Rapport de projet
Qualité Génie Logiciel

Équipe : Qualidad
ID : qualidad

Membres :

- MANUEL Enzo
- BOUZOUBAA Fahde
- GARROT Thibault
- DIJOUX Robin

POLYTECH
NICE-SOPHIA

Sommaire

Description technique

Architecture du projet

Tout d'abord, il faut savoir que la classe Cockpit appelle directement les méthodes initGame et nextRound de notre classe CrewChief. A la création d'une course, nous récupérons les données fournies par initGame pour créer un groupe d'entités (rames, gouvernail, vigie...) pour chaque marin présent sur le bateau.

Concernant la gestion de décision à chaque tour, nous avons 2 phases principales. La première consiste à récupérer le prochain checkpoint à atteindre, à créer un quadrillage pour établir le chemin le plus court pour arriver au checkpoint grâce à la méthode de Dijsktra, et à optimiser ce chemin en enlevant des points d'arrêts inutiles et en mesurant la position la plus avantageuse à viser pour valider le checkpoint et repartir de plus belles vers le checkpoint suivant (s'il y en a un).

Une fois cette étape réalisée, nous passons à la deuxième phase. Nous devons donner les ordres d'actions que les marins doivent effectuer pour arriver au plus vite au point visé. Dans un premier temps, nous allons vérifier les actions possibles que nous avons sur le bateau (la vigie, les voiles, le gouvernail...) de manière bien définie. Nous avons ajouté une condition supplémentaire pour utiliser la vigie, qui est de l'utiliser uniquement si nous nous sommes déplacés de plus de 1000 unités depuis la dernière utilisation de celle-ci. On regarde la disponibilité de chaque type d'action dans un ordre bien défini, qui est la vigie, la voile, le gouvernail et enfin les rames. Les actions de ramer sont attribuées et définies en fonction du nombre de marins restants.

En règle générale, une architecture qui est adaptable est une architecture qui est bien définie et bien organisée. Tout au long du projet, notre architecture a été réorganisée plusieurs fois afin de bien déléguer les responsabilités aux classes concernées, et ne pas trop les encombrer. De plus, nous avons justement pu tester la flexibilité de notre architecture lors de l'implémentation du Dijsktra, qui est une stratégie que nous avons développée tardivement, après avoir atteint les limites de notre système d'évitement d'obstacles. Nous avons pu établir la stratégie du Dijsktra indépendamment du reste du projet, et une fois ce code stable avec les tests effectués, nous l'avons connecté à l'architecture principale via la classe CrewChief en ajoutant seulement quelques conditions à l'usage de cette stratégie. En nous basant sur cette étape de notre projet, nous pensons effectivement que notre architecture est extensible, et que l'ajout d'un nouveau mode de jeu ou d'une nouvelle fonctionnalité ne devrait pas être problématique.

Concernant nos choix de conception qui ont influencé notre architecture, nous pouvons mentionner notre package actions qui présente une forte complexité. Nous avons décidé d'y mettre toute les méthodes calculatoires (ou en tout cas une grosse partie), la répartition des équipements, des marins, etc. Dans la même lignée, nous avons décidé de créer un package path qui contient toutes les classes permettant de réaliser notre pathfinding (Graph pour le traçage du quadrillage, Node pour représenter les nœuds du graph, et Dijkstra pour la recherche du chemin le plus court).

Impact des contraintes imposées sur notre architecture

La première contrainte à laquelle nous avons été confronté est celle de la compréhension rapide du fonctionnement de Jackson dans Java. Les consignes du projet étant déjà très complètes, il y a eu un flux de connaissances importantes, ce qui a été compliqué à digérer pour un début de projet. Néanmoins nous avons plutôt bien implémenté cette partie en tournant cette contrainte à notre avantage. Toute les classes qui allaient générer des objets via json ont directement été implémentées, même les classes non utilisées au début du projet. Ceci nous a permis d'avoir une base pour notre architecture, et nous ne nous en sommes occupés qu'une seule fois pendant le module.

