

Rapport final

Les qualituriers - qualituriers



ARCIL - CLODONG - D'ANDREA - BEDJGUELAL

Sommaire

Description technique:

- Phases du programme dans la prise de décision et lien entre ces étapes
- Choix de conception & architecture du projet
 - Instanciation du jeu / lecture des informations de jeu dans les JSON en entrés, génération des actions au format JSON
 - Génération de checkpoints intermédiaires
 - Actions permettant de faire bouger le bateau
 - Système de détection de collision

Application des concepts vu en cours:

- Branching Strategy
- Qualité du code
- Refactor du code
- Automatisation des tâches

Etude fonctionnelle et outillage additionnels

Conclusion

Description technique:

L'objectif de ce projet est de réaliser un algorithme (ou plutôt un bot) qui permet de réaliser une course navale, qui peut s'assimiler à une bataille navale, sauf que nous interagissons avec les éléments du bateau (marins, rames, voiles, vigies, gouvernail ...). Pour plus d'informations, la document github est présente à ce lien : <https://github.com/mathiascouste/qgl-2021>

Phases du programme dans la prise de décision et lien entre ces étapes

Au départ de jeu, l'arbitre de jeu nous fournit un JSON possédant toutes les informations sur le bateau (emplacement des marins, du bateau, informations sur la mer ...). Et ensuite, à chaque tour de jeu, l'arbitre nous fournit des informations complémentaires sur la partie. Nous avons décidé d'enregistrer toutes ces informations fournies dans une classe nommée `GameInfo`. Nous allons au départ du jeu, (lorsque la méthode `initGame` est appelée), initialiser `GameInfo`. Et ensuite, à chaque tour de jeu (lorsque la méthode `nextRound` est appelée), nous appelons une classe `Render` qui va :

- Mettre à jour le `GameInfo`
- Appeler une classe `MainPathFinding` qui va dire au bateau "pour aller au prochain checkpoint, il faut que tu passes par des checkpoints intermédiaires pour éviter de rentrer en collision avec des éléments de la mer", et donc, cette classe va donner ce checkpoint intermédiaire afin que le bateau puisse y aller.
- Appeler une classe `NewHeadQuarter` qui va gérer tout ce qui est interne au fonctionnement du bateau (gérer les marins, les rames, les voiles ...) afin que le bateau puisse se déplacer vers le checkpoint fourni par `MainPathFinding`.

Dans la conception de ce projet, nous avons essayé de créer un arbre de prise de décision similaire à une hiérarchie classique, afin d'en avoir le plus grand contrôle possible, autant au niveau de la maintenance qu'au niveau de la compréhension du code. Par exemple, la classe `Render` génère un checkpoint avec `MainPathFinding` et elle transmet ensuite ce checkpoint à `NewHeadQuarter`. Ensuite `NewHeadQuarter` appelle plusieurs classes qui vont générer des actions afin de permettre de faire avancer le bateau. Le fait de créer plusieurs petits "modules" a permis de modéliser le code à notre guise, remplacer des modules par d'autres, afin d'avoir de bonnes performances. Et ainsi, nous pouvons redéplacer chaque petit module si un nouveau mode de jeu est introduit.

Choix de conception & architecture du projet

Instanciation du jeu / lecture des informations de jeu dans les JSON en entrées, génération des actions au format JSON

Les informations de jeu arrivent au niveau de la classe Cockpit, cette méthode va instancier une classe GameInfo grâce au JSON en entrée et à l'api Java Jackson qui permet de parser un JSON en une instance de classe, si cette classe possède des tags @JsonProperty("nom_de_l'element_dans_le_json") avant chaque paramètre de classe. Nous instancions donc plusieurs instances décrivant chaque objet du json, et ces instances, nous les stockons dans GameInfo. Nous faisons la même chose lorsque la méthode nextRound est appelée, sauf que nous remettons juste à jour les instances de jeu.

Pour la génération d'actions en fin de traitement, nous faisons juste le chemin inverse grâce à Jackson. Finalement, nous avons une liste d'actions à effectuer, que nous parsons en JSON et que nous transmettons en sortie de nextRound au format String.

