Projet de Software Evolution

Rapport final

Maximilien Charlier
Jannou Brohée
Julien Delplanque

UMONS
Faculté des Sciences
MaB1 Sciences - Informatiques

Année académique
2015 - 2016

Table des matières

1 Introduction

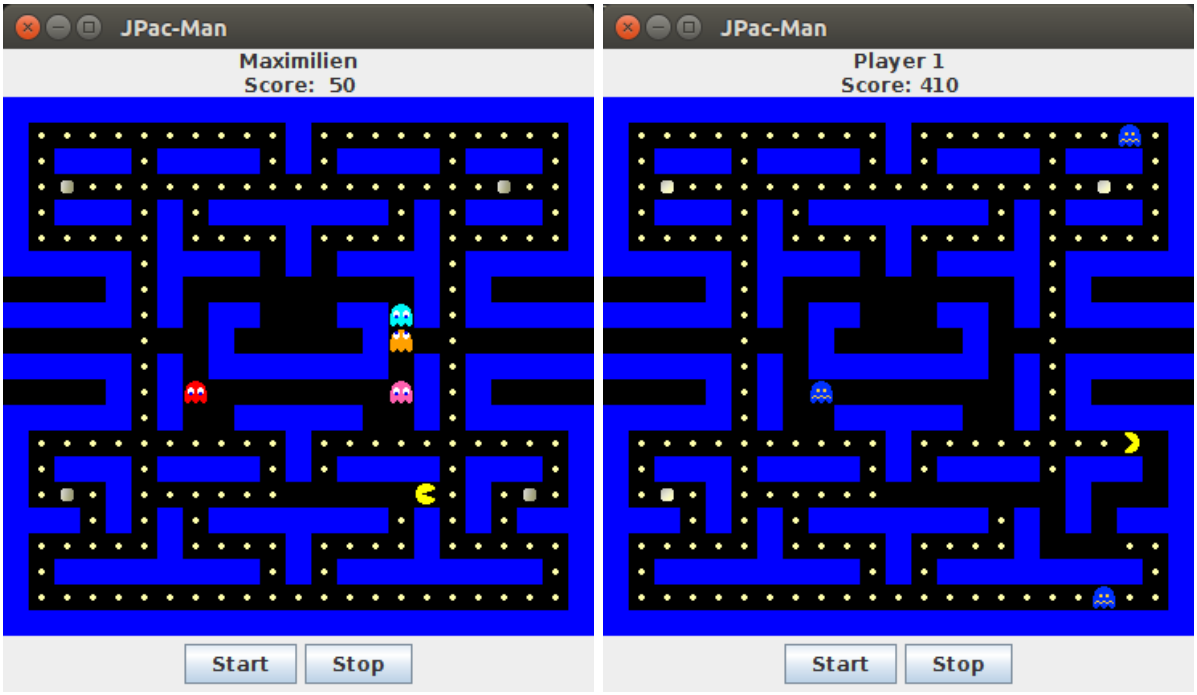
Dans le cadre du cours de Software Evolution, il nous a été demandé d'analyser et de modifier un logiciel. Ce logiciel s'appelle `JPacman` et a été développé principalement par *Arie van Deursen* et *Jeroen Roosen*[1]. Il consiste en une implémentation basique du jeu PacMan en Java. Dans cette implémentation, les extensions du jeu sont volontairement absentes. Ne sont présents qu'une carte, les fantômes, PacMan, les gommes et un système de score. Ce logiciel est disponible librement sur la plate-forme `Git Hub` disponible à cette adresse `https://github.com/SERG-Delft/jpacman-framework`. Ce projet a été réalisé par groupe de trois, chaque étudiant a dû implémenter une extension individuellement. Ces trois extensions ont ensuite du être fusionné dans un seul logiciel. Pour cela, l'utilisation de la fonction `Pull Request` de `GitHub` a été demandée. Une fois les trois extensions intégrées, une analyse pour comparer la qualité du code source final avec celle du code source initial en se basant sur les concepts vus en cours a été effectuée.

