UNIVERSITÉ CÔTE D'AZUR

POLYTECH NICE SOPHIA

SI3 : QUALITÉ GÉNIE LOGICIEL

Rapport du projet : QueLeGlitch
id :queleglitch
20/05/2021

Étudiants :

Antoine Le Calloch
Antoine Huot-Marchand
Eric Naud
Loic Madern

Sommaire

Introduction

Dans le cadre de notre première année de cycle ingénieur en informatique à Polytech Nice Sophia, nous avons l'opportunité de faire un projet informatique de quatre mois.

Ce projet est la réalisation d'une version informatisée d'une course de bateau par équipe. Chaque équipe étant à la tête d'un équipage de marins qui doivent exécuter des actions sur les différents éléments du bateau afin de piloter au mieux le navire.

Le programme a été codé en java.

I Description technique

1.1 Architecture

Les différentes phases de prise de décision dans notre programme sont les suivantes :

Tout d'abord le projet s'articule autour de la classe Cockpit faisant appel à l'interface *ICockpit*. Cela permet de faire le lien entre le moteur du jeu et l'intelligence artificielle (I.A) mis en place dans notre programme. Dans cette classe on va donc initialiser la partie à partir de *initGame*() qui va lire les informations contenues dans la classe *InformationGame* et on va faire appel à *nextRound*() qui va lire la liste d'action à effectuer sous format Json à l'aide de la classe *NextRound*.

Ainsi, dans la classe *InformationGame* sont implémentées les méthodes qui permettent de récolter les informations liées aux éléments du jeu tels que le bateau, le vent, le mode de jeu, les checkpoints …

Dans cette classe nous allons donc faire appel à la méthode *setNewRound*(*NextRound nextRound*) pour mettre à jour les informations concernant les éléments du jeu évoqués auparavant.

Cette mise à jour s'effectue donc via la classe *NextRound*.

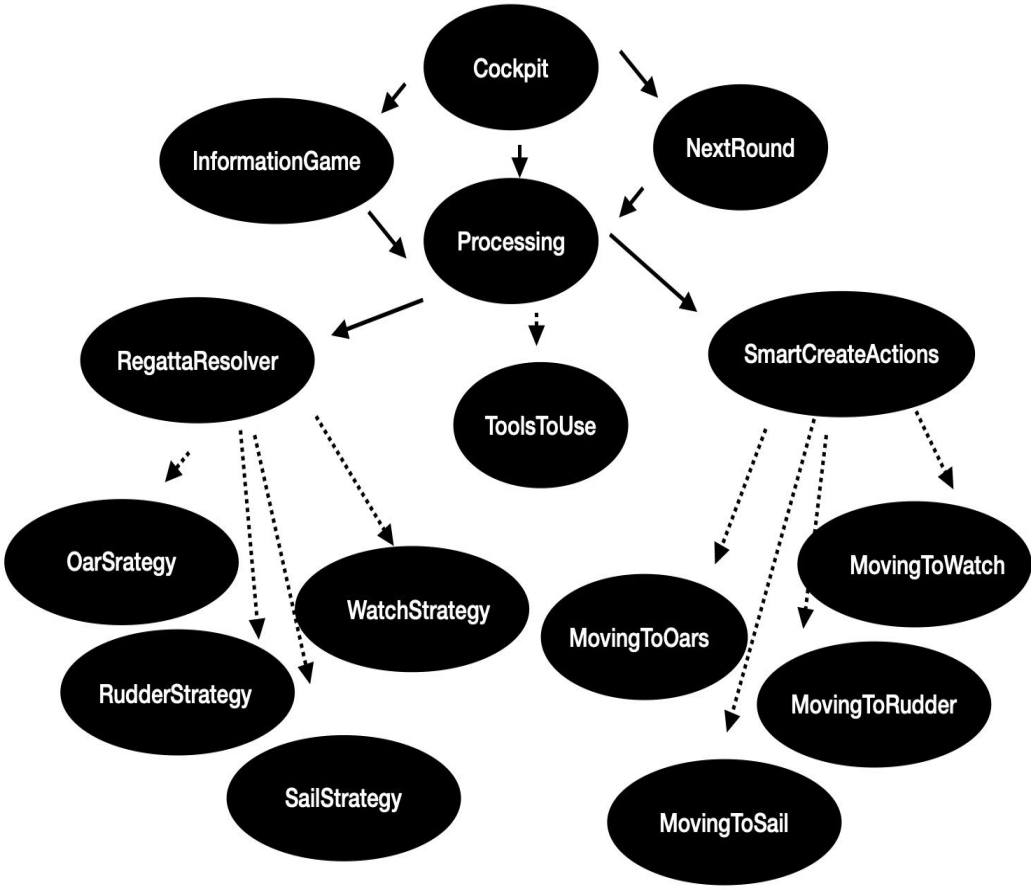
En amont, pour savoir quelles actions sont à effectuer, on va utiliser la classe *Processing*. Cette classe qui aura donc connaissance des informations contenues dans *InformationGame* va donc appeler la méthode *resolveToolsToUse*() de la classe *RegattaResolver* avec des paramètres qui vont donc différer selon les informations connues en avance telle que la position du checkpoint à atteindre. Enfin c'est aussi dans la classe Processing que nous allons demander de créer les actions à effectuer via un appel à la méthode *createActions*() de la classe *SmartCreateActions*.

