Rapport De Projet

Qualité et Génie Logiciel (2020)

Nom : The Black Pearl

ID : theblackpearl

Équipe FISE :

Julien N'Diaye

Sommaire

Description technique

Architecture du projet

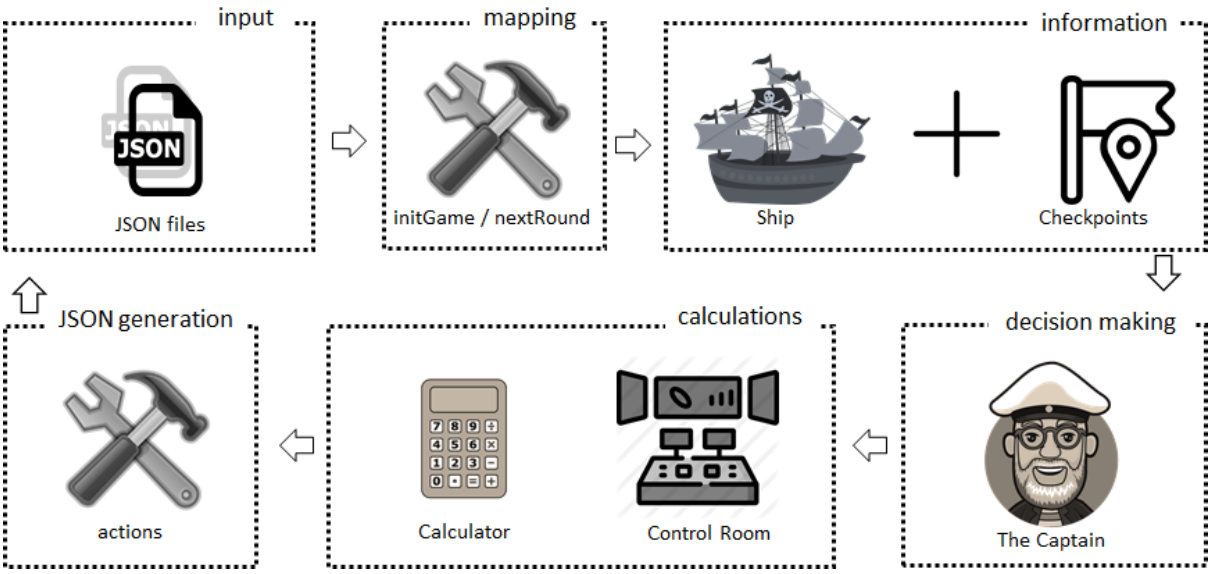
Présentation générale de l'architecture

L'architecture du projet consiste en deux types de classes, celles purement objets, qui pour la plupart consistent en la retranscription des données des fichiers JSONs, et les classes de décisions où se déroulent réellement les calculs et l'exécution des choix sur ces données.
Par ailleurs, pour ne pas mélanger des calculs abstraits, qui peuvent être longs, dans les autres classes de décision, nous avons créé une classe Calculator qui sert "d'encyclopédie mathématique". Cela nous a permis d'obtenir des classes plus lisibles et faciles à modifier lorsque nous ajoutions des fonctionnalités au fil des semaines.

Présentation plus détaillée de l'architecture

Détaillons un peu plus l'architecture de notre projet à travers l'exécution d'un tour. Nous allons présenter les différentes phases de notre programme lors de la prise de décisions. On va donc s'intéresser à la méthode *nextRound* de la classe *Cockpit*.
La première étape consiste à parser le fichier Json envoyé en paramètre qui contient toutes les informations relatives au tour suivant et de mettre à jour l'objet correspondant. Cette classe objet s'appelle dans notre cas *NextRound*.
Une fois que cela est fait, on réinitialise le statut des marins. On les rend à nouveau disponibles puisqu'on va tout recalculer pour le tour suivant. On fait ensuite appel à l'objet Captain qui comme son nom l'indique va prendre des décisions et distribuer les tâches aux marins. Ces responsabilités sont entre autres, de déterminer la meilleure configuration possible des marins pour réaliser l'angle le mieux adapté pour atteindre le prochain checkpoint. C'est aussi lui qui s'occupe de déterminer quel est le prochain checkpoint à viser, une fois qu'il s'est assuré que le précédent a bien été atteint. Enfin pour mener à bien sa mission de chef d'orchestre, le *Captain* va avoir recourt à deux autres classes importantes, la *ControlRoom* et *Calculator*.

