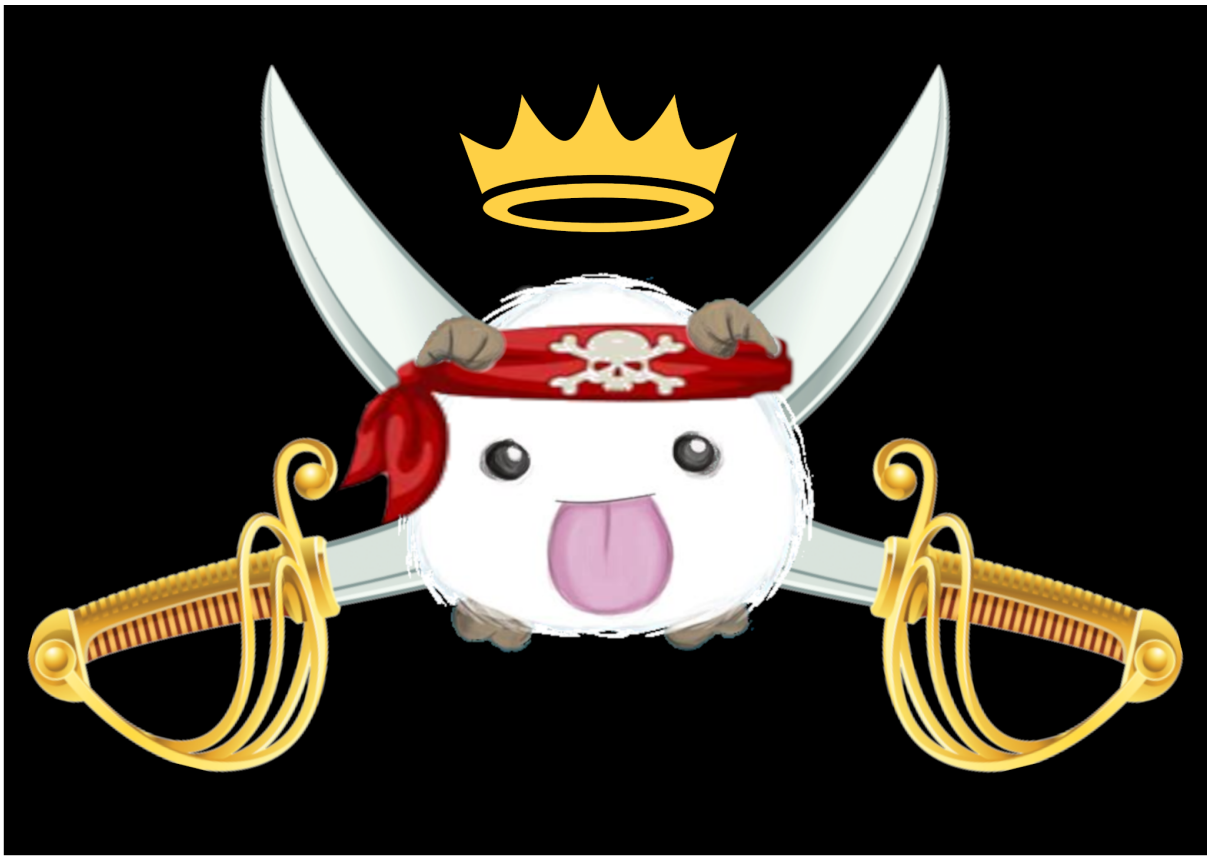
Rapport QGL

Royal Fortune - royal_fortune



KARRAKCHOU Mourad

-

BONNET Kilian

-

IMAMI Ayoub

-

LE BIHAN Léo

Sommaire

Description technique

Architecture du projet

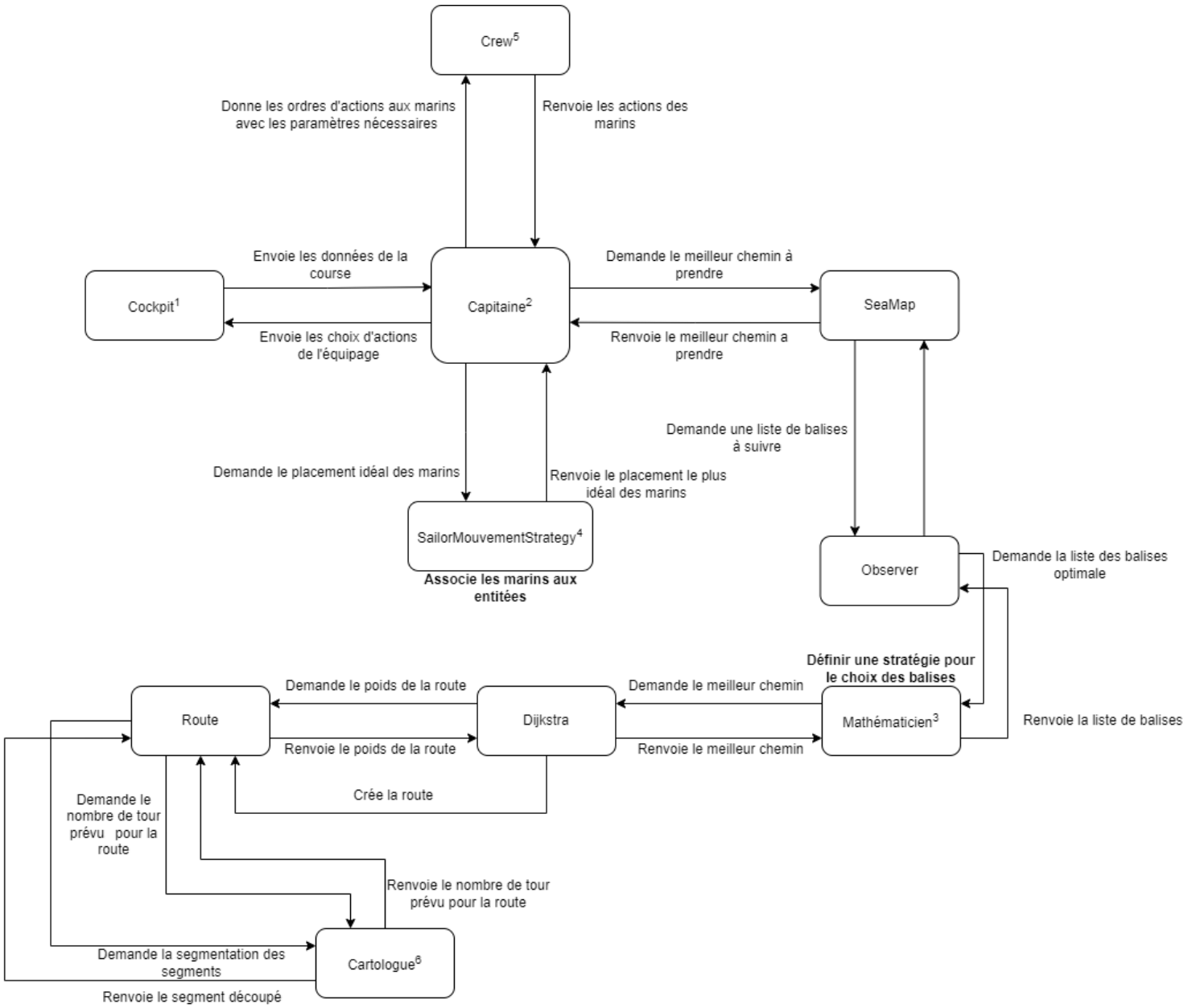


Schéma simplifié de l'architecture de notre projet

1- Cette classe respecte le principe de responsabilité unique : servir d'interface entre le runner et le **Capitaine**.

2- La classe **Capitaine** est l'élément central du projet. Elle permet - à l'instar d'un réel capitaine de navire - de faire la passerelle entre les divers membres d'équipage : placer les marins en fonction des informations reçues par le **Cartologue**.
Le **Capitaine** n'a ainsi plus la responsabilité de s'occuper des éléments extérieurs au bateau (récifs, courants, checkpoints).

3- Une fois la carte balisée, le **Cartologue** fait appel au **Mathématicien**. La responsabilité du Mathématicien est d'appliquer l'algorithme de Dijkstra sur une liste de balises lui étant donné en entrée. Si nous voulions implémenter un autre algorithme il suffirait de changer l'appelle dans le **Mathématicien**.

