Rapport de Qualité et Génie Logiciel

La famille royale de l'IHM
Equipe KIHM

Clervie CAUSER
Aurélia CHABANIER
Théo JEANNES
Matis HERRMANN

Année 2021-2022
Polytech Nice-Sophia
Sciences Informatique 3ème année

Sommaire

Description technique

Prise de décision

Notre prise de décision est séparée en quatre phases: calcul du cap, calcul de l'équipement nécessaire pour l'atteindre, déplacement des marins sur ces équipements, et utilisation des équipements sur lesquels se trouvent un marin afin de créer la liste des actions Json à retourner au *ICockpit*.

Étape 1 : Calcul du cap

Tout d'abord, nous visons le premier checkpoint dans notre liste de checkpoints. De ce checkpoint nous calculons l'angle qu'il y a entre de notre bateau et le point le plus proche appartenant au checkpoint, ce qui nous permet de trouver le chemin le plus court. Notre implémentation ne demande qu'un bateau et un checkpoint, peu importe sa forme, pour respecter au mieux les spécifications techniques.

Nous avons une fonctionnalité supplémentaire afin de gagner jusqu'à deux tours comparé aux autres équipes. Il s'agit du fait que s'il y a un autre checkpoint qui se trouve dans la liste des checkpoints, nous essayons de le viser à la place du premier checkpoint, à condition de finir quand même le tour dans le premier checkpoint.

Pour finir, nous modifions l'angle du cap en fonction de notre algorithme d'évitement simple. Il part de l'angle visé, vérifie si nous allons nous trouver en collision avec un obstacle, et si c'est le cas il itère avec un angle de 1° une fois à droite puis une fois à gauche jusqu'à trouver le premier angle où nous ne serions plus en collision. Nous ne prenons pas en compte les angles compris entre -135° et 135° car il s'agit d'angles que nous ne pouvons pas atteindre comme nous sommes limités à 90° avec les rames et 45° avec le gouvernail.

Étape 2 : Calcul de l'équipement nécessaire

Notre stratégie pour choisir l'équipement nécessaire s'agit de toujours faire le déplacement qui nous fera avancer le plus vite possible durant le tour, et qui en même temps permet de s'aligner sur le cap demandé.

En utilisant en entrée l'angle que nous devons viser ainsi que le nombre de marins qui nous sont disponibles, nous calculons combien de personnes nous devons assigner sur les rames à droite et à gauche, le gouvernail, les voiles et la vigie.

Nous assignons en priorité les rames et l'angle du gouvernail avec la meilleure combinaison possible pour avancer le plus possible. Ensuite nous décidons de si l'ouverture des voiles doit se faire ou non, à condition que le vent soit favorable (entre -90° et 90°) au début du tour, mais aussi favorable avec l'alignement que nous projetons d'avoir à la fin du tour. Nous considérons les voiles plus

importantes que les rames car nous sommes peu susceptibles de changer d'orientation à tous les tours, donc nous préférons d'abord ouvrir les voiles et moins ramer, tout ceci en fonction du nombre de marins que nous avons à notre disposition. Le tour d'après nous aurons déjà les voiles d'ouvertes et le cap aligné, et nous n'avons plus qu'à ramer le plus possible. Puis nous libérons l'assignation de certains marins deux par deux pour garder le même angle avec les rames et pour satisfaire notre nouvelle contrainte sur les voiles. Et pour finir, dans le cas où après toutes ces étapes nous aurions au moins un marin disponible, nous l'assignons à la vigie. Toutes ces assignations sont des nombres qui correspondent au nombre d'équipement de ce type qu'il nous faut, sauf le gouvernail qui est directement un angle et la vigie qui est un booléen. Ces nombres sont stockés dans notre classe *Layout.*

Étape 3 : Déplacement des marins sur ces équipements

Concernant le déplacement des marins, avec la liste des assignations des équipements, nous plaçons les marins sur les équipements nécessaires à l'aide de notre algorithme de placement de marins.

Notre algorithme est capable de prioriser les marins ayant le moins de choix de déplacement vers un équipement que nous souhaiterions utiliser (car chaque marin a un nombre de cases de déplacement limité à 5), et ceux sur les équipements à proximité des types d'équipement encore disponibles utilisant notre classe *Layout*.

Quand nous déplaçons un marin sur un équipement, le nombre associé au type de l'équipement choisi est ainsi réduit de un et nous cherchons le nouveau marin à prioriser puisque l'équipement choisi peut faire partie des choix de déplacement d'autres marins. À chaque déplacement de marin, nous créons une action Json MOVEMENT décrite dans la documentation que nous passerons à la fin de l'algorithme à la phase suivante. À la fin de l'algorithme tous les marins qui n'ont pas eu besoin de se déplacer sont assignés à la case du bateau se trouvant au milieu pour qu'au tour suivant les marins se trouvent à une distance potentielle plus proche de l'ensemble des équipements.

Étape 4 : Utilisation des équipements

La dernière phase de notre programme correspond à l'utilisation des équipements afin de créer la liste des actions au format Json. En récupérant la liste des actions de mouvement de la phase précédente, nous rajoutons les actions d'utilisation d'équipements pour chaque marin se trouvant sur un équipement, à condition que notre classe *Layout* l'autorise.

Vous trouverez en annexe le diagramme UML de notre projet qui montre le lien entre toutes nos classes (Annexe 1).

Extension du projet

Notre projet est développé selon une architecture extensible, en isolant au maximum les parties fonctionnelles, pour limiter le couplage. Un nouveau mode de jeu pourrait être facilement implémenté, en utilisant les parties de codes adaptées. Cependant cela est vrai jusqu'à une certaine mesure. En effet, la classe qui gère les prises de décisions pour une régate n'a pas assez d'abstractions. Ajouter une interface à cette classe pour choisir l'implémentation selon le mode de jeu, puis séparer les fonctions pour avoir une classe par mode de jeu nous permettrait de perfectionner la possibilité d'étendre notre code. En dehors de ce point, changer les spécifications techniques pour modifier une valeur, ajouter ou supprimer un équipement ou un type d'obstacles sera très facile à faire, grâce aux abstractions.

Choix d'architectures

Nous avons d'abord découpé nos classes pour correspondre à des blocs logiques, comme un équipement ou un marin par exemple. Ceci nous a permis de créer des héritages, ce qui favorise l'abstraction pour les entités plus précises. Par exemple, les classes *Rudder* et *Sail* découlent de la classe *Equipment*. Nous nous sommes donc rendus compte que certaines classes, comme *Equipment*, peuvent être abstraites, puisqu'elles ne servent qu'à représenter une abstraction.

Nous nous sommes ensuite assurés que chaque classe ne répondait qu'à une seule responsabilité, en séparant les classes qui devaient l'être. Un autre point important de notre architecture concerne le traitement du fichier JSON. En effet, les fonctions concernant le traitement des informations d'entrées mais également du retour à la fin du tour ont été découpées et répartis dans des sérialiseurs.

