Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

UNIVERSITÉ DE NAMUR

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Learning to assert in software testing using mutants

VAN de PUT, Samuel

Award date:
2022

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

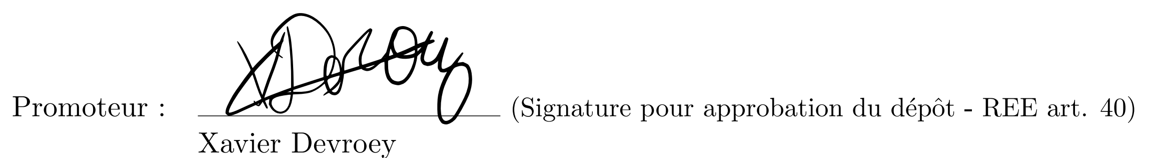
Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 04. May. 2026

Université de Namur
Faculté d'informatique
Année académique 2021–2022

Learning to assert in software testing using mutants

Van de Put Samuel



Promoteur : (Signature pour approbation du dépôt - REE art. 40)
Xavier Devroey

Co-promoteur : Benoit Vanderose

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Remerciements

J'adresse mes remerciements sincères à mes deux promoteurs, le Professeur Xavier Devroey et le Professeur Benoît Vanderose pour leur attention, leur aide et leurs conseils apportés tout au long de la réalisation de ce mémoire de fin d'études. Mes reconnaissances s'adressent également aux étudiants qui m'ont aidé et qui ont participé à l'expérience menée dans ce mémoire. Je remercie également mes proches pour leurs soutien durant la réalisation de ce mémoire.

Résumé

L'apprentissage des tests de logiciels est une matière délaissée dans les cursus informatique. Au fil des années, des méthodes et outils sont apparus pour fournir un support pédagogique vis-à-vis de cet apprentissage. Les tests par mutation est une technique utilisée pour évaluer l'efficacité des suites de test. Récemment, une variante appelée tests par mutations extrêmes qui réduit les coûts en calcul et en temps est apparue. Descartes, un moteur de mutation extrême a été développé. Avec l'appui d'une extension de plugin appelée Reneri, Descartes peut générer un rapport fournissant des informations au développeur sur les raisons potentielles pour lesquelles les mutants restent non détectés. Dans ce mémoire, une extension de Visual Studio Code a été développée afin d'incorporer les informations générées par Descartes et Reneri. Le but de l'expérience est d'évaluer si l'inclusion de ces données peut aider des étudiants en master à améliorer les assertions de leurs tests. Les résultats ont démontré que ces informations intégrées dans un éditeurs ont bien été reçues par les étudiants et qu'elles les ont guidé vers un affinement de leur suite de tests.

Abstract

Learning software testing is a neglected subject in computer science courses. Over the years, methods and tools have appeared to provide educational support for this learning. Mutation testing is a technique used to evaluate the effectiveness of test suites. Recently, a variant called extreme mutation testing that reduces computational and time costs has emerged. Descartes, an extreme mutation engine was developed. With the support of a plugin extension called Reneri, Descartes can generate a report providing information to the developer on potential reasons why mutants remain undetected. In this thesis, an extension of Visual Studio Code has been developed in order to incorporate the information generated by Descartes and Reneri. The purpose of the experiment is to assess whether the inclusion of this data can help master's students improve their test assertions. The results showed that this information integrated into an editor was well received by the students and that it guided them towards a refinement of their suite of tests. .

Table des matières

1 Introduction **6**
- 1.1 Contexte . . . 6
- 1.2 Objectifs . . . 7
- 1.3 Question de recherche . . . 7
- 1.4 Plan du mémoire . . . 7

2 Etat de l'art **8**
- 2.1 Mutation testing . . . 8
  - 2.1.1 Le processus du Mutation Testing . . . 9
  - 2.1.2 Validité . . . 11
  - 2.1.3 Outils de Mutation Testing . . . 12
- 2.2 Extreme Mutation testing . . . 14
  - 2.2.1 PITest Descartes . . . 16
  - 2.2.2 Descartes Reneri . . . 16
- 2.3 Education du software testing . . . 17
- 2.4 Difficultés rencontrées . . . 17
  - 2.4.1 Difficultés rencontrées par les étudiants . . . 17
  - 2.4.2 Difficultés rencontrées par les professeurs . . . 18
- 2.5 Outils et approches existants . . . 19
  - 2.5.1 Développement piloté par les tests . . . 20
  - 2.5.2 Gamification . . . 21
  - 2.5.3 Outils collaboratifs . . . 22
  - 2.5.4 Évaluation par les pairs . . . 23

3 Conception **24**
- 3.1 Définition des bases du travail . . . 24
  - 3.1.1 Choix de l'EDI . . . 24
  - 3.1.2 Choix du langage . . . 24
- 3.2 Solution proposée . . . 24
  - 3.2.1 Exigences fonctionnelles . . . 25
  - 3.2.2 Exigences non-fonctionnelles . . . 25
  - 3.2.3 Maquette . . . 25
  - 3.2.4 Cas d'utilisation . . . 28
  - 3.2.5 BPMN . . . 30
- 3.3 Implémentation . . . 30
  - 3.3.1 Exécuter le moteur de mutation de Descartes . . . 31
  - 3.3.2 Exécuter les commandes d'observations de Reneri . . . 33
  - 3.3.3 Afficher les décorations . . . 37
  - 3.3.4 Cacher les décorations . . . 41
  - 3.3.5 Consulter les rapports HTML de PITest . . . 42
  - 3.3.6 Consulter le score de mutation . . . 42

3.3.7 Consulter la liste des méthodes pseudo ou partiellement testées . . . 43
3.3.8 Wizard . . . 44

4 Discussion 46
4.1 Tests d'intégration . . . 46
4.2 Maintenance et pérennité . . . 47

5 Validation 48
5.1 Methodologie . . . 48
5.2 Résultats . . . 50
5.3 Conclusion . . . 54

6 Travaux futurs 55
6.1 Amélioration des conseils . . . 55
6.2 Consulter les résultats précédents . . . 55
6.3 Support à l'utilisation . . . 56
6.4 Consulter les documents HTML de PITest . . . 56
6.5 Paramétrage et visualisation . . . 57
6.6 Service web et base de données . . . 57
6.7 Gamification . . . 57
6.8 Automatisation . . . 58
6.9 Accès rapide aux méthodes et tests . . . 58
6.10 Améliorer la validation . . . 58

7 Conclusion 59

8 Annexes 64

Table des figures

1 Code based mutation testing process 10
2 µJava 13
3 Exemple de rapport fournit par PIT 14
4 Maquette de l'extension 27
5 Maquette de l'extension 27
6 Diagramme de cas d'utilisation de l'extension 29
7 Diagramme BPMN de l'extension 30
8 Diagramme de classe des rapports de Descartes 34
9 Diagramme conceptuel des données générées par Reneri . . . 37
10 Bouton d'activation des surbrillances. 38
11 Décoration générée sur base des conseils de Descartes pour une méthode. 39
12 Décoration générée sur base des conseils de Descartes pour une méthode non couverte. 40
13 Décoration générée sur base des conseils de Descartes pour un test. 40
14 Décoration générée sur base des conseils de Reneri pour un test. 41
15 Implémentation de la barre de status. 43
16 Barre de statut affichant le score de mutation. 43
17 Dialogue affichant les mutations survivantes. 44
18 Première étape du Wizard. 45
19 Questionnaire d'expérience utilisateur (UEQ). 50
20 Résultats du questionnaire. 51
21 Benchmark du plugin LASoT. 52
22 Vue Web intégrée à l'EDI. 56
23 UEQ : Résultat par question. 64
24 UEQ : Distribution des réponses. 65
25 UEQ : Variance par aspect qualité. 65

1 Introduction

1.1 Contexte

Alors que nous entrons dans la 3ème décennie du 21e siècle, la qualité des logiciels devient de plus en plus indispensable à toutes les entreprises et la connaissance des tests devient nécessaire pour tous les ingénieurs en logiciel. Aujourd'hui, notre civilisation dépend de beaucoup de systèmes informatiques tels que les routeurs de réseau, les moteurs de calcul financier, les réseaux de commutation, le Web, réseaux électriques, etc. L'industrie du logiciel est devenue beaucoup plus grande depuis ces trois dernières décennies. Elle compte beaucoup plus de logiciels, elle est plus compétitive et compte plus d'utilisateurs. Le logiciel est un élément essentiel d'applications embarquées telles que les avions, les vaisseaux spatiaux et les systèmes de contrôle du trafic aérien, ainsi que des appareils banals tels que des montres, des voitures et des téléphones portables. Les tests sont le principal moyen utilisé par l'industrie pour évaluer les logiciels pendant leur développement. Les tests unitaires (se concentrant sur une unité du programme) sont fortement mis en avant dans le processus de développement dirigé par les tests et ils sont les clés pour la validation des exigences fonctionnelles. Les tests ont une plus grande signification pour le succès des produits logiciels.

L'un des principaux défis lors de l'apprentissage du test logiciel est de définir de bons oracles pour les tests. Un oracle est un mécanisme permettant de décider si un test passe ou échoue en fonction de son exécution. Pour les tests unitaires, un oracle prend la forme d'assertions ajoutées au code des tests unitaires pour vérifier des valeurs de sortie spécifiques. Lorsque les tests sont réalisés, comment pouvons-nous être sûr de la qualité de notre suite de tests? Sont-ils suffisamment bons pour révéler les fautes? Au fil des années, plusieurs techniques, comme la couverture des lignes ou des branches, ont été développées pour permettre aux développeurs de vérifier la qualité de leurs tests. Parmi ceux-ci, le mutation testing injectent des défauts, à l'aide d'opérateurs de mutation, dans un code source donné afin de vérifier si les tests actuels ont la capacité de les détecter.

Récemment, Vera-Perez et al. [32] ont développé Descartes[1], un moteur de mutation s'appuyant sur des opérateurs de mutation extrêmes pour évaluer la qualité des assertions d'une suite de tests. Ces chercheurs ont également développé un plugin nommé Reneri[2] qui observe l'exécution des transformations extrêmes non détectées signalées par Descartes. Sur base de ces observations, ce plugin génère des rapports fournissant des informations au développeur sur les raisons potentielles pour lesquelles les mutants ne sont pas détectés.

[1] https://github.com/STAMP-project/pitest-descartes
[2] https://github.com/STAMPproject/descartes-reneri

1.2 Objectifs

Le but de ce mémoire est de développer une extension d'un Environnement de développement (EDI) qui inclut les informations des plugins Descartes et Reneri. Et d'évaluer dans quelle mesure l'incorporation de ces données dans l'éditeur de code des étudiants peut les aider à affiner les assertions de leurs tests et à l'enseignement des tests de logiciels.

1.3 Question de recherche

Comment utiliser les rapports fournis par les plugins Descartes et Reneri dans un contexte éducatif afin d'aider des étudiants à affiner les assertions de leurs tests?

Sur base de cette question de départ et afin d'y répondre au mieux, d'autres sous-questions sont émergées :

- Quelles sont les difficultées rencontrées par les étudiants et les professeurs dans le domaine de l'éducation du software testing?
- Quelles sont les préférences des développeurs du point de vue visuel concernant l'incorporation des données externes dans un EDI?

1.4 Plan du mémoire

La conception d'une extension d'un EDI est le fruit de plusieurs étapes détaillées dans ce mémoire.

Le chapitre suivant consiste en un état de l'art à propos du mutation testing, de l'éducation du software testing en générale, des difficultés rencontrées par les étudiants dans ce domaine et une exploration des outils existants qui implémentent ces techniques. Une recherche à propos des capacités des plugins fournies par les différents EDI a également été menée.

Dans le troisième chapitre de ce mémoire, les différentes étapes de la réalisation du plugin sont détaillées, depuis l'analyse des exigences jusqu'au choix de l'environnement de développement et du langage de programmation. Finalement, une expérience a été menée afin d'évaluer les capacités de l'outil par des étudiants en master.

2 Etat de l'art

Ce chapitre a pour objectif d'apporter suffisamment de background à propos du mutation testing et pour aider à comprendre les difficultés rencontrées par les étudiants dans ce domaine.

Les volets concernant mutation testing permet de comprendre les principes techniques des tests de logiciel cités dans ce mémoire. Ensuite, le thème est abordé au point de vue éducationnel (section 2.3). Dans cette section, il est nécessaire de s'attarder sur les difficultés (section 2.4) que les étudiants et professeurs peuvent éventuellement rencontrer concernant l'ajout de concepts de tests de logiciels dans les cours de programmation. Puis, les méthodes existantes afin de subvenir à ces problèmes sont exploitées (section 2.5).

2.1 Mutation testing

Software testing est une pratique populaire pour identifier des bugs [1]. Ce processus est effectué en exercant des cas de test si un système déterminé (System under test, SUT). Ces tests contrôlent si ce dernier se comporte comme souhaité. Pour analyser la précision des tests, il existe plusieurs critères qui spécifient les exigences du testing. Quand les critères sont remplis, ils fournissent une certaine confiance dans le système qui est testé. Des études empiriques menées par Checkam et al. [5] ont démontré que le mutation testing est efficace pour détecter des fautes, et est capable d'englober presque toutes les techniques de tests de logiciels [1].

Le mutation testing réalise l'idée d'utiliser des défauts artificiels pour soutenir les activités de tests. La mutation est généralement utilisée comme moyen pour évaluer l'adéquation du test, pour guider la génération de cas de test et soutenir l'expérimentation. Le mutation testing a atteint une phase de maturité et gagne progressivement en popularité à la fois dans le milieu universitaire et dans l'industrie.

Pouvons-nous être confiant dans notre suite de tests? L'analyse par mutations répond à cette question en vérifiant la capacité de nos tests à détecter des erreurs artificielles. Dans le cas où nos tests échouent à les détecter, nous ne devrions pas être confiant en nos tests et devrions les améliorer. Les tests par mutation réalisent cette idée et mesurent la confiance que notre suite de tests nous inspire.

L'analyses par mutation est un processus qui transforme automatiquement la syntaxe d'un programme avec le but de produire des variantes sémantique de celui-ci. Dans le cas du mutation testing, en générant des défauts artificiels. Ces variations du programme sont appelées des mutants. Les tests par mutation se réfèrent au processus d'utilisation de l'analyse par mutation pour appuyer le testing en quantifiant la robustesse de la suite de

tests [1].

Donc, les tests qui sont capables de distinguer le comportement des programmes mutants de la version originale du programme remplissent les objectifs. Quand un test détecte un mutant, on dit que le mutant est "tué" ou "detecté". Dans les autres cas, on dit que le mutant est vivant. L'idée du mutation testing est de forcer les développeurs à penser différemment et à concevoir des tests qui révèlent des bugs cachés.

Le nombre de publications scientifiques qui s'appuient sur l'analyse par mutations est en constante augmentation. Le Mutation testing a été utilisé pour presque toutes les formes de testing. Principalement le Unit testing mais également à d'autres niveaux comme spécification, conception, integration et niveaux système. Cette méthode a été appliquée sur les langages les plus populaires comme le C, C++, Java, CSharpe, Javascript, Ruby et également des langages de modélisation.

Les mutants sont générés en altérant la syntaxe du programme. Donc, il y a des règles de transformations syntaxiques appelées les "opérateurs de mutation" qui définissent comment introduire des changements syntaxiques dans le programme. Un opérateur mutant arithmétique altère un opérateur arithmétique du langage de programmation, changeant le '+' en '-' par exemple.

Donc, l'objectif est de concevoir des tests qui tuent les mutants. On peut définir le score de mutation comme le rapport entre les mutants qui sont tués par nos tests et le nombre total de transformations générées. Essentiellement, ce score indique le degré de réalisation de nos cas de test dans la réalisation des objectifs de test. Ainsi, le score de mutation peut être utilisé comme indicateur d'adéquation. [1].

2.1.1 Le processus du Mutation Testing

Le processus traditionnel du mutation testing commence par la sélection d'un ensemble de mutants qu'on souhaite appliquer à notre suite de tests. Pour se faire, il faut sélectionner un ensemble d'opérateurs de mutation qui vont définir les transformations syntaxiques qui seront appliquées au programme original. Cette opération peut engendrer un très grand nombre de mutants. Il existe des opérateurs de mutation pour des langages de programmation spécifiques comme par exemple AspectJ qui est un langage orienté aspect ou des langages de niveau intermédiaires comme la compilation d'un programme C# ou encore des mutants générés dynamiquement à l'exécution du programme ce qui permet de profiter de l'information de typage.

Ensuite, il est nécessaire de retirer des mutants problématiques. Parmi ceux-ci, les mutants équivalents sont des transformations du programme original qui sont sémantiquement équivalentes à celui-ci malgré leurs différences syntaxiques. D'autres, les mutants redondants sont ceux qui sont tués systématiquement lorsque d'autres sont détectés. Ces mutants posent un

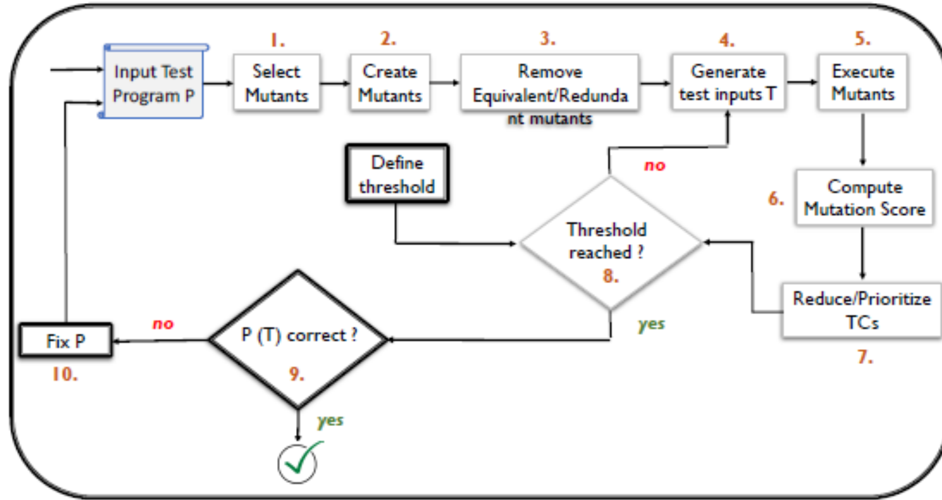


Figure 1: Code based mutation testing process

réel problème dans le calcul du score de mutation. Les mutants redondants engendrent un score de mutations plus élevé. Pour éviter ce problème d'inflation, seul les mutants qui contribuent au score de mutation devraient être utilisés [27].

