Université Nice Sophia Antipolis | POLYTECH NICE-SOPHIA

Membre de UNIVERSITÉ CÔTE D'AZUR

Rapport technique – QGL

Équipe

StormBreakers

ID: stormbreakers

FISE

Membres

ANAGONOU Patrick

BOUZIDI Héba

LEBRISSE David

STRIEBEL Florian

Université de Nice-Sophia Antipolis

Polytech'Nice – SI3 – 2020

Table des matières

Description technique

1. Architecture du projet

- Phases du programme
    1. Traitement des inputs (gameInit et nextRound)
    2. Calcul d'un chemin
    3. Détermination d'un objectif de vitesse et d'orientation
    4. Détermination des actions des marins et mise à jour de leurs positions
    5. Création de l'output

Nous nous sommes inspirés des designs patterns Observer et Dependency Injection pour concevoir notre architecture de haut niveau. Nous avons une classe MainLinker qui, à partir des informations de l'initGame, instancie les classes principales de notre programme

Les classes instanciées sont dans l'ordre :

```
Wind, CrewManager, EquipmentsManager, CheckpointsManager, Boat,
Coordinator, WeatherAnalyst, StreamManager, GraphCartographer,
TargetDefiner, Captain
```

Les objets sujets à des changements d'état liés aux nextRounds reçoivent à leur construction une instance d'une classe implémentant l'interface InputParser (en l'occurrence ici un JsonInputParser). Ils implémentent l'interface PropertyChangeListener de java.beans et sont abonnés à un observable ObservableData. Ainsi lorsqu'on reçoit un nextRound on met à jour l'observable et tous les objets concernés peuvent automatiquement mettre à jour leurs données. Les classes jouant le rôle d'*observateurs* concernées sont : Wind (vent donné dans les tours uniquement avec possibilité de changement entre deux tours), EquipmentsManager (ouverture, fermeture des voiles), Boat (la vie du bateau), StreamManager (entités externes données seulement pendant les tours et changements possibles entre deux tours).

Au lancement d'un tour, les données sont mises à jour comme indiqué précédemment. Ensuite le **Captain** doit déterminer les actions à effectuer. Pour cela il utilise le **TargetDefiner** pour obtenir un objectif de vitesse et d'orientation à appliquer au bateau. Le **TargetDefiner :**

- Met à jour les checkpoints restants par l'intermédiaire du **CheckpointManager**
- Obtient du CheckpointManager le prochain checkpoint
- Utilise le StreamManager pour vérifier s'il y a un obstacle entre le bateau et le prochain checkpoint
- Si c'est le cas, délègue la recherche d'un chemin à **Cartographer** qui lui renvoie un point (IPoint)
- Utilise le point venant du Cartographer (ou le checkpoint qui implémente IPoint) pour calculer l'orientation et la vitesse pour atteindre ce point. (Des ajustements sont faits et un troisième paramètre booléen ***strict*** est utilisé pour gérer certains cas particuliers.)
- Permet de retourner au Captain un objet regroupe l'orientation, la vitesse et le paramètre strict.

Le Captain prend en compte cet objectif et gère le placement des marins et la mise à jour de leurs positions à travers la classe Coordinator qui à son tour utilise EquipmentsManager et CrewManager.

On commence par assurer l'orientation, puis on s'occupe de la vitesse, et enfin s'il y a des marins non utilisés on les rapproche vers les rames ou outils.

Le Captain renvoie enfin les actions et celles-ci sont transformées sous format JSON avec l'OutputBuilder.

2. Justification des choix architecturaux

Nous avons créé des interfaces pour l'analyse des données et des classes pour implémenter ces interfaces dans le contexte du JSON. L'avantage est que cela nous permet de n'analyser un élément que s'il est utile pour la livraison ou pour l'optimisation. Par exemple, nous n'avons ajouté la forme « Shape » du bateau à notre classe Boat que lorsque nous avons voulu gagner des tours en validant les checkpoints par collision avec la forme du bateau. Un autre exemple est le fait que notre projet ne s'interrompt pas lorsqu'un élément non supporté est présent.

Nous avons une séparation claire entre le groupe de classes qui s'occupe de l'environnement externe du bateau et celui qui s'occupe des actions des marins dans le bateau. Voir annexe « Diagrammes des classes principales de la solution » pour une illustration de ces regroupements de classes.

De plus notre architecture a évolué au cours du temps tout en restant stable. Par exemple, au début on n'utilisait pas de pathfinding. C'était TargetDefiner qui devait chercher les solutions de contournement d'obstacles à l'aide de la classe StreamManager (qui gère les entités de la mer). Pour ajouter le pathfinding, nous n'avons eu qu'à créer la classe Cartographer qu'on passait dans le constructeur de TargetDefiner qui désormais délègue la tâche de recherches de solutions de contournement à Cartographer. Aussi, notre pathfinding utilise Dijkstra dont l'algorithme est implémenté dans une nouvelle classe « Graph ». Ainsi changer l'algorithme du pathfinding n'affecte pas du tout TargetDefiner (et ses classes clientes) et très peu voire pas du tout **Cartographer**.