Nous avons essayé de faire correspondre au maximum les objets et notamment les noms des attributs, aux noms utilisés dans le fichier d'entrée. C'est pour cela que notre projet est entièrement en anglais. Pour faire correspondre notre application aux spécifications techniques sur les objets plus complexes, nous avons utilisé des serialiseurs que nous avons créés, notamment pour fonctionner avec les positions et les formes, pour lesquelles il faut transformer les tableaux JSON en objets Java. Pour les formes, nous avons une classe dédiée qui permet de convertir les nœuds en formes en convertissant les formes en Polygone ou en Ellipse, en adaptant les positions et les orientations pour faire correspondre les données reçues aux formes nécessaires pour que le bateau puisse naviguer dans l'océan.

Application des concepts vu en cours

Branching Strategy

Figure 1 : branches actuelles

Au début de notre projet, nous n'avions pas de branching strategy. Nous mettions tous nos commits sur la branche principale et nous nous assurions que nos commits soient fonctionnels et testés avant de les envoyer sur le répertoire. Dès la deuxième semaine , nous avons décidé de changer de stratégie et d'adopter la stratégie Github Flow. Nous avons donc créé une branche pour chaque nouvelle fonctionnalité, que nous pouvons fusionner avec la branche principale lorsque la fonctionnalité est terminée. Chaque branche s'appelle Feature-[NomFeature]-#N°Issue. Cela nous a permis de garder la branche master stable, mais aussi d'être plusieurs à travailler sur la même fonctionnalité pour pouvoir partager notre avancée, fonctionnelle ou non. L'un des avantages avec une branche principale stable est de pouvoir tagger facilement la dernière avancée dans celle-ci pour le rendu hebdomadaire.

Lorsque nous finissions une nouvelle fonctionnalité, si les changement induits étaient nombreux, nous faisions des pull-requests pour soumettre notre travail au reste du groupe. Cela permet d'une part de vérifier que le code est cohérent, et d'autre part de s'assurer qu'il n'y aura pas de conflit avec une fonctionnalité développée en parallèle. Lorsque la branche était finalement fusionnée dans la branche principale, nous récupérions son contenu pour le mettre sur notre branche en développement. Cela évitait d'avoir trop de disparités avec la master quand nous codions une fonctionnalité en parallèle.

Mesures et qualité du code

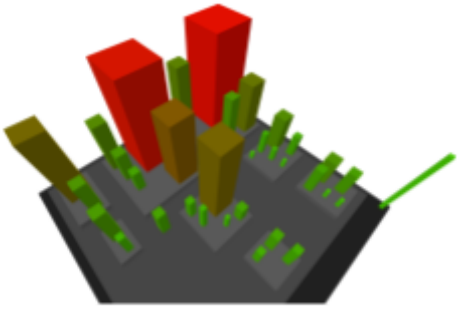
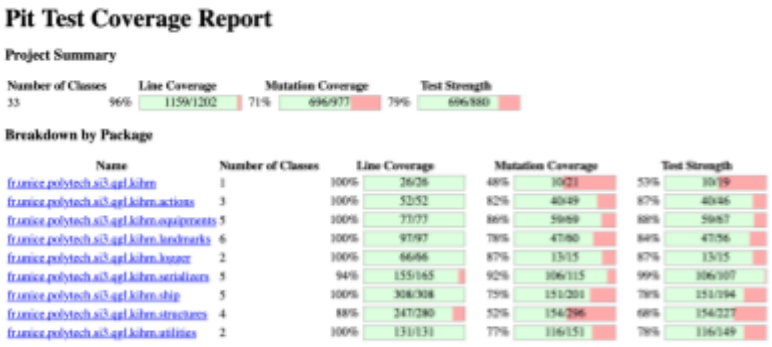


Figure 2 : Complexité et couverture de nos classes

En s'appuyant sur SONAR et sur PITest, nous pouvons mesurer concrètement la qualité de notre code. Il est possible de mesurer dans un premier temps, le nombre de bugs et mauvaises pratiques dans notre code (Annexe 8). A ce niveau là, les seuls problèmes remontés sont des tâches à faire, ce qui nous donne donc une bonne qualité du code. Un autre indicateur important à vérifier pour la qualité de notre code est la complexité (Figure 2 et Annexe 3). En effet, une complexité élevée, tant cyclomatique que cognitive, est synonyme de code difficile à tester et à maintenir. Nous avons donc maintenu la complexité cyclomatique des méthodes en dessous de 15, valeur communément admise comme une limite acceptable, afin de pouvoir facilement faire des tests unitaires. Nous avons également limité la complexité des classes, et séparé les responsabilités pour éviter les classes-dieu. Le graphique

ci-contre représente nos classes, avec la complexité en hauteur, et la couleur représente la couverture de test. Nous pouvons donc voir que toutes les classes sont bien couvertes par des tests, et qu'il n'y a pas de classe se démarquant du reste en termes de complexité. Un groupe de quatre classes est tout de même légèrement supérieur au reste en termes de complexité. Ces classes sont en fait des classes contenant beaucoup de calculs mathématiques. Par exemple, *ShapeFunctions* qui s'occupe de la transformation des formes entre le fichier JSON et les objets, contient notamment des fonctions pour déplacer les sommets des polygones en fonction de l'orientation et de la position, ce qui implique des produits matriciels entre les coordonnées de chaque point et la position, et une rotation par rapport à l'origine. Le pathfinding, comme il utilise un algorithme A-Star, est intrinsèquement complexe mais inclut des fonctions pour convertir les entrées et les sorties liés à l'algorithme qui ajoute encore de la complexité à la classe.

Le dernier élément pour mesurer la qualité du code, est la couverture de test mais également la qualité des tests associés (Figure 3 et Annexe 4). Notre code est bien couvert par les tests, à 96 % pour être exact. Les parties non couvertes par les tests, sont d'une part le cockpit, car nous n'avons pas modifié le code fourni, et des accesseurs et mutations, ainsi que certains hashCode, constructeurs et toString. Nous avons considéré qu'il n'était pas nécessaire de tester ces parties car leur logique est simple, et consiste simplement à initialiser ou à déclarer des variables et attributs ou à afficher une chaîne de caractères avec les attributs.