C'est donc dans *RegattaResolver* que nous allons appeler les différents classes « stratégies » (*OarStrategy*, *RudderStartegy*, *SailStrategy* …) qui permettent de gérer les actions à faire pour chaque élément du jeu telles que les rames, la voile (…)

Selon les stratégies adoptées on va donc devoir créer différentes actions à partir de *SmartCreateActions* comme expliqué ci-dessus. Cette classe va donc bénéficier de l'appui de 4 autres classes (*MovingToOars*, *MovingToRudder*, *MovingToSail*, *MovingToWatch*) dont le rôle est de déplacer les marins nécessaires à la tâche qui leur est associée, sur les bons emplacements.

Finalement après que les actions aient été créés et renvoyées à *Processing* pour les effectuer, les éléments du jeu (par exemple : position du bateau) vont changer. Cela sera mis à jour dans *NextRound* pour le prochain tour et on répète le même fonctionnement jusqu'à finir la course.

Cette architecture est assez facilement extensible. En effet pour implémenter un nouveau mode de jeu, il suffit de créer une classe qui correspond à ce mode et de l'ajouter aux informations dans *InformationGame*. Ensuite on peut l'utiliser de la même façon que *RegattaGoal* est appelée dans *Processing*.



Concernant les choix de conception et les choix algorithmiques, nous avons choisi de répartir les responsabilités en créant une classe pour chaque tâche comme expliqué au-dessus (comme pour le déplacement des marins selon les entités à utiliser). L'un de nos choix algorithmiques concerne justement, le déplacement des marins pour lequel nous avons décidé dans un premier temps de regarder quelles étaient les entités du bateau à utiliser pour respecter le cap que l'on se fixe de manière la plus efficace. Une fois que les entités à utiliser sont connues nous faisons appel à la classe *Tooling* qui permet de déplacer les marins les plus proches des entités à utiliser de manière optimale tout en faisant attention aux placements d'origine des marins qui pourraient se trouver trop loin des cases à atteindre pour se déplacer en un seul tour. Ainsi cette classe est utilisée dans chacune des autres classes *MovingTo*… qui ont pour rôle d'assurer le placement du bon nombre de marins sur les cases qui correspondent à l'entité souhaitée.

Une fois les marins bien placés nous avons fait un deuxième choix fort dans notre conception qui concerne l'utilisation des entités afin de faire avancer le bateau vers son objectif. Pour cela nous avons décidé de faire un choix plutôt risqué qui a été d'optimiser la trajectoire le plus possible pour gagner en vitesse et finir la course avant les autres. C'est pourquoi dès le début du projet nous avons décidé d'implémenter des checkpoints virtuels qui permettent de prendre les virages de la façon la plus serrée au lieu de passer au milieu des checkpoints à chaque fois. Pour se faire nous avons alors adapté l'utilisation des rames et du gouvernail pour que l'angle de déplacement réalisé par le bateau respecte notre vision qui nous semble optimale. Malheureusement ce choix a été quelque peu désavantageux au début du projet puisque le fait de prendre ces virages serrés a engendré des erreurs car le bateau traversait le checkpoint en moins d'un tour et nous nous rendions donc pas compte de la traversée de ce dernier. Néanmoins même si cela a compromis la réussite de plusieurs courses, ça nous a permis de réaliser de meilleurs résultats par la suite.