Schéma récapitulatif du déroulement de notre programme :



La *ControlRoom* est la classe où des choix plus précis sont faits et exécutés. Ceux-là concernent notamment le placement des marins, l'utilisation de la voile et l'utilisation du gouvernail. Nous reviendrons plus en détail sur les choix qui ont été faits dans l'étude fonctionnelle.

Pour ce qui est de la classe *Calculator*, elle réunit les calculs essentiels du projet comme ceux qui permettent de déterminer l'angle théorique optimal pour atteindre un checkpoint, ceux qui permettent de déterminer les éventuelles collisions à venir avec des récifs ou encore ceux qui permettent de calculer la nouvelle position du bateau.

Lorsque tous les choix ont été faits et que la liste des actions à réaliser par chaque marin a été établie par le *Captain*, on la renvoie alors sous la forme d'un fichier Json depuis la méthode *nextRound* du cockpit.

Pour conclure sur l'architecture de notre projet, nous dirions que les points positifs sont sa clarté globale, dans le découpage des packages et donc pour comprendre rapidement quels sont leurs rôles. En revanche certaines de nos méthodes sont assez longues et il est difficile de les refactorer sans changer toute la structure du code. Ainsi, la règle une méthode = une tâche n'a d'ailleurs pas pu être appliquée partout, et ce serait clairement un axe d'amélioration pour apporter plus de lisibilité au code.

Quant à l'extensibilité de l'architecture, dans le cadre d'un nouveau mode de jeu tel que la bataille navale par exemple, il suffirait de rajouter la classe correspondant au nouveau mode de jeu dans le package goal. Puis, Captain pourrait se charger d'un certain nombre de nouvelles décisions à appliquer en fonction du mode de jeu.

Contraintes imposées

En ce qui concerne l'impact de l'interface imposée par ICockpit et par le schéma des JSONs initGame et nextRound, il a clairement été positif. En effet le format des fichiers JSONs nous a permis de nous faciliter la construction et l'implémentation de nos modèles de données, à travers nos classes objets, en nous renseignant directement sur les attributs principaux qui devaient être créés. De même, le découpage des tâches défini par l'interface ICockpit nous a donné un cadre clair de ce que nous devions faire et à quel endroit.

Application des concepts vus en cours

Git

La branching strategy que nous avons mise en place pendant le projet a consisté à travailler sur deux branches distinctes de master. Nous avons en effet développé un Referee assez vite dans le semestre que nous avons implémenté sur une branche éponyme. Une fois que nous étions sûrs qu'il fonctionnait correctement, nous avons poussé le code correspondant sur la branche principale master. Plus tard dans le projet, notamment lorsqu'il a fallu commencer à implémenter des stratégies de repérage des récifs, nous nous sommes mis d'accord pour avancer sur une branche de développement dédiée nommée dev. Cela nous a permis de tester plusieurs versions du code, d'éviter des remaniements de master et d'avoir une certaine stabilité de la branche principale. En fin de projet, nous avons travaillé sur cette branche dev pour développer certaines méthodes et trouver des solutions à des bugs.
Pour le reste, nous avons travaillé directement sur master puisque l'envergure des commits était restreinte et les fonctionnalités relativement directes à implémenter.