L'autre contrainte concerne les tests. Les méthodes initGame et nextRound recevant toutes les deux des chaînes de caractères json en paramètre sont assez compliquées à tester intégralement. La méthode initGame permet d'attribuer les données aux attributs de la classe CrewChief et ne retourne rien. Nous aurions pu vérifier que les attributs récupèrent bien les bonnes données, mais nous aurions dû passer ces variables en public avec des getters ou les passer en statique, ce qui n'est pas très propre.

Application des concepts vus en cours

Git et branching strategy

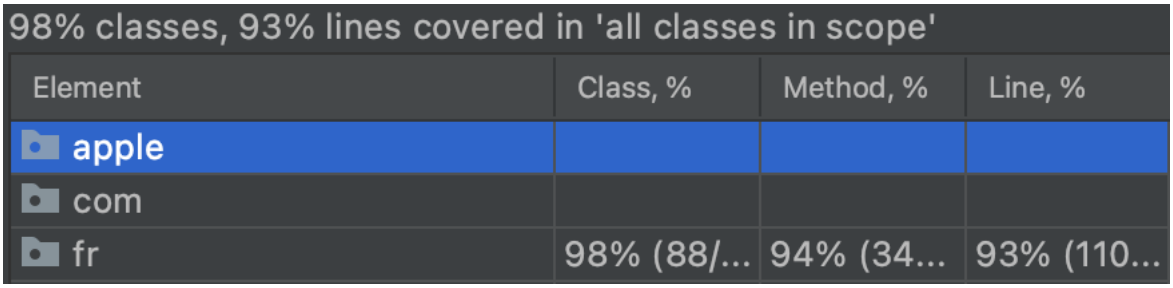
Nous avons opté pour la stratégie GitFlow car elle nous aide à mieux nous organiser et développer de manière agile. En effet, elle segmente bien la partie stable et la partie en développement du projet. Cette stratégie est très adaptée pour une production avec rendus réguliers : dans notre module, nous avions un rendu hebdomadaire à fournir et c'est pour cela que nous avons choisi cette stratégie.

Nous avons donc notre branche master qui est actualisée uniquement via une branche develop. Chaque nouvelle feature est implémentée dans une branche qui lui est dédiée, ouverte à partir de develop, afin de limiter les conflits lors de merge. Une fois cette feature aboutie, on la merge dans develop. Cette dernière est une branche qui n'est pas toujours stable, et qui regroupe toutes les fonctionnalités codées et nous aide à nous assurer que la version du code est stable avant de la merge dans master.

Au début du projet, nous n'avions qu'une seule branche master qui contenait tout le code. Puis nous avons amélioré notre branching stratégie en rajoutant develop et les branches feature qui correspondent à chaque nouvelle fonctionnalité. Plus tard, nous nous sommes rendu compte que lorsque nous voulions optimiser ou revenir sur le code d'une Milestone (qui correspond à nos WEEK), nous modifions potentiellement une partie du code qui était stable et fonctionnelle pour une précédente WEEK. Pour y remédier nous avons créé des branches pour les Milestone, des branches qui contenaient les nouvelles versions hebdomadaires. Cela nous a permis de créer un vrai versionning de notre logiciel, clair et facilement maintenable.

Qualité du code

Tout au long du développement de notre projet, nous nous sommes efforcés de valoriser la qualité du code, au même titre que son efficacité.
A chaque ajout de nouvelle feature, nous avons essayé de conserver un bon niveau de qualité. Pour cela, nous testions toutes les méthodes (ou presque).
Nous utilisons les tests unitaires JUnit exécutés sous maven afin de vérifier que le produit codé fonctionne comme prévu, selon des scénarios prédéfinis et représentatifs. Nous avons un coverage (couverture de test du code) de 98% pour nos classes et de 94% pour nos méthodes, ce qui à nos yeux est assez satisfaisant.



Couverture des tests sur Intellij

En plus de ces tests unitaires, nous avons utilisé des outils additionnels permettant d'avoir une meilleure efficacité et un suivi plus précis de la qualité.