Enfin le dernier choix important de notre projet a été l'utilisation de la méthode de Dijkstra pour le *pathfinding*. Ainsi à l'aide d'une grille découpant la carte en de minimes parties, il était plus facile de repérer les obstacles et donc de choisir le trajet optimal. Cette méthode nous a pris beaucoup de temps à implémenter mais nous a finalement permis de finir 2ème de la dernière course avec un tour de retard, ce qui a été payant.

1.2 Impact des différentes contraintes

L'interface imposée par *ICockpit* et le schéma *JSONs* nous ont obligé dans un premier temps de récolter les données initiales d'une partie dans Cockpit grâce à la méthode *initGame*(), puis de récolter les données de chaque tour grâce à *nextRound*().

Cette récupération de données est une contrainte puisqu'elle doit être mise à part du traitement de ces dernières. Ainsi la récolte de données forme la première partie de notre architecture grâce au Cockpit.

Vient ensuite le traitement des données reçues au format *JSONs* qui sont parsées puis instanciées en objet grâce à l'API Jackson pour que les données reçues puissent être analysées. Cette étape requiert à nouveau d'être isolée pour pouvoir respecter le principe d'une seule responsabilité. Cela est respecté grâce à la classe *Processing*.

Enfin une fois les décisions prises, elles sont encodées sous le format JSON pour être envoyées à *nextRound* grâce à la classe *Processing*.

Les données ont donc suivi un cycle à chaque nouveau tour: la collecte des données, la centralisation des données, l'analyse des données, la restitution et enfin l'amélioration.

II Application des concepts vu en cours

2.1 Branching strategy

Nous avons choisi le *Github* Flow comme stratégie de branchement. En effet, c'est le compromis idéal entre le *GitLab* Flow et le Git Flow.

Nous pouvons observer les avantages et inconvénients de ces deux stratégies non retenues :

	Inconvénients	Avantages
GitLab	• difficulté pour lire l'évolution de projet • forte probabilité d'instabilité	• simple à implémenter et à maintenir
Git	• lourd à mettre en place et à maintenir • pas adapté à un projet mono version	• stabilité de la branche • production en parallèle

Github Flow possède la simplicité d'implémentation et de maintien de GitLab Flow mais elle possède aussi une bonne stabilité et une bonne légèreté comme Git Flow ce qui nous a permis de pouvoir produire du code de façon sereine tout au long du projet.

2.2 Qualité du code

2.2.1 Notre qualité

Dans l'ensemble, la qualité de notre code est bonne.

Tout d'abord, les tests sont de bonnes qualités car ils couvrent la quasi-totalité du code implémenté ainsi que les cas limites. D'après *piTest*, nous tuons 76% de mutants (voir annexe 1) et nous couvrons 87% des lignes du code. Et sonar nous indique aucun bug et seulement 22 *code smells* qui sont des problèmes d'optimisation de tests. (voir annexe 2).

De plus, nous avons essayé de respecter le principe de *SOLID* le mieux possible. Nous avons bien divisé les différentes parties de notre code en différentes classes. Par exemple, les calculs de stratégies, l'algorithme de *pathfinding* et la stratégie de décision des actions des marins sont bien divisées en plusieurs packages.

D’ailleurs, le schéma de building de sonar (voir annexe 3) ne montre pas qu’il y a eu un problème majeur avec les responsabilités des classes.