Notre architecture est donc très modulable, et est également facilement testable puisqu'en général les classes n'instancient pas les objets qu'ils utilisent mais les reçoivent en paramètre de constructeur. Il est donc aisé de tester chaque classe en « mockant » les autres.

- Extensibilité de l'architecture

Fonctionnement global actuel

Pour l'instant notre projet ignore certains attributs du jeu comme le mode de jeu. Mais nous pensons que notre solution est flexible et adaptable facilement à d'autres modes de jeu. En effet, la communication entre les classes qui analysent l'environnement externe du bateau et celles qui gèrent la partie "intérieure" du bateau se fait au travers d'un objet "objectif" *TupleDistanceOrientation.* En fonction du mode de jeu l'objet va devoir différer. Pour passer à un projet multimode on peut

appliquer ici le **pattern Visitor** et rajouter du polymorphisme. Voir annexe « Exemple de modifications à effectuer pour supporter d'autres modes de jeu » pour plus de détails.

Ainsi notre architecture est assez facilement extensible pour supporter plusieurs modes de jeux.

3. Contraintes du schéma des JSON

Au début du projet nous avions défini des classes suivant le format des schémas des JSON en entrée (initGame et nextRound), et à cause de cela on avait des classes imbriquées les unes dans les autres. Mais cette organisation a été très vite remplacée notamment pour les classes de haut niveau, plus modulaires, créées par le MainLinker qui instanciaient les classes. En revanche, pour les classes de "bas niveau" (comme Equipment ou SailorAction), le modèle suit généralement celui du format d'entrée. Nous avons également poussé le détachement par rapport au JSON original en extrayant des interfaces : **InputParser** et **OutputBuilder.** Dans le projet nous avons implémenté ces interfaces par des JsonInputParser et des JsonOutputBuilder. Ainsi, s'il faut changer le format d'entrée, il suffira d'implémenter ces interfaces par des classes telles que XmlInputParser, XmlOutputBuilder ….

La principale limitation du format des JSON a été au niveau des formes : le fait qu'elles n'aient pas de position nous obligeait à plus de calculs (translations/rotations/changements de repères) et cela a été en partie la cause de notre UnfinishedError sur la Week6.

Application des concepts vu en cours

1. Organisation GitHub

Voir annexe « Illustration de la Git Strategy finale », explications dans ce point.

La branching strategy actuelle est inspirée du Gitlab flow, seulement l'ordre des étapes des merges est entièrement inversé : nous allons des features à la branche master (qui a, chez nous, toujours été stable). Parlons tout d'abord des principales évolutions de notre stratégie.

À un stade avancé de notre solution, nous avons créé une branche **prod**, des branches **feature,** et des branches **hotfix** en plus du master. Ainsi pour chaque release nous développions les fonctionnalités nécessaires sur les branches **features/**<...>. Des pull requests étaient ensuite lancées vers prod. Ainsi d'autres membres de l'équipe inspectaient les pull requests puis les validaient, ou demandaient des corrections si nécessaire. Lorsque toutes les nouvelles fonctionnalités étaient prêtes, on vérifiait que tout fonctionnait bien sur prod puis on faisait une **pull request** vers **master** pour enfin placer le tag.

Mais cette stratégie avait ses limites. En effet, nous commencions souvent à travailler sur notre projet le jeudi, sachant que la livraison était pour le lundi qui suivait : le délai était court pour implémenter nos **features**, lire les pull requests des coéquipiers, demander les corrections, etc. De plus certaines fonctionnalités avaient besoin que d'autres soient prêtes pour être développées. Nous avons alors instauré une branche **develop** qui permettait de merger certaines **features** sans attendre de pull requests. Cela permettait au développeur qui avait besoin de certaines fonctionnalités en cours de développement de pouvoir les avoir plus vite. Il pouvait également corriger les erreurs plus rapidement.

