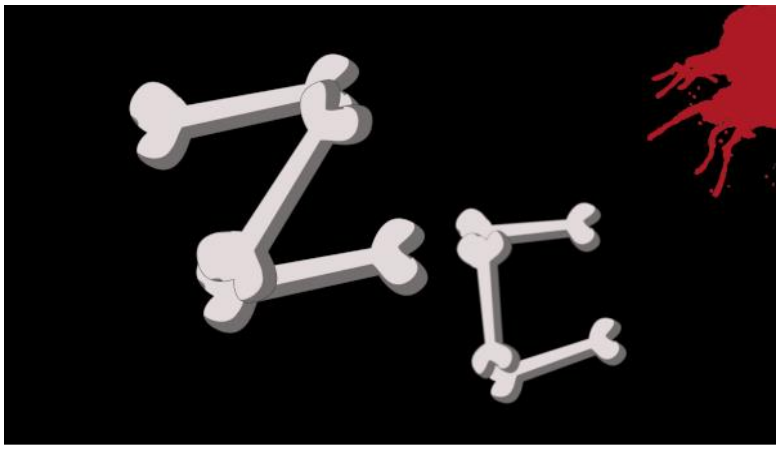
Rapport du projet QGL

Ze CommiT



Catégorie : FISE

Membres de l'équipage :

Bertolotto Loïc,

Delmotte Vincent,

Liebgott Joris,

Meulle Nathan,

Poueyto Clement.

2020

Sommaire

I. Introduction

Ce projet consiste à diriger un équipage de pirates dans le but de conquérir les mers. En effet chaque semaine, une nouvelle course, un nouveau défi, un nouvel aspect technique nous a été donné. Durant toute la durée du projet, des obstacles ont été mis sur notre route mais l'équipage de ZeCommit a su garder la tête hors de l'eau et dompter les océans. Dans ce rapport, nous vous expliquerons les secrets bien gardés de leur réussite. Alors accrochez-vous moussaillons, on met les voiles !

II. Description technique

A. Architecture

Lors de la prise de décision, notre capitaine effectue les opérations suivantes pour chaque nextRound : il génère une **table d'orientation** (*OrientationTable*) correspondant aux différents angles réalisables en fonction du nombre de marins, ainsi que les compositions de marins permettant de tourner avec différentes vitesses.

Avec 4 marins et 4 rames sur notre bateau, nous obtenons ainsi :

La table d'angles : [-1.5707963267948966, -1.0471975511965979, -0.5235987755982989, 0.0, 0.5235987755982988, 1.0471975511965976, 1.5707963267948966]

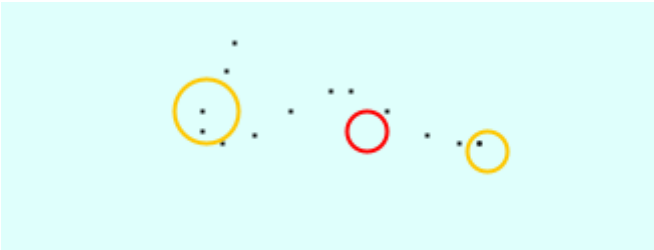
Et les compositions permettant de faire chacun de ces angles avec les rames disponibles sur le bateau (avec R le nombre de marins à droite et L le nombre de marins à gauche) :

- pour tourner à droite : [C{R=0, L=3}], [C{R=0, L=2}, C{R=1, L=3}], [C{R=0, L=1}, C{R=1, L=2}, C{R=2, L=3}],
- pour aller tout droit [C{R=1, L=1}, C{R=2, L=2}, C{R=3, L=3}],
- pour tourner à gauche : [C{R=1, L=0}, C{R=2, L=1}, C{R=3, L=2}], [C{R=2, L=0}, C{R=3, L=1}], [C{R=3, L=0}]

Le capitaine détermine ensuite l'**angle pour atteindre le prochain checkpoint** et choisit la composition la plus appropriée. La différence entre l'angle réalisé par les rameurs et l'angle souhaité sera ajustée ensuite grâce au gouvernail.