1.1 Extensions individuelles

1.1.1 Super-gomme

Une super-gomme est une gomme qui une fois mangée par Pacman inverse les règles du jeu. À la place d'être chassé par les fantômes, ceux-ci deviennent bleus et Pacman peut les dévorer. L'effet de la super-gomme est limité dans le temps et n'affecte que les fantômes "vivant" au moment où l'on mange la super gomme. Certains fantômes peuvent en effet être "mort" (mangé précédemment) lorsque l'on mange une super gomme, ceux-ci réapparaîtront en mode chasseur directement. La figure 1.2(a) illustre le jeu en mode super-gomme. On remarque que avant d'avoir mangé une super-gomme, gomme en surbrillance présente dans chaque coin du jeu, le jeu se comporte comme le jeu original. La figure 1.2(b) illustre quant à elle le jeu après avoir mangé une super-gomme. Comme on peut le constater, les fantômes sont devenus bleus et Pacman peut les manger, un des fantômes est d'ailleurs absent, car déjà mangé par Pacman. Les fantômes une fois mangés, réapparaissent au milieu du jeu après un certain laps de temps. Ils réapparaissent toujours en mode chasseur.

1. Source :`https://github.com/SERG-Delft/jpacman-framework`

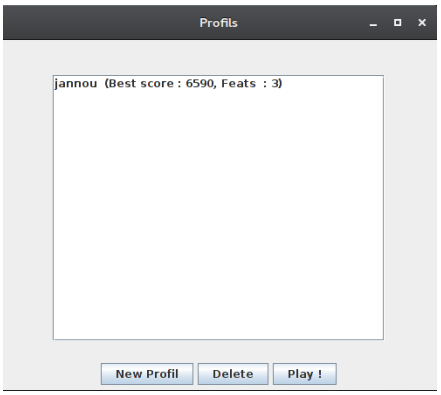


(a) Avant d'avoir mangé une super-gomme. (a) Après avoir mangé une super-gomme.

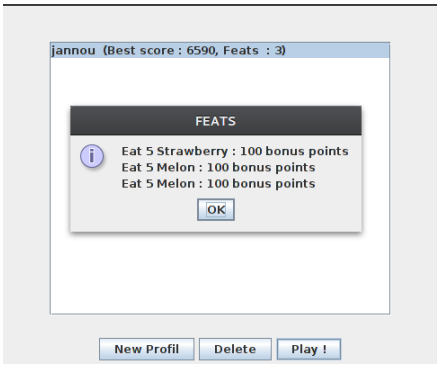
FIGURE 1.1 – Le jeu Pacman en mode super-gomme.

1.1.2 Score

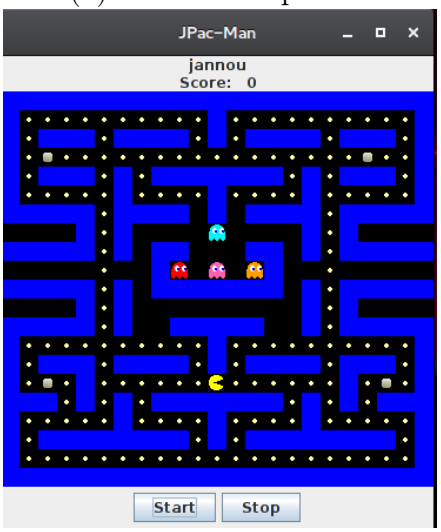
Score est une extension permettant à un joueur de choisir un profil. Pour chaque profil on enregistre diverses actions et informations telles que le score maximal réalisé et des exploits. Un exploit est une action spécifique que le joueur réalise. Chacune de ces actions rapporte un certain nombre de points au joueur lorsqu'il les réalise. De plus, en fin de chaque partie et pour tous les joueurs, l'extension affiche un tableau des 10 meilleurs scores réalisés.
Les différents aperçus suivants illustrent l'apport de cette extension par rapport à la version de base du jeu.



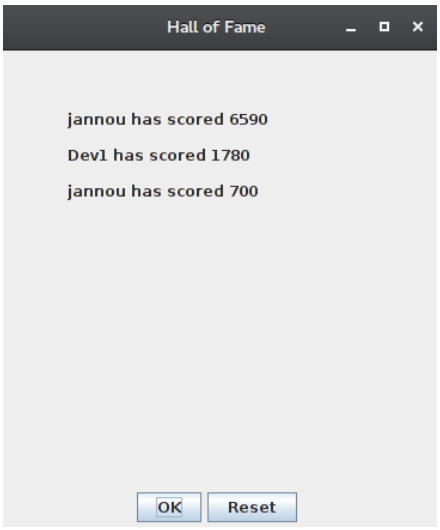
(a) Choix d'un profil .



