qualité du code

Un code de qualité plus faible nous aurait fait douter de la fiabilité de notre bateau, spécifiquement dans la prise de décision. Mais cela aurait aussi pu amener des erreurs à l'exécution ou alors des bugs, et il est plus difficile de trouver leur origine si le code est mal conçu. Les tests nous permettent d'éviter ça. Dans un code de mauvaise qualité, il est de plus en plus difficile d'ajouter des nouvelles fonctionnalités, et la dette technique peut s'accumuler très rapidement.

Avec une qualité plus élevée : on serait plus confiant de notre code et des actions de notre bateau. Pour le moment, on utilise les tests pour s'assurer du bon comportement et de la prise de décision. Le fait d'avoir plus de tests et avec une meilleure force aurait permis de renforcer cette assurance. Avec un code de meilleure qualité, il est plus facile d'ajouter des nouveaux éléments et fonctionnalités (en utilisant par exemple l'inversion de dépendance ou encore le principe Open/Close).

Enfin avec une meilleure qualité, le code aurait pu être plus lisible, en séparant mieux les responsabilités et en découpant mieux certaines méthodes.

2.2.3 Impact de la mesure de la qualité de notre projet sur nos livraisons et notre organisation

Le but de ce projet n'est pas de faire un code qui comprend la totalité des fonctionnalités, mais de réussir à réaliser un code propre en appliquant les concepts vus en cours, par exemple en appliquant les patrons de conception avec les bonnes abstractions. Les cours magistraux nous ont permis de comprendre et d'acquérir des notions indispensables pour garder un code de qualité, et nous nous sommes efforcés de les appliquer le plus possible.

Nous avons donc essayé d'utiliser les outils vus en cours (Sonar, PITest) mais aussi de suivre l'évolution des métriques Sonar au fur et à mesure. Le fait d'avoir une livraison chaque semaine renforçait cette contrainte de qualité, car nous savions que nous allions devoir implémenter des nouvelles fonctionnalités et donc qu'il fallait avoir un code propre pour pouvoir continuer sur une bonne voie, sous peine d'accumuler de la dette technique.

2.3 Refactoring globaux

2.3.1 Nombre de refactoring globaux et justification de ces refactoring

Nous avons effectué plusieurs refactor afin d'adapter notre code aux nouvelles fonctionnalités à ajouter. Nous avons modifié notre architecture de la stratégie que nous détaillons juste après.
De plus, nous avons centralisé le calcul d'intersection entre les différentes shape. Ces méthodes étaient uniquement utilisées dans ValidationCheckpoint mais nous en avions besoin dans différentes classes. Ainsi ce refactor nous a permis d'avoir une seule classe effectuant ces calculs. Un package Maths est entièrement dédié pour les calculs utilitaires.

Nous avons également dû effectuer un refactor sur notre package s'occupant de la gestion du pathFinding. Au départ, nous ne savions pas combien de classes et de lignes seraient nécessaires pour obtenir un A* fonctionnel, et dès que nous avons obtenu un programme testé et efficace, nous avons amélioré la qualité du code et l'architecture de cette partie.

2.3.2 Description d'un refactoring effectué dans le détail

L'un des premiers *refactor*s que nous avons faits concerne la classe *Strategy*. Au tout début du projet, nous avions une classe Dieu, qui gérait à la fois les placements des marins et les déplacements du bateau. Nous avons alors découpé cette classe en plusieurs : une classe centrale *Strategy*, qui coordonne d'autres classes (*GestionMarin*, *CalculDéplacement*…).

Le problème majeur est la communication d'informations entre les différentes classes coopérantes sur la stratégie. Nous avons créé une classe *Data*, qui a pour responsabilité de stocker les données du jeu et les informations nécessaires pour la mise en œuvre de la stratégie. Les autres classes utilisent les données de Data pour effectuer les calculs.

Avec plus de temps, on aurait voulu refaire un refactor de ces classes, comme expliqué dans la partie auto-critique de la qualité de notre projet.