Génération de checkpoints intermédiaires

Pour créer le chemin nous avons utilisé une approche de waypoint, cette approche suffit et permet d'utiliser les systèmes qui étaient déjà implémentés et testés à ce stade. Pour se faire nous avons d'abord pensé à un système d'évitement d'obstacle basé sur un cercle "circonscriit" aux obstacles (le plus grand cercle passant par l'un des sommets et positionné à la moyenne des sommets).

Ceci s'étant avéré trop imprécis pour pouvoir atteindre tous les checkpoint (les obstacles fins deviennent trop grands). Nous avons donc dû utiliser un système tenant en compte la forme de l'obstacle.

Pour ce faire nous avons créé un algorithme de pathfinding utilisant les sommets des obstacles comme nœuds relié entre eux par des routes maritime ce qui à donné un graphe dont les routes sont pondérées par la distance entre ses deux extrémités. De là nous avons utilisé une variante de Dijkstra pour savoir quelles routes utiliser. cf: [alg.gif](#).

Afin d'optimiser les déplacement du bateaux et surtout empêcher les boucles infinies dues à des configurations de map particulières, le système stocke les obstacles immobile déjà rencontrés ce qui permet de créer une route la plus directe possible notamment sur des aller-et-retours.

Enfin si un chemin direct est possible, le système donne directement le prochain checkpoint pour éviter de repasser par dijkstra.

Actions permettant de faire bouger le bateau

Cette partie est générée par le package headquarter (lieu relatif à la prise de décision du bateau). Nous recevons le paramètre GameInfo lors de l'instantiation de la classe NewHeadQuarter, et nous allons travailler sur ce GameInfo, car GameInfo est relatif à tout le jeu.

Tout d'abord, nous allons donner des missions aux marins grâce à la classe GiveMissionToSailor (missions présentent dans l'énumération SailorMission). Les missions principales seront de ramer, cependant, selon les éléments présents dans le jeu (ou bien à quel moment de jeu nous sommes), nous allons vouloir donner à un marin une autre mission que de ramer. Tout ça est géré dans la classe qui NewHeadQuarter, qui a des méthodes permettant de dire "si nous sommes dans cette configuration, affecte un marin à une mission WATCH ou SAIL". Dans tous les cas, cette méthode affecte minimum un marin sur le gouvernail (si présent), et tout le reste des marins sur les rames (et il y aura un seul marin qui sera affecté à la WATCH et la SAIL et qui bougera entre les deux selon les circonstances de jeu).

Ensuite, nous allons affecter les marins à des emplacements sur le bateau et nous allons les bouger sur ces emplacements grâce aux classes AffectSailorsWith-ObjectivesToTheirBoatEntities et la classe MoveSailorsToTheirAffectedBoatEntities. Ces classes vont faire bouger ces marins de manière judicieuse grâce à une classe BoatPathFinding qui permet de trouver la meilleure destination pour le marin. Et enfin, nous allons faire ramer le bon nombre de marins et faire tourner le gouvernail grâce à la classe OarTheSailorsAndTurnRudder. De plus, les actions Watch et Sail sont gérés directement dans la classe NewHeadQuarter car ils sont un lien entre différentes classes, et avoir cette gestion ici facilite la compréhension du code, mais aussi la maintenance de celui-ci si une autre action venait à voir le jour.

Et finalement, nous récupérons la liste de toutes les actions faites précédemment que nous fournissons en sortie de la méthode launch().

Système de détection de collision

Assez tôt dans le programme, on s'est aperçu qu'on devait avoir un système de détection de collision, par exemple pour savoir que le bateau a bien touché un checkpoint. Au départ, elle était simple et suffisante mais elle n'était pas extensible puisqu'il y avait presque une méthode par couple de détection de forme (cercle-polygone, rectangle-polygone...). Comme on savait que les formes des polygones allaient toujours être convexe, on a opté pour l'algorithme des axes séparateurs. Il consiste à dire que deux formes convexes ne se touchent pas si on peut trouver une droite pour laquelle il existe une espace entre la projetés des deux formes sur cette droite. Pour sa mise en place, cela a consisté à utiliser les PositionableShape, qui sont des formes avec une position, en ajoutant des méthodes permettant de déléguer certains calculs qui leurs sont propres, comme les axes par exemple. Ainsi, l'algorithme ne contient qu'une méthode qui prend deux PositionableShape, ce qui fait que si d'autres formes devaient être ajoutées, il suffit

de créer une class qui extends PositionableShape, l'algorithme n'étant plus à être modifié.