Une fois que l'ensemble des mutants est formé, une suite de test basée sur ceux-ci peut être créée (étape 4). Ces tests peuvent être générés manuellement ou automatiquement de manière à ce qu'ils aient le potentiel de détecter les mutants. Le modèle RIPR[1](Reachability, Infection, Propagation, Revealability) définit quatres conditions qui peuvent être utilisées pour générer des tests. Pour qu'un bug soit observé il faut qu'un test puisse atteindre ce mutant (Reachability). Ce bug doit causer un changement sur l'état du programme et manifester ce changement par propagation vers la sortie observable accessible par le test (Revealability).

Ensuite ces mutants sont exécutés (étape 5). Cette étape est la plus coûteuse du processus car elle consiste en l'exécution des tests pour chaque mutation. Pour un nombre n de tests et m de mutations, il sera compté au maximum n*m exécutions. Étant donné le nombre de mutants élevé, en générale, ceci rend le processus chronophage. Plusieurs solutions ont été proposées afin de réduire le coût de ce processus. Dans un premier scénario, il est proposé d'exécuter un mutant jusqu'à ce qu'il soit détecté. Une optimisation supplémentaire à ce scénario est d'exécuter les tests dans un certain ordre. D'autres améliorations consistent à retirer des mutations qui ne sont pas détectables ou qui génèrent des boucles infinies. Une fois l'exécution des mutants terminée, il est possible de déterminé la robustesse des tests en calculant le score de mutation (étape 6). Ce nombre est le rapport entre les

mutants tués et ceux qui ont survécus. Ensuite, la suite de test est réduite ou priorisée (étape 7). C'est à dire que les tests qui n'ont pas d'impact sur le score de mutation sont retirés et les tests restant sont triés de manière à ce que les mutants soient détectés le plus tôt possible. L'étape suivante consiste a évaluer si le score de mutation est acceptable. Les étapes de 4 à 7 sont répétées dans le cas ou le score de mutation n'atteint pas le seuil défini. Ce seuil reflète le niveau de confiance qu'on les développeurs en leurs tests. Chekam et al. [5] affirment, suite à leur étude, que les tests de mutation forte (strong mutation), c'est à dire des mutations qui exposent une variation en sortie par rapport au programme original, permettent de produire une révélation de défauts élevée. Et seuls, les niveaux les plus élevés de couverture de mutation (mutation coverage) bénéficient d'un fort potentiel révélateur de bugs. Les testeurs ne doivent donc pas être confiant d'une suite de test qui n'atteint pas un score de mutation élevé. La dernière partie du processus consiste à concevoir des oracles de tests qui assurent que le programme ait le comportement souhaité. Des recherches ont été effectuées dans le but d'automatiser cette tâche. Des outils comme EvoSuite[3][13] ont été développés afin de générer des oracles de tests en s'appuyant sur l'analyse par mutation. En cas d'erreurs, les développeurs doivent corriger les fautes du programme. Des techniques basées sur les mutants comme "Metallaxis" permettent de localiser ces fautes de manière efficace. D'autres techniques basées sur les mutants permettent de corriger ou débuguer ces fautes. Ces techniques se basent sur le fait que beaucoup de fautes sont corrigées par de simples changement syntaxiques. Les mutants sont donc de bons candidats pour effectuer ces corrections. Le processus est relancé jusqu'à ce qu'on ne retrouve plus de fautes et que le score de mutation est acceptable.

2.1.2 Validité

Le principe du mutation testing est de créer des bugs et de concevoir des cas de tests qui les détectent. Cette technique fournit un mécanisme par lequel il devient possible de falsifier l'efficacité des tests. Si certains mutants ne sont pas détectés, cela suggère une incapacité à détecter certains types de défauts. Offut et al.[22] ont démontré qu'une suite de test qui détecte des mutants de premier ordre, peut également détecter des mutations plus complexes. C'est ce qui est appelé le "Coupling Effect". Donc le mutation testing permet de révéler un ensemble plus grand de bugs.

Il faut tout de même rester vigilant quant à l'utilisation du score de mutation comme mesure d'adéquation car sa précision est questionnable [27]. Malheureusement tous les mutants ne sont pas égaux. Comme expliqué dans l'étape 3 du processus du mutation testing basé sur le code, certains d'entre

[3]https://www.evosuite.org/

eux sont des mutants équivalents ou redondants. Ils ont un impact sur le score de mutation mais pas sur la capacité de la suite de tests de détecter des fautes. Depuis les années 1970, les chercheurs ont travaillé dans le domaine afin de résoudre ce problème soit en détectant ces mutations, soit en les évitant. Chaque technique a ses avantages et inconvénients, toutefois, les techniques d'optimisation du compilateur, les mutations de haut rang, l'approche de résolutions par contraintes ou l'approche basée sur l'impact sur la couverture sont préférables par rapport à la détection et classification manuelle de mutants.

La qualité des mutants est importante. Dans le cas ou les mutants utilisés sont triviaux, le résultat n'indiquera qu'une couverture d'une partie du code mais pas la capacité des tests à détecter des fautes. Les mutants redondants sont une grande menace à la validité car ils faussent le score de mutation. Une solution proposée serait l'utilisation de mutants disjoints qui est un petit ensemble de mutants représentatif de l'ensemble complet des mutants. On considère que la suite de test qui détecte l'ensemble réduit de mutants, détecte également l'ensemble complet de mutants. Calculer cet ensemble est impossible mais on peut s'en approcher par approximation par une suite de test.

Certaines recherches ont été effectuées concernant la capacité du mutation testing à révéler des fautes réelles [25]. Papadakis et al. ont trouvé que certains mutants sont capables de représenter le comportement de fautes réelles. Toutefois, la majorité ne le sont pas, ce qui signifie que le score de mutation est gonflé et la corrélation entre le score de mutation et la capacité à détecter des bugs est faible. Titcheu et al.[31] ont récemment proposé une autre perspective qui consiste en la sélection de mutants qui révèlent des fautes. Cette technique se base sur l'utilisation du "machine learning" pour aider à la sélection. Leur travail constitue une première étape dans cette lutte contre les mutants triviaux et des recherches futures dans cette direction permettraient d'améliorer les résultats.

2.1.3 Outils de Mutation Testing

Appliquer le mutation testing à des programmes du monde réel nécessite l'utilisation de frameworks automatisés. Le tableau exhaustif des applications existantes depuis l'année 2001 présenté par Amman et al. [1] comprend 76 outils et démontre une apparition croissante de ceux-ci. Au niveau de l'implémentation, les outils sont ciblés majoritairement dans les langages de programmation Java et C. Afin de générer un résultat fiable, ces outils doivent prendre en considération l'élimination des mutants équivalents et redondants. Kintis et al. [17] ont évalué la capacité de quatre des outils les plus populaires utilisés pour Java. Trois de ceux-ci sont présentés plus en détail dans cette section.

- µJava (muJava)[4], sorti en 2003, est un des plus anciens système de mutation pour des programmes en Java. Il génère automatiquement des mutants sur base d'opérateurs de méthodes et de classes. La figure 2 présente la vue d'une mutation traditionnelle. On peut observer le code original et la mutation. Sur la gauche de celle-ci est listé l'ensemble des opérateurs utilisés. Par exemple, *AOIS* insert des opérateurs arithmétiques comme "++" et un *AODS* les supprime.

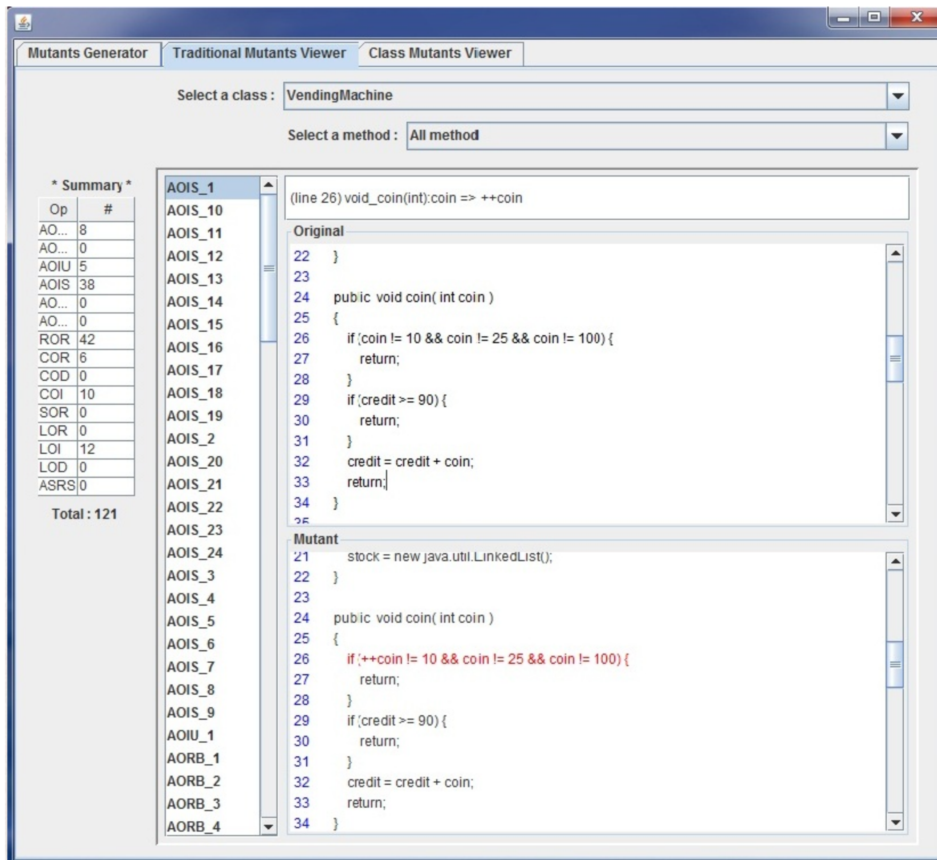


Figure 2: µJava

- PIT (PITest)[5][10], sorti en 2010, est un framework open source qui cible l'utilisation du mutation testing dans le domaine industriel. Il utilise la manipulation du "bytecode" du programme ciblé ce qui lui permet de gagner en rapidité. PIT utilise des opérateurs de mutation qui affectent des fonctionnalités primitives du langage de programmation de la même façon que *muJava*. Il implémente néanmoins différemment certains opérateurs de mutation spécifiques par rapport

[4]https://cs.gmu.edu/ offutt/mujava/
[5]https://pitest.org/

à *muJava*. Additionnellement, PIT apporte de nouveaux opérateurs tel que le "Void Method Call" (VMC) qui supprime les appels de méthode qui ne renvoient rien. PIT fournit des rapports faciles à lire sous format HTML. Un exemple est donné à la figure 3. Les lignes vertes montrent la couverture de ligne et celles vert foncé montrent la couverture de mutation. Les lignes rose clair et rose foncé montrent respectivement un manque de couverture de code ou un manque de couverture de mutation. En 2017, Laurent et al.[19] ont effectué des recherches à propos des faiblesses de PIT concernant la qualité des mutants. Ils l'ont équipé avec une extension d'opérateurs qui le rend plus robuste et plus fiable. Ces opérateurs sont intégrés dans la version officielle de PIT et sont exposés dans une version externe sur GitHub[6].

```
122              // Verify for a ".." component at next iter
123 3            if ((newcomponents.get(i)).length() > 0 
124              {
125                  newcomponents.remove(i);
126                  newcomponents.remove(i);
127 1                i = i - 2;
128 1                if (i < -1)
129                  {
130                      i = -1;
131                  }
132              }
133          }
```

Figure 3: Exemple de rapport fournit par PIT

- Major[7], est un framework de mutation testing. Contrairement aux deux outils précédents, il manipule l'arbre syntaxique abstrait (AST) du programme testé. Ce type de conception architecturale lui permet d'être intégré au compilateur Java, d'être non invasif et de réduire le temps de génération des mutants. Il utilise des opérateurs de mutation qui ont une portée similaire à la celle de *PIT* et *muJava*.

La comparaison effectuée par Laurent et al.[19] sur bases de fautes réelles démontre que l'extension d'opérateurs de PIT fournit de meilleurs résultats, jusqu'à 6% de défauts détectés en plus, par rapport aux trois autres outils. L'efficacité de la méthode dépend donc fortement des particularités des outils employés.

2.2 Extreme Mutation testing

Niedermayr et al. [21] ont introduit l'analyse par mutation extrême. C'est une méthode alternative à la mutation traditionnelle qui consiste à effectuer

[6] https://github.com/laurenttho3/extendedpitest
[7] https://mutation-testing.org/

des transformations plus grandes en éliminant d'un seul coup, tous les effets de bord d'une méthode. Par exemple, une méthode **void** se verra vidée complètement de toutes ses instructions. Pour d'autres types de retour, le corps des méthodes est remplacé par une expression **return** et une valeur prédéfinie. Le testing par mutation extrême apporte une réponse aux désavantages du mutation testing. Il crée moins de mutants et évite la plupart des mutants équivalents. Ce qui ne signifie pas qu'il les supprime totalement. En plus du score de mutation, cette analyse fournit une liste des méthodes les plus mal testées. Ces méthodes sont appelées pseudo-testées si aucune mutation extrême appliquée sur celle-ci n'a été détectée par la suite de tests alors que ces derniers couvrent cette méthode. Dans le code source montré dans le listing 1 la méthode *incrementCount()* de la classe M est pseudo testée. En effet, le corps de cette méthode est retiré dans le mutant et le test *testM* déclenche l'appel de cette méthode mais n'évalue pas ses effets. Le mutant n'est donc pas détecté.

```
1  public class M {
2      private int count = 0;
3
4      public int m(int x) {
5          if (x == 5) {
6              incrementCount();
7              return 3;
8          }
9          return (x * 8);
10     }
11
12     private void incrementCount(){
13         count++;
14     }
15 }
16
17 public class TestM{
18     @Test
19     public void testM(){
20         M m = new M();
21         int result = m.m(5);
22         Assert.assertEquals(3, result);
23     }
24 }
25
26 // Mutant
27 private void incrementCount() { }
```

Listing 1: Exemple de méthode pseudo testée

2.2.1 PITest Descartes

Descartes est un moteur de mutation extrême pour **PIT**. Il apporte un ensemble d'opérateurs de mutation extrême à **PIT** et permet de détecter les méthodes pseudo-testées [32]. Descartes peut générer les rapports fourni par PIT sous différents formats (*XML*, *CSV* et *HTML*). Descartes fourni également des nouveaux rapports :

- Un rapport général supportant l'extension *JSON*.
- Une extension de rapport (appelée METHOD) conçue pour Descartes qui génère un fichier *json* avec des informations à propos des méthodes pseudo testées ou partiellement-testées.
- Un rapport lisible par l'homme qui contient seulement la liste des méthodes pour lesquelles il y a des problèmes de test, appelé ISSUES.

Le rapport METHOD contient la liste détaillée des méthodes faisant partie de l'analyse. Pour chacune d'entre elles, on peut retrouver des méta-données, leur classification et les mutations appliquées. Le Rapport ISSUES est un document sous format *HTML* reprenant des informations générales de l'analyse et des détails sur les classes contenant des méthodes signalées.

Dans les rapports fournis, une méthode est dite :

- Non couverte si elle n'est pas couverte par la suite de tests.
- Pseudo-testée si elle est couverte par la suite de tests mais aucune mutation extrême appliquée à la méthode n'a été détectée par aucun cas de test.
- Partiellement testée si elle a des résultats mitigés. C'est à dire que des mutations ont étés détectées et d'autres ne l'ont pas été.
- Testée si toutes les transformations extrêmes sont détectées par la suite de tests.

2.2.2 Descartes Reneri

Reneri est un plugin qui génère des conseils pour améliorer une suite de tests en observant l'exécution de transformations extrêmes non detectées découvertes par *Descartes*. Ce plugin utilise donc les rapports de *Descartes* pour générer ces conseils.

Ce plugin fournit trois commandes *Maven*. Deux commandes pour observer l'exécution des méthodes signalées et des tests qui exécutent ces méthodes. Ces commandes reportent les différences observées. La troisième commande génère des conseils d'amélioration en fonction des résultats obtenus avec l'exécution des deux commandes précédentes.

Dans le cadre de ce mémoire, les outils Descartes et Reneri vont être utilisés afin d'incorporer les informations fournies dans les rapports générés par ceux-ci dans un EDI. Ces intégrations seront évaluées dans le but de savoir si celles-ci peuvent aider les étudiants à améliorer leur suite de tests.

2.3 Education du software testing

Actuellement, dans le domaine des technologies informatiques, il y a une forte demande de développeurs hautement qualifiés. Il y a une pénurie dans le domaine de la technologie en raison du manque de professionnels bien préparés. Malheureusement, il arrive régulièrement que les méthodes d'enseignement de la programmation ne répondent pas aux attentes des étudiants [1]. Les programmeurs novices souffrent d'un large éventail de déficits et de limitations professionnelles en raison d'une négligence de la composante des tests dans les programmes d'études en informatique.

Une enquête a été menée concernant la qualité des tests écrits par des étudiants. Il était question d'évaluer la capacité de ceux-ci à détecter de défauts authentiques créés par l'homme. Les résultats indiquent que, alors que les étudiants ont atteint une couverture de branche moyenne de 95,4% sur leurs propres solutions, leurs suites de tests n'ont pu détecter qu'une moyenne de 13,6% des défauts présents dans l'ensemble du programme. De plus, il y avait un degré élevé de similarité parmi 90% des suites de tests des étudiants. L'analyse des suites suggère que les étudiants utilisaient des assertions simples et suivaient des tests naïfs. Ils suivaient le "happy path testing" qui signifie qu'il écrivaient des tests basiques couvrant le comportement habituel plutôt que d'écrire des tests conçus pour détecter des bugs cachés. Ces résultats indiquent que les éducateurs devraient s'efforcer de renforcer les techniques de conception de test destinées à trouver des bugs, plutôt que de simplement confirmer que les fonctionnalités fonctionnent comme prévu [12].

2.4 Difficultés rencontrées

Il y a beaucoup d'approches différentes pour tester des logiciels et il est important de comprendre et de détecter les difficultés que les développeurs novices éprouvent et pourquoi certains ne testent pas du tout ou peu alors que ce serait bénéfique pour leur projet.

2.4.1 Difficultés rencontrées par les étudiants

En générale, les étudiants perçoivent le testing comme une tâche secondaire improductive parce qu'ils doivent générer une suite de tests sans une évidence tangible de ces effets bénéfiques. Ils ne font pas face aux défauts dont ces tests pourraient les aider à éviter. La difficulté signalée la plus répandue

dans les études était la réticence des étudiants à effectuer des tests de logiciels, même lorsqu'ils reconnaissent l'importance de cette pratique dans la résolution des devoirs de programmation [29].