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
33	96% 1159/1202	71% 696/977	79% 696/880

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
fr.unice.polytech.si3.qgl.kibm	1	100% 26/26	48% 10/21	53% 10/19
fr.unice.polytech.si3.qgl.kibm.actions	3	100% 52/52	82% 40/49	87% 40/46
fr.unice.polytech.si3.qgl.kibm.equipments	5	100% 77/77	86% 59/69	86% 59/67
fr.unice.polytech.si3.qgl.kibm.landmarks	6	100% 97/97	78% 47/60	84% 47/56
fr.unice.polytech.si3.qgl.kibm.kuiper	2	100% 66/66	87% 13/15	87% 13/15
fr.unice.polytech.si3.qgl.kibm.serialisers	5	94% 155/165	92% 106/115	99% 106/107
fr.unice.polytech.si3.qgl.kibm.ship	5	100% 308/308	75% 151/201	78% 151/194
fr.unice.polytech.si3.qgl.kibm.structures	4	88% 247/280	52% 154/296	68% 154/227
fr.unice.polytech.si3.qgl.kibm.utilities	2	100% 131/131	77% 116/151	78% 116/149

Figure 3 : Rapport de résistance aux mutations généré par PIT

Au-delà de la couverture de tests, la spécificité de ceux-ci est importante. Pour mesurer cette spécificité, et s'assurer que les tests permettent bien de tester uniquement les cas attendus, nous pouvons utiliser Pitest pour générer un rapport. En utilisant ce rapport, nous pouvons constater quelles classes sont sensibles aux mutations, et donc sont mal testées. Nous pouvons constater que nos classes résistent bien aux mutations, avec 79 % de couverture. Cela nous garantit donc une bonne qualité de code, et surtout que le code est correctement testé. La partie qui ne résiste pas aux mutations comprend notamment le *Cockpit*, fourni et donc peu testé, et le package *Structures* contenant notamment le *Pathfinder*. Ce sont des méthodes qui sont un peu plus compliquées à tester puisque c'est notre classe pour notre évitement. Néanmoins, la proportion non testée est acceptable.

Avec une qualité moindre, nous n'aurions aucune garantie que notre programme fonctionne comme attendu. Certains problèmes pourraient également ne pas être remarqués jusqu'à un cas-limite, ou encore lorsqu'une nouvelle fonctionnalité utilisera le bout de code pas, ou peu, testé. L'extension et la modification du code serait également plus compliqué, puisqu'il faudrait par exemple

d'abord réduire le couplage de certains objets avant de les réutiliser ou encore découper des fonctions pour les tester. Globalement, une mauvaise qualité compliquerait le développement du projet, l'intégration de nouvelles fonctionnalités et le test de celles existantes. Une meilleure qualité nous permettrait quant à elle d'intégrer encore plus facilement de nouvelles fonctionnalités, ou de modifier le fonctionnement de certaines fonctionnalités sans provoquer d'effets de bord.

Organisation et Refactor du projet

Les premières semaines nous repoussions la réalisation des tests pour nous concentrer sur l'aspect fonctionnel. Une fois que le démarrage du projet était passé, nous nous sommes concentrés sur les tests afin de faire monter la couverture de test de notre projet. Durant la phase de production d'un produit minimum viable (MVP), nous essayons de garder notre couverture de tests au-dessus de 75%, ce que nous trouvions raisonnable.

Nous avons remanié le code à deux occasions. La première fois, nous avions refactorer le code pour retirer le simulateur que nous avons créé de la partie *Player* pour le transférer dans *Tooling*. A cette occasion, nous avons donc séparé les responsabilités du simulateur entre les méthodes utiles au fonctionnement du système, que nous avons dans une classe *Calculator*, et les méthodes destinées à simuler une partie, maintenant dans *Tooling.* Le second refactor global effectué concerne la classe *Layout* qui permet de renvoyer la liste des équipements à utiliser. Pendant la majeure partie du projet, ces informations étaient simplement stockées dans un dictionnaire clé-valeur, représenté par une *Map*. Cette structure étant fréquemment nécessaire, nous avons donc décidé de remplacer cette *Map* par une nouvelle classe, à part entière, ce qui permet de l'utiliser à de multiples endroits sans avoir à tout initialisé à chaque fois.

Automatisation du projet

Pour libérer du temps et nous permettre de nous concentrer sur le développement, nous avons mis en place plusieurs actions grâce à l'outil GitHub Action. Outre le temps libéré, cela permet aussi d'éviter les erreurs ou les oublis liés aux actions humaines.

La première action est la vérification du code soumis. A chaque commit, le code est compilé et testé, puis envoie un mail en cas d'échec d'un test minimum, ou si le code ne compile pas. Nous nous sommes ensuite concentré sur l'automatisation autour des pull-requests. Nous avons d'abord décidé d'envoyer un message discord automatiquement aux personnes assignées à une revue de code. Ensuite, dans l'objectif de gagner du temps, et en accord avec notre branching strategy, lorsque la moitié du groupe valide un commit, celui-ci est fusionné avec la branche principale puis sa branche d'origine est fermée. Le second point sur lequel notre attention s'est portée est le rendu mensuel. Pour éviter les tâches récurrentes liées au rendu, nous les avons automatisés. Lors d'un tag, en plus des tests unitaires, nous générons un rapport PIT que nous stockons dans l'artefact. De plus, un message est envoyé avec le résultat, à la fois sur le canal Slack de notre groupe et sur discord.

Etude fonctionnelle et outillage additionnel

Stratégie de victoire

Notre toute première stratégie était de ramer un maximum pour être le plus rapide possible et arriver en premier au checkpoint. Actuellement, nous avons choisi de ralentir lorsque nous nous rapprochons du checkpoint pour être sûr de ne pas nous arrêter trop loin à la fin du tour.

Quand on nous a demandé d'ajouter par exemple le gouvernail après avoir implémenter les rames, nous avons fait une méthode très simple où nous utilisions les rames d'abord et nous corrigions notre trajectoire à l'aide du gouvernail. Par la suite, nous avons décidé de prioriser certains équipements par rapport à d'autres. Nos marins vont d'abord se placer sur le gouvernail si l'angle nécessaire pour atteindre le checkpoint est supérieur à Pi/4. Nous plaçons ensuite des marins sur les rames avec le nombre de rames maximum demandé. Avec les marins qui restent, nous allons les placer sur les voiles et finalement sur la vigie.

Pour chaque équipement nous vérifions si l'équipement existe vraiment. Si il n'existe pas nous l'ignorons et nous passons donc aux autres équipements. Cette rétrocompatibilité avec toutes les semaines a été faite dès la semaine 4.

Les marins se répartissent sur les équipements les plus proches d'eux. S'ils ne sont pas utilisés pendant ce tour, ils se mettaient au milieu du bateau pour permettre au prochain tour de pouvoir accéder au plus d'équipements possibles.

Concernant l'évitement, plutôt que d'implémenter un évitement très simple qui consisterait à voir un obstacle, prendre une direction abstraite et avancer jusqu'à ce qu'il n'y ait plus d'obstacles, nous avons une "antenne" fictive au bout du bateau (Figure 4 et Annexe 5). Si au tour suivant cette antenne va toucher un obstacle, alors nous recalculons l'angle nécessaire pour atteindre le checkpoint. Cet angle va alterner entre (angle + i°) et (angle - i°) avec i = 1 au départ, et qui augmente de 1 tant que le bateau entre en collision avec l'obstacle.