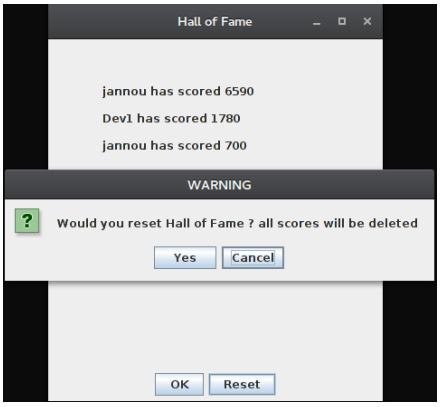
(b) Proposition d'exploits qui n'ont pas encore été réalisés.



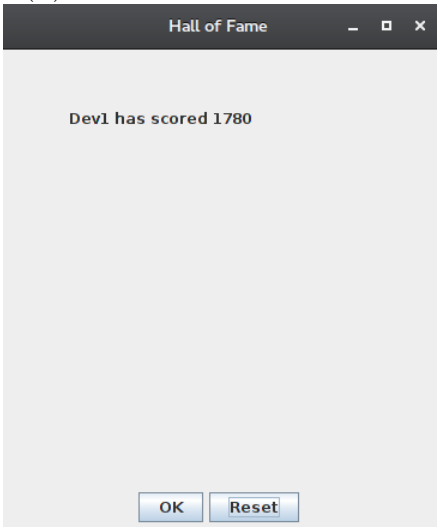
(c) Partie en cours pour le profil "jannou".



(d) Table des meilleurs scores



(e) Remise à zéro des meilleurs scores réalisés par les joueurs.



(f) Dev1 est un score à battre et est automatiquement le premier score enregistré dans Hall of Fame.

FIGURE 1.2 – Le jeu Pacman en mode super-gomme.

1.1.3 Multi-joueur sans Pacman

Cette extension bouleverse complètement le "gameplay" original. Elle consiste en un mode multijoueur où chaque joueur incarne un fantôme. À intervalle régulier, les fantômes se transforment tour à tour en Pacman. Lorsqu'un fantôme prend la peau de Pacman, il devient alors chasseur des autres fantômes et peut tenter de les manger afin d'augmenter son score et de diminuer celui du fantôme mangé. Le reste du temps, les joueurs fantômes ont pour but de manger le plus de "pellets" possible et d'éviter de se faire manger par le joueur qui chasse.
Ce mode de jeu est jouable par un groupe de 2 à 4 joueurs. Si moins de 4 joueurs participent aux jeux, des IA remplacent les joueurs humains manquants.
Un aperçu de l'extension est visible en FIGURE 1.3.

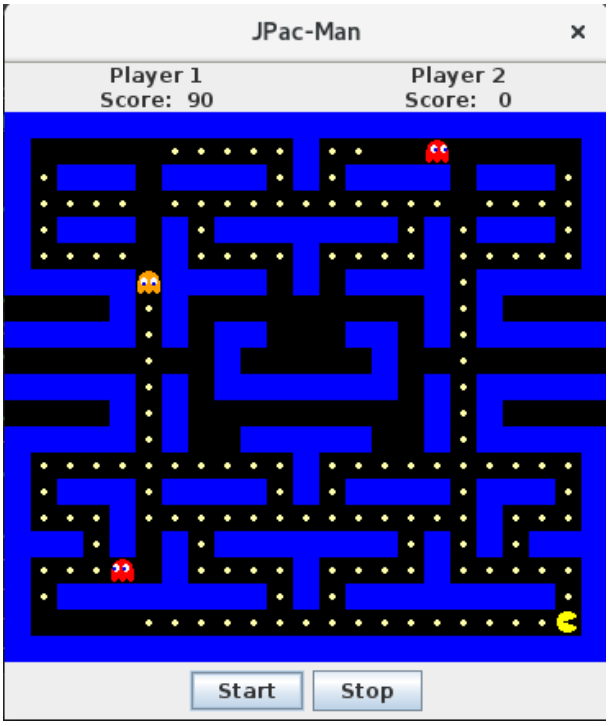


FIGURE 1.3 – JPacman en mode "Multiplayer without pacman".

1.2 Dépôt `Git Hub`

Chaque extension a été développée individuellement par un membre du groupe. Nous illustrons dans la TABLE 1.1 les adresses de chaque dépôt du groupe ainsi que l'extension développée par le membre et le `tag` illustrant la fin de la première partie du projet.

Membre	Dêpot `Git Hub`	Extension implémentée	Tag
Charlier Maximilien	Neosw	Super-gomme	echeance1
Brohée Jannou	Jannou	Score	Personal_Work
Delplanque Julien	juliendelplanque	Multi-joueur sans Pacman	personal-work

TABLE 1.1 – Dépôt `Git Hub` utilisé par les membres du groupe.