4- Lorsque le capitaine souhaite diriger son navire vers un point B ce dernier fait une requête au système de placement des marins qui pourrait être retranscrite par la phrase suivante : *"J'aimerais un différentiel de trois marins entre la gauche et la droite, un marin sur la vigie, aucun sur les voiles et un sur le gouvernail"*. Après l'application d'un algorithme prévenant au maximum d'éventuelles famines parmis les entités du navire, la réponse de la stratégie du placement des marins au Capitaine pourrait être retranscrite par la phrase suivante : *"J'ai pu te mettre un différentiel de trois marins, un marin sur la vigie, aucun sur les voiles, cependant aucun marin n'a pu être positionné sur le gouvernail"*.
Nous avons opté pour ce choix de conception pour deux raisons :

- Le dialogue entre la stratégie du placement des marins et le capitaine a pu être rendu simple grâce à une classe de communication nommée **SailorPlacement**.
- Le capitaine retrouve sa responsabilité unique de coordonner les différents éléments du bateau et de donner les instructions sans être en plus chargé par la gestion du placement des marins. La gestion du placement des marins s'étant progressivement complexifiée, cette dernière ne pouvait plus rester dans la classe capitaine comme l'était l'algorithme de placement naïf au début du projet.

5- L'utilisation d'une classe d'association nous permet de facilement mettre à jour notre code dans le besoin. Imaginons l'arrivée d'un canon nécessitant deux marins pour le manoeuvrer. Bien que l'algorithme de placement des marins doit prendre en compte l'existence du canon, le système d'association pourra rester identique.

6- La classe **Cartologue** a la responsabilité de traiter les éléments extérieurs au navire : récifs, courants, vent. Chaque SeaEntity possède une shape qui nous permet de placer des balises aux coins de ces dernières, ou tout autour dans le cas où la shape serait un cercle. Nous avons décidé de lier ces différents points à l'aide de segments, la longueur du segment peut être infinie si ce dernier traverse un récif. Facilement extensible, une nouvelle entité peut être ajoutée à notre système de balisage une fois l'implémentation de 'generateBeacons' effectuée.
Nous avons opté pour ce choix de conception pour deux raisons majeures : le quadrillage de la carte aurait pu fournir un chemin approximatif si la taille des cases était trop importante. La seconde raison étant que la trajectoire la plus courte entre le bateau et un checkpoint est une ligne droite. Si un récif vient interférer dans cette ligne droite, le chemin le plus court passera indéniablement par un des sommets du récif. Notre implémentation respecte ce principe.

Extensibilité de l’architecture

A ce jour, si de nouvelles entités ou de nouveaux récifs faisaient leur apparition nous serions rapidement capable d’adapter notre code à ces nouvelles contraintes. Cette extensibilité est en majeure partie due à l’utilisation d’interface et d'héritage entre les différentes classes. Les entités marines vont par exemple toutes être héritées de la classe principale SeaEntity. Dès lors, l’ajout d’une nouvelle entité ayant une “shape” inconnue par notre programme devra simplement implémenter les méthodes generateBeacon et computeIntersectionWith utiles pour le pathfinding. En termes d’interface, nous pouvons par exemple citer les checkpoints, checkpoints fictifs et balises qui implémentent l’interface IPositionable permettant à Dijkstra de traiter plusieurs entités dont le point commun est leurs positions.
Si nous avions à implémenter une bataille navale par exemple, nous pourrions ajouter les différentes classes de calcul de trajectoire et de stratégie au mathématicien, chargé de traiter ces informations. Le placement des marins, comme vu précédemment, serait - quant à lui - facilement adaptable aux nouvelles exigences.

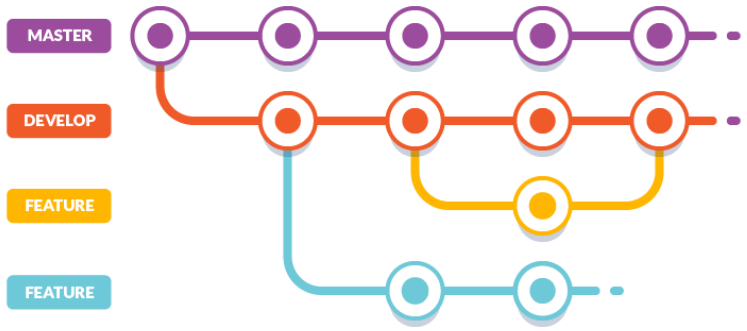
Interface ICockpit et schéma des JSONs

Afin de prendre en charge les entrées données sous format JSON nous avons mis en place un patron de conception permettant le lien entre la couche métier et la couche de persistance (ici les fichiers JSONs). Ce patron de conception se nomme le DAO. Nous avons choisi d’utiliser un InitGameDAO et un NextRoundDAO et cela nous a rendu le travail plus facile car nous avions simplement à désérialiser le json a en son DAO associé en utilisant la librairie Jackson. Par la suite il suffit d’utiliser les getters de nos DAO afin de récupérer toutes les données du JSON.