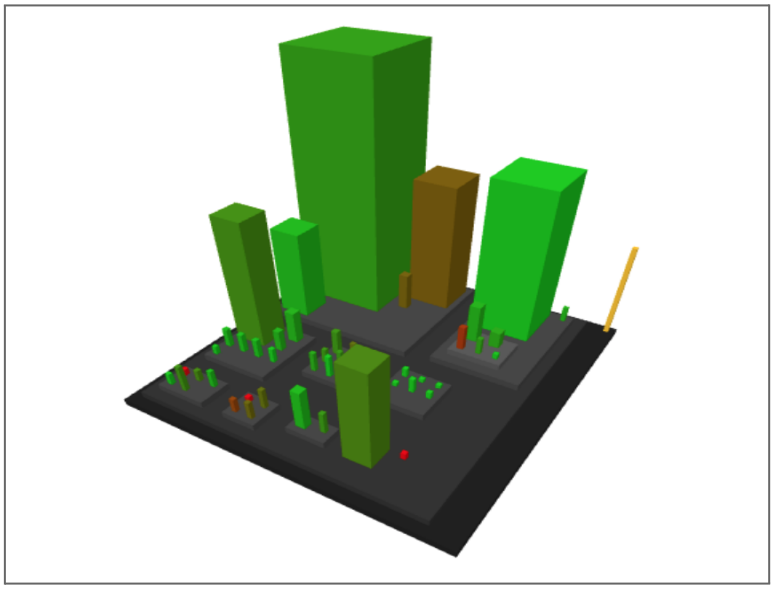
Qualité du code

La qualité de notre code est bonne. Comme nous l'avons déjà évoqué, nous avons une architecture globale plutôt claire, puisqu'elle est bien structurée. De plus, le code est entièrement écrit en anglais afin d'éviter les mélanges de langue, et nous avons tâché de nommer les méthodes de façon cohérente afin de faciliter la compréhension du code.
En ce qui concerne la couverture de test de notre projet, elle est également relativement satisfaisante. En effet, les PITests nous ont permis d'augmenter sa qualité au fil des semaines et de nous focaliser sur les parties du code les plus importantes. Ainsi, nous avons pu concentrer la plupart de nos tests sur les classes effectuant des calculs mathématiques et des prises de décision afin d'être sûr qu'ils soient corrects dans tous les cas grâce aux mutations indiquées par les PITests. Ces tests nous ont permis d'identifier de nombreuses erreurs au sein du *Calculator,* et ainsi nous avons pu les corriger pour améliorer nos résultats.

Cependant, nous avons parfois eu beaucoup de mal à tester certaines méthodes du fait de leur complexité, même avec l'aide des Mocks. En effet, nous avons fait l'erreur d'écrire les tests après avoir fait la majorité du code. Ceci nous a fait prendre conscience que notre code était à de nombreux endroits mal conçu et qu'il fallait le refactorer pour pouvoir le tester correctement. Cette erreur nous a fait perdre beaucoup de temps et malheureusement nous n'avons pas pu refactorer la totalité du code mal conçu. C'est pourquoi, à l'avenir, nous tâcherons de rédiger les tests en même temps que la conception du code.

Enfin, Sonar a également été un outil très utile pour nous car il a permis d'identifier les duplications de code au sein de notre projet mais aussi les lignes inutiles. Nous nous en sommes servis principalement pour nettoyer le code lors des refactoring mais également pour mieux visualiser notre projet au niveau de sa complexité et de sa couverture de test.

La représentation ci-dessous a été obtenue grâce à une fonctionnalité de Sonar et nous a permis tout au long de notre projet d'avoir une vue d'ensemble sur celui-ci. En effet, grâce à elle, nous avons pu constater les classes (tours) les plus complexes (largeur de la tour), celles contenant le plus de ligne (taille de la tour) mais aussi celles les mieux testées (couleur de la tour). Ainsi, nous avons pu orienter nos refactorings et nos tests en fonction des faiblesses constatées.



Si la qualité actuelle était moindre, ne serait-ce que pour les principes SOLID et en particulier la règle une méthode = une tâche, le code serait très complexe à comprendre et à lire.