En cas de récif sur notre trajet, un checkpoint intermédiaire est positionné avant d'effectuer toutes ces étapes. Notre classe *Prédiciton* détermine la prochaine position du bateau et vérifie s'il y a collision (classe *Collision* qui découpe chaque polygone en triangle et vérifie si le bateau – assimilé à un point car très petit devant la taille des récifs– est dans ce triangle). Si c'est le cas, la classe *Calcul* génère différentes positions et choisit celle n'ayant pas d'obstacle sur son trajet pour **positionner un checkpoint intermédiaire**.

Exemple d'évitement d'un récif (en rouge) :



Le capitaine envoie ensuite sa décision au second (*CaptainMate*) qui va se charger d'effectuer les différentes actions : positionner les marins, orienter le gouvernail, donner l'ordre d'ouvrir la voile…

Chacune de ces étapes s'effectue dans un **ordre précis** : le *CaptainMate* demande ses instructions une par une au *Captain*. Ainsi 7 instructions sont demandées au *Captain* (déplacement des marins, checkpoint intermédiaire, rames à activer, angle du gouvernail, lever ou baisser la voile, utiliser ou non la vigie). Une fois l'information transmise, celle-ci est directement ajoutée à une liste d'actions qui sera ensuite sérialisée.

Toutes les informations utiles au *Captain* sont contenues dans un objet *Game* mis à jour à chaque tour.

Notre architecture est **extensible** : nous avons en effet un Capitaine qui implémente une interface (*CaptainInterface*). Chacune des méthodes de notre capitaine a été construite en vue de répondre au mieux au mode de jeu Regatta. Pour l'introduction d'un nouveau mode de jeu, il suffit de créer un nouveau capitaine qui implémente *CaptainInterface* et de réécrire les méthodes nécessaires à la prise de décisions.

L'utilisation de classes abstraites (*Shape* par exemple) nous permet de respecter le principe de "Liskov substitution" (L de SOLID). Ainsi l'ajout des polygones dans notre code n'a pas posé de problème et envisager une nouvelle forme serait facile à implémenter.

B. Interface Imposée

Pour le développement de ce projet, une interface nous a été imposée, à savoir ICockpit (avec les méthodes initGame et nextRound) ainsi que les formats Jsons d'entrée et de sortie entre notre code et le simulateur.

Cette dernière a influencé l'implémentation de notre code sous différents aspects. En effet, initGame vient nous **fournir les données** à l'initialisation de la simulation et nextRound nous envoie les **données actualisées** après chaque round de la simulation.

Au début, nous avions choisi de **recréer tous les objets** fournis via nextRound, afin de bien avoir les différents objets contenant les bonnes données. Pour cela nous avions deux classes de données - *InitGame* et *NextRound* - notre code prenant les données de InitGame lors du premier tour puis celles de NextRound, recréées à chaque tour.

Très vite cette implémentation nous **a posé problème**. Par la séparation de nos données de jeu, certaines n'étaient pas actualisées au cours de la simulation.

Par exemple, lors de notre gestion des interactions entre marins et entités : en effet, dans un souci de simplification, nous avons pris le parti d'attribuer dès le premier round les entités aux marins. Cette disposition est ensuite conservée tout au long de la simulation. Cela posait problème car le bateau et les entités étant recréés à chaque round, les marins étaient liés par référence d'objets avec des entités n'étant plus utilisées, générant un certain nombre d'erreurs d'exécution.

Nous avons donc décidé de créer une **classe de données unique**, *Game*, permettant de partager les données entre les différentes classes, et de venir actualiser les attributs des différentes classes qui étaient modifiées en cours de simulation.

Dans le but d'accorder notre code avec les spécificités techniques, nous avons dû créer beaucoup de classes dites "data", comme les classes d'entités et d'actions, ce qui est d'usage à éviter.