Enfin, toutes les fonctionnalités de jeu ont été implémentées à part l’utilisation des canons. Mais, il nous est arrivé de ne pas réussir une semaine car le temps pour prendre les décisions pendant un tour était trop long

2.2.2 Impact de la qualité

La qualité de notre projet a eu un énorme impact sur notre livraison et notre organisation. En effet, la principale difficulté du projet est de livrer un code de qualité dû au fait que nous ne pouvons pas tester notre code autant de fois que l’on souhaite dans les conditions réelles.

Au début du projet, nous livrons toujours notre dernier code créé, mais à cause de cela, nous n’avons pas passé certaines semaines.

Ainsi, au cours des semaines nous avons appris à livrer un code de qualité même si ce code ne comportait la totalité des fonctionnalités.

De plus, en fonction de la qualité de notre code, nous avons passé plus ou moins de temps pendant une semaine sur la réduction des problèmes de qualité en utilisant sonar.

2.3 *Refactoring*

Au cours de ce projet, nous avons fait seulement un *refactoring* global. En effet, durant la cinquième semaine, nous avons modifié le système de création des actions des marins.

Avant ce *refactoring*, le choix des actions et les calculs qui permettent de savoir quelle action chaque marin doit appliquer étaient liés. Or, avec l’ajout d'éléments comme le vent, nous devions créer une stratégie plus complexe. Les stratégies d’utilisation des entités du bateau ont donc été divisées ainsi que le choix des actions que les marins devaient faire.

Ainsi, durant cette semaine, nous avons pris comme décision de corriger ce problème. Nous avons donc codé une première partie de code qui était dédiée aux calculs pour trouver l’angle du gouvernail ou encore pour trouver le nombre qui représente la différence entre les marins de droite et de gauche.

Et, nous avons codé une deuxième partie du code qui sert à créer les actions des marins en utilisant l'objet *ToolsToUse* possédant en attribut tous les résultats des calculs faits auparavant.

De cette manière, avec cette nouvelle configuration, l'ajout des derniers éléments de jeu ont été beaucoup plus simple à implémenter.

2.4 Automatisation

Nous avons créé deux actions qui permettent l'optimisation de certaines tâches.

Cela permettait le lancement du *build* de l'application dès que nous faisions un push ou un *pull-request*.

Ces automatisations nous ont permis de contrôler rapidement si un push était bon ou pas et ainsi d'augmenter la qualité de notre code. En effet, au début du projet, il nous est arrivé de faire un tag sur un push qui avait une erreur de *build*. Or, après l'implémentation de ces actions, nous n'avons pas eu de nouveau ce problème.

III Etude fonctionnelle et outillage additionnels

3.1 Stratégie

3.1.1 Notre stratégie

Pour ce projet nous avons adopté une stratégie risquée mais efficace. En effet cette stratégie est de réaliser la trajectoire la plus serrée possible et donc d'emprunter le chemin le plus court qui existe entre deux checkpoints. Malgré la difficulté que cela a pu engendrer au début du projet, cette méthode nous semble en fin de compte très efficace.

De plus avec l'ajout des courses multijoueur nous avons détecté un autre point fort dans notre stratégie. En effet lors des courses multijoueur il est désormais possible que les bateaux se heurtent entre eux mais sachant que notre trajectoire est synonyme de rapidité nous pensons que le fait de prendre une avance certaine dès les premiers mètres de la course nous assure une certaine sécurité puisque les bateaux adverses seront trop loin pour nous causer des dégâts.

Le point faible de notre stratégie réside dans le fait que nous ne prenons pas en compte l'évitement d'autres bateaux et donc il serait dans l'absolu possible que nous heurtions le bateau d'une équipe adverse. Avec notre stratégie nous pensions dans un premier temps que le fait d'aller plus vite que les autres nous sécurisait d'un quelconque accrochage alors que ce n'était pas exact. Effectivement il se peut que la course ne soit pas linéaire et qu'elle repasse plusieurs fois par le même endroit. Cela pose donc un léger problème puisque avec ce genre de schéma de course notre bateau pourrait alors recroiser des bateaux adverses en contre sens même si ceux-ci ont beaucoup de retard.