Les impacts de ces contraintes sur notre architecture ont majoritairement été sur le nommage et le typage des données. Nous avons adapté toute notre architecture au JSON que nous recevons en entrée et que nous devons renvoyer en sortie. Cela implique de respecter les conventions de nommage imposées par le JSON, mais aussi utiliser des annotations Jackson afin de définir précisément les “subtypes” de certaines classes mère, et ce afin que nos objets puissent être sérialiser et désérialiser sans aucun souci.

Application des concepts vu en cours

Branching strategy



La branching strategy utilisée est une version légèrement revisitée de GitFlow. Cette dernière se scinde en trois branches :

- Une branche stable (**Master**)
    - Cette branche a pour vocation d'être la branche la plus stable possible.
    - Prêt pour le tag, cette branche est parfois en retard sur d'éventuelles features ajoutées sur **Develop**.
    - Les tests couvrent la majeure partie du code ainsi que des mutants, qui sont tués lors du PiTest.

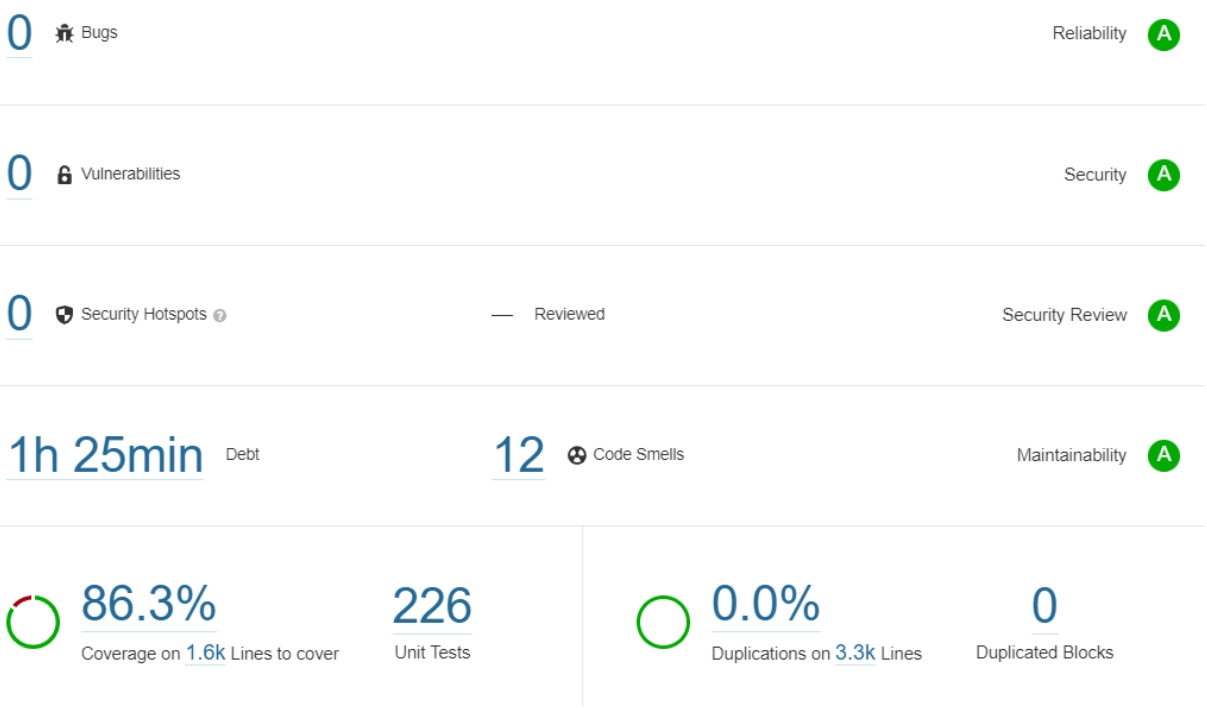
- Une branche presque stable (**Develop**)
    - Cette branche a pour vocation d'être une branche quasiment stable.
    - Les nouvelles features ajoutées sont en partie testées.
    - Des hotfix sont parfois nécessaires.

- Branches de feature
    - Ces branches n'ont pas pour vocation d'être stables et peuvent échouer lors de la compilation par Maven.
    - La convention de nommage est feature/NOM_FEATURE#ID_ISSUE

De cette façon, nous pouvons tous travailler simultanément sur nos branches sans créer de conflits. Le code déployé sur **Develop** et **Master** étant testé, cela nous assure un maintien de la qualité sur ces branches. Par ailleurs, la convention de nommage choisit pour les branches de feature nous permet de savoir exactement l'utilité de la branche en question.