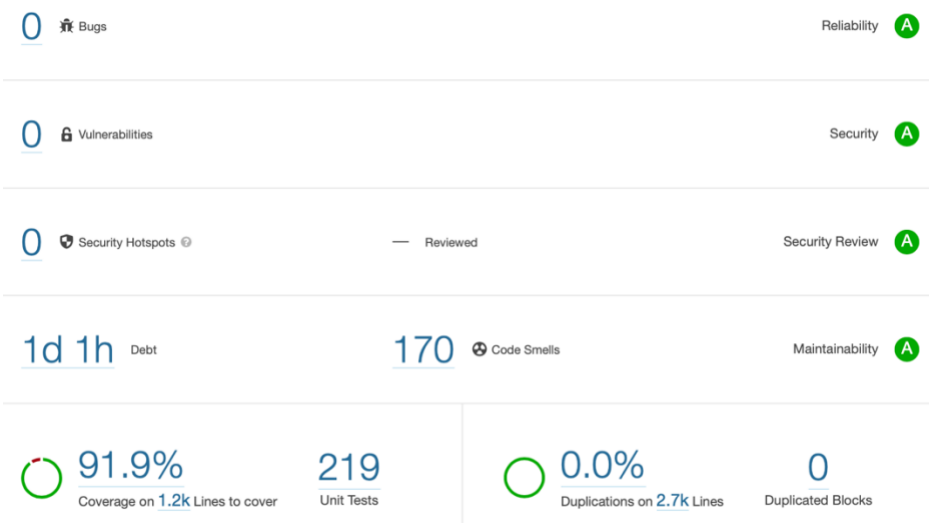
Parmi ces outils, nous pouvons citer PiTest. PiTest est un plugin maven qui fait évoluer le code exécuté lors des tests grâce à une stratégie de « tests mutants ». Il applique de légères modifications dans nos classes de test pour voir comment ils réagissent aux changements. Une fois ces tests mutants exécutés, PiTest nous indique quels sont les tests qui ont échoué. Il s'agit d'un bon indicateur de la qualité de nos tests. Nous « tuons » 84% de ces mutants (ce qui signifie que 84% des tests mutants réussissent). Là aussi nous sommes satisfaits.

```
================================================================================
- Statistics
================================================================================
>> Generated 837 mutations Killed 701 (84%)
>> Ran 1790 tests (2.14 tests per mutation)
```

Statistiques générés par PiTest pour les tests mutants

Évidemment la seule lecture de ces statistiques n'est pas suffisante, l'outil permet juste de mettre en évidence des tests qui ont flanché à un moment donné, et il est important de voir en précision ces mutants afin de déterminer les vrais négatifs et agir en conséquence.

Afin d'avoir un suivi plus précis de la qualité du code, nous avons également mis en place l'outil SonarQube. Sonar est un analyseur de code statique, un autre plugin maven qui analyse le code et trouve des « code smells » (du mauvais code), des bogues et fournit de nombreuses autres informations intéressantes à travers des métriques.



Métriques fournies par SonarQube au 18/05/2021

Tous nos indicateurs sont au vert, malgré le fait qu'il reste 170 « code smells » que nous aurions surement pu améliorer si nous avions eu plus de temps.

En prenant un peu de recul, nous nous rendons compte que la prise en charge de la qualité et sa mesure ont été assez contraignantes dans la mesure où nous partions d'un code qui n'étais pas optimisé et « propre ». Semaine après semaine, nous avons découvert de

nouveaux outils (PiTest, SonarQube, etc…) qui une fois mis en place, nous montraient que nous sous-estimions l'importance des tests unitaires, de la mise en pratique du principe SOLID, etc. Nous avons dû repasser plusieurs fois sur des parties du code mal testées et/ou mal implémentées afin d'affiner ce que nous avions écrit. Effectuer cela tout en développant une solution fonctionnelle pour passer les courses hebdomadaires prenait du temps, mais nous a permis de mesurer l'importance de la qualité du code dans le développement d'une solution.

Refactoring

Au cours du développement, nous n'avons pas procédé à un refactoring majeur de notre base de code, mais plutôt à plusieurs léger refactoring et correctifs. Chacun de ces correctifs avaient pour but de réduire les dépendances entre les différentes classes (transmettre des objets de complexité inférieure ou alors des objets ne servant qu'à la transmission d'informations), ajuster les responsabilités au fur et à mesure des fonctionnalités que nous devions implémenter (par exemple créer une nouvelle classe dédiée à une fonctionnalité qui se complexifie, ou la découper en plusieurs méthodes) ou simplement déplacer une méthode dans une autre classe quand nous jugions qu'elle n'avait plus sa place dans sa classe d'origine.