2. Qualité du code

a) Sonarqube et SonarLint

Pour essayer d'avoir une meilleure qualité pour notre code nous avons utilisé les outils Sonarlint ainsi que Sonarqube. Ces outils analysent notre projet et repèrent du code qui n'est pas écrit selon les conventions d'écriture et qui pourrait par exemple lever des erreurs. Le premier nous permet de repérer directement lorsque nous développons dans notre IDE les parties de code problématiques (comme la comparaison entre deux objets de types différents qui retournera toujours faux, ou bien des variables non utilisées qui peuvent indiquer que le développeur a oublié de prendre des cas en compte). Le second outil, Sonarqube, a un fonctionnement plus poussé : il repère les parties de code pouvant être problématiques et génère un rapport sous la forme d'une page web sur laquelle ces lignes sont triées par sévérité de « code smell ».

b) Tests unitaires, tests d'intégration et mutations de tests (PiTest)

Si les outils Sonar nous permettent d'obtenir du code de qualité, il nous faut bien un autre moyen pour vérifier que notre code produit fait bien ce qui est attendu.

Pour cela, nous testons avec des tests unitaires nos méthodes via le Framework JUnit et Mockito. Nous avons couvert de façon méthodique les classes de bas niveau (mathématique, géométrie). L'architecture des classes de haut niveau a rendu l'utilisation des mocks très faciles pour contrôler l'environnement de nos classes pour mieux tester leur comportement. Le problème est que nous ne sommes pas toujours sûrs que nos tests couvrent tous les cas. Pour faire face à cela, nous utilisons dans un premier temps l'outil coverage des IDE pour voir si toutes les branches de nos méthodes (ou

au minimum les branches critiques) sont couvertes par les tests. Mais cela ne suffit pas pour s'assurer que les tests couvrent bien la méthode. Nous utilisons alors l'outil PiTest qui crée des versions alternatives de notre code et lancent nos tests dessus. Cela permet de savoir si nos tests sont assez précis, et couvrent assez de cas. Bien sûr, PiTest n'est qu'un outil indicateur.

Par exemple pour le pathfinding, le fait que des mutations survivent ne veut pas forcément dire que le code est mal testé. Certains de nos tests sont assez généraux et flexibles. Par exemple si on vérifie l'assertion "l'algorithme trouve un chemin" le fait que des mutations survivent peuvent simplement dire qu'un autre chemin est trouvé malgré certaines mutations.

Toutefois, sur des parties comme les mathématiques et la géométrie un taux élevé de survie peut indiquer que le code est mal testé.

Enfin nous avons réalisé, dans la classe MainLinker, des tests de type intégration qui nous permettent de savoir si notre programme s'interrompt dès le début du lancement. Nous nous assurons ainsi que nous évitons des erreurs de type GenericClientException à la livraison.

c) Impacts du niveau de qualité

Pour la qualité de code nous avons les chiffres suivants :

Duplications de code	Couverture de tests	Complexité cognitive	Complexité cyclomatique
0	88%	532	1103

Ces bons résultats s'expliquent par le fait que dès le début du projet nous avions pris soin d'utiliser SonarQube et SonarLint pour nous assurer que le code livré était aux normes. Nous avons dans certaines situations privilégié la qualité du code à l'optimisation. Par exemple pour renvoyer les actions d'un marin nous incluons systématiquement le déplacement même si ce déplacement peut se révéler être de (0,0). Cela nous permet de généraliser et réduire la complexité cognitive de notre code. De même dans les cas où l'orientation est de Pi, nous avons arbitrairement choisi de tourner à gauche. Mais une fois encore nous avons préféré garder un code simple quitte à perdre quelques tours.

Aussi, en réalité c'est dans l'esprit de réduire la complexité cognitive de notre projet que nous avons utilisé les graphes. En effet, nous nous sommes vite rendu compte du grand nombre de cas à gérer pour éviter les courants défavorables et les récifs, emprunter les courants favorables, gérer les récifs se trouvant dans des courants... Nous avons donc ajouté des classes pour le pathfinding avec des notions de coûts pour traduire l'impact des vents, courants favorables/défavorables, etc. Au niveau de la surface des obstacles, les nœuds ne sont pas créés. Ces abstractions ont permis de gérer d'un coup beaucoup des préoccupations précédentes.

Ainsi si notre qualité avait été plus faible nous aurions raté plus de courses. Il faut aussi dire que malgré les bons chiffres de qualité nous aurions certainement pu faire mieux étant donné que le score variait parfois sur la même course avec des modifications normalement mineures sur le code. Une meilleure qualité nous aurait donc permis de pouvoir garantir au point près nos scores sur les courses.

3. Refactorisation

a) Refactorisations effectuées

Nous avons effectué jusqu'à trois refactorisations importantes dans notre projet. La première a eu lieu après la fin de la release Week2 et a été finie pour la release de Week4. Au départ nous avions construit des classes qui reflétaient la structure des fichiers JSON en entrée (par exemple, nous avions des classes NextRound et InitGame). Nous avons donc progressivement supprimé ces classes pour construire des classes plus modulaires et responsabilisées.