Des chercheurs ont effectués une étude et ont interviewé des étudiants afin d'explorer leur expériences, attitudes et perceptions vis à vis d'un projet test. Cette étude a déterminé que les étudiants ne trouvaient pas de temps pour étudier le testing car le développement de projet en lui-même prenait la plupart du temps. Ceux-ci ont également perçu que le projet comme non critique et pas assez complexe pour justifier des tests. Les participants n'ont pas ressenti de satisfaction lors du testing par rapport à l'implémentation du projet. Certains se sont même senti improductifs lors du testing. Ils sont conscients du bénéfice du software testing mais ils savent aussi que ces techniques ont un coût qui n'est pas négligeable. Tout ceci les rend démotivés pour écrire des tests [28]. D'autre part les étudiants n'aiment pas tester leurs compétences en programmation en révélant les fautes dans leurs programmes [11].

Pour effectuer du testing (automatisé), les étudiants doivent apprendre des nouveaux outils et librairies qui ne sont pas toujours évidents à apprendre surtout lorsqu'il s'agit d'étudiants de première ou seconde année. Cet apprentissage peut devenir un challenge lorsqu'ils ont déjà des difficultés pour apprendre des concepts basiques de programmation. Souvent, les étudiants sont submergés par la quantité d'outils de tests de logiciels. Ils ne comprennent pas toujours le concept de trouver des bugs plutôt que de montrer simplement que leur programme fonctionne avec un ensemble d'entrées parfaites [3].

Un autre facteur qui affecte les étudiants par rapport au software testing est la charge cognitive ajoutée lors de l'apprentissage de ce concept. Lappalainen et al [18] a remarqué que transmettre le Test Driven Development à des étudiants novices en programmation est un challenge car cela augmente la charge cognitive technique.

2.4.2 Difficultés rencontrées par les professeurs

Le challenge le plus difficile lorsqu'on incorpore le Unit Testing dans un cours est de s'assurer que les étudiants surmontent leur idée négative pour qu'ils découvrent les bénéfices du testing fonctionnel.

Ajouter des activités pratiques de Software Testing à des cours de programmation est un idée solide. Cependant, il faut évaluer les élèves et leur donner des commentaires sur les façons de s'améliorer pour renforcer cette activité. Les outils d'évaluation existants utilisent des mesures de couverture de code, telles que la couverture de déclaration ou la couverture de branche pour évaluer la force des tests écrits par les étudiants. Malheureusement, ces mesures ont des limites. Dans le cas de l'expérience menée par Oliveira et al. [23], ils ont étudié la qualité des tests écrits par les étudiants en

fonction des bugs naturels que ceux-ci pouvaient détecter. Ils ont obtenu des résultats qui apportent l'évidence que le mutation testing peut faciliter l'apprentissage des programmeurs novices dans les tests.

Fournir des cours pratiques de software testing et proches des besoins industriels convient aux besoins des étudiants. Cependant, il y a un défi permanent de maintenir des cours afin de les maintenir alignés sur l'industrie. Du point de vue des éducateurs, il est difficile de maintenir des cours de testing à jour fournis avec des exercices réalistes [2]. Les étudiants perçoivent l'écriture de tests hors sujet dû à la petite taille des tâches de programmation. Il est difficile de trouver le bon cas d'étude qui est d'une complexité suffisante pour nécessité du testing mais pas trop complexe pour ne pas submerger les étudiants. Utiliser des projets réels peut aider mais en trouver qui sont open-source ou implémenté par l'instructeur n'est pas une chose banale. Additionnellement, il est possible que l'instructeur doive écrire une suite de tests avec le projet. D'un autre côté, si on demande aux étudiants d'écrire ces tests, les instructeurs ont la responsabilité de les évaluer. Au vu de la quantité de retours à fournir aux étudiants, il y a un besoin d'évaluer la qualité des tests automatiquement. Les professeurs ont un temps limité pour évaluer les étudiants et évaluer des milliers d'affectations dans un temps limité est un défi [6]. En générale, les étudiants reçoivent des feedbacks uniquement lors du résultat final ce qui ne leur permet pas d'augmenter la qualité de leur solution. Ils devraient recevoir des retours constamment afin d'avoir l'opportunité d'augmenter leur performance et d'apprendre de leurs erreurs [29]. Bufardi et al. [4] déclare que fournir des retours constants aux étudiants offre un renforcement immédiat à leur comportement.

Un challenge important à mentionner, est celui du temps. Il y a déjà trop de sujets à couvrir dans les cours d'ingénierie logicielle et il n'y a plus assez de temps pour enseigner les software testing [7].

Une approche commune afin de vérifier les tests des étudiants est la couverture de code. Mais ce paramètre à lui seul ne suffit pas pour évaluer la qualité des tests. Un étudiant pourrait très bien couvrir tout le code d'un programme sans écrire une seule assertion. Des techniques avancées comme le mutation testing pourraient aider vis à vis de ce problème mais malheureusement ce n'est pas bien connu en dehors des spécialistes et les outils permettant d'appuyer l'enseignement de ces méthodes ne sont pas satisfaisant [15].

2.5 Outils et approches existants

Une façon possible d'améliorer l'apprentissage des programmeurs novices est par l'introduction d'activités de software testing dans les salles de cours. Enseigner des techniques de test au tout début des cours de programmation peut représenter une pratique efficace pour améliorer les compétences des élèves.

Conceptuellement le Software Testing est un processus visant à vérifier si un logiciel correspond à ses spécifications. Les testeurs doivent utiliser des techniques et des critères de tests pour obtenir de meilleurs résultats. Ces techniques et critères fournissent une façon systématique de trouver des conditions qui permettent d'augmenter la probabilité de trouver des erreurs. Plusieurs recherches on été effectuées concernant l'application de ces pratiques dans des cours de programmation et celles-ci peuvent conduire à aider l'élève à apprendre des concepts de programmation plus rapidement et créer des habitudes d'application de techniques de test dans leurs systèmes en développement.

2.5.1 Développement piloté par les tests

L'activité de test la plus populaire est le TDD (Test Driven Development). C'est une approche récurrente dans les cours d'introduction à la programmation. Avec le TDD, les activités sont définies dans un certain ordre. Premièrement le programmeur développe des cas de test. Cet aspect est plus connu sou le nom de "test-first programming". Ensuite le développeur exécute les tests pour finalement écrire du code pour les tests qui échouent. Ces étapes sont répétées jusqu'à ce que le "code unit" soit complet [29].

En 2007 Gaspar et al. [16] intègre le concept de qualité en utilisant le TDD. Leur intention était de changer la concentration des étudiants non plus sur le résultats final mais sur l'exactitude du code. Cette méthode a permis de changer la perception des étudiants vis à vis de la programmation. Ils ne voient plus cette activité comme de l'implémentation simple mais plutôt comme une activité qui englobe plusieurs concepts comme la conception, l'implémentation, le testing et le débogage. L'objectif des auteurs étaient de motiver les étudiants à adopter une attitude plus défensive dans le développement. Cet expérience démontre que l'intégration du testing dans un cours de programmation est bénéfique et permet de leur ouvrir l'esprit sur d'autres concepts que l'implémentation.

Afin d'encourager les étudiants dans le TDD, Buffardi et al.[4] ont tiré parti d'un système de cotation automatisé **Web-CAT**[8] pour fournir des retours aux étudiants lorsqu'ils travaillent. Ce système fonctionne sur un serveur et fournit toutes ses fonctionnalités via des interfaces web. Via ce dernier, les étudiants peuvent soumettre leurs travaux et consulter leur évaluation. La qualité des tests est évaluée par des critères de couverture du code, de pourcentage d'instructions, de conditions et branches exécutées par les tests unitaires de l'étudiant. De plus, Web-CAT met en évidence lorsque la couverture de code est manquante. Les auteurs estiment que l'association fréquente du software testing avec des activités de débugage permet de faire

[8]https://web-cat.org/projects/Web-CAT/WhatIsWebCat.html

changer l'idée qu'on les étudiants envers le testing et de leur permettre de voir cette activité comme une façon de résoudre les problèmes plutôt que de les éviter.

Lapalainen et al. [18] ont mené une expérience en utilisant un outil nommé **ComTest** afin de réduire la charge cognitive des étudiants concernant le TDD. **ComTest** est un outil qui a été développé dans le but de faciliter la rédaction de tests et la compréhension en la lecture de ceux-ci. L'idée est d'écrire des tests unitaires dans des commentaires JavaDoc en utilisant un langage spécifique défini par de courtes et simples syntaxes. Les observations ont montré que les étudiants utilisant *ComTest* ont écrit plus de tests que les étudiants qui ont écrit des tests avec *JUnit*.

2.5.2 Gamification

Les développeurs perçoivent souvent le software testing comme ennuyant et difficile comparé à la programmation ou des activités de conception. Clegg et al.[9] ont introduit le jeu **Code Defenders**[9,2] afin de rendre l'éducation du testing plus agréable en engageant les étudiants dans des activités de testing de manière plus fun et plus compétitives. **Code Defenders** intègre les principaux aspects du mutation testing comme la création de mutants, la détection de mutants et la vérification d'équivalence de mutants dans un framework général qui implémente le jeu. Le principe du jeu comprend deux rôles, les attaquants et les défenseurs. Les attaquants créent des mutants les plus subtiles possible du programme en test et les défenseurs créent des tests les plus robustes possible de manière à pouvoir détecter toutes les mutations. Le framework intègre le concept de mutation équivalente. Les attaquants ont la possibilité de créer des mutants équivalents. Dans ce cas, les défenseurs peuvent prétendre une équivalence d'une mutation et les attaquants doivent fournir un test qui la détecte si celle-ci ne l'est pas ou l'accepter dans le cas contraire. Le jeu comprend également un mode individuel ou les tests sont générés par l'outil **EvoSuite**. Ces méthodes ont permis de développer des exercices qui enseignent un ensemble de concepts de software testing.

Fraser et al.[14] ont également expérimenté l'intégration de ce framework dans des cours pour des étudiants universitaires durant un semestre. Durant cette période, des activités hebdomadaires de sélection de code à tester, de gestion de jeux et d'évaluation de performance de tests ont été données. Les résultats de leur expériences ont démontrés que l'utilisation de **Code Defenders** était bien reçue par les étudiants et leur a permis de pratiquer pleinement des activités de test. La progression de leurs performances permet de détecter un effet positif sur leur apprentissage. La gamification peut résoudre le problème de motivation et de réticence des étudiants à tester

[9] https://code-defenders.org/

[2] http://github.com/jmrojas/codedefenders

leur propre code. Avec ce type de méthode, l'esprit de compétition prend le dessus.

2.5.3 Outils collaboratifs

Une étude menée par Oliveira et al. [24] a évalué l'utilisation de l'outil **Pascal Mutants** afin d'appliquer le mutation testing dans des programmes en Pascal. Cet outil est spécifiquement conçu pour appuyer le processus d'apprentissage. Les mutations sont souvent similaires à des programmes défectueux écrits par des étudiants novices. Cependant, le coût du mutation testing a toujours été élevé et il ne peut pas être appliqué sans un outil automatisé. Cet analyse a été effectuée sur un groupe d'étudiants de premier cycle. Leurs résultats montrent que l'utilisation du concept de mutation testing peut fournir une compréhension plus complète et plus précise de l'ensemble du fonctionnement d'un programme pour les programmeurs novices.

Les chercheurs ont continuer leurs expérimentations [23]. Ils ont confronté deux groupes d'étudiants à un questionnaire capable d'évaluer le niveau de compréhension d'un programme. Le premier groupe a mené une expérience en utilisant un compilateur Pascal habituel. Tandis que le deuxième groupe a mener l'expérience en utilisant **Pascal Mutants** et en explorant les concepts des tests de mutation. Les résultats de cette expérience révèlent que l'utilisation des concepts d'analyse des mutations contribue positivement au processus d'apprentissage. Ils ont également mené un enquête avec des étudiants seniors concernant leur avis par rapport à l'intégration du mutation testing dans des cours de programmations dès le début de l'apprentissage. La plupart d'entre eux étaient convaincus que ces pratiques pourraient être utiles pour complémenter leurs expériences. Selon les chercheurs, le résultat de cette enquête pourrait être menacé par le fait que ces étudiants seniors étaient peut-être déjà trop éloignés de leur première année pour réellement se souvenir de ce que c'était. Ce qui pourrait les avoir mené à mal interpréter celle-ci.

Clarke et al.[8] ont adopté l'approche d'intégrer un outil de software testing dans leur cours d'ingénierie logicielle. Ils ont utilisé **WResTT** (Web-Based Repository of Software Testing Tools) qui est un outil destiné au support des besoins pédagogiques des étudiants et des professeurs dans les cours de programmation et d'ingénierie logicielle. Il fournit un accès à un ensemble de tutoriels à propos de concepts de test et des outils de test. Ce système emploie l'apprentissage collaboratif et des fonctionnalités de réseaux sociaux. **WResTT** implémente également le concept de gamification. Les tutoriels utilisent une approche collaborative d'apprentissage ou les étudiants sont groupés par équipes et chacune est récompensée de points en fonction des tâches accomplies. Les auteurs soutiennent l'idée qu'intégrer un outil tel

que **WResTT** dans un cours d'ingénierie logicielle expose les étudiants aux outils de test et améliore leurs compétences pratiques en software testing.

2.5.4 Évaluation par les pairs

Un challenge dans l'éducation du software testing est de faire percevoir aux étudiants le bénéfice d'écrire des tests et d'évaluer leur qualité avec des techniques avancées. Le mutation testing et l'évaluation par les pairs sont des stratégies mises en évidence dans la littérature qui permettent d'augmenter la motivations des étudiants. Mutation testing les met en contact avec une version défectueuse du programme. Ils peuvent donc confronter leur suite de tests avec cette version. L'évaluation par les pairs aide les étudiants à identifier les problèmes dans leur code et leur permet également d'apprendre les solutions apportées par les autres au même problème. Delgado-Perez et al. [11] ont introduit ces techniques dans un cours de software testing durant trois années. Les résultats ont montré que les étudiants ont une meilleure compréhension de l'importance du critère de couverture en appliquant des techniques de testing avancées. L'expérience a également démontré que l'évaluation par les pairs aide les étudiants à calibrer la qualité de leur suite de tests. Ils ont été capable de percevoir les différences entre leur suite de tests et celle des autres et que la couverture de mutation corrélait avec les notes attribuées.

3 Conception

Le travail a consisté en la réalisation d'un plugin éducationnel dans le domaine du Mutation Testing en incorporant les informations générées par les plugins Descartes et Reneri dans un environnement de développement intégré (EDI) tel que Microsoft Visual Studio Code ou JetBrains IntelliJ. Les différentes étapes depuis la conception jusqu'à la mise en service vont être détaillées.

3.1 Définition des bases du travail

Cette section aborde les choix effectués concernant l'Environnement de développement intégré (EDI) et le langage de programmation.

3.1.1 Choix de l'EDI

Etant donné les contraintes posées par les plugins sur lesquelles l'extension est posée telles que le langage de programmation Java et l'outil Maven (qui permet la gestion et l'automatisation de la production des projets logiciels codés en Java), le choix de l'EDI s'est porté sur l'EDI Microsoft Visual Studio Code (VSCode). Cet EDI est open-source, cross-platforme et supporte plusieurs langages. Il supporte le Java et dispose d'une extension pour Maven. C'est un des outils les plus populaires aujourd'hui. Il est également rapide, léger, il fournit un framework permettant de créer des extensions. Il est également hautement personnalisable. Ce framework est supporté par un tutoriel détaillé des différentes possibilités visuelles offertes.

3.1.2 Choix du langage

Visual Studio Code fournit la possibilité de choisir entre le TypeScript et le Javascript pour développer une extension de l'EDI. Néanmoins Visual Studio Code est développé en TypeScript et ce langage est utilisé dans le guide de développement d'un plugin. Pour cette raison, le langage TypeScript fut choisi pour l'implémentation. Le TypeScript est un langage de programmation open-source développé par Microsoft qui est compilé vers du Javascript. Il est apparu en 2012 et est toujours en développement et continue de gagner en popularité.

3.2 Solution proposée

Dans ce point est décrite la solution mise en place, les exigences auxquelles elle répond, les cas d'utilisations, les divers outils et technologies utilisés. Les choix d'implémentation sont également détaillés dans cette section. Pour terminer, la possibilité d'une intégration avec un système existant est évoquée.

3.2.1 Exigences fonctionnelles

Pour atteindre ses objectifs, la solution mise en place doit satisfaire différentes exigences fonctionnelles. L'extension doit :

- Permettre d'exécuter les commandes relatives au plugins Descartes et Reneri.
- Guider les étudiants dans les différentes étapes du processus qui permet d'accéder aux informations.
- Afficher le score de mutation.
- Guider l'étudiant vers les méthodes pseudo ou partiellement testées.
- Permettre d'accéder à des informations concernant les méthodes pseudo ou partiellement testées.
- Permettre d'accéder à des informations concernant les tests qui exécutent des méthodes pseudo ou partiellement testées.
- Mettre en évidence les informations générées par les plugins Descartes et Reneri dans l'éditeur de code.

3.2.2 Exigences non-fonctionnelles

L'extension doit également répondre à un ensemble d'exigences non-fonctionnelles :

- Exigences d'utilisabilité : le plugin doit permettre de faciliter l'emploi des plugins Descartes et Reneri.
- Exigences visuelles : les représentations visuelles utilisées dans le plugin doivent indiquer de manière claire et concise l'information. Ces représentations ne doivent pas entraver la productivité de l'utilisateur.
- Exigences de maintenabilité : le plugin doit être conçu de manière à permettre l'ajout de fonctionnalités ou d'informations supplémentaires. Ce qui veut dire que le système est assez modulable et permet l'ajout ou la suppression d'éléments de manière simple. Le code doit être documenté afin de faciliter la compréhension du système pour quiconque souhaiterait reprendre le projet par la suite.

3.2.3 Maquette

L'objectif de ce projet est d'incorporer les informations des plugins Descartes et Reneri dans l'EDI afin d'alléger la charge cognitive de l'étudiant. Selon Liu et al. [20], les développeurs se reposent de plus en plus sur des outils ou

services externes. L'information se retrouve éparpillée à travers différents programmes. Pour y accéder le développeur doit passer d'une application à une autre. Par exemple, pour augmenter la couverture du code à partir des rapports générés par PITest, les utilisateurs doivent quitter leur EDI pour ouvrir un navigateur et se rendre vers la page générée. Deuxièmement, ils doivent chercher pour trouver les lignes non couvertes. Troisièmement, ils doivent retourner dans leur EDI, ouvrir le fichier source et développer des nouveaux tests pour ces lignes.