2 Intégration

Avant de commencer l'implémentation des extensions individuelles, les membres du groupe se sont concertés et ont fixés quelques règles à respecter afin de rendre la fusion des trois extensions la moins difficile possible. Ces règles sont les suivantes :

1. Limiter les modifications de classes déjà présentes dans le système. Préférer plutôt l'ajout de sous-classes pour modifier un comportement.
2. Dans la même logique que la règle 1, créer un lanceur séparé pour chaque extension (et donc ne pas modifier le lanceur original).

Cette démarche nous a permis de n'avoir que très peu de conflit (comme expliqué ci-dessous).

2.1 Conflit & résolution

Tout d'abord, l'extension "Multiplayer without Pacman" a été intégrée au projet original jPacman. Cette première intégration n'a créé aucun conflit et la "pull-request" associée a été résolu automatiquement par Github.
Ensuite, l'extension "Super gomme" a été intégrée. Celle-ci n'a créé que quelques conflits de Java-doc, quelques variables d'instances qui passaient de "private" à "protected" et un conflit d'une ligne de code dans qui créait un bug dans le programme.
Finalement, l'extension "Score" a été intégrée au projet sans créer de conflit mis à part quelques variables d'instances qui passaient de "private" à "protected".
Nous pouvons donc conclure que les règles que nous nous sommes imposés lors du développement des extensions individuelles ont porté leurs fruits.

2.2 Mise en place d'une interface commune

Une fois les trois extensions intégrées au projet, une interface graphique a été créée afin de pouvoir démarrer les différents modes de jeu depuis une même fenêtre. Cette fenêtre (Figure 2.1) propose quatre boutons, le premier pour le mode de jeu sans extension, le second pour le mode de jeu "Super gomme" et "Score" et les suivants pour le mode de jeu "Multiplayer without Pacman".

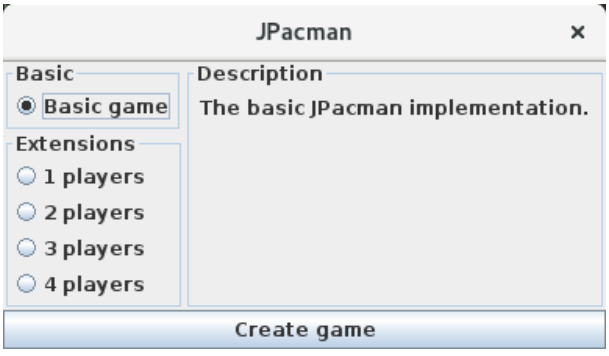


Figure 2.1 – Interface graphique commune à tous les modes de jeu.

Remarquons que les combinaisons "Super gomme" - "Multiplayer without pacman" et "Score" - "Multiplayer without pacman" n'étant pas compatible au niveau du

"gameplay", il est impossible de lancer le programme avec de telles combinaisons d'extension. Cette limitation est réalisée par le biais de cette interface graphique commune.

3 Analyse

Nous allons comparer la qualité du code source avant et après nos modifications. Pour cela, nous allons utiliser deux techniques. L'analyse des métriques et l'analyse des "Bad Smells" qui est la détection des mauvaises pratiques dans le code.

3.1 Analyse métrique

L'analyse métrique s'intéresse au nombre de ligne de code. Pour les compter, nous avons utilisé un plug-in pour `IntelIJ IDEA`. Celui-ci s'appelle `Metrics Reloaded` et est disponible sur `GitHub` à l'adresse suivante :
https://github.com/BasLeijdekkers/MetricsReloaded. Celui-ci s'installe via le menu `File` > `Settings` > `Plugins` > `Browse Repositories`. Pour l'utiliser, il suffit d'aller dans `Analyze` > `Calculate Metrics`.

3.1.1 Définition des métriques

- **Lines of code** (LOC) est la somme du nombre total de lignes de code du logiciel. Cette somme prend en compte les commentaires mais pas les espaces. On a vu au cours que cette métrique doit croître de façon linaire pour que le programme puisse continuer a être maintenu sur le long terme.
- **Lines of product code** (LOC_p) est la somme du nombre de ligne de code du logiciel en ne comptant pas les tests.
- **Lines of test code** (LOC_t) est la somme du nombre de ligne de code du logiciel en ne comptant que les tests.