Figure 4 : Schéma probleme évitement simple - GeoGebra

Pour choisir la direction que le bateau doit emprunter, nous avons par la suite essayé d'ajouter un algorithme A-Star pour renvoyer le chemin sous forme d'une liste de checkpoints au bateau. Pour déterminer le meilleur chemin, le plus rapide dans ce cas précis, nous devons calculer les coûts et les heuristiques de chaque case. Pour cela, l'unité pour mesurer le coût et l'heuristique doit être la même. Dans notre cas, nous

avons choisi la distance entre le bateau et le checkpoint suivant. L'heuristique de chaque case est donc sa distance par rapport au checkpoint. Son coût est un peu plus compliqué à calculer. Pour une case avec une intersection avec un obstacle, le coût est considéré comme infini, sinon il vaut la distance entre le point précédent du graphique et la case en question.

Cette fonctionnalité ne marchant pas tout à fait, nous avons préféré la garder sur sa branche. Vous pouvez voir ce que nous avons fait en allant sur notre branche appelé Feature-PathFinding-#63. Une des difficultés majeures que nous avons rencontrées était d'implémenter l'orientation du bateau à chaque case, ce qui nous aurait permis de calculer des coûts plus réalistes en prenant mieux en compte les courants marins et le vent. Les raisons pour lesquelles nous n'avons pas réussi à faire fonctionner notre A-Star avec notre projet, déjà bien avancé, sont que nous nous sommes pris un peu tard, ou bien le manque d'abstraction dans notre code.

Notre stratégie d'évitement fonctionne pour toutes les courses en principe, mais nous avons rencontré un problème lors de la course de la semaine 9. À partir du tour 11, notre bateau ne contournait plus les obstacles. Effectivement nous avions l'impression que le bateau cessait de vouloir éviter les obstacles alors qu'il commençait à le faire avec le premier obstacle.

Nous avons analysé les logs et comparé nos résultats, mais nous ne savions pas pourquoi nous ne tournions pas correctement. Comme vous pouvez le voir sur ce schéma, nous faisions bien les bons calculs pour pouvoir tourner mais nous ne nous retrouvions pas au bon endroit au début du tour 12.

Nous avons fini par trouver d'où venait notre problème, cela ne venait pas de notre stratégie d'évitement mais de notre stratégie de replacement des marins. Pendant un tour, si les marins n'étaient pas utilisés, nous les placions au centre du bateau. Il s'avère que nous avions un cas limite où un marin se déplaçait à plus de 5 cases, ce qu'il n'a en effet pas le droit. Nous avons remarqué que si nous enlevions cette stratégie de replacement au centre du bateau nous passions la semaine, c'est donc la solution que nous avons choisi.

Si nous avions pu faire une course en mode bataille navale, nous aurions eu la stratégie d'attaquer tous ceux qui se trouveraient dans notre champ de vision. Nous nous serions mis à 90° par rapport à l'autre bateau pour pouvoir leur tirer dessus.

Outils utilisés et créés

Assez rapidement nous avions un simulateur et une interface graphique. Le simulateur faisait ce que l'arbitre fait pendant les courses sur QglWebRunner. L'interface graphique nous servait strictement pour visualiser où était placé les checkpoints, puisqu'il n'y avait pas encore d'obstacles.

Nous nous basions seulement sur les logs de notre terminal ce qui nous donnait le numéro du tour auquel nous étions, les nouveaux éléments de ce tour et les actions effectuées (Figure 5 et Annexe 6). Grâce à ceux-ci, nous avons pu

trouver beaucoup d'erreurs que nous n'avions pas pensé à tester. Nous avons gardé ceux-ci même après avoir fait notre simulateur visuel.

```
[05/05/2022 10:45:17][FINEST]
[05/05/2022 10:45:17]  [INFO]   === Tour 90 ===
[05/05/2022 10:45:17]  [INFO]   Next round input: {"ship":{"type":"ship", "life":2500, "position":{"x":2610.5690424615213, "y":3161.547048612852, "orientation":
[05/05/2022 10:45:17]  [INFO]   Actions: [{"sailorId":2,"type":"MOVING","xdistance":-1,"ydistance":-2},{"sailorId":4,"type":"MOVING","xdistance":-1,"ydistance":
[05/05/2022 10:45:17][FINEST]
[05/05/2022 10:45:17]  [INFO]   === Tour 91 ===
[05/05/2022 10:45:17]  [INFO]   Next round input: {"ship":{"type":"ship", "life":2500, "position":{"x":2713.301498683669, "y":3197.311370281611, "orientation":-
[05/05/2022 10:45:17]  [INFO]   Actions: [{"sailorId":9,"type":"MOVING","xdistance":0,"ydistance":2},{"sailorId":10,"type":"MOVING","xdistance":-1,"ydistance":2
[05/05/2022 10:45:17][FINEST]
[05/05/2022 10:45:17]  [INFO]   === Tour 92 ===
[05/05/2022 10:45:17]  [INFO]   Next round input: {"ship":{"type":"ship", "life":2500, "position":{"x":2867.634110964246, "y":3090.84903630992, "orientation":-0
[05/05/2022 10:45:18]  [INFO]   Actions: [{"sailorId":17,"type":"MOVING","xdistance":0,"ydistance":2},{"sailorId":19,"type":"MOVING","xdistance":-1,"ydistance":
[05/05/2022 10:45:18]  [INFO]   === Game has finished in 92 tours ===
```

Figure 5 : Exemple de log

Notre application visuelle affiche tous les éléments de la carte et la trajectoire du bateau. Nous pouvons choisir de voir tour par tour l'avancée du bateau, ou d'afficher toute la trajectoire d'un coup (Figure 6 et Annexe 7).