2.4 Automatisation

Comment l'automatisation d'une partie des tâches de votre projet vous a-t-elle permis de gagner en efficacité et en qualité ?

L'automatisation nous a permis de gagner en efficacité en réalisant des actions répétitives automatiquement (comme le build, le lancement des tests ou encore la création d'un jar avec Maven). Ceci nous permet de gagner du temps et d'éviter d'oublier des actions à faire.

Pour gagner en qualité, nous avons décidé d'utiliser Github Actions pour lancer PITest lors de chaque Tag. Les résultats sont publiés sur une autre branche à part. Nous avons pu avoir un suivi détaillé de la force de nos tests, et cela nous a permis de gagner du temps avec un PITest global pour tous les membres du groupe. On pouvait donc facilement faire un débrief et se mettre d'accord sur les parties où il manque des tests.

Etude fonctionnelle et outillage additionnels

3.1 Comment améliorer nos chances de victoire

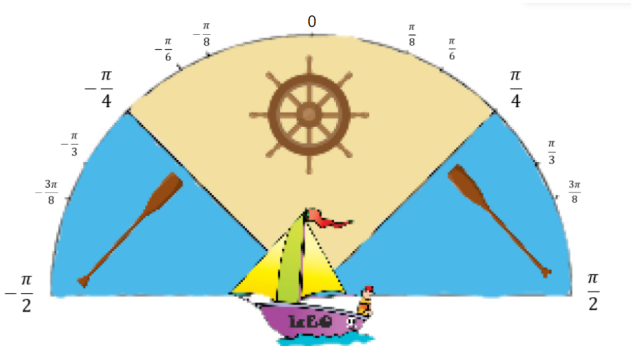
Objectif de la stratégie: Finir une course le plus rapidement possible en passant par divers checkpoints et en évitant les récifs.

3.1.1 Stratégie implémenté et justification des choix

Stratégie sans le gouvernail (correspondant aux premières semaines):

Tout d'abord nous savions que nous étions plus rapides quand on avançait tout droit (vitesse = 165). Nous avons ainsi privilégié les lignes droites. Pour ce faire, nous ordonnons au bateau de tourner seulement lorsque l'angle entre le bateau et le checkpoint correspond à un angle de rotation possible à effectuer. Par exemple, pour un bateau avec 4 rames, on tourne seulement lorsque l'angle est égal ou supérieur en valeur absolue à $\pi/4$ ou $\pi/2$. Grâce à cette stratégie, on garantit une vitesse optimale tout le long de la course.

Stratégie avec le gouvernail (à partir de la WEEK 4):



Lors de l'implémentation du gouvernail nous avons décidé de continuer notre stratégie des premières semaines à l'exception que lorsque le bateau a un angle inférieur ou égal à $\pi/4$ avec le checkpoint nous utilisons le gouvernail. L'avantage du gouvernail est qu'il permet d'effectuer un angle très précis entre -$\pi/4$ et $\pi/4$ tout en ramant à la vitesse maximale égale à 165. Tout comme la stratégie des premières semaines nous garantissons une vitesse maximale tout au long de la course ce qui nous permet de marquer un maximum de points chaque semaine.

Stratégie avec la voile (à partir de la WEEK 5):

Pour la stratégie de la voile, elle consiste à hisser celle-ci uniquement lorsque le vent est dans le même alignement que l'orientation du bateau.
On considère l'alignement "intéressant" lorsque le navire est dans l'intervalle]$-\ \pi/2 : \pi\ /\ 2$ [.