La seconde refactorisation a été faite pour résoudre le problème de l'orientation dans les positions. (Voir point suivant pour détails)

Enfin la troisième refactorisation a eu pour but de changer l'emplacement de certaines classes et d'extraire les interfaces InputParser et OutputBuilder pour rendre notre solution plus générique et facilement extensible à une situation où les données en entrée/sortie ne seraient pas en JSON uniquement. Nous avons également amélioré l'abstraction en enlevant le lien entre la classe Captain et les classes CheckpointsManager et Boat. Ainsi du point de vue du Captain les checkpoints n'existent plus et il est donc uniquement responsable de ce qui se passe à l'intérieur du bateau (actions des marins).

b) Description d'une refactorisation effectuée

La seconde refactorisation a été faite pour résoudre le problème de l'orientation dans les positions. En effet, les positions telles que données dans la spécification avaient des orientations. Cette orientation était inutile pour certaines opérations. Ainsi nous avons dans un premier temps créé un objet Point2D qui ne contenait que les attributs (x, y). Mais cette classe ne suffisait pas. De plus on devait écrire deux méthodes *distanceTo(),* une pour Position et une pour Point2D. Ainsi pour régler cela, nous avons utilisé une interface IPoint qui définit les méthodes ci-dessous.

```
double x (); double y ();
default double distanceTo(IPoint other) {
   return Math.hypot(other.x() - this.x(), other.y() - this.y());}
```

Les avantages de cette interface par rapport à d'autres solutions sont que :

- On définit la méthode *distanceTo()* une seule fois
- On a fait implémenter cette interface à tous les objets qui peuvent être vu comme des points (Position, Point2D, Boat, Checkpoint, …)
- Facilitation de la compréhension du code qui est de plus haut niveau. Comme *checkpoint.distanceTo(boat)*

4. Automatisation des tâches

La seule automatisation de tâches que nous avons effectuée a été l'utilisation de l'outil Travis CI. Il permet de réaliser des tests d'intégration en continu, de manière automatique, via GitHub. Il est possible de le paramétrer pour réaliser de nombreuses tâches différentes, mais pour ce projet nous nous sommes contentés de l'utiliser comme dernier vérificateur de commit : avec la commande « mvn clean test », nous avons pu faire en sorte que chaque commit ou pull request ne « cassait » pas la solution, et que tous les tests réussissaient. Nous avons également activé un « hook » Travis sur Discord (le moyen de communication que nous avons choisi) nous permettant de recevoir un message à chaque retour de Travis, et de savoir si le commit échoue ou pas.

Étude fonctionnelle et outillage additionnels

1. Stratégies et algorithmes

a) Limites des stratégies de jeu et algorithmes

Nous avons fait assez d'abstractions et de simplifications du problème pour pouvoir le résoudre plus facilement. Cela a bien sûr des impacts sur les scores en fonction de certaines situations.

- Les checkpoints sont vus comme des points : pour rechercher un chemin vers un checkpoint nous le considérons comme un point (son centre / position) et non une surface. Cela peut avoir comme conséquence dans certains cas extrêmes que l'on puisse ne pas trouver de chemin. Ou encore que nous perdions des tours qui pourraient être gagnés si on visait plutôt les bords des checkpoints.
- Aucune vérification n'est faite pour corriger la trajectoire dans le cas où le bateau rentre en collision. Il refait dont le même raisonnement tout le temps et potentiellement finit par être détruit.
- Afin d'éviter les collisions de récifs nous faisons du surplace à certains tours pour permettre au bateau de se mettre face à son objectif avant de se déplacer vers celui-ci.

b) Limites de l'organisation sur le projet

Globalement le projet a été assez bien géré puisque nous n'avons ni manqué de date butoir obligatoire, ni obtenu d'erreurs de runtime pour les livraisons. Aussi, comme mentionné précédemment, notre stratégie avait quelques cas limites qui n'étaient pas traités soit par manque de temps, soit car nous n'avions pas pensé à ces cas, soit parce que nous pensions que nous ne les rencontrerions pas (cas de la Week10 où la distance entre notre bateau et le point que l'on devait atteindre était inférieure à la distance minimale que l'on pouvait faire en activant une rame de chaque côté : résultat on restait sur place jusqu'à la fin).