Application des concepts vu en cours:

Branching Strategy

Nous avons opté au départ pour une variante de la "Git Flow Branch Strategy" où nous avons fait le choix de ne pas utiliser la branche HotFix car nous ne la jugeons pas nécessaire, du fait que nous ne sommes pas suffisamment de personnes sur le projet pour que cette branche ai vraiment un intérêt. Donc au départ nous avons les branches Master, Develop, Feature et Pre-Render. Nous avons très rapidement laissé de côté la branche et Pre-Render, car en pratique, elle ne nous été pas vraiment utile. Cependant, nous avons quand même gardé les branches Develop, Master et Feature. Sur Feature, nous travaillons sur des nouvelles implémentations, que nous mergions ensuite sur Develop, et quand la version sur celle-ci était stable, nous la mergions sur Master. Nous n'avons pas utilisé, du moins pas en majorité, de rebase, à l'inverse du merge.

Qualité du code

Nous avons testé notre code au fur et à mesure du projet grâce à SONAR et pīTest, notre objectif initial était d'atteindre les 80% de couverture de test et atteindre les 70% de mutation coverage. Durant une grande partie du projet, nous étions aux alentours de 70% de coverage et 60% de mutation coverage, ce que nous jugions trop faible. C'est pour cela que lors du refactor final, nous avons mis un point d'honneur sur l'amélioration globale des tests ainsi que de la qualité des tests.

Nous avons été très vigilant à avoir de bon score de tests aux niveau des packages d'intelligence (pathfinding & gestion de l'intérieur du bateau) car ce sont des blocs très importants du projet. C'est pour cela qu'en fin de projet nous avons une couverture de test sur ces blocs qui dépasse les 85 / 90% et un score de mutation coverage qui dépasse les 70%. Les scores finaux pīTest et Sonar sont disponibles en annexe.

Ces 2 blocs intègrent une partie de test unitaire, mais aussi de tests d'intégration, afin de voir, par exemple, si les marins se déplacent au bon endroit et génèrent les bonnes actions.

Nous avons aussi conçu notre propre Runner afin de pouvoir effectuer des tests End-To-End très souvent (à chaque build nous validons par un test sur notre runner), ce qui nous permettait de voir si le bateau arrivait bien à destination.

Nous avons essayé au maximum de respecter l'acronyme SOLID. C'est à dire que nous avons essayé d'avoir le maximum de classe avec une seule responsabilité (GiveMissionToSailors), nous avons essayé de laisser le code ouvert à l'extension avec les classes Render par exemple, le principe de Substitution de Liskov nous a été utile lors de la génération des JSON, par exemple une battleGoal découle de Goal. Nous avons également essayé de respecter le principe de ségrégation des interfaces au niveau de l'interface du Logger, et enfin, nous avons essayé au mieux de respecter l'inversion de dépendances en utilisant au maximum les abstractions, notamment les abstractions liées aux Point et Transform (présent un peu partout dans le projet)


Refactor du code

Au niveau du HeadQuarter, nous avons eu 2 refactors majeurs. Le premier refactor a eu lieu quand le gouvernail est arrivé. En effet, avant nous n'avions pas de HeadQuarter, nous avions un peu de code un peu partout, et donc, ceci était un code non-maintenable. En effet, le code se trouvant un peu partout, il était compliqué d'ajouter de nouvelles fonctionnalités dans la prise de décision car on devait modifier beaucoup de méthodes et de classes. En somme, le code était difficilement maintenable et extensible. Pour résoudre ce problème, nous avons créé une classe HeadQuarter, qui a la responsabilité de prendre des décisions sur les actions à effectuer par les marins. Ainsi, le code devenait bien plus maintenable et extensible grâce au fait qu'il était centralisé. Ensuite, nous avons refactorisé à nouveau le HeadQuarter pour passer au NewHeadQuarter qui était plus stable, qui évitait des interdépendances entre les classes et qui était plus facile à comprendre, car sur NewHeadQuarter, chaque classe a une seule mission (ou deux mais pas plus), et la connexion entre ces missions se fait plus clairement dans la classe NewHeadQuarter. Par la suite, cette version s'est avérée de moins en moins fiable au fur à mesure de l'augmentation de la difficulté dans la prise des décisions. et le code devenait compliqué à lire car la cohésion entre les classes augmentait. C'est pourquoi nous avons créé NewHeadQuarter qui réglait tous ces problèmes grâce au fait que chaque classe utilisée avait au plus deux missions, ainsi la connexion entre ces missions était plus lisible dans le code, et moins dépendante entre elles.