3.1.2 Résultat avant intégration

Nous avons analysé les métriques sur le logiciel avant l'intégration de nos extensions. Les résultats sont stockés dans le tableau TABLE 3.1. On constate que la proportion de ligne de code attribué au test est d'environs 20%. Après intégration, cette proportion devrait en tout logique rester le même ou augmenter. On constate que ce ne sont pas forcement les classes qui contiennent le plus de lignes de code qui ont le LOC_t le plus grand. Par exemple la classe `level` contient le plus grand nombre de lignes de code mais a seulement 3e LOC_t le plus grand. Quant à la classe `board`, elle a le plus grand LOC_t mais contient presque 4 fois moins de ligne de code que la classe `level`.

3.1.3 Comparaison des résultats

La tableau TABLE 3.1 mets en avant le nombre de ligne de code avant et après nos ajout de fonctionnalité. On remarque que le nombre total de lignes de code a augmenté d'environs 60%. La métrique LOC_t a augmenté plus grandement que la métrique LOC_p, ce qui est un bon indicateur de la qualité des tests. La proportion de ligne de code dans les tests à augmenté de plus de 6% après l'ajout de nos fonctionnalité.

Métrique	**Avant intégration**	**Après intégration**	**Augmentation**
Lines of code (LOC)	5853	9294	+58.79%
Lines of product code (LOC_p)	4791	7497	+56.48%
Lines of test code (LOC_t)	1062	1797	+69.20%
Pourcentage de line de code de test $\frac{LOC_t}{LOC}$	$18,14\%$	19.33%	+**6.56%**

TABLE 3.1 – Comparaison des métriques avant et après intégration des extensions.