Ces spécifications nous ont aussi poussées à devoir **créer des énumérations**, afin de pouvoir typer de manière propre les différents objets d'actions, entités ou formes, et ainsi pouvoir facilement re-typer les différents objets créés à partir de l'instance de la classe mère abstraite, permettant ainsi de nous éviter de créer des conditions faisant appel à l'instruction "instanceof" ou à la comparaison de chaînes de caractères.

III. Application des concepts vus en cours

IV. Branching Stratégie

Après avoir vu le concept en cours, nous avons implémenté une *branching stratégie* au projet. Nous avons adopté une méthode qui associe **une fonctionnalité par branche**.

Dans un premier temps, nous avions plusieurs fonctionnalités à développer, ainsi une branche par fonctionnalité a été créée : une pour le *désérialiser*, une pour l'outil de *visualisation,* une pour le *refactoring*, une pour l'ajout du gouvernail… A l'issue de l'implémentation de chaque fonctionnalité, une pull request était proposée à l'équipe, le code été examiné et ensuite mis sur master avec un merge s'il était correctement testé.

Cette stratégie, qui nous semble être la plus adaptée pour notre problème, a permis d'augmenter la qualité des rendus. Si des bugs persistaient dans le programme et n'étaient pas résolus, il était possible de conserver une version plus ancienne du code et ainsi passer la course sans erreur. Ainsi on évite de régresser et d'avoir des erreurs lors des courses dûes à l'ajout d'une fonctionnalité.

Ainsi, la mise en place de ce branching stratégie nous a permis d'aborder une **approche professionnelle** du développement. En effet, nous avons appris à utiliser les *pull request* et à se familiariser avec une démarche qualité.

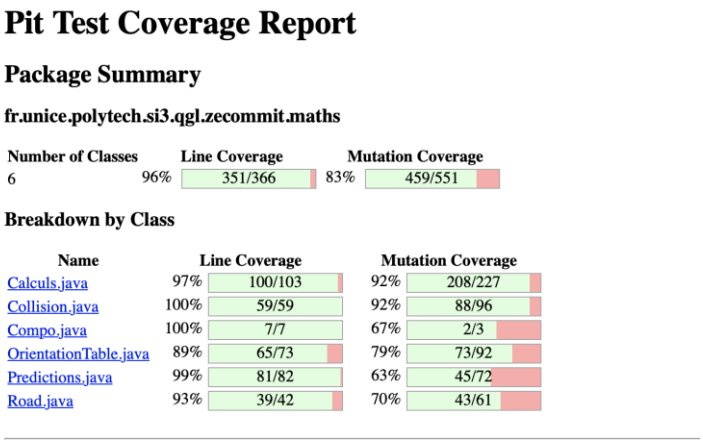
V. Qualité du code

Nous pensons avoir une bonne qualité de code. En effet, l'ensemble de notre code est testé à **81,2%** (package moteur et visualisation exclus) avec **153 tests unitaires** permettant de s'assurer du **bon fonctionnement de chacune de nos méthodes**.

Nous avons également ajouté **12 tests d'intégration.** Chacun de ces tests lance une course entière, correspondant à une simulation différente d'une ancienne course effectuée, et vérifie qu'il n'y a pas d'erreur (null Pointer, course non finie, collision avec un récif par exemple) dans le but de **ne pas régresser dans notre stratégie** au fil de l'avancement du projet.

Les tests que nous avons construits **résistent aux mutations** générées par PIT-Test. Nous obtenons ainsi 61% de mutation coverage (package moteur et visualisation inclus) et pour les packages les plus importants (i.e. ceux effectuant les calculs et prises de décisions) plus de 80%.

Par exemple le package Maths à une **couverture de test de 96%** et **83% de *mutation coverage*** (cf. image ci-dessous pour le détail). Cela nous indique, que les tests réalisés sont fiables et testent correctement nos méthodes de calcul. Nous pouvons ainsi être confiant dans le calcul des orientations par exemple.