Qualité du code

Nous estimons que la qualité de notre code est satisfaisante notamment grâce aux métriques de Sonar. Toutefois nous avions en moyenne 75% de mutants tués chaque semaines et nous estimons que ce chiffre aurait pu atteindre les 80%.
Nous avons appris à utiliser l'outil sans appliquer systématiquement toutes ses recommandations lorsque nous estimons qu'une partie du code doit rester telle qu'elle est. De ce fait, nous pouvons tirer le meilleur de Sonar. Actuellement nous sommes à 86.3% de coverage, 0 Bugs, 0 vulnérabilités, 12 code smells, 0% de duplication, ce qui donne une note A à notre projet.



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
48	88%	1519/1721	81%	928/1143	87%	928/1064

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
fr.unice.polytech.si3.qgl.royal_fortune	1	89%	51/57	78%	14/18	78%	14/18
fr.unice.polytech.si3.qgl.royal_fortune.action	6	75%	62/83	100%	12/12	100%	12/12
fr.unice.polytech.si3.qgl.royal_fortune.calculus	7	99%	334/336	94%	313/333	94%	313/333
fr.unice.polytech.si3.qgl.royal_fortune.calculus.dijkstra	2	98%	54/55	100%	23/23	100%	23/23
fr.unice.polytech.si3.qgl.royal_fortune.captain	3	98%	170/174	87%	115/132	88%	115/130
fr.unice.polytech.si3.qgl.royal_fortune.captain.crewmates	4	91%	288/318	73%	156/214	81%	156/193
fr.unice.polytech.si3.qgl.royal_fortune.dao	2	91%	29/32	100%	9/9	100%	9/9
fr.unice.polytech.si3.qgl.royal_fortune.environment	7	86%	128/148	71%	71/100	86%	71/83
fr.unice.polytech.si3.qgl.royal_fortune.environment.shape	5	80%	158/197	67%	116/174	78%	116/149
fr.unice.polytech.si3.qgl.royal_fortune.json_management	1	39%	25/64	50%	5/10	100%	5/5
fr.unice.polytech.si3.qgl.royal_fortune.ship	3	80%	96/120	77%	33/43	85%	33/39
fr.unice.polytech.si3.qgl.royal_fortune.ship.entities	3	84%	32/38	88%	22/25	96%	22/23
fr.unice.polytech.si3.qgl.royal_fortune.target	4	93%	92/99	78%	39/50	83%	39/47

Report generated by PIT 1.7.3

La mesure de la qualité nous permet de faire un gain de temps conséquent.
Si la qualité avait été moindre nous aurions sûrement eu à faire plusieurs refactors majeurs et perdu du temps à résoudre des problèmes mineurs dû à l'accumulation de mauvaises pratiques. Un code de faible qualité est difficilement maintenable et rend compliqué sa compréhension ainsi que la résolution de problèmes.
Si la qualité était encore plus élevée, le code serait très facilement compréhensible par chaque membre du groupe ne l'ayant pas écrit or actuellement, bien que la javadoc soit présente, il nous faut parfois communiquer des explications quant au fonctionnement de certaines parties du code.
La mesure de la qualité nous permet de mieux cerner les points importants, nous pouvons ainsi organiser le travail et la répartition des tâches en fonction des priorités. Il y a néanmoins eu quelquefois où nous étions pris par le temps mais les points importants étaient gérés assez tôt et n'avions donc aucun problème de livraison.

Refactoring

L'architecture de notre projet n'a pas nécessité de refactors majeurs, comme l'avait pu être celle de Citadelles lors de PS5. Toutefois, lorsqu'une classe commençait à être chargée et surtout lorsqu'elle commençait à avoir des responsabilités qu'elle n'était pas censée avoir, nous effectuions un refactor de cette dernière. Nous pouvons citer la classe DirectionManager comme exemple de classe que nous avons implémentée afin de décharger les responsabilités de la classe Captain : en effet, le Captain était censé seulement donner des ordres sur les actions que devait effectuer les marins, mais il commençait à devoir faire des calculs de directions qu'il n'était pas censé faire, d'où la création d'une nouvelle classe récupérant cette responsabilité.