Le challenge est de trouver le bon mélange entre actionabilité et risque de surcharger le développeur d'informations. Quatres principes de présentations ont été suivis :

- Non intrusif : ne pas distraire le développeur. Focus sur le code source. Présentation visuelle minime de préférence. Devrait être caché jusqu'à activation par le contexte. Les couleurs utilisées ne doivent pas se superposer à celles déjà utilisées.
- Expression intuitive : La sémantique de représentation des informations externes doit être considérée.
- Extensibilité : l'intégration des informations dans le code source sera naturellement réduite par rapport à celles fournies par les outils. Il faut donc permettre au développeur d'accéder à toute cette information si il le souhaite.
- Unicité : Eviter la confusion avec d'autres informations.

Sur base de l'API (Interface de programmation) permettant la création de plugin fournie par l'équipe de développement de Visual Studio Code et en essayant de respecter un maximum les principes de représentations, une maquette a été établie afin d'avoir une idée visuelle des objectifs à atteindre.

Les figures 4 et 5 présentent les représentations implémentées dans l'extension. Les cercles bleus permettent uniquement de cibler les représentations et ne font pas partie de l'aspect visuel.

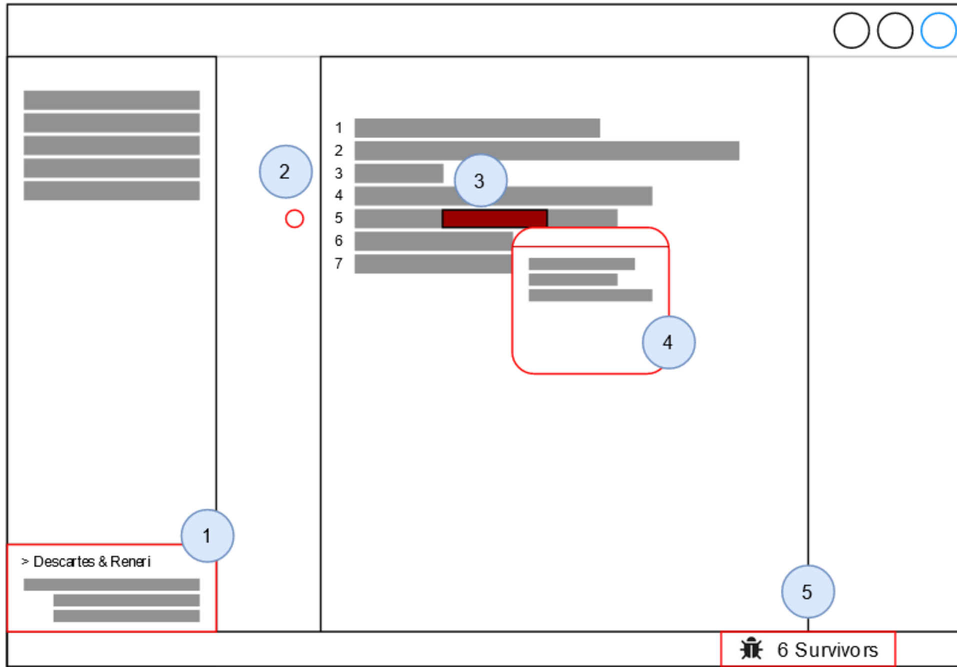


Figure 4: Maquette de l'extension

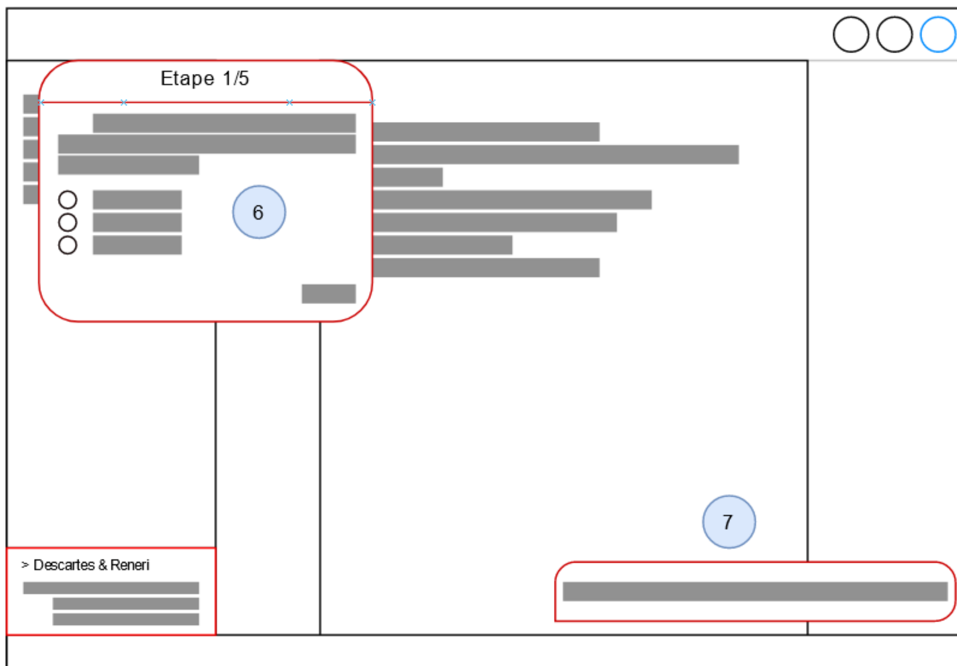


Figure 5: Maquette de l'extension

1. Cette fonctionnalité, située dans le coin inférieur gauche est une arborescence. Celle-ci permet d'actionner rapidement aux commandes des différents plugins (Descartes, Reneri, LASoT).

2. Le Gutter, placé à gauche de chaque ligne de code, peut accueillir un petit graphique, petit mot ou icône. Cette indicateur peut permettre d'indiquer sur quelle ligne de code une décoration a été ajoutée.

3. Cette représentation est une décoration du code source dans l'éditeur. C'est avec ces décorations que les méthodes signalées par Descartes vont être mises en évidence.

4. Cette représentation visuelle est une sur-couche. C'est un panneau qui apparaît quand l'étudiant passe le curseur par dessus une décoration. Il permet d'afficher une représentation plus complexe contenant plus d'informations. Cette spécificité permet au développeur de garder le contrôle pour afficher uniquement ces sur-couches lorsqu'il en a besoin.

5. La barre de statut. Cette outil permet d'afficher succinctement une information avec une icône et une donnée textuelle courte. Il permet également de capturer l'événement du clic sur cet objet.

6. Cette représentation est appelée QuickInput dans l'API de VSCode. Elle permet d'afficher un dialogue permettant de récupérer des entrées du développeur. Celle-ci peut également comporter plusieurs étapes.

7. La notification est une vue utilisée pour notifier les développeurs de différents événements. Des icônes peuvent être placées pour informer l'utilisateur qu'il y a des informations externes supplémentaires. Des boutons peuvent être ajoutés sur cette représentation afin d'effectuer des actions en rapport avec l'évènement. Afin d'éviter toute distraction, les notifications sont subtiles et non-intrusives. Il est tout de même possible que cela puisse augmenter la probabilité que les développeurs manquent une notification importante si cela aurait pu aider le développeur dans sa tâche [20].

3.2.4 Cas d'utilisation

Le plugin est développé dans le but d'être utilisé par des étudiants en master ayant des notions en software testing. Malheureusement ce type d'outil ne convient pas à des étudiants novices en programmation. Il pourrait par contre être tout de même utilisé par des développeurs aguerris souhaitant former leur équipe au testing. Il n'y aura donc que deux types d'acteur pour cet outil.

La figure 6 montre les différentes fonctionnalités offertes par l'extension. Il est nécessaire que les plugins Descartes et Reneri soient installés au préalable

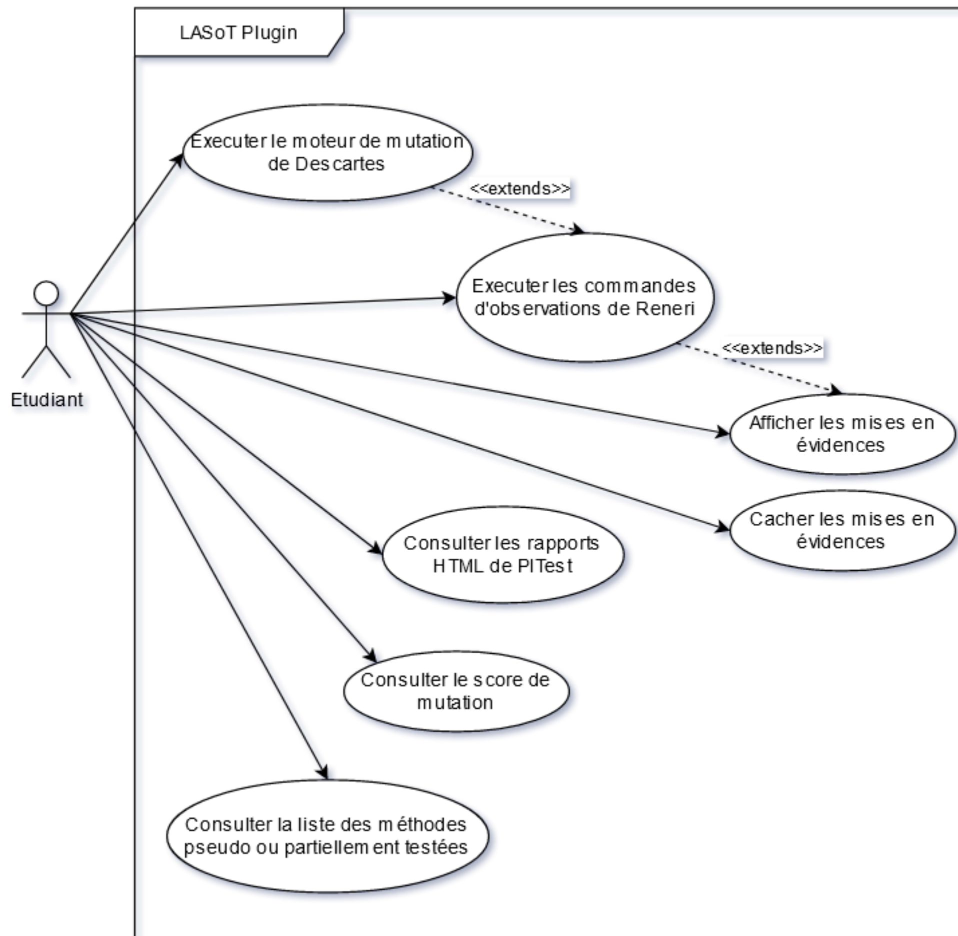


Figure 6: Diagramme de cas d'utilisation de l'extension

pour que ces cas d'utilisation puissent fonctionner. Certaines d'entre elles dépendent de l'exécution d'autres au préalable. Les commandes d'observation du plugin Reneri dépendent des rapports générés par Descartes pour fonctionner. L'incorporation des informations dans l'EDI quant à elle nécessite au minimum l'exécution du moteur de mutation de Descartes pour profiter des rapports générés.

3.2.5 BPMN

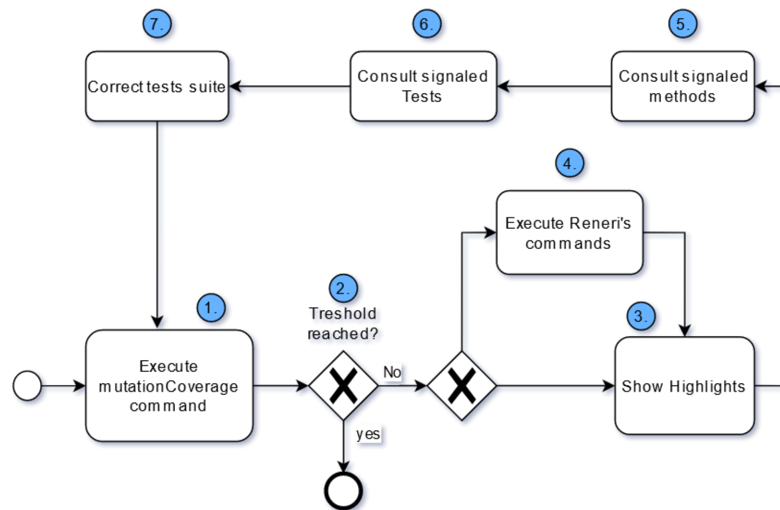


Figure 7: Diagramme BPMN de l'extension

La figure 7 présente le diagramme Business Process Model and Notation (BPMN) de l'extension. Il définit le schéma globale d'utilisation du plugin. La première étape consiste à l'exécution du moteur de mutation Descartes. La deuxième se résume à la détermination de l'atteinte des objectifs. Le score de mutation a-t-il atteint de seuil? Ce point correspond à l'étape 9 du processus du mutation testing basé sur le code [26] cité dans l'état de l'art. Si le score n'atteint pas l'objectif fixé on passe aux étapes 3 et 4. L'étape 4 n'est pas obligatoire pour consulter les rapports fournis par Descartes. Dans le cas où elle n'est pas exécutée, les informations générées par le plugin Reneri ne sont pas affichées. L'étape 3 active la surbrillance des méthodes pseudo testées et des tests qui exécutent ces méthodes. Les étapes de 5 à 8 concerne la lecture des informations incorporées dans l'éditeur de code et l'adaptation de la suite de tests. Finalement, on exécute à nouveau le moteur de mutation pour observer le résultat. Ce processus est renouvelé jusqu'à ce que le score de mutation ait atteint le seuil d'acceptation déterminé.

3.3 Implémentation

À travers ce point est décrite l'implémentation de la solution nommée *LA-SoT*, c'est à dire les choix techniques et les éléments mis en place pour assurer le bon fonctionnement de l'extension. Ces implémentations sont effectuées par rapport aux cas d'utilisation relevés à la figure 6.

3.3.1 Exécuter le moteur de mutation de Descartes

Cette fonctionnalité permet à l'étudiant d'exécuter en un clic la commande de PITest avec le moteur de mutation de Descartes. Cette commande va générer les mutants sur base des opérateurs de mutation et les confronter à la suite de tests du projet.

Afin de fournir un accès rapide à cette commande, une arborescence est ajoutée dans la barre latérale gauche de l'EDI comme indiqué sur le premier point de la figure 4. Cette arborescence est implémentée à l'aide de l'API "TreeView" de VSCode [10] configurée avec la localisation "explorer". Cette fonctionnalité rend l'utilisation du plugin Descartes plus aisée ce qui répond à la première exigence fonctionnelle. D'autre part, elle réduit la complexité de la tâche que doit accomplir l'étudiant concernant l'utilisation des outils. Elle évite au développeur de devoir aller rechercher cette commande dans les dépendances présentées par le plugin de Maven ou dans la documentation de Descartes.

Lorsque le processus est terminé, des rapports concernant les mutations et les méthodes pseudo ou partiellement testées sont générés. L'extension *LASoT* va ensuite lire ceux-ci contenu dans le dossier correspondant à la date d'exécution. Ces informations sont rassemblées dans deux fichiers json. Un fichier *methods.json* et *mutations.json*. Le premier reprend l'ensemble des méthodes pour lesquelles des mutations ont été créées.

```
1  {
2      "name": "updateScore",
3      "description": "(I)V",
4      "class": "GameController",
5      "package": "be/unamur/game2048/controllers",
6      "file-name": "GameController.java",
7      "line-number": 154,
8      "classification": "not-covered",
9      "detected": [],
10     "not-detected": [
11         "void"
12     ],
13     "tests": [],
14     "mutations": [
15         {
16             "status": "NO_COVERAGE",
17             "mutator": "void",
18             "tests-run": 0,
19             "tests": [],
20             "killing-tests": [],
```

[10] https://code.visualstudio.com/api/extension-guides/tree-view

```
21                 "succeeding-tests": []
22             }
23         ]
24 }
```

Listing 2: Exemple partiel du rapport methods.json fourni par Descartes concernant une méthode

Ce document fourni des informations propres aux méthodes telles que le nom, une description et des informations de localisation comme la classe, le "package" et le fichier dans lequel se trouve cette méthode et sa position dans ce fichier indiqué par le numéro de la ligne. La description indique le type de paramètre que prend cette méthode et son type de retour. Dans le cas de l'exemple donné 2, (I)V détermine que la fonction prend un entier en paramètre et contient un type de retour Void. Ce rapport est composé également de données relatives à l'analyse, le temps pris pour exécuter le processus et les mutateurs configurés.

Descartes fourni également des informations par rapport aux transformations générées. Il classifie cette méthode en fonction du statut de ces mutations. Une méthode est dite non couverte si elle n'est pas couverte par la suite de tests et donc pour laquelle aucune mutation n'est détectée. Elle est dite pseudo-testée si elle est couverte par la suite de tests, mais aucune mutation extrême appliquée à la méthode n'a été détectée par aucun cas de test. Une méthode est dite partiellement testée si elle a des résultats mitigés. Enfin, une méthode est classée comme testée si toutes les transformations extrêmes sont détectées par la suite de tests [21]. Finalement Descartes cite les tests qui exécutent la méthode et les mutations appliquées sur celle-ci.

Le second fichier, *mutations.json*, contient des informations du point de vue des transformations et les mutateurs utilisés lors du processus. Un exemple partiel d'un document est fourni au listing 3. Pour chaque mutation, il y a des informations sur ses méta-données (mutateur, fichier et ligne,..), son statut (couverte ou non, tuée ou non), la méthode sur laquelle elle a été appliquée et les tests qui ont exécutés cette mutation. Pour ces derniers, le champ "run" indique le nombre de tests qu'il a fallu exécuter par rapport à la liste "ordered" pour tuer cette mutation. Dans le cas ou la mutation a survécu, tous les tests sont exécutés. Ces derniers sont rangés en fonction de leur résultat par rapport à cette dernière. *Killing*, pour les tests qui ont tué la mutation et *succeeding*, les tests qui ont réussi malgré la transformation.

```
1 {
2     "detected": false,
3     "status": "NO_COVERAGE",
4     "mutator": "void",
```

```
5      "line": 164,
6      "block": 0,
7      "file": "GameController.java",
8      "index": 1,
9      "method": {
10         "name": "afterMove",
11         "description": "()V",
12         "class": "GameController",
13         "package": "be.unamur.game2048.controllers"
14     },
15     "tests": {
16         "run": 0,
17         "ordered": [],
18         "killing": [],
19         "succeeding": []
20     }
21 },
```

Listing 3: Exemple partiel du rapport mutations.json fourni par Descartes concernant une mutation

Afin de mieux visualiser les informations fournies par Descartes et de faciliter l'incorporation de celles-ci dans le plugin, un modèle conceptuel a été établi. La classe "DescartesReports" est composée d'un rapport de chaque type. Le rapport des mutations et le rapport des méthodes.

Ces classes ont été implémentées sur base de ce schéma et sont stockées dans le fichier "DescartesModels" contenu dans le dossier "models" du projet.