Automatisation

Nous avons également pu automatiser certaines tâches grâce à la fonctionnalité GitHub Action de Git. Nous avons mis en place une action qui nous permet d'être notifiés sur Slack lorsqu'un commit est poussé sur notre branche Master ou develop (cf. Git et branching strategy pour plus d'info sur ces branches). Nous avons fait une utilisation assez sommaire de cette fonctionnalité d'automatisation, mais nous aurions pu aller bien plus loin, notamment au profit de la qualité du code. Nous aurions pu par exemple mettre à jour SonarQube (cf. Qualité du code) dès que de nouvelles lignes de code sont push sur le repository, pourquoi pas également lancer des tests par mutation (même si cette opération peut prendre un peu plus de temps). Le champ des possibles est large, et nous regrettons de n'avoir pu, faute de temps, mettre en place ces actions très pratiques qui aurait pu nous faire gagner en qualité.

Étude fonctionnelle et outillage additionnel

Étude fonctionnelle

Afin d'augmenter nos chances de victoires, nous avons mis en place des stratégies à différentes étapes lors de la course.

Au début de la partie, lorsque notre programme prend connaissance des marins et des équipements présents sur le bateau, nous procédons à un regroupement de ces derniers afin de créer des lots. Ces lots d'équipements sont créés stratégiquement afin d'optimiser au cours de la navigation les actions des marins. Ainsi, nous regroupons prioritairement le gouvernail, la voile et la vigie car d'une part le gouvernail est un élément essentiel grâce à la maniabilité qu'il nous apporte, et d'autre part la voile et la vigie ne sont utilisées que très peu de fois (explications plus bas). Il est donc important pour un marin d'être la plupart du temps au gouvernail. Ensuite viennent les rames, qui sont identifiées et regroupées selon leur côté sur le navire : une rame à tribord couplée à une rame à bâbord non lointaine. Cela permettra au marin qui s'est vu attribuer ce lot de pouvoir jongler en un seul tour entre rame à bâbord et à tribord. Une fois ces lots créés, ils sont attribués aux marins et cela ne changera pas jusqu'à la fin de la partie.

Nous utilisons la vigie (et les voiles, avant de suspendre son usage pour des raisons expliquées plus bas) à des moments bien précis. Comme expliqué dans la partie architecture du projet, la vigie est utilisée uniquement lorsque nous nous sommes éloignés de plus de 1000 unités depuis la dernière utilisation de celle-ci. Nous avons estimé que la vue de 5000 offerte par la vigie nous permettait d'espacer dans le temps son utilisation. Pour la voile, nous l'ouvrions uniquement lorsque nous avions un vent de dos, et nous la fermions lorsque nous avions un vent de face.

Autre point que nous avons mis en place pour gagner du temps lors des courses, pour chaque checkpoint visé, nous ne visions pas le centre de celui-ci, mais un point qui d'une part nous permettait de valider ce checkpoint, mais également d'être dans la meilleure position et orientation pour repartir rapidement vers le prochain checkpoint. Grâce à cette stratégie, nous arrivons à plein régime plus rapidement.

Pendant la majeure partie du module, notre stratégie de navigation reposait sur notre manœuvre d'évitement. Le principe est le suivant : nous traçons un rectangle de la même largeur que celle de notre navire, et de longueur la distance entre notre position (celle du navire) et le prochain checkpoint que nous visons : c'est le chemin que nous allons prendre. Nous appliquons notre algorithme de collision entre ce rectangle et chacun des récifs visibles. Si un récif est en collision avec notre chemin, nous créons un point intermédiaire à viser afin d'esquiver ce récif et repartir en direction du checkpoint. Ce point se situe sur la droite qui passe par le centre du récif et sa projection sur notre chemin. Cette manœuvre d'évitement était efficace et nous a permis de passer de nombreuses courses. Mais des parcours plus compliqués ont révélé les limites de cette stratégie. Nous avons alors dû opter pour une

stratégie de pathfinding, que nous avons essayé d'implémenter grâce à l'algorithme de Dijkstra.