Automatisation

Le test de compilation par Maven depuis GitHub nous alerte des éventuels tests qui ne passeraient plus et l'automatisation des PiTest nous permet de toujours garder une certaine qualité grâce à un seuil de mutant à conserver. De plus nous avons ajouté une Git Action qui, lorsqu'une une nouvelle version est push sur master, envoi sur un canal slack la dernière course réalisée par notre bateau sur notre WebRunner. Cela nous évite de relancer notre WebRunner manuellement si l'on souhaite avoir un visuel de la trajectoire du bateau.
De plus avant de mettre en place la Git Action slack qui nous notifie rarement, nous avons créé une autre Git Action, qui, lorsqu'une nouvelle version du code est déployée sur Master et Develop nous envoie un message sur le discord afin de nous permettre de suivre l'actualité du projet.

Etude fonctionnelle et outillage additionnels

Les checkpoints fictifs

Afin de viser les checkpoints de façon plus précise, l'une des premières stratégies que nous avons implémentés était la mise en place de checkpoints fictifs.
Soit C1 un checkpoint de rayon R1 et C2 le checkpoint suivant, nous créons un checkpoint fictif CF1 de rayon R1 / 2 et nous plaçons son centre à une distance R1 / 2 du centre de C1 et le plus proche possible de C2. Celà nous permet d'avoir un checkpoint qui nous rapproche du prochain checkpoint tout en prenant le checkpoint original.
À cela nous avons ajouté une stratégie de ralentissement au voisinage du checkpoint pour s'arrêter le plus tôt possible dans le checkpoint fictif et ne pas s'y enfoncer. Nous avons fait ce ralentissement car cela nous permettait d'avoir une meilleure trajectoire au voisinage d'un checkpoint et de gagner du temps.

Le pathfinding

Nous avons ensuite dû développer un moyen de naviguer en esquivant des récifs et en prenant des décisions sur la stratégie à aborder aux abords des courants. Nous avons opté pour un algorithme de Dijkstra qui se base sur un système de route et de balises.
Une route est une classe possédant un poids représentant le nombre de tours nécessaire afin parcourir cette dite route(ce qui sera utile pour Dijkstra). Elle se construit à la manière d'un arbre binaire : tant que la route coupe une SeaEntity elle crée 2 routes à partir des intersections avec cette dernière, qu'elle va posséder. Les feuilles de l'arbre sont alors les routes qui ne coupent aucun côté d'une SeaEntity (elles peuvent être dans la SeaEntity). Le poids d'une route nœud est alors la somme de la longueur de ses 2 routes filles. Le poids d'une route feuille est calculée en fonction de 3 cas:

-La route n'est pas associée à une Sea Entity, le poids de la route est la distance/la vitesse du bateau à plein régime.

-La route est dans un Reef, le poids est donc +l'infini.

-La route est dans un Stream, le poids est calculé en prenant en compte le courant entre les 2 points de la route et la distance.

Dans un second temps nous devons créer des balises sur lesquelles nous pourrons appliquer Dijkstra. Nous savons qu'il existe un chemin idéal qui est le chemin le plus rapide. Notre but est d'avoir, dans les balises que nous générons, des balises présentes sur ce chemin afin de pouvoir le recréer. Pour les Reefs nous savons que le chemin idéal passe forcément par les sommets des figures géométriques (pour le disque sur le cercle). C'est donc à ces positions que nous avons disposé nos balises. Pour les Streams, les balises sont placées sur les quatres segments du rectangle.
La dernière étape est d'appliquer l'algorithme Dijkstra sur les balises et celui-ci trouve le chemin le plus court en créant des routes entre les différentes balises et en utilisant le poids de chacune de ces routes. Enfin nous utilisons les balises retournés par Dijkstra comme des checkpoints intermédiaires sur lesquels le ralentissement n'est pas appliqué. Dans le cas où deux balises seraient alignées, par exemple, ralentir pour entrer dans la première balise ne serait pas cohérent avec la recherche du temps de navigation minimale désiré.

La stratégie de placement des marins

Lors du refactor de la classe Capitaine, qui avait pour but de l'alléger du traitement du placement des marins, nous en avons profité pour remplacer l'algorithme naïf par un algorithme plus poussé et améliorer son adaptabilité à l'ajout de nouvelles entités.