3.3.2 Exécuter les commandes d'observations de Reneri

- Précondition : Le plugin Reneri dépend des rapports sous format *Json* fournis par Descartes pour effectuer ses observations. Il est donc nécessaire au préalable d'exécuter la commande "mutationCoverage" de PITest avec le moteur de mutation extrêmes de Descartes. Il faut également paramétrer Maven ou Gradle afin de générer les rapports nécessaires. Les fichiers en question (methods.json et mutations.json) doivent être déplacés dans le dossier "target" du projet avant l'exécution des commandes de Reneri.

- Postcondition : Reneri génère des conseils sous forme de documents *Json* pour améliorer une suite de tests. Ceux-ci sont produits pour les méthodes et pour les tests. On peut donc retrouver deux dossiers (methods, tests) dans lesquels les rapports sont insérés.

Le schéma ci-dessous présente l'arborescence des rapports du plugin Reneri.

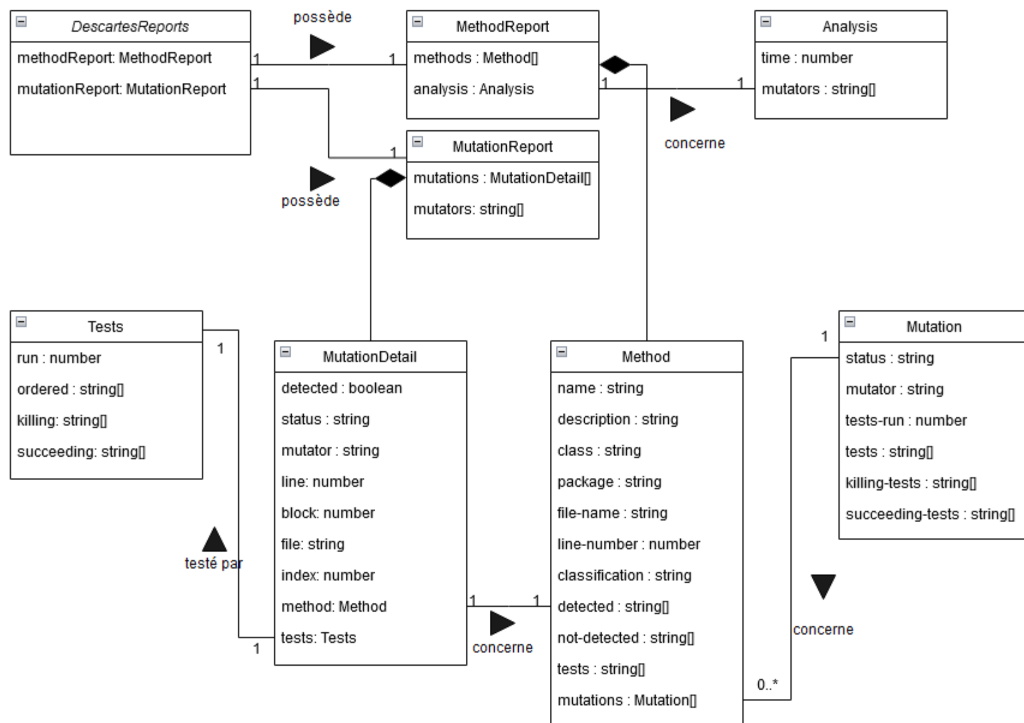
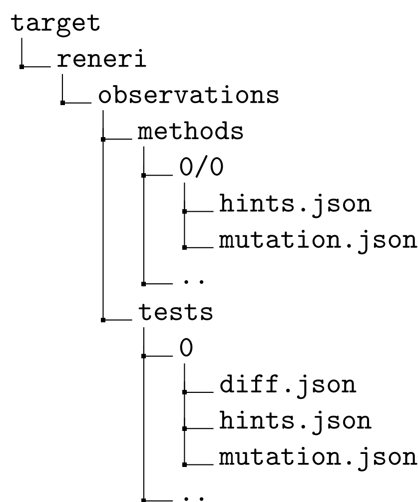


Figure 8: Diagramme de classe des rapports de Descartes



Dans le dossier des observations des méthodes, un répertoire est créé pour chaque méthode pour laquelle au moins, une mutation n'a pas été détectée. Chacun de ces répertoires peut contenir jusqu'à trois fichiers :

- mutation.json : est un fichier qui contient des informations concer-

nant la mutation comme le mutateur, des méta-données par rapport à la méthode sur laquelle cette mutation a été appliquée (méthode, description, package, class) et les tests qui exécutent cette méthode. Le listing 4 donne un exemple de document *mutation.json*.

```
1  {
2    "mutator": "true",
3    "class": "Tile",
4    "package": "be.unamur.game2048.models",
5    "method": "equals",
6    "description": "(Ljava/lang/Object;)Z",
7    "tests": [
8      "be.unamur.game2048.Test2048"
9    ]
10 }
```

Listing 4: Document mutation.json fourni par Reneri

- diff.json : est un document dans lequel sont données les valeurs retournées par la méthode pour le code original du programme et ses transformations. Ces différentes valeurs sont liées à un *pointcut*. Un *pointcut* est un ensemble de point de jonction qui déterminent ou la transformation a été effectuée. Le listing 5 affiche un exemple de ce document pour une méthode.

```
1  {
2      "pointcut": "be.unamur.game2048.controllers.
           GameController|moveUp|(Z)Z|0|#result",
3      "expected": [
4        {
5          "literalValue": "true",
6          "typeName": "boolean"
7        }
8      ],
9      "unexpected": [
10       {
11         "literalValue": "false",
12         "typeName": "boolean"
13       }
14     ]
15 }
```

Listing 5: Document mutation.json fourni par Reneri

- hints.json : contient des informations relatives au type de conseil et au points d'entrée. Les points d'entrées sont des méthodes. Ces in-

formations me semblent inexploitables dans le cadre du plugin de ce mémoire.

Le dossier des observations des tests contient des répertoires pour chaque mutation non détectée et pour laquelle un test exécute la méthode sur laquelle la transformation a été appliquée. Ces répertoires peuvent contenir les mêmes types de fichiers que ceux retrouvés dans les dossiers des observations des méthodes. Les fichiers sont structurés de la même manière sauf pour le document *hints.json*. Le listing 6 donne un exemple de la structure du fichier *hints.json*.

```
1  {
2      "pointcut": "be.unamur.game2048.Test2048|
           testMoveAvailableOnFullGridWithMergingPossibility()|
           25",
3      "hint-type": "observation",
4      "location": {
5          "point": "be.unamur.game2048.Test2048|
               testMoveAvailableOnFullGridWithMergingPossibility
               ()|25",
6          "type": "boolean",
7          "from": {
8              "line": 100,
9              "column": 27
10         },
11         "to": {
12             "line": 100,
13             "column": 49
14         },
15         "file": "C:\\Users\\vandeputs\\vscode-projects\\
               lasotExperiment\\2048\\src\\test\\java\\be\\
               unamur\\game2048\\Test2048.java"
16     }
17 }
```

Listing 6: Document hints.json fourni par Reneripour une observation des tests.

Ce document fournit des informations sur la localisation des valeurs observées par rapport au code original. Le répertoire des tests contient également un document *locations.json* dans lequel sont enregistrés un ensemble de *pointcut*. Chaque objet contenu dans ce fichier reprend un *pointcut* qui référence un test, le chemin d'accès du fichier ou se trouve le test et les coordonnées de l'appel de la méthode pour laquelle la mutation indiquée dans le fichier *mutation.json* n'a pas été détectée.

Afin de pouvoir manipuler les données des rapports fournis par Reneri, un modèle logique des données a été établi 9. Celui-ci permet également

d'avoir une meilleure visualisation des informations fournies et leurs relations entre elles. La classe "Observation" est composée d'un ensemble de méthodes signalées. Chaque méthode signalée contient les différents fichiers cités précédemment.

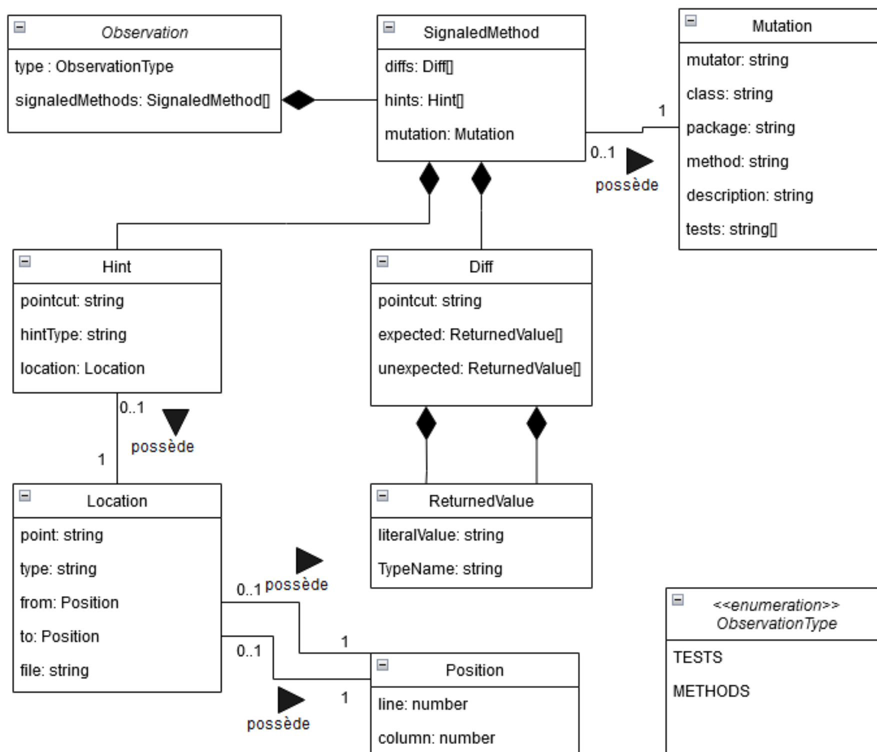


Figure 9: Diagramme conceptuel des données générées par Reneri

Ces classes ont été implémentées sur base de ce schéma et sont stockées dans le fichier "ReneriModels" contenu dans le dossier "models" du projet.

3.3.3 Afficher les décorations

- Précondition : L'affichage des décorations nécessite au préalable l'exécution au minimum de la commande de Descartes. En effet, le plugin requiert la sérialisation des données générées par Descartes et tout au plus Reneri.

- Postcondition : Affiche les décorations dans le code source du projet ainsi que sur les méthodes des tests. Dans le cas ou seul Descartes a été exécuté, seul les signatures des méthodes pseudo ou partiellement testées sont mises en surbrillance. Si les trois commandes de Reneri sont exécutées, des décorations supplémentaires sont affichées au sein des méthodes. Chaque décoration est générée avec un message

comportant des informations détaillées disponibles lors du survol de celle-ci par le curseur.

Pour pouvoir interagir avec l'utilisateur la fonctionnalité Command[11] fournie par l'API de VSCode est utilisée. Celle-ci permet d'exposer des fonctionnalités et déclencher des actions implémentées par de la logique interne. Cette commande est ajoutée au fichier *package.json* de l'extension. Celle-ci est également liée à l'*explorer* avec un bouton sur la ligne correspondant à la mise en surbrillance des informations.

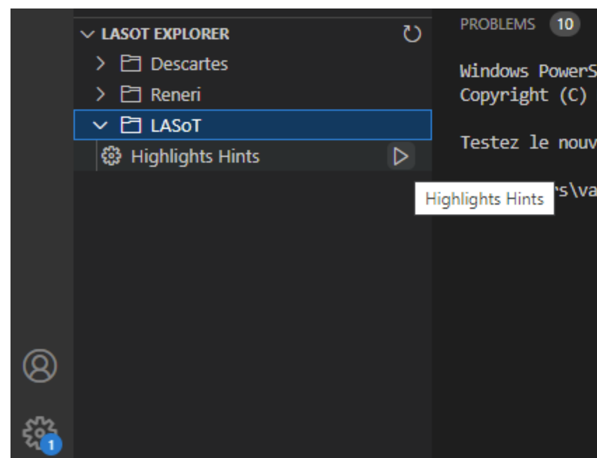


Figure 10: Bouton d'activation des surbrillances.

Afin d'être sûr d'afficher des données à jour par rapport aux dernières informations générées par Descartes et Reneri, une vérification est effectuée avant de lire celles-ci. Ce contrôle est réalisé à l'aide d'une date enregistrée lors de la sérialisation des données. Celle-ci est alors comparée avec le dernier dossier généré par Descartes.

Les informations affichées en sur-couche doivent aller droit au but, être concises et suffisamment compréhensibles pour un étudiant en master afin de le mener à l'étape suivante. Cette sur-couche doit donc indiquer le ou les tests qui exécutent cette méthode pour laquelle une mutation n'a pas été détectée. L'opérateur de mutation est également une information intéressante. Elle permet d'informer l'étudiant de la valeur retournée par la méthode transformée. Une petite explication de la transformation appliquée peut être utile et évite le développeur de devoir aller consulter dans la documentation et de quitter l'EDI.

Ces représentations sont implémentées à l'aide de l'API de VSCode et de

[11] https://code.visualstudio.com/api/extension-guides/command

la liste d'exemples fournie sur GitHub [12]. Deux évènements sont capturés pour appliquer les décorations à l'éditeur.

- vscode.window.onDidChangeActiveTextEditor : Est un événement qui est déclenché quand l'éditeur actif a changé. Par exemple, lorsque le développeur passe d'un onglet à un autre.
- vscode.workspace.onDidChangeTextDocument : Est un événement qui est émis lorsqu'un document est changé. Il arrive régulièrement lorsque le contenu d'un document change.

Le déclenchement de ces événements appellent des méthodes qui vont mettre à jour les décorations dans l'éditeur. Ces méthodes sont encapsulées dans une Classe "Decorator.ts" située dans le répertoire "decorator" de l'extension. Cette Classe expose les méthodes "activate()" et "triggerUpdateDecorations()". Cette dernière permet la mise à jour des décorations si l'objet a été activé au préalable. La figure 11 montre une décoration incorporée sur la signature d'une méthode. Celle-ci indique la classification de la méthode, le mutateur appliqué à la transformation et les tests qui exécutent cette méthode.

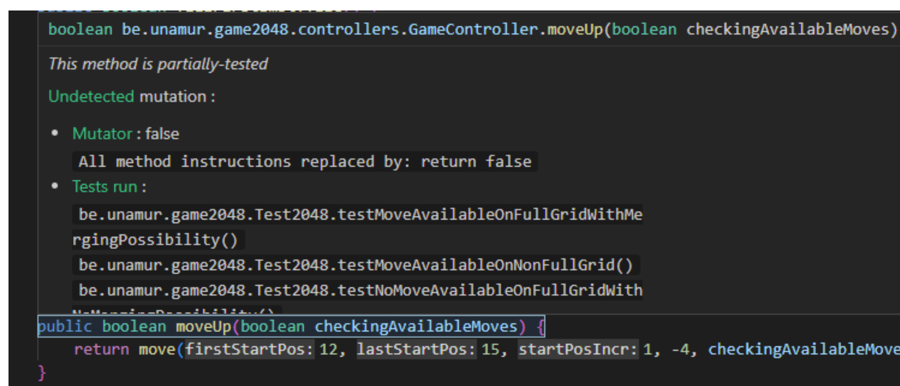


Figure 11: Décoration générée sur base des conseils de Descartes pour une méthode.

Descartes indique également les méthodes non couvertes. Ce que ne fait pas Reneri. La figure 12. Ce message comprend la classification de la méthode et le mutator appliqué à la transformation.

Les figures 13 et 14 montrent des messages affichés sur les tests sur base des rapports de Descartes et Reneri respectivement. Pour ce dernier on peut remarquer que le code pointé par Reneri est directement situé dans

[12]https://github.com/microsoft/vscode-extension-samples/blob/main/decorator-sample/USAGE.md

```
157    void be.unamur.game2048.controllers.GameController.afterMove()
158    This method is not-covered
159
160    Not covered mutation :
161
162    • Mutator : void
163    private void afterMove() {
164        if (highestScore < GameParams.scoreTarget) {
165            clearMerged();
166            fillFirstEmptyTile();
167            if (!movesAvailable()) {
168                gamestate = GameState.over;
169            }
170        } else if (highestScore == GameParams.scoreTarget)
171            gamestate = GameState.won;
172    }
```

Figure 12: Décoration générée sur base des conseils de Descartes pour une méthode non couverte.

l'assertion. Ce qui indique plus précisément à l'étudiant ou la modification doit être appliquée.

```
68     @Test
69     public void testNoMoveAvailableOnFullGridWithNoMergingPossibility(){
70     Undetected Mutation :
71                                               ue: 4),new Tile(value: 8),new
72     • Mutator : false                        lue: 64),new Tile(value: 128),
73     • On method : GameController.moveDown()   ue: 4),new Tile(value: 8),new
74                                               lue: 64),new Tile(value: 128),
75     Undetected Mutation :
76     • Mutator : false
77     • On method : GameController.moveLeft()
78
79     Undetected Mutation :
80
81     • Mutator : false                         ableMoves: true)
82     • On method : GameController.moveRight()  ailableMoves: true)
83                     || controller.moveDown(checkingAvailableMoves: true)
84                     || controller.moveLeft(checkingAvailableMoves: true));
85     }
```

Figure 13: Décoration générée sur base des conseils de Descartes pour un test.

Il arrive que des méthodes signalées par Reneri ne contiennent pas de fichier "hints.json", ce qui signifie qu'aucun conseil au sein d'un test n'a pu être généré. Ce qui ne veut pas dire qu'aucune action ne doit être prise par le développeur. Une mutation non détectée est tout de même pointée. Dans ce cas, le message appliqué à la décoration en sur-couche sera composé d'informations retirées des rapports de Descartes.

L'un des problèmes du mutation testing est exposé à partir de ce point. Les mutations équivalentes citées dans l'état de l'art font partie des décorations. L'extrême mutation testing permet de réduire le nombre de ce type de transformations mais ne résout pas le problème totalement [32]. Elles ne doivent

Figure 14: Décoration générée sur base des conseils de Reneri pour un test.

pas mener à une action particulière du développeur. Or, dans un premier temps, ce dernier sera mené à la réflexion et à la compréhension du code pour finalement se rendre compte qu'il est impossible d'éliminer cette mutation. Certaines mutations resteront donc vivantes.

3.3.4 Cacher les décorations

- Précondition : Les décorations doivent être au préalable activées.
- Postcondition : Masque l'affichage des surbrillances du code source au niveau des méthodes et des tests.