Nous avons donc pesé le pour et le contre de notre stratégie et avons décidé que c'était un bon compromis, car les faiblesses de celle-ci ne se manifestent que dans des cas précis. De plus l'une de nos réflexions a été la suivante : si jamais un bateau venait à se mettre en travers de notre chemin alors il suffit que l'équipe adverse qui contrôle ce bateau ait quant à elle implémenté l'évitement d'autres bateaux. Cela nous permettrait donc de garder notre bateau intact puisque l'autre équipe s'occuperait de l'éviter et de garder un avantage positionnel certain, car nous n'aurions alors pas besoin de changer de cap ce qui permettrait de continuer à emprunter le chemin le plus court.

3.1.2 Stratégie à mettre en place

Pour nous assurer une meilleure course et augmenter nos chances de victoire encore plus, il faudrait donc implémenter l'évitement de bateau car notre stratégie ne peut pas répondre à ce problème dans 100% des cas. De plus on pourrait dans l'avenir réduire la taille des cases qui composent la grille nécessaire au *pathfinding* et de réduire les marges de sécurité que nous nous sommes mises pour l'évitement des récifs afin d'avoir des informations encore plus précises et un maillage optimal de la carte de manière à aller encore plus vite.

Nous aurions aussi pu implémenter d'autres fonctionnalités qui auraient permis à notre bateau de rejoindre des checkpoints plus vite. En effet, nous prenons en compte seulement la trajectoire du bateau en prenant en compte le vent et les courants qu'il y a sous notre bateau. Nous évitons alors les courants à contresens et nous utilisons correctement nos voiles en fonction du vent. Mais, il aurait été possible de calculer une trajectoire moins directe en observant les courants et le vent avec note vigie, un détour sur un courant rapide peut être plus rapide qu'une ligne droite.