Pour le pathfinding, le système implémenté à la base permettait de rajouter des modules au pathfinding afin d'améliorer ses performances grâce aux différents éléments présents sur la carte comme par exemple les courants marins ou le sens du vent.

Etant donné que nous ne nous en sommes pas servi, ce système a été supprimé pour limiter le nombre de classes et donc la lisibilité.

Automatisation des tâches

D'un point de vue personnel, la découverte de Github Action et de l'automatisation des tâches fut un peu une révélation. En effet, le premier élément que nous avons souhaité implémenté était un build automatique lorsqu'il y a un push sur Develop et sur Master, afin que la petite icône  sur la page github à côté des commit puisse nous indiquer si le code était prêt pour le rendu du mardi soir. Cette petite icône nous a permis à plusieurs reprises d'aller re-vérifier certains tests ou certains éléments du code car le build ne passait pas. Nous avons aussi implémenté l'envoi d'une notification sur notre canal slack lorsqu'un push est fait sur Master. Nous souhaitions aussi lancer une analyse PiTest et SonarQube automatiquement à chaque merge sur Master, mais nous avons peur que cela nous prenne trop de temps de calcul, et donc que le cota de temps de calcul soit dépassé et que l'on ne puisse plus par la suite automatiser les build, or ils nous servaient énormément.

Etude fonctionnelle et outillage additionnels

Notre stratégie nous a permis d'être plutôt en haut du classement, par exemple nous sommes arrivés troisième lors de la dernière course.

Si une course contient plusieurs bateaux, notre bot va calculer un cercle de 100 autour de chaque bateau et si la trajectoire de notre bateau venait à rencontrer un des cercles, le bot va alors décider de ne pas bouger. Grâce à ça, nous n'avons eu aucun crash durant les courses. Mais cette stratégie n'est pas sans risque: si un bateau vient de face et adopte la même stratégie, notre bateau ne va plus bouger de la course.

Elle n'est cependant pas parfaite, plusieurs points peuvent être améliorés pour faire encore mieux.

Premièrement, notre algorithme qui calcul le meilleur checkpoint intermédiaire est trop lent, ce qui fait que nous empruntons pas forcément le meilleur chemin. L'optimiser aurait pu peut-être régler ce problème.

Deuxièmement, la vigie n'est pas assez utilisée, ce qui nous a valu de ne pas avoir beaucoup de points durant la 10ème course.

Troisièmement, on n'utilise pas les courants marins, ce qui nécessite de les esquiver et donc de perdre des points.

Quand après quelques courses passées nous remarquons que nous n'arrivons aucune course, et qu'en plus le jeu se complexifie, nous avons décidé de créer notre propre moteur de jeu. Fait avec Java Swing car c'est ce que nous maîtrisons, ils nous as permis de parfaire notre code et notre stratégie, et enfin de pouvoir réussir presque toutes les futures courses. Même s'il fallait du coup qu'une personne soit presque consacrée qu'à lui, le gain en valait la peine.

Conclusion:

Durant ce projet, nous avons d'abord appris à mieux nous organiser, notamment comparé à la dernière fois grâce à plus de réunions et plus de communication orale. Cela était nécessaire du fait du délai plutôt court que nous avons, mais aussi avec le fait de devoir garder un certain niveau de qualité de code entre chaque semaine.

De plus, nous avons appris à devoir garder un certain niveau de qualité de code en utilisant les principes de conceptions SOLID, GRASP, DRY et KISS., ainsi que SonarQube.

Aussi, nous avons appris à faire des tests plus complets avec Mockito, de trouver des tests manquants avec Pytest, et des tests d'intégrations pour s'assurer toujours plus du bon fonctionnement du code et qu'une nouvelle fonctionnalité aller cassé d'autre partie du code.

Également, apprendre à utiliser GitHub Actions nous a permis de réduire les risques de mettre du code non-fonctionnel sur Master.

Ensuite, nous avons appris à utiliser Maven, ce qui nous a permis de mieux gérer les dépendances, et de créer des profiles, ce que nous avons utilisé pour notre moteur de jeu.