Une instanciation de type "Commande" supplémentaire ayant comme définition "Hide Highlights" est enregistrée dans le fichier *package.json*. Dans ce document, la commande est liée à un élément spécifique de l'*explorer*. Cette définition est représentée par le listing 7. La propriété **when** détermine sur quelle représentation et sur quel élément précisément cette commande va être liée. Un événement est également écouté lors du clic sur l'icône correspondant à cette commande. L'action exécutée lors de cet événement désactive les décorations à l'aide de la méthode `deactivate()` de l'objet `Decorator`.

```
1 {
2   "command": "lasot.DeactivateHighlights",
3   "when": "view == lasotExplorer && viewItem == lasot",
4   "group": "inline"
5 }
```

Listing 7: Liaison d'une commande à un élément de la vue Explorer

Cette fonctionnalité a été développée afin de répondre aux exigences fonctionnelles et de respecter la caractère non intrusif du plugin. Ces représentations

ne doivent pas entraver la productivité de l'utilisateur. Si c'est le cas, ce dernier peut à tout moment les retirer en un clic.

3.3.5 Consulter les rapports HTML de PITest

- Précondition : La commande de PITest "mutationCoverage" doit être exécutée au préalable. La dépendance "PITest" du projet doit être configurée de manière à utiliser le moteur de mutations extrêmes de Descartes.
- Postcondition : Affichage des rapports sous format HTML de l'extension PITest.

Afin de consulter les rapports sous format HTML fournis par PITest, le développeur doit quitter l'EDI pour afficher le document à partir d'un Navigateur. L'API de VSCode fournit une fonctionnalité WebView[13] permettant d'afficher directement du contenu HTML dans une vue intégrée dans l'EDI. Cette implémentation pourrait permettre de réduire le nombre de tâches à accomplir par l'étudiant pour consulter les résultats fournis par PITest et lui éviter de quitter l'EDI. Cette fonctionnalité n'a pas encore été implémentée et fait partie des travaux futurs.

3.3.6 Consulter le score de mutation

- Précondition : La génération du score de mutation nécessite l'exécution de la commande de PITest "mutationCoverage". La dépendance "PITest" du projet doit être configurée de manière à utiliser le moteur de mutations extrêmes de Descartes.
- Postcondition : Affichage du score de mutation incorporé dans l'EDI.

Le score de mutation doit pouvoir être directement accessible à la vue de l'étudiant, sans devoir effectuer aucune action supplémentaire mise à part les préconditions citées ci-dessus. L'outil utilisé est la barre de statuts. L'API de VSCode fournit un objet *StatusBarItem*[14] qui permet facilement d'implémenter cette fonctionnalité. Cette représentation est utile pour afficher une progression. Dans ce cas-ci, c'est la progression du score de mutation qui sera affiché. Il est également possible de configurer la couleur de l'arrière-plan à cet élément (Rouge ou orange). Cette fonctionnalité ne sera pas utilisée afin de ne pas rendre cet élément trop intrusif et qu'il vienne perturber la concentration du développeur [20]. De plus, cette élément serait la plupart du temps rouge ou orange car il est rare d'atteindre un score de mutation parfait dû au problème des mutations équivalentes [32].

[13] https://code.visualstudio.com/api/ux-guidelines/webviews
[14] https://code.visualstudio.com/api/ux-guidelines/status-bar

```
// --- Status bar

// Create
statusBarItem = vscode.window.createStatusBarItem(vscode.StatusBarAlignment.Right, 100);
context.subscriptions.push(statusBarItem);
```

Figure 15: Implémentation de la barre de status.

La figure 15 présente le code source permettant d'ajouter un élément sur la barre d'outils. La méthode `createStatusBarItem()` permet facilement l'ajout d'un élément dans la barre de statuts. Cette méthode prend un paramètre d'alignement, soit à gauche ou à droite de la barre de statuts et un paramètre de priorité qui détermine son positionnement par rapport aux autres éléments situés à côté. Plus l'indice de priorité est élevé, plus l'élément est déplacé sur la gauche. Ensuite, l'élément de la barre de statuts est ajouté au contexte. Afin de calculer le score de mutation, une méthode `getMutationScore()` a été ajoutée à la classe `DescartesState`. Cette méthode fait le rapport entre les mutations qui ont survécu et le nombre total des mutations. Finalement, cette fonction est appelée dans une méthode `updateStatusBarItem()`. Cette dernière va mettre à jour le score de mutation et l'afficher dans la barre de statuts. Cette méthode est appelée lorsque lors de mise en surbrillance des méthodes et des tests.

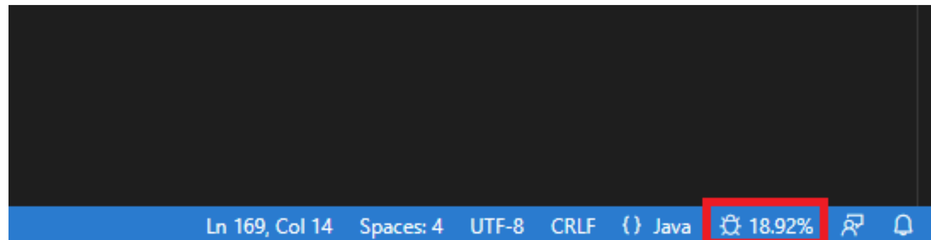


Figure 16: Barre de statut affichant le score de mutation.

3.3.7 Consulter la liste des méthodes pseudo ou partiellement testées

- Précondition : L'affichage des méthodes pseudo ou partiellement testées, nécessite l'exécution de la commande de PITest "mutationCoverage". La dépendance "PITest" du projet doit être configurée de manière à utiliser le moteur de mutations extrêmes de Descartes.
- Postcondition : Affichage des méthodes pseudo ou partiellement testées intégré à l'EDI.

Lorsque la commande *mutationCoverage* de PITest Descartes ou celles de Reneri ont été exécutées, l'étudiant doit pouvoir accéder facilement aux

méthodes pour lesquelles des mutations n'ont pas été détectées. Une fois cette information recueillie, l'étudiant sait ou il doit se diriger pour consulter de plus amples informations.

La Classe `StatusBarItem`[15] de l'API VSCode contient une propriété qui lui permet d'être lié à une *Commande*. Il est alors possible de récupérer un événement lors d'un clic sur cette objet. Sur base de cet événement, il est possible d'afficher des informations supplémentaires via une notification. Dans Visual Studio Code, les notifications[16] peuvent prendre plusieurs formes. Elles peuvent donner une information succincte pour avertir l'utilisateur d'un problème et récupérer un action de celui-ci avec un bouton. Elles peuvent être configurées pour fournir une information sur la progression d'un processus. Ces notifications peuvent également prendre la forme d'un dialogue affiché au centre de l'écran. Cette dernière configuration permet l'affichage d'une plus grande quantité d'informations. La figure 17 présente l'affichage de ce dialogue.

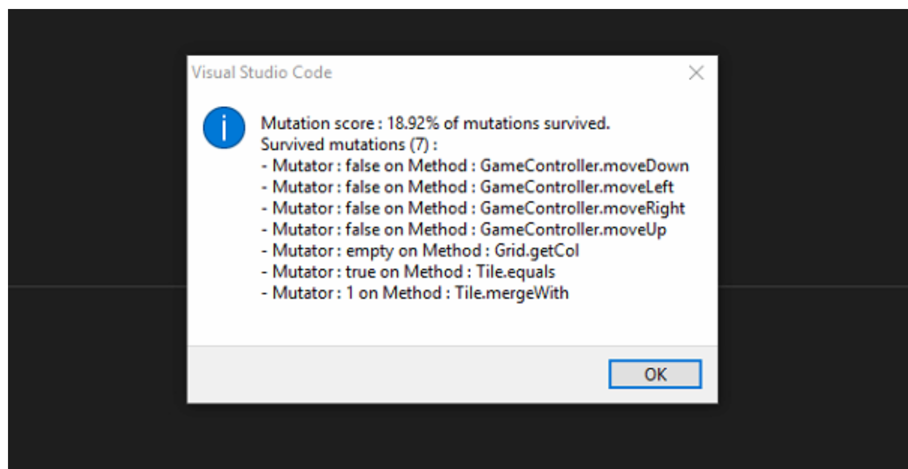


Figure 17: Dialogue affichant les mutations survivantes.

3.3.8 Wizard

Les étapes à suivre pour afficher les données dans l'EDI peuvent sembler lourdes et floues. De plus, certaines d'entre elles peuvent prendre un certain temps en fonction de la taille du projet. Actuellement l'extension n'apporte aucune indication de l'ordre dans lequel les étapes doivent être suivies et ce qu'elles exécutent réellement. Pour pallier à ce manquement et guider l'étudiant dans le processus, un Wizard a été implémenté. La fonctionnalité

[15] https://code.visualstudio.com/api/references/vscode-api#StatusBarItem
[16] https://code.visualstudio.com/api/ux-guidelines/notifications

QuickPick[17] de l'API VSCode permet l'affichage d'un dialogue permettant d'effectuer des actions et récupérer des entrées de l'utilisateur. De plus, celle-ci peut comporter plusieurs étapes (*MultiStepInput*). Cet implémentation a été appelée *Wizard* pour sa ressemblance à un guide d'installation d'un programme. Comme pour ce dernier, il suffit de lire et suivre les étapes pour arriver au but final. Ce *Wizard* est appelé via la palette de commandes de Visual Studio Code. La figure 18 présente la première étape de cet outil.

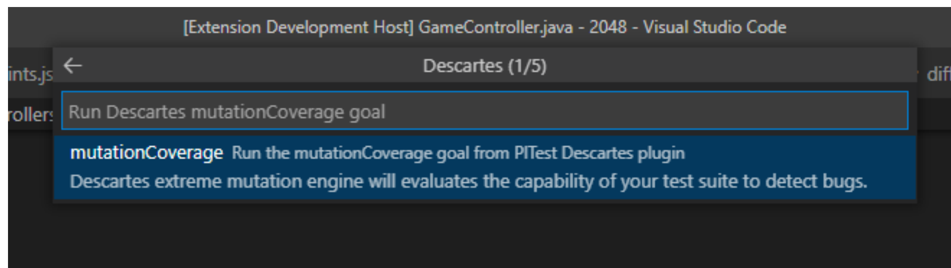


Figure 18: Première étape du Wizard.

[17] https://code.visualstudio.com/api/ux-guidelines/quick-picks

4 Discussion

Quelques points spécifiques au projet doivent être abordés et de même quelques limites doivent être annoncées afin d'entrevoir d'autres pistes de réflexion pour le futur.

4.1 Tests d'intégration

Un scénario de test a été développé dans le but d'assurer la présence et l'activation des extensions *Java Extension Pack* et *Maven*. Visual Studio Code utilise **Mocha**[18] pour exécuter une suite de tests. Mocha est un framework de test JavaScript fonctionnant sur *Node.js*. Le fichier de test *extension.test.ts* est initié à l'aide du mot-clé **suite**, et ensuite par des "tests" commençant par le mot-clé **test**. Chaque test est instancié par un nom et une méthode qui va effectuer le test. De cette manière, les noms de tests sont lisibles, par exemple, le premier test se nomme *Java Extension should be present*. Ce test vérifie que l'extension Java est présente. Le second test vérifie que cette extension est bien activée.

Le dernier test appelé **Can list LASoT Explorer Menus** vérifie que la représentation **LASoT Explorer** est bien initialisée avec ses trois menus.

Ces tests peuvent permettre de vérifier la régression lorsque la commande **npm run test** est lancée. Le chemin idéal serait d'inclure cette action dans un job lorsque des *commit* effectués sur une branche. Ce processus permettrait de vérifier que le code n'ajoute pas de régressions aux fonctionnalités existantes.

Le listing 8 présente une partie du code source des tests de l'extension expliqués ci-dessus.

```
1  suite('Extension Test Suite', () => {
2
3    test("Maven Extension should be present", () => {
4          assert.ok(vscode.extensions.getExtension("vscjava.vscode
              -maven"));
5     });
6
7     test("Maven should be activated", async function() {
8          await vscode.extensions.getExtension("vscjava.vscode-
              maven")!.activate();
9          assert.ok(true);
10    });
11
12    test("Can list LASoT Explorer Menus", async () => {
```

[18]https://mochajs.org/

```
13        const lasotExplorerProvider = new LASoTExplorerProvider
            ();
14        const roots = await lasotExplorerProvider.getChildren();
15        assert.equal(roots?.length, 3, "Number of root node
            should be 3");
16
17        let menuNode = roots![0] as Menu;
18        assert.equal(menuNode.label, "PITest Descartes", "First
            menu label should be \"PITest Descartes\"");
19        menuNode = roots![1] as Menu;
20        assert.equal(menuNode.label, "Reneri", "Second menu
            label should be \"Reneri\"");
21        menuNode = roots![2] as Menu;
22        assert.equal(menuNode.label, "LASoT", "Third menu label
            should be \"LASoT\"");
23    });
24
25 });
```

Listing 8: Code source des tests de l'Extension

4.2 Maintenance et pérennité

Cette extension pourrait être amenée à évoluer avec le temps. Il est donc très important de garder une trace des décisions prises dans le cadre de l'implémentation de la solution. Un dépôt sur GitHub[19] est mis en place à cet effet. La partie implémentation de ce mémoire peut faire l'objet d'une documentation. Le code source est commenté afin de faciliter la compréhension. L'idée est d'avoir une documentation expliquant les grandes étapes et les points sensibles du projet tandis que les commentaires sont là pour comprendre plus en profondeur le code en lui-même.

[19]https://github.com/SamuelVandePut/LASoT

5 Validation

Dans ce chapitre, les tests utilisateurs sont réalisés par des étudiants en master, des professeurs et assistants. Ces tests consistent à la mise en oeuvre d'un SUT (Software Under Test)[20] qui a été mis à disposition sur GitHub avec un ensemble de consignes à suivre. Le SUT est un projet Java qui est une version adaptée du célèbre jeu 2048 [21]. Ce projet est le même que celui qui a été implémenté dans le mémoire Mutation Testing Education[22] (MuTEd). Dans la version utilisée dans ce mémoire, une suite de tests a été implémentée de manière à ne pas détecter certaines transformations. Comme précisé dans l'état de l'art, il a été constaté que les étudiants ne trouvaient pas pertinent le fait d'effectuer des tests de logiciel sur des projets de petite taille [11]. Afin de répondre à ce critère, le SUT est un projet de complexité basse à modérée. Le but est de faire apparaître dans l'EDI, les informations fournies par les outils Descartes et Reneri et d'évaluer si celles-ci vont permettre à l'étudiant d'améliorer la suite de tests en éliminant les "survivants". Pour l'expérience, une machine configurée et accessible en bureau à distance a été mise en place afin de faciliter l'étape d'installation des extensions, du SUT et de ses dépendances.

5.1 Methodologie

Afin de récupérer une évaluation de qualité, les participants sélectionnés pour l'expérience doivent convenir à un profil défini. Ceux-ci doivent de préférence être des étudiants en master et avoir de bonnes notions en développement avec le langage Java et en software testing.

Avant de commencer, le participant reçoit par message une explication du concept du mémoire et de l'expérience qui va être menée. Dans ce message, des liens vers les dépôts de l'expérience[20] et du plugin *LASoT*[23] sont fournis. Il lui est demandé de les consulter.

Le dépôt de l'expérience contient une introduction théorique sur le mutation testing et une explication générale des plugins Descartes et Reneri. Ensuite, Il lui est demandé d'ouvrir le projet "2048" avec Visual Studio Code. Le plugin Descartes est configuré de manière à ne générer qu'une partie des classes du projet. Le participant ne doit donc pas tenir compte des autres classes pour améliorer la suite de tests. Il doit ensuite exécuter, dans l'ordre, les différentes étapes pour pouvoir consulter les informations générées par les plugins Descartes et Reneri. L'étape suivante consiste à

[20] https://github.com/SamuelVandePut/LASoT-Experiment
[21] https://play2048.co/
[22] https://zenodo.org/record/6645253#.YvykjxxByUk
[23] https://github.com/SamuelVandePut/LASoT

consulter le score de mutation et afficher la liste des méthodes signalées. Une fois les méthodes connues, il peut atteindre une de celles-ci pour consulter les informations incorporées dans le code. Ces données lui permet de connaître les tests pour lesquels il faut apporter des corrections. Lorsque les tests sont adaptés, l'utilisateur peut relancer le processus complet des plugins Descartes et Reneri et consulter à nouveau le score de mutation. Ce cycle est répété jusqu'à ce que le score de mutation ait atteint une valeur de 98%. Une mutation équivalente rend l'atteinte du score parfait impossible. Durant tout le déroulement de l'expérience, le participant est observé et il lui est possible de poser des questions concernant l'utilisation ou pour avoir des informations complémentaire à propos des différents concepts du mutation testing.

Une fois l'expérience terminée, le participant est demandé de remplir un User Experience Questionnaire (UEQ)[2] standard afin d'obtenir une impression globale de l'expérience de l'utilisateur. Ce questionnaire évalue le produit sous différents aspects de qualité. L'aspect de qualité pragmatique (Perspicuité, Efficacité, Fiabilité) et de qualité hédonique (Stimulation, Originalité). Le côté pragmatique décrit les aspects liés à la tâche et le côté hédonique les aspects non liés à la tâche.

Plus précisément, le questionnaire permet de juger un projet sur six aspects [30].

- Attractivité : Impression générale du produit. Les utilisateurs aiment-ils ou n'aiment-ils pas le produit?
- Compréhensibilité : Est-il facile de se familiariser avec le produit? Est-il facile d'apprendre à utiliser le produit? Est-il facile à comprendre et clair?
- Efficacité : Les utilisateurs peuvent-ils résoudre leurs tâches sans efforts inutiles? L'interaction est-elle efficace et rapide?
- Contrôlabilité : L'utilisateur a-t-il le sentiment de maîtriser l'interaction? Peut-il prédire le comportement du système? Se sent-il en confiance en utilisant le produit?
- Stimulation : L'utilisation du produit est-elle excitante et motivante?
- Originalité : Le produit est-il innovant et créatif? Est-ce qu'il capture l'attention des utilisateurs?

Pour l'attractivité, il y a six questions, tandis que toutes les autres questions en comportent quatre. La figure 19 montre la structure du questionnaire. Un **Google Form** a été réalisé pour l'expérience. Ce dernier

[2]https://www.ueq-online.org

comporte les 26 questions du *UEQ* et également des questions plus ouvertes permettant de recueillir des informations plus détaillées sur leur ressenti.

Une fois l'expérience menée et les résultats obtenus, il est possible d'interpréter les données récoltées à travers des graphes mis à disposition par les créateurs de cette méthodologie. La section 5.2 détaille l'analyse de ceux-ci.