L'algorithme par détection de famines

Plus les weeks avançaient, plus le bateau présentait un nombre de marins conséquent ainsi qu'un deck plus grand. L'algorithme naïf qui au début s'avérait performant a commencé à présenter ses limites lorsque le déplacement d'un marin sur le deck du bateau pouvait dépasser la limite imposée de 5 cases.
L'un des problèmes rencontrés par notre algorithme fut le non-traitement d'entités sous famine, l'association naïve empêchant certaines entités d'être associés tout le long de la partie. Afin de résoudre ce problème nous avons conçu un algorithme en trois phases prêtant une attention particulière aux entités affamées. Ces trois phases sont les suivantes :

- Associer les entités affamées tant qu'il y a des entités affamées et que la requête du Capitaine n'est pas satisfaite.
- Finir d'associer les éventuels voiles, gouvernail et vigie manquant en leur associant le marin le plus proche.
- Associer les entités les plus proches des rames afin de créer l'éventuel différentiel de marins entre les rames de gauche et de droite.
- Associer les marins restants afin qu'ils se disposent par paire sur une rame de gauche et de droite. L'association prend en compte le fait que le marin soit à proximité d'une rame à gauche, une rame à droite ou une rame à gauche et à droite.

Une fois l'association terminée et comme vu précédemment, la stratégie de placement des marins enverra au capitaine un objet SailorPlacement lui indiquant si oui ou non les associations ont pu être faites et à quel degré de similarité par rapport à la demande du capitaine.

La sécurité

Afin de s'assurer de la sécurité du navire lors de l'application des différents algorithmes de calcul de trajectoire, nous avons décidé d'augmenter artificiellement la taille des récifs. Notre algorithme de calcul de trajectoire repose, comme vu précédemment, sur un calcul de trajectoire parmi des balises situées proches des récifs. Cette proximité vis-à-vis des récifs nous a incité à implémenter rapidement une solution visant à ne plus faire passer notre bateau au pixel près des récifs.

Notre runner

Nous avons très vite senti le besoin de développer notre propre runner de course de bateau afin de visualiser les courses, de les perfectionner au maximum et de tout simplement pouvoir tester notre code avant le rendu hebdomadaire. Nous avons donc développé un runner en 2 partie:

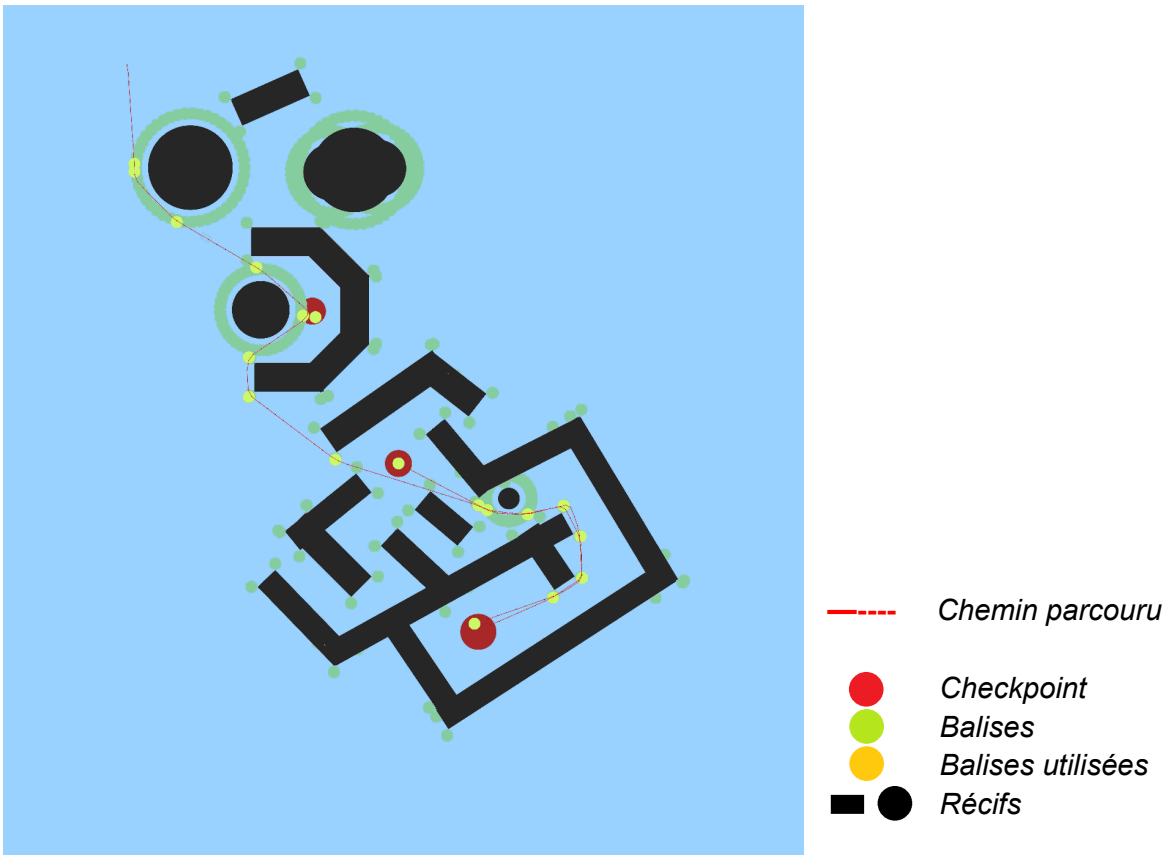
- Un tool java qui envoie les JSONs à notre bateau, et simule son avancement en fonction des actions réalisées par les marins.
- Un webtool réalisé en HTML/JS/CSS qui va nous permettre d'avoir un rendu graphique de la course, il prend en entrée un fichier construit par notre tool java qui va décrire la structure de la course réalisée (les checkpoints, les courants et les récifs), ainsi que le trajet du bateau.