2. Outils en plus

- Un outil de visualisation

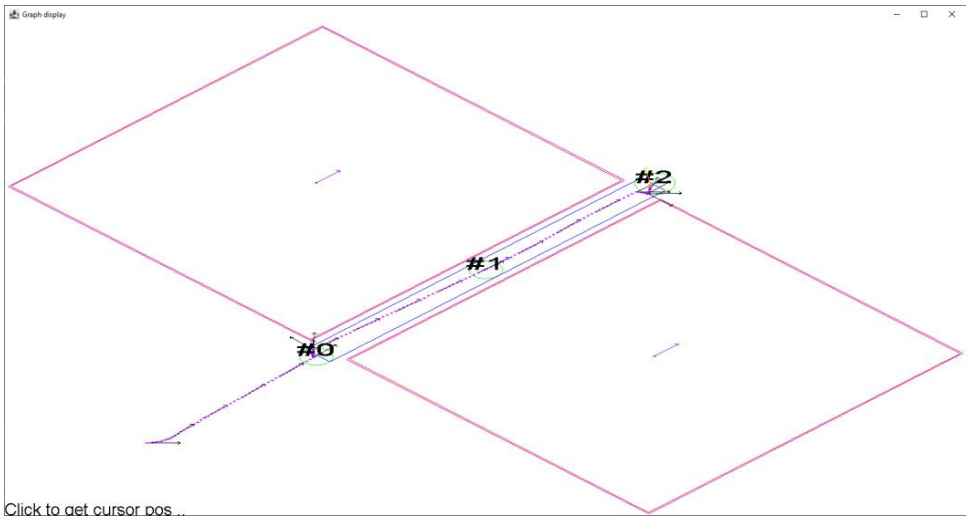
Dès les premières semaines, l'analyse de la carte via des fichiers JSON et des logs nous a paru peu efficace pour comprendre l'environnement du bateau et analyser les actions réalisées par celui-ci.

C'est pourquoi nous avons décidé de créer un **outil de visualisation** "visuals" qui affiche encore aujourd'hui, à partir des *output_bump.txt*, les différents éléments de la carte, ainsi que le parcours du bateau. Cet outil a connu plusieurs transformations pour répondre de plus en plus à nos besoins.

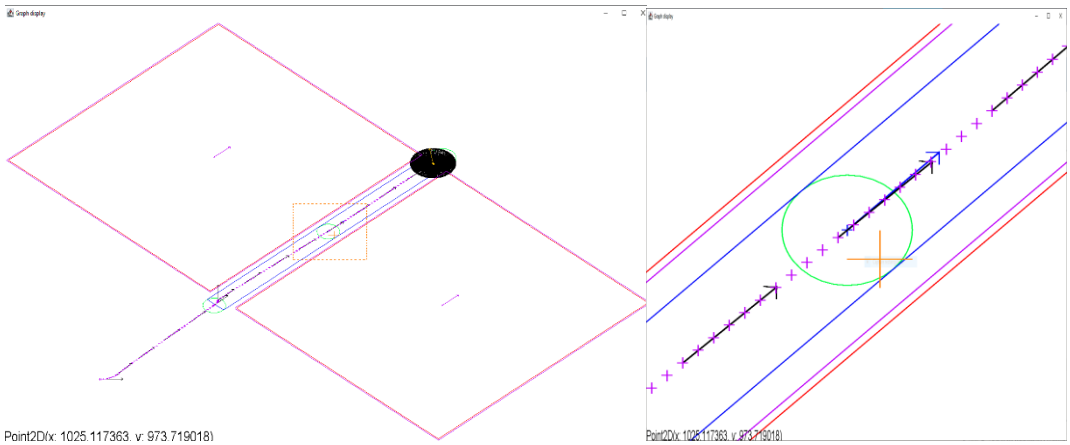
Sa première version était réalisée en python à l'aide de *Matplotlib*, seuls des points et vecteurs apparaissaient, aucune forme n 'était représentée (rectangle/cercle). Voir annexe « Illustrations de la 1ère version de l'outil de visualisation »

Nous sommes ensuite passés à une version en java, utilisant le Graphics2D, permettant d'afficher des formes de plus en plus complexes telles que les cercles, rectangles et enfin polygones. Cette version nous a permis de passer plus rapidement des données de notre programme à l'affichage, ce qui a été d'une grande aide pour visualiser les choix que notre navire prenait.

A terme, il était possible d'y afficher les courants, récifs, checkpoints, des chemins, les périmètres de sécurité autour des récifs, mais également des labels servant à numéroter les checkpoints ou les récifs pour les identifier facilement.



Aussi, il est possible d'y marquer une position par un simple clic pour en obtenir les coordonnées, ou encore de zoomer sur une zone pour y distinguer plus de détails.



- Un outil de simulation :

Ensuite, assez tard dans l'avancement du projet, alors que les visuels étaient achevés, nous avons réalisé un outil de simulation "runner" car nous tombions systématiquement à cours de crédits lors de nos essais sur le webrunner.

Ce runner a pour base le projet visuals, il permet en plus de lire les fichiers *game.json* et *initgame.json* (issus du webrunner et des résultats des releases) caractérisant à eux deux une course, afin de pouvoir avoir un aperçu des réactions de notre bateau en situation.