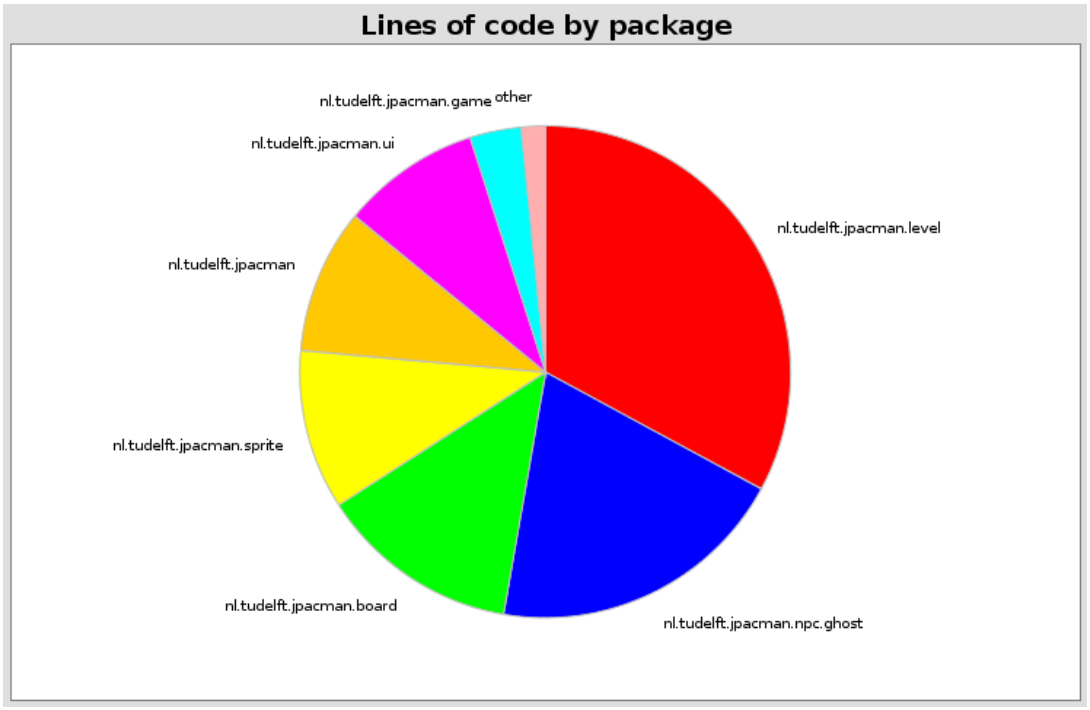


FIGURE 3.1 – (a) Avant

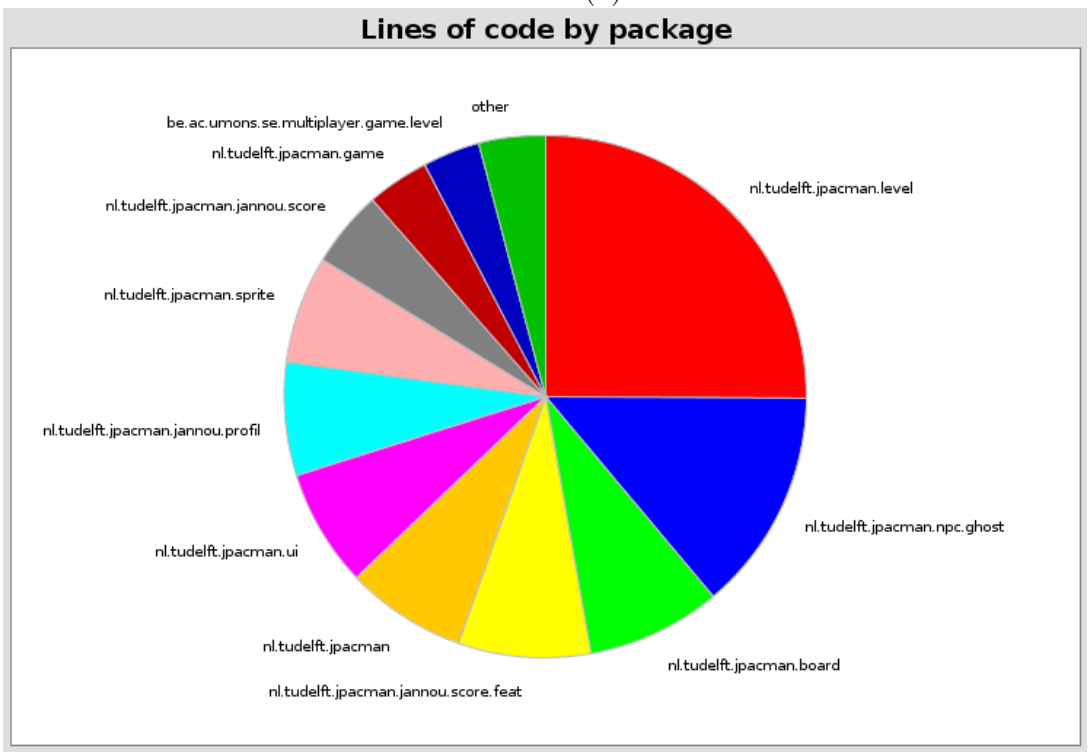


FIGURE 3.2 – (b)Après

FIGURE 3.3 – Pie chart du **Lines of code** (LOC) par paquet avant et après intégration des extensions.