3.2 Outils utilisés

Un seul outil a été développé en plus du code : un lecteur de fichier qui nous a permis de transformer un fichier lu au format String, en format *JSONs* pouvant être lu par la méthode main de la classe Application présente dans tout *Tooling*

```
public class FileOpener {

    String path;

    public FileOpener(String path) { this.path = path; }

    public String getTxtInFile(String name){
        try {
            BufferedReader file = new BufferedReader(new FileReader( fileName: path + name));
            String line;
            StringBuilder txt = new StringBuilder();
            while ((line = file.readLine()) != null)
                txt.append(line);
            file.close();
            return txt.toString().replaceAll( regex: "\\s",  replacement: "");
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

Cet outil nous a donc permis de lire ce que nous renvoie notre programme d'un tour à l'autre suivant les données d'un *nextRound* et d'un *initGame*. Cet outil nous a été utile lorsqu'il nous restait peu de crédits sur le *webRunner* ou lorsque notre code était complexe. En effet, parfois il était difficile de pouvoir tester notre code de manière exhaustive, notamment lors du développement du *pathFinding* qui a augmenté considérablement la complexité algorithmique. Lorsque le bateau échoue lors d'une course d'entrainement sur le *webRunner*, nous nous replaçons dans ce cas exact et ajustions jusqu'à ce que l'IA de notre bateau puisse retourner une décision satisfaisante selon notre critère de performance.

Nous aurions pu créer un autre outil difficile à implémenter : la création du code de la partie *tooling* avec notre propre *webRunner*. Cela nous aurait permis de tester notre code dans les mêmes conditions que les rendus hebdomadaires.

IV Conclusion

Tout d'abord, lors de ce projet nous avons appris à développer un programme qui suit des contraintes prédéfinies : à savoir un obtenir un *JSON* en entrée, et l'utiliser pour sortir un autre *JSON*.

Ensuite, ce projet nous a permis de comprendre et d'appliquer des principes de qualités. Ainsi, nous savons dorénavant utiliser *sonar* et *pitest*, ainsi qu'utiliser des automatisations de tâche qui permettent de mettre en production du code ou bien d'en tester. Nous avons aussi appris à utiliser des stratégies de *branching* pour pouvoir contribuer à des projets, ou pour pouvoir produire du code de manière « industrielle ».

Tous ces éléments sont primordiaux pour apprendre à créer des programmes de qualités.