Bien qu'au début, il ne détectait les collisions qu'au points précis des étapes, nous avons fini par ajouter les collisions sur les chemins entre deux étapes en l'associant à une simple droite, afin de rendre cet outil de simulation plus fiable.

Cela dit, la simulation n'est pas parfaite. Elle n'a pas de condition d'arrêt mais réalise un nombre donné d'itérations. Elle ne prend pas en compte les dégâts des collisions, bien qu'elle empêche le

bateau de traverser les entités « solides ». Aussi, elle ne permet pas de lancer des parties en multijoueur.

C’est pourquoi cet outil nous a servis à vérifier plus fréquemment que nous terminions bien telle ou telle course mais n’a pas remplacé l’utilité du webrunner qui nous aura servis jusqu’à la dernière course.

Conclusion

Au cours de ce projet, nous avons principalement appris l'importance de définir une "branching strategy" adaptée au rythme et délai de livraisons. Nous avons appris l'importance de pouvoir se détacher du format d'entrée d'un problème pour pouvoir construire une solution extensible. Nous avons également appris que la relecture de code ne suffit pas pour valider du code. Relire et rajouter des tests pour les méthodes écrites par les coéquipiers est une pratique à adopter pour de futurs projets. Aussi nous avons compris que bien répartir le travail dans une équipe était important pour éviter que seule une personne maitrise certaines parties du code, donc la spécialisation.

Ensuite nous avons évidemment appris de nouvelles techniques pour améliorer la qualité du code, et l'efficacité de l'organisation du projet, comme Travis ou Sonarqube. Rétrospectivement, nous aurions pu pousser l'utilisation de Travis plus loin que les consignes de cours, notamment en l'associant à Sonar et Pitest pour mieux maitriser le projet et gagner du temps.

Lié à cela, nous avions dû trouver des solutions pour remédier aux code smells comme l'usage correct des Optionals, que l'on avait déjà découvert lors du semestre dernier.

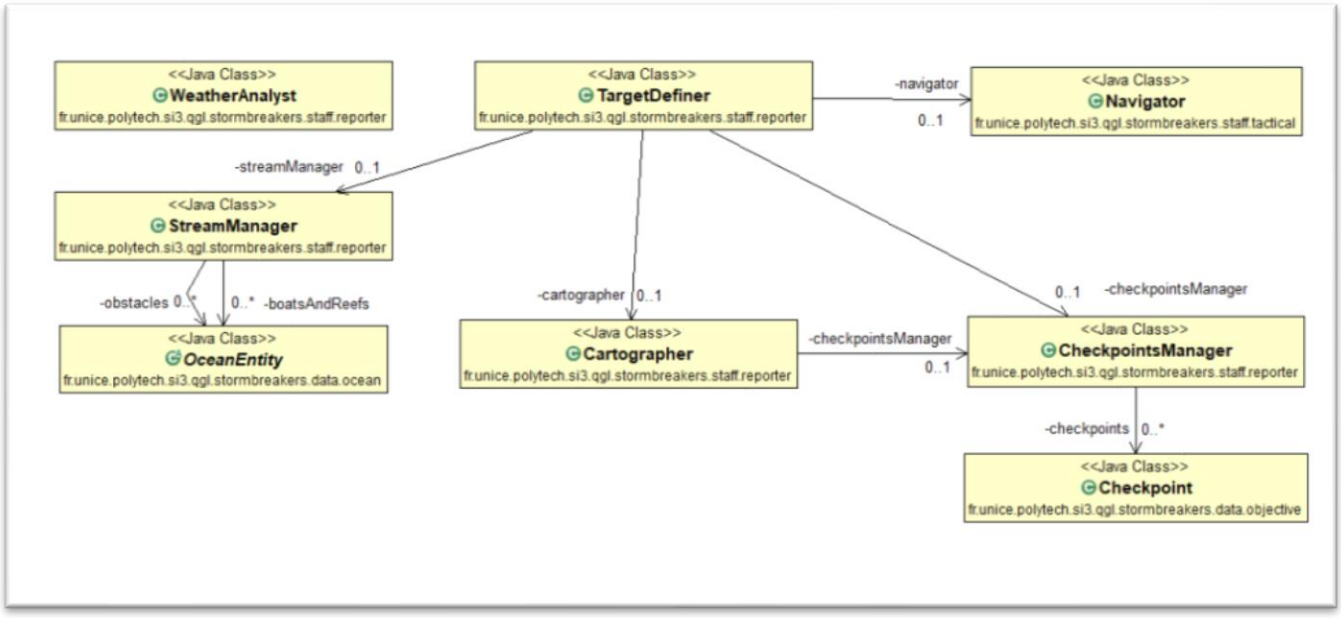
Nous avons utilisé au mieux de nombreuses notions provenant de plusieurs cours, des plus basiques comme les traitements de listes en POO, aux plus délicats comme le Design Pattern Observable (découvert en PS6 et en IHM). Bien entendu, toutes les notions apprises en QGL ont été appliquées, de manière plus ou moins importante.