Stratégie de validation du checkpoint :

On s'est aperçu que nous visions tout le temps le centre des checkpoints ce qui nous faisait perdre du temps car on devait parcourir plus de distance. On a donc fait en sorte d'optimiser cette visée de checkpoint. Au lieu du centre, on vise à présent l'extrémité d'un checkpoint. Cette extrémité correspond à un point. Avec ce point on a créé un faux checkpoint à viser à la place de l'original. Ce faux checkpoint est situé en fonction de la position du checkpoint suivant à valider. Ainsi, lorsqu'un checkpoint est validé nous n'avons qu'à avancer tout droit ou modifier très légèrement notre trajectoire. C'est un véritable gain de temps car on a moins de distance à parcourir et nous sommes presque tout le temps à vitesse maximale.

Stratégie avec le pathfinding (lorsqu'il y a des récifs sur la carte):

Nous traçons un segment entre notre bateau et le checkpoint visé. S'il intercepte un récif, alors on lance le calcul de l'Algorithme A*. Dès que le bateau évite l'obstacle qui a enclenché le calcul de A*, on supprime tous les checkpoints intermédiaires s'il n'y a plus d'obstacles sur la route. Nous utilisons la vigie pour s'assurer de réussir à trouver un chemin même dans un labyrinthe.

Stratégie bataille navale : Nous n'avons pas implémenté l'utilisation des canons ni la collision possible entre les différents bateaux. Cependant, nous sommes **la seule et unique** équipe **survivante** de la **WEEK 13** 😄 !

3.1.2 Stratégie planifiées non implémentés

Pour le moment nous avons implémenté des stratégies qui ne prennent en compte que notre bateau ou des éléments inertes comme les récifs ou les courants.

Pour le mode de jeu "bataille navale" on ne sait pas encore comment on voudrait procéder en ce qui concerne la prise en compte des bateaux ennemis. D'une part, on hésite à partir du principe que les bateaux ennemis nous prendront en compte. Auquel cas on pourrait faire notre course comme s'il n'y avait aucun bateau en jeu et ne pas perdre de temps à éviter les autres bateaux. D'autre part, on pourrait prendre en compte les bateaux dans notre pathfinding en tant que récif. Comme les bateaux se déplacent aussi dans la mer on pourrait ajouter une zone de potentiel déplacement de ces derniers. Ainsi on aurait de plus gros récifs comprenant le bateau mais aussi leur potentiel déplacement durant le tour.

Dans la bataille navale il y a bataille, qui dit bataille dit canon. Dans notre code nous avons déjà implémenté l'entité de canon ainsi que les actions AIM, FIRE et RELOAD. En revanche, nous ne les utilisons pas. Nous voulions implémenter une stratégie d'utilisation des canons utile lors des batailles navales. Pour ce faire il faut qu'on prenne en compte les bateaux ennemis mais aussi qu'on attribue les marins aux canons lorsqu'un tir est prévu. Pour viser on devra calculer l'angle entre notre bateau et le bateau à viser.

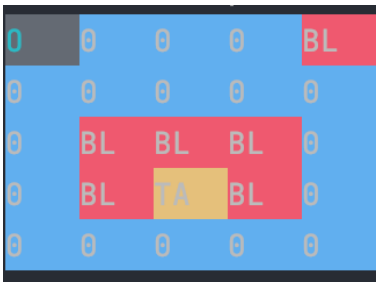
Nous aurions aussi voulu lisser les checkpoints après le calcul de A*, mais notre trajectoire reste assez droite dans l'ensemble donc ce n'était pas une priorité pour nous de l'implémenter.

3.2 Utilisations d'autre outils

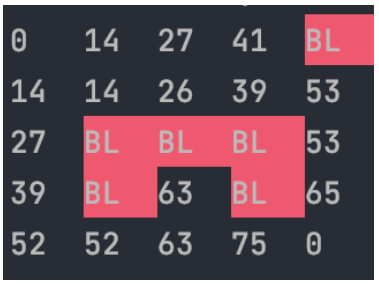
3.2.1 Outils créés (on peut parler du display dans A Star ?)