Nous avons dû faire des choix pour cette stratégie de navigation. Tout le long du module nous avons essayé d'être le plus rapide et le plus efficace possible. Sur les 8 premières semaines, notre système de validation de checkpoint couplé à l'organisation des marins sur le bateau, ainsi que notre manœuvre d'évitement implémentée un peu plus tard nous a permis de passer un bon nombre de courses. C'est à la 9eme semaine que nous avons rencontré un problème de taille dans notre manœuvre d'évitement, celui de passer entre deux récifs. Nous avons donc décidé d'implémenter l'algorithme de Dijsktra couplé à l'utilisation du vent et des courants, ce qui est très efficace, mais beaucoup moins rapide que la manœuvre d'évitement. Nous avons finalement décidé de rendre le projet plus stable et plus sûr en enlevant l'utilisation du vent, et en évitant les courants pour établir le chemin avec le dijsktra car ces deux paramètres engendraient trop d'approximations. Nous avons néanmoins décidé de coupler le dijsktra et la manœuvre d'évitement, ce qui nous permet de réduire le nombre de points de passages obligatoires donnés par le chemin tout en ayant une sécurité au cas où (lors d'un déplacement en diagonal avec une légère sortie du chemin). Malheureusement, nous avons eu plusieurs erreurs sûrement dues à des conflits entre la stratégie du dijsktra et celle de l'évitement. N'ayant pas réussi à résoudre ces erreurs, nous avons décidé de laisser seulement la stratégie d'évitement fonctionnelle pour le rendu final (la méthode de Dijsktra étant codée dans le package path, ou directement fonctionnelle dans la branche "pathfinding"). Notre stratégie, bien que non entièrement fonctionnelle, nous laisse penser que nous n'étions pas loin d'avoir un système solide, pouvant nous faire naviguer sur une multitude de mers. Ce serait sur ce point qu'il faudrait travailler si nous avions encore du temps pour le projet.

Outillage additionnel

Concernant l'outillage additionnel, nous n'avons utilisé que les outils vus en cours, mais nous aurions aimé développer un runner en local. En effet nous aurions pu optimiser notre stratégie car nous aurions pu faire plus de tests fonctionnels, afin de voir durant une course émulée les changements qu'apportent les modifications que nous faisions. Nous voulions aussi utiliser JArchitect, principalement car il offre une représentation graphique (ou sous forme de matrice) des dépendances entre les packages. Grâce à cette option, nous aurions pu localiser plus facilement les problèmes que nous avons rencontré car nous aurions cherché seulement dans les packages ayant une relation avec le problème en question.

Conclusion

Ce projet nous a permis d'y voir plus clair sur le concept de qualité au sein d'un code informatique, et nous a permis de découvrir des outils extrêmement efficaces pour suivre l'évolution de la qualité d'un projet à l'aide de métriques. Nous avons également appris à mettre notre code à l'épreuve des mutants afin de tester la valeur de nos tests, et même d'optimiser notre productivité en automatisant certaines tâches.

Au cours de ce module, nous avons pu mettre en pratique des connaissances que nous avons acquis dans d'autres cours. En effet, nous avons pu exploiter les bonnes pratiques en orienté objet que nous a inculqué le cours de Programmation Orienté Objet. Aussi, comme mentionné plus haut, nous avons réutilisé l'outil GitHub, que nous avions découvert dans le cadre du cours de PS5. Nous avons également tenté de mettre en pratique des patterns vus dans ce cours (GRASP, SOLID) afin de suivre des règles pour produire un code de qualité facilement extensible et maintenable.

Pour conclure, ce projet a été très bénéfique à notre manière de coder et à notre vision du développement informatique. A présent nous mesurons l'importance de l'aspect qualité de code que nous n'avions jusqu'à maintenant pas assez (voir pas du tout) pris en compte dans nos projets. Nous avons réalisé que la qualité représente un axe important dans la production d'un code, et qu'il existe de nombreux outils que nous pouvons assez facilement mettre en place afin de l'améliorer.

Ce projet nous a également permis de nous familiariser encore un peu plus avec l'outil GitHub que nous avons découvert cette année, et d'affiner encore notre organisation dans un projet informatique en groupe.