Durant ce projet, nous avons eu l'occasion d'appliquer des connaissances apprises dans d'autres cours. Par exemple, grâce aux connaissances du projet ps5, nous avons pu d'appliquer et mieux comprendre des concepts tels que les principes *SOLIDs* pour produire du code qui puissent être facile à entretenir (ouvert à l'extension, mais fermée à la modification) ou encore l'utilisation basique de *gitHub*.

Nous avons aussi utilisé notre expérience pour produire du code en équipe et d'utiliser le langage java.

Également, les concepts vus en Programmation orientée objet ont servi. Savoir raisonner en orientée objet, avec le langage Java. Les notions vues en algorithmique et structure des données ont servi pour l'élaboration du *pathFinding* en particulier l'algorithme de Dijkstra.

Enfin, nous avons appris à utiliser des spécifications d'un projet et sa documentation, à les lire et à les comprendre pour nous en servir pour mettre en œuvre notre projet.

Nous retenons de ces quatre mois que la qualité d'un code est vraiment nécessaire pour la stabilité et la pérennité d'un projet informatique. Utiliser les principes de qualités est de ce fait, le meilleur moyen pour obtenir un programme fonctionnel et maintenable.

Annexes

Pit Test Coverage Report

Project Summary

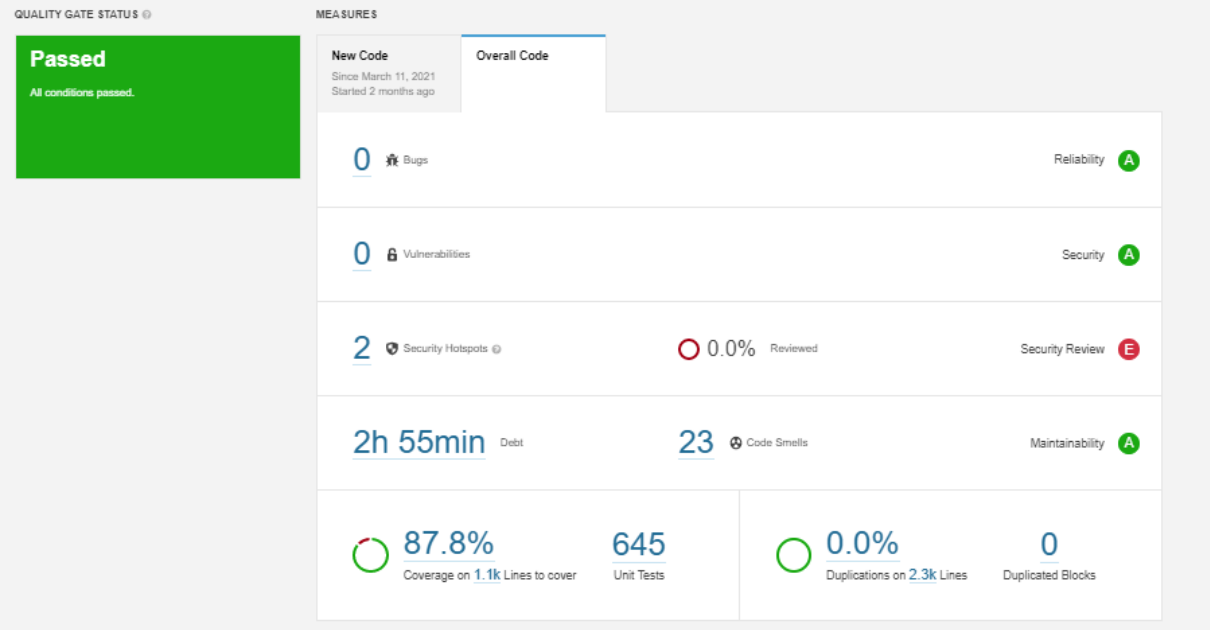
Number of Classes	Line Coverage		Mutation Coverage	
51	92%	1023/1114	76%	796/1048

Breakdown by Package

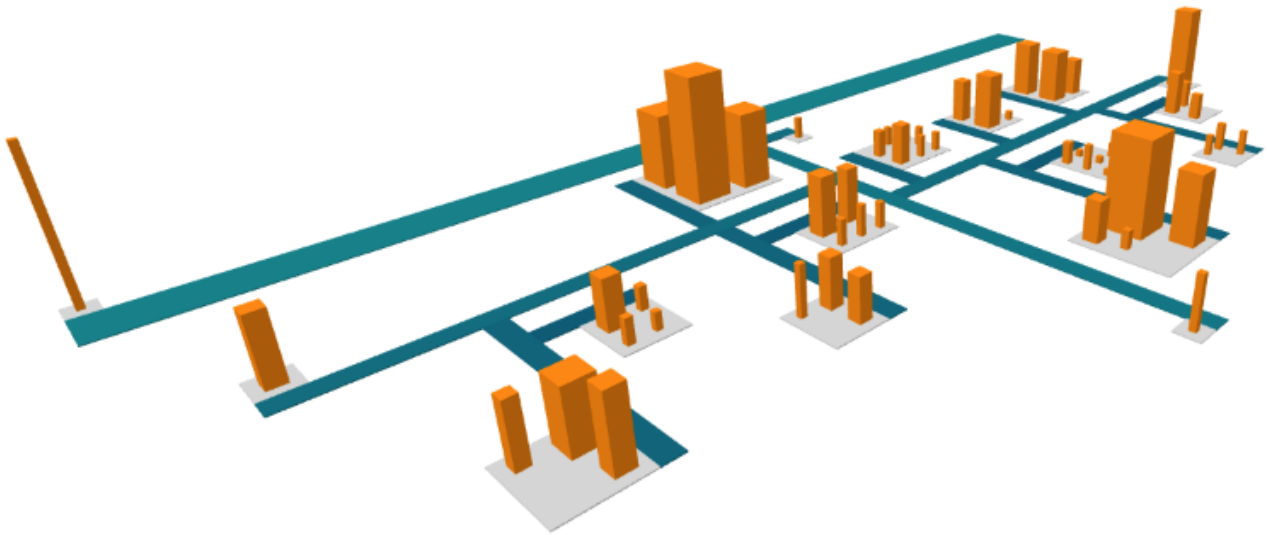
Name	Number of Classes	Line Coverage		Mutation Coverage	
fr.unice.polytech.si3.qgl.queleglitch	1	50%	12/24	33%	3/9
fr.unice.polytech.si3.qgl.queleglitch.enums	1	17%	1/6	0%	0/3
fr.unice.polytech.si3.qgl.queleglitch.game	1	88%	29/33	42%	8/19
fr.unice.polytech.si3.qgl.queleglitch.game.building	3	97%	57/59	78%	31/40
fr.unice.polytech.si3.qgl.queleglitch.game.building.smartmoving	5	99%	94/95	91%	51/56
fr.unice.polytech.si3.qgl.queleglitch.game.pathfinding	5	93%	198/213	73%	210/288
fr.unice.polytech.si3.qgl.queleglitch.game.resolver	3	98%	91/93	84%	124/147
fr.unice.polytech.si3.qgl.queleglitch.game.resolver.strategie	4	98%	61/62	78%	56/72
fr.unice.polytech.si3.qgl.queleglitch.json	1	82%	46/56	55%	12/22
fr.unice.polytech.si3.qgl.queleglitch.json.action	7	85%	61/72	41%	23/56
fr.unice.polytech.si3.qgl.queleglitch.json.game	4	93%	112/120	88%	80/91
fr.unice.polytech.si3.qgl.queleglitch.json.game.entitie	3	92%	23/25	54%	7/13
fr.unice.polytech.si3.qgl.queleglitch.json.goal	2	97%	57/59	93%	43/46
fr.unice.polytech.si3.qgl.queleglitch.json.nextround	3	97%	36/37	80%	8/10
fr.unice.polytech.si3.qgl.queleglitch.json.nextround.visibleentities	3	91%	40/44	71%	24/34
fr.unice.polytech.si3.qgl.queleglitch.json.shape	5	91%	105/116	82%	116/142

Report generated by PIT 1.4.11

Pitest



Sonar



SoftVis3D Viewer