La morale de ce projet fil rouge est que la communication est primordiale, il faut s'assurer que l'équipe entière soit sur la même longueur d'onde, et aussi vérifier les codes de chacun pour ne pas avoir de surprises à la fin.

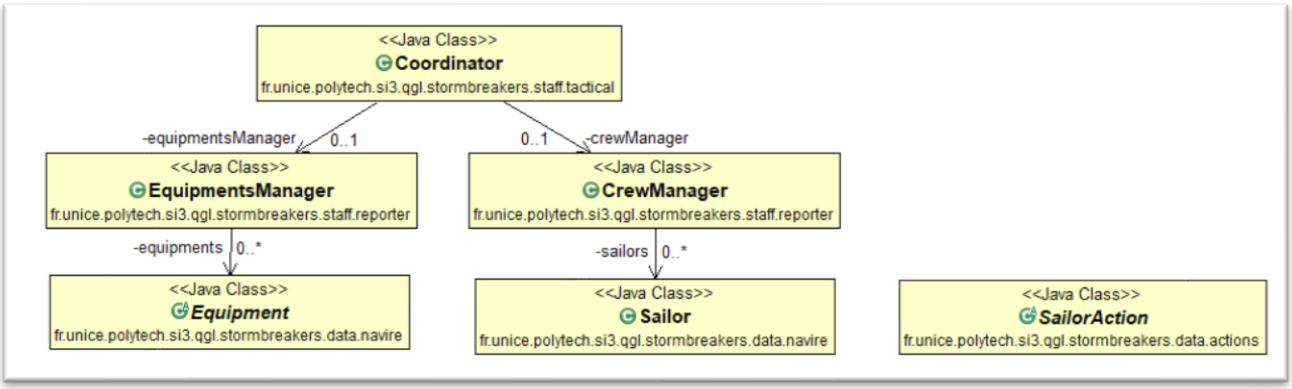
Aussi, améliorer la qualité de code d'un projet peut être beaucoup plus délicat qu'aux premiers abords.

Annexes

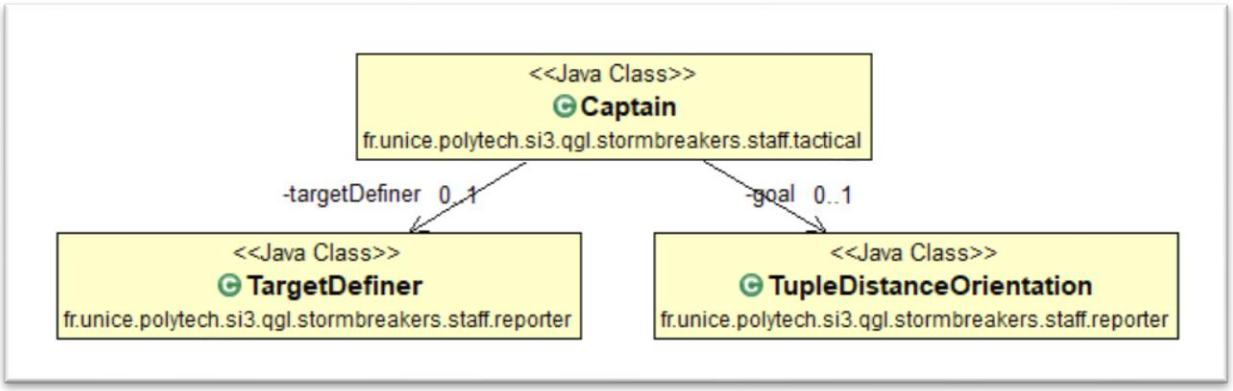
a) Diagrammes des classes principales de la solution



Classes concernant la gestion des éléments extérieurs au bateau



Classes concernant la gestion des éléments présents à bord du bateau



Classes servant à la détermination du cap à prendre

b) Exemple de modifications à effectuer pour supporter d'autres modes de jeu

Étape 1 : On extrait une interface **TargetDefiner**

```java
public interface TargetDefiner {

    Objectif defineNextTarget();

}
```

```java
public class RegattaTargetDefiner implements TargetDefiner {

    private CheckpointsManager checkpointsManager;

    private StreamManager streamManager;
    private Boat boat;
    private Navigator navigator;

    private Cartographer cartographer;

    public RegattaTargetDefiner(CheckpointsManager checkpointsManager, StreamManager streamManager, Boat boat,
            Navigator navigator, Cartographer cartographer) {

        this.checkpointsManager = checkpointsManager;
        this.streamManager = streamManager;
        this.boat = boat;
        this.navigator = navigator;
        this.cartographer = cartographer;
    }

    public RegattaTargetDefiner(CheckpointsManager checkpointsManager, StreamManager streamManager, Boat boat,
            Navigator navigator) {
```

On implémente cette interface pour chaque mode de jeu différent qu'on veut.