Pit Test Coverage Report

Package Summary

fr.unice.polytech.si3.qgl.zecommit.maths

Number of Classes	Line Coverage		Mutation Coverage	
6	96%	351/366	83%	459/551

Breakdown by Class

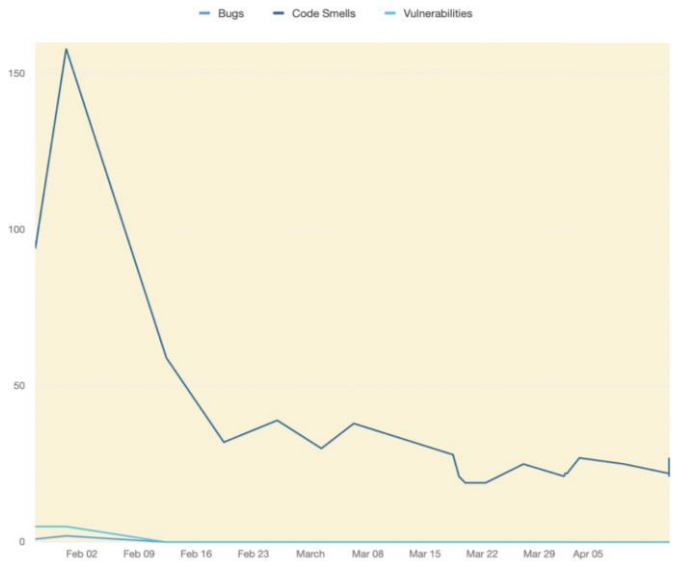
Name	Line Coverage		Mutation Coverage	
Calculs.java	97%	100/103	92%	208/227
Collision.java	100%	59/59	92%	88/96
Compo.java	100%	7/7	67%	2/3
OrientationTable.java	89%	65/73	79%	73/92
Predictions.java	99%	81/82	63%	45/72
Road.java	93%	39/42	70%	43/61

Pourcentage de mutation coverage sur différentes classes

Notre code est de qualité puisqu'il comporte peu de code smells et **pas de bug** : nous avons **20 code smells** soit 2h de dette ce qui est très peu. De plus notre code ne comporte **pas de duplication de code**.

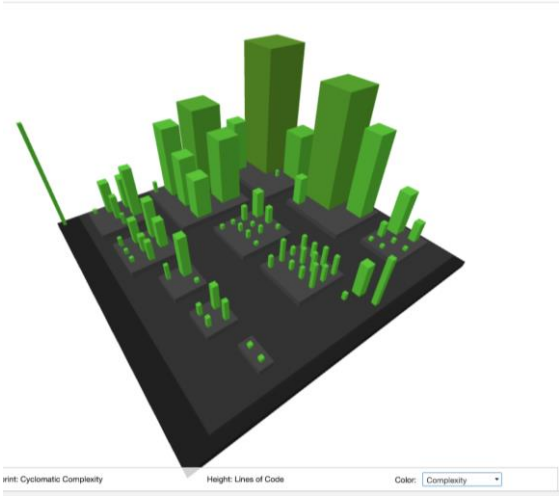
Une qualité moindre aurait rendu notre code plus **difficilement compréhensible**, avec probablement **plus d'erreurs** : par exemple, sonar nous a permis d'identifier rapidement les passages de code sensibles aux erreurs : cas d'un dénominateur nul non pris en compte, variable non initialisée, existence d'un objet *Optional* non vérifiée...

C'est pourquoi nous avons cherché tout au long de ce projet à réduire le nombre de code smells et éliminer tout bug comme le montre le graphique sonar suivant :



Évolution des code Smells en fonction du temps

Nous avons utilisé l'outil de Sonar permettant de visualiser rapidement la complexité de chaque classe. L'une de nos classes paraît plus complexe que les autres : il s'agit de la classe *Captain*. En effet sa complexité est plus élevée puisque nous avons fait le choix de centraliser toutes les prises de décisions au sein de cette classe.