	1	2	3	4	5	6	7		
Agaçant	○	○	○	○	○	○	○	Agréable	1
Incompréhensible	○	○	○	○	○	○	○	Compréhensible	2
Moderne	○	○	○	○	○	○	○	Sans fantaisie	3
Appropriation simple	○	○	○	○	○	○	○	Appropriation compliquée	4
Apporte de la valeur	○	○	○	○	○	○	○	Peu de valeur ajoutée	5
Ennuyeux	○	○	○	○	○	○	○	Captivant	6
Inintéressant	○	○	○	○	○	○	○	Intéressant	7
Imprévisible	○	○	○	○	○	○	○	Prévisible	8
Rapide	○	○	○	○	○	○	○	Lent	9
Original	○	○	○	○	○	○	○	Conventionnel	10
Rigide	○	○	○	○	○	○	○	Facilitant	11
Bien	○	○	○	○	○	○	○	Médiocre	12
Compliqué	○	○	○	○	○	○	○	Simple	13
Repoussant	○	○	○	○	○	○	○	Attractif	14
Habituel	○	○	○	○	○	○	○	Avant-gardiste	15
Désagréable	○	○	○	○	○	○	○	Agréable	16
Sécurisant	○	○	○	○	○	○	○	Insécurisant	17
Stimulant	○	○	○	○	○	○	○	Soporifique	18
Répond aux attentes	○	○	○	○	○	○	○	Ne répond pas aux attentes	19
Inefficace	○	○	○	○	○	○	○	Efficace	20
Clair	○	○	○	○	○	○	○	Déroutant	21
Non pragmatique	○	○	○	○	○	○	○	Pragmatique	22
Sobre	○	○	○	○	○	○	○	Surchargé	23
Attrayant	○	○	○	○	○	○	○	Rébarbatif	24
Sympathique	○	○	○	○	○	○	○	Inamical	25
Conservateur	○	○	○	○	○	○	○	Innovant	26

Figure 19: Questionnaire d'expérience utilisateur (UEQ).

5.2 Résultats

L'outil d'analyse des données fournit par l'équipe UEQ, permet de dégager certaines tendances et points d'attention. Il est important de signaler que les résultats obtenus auprès des étudiants et professeurs sont basés sur un échantillon de six personnes. Ces données pourraient être plus justes avec un plus grand groupe d'utilisateurs.

La figure 20 présente le graphe des résultats générés par l'outil d'analyse des réponses du questionnaire. Les valeurs comprises entre -0.8 et 0.8

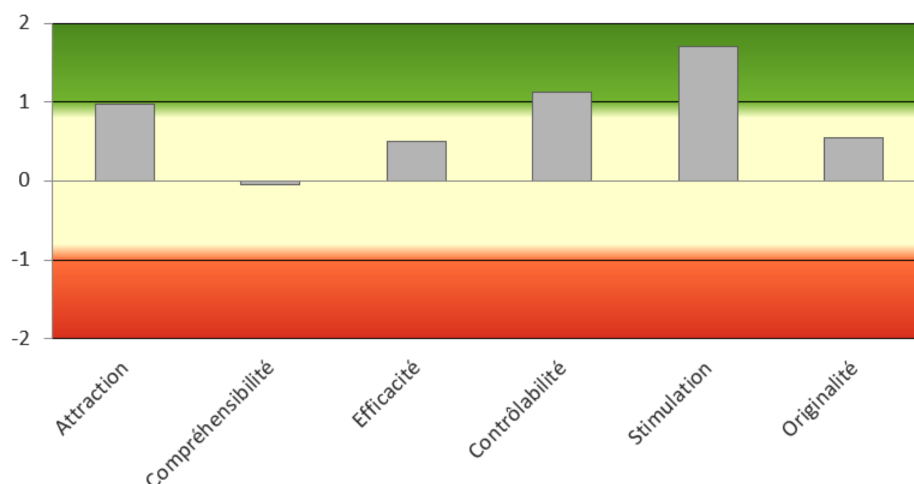


Figure 20: Résultats du questionnaire.

représentent une évaluation plus ou moins moyenne. Les valeurs supérieurs à 0.8 sont des évaluations positives tandis que les valeurs inférieurs à -0.8 représentent une évaluation négative. Ce graphe démontre que le plugin **LASoT** obtient une bonne note concernant l'aspect qualité de stimulation. C'est à dire que l'outil est intéressant et qu'il apporte de la valeur. Ce résultat correspond au ressenti observé. Une fois un test amélioré, l'étudiant comprend qu'un bug est détecté et constate une amélioration du score de mutation. Cette satisfaction est en réalité double car en plus d'avoir résolu un bug, l'étudiant a une meilleure compréhension du fonctionnement du projet. Cette expérience le motive à atteindre le score maximum et à éliminer tous les mutants. Du point de vue de la contrôlabilité, la valeur est positive. Autrement dit, les étudiants se sont senti confiant et ont ressenti un sentiment sécurisant. Ces sentiments peuvent être expliqués par l'apport du Wizard et du constat de progression du score de mutation. Ce graphe montre également que l'extension manque de compréhensibilité. Ce qui signifie que les participants ont eu des difficultés à se familiariser avec le produit et qu'il n'était pas assez clair. Durant l'expérience, une assistance a du être apportée aux participants du à ce manquement. Ceci s'explique naturellement par le fait que les utilisateurs ne connaissaient pas le mutation testing avant l'expérience mais ça ne veut pas dire non plus que le plugin ne pourrait pas être amélioré au point de vue clarté. L'efficacité obtient une note mitigée. Ce qui signifie que les utilisateurs ont eu des difficultés modérées à résoudre les tâches et qu'elles leur ont pris un certain temps. L'exécution des commandes du plugin Reneri nécessite un peu de temps ce qui peut expliquer ce résultat. L'originalité obtient également une note neutre. Les participants ont un avis neutre concernant le caractère innovant de l'outil. Finalement, l'attractivité obtient une note positive. En d'autres termes, les étudiants

ont un avis général positif. Ce graphe permet de mettre en évidence les points importants de cette expérience. L'outil semble apporter une valeur ajoutée et un sentiment de sécurité. Par contre, il n'est pas suffisamment compréhensible.

L'outil d'analyse offre un benchmark qui contient pour le moment les données d'évaluation de 452 produits effectuées avec l'UEQ avec un total de 20190 participants. Cette base de données est mise à jour, une fois par an et est disponible sur le site officiel[24]. Ce *benchmark* se présente sous forme d'un graphe. La figure 21 présente le résultat généré sur base des données recueillies pour le plugin *LASoT*.

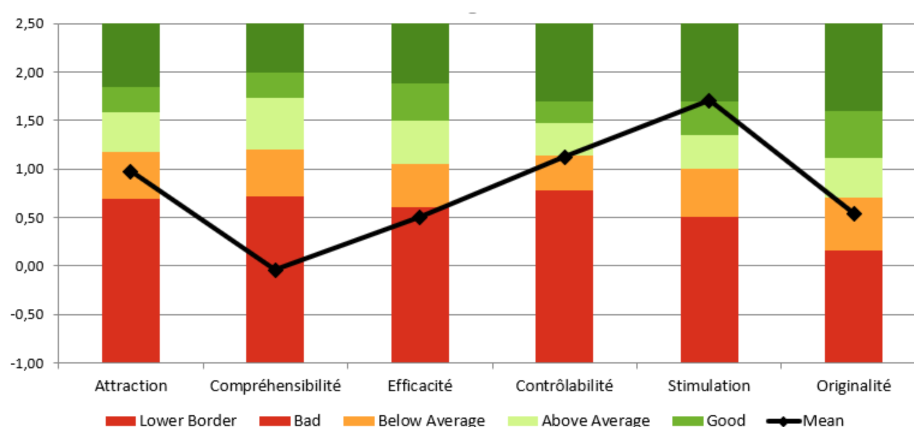


Figure 21: Benchmark du plugin LASoT.

Selon la situation des points formant la courbe, il est possible de savoir si la solution se trouve dans la moyenne, en dessous ou au-dessus. Un résultat *excellent* signifie que le produit se place dans les 10% des mieux cotés pour cette caractéristique. A l'opposé, *mauvais* détermine que le produit se situe dans les 25% des pires. Il en ressort que *LASoT* se démarque par sa stimulation. Sa contrôlabilité le place dans la norme. Par contre, la solution est en dessous de la moyenne du points de vue des aspects d'attractivité, compréhensibilité, efficacité et originalité. on peut en déduire, par rapport au graphe précédent, que les résultats positifs donnés précédemment ne le sont pas forcément comparés aux 452 autres produits évalués.

L'outil d'analyse fournit encore d'autres graphes qui permettent d'avoir une vue plus détaillée des résultats pour chaque question. Ceux-ci permettent d'avoir un avis plus précis sur ce qui a bien fonctionné ou pas. Ces graphes sont disponibles en annexes avec les figures 23 et 25.

[24]https://www.ueq-online.org/

Dans le formulaire, quelques questions plus ouvertes ont été posées :

- Comment s'est passée l'utilisation du plugin de manière générale ? Quel est le ressenti ?
- L'extension pourrait-elle aider des étudiants en master à améliorer une suite de tests (tout en sachant qu'elle pourrait être améliorée dans le futur) ?
- Quelle est la fonctionnalité la plus utile?
- Quelles améliorations pourraient être apportées à la solution ?

Le but de ces questions est de récupérer plus en profondeur le ressenti et avis des étudiants suite à l'utilisation du plugin. Il était également mis à disposition un champ non obligatoire leur permettant d'apporter des commentaires supplémentaires. Certains de ces étudiants, travaillent dans le domaine du développement et pourraient apporter des idées précieuses.

Les participants ont tous répondu de manière positive concernant la capacité de l'extension à aider les étudiants à améliorer une suite de tests. Concernant le ressenti du déroulement de l'expérience de manière générale, la plupart sont d'accord sur le fait qu'il faut un temps d'adaptation par rapport à la compréhension de l'utilisation du plugin et au préalable une introduction au mutation testing. Certains d'entre eux signalent un besoin d'encadrement pour des étudiants lors de l'utilisation. La partie qui concerne la génération des informations n'est pas assez intuitive. Il faut prendre en compte le fait que la plupart des participants n'ont pas consulté les consignes qui étaient disponibles dans le dépôts sur GitHub afin de leur permettre de comprendre les fondements du mutation testing avant de commencer l'expérience. Le mutation testing n'étant pas une matière simple à assimiler, l'ajout d'un projet inconnu comme SUT, ajoute une difficulté supplémentaire. L'expérience comporte donc un exercice de lecture de code. L'étudiant doit comprendre qu'une amélioration des assertions des tests passe d'abord par une compréhension des tests et des méthodes exécutées par ceux-ci. De manière générale, les informations incorporées dans l'EDI sont utiles. A partir de la liste des méthodes signalées jusqu'aux informations détaillées de Reneri situées dans les tests, les utilisateurs ont apprécié le fait de pouvoir les consulter directement dans le code. Les suggestions apportées concernant les améliorations sont exploitées dans le chapitre suivant.

5.3 Conclusion

Ces graphes ne permettent pas de déterminer directement ce qu'il faut changer mais ils permettent de faire au moins des suppositions sur les domaines où les améliorations auront le plus d'impact. A partir des graphes fournis par l'outil d'analyse, il est possible de faire au moins quelques hypothèses sur les aspects qui nécessitent des améliorations. En tenant compte du fait que les participants n'avaient pas de notions de mutation testing avant de commencer l'expérience, on peut tout de même observer une assimilation rapide des notions de cette technique. Grâce à la mise en pratique des outils dans un projet adapté au niveau des étudiants, ceux-ci ont pu comprendre l'avantage de cette pratique.

6 Travaux futurs

Dès le début de l'analyse du projet des limites ont été définies concernant les objectifs a atteindre. Le but ici était de concevoir un plugin qui incorpore les informations fournies par les plugins PITest Descartes et Reneri dans un EDI de manière à ce que celles-ci puissent guider l'étudiant à améliorer une suite de tests. Ce projet pourrait prendre une autre envergure dans le domaine de l'éducation ou il pourrait tout simplement être amélioré. Suite à l'expérience menée, beaucoup de retours et d'observations permettent déjà d'enrichir le plugin. Ce dernier est une version expérimentale et donc plusieurs modifications peuvent être envisagées.

6.1 Amélioration des conseils

Certains retours des étudiants remarquent que les conseils affichés dans les sur-couches qui apparaissent lorsque l'étudiant survol une partie de code mis en surbrillance ne sont pas suffisamment clairs. Après la lecture de ceux-ci, certains d'entre eux n'avaient pas compris l'action qu'ils devaient prendre ensuite. La sur-couche des méthodes contient la classification de celle-ci, les mutations qui n'ont pas été détectées et les tests qui exécutent cette méthode. Peut-être que l'ajout d'un commentaire sous forme de question à côté des tests pourrait aider l'étudiant dans la réflexion. "Un de ces tests pourrait-il détecter cette mutation?". La sur-couche des tests quant à elle, contient des informations sur la valeur retournée de la version originale du programme et de la transformation concernant la méthode testée. L'ajout d'un commentaire pourrait également aider la réflexion : "Comment modifier l'assertion de ce test pour détecter cette valeur?". Il ne faut pas omettre le fait que la plupart des étudiants qui ont effectué l'expérience ont eu une tendance à vouloir aller trop vite. Certains ne connaissaient pas le mutation testing et n'avaient pas pris le temps de lire la documentation mise à disposition dans le dépôt. Ils se retrouvaient donc perdu lors de la lecture des informations.

6.2 Consulter les résultats précédents

Lorsque des corrections ont été apportées aux tests et que le score de mutation a été amélioré, l'utilisateur n'a plus accès aux mutations qui ont été détectées. Un rapport sur les améliorations apportées pourrait être intéressant. Ce rapport contiendrait une liste des méthodes avec leur classification présente et précédente. Ce rapport pourrait également contenir les mutations qui ont été détectées pour ces méthodes alors qu'elles avaient survécu lors du dernier cycle. Ces informations apporteraient un retour supplémentaire aux étudiants sur leur progression.

6.3 Support à l'utilisation

L'expérience a permis de mettre en évidence les faiblesses du plugin. Certains des participants ont remarqué le manque d'appuis pédagogique, de guide durant l'utilisation. Arrivés à une certaine étape, ils se retrouvaient perdu. Une documentation était mise à disposition au niveau du dépôt sur GitHub et le Wizard a été développé dans ce sens mais cette présentation n'était peut-être pas la mieux adaptée ou du moins pas assez complète. Il serait utile d'ajouter une information incorporée dans une des représentation qui indique l'état du plugin dans le processus représenté à la figure 7. Cette donnée pourrait être affichée de manière discrète au niveau de l'*explorer* ou plus en détail dans une vue Web incorporée dans l'EDI comme présenté à la figure 22. Le plugin L'API fournie pour développer des extensions sur Visual Studio Code permet de réaliser une page d'accueil sur laquelle il serait possible d'afficher l'état et un commentaire sur l'étape suivante à exécuter.

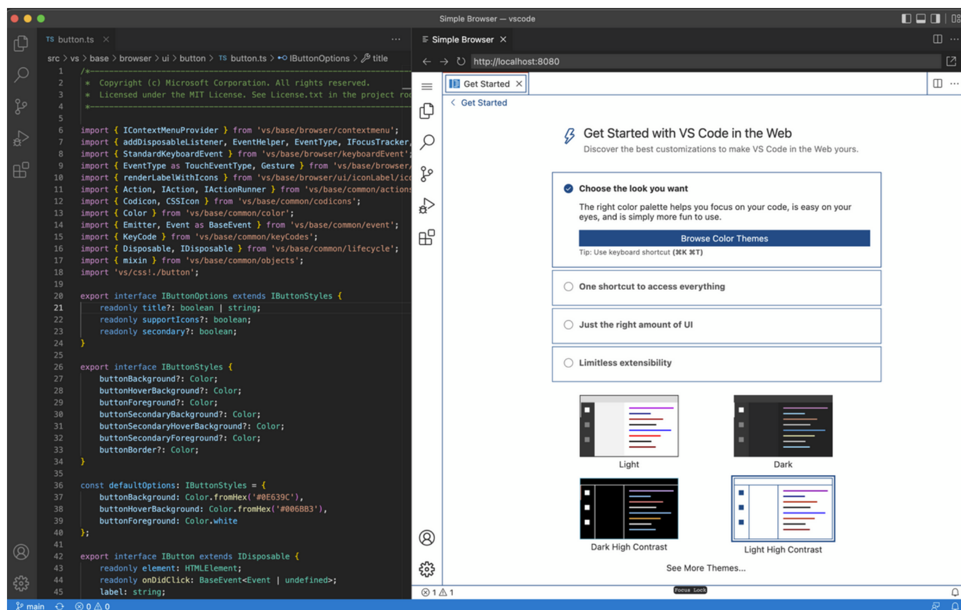


Figure 22: Vue Web intégrée à l'EDI.

6.4 Consulter les documents HTML de PITest

Afin de consulter les rapports sous format HTML fournis par PITest, le développeur doit quitter l'EDI pour afficher le document à partir d'un Navigateur. L'API de VSCode fourni une fonctionnalité WebView[25] permettant d'afficher directement du contenu HTML dans une vue intégrée dans l'EDI. Cette implémentation pourrait permettre de réduire le nombre de tâches à

[25] https://code.visualstudio.com/api/ux-guidelines/webviews

accomplir par l'étudiant pour consulter les résultats fournis par PITest et lui éviter de quitter l'EDI.

6.5 Paramétrage et visualisation

Ajouter des paramètres de configuration de l'extension. Par exemple, un paramètre qui permet d'adapter le style ou les couleurs utilisées pour décorer le code source. Cette fonctionnalité pourrait être utile pour des personnes daltoniennes. Etant donné que la configuration d'une décoration est sur base de CSS (Cascading Style Sheet), les possibilités de mettre du code en évidence sont nombreuses.

Un autre paramètre qui pourrait être intéressant dans le processus du mutation testing est la définition du seuil de score de mutation. Celui-ci pourrait servir de référence par rapport au résultat affiché dans la barre de statuts. Le développeur pourrait être directement averti si le score de mutation convient aux exigences de qualités prédéfinies.

Le moteur de mutations extrêmes de Descartes permet la configuration des opérateurs de mutation. L'extension pourrait aider l'utilisateur au paramétrage de ceux-ci via une page d'accueil ou via le *Wizard*.

6.6 Service web et base de données

Il serait intéressant d'étendre les capacités de l'extension du côté du professeur. Comme expliqué dans l'état d'art les professeurs rencontrent également des difficultés dans le domaine de l'éducation du software testing. Ils doivent évaluer les élèves et leur donner des commentaires sur les façons de s'améliorer. Ces retours leurs demandent beaucoup de temps. Des outils comme *WReSTT* ou *Web-CAT* ont déjà été réalisé dans ce sens. Il serait avantageux d'apporter cette dimension à l'extension *LASoT*. Un service web pourrait jouer le rôle de centre de communication entre les étudiants et les professeurs. Ce service pourrait récupérer les rapports de Descartes et Reneri et générer un compte rendu au professeur. Ensuite, il pourrait communiquer vers les étudiants pour leurs transmettre des conseils ou des évaluations. Avec cette application, on pourrait envisager une évaluation automatique des tests. Par exemple pour un projet particulier avec une suite de tests au préalable établie par les professeurs qui servirait de base pour effectuer des comparaisons. Ceci allégerait fortement la charge de travail des enseignants.