Enfin, nous avons appris différents systèmes de collisions et de pathfinding.

Nous avons aussi durant ce projet pu exploiter des connaissances d'autres cours. Pour s'organiser et définir les missions de chacun, nous avons utilisé le système d'Issue, comme appris durant le cours de QGL au semestre 5, ainsi que la création de tests.

Enfin, plusieurs leçons sont à tirer de ce projet. Tout d'abord, répartir équitablement les missions est compliqué. En effet, quand on fait un système plutôt complexe, comme le pathfinding par exemple, il est dur de répartir en plus petites tâches car il faut que tout s'agence bien et que la conception soit la même. Ce qui fait qu'il est plus facile de donner toute la mission à qu'une personne, quitte à qu'elle passe beaucoup de temps dessus, en plus des tests à faire. Cela va faire aussi que chaque modification ne peut être faite que par cette personne, puisque elle seule connaît bien tout le code.

Annexes:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage
64	89% <div><div></div></div> 1380/1552	78% <div><div></div></div> 1062/1361

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
fr.unice.polytech.si3.qgl.qualituri	3	98% <div><div></div></div> 43/44	89% <div><div></div></div> 32/36
fr.unice.polytech.si3.qgl.qualituri	1	100% <div><div></div></div> 37/37	96% <div><div></div></div> 46/48
fr.unice.polytech.si3.qgl.qualituri	8	81% <div><div></div></div> 95/118	70% <div><div></div></div> 72/103
fr.unice.polytech.si3.qgl.qualituri	1	100% <div><div></div></div> 12/12	100% <div><div></div></div> 11/11
fr.unice.polytech.si3.qgl.qualituri	4	74% <div><div></div></div> 31/42	85% <div><div></div></div> 29/34
fr.unice.polytech.si3.qgl.qualituri	2	98% <div><div></div></div> 138/141	81% <div><div></div></div> 109/134
fr.unice.polytech.si3.qgl.qualituri	3	72% <div><div></div></div> 21/29	55% <div><div></div></div> 12/22
fr.unice.polytech.si3.qgl.qualituri	1	98% <div><div></div></div> 65/66	72% <div><div></div></div> 42/58
fr.unice.polytech.si3.qgl.qualituri	2	96% <div><div></div></div> 141/147	80% <div><div></div></div> 99/123
fr.unice.polytech.si3.qgl.qualituri	3	99% <div><div></div></div> 96/97	82% <div><div></div></div> 67/82
fr.unice.polytech.si3.qgl.qualituri	1	18% <div><div></div></div> 7/38	0% <div><div></div></div> 0/25
fr.unice.polytech.si3.qgl.qualituri	5	91% <div><div></div></div> 139/153	80% <div><div></div></div> 167/209
fr.unice.polytech.si3.qgl.qualituri	5	74% <div><div></div></div> 46/62	79% <div><div></div></div> 34/43
fr.unice.polytech.si3.qgl.qualituri	1	93% <div><div></div></div> 13/14	88% <div><div></div></div> 14/16
fr.unice.polytech.si3.qgl.qualituri	2	100% <div><div></div></div> 18/18	61% <div><div></div></div> 14/23
fr.unice.polytech.si3.qgl.qualituri	2	45% <div><div></div></div> 10/22	43% <div><div></div></div> 3/7
fr.unice.polytech.si3.qgl.qualituri	7	92% <div><div></div></div> 186/202	79% <div><div></div></div> 102/129
fr.unice.polytech.si3.qgl.qualituri	2	98% <div><div></div></div> 44/45	95% <div><div></div></div> 18/19
fr.unice.polytech.si3.qgl.qualituri	7	88% <div><div></div></div> 151/172	79% <div><div></div></div> 121/153
fr.unice.polytech.si3.qgl.qualituri	4	94% <div><div></div></div> 87/93	81% <div><div></div></div> 70/86

MEASURES

New Code

Since May 18, 2021
Started 5 hours ago

Overall Code

0

Bugs

Reliability

A

0

Vulnerabilities

Security

A

0

Security Hotspots

Reviewed

Security Review

A

2d 6h

Debt

415

Code Smells

Maintainability

A

89.0%

Coverage on 1.6k Lines to cover

293

Unit Tests

0.0%

Duplications on 3.8k Lines

0

Duplicated Blocks