En revanche, si la qualité était plus élevée, nous aurions eu plus de temps pour nous focaliser sur l'évitement des récifs.

Refactorings

Nous avons dû effectuer un refactoring du code à chaque fois qu'une classe ou une méthode devenait trop longue, trop complexe ou bien que certaines parties n'étaient plus utiles. Ainsi sur l'ensemble du projet, nous avons effectué environ 5 refactorings globaux. A chaque fois pour apporter un meilleur découpage des responsabilités entre les classes et méthodes, mais aussi pour plus de lisibilité. Lors de l'un d'entre eux, nous avons déplacé une grande partie des décisions dans les classes dédiées *Captain* et *ControlRoom*. En effet, au début du projet, nous avions la majeure partie des décisions placées dans une seule méthode de la classe *Cockpit*. Dès lors que le nombre de décisions à prendre au cours d'un tour a augmenté, la méthode est devenue illisible et beaucoup trop grande. C'est pourquoi nous avons pris cette décision de refactoring.

L'automatisation

L'automatisation des builds par Travis nous a permis de garantir la stabilité de notre code et de constituer une sécurité supplémentaire après chaque push sur la branche master. Plusieurs fois au cours du semestre, il a fallu revenir sur le code après s'être rendu compte que le build ne s'était pas passé correctement. Ainsi, Travis nous a permis d'éviter des erreurs de build au cours de certaines livraisons.

Étude fonctionnelle et outillage additionnels

Notre stratégie

Notre stratégie consiste à viser les checkpoints dans l'ordre exact que l'on reçoit. Tout d'abord nous avons implémenté un petit système qui nous permet de savoir en premier lieu si le vent sera notre allié durant la ruée vers le checkpoint, ce qui nous permet dans le cas échéant d'hisser la voile ou bien de l'affaler.

Après la répartition des marins sur les rames, sur laquelle nous reviendrons plus bas, nous chargeons un marin de se rendre au gouvernail, et de corriger l'angle formé par les rames pour viser très précisément le checkpoint. Cette stratégie devient problématique lorsque l'angle formé par les rames est déjà très précis, nous perdons un marin pour le tour. Aussi une amélioration serait, dans ce cas, de reprogrammer le marin pour qu'il se rende à la vigie, ou de l'intégrer aux rameurs si la trajectoire du bateau résultante reste cohérente par rapport au prochain checkpoint. La limite de cette stratégie d'amélioration est le gaspillage inévitable du marin dans certains cas. Cette potentielle amélioration ne réduirait pas l'espérance mathématique de gâcher un marin à zéro, mais il est nécessaire de faire des choix.

Attaquons nous au plus important, la répartition des marins sur les rames. Pour comprendre les possibles améliorations de notre stratégie, résumons là en quelques points :

1 • Préconfiguration des rames -> choisir un angle réalisable par les rames pour viser le checkpoint + initialiser un calculateur

2 • Configuration des rames -> Positionnement des marins de part et d'autre du bateau

3 • Vérification -> bonne configuration bateau ? Si non -> retour étape 1 •

En ce qui concerne le point 1, l'angle est considéré réalisable par les rames si les rames à elles seules peuvent conférer cet angle. Un point négatif est que nous ne prenons pas en compte le nombre de marins disponibles à ce moment précis sur le bateau. Nous pouvons

par exemple considérer un angle de PI/2 réalisable même s'il n'y a aucun marin disponible pour aller ramer. Une amélioration consisterait à prendre en compte les angles théoriquement faisables par les rames et par les marins.

Attardons nous maintenant sur le point 2. La configuration des rames consiste à placer les marins sur les rames et voir si l'angle formé correspond à l'angle demandé par le capitaine. Nous utilisons pour cela notre calculateur, qui permet de trouver le positionnement de chaque marin et du nombre de marins restant à positionner à gauche ou à droite. Cela nous permet ensuite de les déplacer. Le calculateur clone la liste de rames et retire l'entité correspondante dès qu'un marin est positionné à son emplacement. Une amélioration toute simple serait de rajouter un booléen renseignant sur la disponibilité des entités.