Ce runner nous a beaucoup aidé à progresser et à mieux comprendre l'avancement de notre bateau, et même si cela demandait du travail en plus, c'était un investissement rentable. De plus, nous avons pu afficher des informations qui nous sont très utiles, en particulier les balises que l'on utilise afin de trouver un chemin à travers les récifs.

Enfin nous avons rajouté une partie backend au webtool afin de pouvoir réaliser une capture des courses lancées et enregistrer la dernière course réalisée dans les fichiers du runner. Nous avons ensuite créé une Git Action qui, à travers un webhook slack, nous tiens informer de la dernière course réalisée lorsque nous déployons une nouvelle version sur master.

Comme dit précédemment, cette Git Action nous permet d'avoir rapidement à notre disposition la dernière course de runner lorsqu'on en a besoin sans qu'on ai à lancer une course sur notre tool.

Image d'une course sur notre webtool:



Conclusion

Ce que nous avons appris

Ce projet nous a fait découvrir le potentiel de Git, notamment à travers la notion de branche avec laquelle nous sommes maintenant à l'aise et qui nous a offert une nouvelle façon de travailler ensemble. Nous exploitons beaucoup mieux Git qu'avant ce projet.
La notion de qualité logiciel étant nouvelle, nous avons dû apprendre à utiliser Sonar et PiTest, de manière régulière afin d'avoir un suivi et conserver au mieux une qualité satisfaisante malgré l'avancement rapide du projet. Au début nous utilisions Sonar sans vraiment prendre de recul, nous tentions de modifier tout ce que l'outil indiquait, mais nous avons appris petit à petit à l'utiliser de façon intelligente vis à vis de notre projet.
Nous avons découvert les projets Maven et l'intérêt du pom.xml qui nous permet d'ajouter des plugins comme les PiTests.
Enfin, nous avons appris l'existence et l'utilisation des Git actions qui nous permet d'automatiser une partie de nos tâches et nous faire gagner du temps.
Afin de maintenir une qualité constante lors de l'ajout de nouvelles fonctionnalités, l'application des principes de conception SOLID nous a permis d'apprendre l'importance et la mise en application du principe d'interface et héritage.

Connaissances exploitées

Durant ce projet nous avons chacun pu mettre en pratique des connaissances personnelles acquises les années précédentes, que ce soit en mathématiques pour le calcul de trajectoire, de vecteurs ou de changement de base, mais aussi en développement web pour la réalisation de notre propre webtool.
Les connaissances venant d'autres projets, tels que les projets de PS5 nous ont apportés les bases de l'utilisation de GitHub et de la réalisation de nos premiers tests.
Les cours d'algorithme structure de données (ASD) nous a permis d'approcher l'algorithme de Dijkstra et de placement des marins plus sereinement. En effet, faisant plusieurs appels récursifs et exploitant certaines structures de données comme le graph ou le tas, nos connaissances acquises lors de ce cours nous ont permis de gagner un temps important lors de l'implémentation des ces algorithmes.

Leçon tirées

Les premières semaines, nous étions très investis dans le projet notamment car nous occupions la tête du classement. Mais nous nous sommes rendu compte que cet investissement était plus centré sur le classement que sur la qualité de notre travail. Nous nous sommes remis en question et avons donc décidé de ré-orienté nos efforts dans la qualité, quitte à faire moins de points lors des courses, tout en gardant l'objectif de mettre en place de bonnes stratégies. Le fait de s'être remis en question nous a été bénéfique car une meilleure qualité de code nous a finalement bien aidé à maintenir ce dernier et nous a facilité l'implémentation de nouvelles fonctionnalités. Nous nous sommes donc rendu compte de l'importance de la qualité, surtout sur des projets de longue durée.