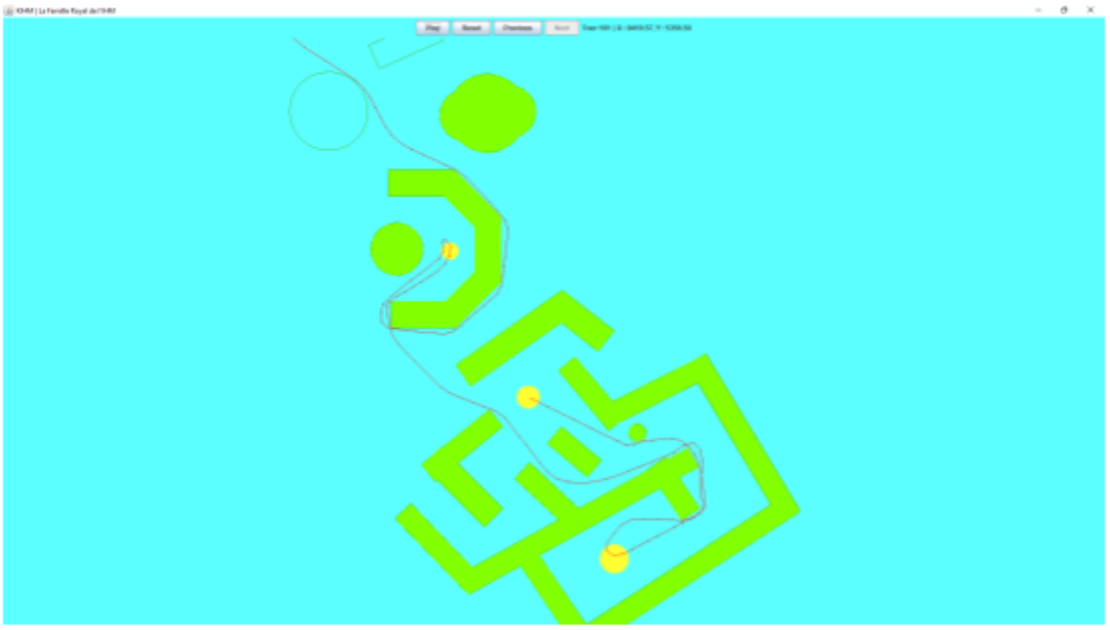


Figure 6 : Exemple de fonctionnement de l'application visuelle sur la WEEK8

Notre simulateur nous a posé problème lors de la course de la semaine 9. Nous avions un décalage entre ce que faisait notre simulateur et QglWebRunner. Sur notre simulateur nous passions la course mais pas sur QglWebRunner, tout ceci à cause du déplacement des marins. Quand il y avait le cas limite d'un marin qui se déplaçait trop nous le remettions à zéro ce qui faisait que nous pouvions quand même continuer la course et ne pas se prendre un mur.

Conclusion

Ce projet nous a permis à tous d'aborder de nouvelles notions d'une part au niveau d'une nouvelle logique de code que nous pouvions implémenter et d'autre part pour améliorer notre façon de coder.

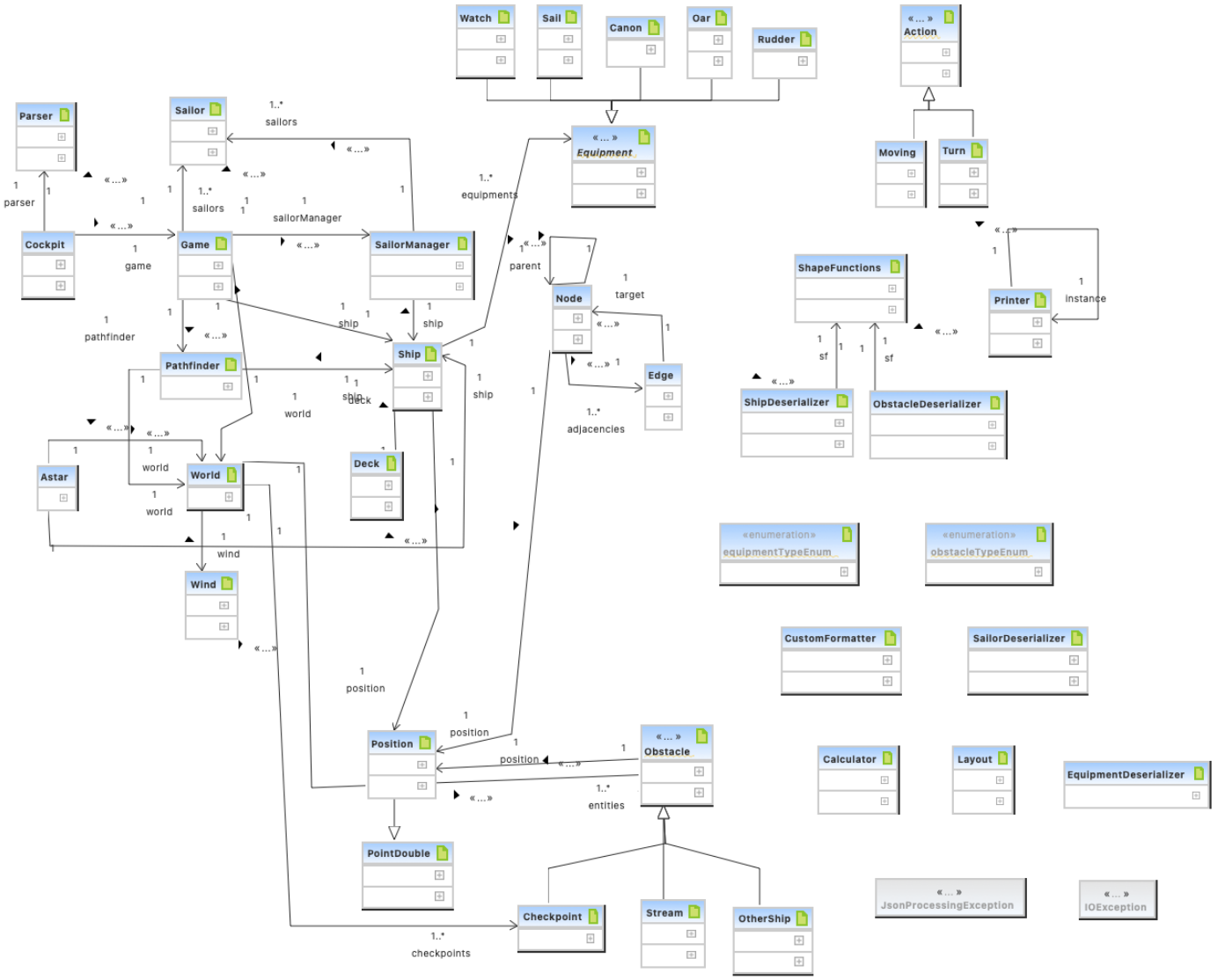
En ce qui concerne une nouvelle logique de code, nous avons appris à trouver un chemin optimal avec l'implémentation de notre algorithme A-Star, bien que nous ayons eu du mal à l'intégrer à notre projet. De plus, nous avons appris à manipuler un fichier JSON, aussi bien en lire un qu'en renvoyer un.

Nous avons appris à parfaire nos tests unitaires à l'aide de PIT qui nous a permis de renforcer nos tests. Nous essayons de faire monter et de garder un bon pourcentage de tests de mutations. De plus nous avons utilisé Sonar qui nous permettait de rendre notre projet avec le moins de failles de sécurité possible et un code le plus propre possible. Nous avons aussi approfondi nos connaissances avec l'outil Maven et plus particulièrement nous avons pu comprendre comment fonctionnait le Pom.xml associé.