Étape 2 : On crée une interface **« Objectif »** qui jouera le rôle de visiteur

```java
public interface Objectif {
    List<SailorAction> executeObjectif(Captain captain);

}
```

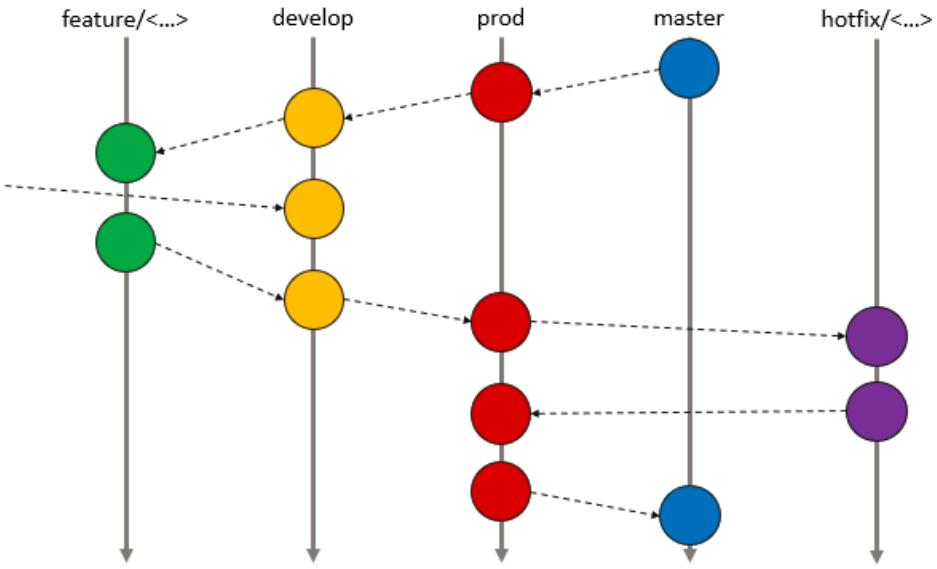
```java
public class ObjectifBattle implements Objectif {

    private double vitesse;
    private double orientationNeeded;
    private Object infosAboutCanonsToUse; //le type à définir
    // ... d'autres attributs potentiellement

    public ObjectifBattle(double vitesse, double orientationNeeded, Object infosAboutCanonsToUse) {
        this.vitesse = vitesse;
        this.orientationNeeded = orientationNeeded;
        this.infosAboutCanonsToUse = infosAboutCanonsToUse;
    }


    @Override
    public List<SailorAction> executeObjectif(Captain captain) {

        List<SailorAction> orientationActions=captain.actionsToOrientate(this.orientationNeeded, this.vitesse);
        double currentSpeed=captain.calculateSpeedFromOarsAction(orientationActions);
        List<SailorAction> speedActions=captain.adjustSpeed(this.vitesse,currentSpeed);
        List<SailorAction> canonsActions=captain.activateCanons(infosAboutCanonsToUse);
        return Utils.concatenate(orientationActions, speedActions, canonsActions);
    }
```

```java
public class ObjectifRegatta implements Objectif {

    private final TupleDistanceOrientation tuple;

    public ObjectifRegatta(TupleDistanceOrientation tuple) {
        this.tuple = tuple;
    }

    @Override
    public List<SailorAction> executeObjectif(Captain captain) {
        //... traitement intermédiaire possible ici

        List<SailorAction> orientationActions=captain.actionsToOrientate(tuple.getOrientation(), tuple.getDistance());
        double currentSpeed=captain.calculateSpeedFromOarsAction(orientationActions);
        List<SailorAction> speedActions=captain.adjustSpeed(tuple.getDistance(),currentSpeed);
        return Utils.concatenate(orientationActions, speedActions ) ;


    }
```

Étape 3 : On modifie le MainLinker pour instancier la bonne instance de TargetDefiner en fonction du mode de jeu.

```java
GameMode gameMode=this.parser.fetchGameMode(game);
if(gameMode.equals(GameMode.REGATTA)){
    targetDefiner=new RegattaTargetDefiner(checkpointsManager, streamManager, boat, navigator, cartographe
}
else if(gameMode.equals(GameMode.BATTLE)){
    targetDefiner = new BattleTargetDefiner();//pass constructor's parameters
}
// others gameModes  here
```

c) Illustration de la Git Strategy finale



d) Illustrations de la 1ère version de l'outil de visualisation