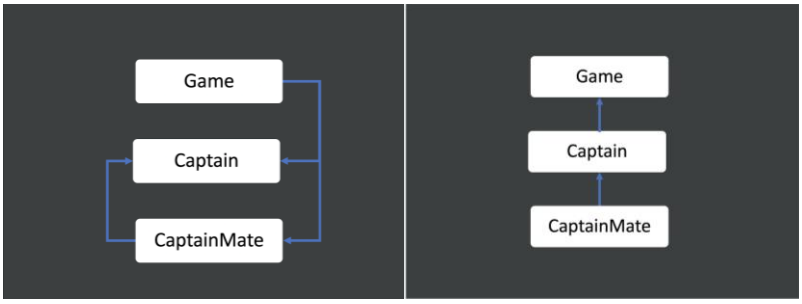


SoftVis3D de notre code avec Sonar

VI. Refactoring

Durant le projet, nous avons pris la décision de “refactor” notre code à 3 reprises afin de respecter le principe SOLID. Ces refactorings étaient nécessaires car l’implémentation de méthodes et l’amélioration de notre stratégie devenaient de plus en plus complexe.

Dans l’exemple ci-dessous la classe *Game* était contenue par les classes *Captain* et *CaptainMate*. Captain permet de prendre des décisions et *CaptainMate* de les effectuer, cette dernière n’avait donc pas besoin de connaître l’état du jeu pour réaliser les actions. Nous avons donc réécrit l’architecture de ces trois classes afin d'empêcher la duplication de code, de **limiter les responsabilités** (Single Responsability de SOLID) au sein d’une même classe et d’éviter les dépendances entre elles.



Avant refactor / Après refactor

VII. Automatisation

L’utilisation de l’outil Travis nous a permis d’**automatiser la vérification** de qualité de notre code en exécutant nos tests à chaque commit : en relançant l’ensemble de nos tests, il permet de vérifier que la fonctionnalité ajoutée ne casse pas nos tests d’intégration par exemple. Ainsi nous pouvons travailler avec une démarche d’amélioration et non de modification perpétuelle du code (principe Open/Closed de SOLID).

Travis nous a également permis de retracer l'**origine de certains bugs** mais aussi d'assurer que notre programme s'exécute correctement à chaque modification sur GitHub.

Nous avons ainsi pu être dans une pratique s'approchant du "**DevOps**" : chaque semaine nous avons planifié les nouvelles fonctionnalités à travers nos milestones, nous testions ensuite notre code au plus tôt avec Travis après chaque commit. Cette intégration continue nous a permis de gagner en efficacité.

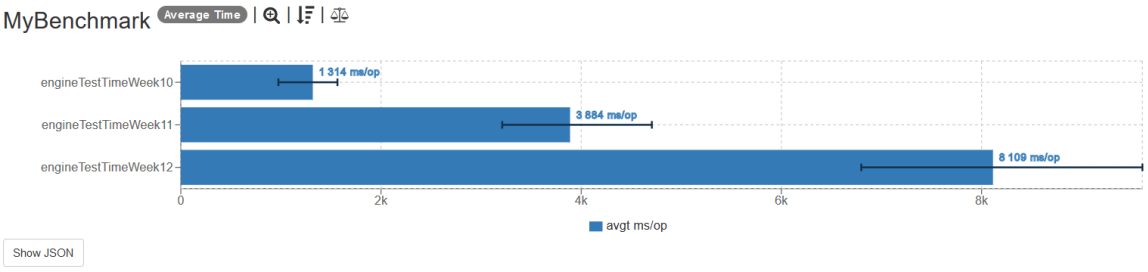
VIII. Étude fonctionnelle et outillage additionnels

A. Stratégie

Nous avons mis en place **une seule stratégie**. Nous aurions cependant pu en mettre plusieurs en place et utiliser notre moteur afin d'adapter pour chacune des courses notre stratégie et choisir la meilleure.