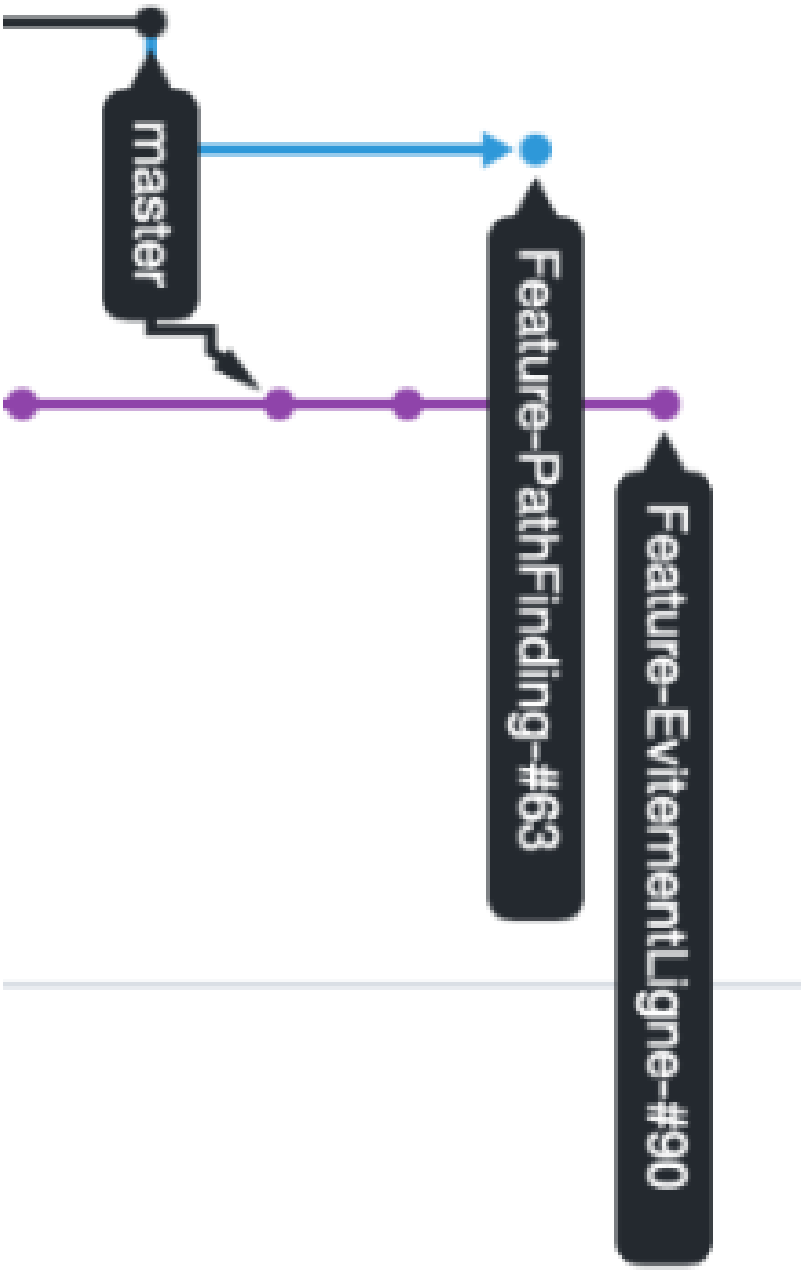
Durant nos dernières séances d'Algorithmique et Structures de Données, nous avons vu la notion de graph et donc de recherche de chemin comme notre A-Star. Nous avons eu ce cours un peu tard par rapport à quand nous en aurions eu besoin dans ce cours de QGL, ce qui nous aurait permis d'implémenter notre algorithme de recherche de chemin plus rapidement.

Ce que nous pouvons tirer de ce projet est que quand nous travaillons en équipe, il est fondamental de garder un code lisible et de suivre les mêmes règles. De plus, avec de bons tests unitaires ou d'intégration il sera plus facile de trouver où se trouve les erreurs qu'il pourrait y avoir dans le code.