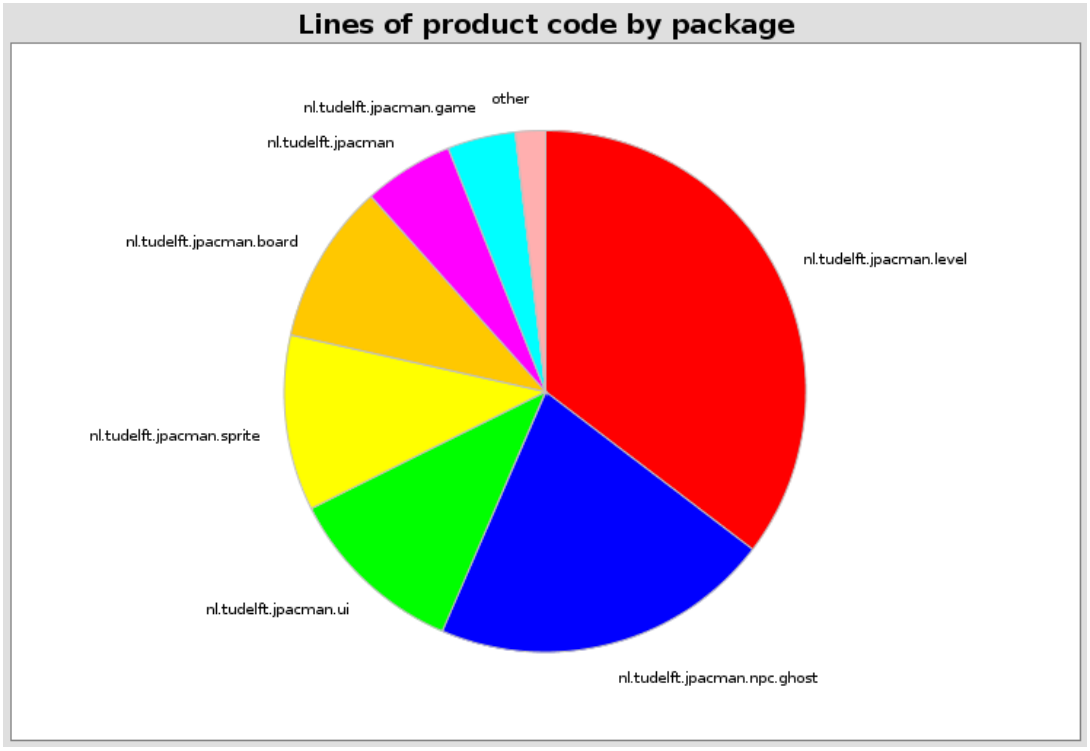


FIGURE 3.4 – (a) Avant

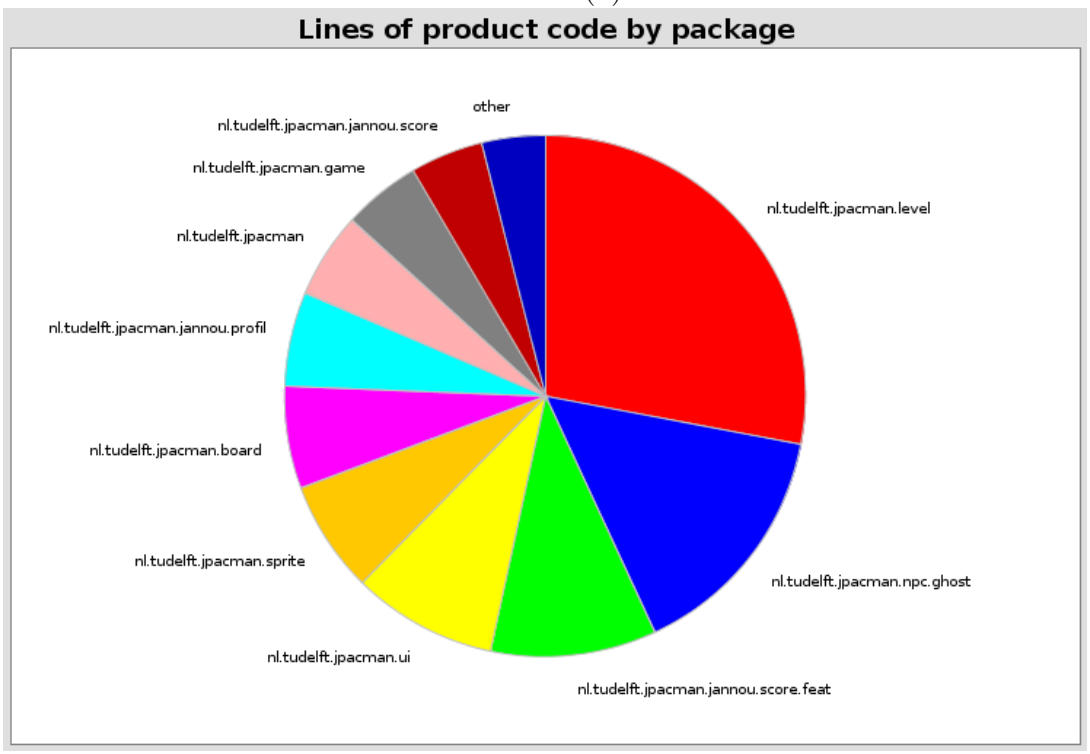


FIGURE 3.5 – (b)Après

FIGURE 3.6 – Pie chart du **Lines of product code** (LOC_p) par paquet avant et après intégration des extensions.