Prenons un exemple simple : notre capitaine privilégie la gauche à la droite pour éviter un récif. Cela pose problème dans la course "The Snake" puisque notre bateau ne rentre pas dans le passage sur sa droite. Ainsi nous aurions pu utiliser notre moteur pour adapter notre stratégie : au début de la course, on lance notre moteur et on constate que privilégier la gauche n'aboutit pas. On change alors de stratégie et nous relançons notre moteur. Si cette nouvelle stratégie fonctionne notre Capitaine la choisit et la met en place.

(Pourquoi cela n'a pas été fait ? Effectuer plusieurs simulations avec notre moteur prend du temps et aurait généré des Timeouts : suite à une **étude benchmarking** nous avons constaté que faire tourner notre moteur pour une course de 250 rounds prend en moyenne 5 secondes, envisager une ou deux simulations par round afin de s'adapter à chaque récif n'est donc pas possible durant le déroulement de la course dont le timeout est de 30 secondes)



Résultat de notre Benchmark (week10 : 43 rounds, week11 : 253 rounds, week12 : 273 rounds)

B. Outils additionnels

1. Moteur

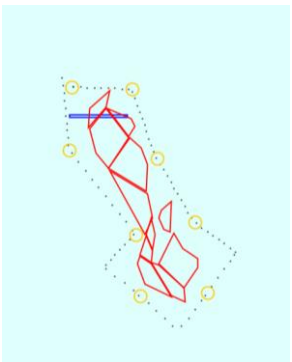
Afin d'avoir un aperçu du comportement de notre bateau dans toutes les situations, nous avons décidé de recréer l'outil de simulation d'une course de Frégate en s'appuyant sur les consignes de l'arbitre en se rapprochant le plus possible de la simulation de base.

L'objectif principal était de comprendre nos erreurs de déplacement du bateau mais aussi nos erreurs de déplacement des marins au sein du navire et donc de permettre une **amélioration de la stratégie** de navigation.

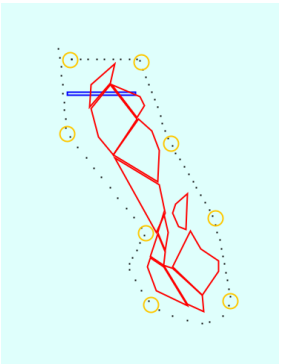
2. Outil de visualisation de la carte (Visualisation Map)

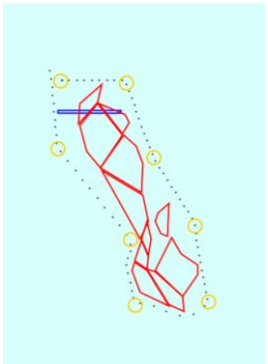
Par la suite nous avons mis en place une visualisation de la simulation pour avoir une **vision globale** et claire de notre parcours, ce qui n'est pas possible avec seulement l'information des coordonnées. Nous nous sommes aperçus que notre stratégie pouvait être peaufinée lors de l'évitement des récifs pour gagner du temps.

Amélioration du parcours de l'île de Ré :



Avant





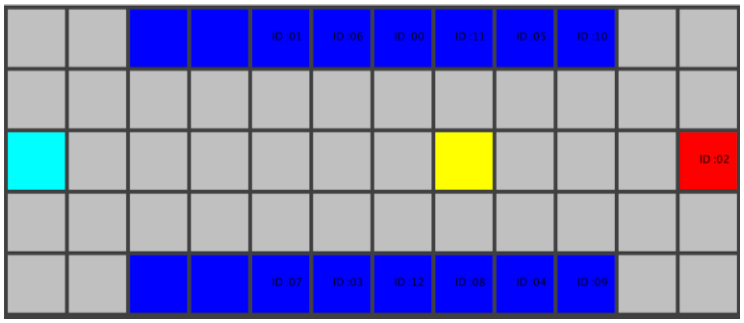
Après

Pour créer une nouvelle course, notre outil prend en paramètre des objets que nous initialisons dans une classe *EngineSettings* dédiée *(*qui elle-même hérite d'une classe mère *EngineSettings*). Un Json est ensuite généré par le moteur pour lancer la partie.
Ce choix de conception nous permet de **générer facilement de nouvelles simulations** : il est en effet plus bien plus facile de créer des objets qu'écrire un Json. Nous avons ainsi pu construire des simulations mettant notre bateau dans des situations bien particulières afin d'améliorer notre stratégie (*EngineSettingsWeek9ZigZag* par exemple).