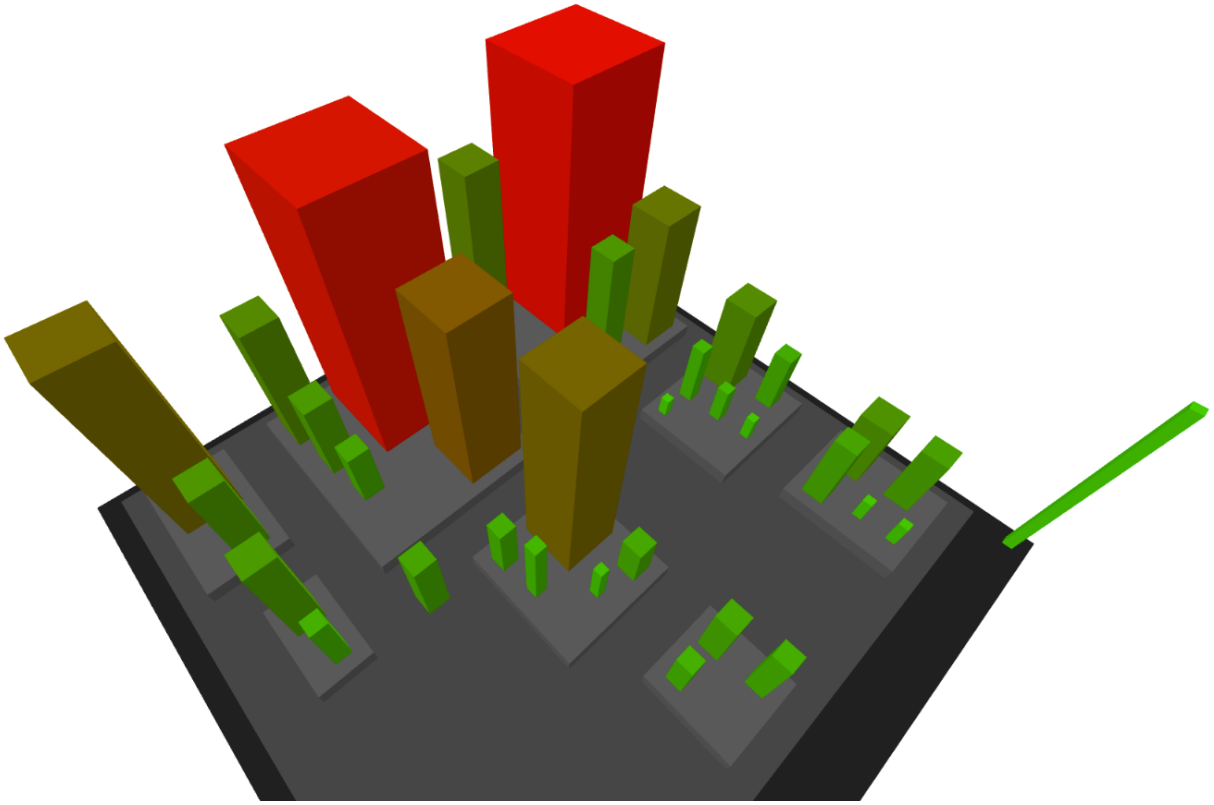
Annexes



Annexe 1 : Diagramme UML



Annexe 2 : Branching stratégie



Annexe 3 : CodeCity Sonar

Pit Test Coverage Report

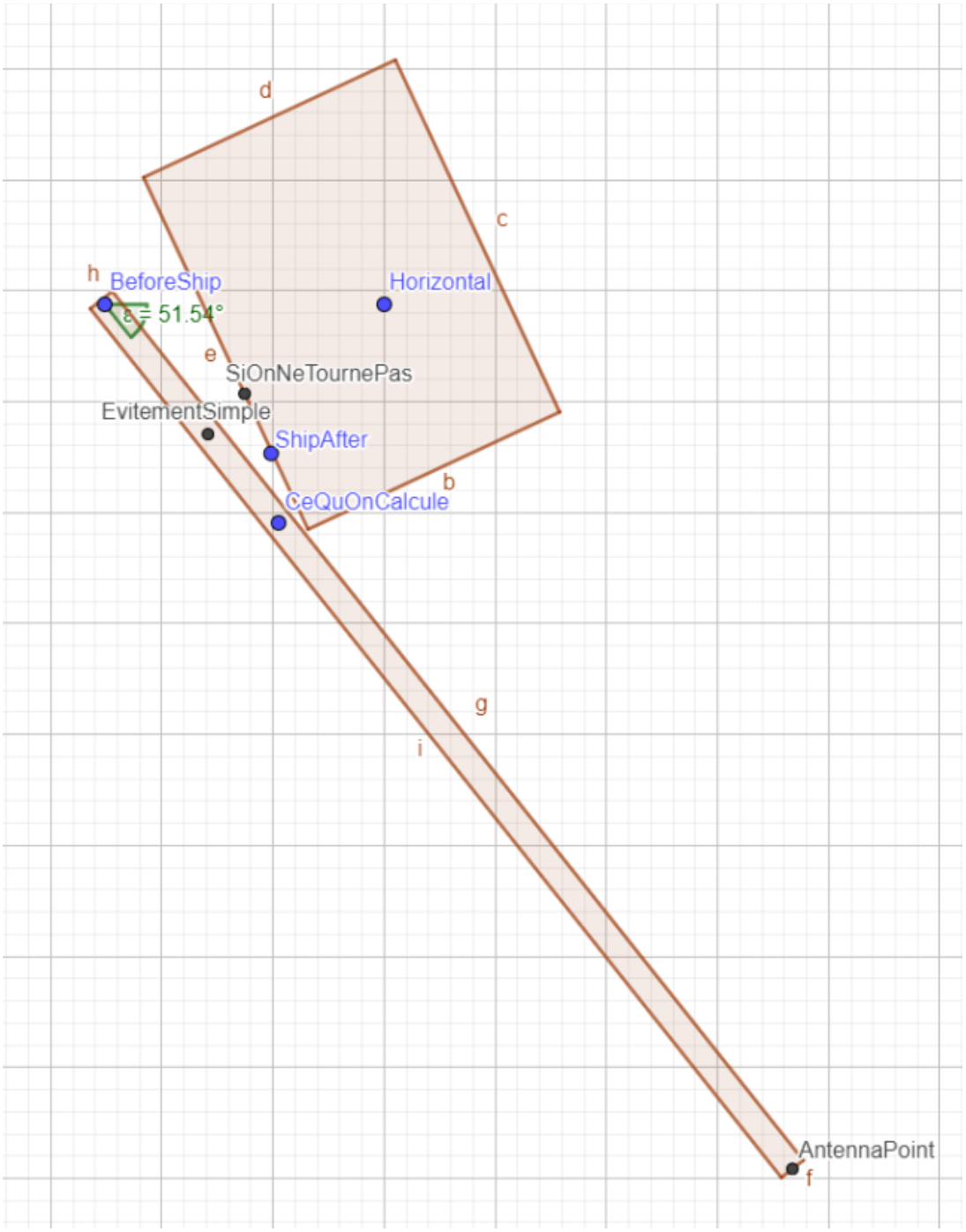
Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
33	96%	1159/1202	71%	696/977	79%	696/880

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
fr.unice.polytech.si3.qgl.kihm	1	100%	26/26	48%	10/21	53%	10/19
fr.unice.polytech.si3.qgl.kihm.actions	3	100%	52/52	82%	40/49	87%	40/46
fr.unice.polytech.si3.qgl.kihm.equipments	5	100%	77/77	86%	59/69	88%	59/67
fr.unice.polytech.si3.qgl.kihm.landmarks	6	100%	97/97	78%	47/60	84%	47/56
fr.unice.polytech.si3.qgl.kihm.logger	2	100%	66/66	87%	13/15	87%	13/15
fr.unice.polytech.si3.qgl.kihm.serializers	5	94%	155/165	92%	106/115	99%	106/107
fr.unice.polytech.si3.qgl.kihm.ship	5	100%	308/308	75%	151/201	78%	151/194
fr.unice.polytech.si3.qgl.kihm.structures	4	88%	247/280	52%	154/296	68%	154/227
fr.unice.polytech.si3.qgl.kihm.utilities	2	100%	131/131	77%	116/151	78%	116/149