Enfin pour le point 3, la vérification consiste à s'assurer que le calculateur indique que tous les marins, pour la configuration qui leur a été donnée, ont été correctement placés de part et d'autre du bateau. Le cas échéant nous gardons cette configuration et recommençons à l'étape 1 jusqu'à sortir de cette boucle sinon. Un point négatif pour la vérification est de ne pas pouvoir réagir à une potentielle erreur qui serait survenue plus haut. En effet, si pour une quelconque raison, aucun angle ne peut se voir parfaitement réalisé par les marins nous sortirons de cette stratégie avec la pire des configurations. En effet nous testons les meilleures configurations jusqu'aux pires configuration avec une sortie forcée de la boucle lors de configuration pour le dernier des angles pour ne pas tomber sur une boucle infinie. Une nette amélioration serait que notre vérificateur détermine si notre configuration nous est trop néfaste ou pas au lieu de garder une configuration qui nous emmènera sur la Lune.

Enfin abordons notre algorithme de détection de collision ainsi que notre stratégie d'évitement des récifs.

Nous avons choisi de faire un algorithme de détection qui se base sur le théorème SAT mais n'utilise pas les mêmes outils que nous trouvons sur internet pour déterminer une collision. Cela nous a permis de comprendre l'entièreté de la détection de collision et d'avoir une plus grande confiance envers notre algorithme si ce n'est une confiance totale. Malheureusement nous avons utilisé beaucoup de instanceof dans notre algorithme, ce qu'il

faut éviter. Cela nous a permis de voir que nous avons insuffisamment utilisé le concept de l'orienté objet par endroit.

Nous aurions pu mettre en place une stratégie d'évitement des récifs pour permettre au bateau de passer différents obstacles, malheureusement nous n'avons pas réussi.

Outils supplémentaires utilisables

Nous avons développé notre propre arbitre pour pouvoir lancer autant de fois que nous voulions certaines courses. Cet arbitre nous a bien aidé à comprendre notamment d'où venaient certains problèmes lors de la détection des récifs.

Il nous a également permis de développer la stratégie de ralentissement du bateau. En effet, grâce à l'implémentation de notre propre referee, nous avons pu simuler la trajectoire du bateau en fonction des actions réalisées pour le tour, avant de les envoyer directement au réferee officiel. Ainsi, lorsque nous détectons que le bateau passe à travers un checkpoint sans le valider, nous pouvons arranger les actions du tour afin de s'arrêter dans le checkpoint.

Conclusion

Au cours de ce projet, nous avons tout d'abord appris à utiliser un certain nombre d'outils qui peuvent s'avérer bien utiles à l'écriture d'un code de meilleure qualité, notamment à l'aide de Sonar, mais aussi à améliorer la qualité des tests, grâce aux PItests. Quant à Travis, il nous a apporté une sécurité supplémentaire dans la détection des erreurs. Mais ce projet nous a aussi permis de mettre en pratique les principes SOLID sur un code de plus grande envergure, bien qu'encore perfectible dans la version actuelle du code.

Plusieurs notions de géométrie plane ainsi que trigonométrie ont été exploitées au cours de ce projet. Que ce soit pour le calcul des positions des objets tels que le bateau, les récifs, les marins ou dans la détection des collisions entre polygones, un certain nombre de connaissances mathématiques ont dû être reprises voire recherchées.

Si nous devions résumer ce que nous retenons avant tout de ce projet, nous mettrions la nécessité de développer les tests systématiquement en parallèle des fonctionnalités . D'autre part, on ne nous le répétera jamais assez mais la mise en place d'une bonne architecture dès le début, d'une bonne distribution des responsabilités et surtout d'un code lisible sont primordiaux. Nous avons en effet eu des difficultés à mettre en place un code limpide et il a parfois été difficile de se relire en particulier en milieu de projet.