Dans le cadre d'une **amélioration possible de cet outil**, nous aurions pu développer une fonctionnalité permettant de lire les fichiers de sortie .bump afin de visualiser nos courses réalisées chaque semaine pour vérifier que notre bateau se comporte de la même manière dans notre simulation.

3. Outil de visualisation du Deck

Cet outil permet d'avoir un **aperçu du bateau** à un certain round, c'est à dire de l'emplacement des différents marins avec leurs identifiants respectifs, des rames représentées en bleu foncé, du gouvernail en rouge, des voiles en jaune et de la vigie en cyan. Il permet de détecter les mauvais placements et d'améliorer la stratégie de gestion des marins.



Nous aurions pu améliorer l'outil en proposant de naviguer de round en round sans relancer l'exécution du programme pour observer l'évolution plus rapidement. Nous avons cependant décidé de ne pas continuer le développement de la visualisation du Deck. Au fur et à mesure de l'avancement du projet le positionnement des marins ne posait plus de problème, nous nous sommes donc concentrés sur d'autres parties du projet.

IX. Conclusion

Grâce à ce projet, de **nombreux concepts** ont pu être abordés et ainsi de nombreux problèmes appréhendés. Tous les outils présentés ont été implémentés dans notre projet. On peut nommer Sonar, PitTest, Travis ou encore Maven.

Tous ces outils nous ont permis de mettre l'accent sur la **qualité** : Tests pertinents (PitTest), intégration continue (Travis), visualisation des zones de dangers dans le code et coverage (Sonar). Ces outils facilitent grandement la gestion des nombreuses lignes de code qui viennent alimenter un projet qui de semaine en semaine a pris de l'importance. De plus nous avons commenté et expliqué le rôle de la plupart de nos méthodes afin d'obtenir une Java Doc la plus claire possible dans le but d'augmenter la **lisibilité** du code. Les **principes SOLID** ont été une ligne directive. En effet, les refactorings ont été réalisés dans le but de coller au plus près de ces principes de développement permettant une qualité accrue du code.

Ce projet nous a enseigné plusieurs aspects de la **programmation en équipe**. Dans un premier temps un projet peut prendre rapidement de l'ampleur et toutes les fonctionnalités ne sont pas développées par une seule personne rendant l'appréhension du code difficile. Heureusement il existe des outils nous facilitant la compréhension et il est important de les utiliser pour gagner du temps mais aussi gagner en qualité.
Il faut être très vigilant lors de l'implémentation de nouvelles fonctionnalités. Nous avons plusieurs fois expérimenté l'ajout de fonctionnalités qui empêche la compilation du code et ainsi anéanti tous les efforts fournis avant si le bug n'est pas résolu.

Tous ces outils, utilisés avec une optique d'amélioration continue, nous ont permis d'augmenter la **fiabilité** du code augmentant ainsi la qualité.

Ainsi la constante recherche de qualité, de lisibilité, de compréhensibilité, de clarté, nous a permis de construire un code solide, testable, extensible.

Ainsi l'équipage ZeCommit a bravé les mers durant ces quelques mois d'expédition, relevant les défis tout en améliorant son bateau et en affûtant la stratégie de son capitaine. Malheureusement, un marin est tombé à l'eau durant le voyage mais l'équipe a fait face ! On dit qu'aujourd'hui ils sont toujours sur les traces de Barbe Noire…