Lors du calcul de notre recherche de chemin (classe *pathFinding*) avec A*, nous souhaitions vérifier que l'algorithme en tant que tel fonctionne correctement. Dans tooling, nous avons créé une classe Visualisation A*. Cette classe permet d'afficher la grille produite par l'algorithme A*. Cette grille met en évidence le carré de départ, celui d'arrivée, les cases bloquées et le schéma tracé. Nous utilisons des couleurs pour représenter ça, et nous affichons aussi une autre grille montrant le coût de passage par chaque case.

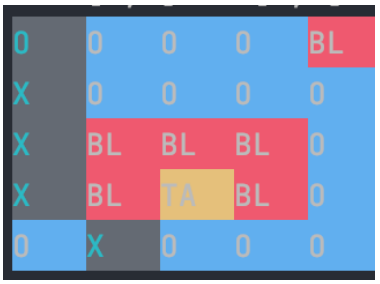
Map



Score



Chemin



3.2.1 Outils que nous aurions pu développer

Nous aurions pu développer un petit simulateur comme le WebRunner que nous utilisions pour visualiser notre avancée semaine après semaine. Cela nous aurait permis de tester rapidement et gratuitement notre code sur certains points à chaque course (le gouvernail, les courants…). Nous avions pensé à développer un outil pour repérer et mettre en évidence les déplacements des marins, pour vérifier leur placement. Mais ce projet a été mis de côté par manque de temps.

Autres idées envisagées :

- Un simulateur pour pouvoir tester nos courses gratuitement
- Un système pour hacker le web runner pour plus de crédits et un meilleur classement dans le championnat.
- Une interface graphique très simple représentant la manière dont le bateau voit les obstacles (pour débugger et éviter des erreurs dans les calculs de position)

Conclusion

Qu'avons nous appris lors de ce projet ?

Diverses compétences ont été apprises durant ce projet. Tout d'abord, la manière d'évaluer, d'augmenter et contrôler la qualité du code. L'utilisation de Sonar nous permet de suivre des indicateurs et d'obtenir des conseils pour augmenter la qualité, en découpant les Classes trop complexes ou en évitant des "codes smells". Nous avons également découvert un peu mieux Maven, en gérant les différents profils et en comprenant mieux les fameux fichiers "pom.xml".
D'autres compétences ont été acquises, comme l'utilisation des branches sur Github, les rebases, et l'automatisation avec Github Actions. De plus, nous avons appris l'importance de faire des tests pertinents, et comment évaluer la solidité de nos tests à l'aide de PITest. Enfin nous avons gagné de l'expérience avec les concepts SOLID et des bonnes pratiques de Génie Logiciel.

Quelles connaissances venant d'autres cours ont pu être exploitées sur ce projet ?

Le cours de PS5 nous a permis d'acquérir les méthodes essentielles pour mener à bien un projet informatique (utilisation de GitHub, qualité de code, responsabilités et principe de génie logiciel, découpages et Kanban…). Les compétences Java du cours du POO ont été indispensables pour développer en Java.

Des connaissances en ASD nous ont guidés dans la réalisation du PathFinding. Nous avons exploité des notions vues en cours, comme les graphes, les calculs de complexité ou encore les files de priorité afin d'implémenter notre Algorithme A*.

Quelles leçons pouvons-nous tirer de ce projet ?

Nous finissions ce projet en retenant un principe : La qualité de code est primordiale. Des indicateurs existent pour l'évaluer et nous devons chercher continuellement à la suivre et l'améliorer pour la postérité d'un projet. Les tests sont aussi un élément-clé, il ne suffit pas de couvrir l'intégralité du code de test, mais il faut aussi s'assurer de leur pertinence pour obtenir des résultats solides et concrets. Enfin, nous avons découvert un concept magique : l'automatisation. Cela permet d'éviter de répéter des tâches, donc de gagner du temps mais aussi d'éviter des potentielles erreurs. Nous sommes persuadés que ce cours nous a donné les principes clés pour bien réussir un projet, et ils nous seront d'une grande aide pour nos futurs projets.