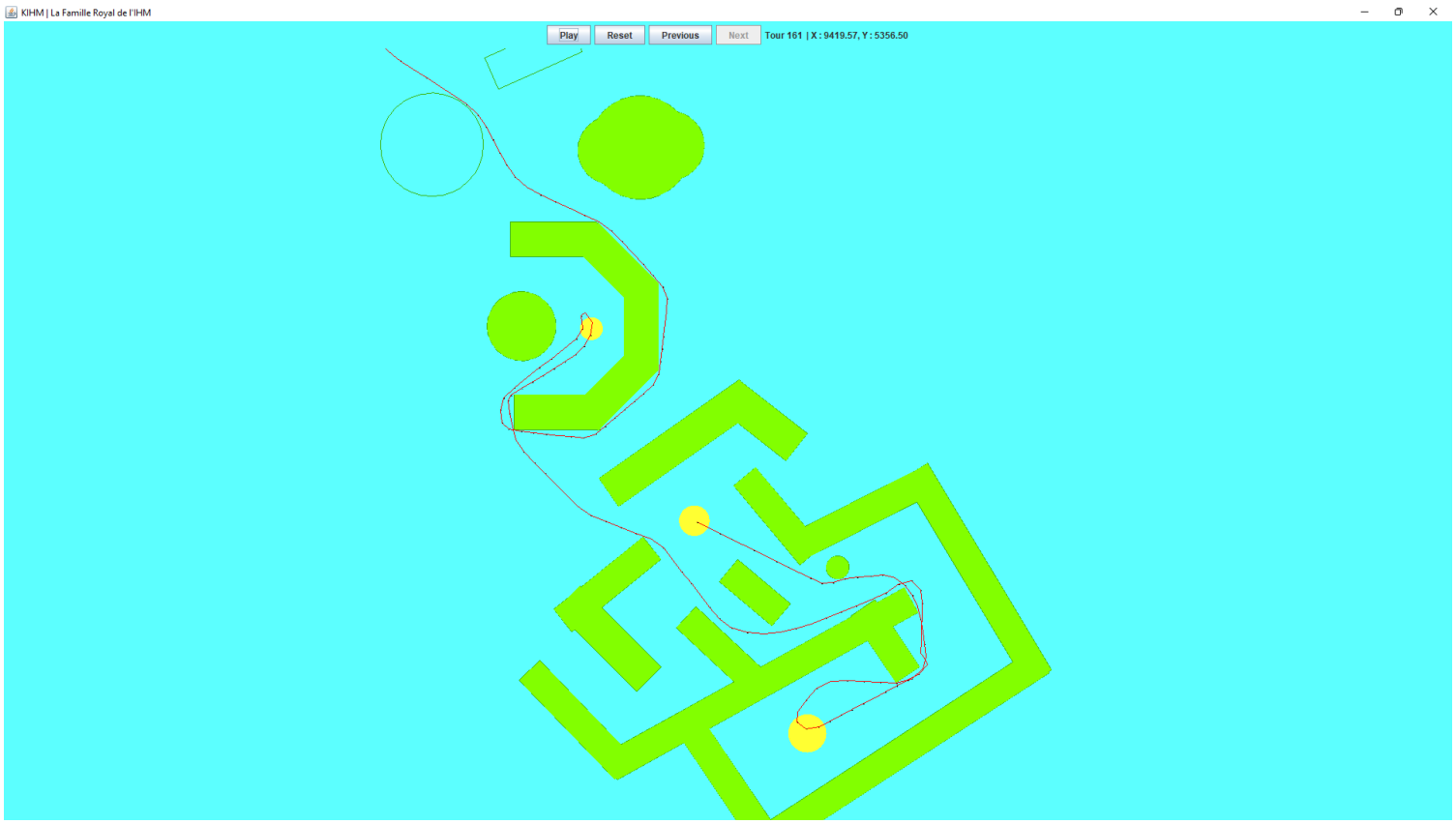
Annexe 4 : PITest



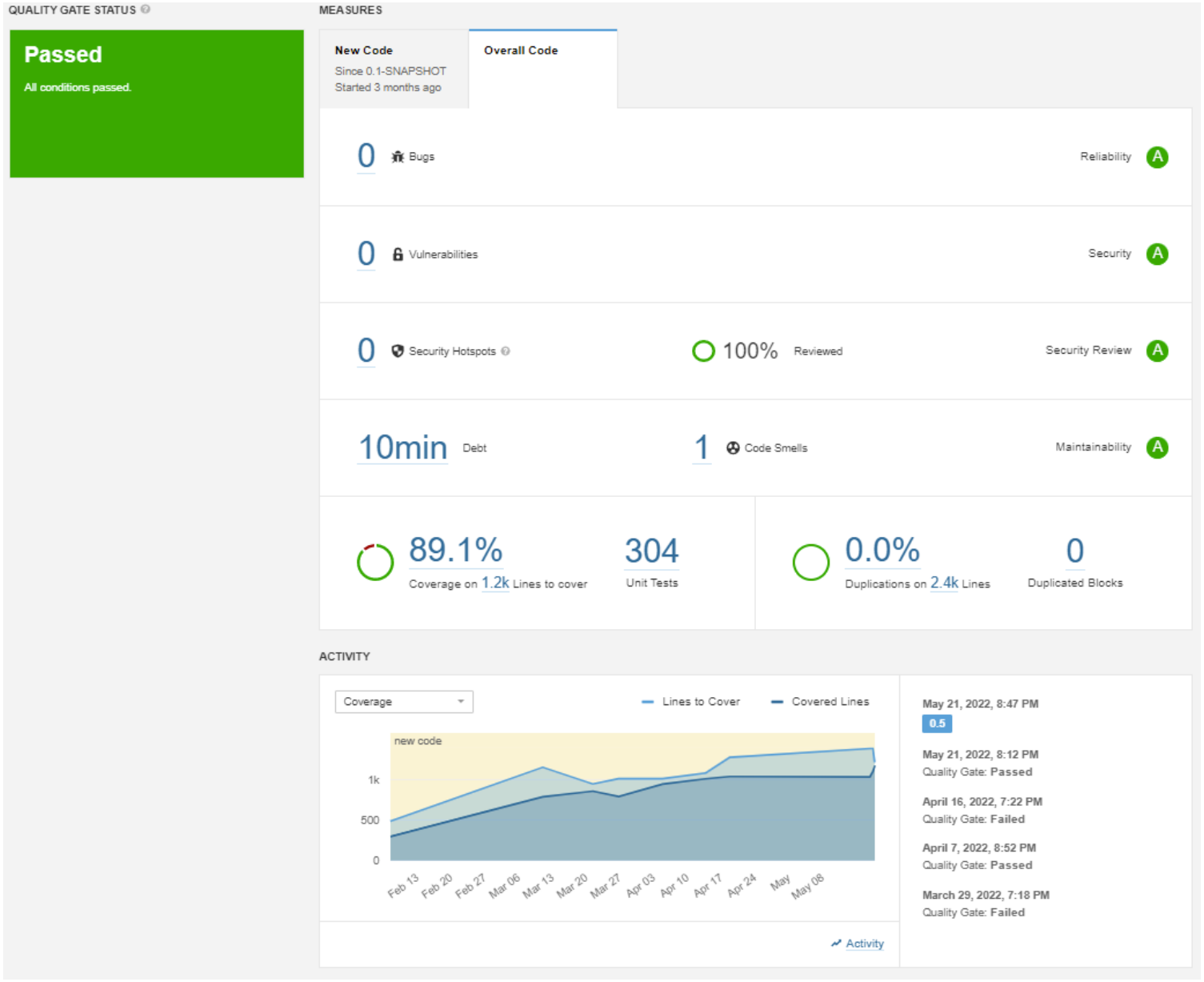
Annexe 5 : Schéma de l'évitement simple - GeoGebra

```
[05/05/2022 10:45:17][FINEST]
[05/05/2022 10:45:17]  [INFO]   === Tour 90 ===
[05/05/2022 10:45:17]  [INFO]   Next round input: {"ship":{"type":"ship", "life":2500, "position":{"x":2610.5690424615213, "y":3161.547040612052, "orientation":
[05/05/2022 10:45:17]  [INFO]   Actions: [{"sailorId":2,"type":"MOVING","xdistance":-1,"ydistance":-2},{"sailorId":4,"type":"MOVING","xdistance":-1,"ydistance":
[05/05/2022 10:45:17][FINEST]
[05/05/2022 10:45:17]  [INFO]   === Tour 91 ===
[05/05/2022 10:45:17]  [INFO]   Next round input: {"ship":{"type":"ship", "life":2500, "position":{"x":2713.301498683669, "y":3197.311370281611, "orientation":-
[05/05/2022 10:45:17]  [INFO]   Actions: [{"sailorId":9,"type":"MOVING","xdistance":0,"ydistance":2},{"sailorId":10,"type":"MOVING","xdistance":-1,"ydistance":2
[05/05/2022 10:45:17][FINEST]
[05/05/2022 10:45:17]  [INFO]   === Tour 92 ===
[05/05/2022 10:45:17]  [INFO]   Next round input: {"ship":{"type":"ship", "life":2500, "position":{"x":2867.634110964246, "y":3090.84903630992, "orientation":-0
[05/05/2022 10:45:18]  [INFO]   Actions: [{"sailorId":17,"type":"MOVING","xdistance":0,"ydistance":2},{"sailorId":19,"type":"MOVING","xdistance":-1,"ydistance":
[05/05/2022 10:45:18]  [INFO]   === Game has finished in 92 tours ===
```

Annexe 6 : Exemple de logs



Annexe 7 : Notre simulateur



Annexe 8 : Statistiques Sonar