6.7 Gamification

Dans le cadre ou une application web est développée pour centraliser les informations des étudiants, une dimension *Gamification* pourrait être implémentée.

Celle-ci pourrait prendre la forme de récompenses et de progressions par rapports à des projets de tests de niveaux de difficulté différents. Ces aspects peuvent inciter les étudiants à continuer de s'entraîner et d'améliorer leurs compétences. Ils permettent notamment de donner un objectif supplémentaire. Il est également possible d'imaginer un aspect de compétition avec des classements et l'accès à des statistiques ce qui permettrait aux élèves de se confronter entre eux et de leur donner une raison supplémentaire pour s'améliorer et devenir meilleur que les autres.

6.8 Automatisation

Un utilisateur a retourné l'idée de pousser cette technique encore plus loin jusqu'à la correction des tests de manière automatique. Des recherches ont été effectuées dans le but d'automatiser la tâche qui consiste à concevoir des oracles de tests afin que ceux-ci assurent le fait que le programme ait le comportement souhaité. Des outils comme EvoSuite [13] [26] ont été développés afin de générer des oracles de tests en s'appuyant sur l'analyse par mutation. Ces recherches pourraient être un point de départ afin d'automatiser ce processus.

6.9 Accès rapide aux méthodes et tests

Des participants de l'expérience ont fait part du manque de raccourcis pour accéder aux méthodes ou aux tests. Ils devaient retenir le nom de ceux-ci et chercher leur emplacement dans les fichiers. Ces raccourcis pourraient réduire la charge cognitive de l'étudiant et le nombre d'actions nécessaires à effectuer pour accéder au code problématique.

6.10 Améliorer la validation

Lors de la validation, la plupart des participants n'avaient pas pris le temps de lire la documentation fournie disponible sur le dépôt à propos du mutation testing et des plugins Descartes et Reneri. Les consignes généraux ne mettaient peut être pas suffisamment d'importance à la compréhension des différents concepts avant de commencer l'utilisation du plugin *LASoT*. Une expérience future devrait souligner ce point en fournissant une meilleure documentation plus attractive. De plus, les améliorations apportées au plugin concernant l'accompagnement de l'utilisateur dans l'utilisation du plugin devrait faciliter la compréhension de celui-ci.

[26] https://www.evosuite.org/

7 Conclusion

Le software testing est une matière délaissée dans les cursus informatique. L'intégrer dans des cours de programmation nécessite des outils automatisés afin de réduire la charge de travail des professeurs et des étudiants. Les tests par mutation ont prouvé leurs avantages du point de vue de la génération des défauts et corrige certaines lacunes du critère de couverture de code des tests traditionnel. D'autre part, les concepts de gamification et de collaboration ont prouvé qu'ils pourraient apporter une réelle solution à l'apprentissage du testing avec les outils **Code Defender** ou **WReSTT**. Avec le moteur de mutation extrême de **Descartes**, ses rapports et les informations détaillées du plugin **Reneri**, le mutation testing prend une dimension éducationnelle. L'incorporation des informations générées par ces outils dans un environnement de développement tel que Visual Studio Code prouve que cette méthode peut aider les étudiants à écrire de meilleurs tests unitaires. Les tests par mutation extrême sont complémentaires aux pratiques actuelles de test de logiciels. L'utilisation du plugin **Reneri** est un processus assez lent qui terni un peu la rapidité des tests par mutation extrême. Néanmoins les informations apportées par cet outil permette de guider plus en profondeur les étudiants dans l'assertion de leurs tests. Les futures recherches devraient enquêter sur une manière d'intégrer ce type de plugin dans un outil plus complet qui englobe les concepts de collaboration et de gamification.

References

[1] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[2] Mauricio Aniche, Felienne Hermans, and Arie Van Deursen. "Pragmatic software testing education". In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 2019, pp. 414–420.

[3] Ingrid A Buckley and Winston S Buckley. "Teaching software testing using data structures". In: *International Journal of Advanced Computer Science and Applications* 8.4 (2017).

[4] Kevin Buffardi and Stephen H Edwards. "A formative study of influences on student testing behaviors". In: *Proceedings of the 45th ACM technical symposium on Computer science education*. 2014, pp. 597–602.

[5] Thierry Titcheu Chekam et al. "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE. 2017, pp. 597–608.

[6] Zhenyu Chen, Atif Memon, and Bin Luo. "Combining research and education of software testing: a preliminary study". In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 2014, pp. 1179–1180.

[7] Peter J Clarke et al. "Integrating testing into software engineering courses supported by a collaborative learning environment". In: *ACM Transactions on Computing Education (TOCE)* 14.3 (2014), pp. 1–33.

[8] Peter J. Clarke et al. "Collaborative Web-Based Learning of Testing Tools in SE Courses". In: SIGCSE '11. Dallas, TX, USA: Association for Computing Machinery, 2011. ISBN: 9781450305006. DOI: 10.1145/1953163.1953208.

[9] Benjamin S Clegg, José Miguel Rojas, and Gordon Fraser. "Teaching software testing concepts using a mutation testing game". In: *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET)*. IEEE. 2017, pp. 33–36.

[10] Henry Coles et al. "PIT: a practical mutation testing tool for Java (demo)". In: 2016, pp. 449–452. DOI: 10.1145/2931037.2948707.

[11] Pedro Delgado-Pérez et al. "Mutation Testing and Self/Peer Assessment: Analyzing their Effect on Students in a Software Testing Course". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 2021, pp. 231–240. DOI: 10.1109/ICSE-SEET52601.2021.00033.

[12] Stephen H. Edwards and Zalia Shams. "Do Student Programmers All Tend to Write the Same Software Tests?" In: *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education*. ITiCSE ́14. Uppsala, Sweden: Association for Computing Machinery, 2014, pp. 171–176. ISBN: 9781450328333. DOI: 10.1145/2591708.2591757.

[13] Gordon Fraser and Andrea Arcuri. "Evosuite: automatic test suite generation for object-oriented software". In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011, pp. 416–419.

[14] Gordon Fraser et al. "Gamifying a software testing course with code defenders". In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 2019, pp. 571–577.

[15] Vahid Garousi et al. "Software-testing education: A systematic literature mapping". In: *Journal of Systems and Software* 165 (2020), p. 110570. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110570.

[16] Alessio Gaspar and Sarah Langevin. "Active learning in introductory programming courses through student-led "live coding" and test-driven pair programming". In: *International Conference on Education and Information Systems, Technologies and Applications, Orlando, FL*. 2007.

[17] Marinos Kintis et al. "How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults". In: *Empirical Software Engineering* 23.4 (2018), pp. 2426–2463.

[18] Vesa Lappalainen et al. "ComTest: a tool to impart TDD and unit testing to introductory level programming". In: *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*. 2010, pp. 63–67.

[19] Thomas Laurent et al. "Assessing and improving the mutation testing practice of pit". In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2017, pp. 430–435.

[20] Xinhong Liu and Reid Holmes. "Exploring developer preferences for visualizing external information within source code editors". In: *2020 Working Conference on Software Visualization (VISSOFT)*. IEEE. 2020, pp. 27–37.

[21] Rainer Niedermayr, Elmar Jürgens, and Stefan Wagner. "Will My Tests Tell Me If I Break This Code?" In: *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)* (2016), pp. 23–29.

[22] A. Jefferson Offutt. "Investigations of the Software Testing Coupling Effect". In: *ACM Trans. Softw. Eng. Methodol.* 1.1 (Jan. 1992), pp. 5–20. ISSN: 1049-331X. DOI: 10.1145/125489.125473.

[23] Rafael A. P. Oliveira et al. "Evaluation and assessment of effects on exploring mutation testing in programming courses". In: *2015 IEEE Frontiers in Education Conference (FIE)*. 2015, pp. 1–9. DOI: 10.1109/FIE.2015.7344051.

[24] Rafael AP Oliveira et al. "On using mutation testing for teaching programming to novice programmers". In: *Proceedings of the 22nd International Conference on Computers in Education (ICCE'14), Nara, Japan*. 2014, pp. 394–396.

[25] Mike Papadakis et al. "Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE. 2018, pp. 537–548.

[26] Mike Papadakis et al. "Chapter Six - Mutation Testing Advances: An Analysis and Survey". In: ed. by Atif M. Memon. Vol. 112. Advances in Computers. Elsevier, 2019, pp. 275–378. DOI: https://doi.org/10.1016/bs.adcom.2018.03.015.

[27] Mike Papadakis et al. "Threats to the validity of mutation-based test assessment". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 2016, pp. 354–365.

[28] Raphael Pham et al. "Enablers, Inhibitors, and Perceptions of Testing in Novice Software Teams". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, pp. 30–40. ISBN: 9781450330565. DOI: 10.1145/2635868.2635925. URL: https://doi.org/10.1145/2635868.2635925.

[29] Lilian Passos Scatalon, Ellen Francine Barbosa, and Rogerio Eduardo Garcia. "Challenges to integrate software testing into introductory programming courses". In: *2017 IEEE Frontiers in Education Conference (FIE)*. 2017, pp. 1–9. DOI: 10.1109/FIE.2017.8190557.

[30] Martin Schrepp, Jörg Thomaschewski, and Andreas Hinderks. "Construction of a benchmark for the user experience questionnaire (UEQ)". In: (2017).

[31] Thierry Titcheu Chekam et al. "Selecting fault revealing mutants". In: *Empirical Software Engineering* 25.1 (2020), pp. 434–487.

[32] Oscar Vera Pérez, Martin Monperrus, and Benoit Baudry. "Descartes: A PITest Engine to Detect Pseudo-Tested Methods - Tool Demonstration". In: Sept. 2018, pp. 908–911. DOI: 10.1145/3238147.3240474.

8 Annexes

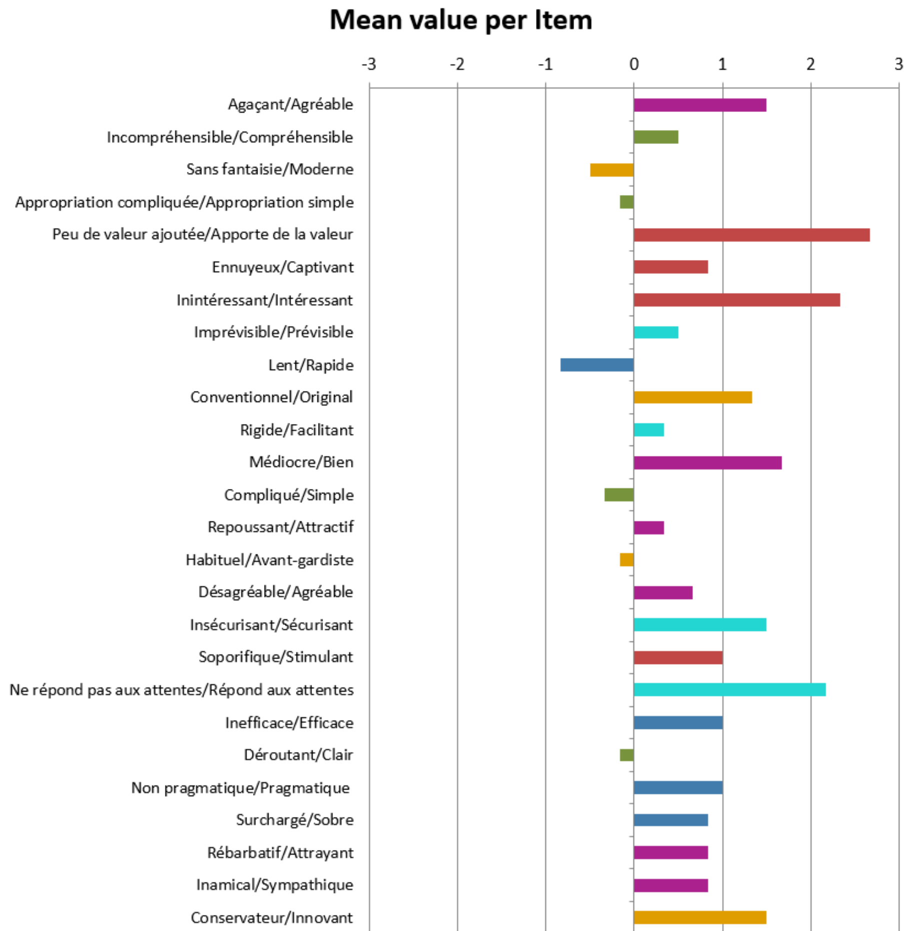


Figure 23: UEQ : Résultat par question.

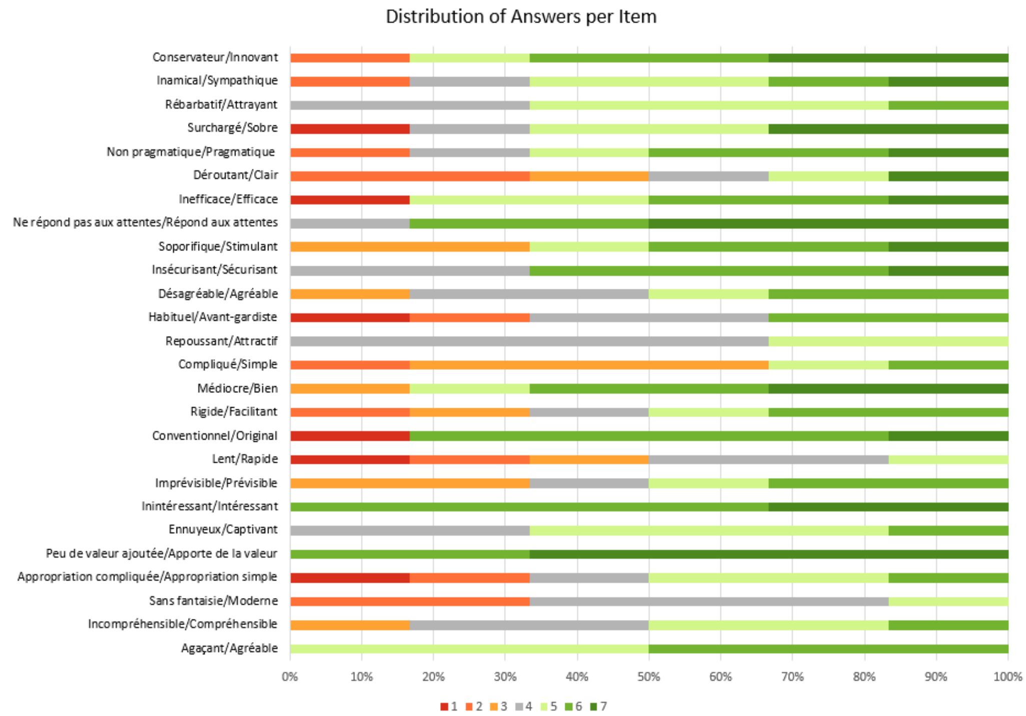


Figure 24: UEQ : Distribution des réponses.

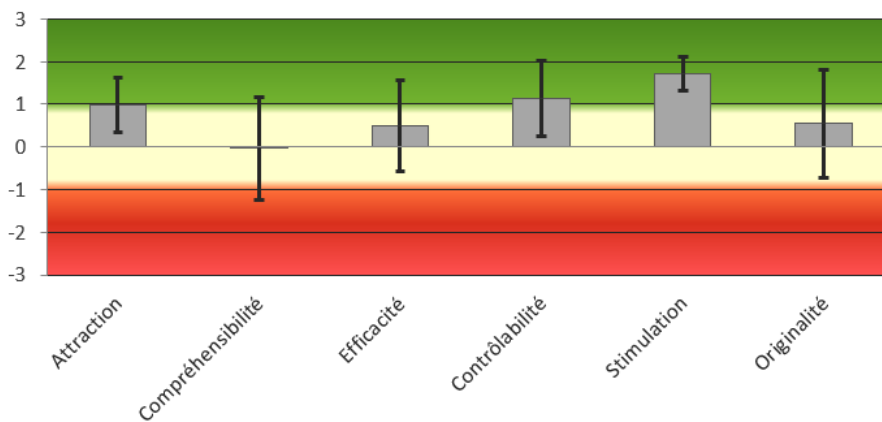


Figure 25: UEQ : Variance par aspect qualité.

Acronyms

API Interface de programmation. 25

AST arbre syntaxique abstrait. 13

BPMN Business Process Model and Notation. 32

CSS Cascading Style Sheet. 59

CSV Comma-separated values. 14

EDI Environnement de développement. 7, 15

HTML Hypertext Markup Language. 13–15

JSON JavaScript Objet Notation. 14

MuTEd Mutation Testing Education. 50

RIPR Reachability, Infection, Propagation, Revealability. 10

SUT System under test. 8

TDD Test Driven Development. 18, 19

UEQ User Experience Questionnaire. 51

WResTT Web-Based Repository of Software Testing Tools. 21

XML Extensible Markup Language. 14

Glossary

benchmark une norme ou un point de référence par rapport auquel les choses peuvent être comparées. 54

bytecode est un code intermédiaire entre les instructions machines et le code source, qui n'est pas directement exécutable. 12

Coupling Effect suite de test qui détecte des mutants de premier ordre, peut également détecter des mutations plus complexes. 11

EvoSuite EvoSuite est un outil qui génère automatiquement des tests unitaires pour des programmes en Java. 20

framework ensemble d'outils et de composants logiciels à la base d'un logiciel ou d'une application.. 12

GitHub est un service d'hébergement pour le développement de logiciels et le contrôle de version à l'aide de Git. 58

JUnit JUnit est un framework de test logiciel pour le langage Java. 19

Major est un framework de mutation testing qui manipule l'arbre syntaxique abstrait (AST) du programme testé. 13

Maven est un outil de gestion et d'automatisation de production des projets logiciels Java. 15

muJava sorti en 2003, est un des plus anciens système de mutation pour des programmes en Java. 12

Node.js est un environnement d'exécution JavaScript open-source, cross-plateforme et back-end qui s'exécute sur un moteur JavaScript et exécute du code JavaScript en dehors d'un navigateur Web, conçu pour créer des applications réseau évolutives. 48

PITest sorti en 2010, est un framework open source qui cible l'utilisation du mutation testing dans le domaine industriel. 12

plugin est un module personnalisé. 25

strong mutation mutations qui exposent une variation en sortie par rapport au programme original. 10

tests unitaires tests se concentrant sur une unité de programme. 6

VSCode est un éditeur de code source créé par Microsoft for Windows, Linux and macOS. 23