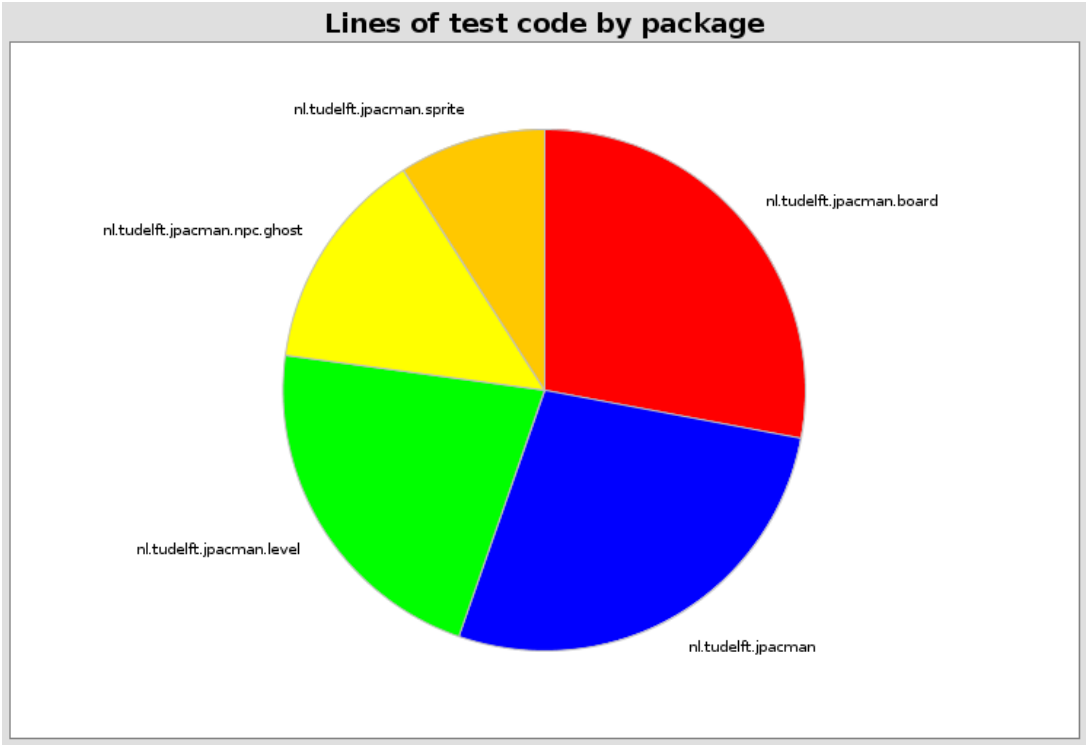


FIGURE 3.7 – (a) Avant

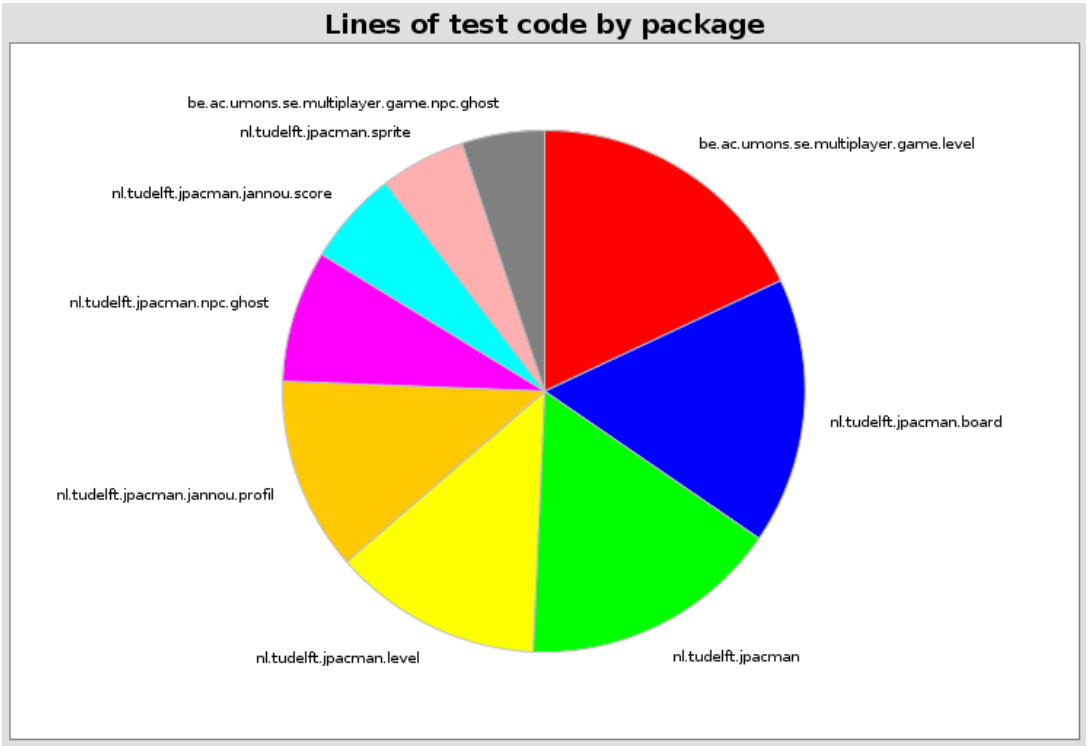


FIGURE 3.8 – (b)Après

FIGURE 3.9 – Pie chart du **Lines of test code** (LOC_t) par paquet avant et après intégration des extensions.

3.2 Analyse Bad Smell

L'analyse des Bad Smell s'intéresse maintenant au respect/violation de conventions diverses et variées permettant d'avoir une ligne directrice quant aux bonne pratiques de programmation. Pour ce faire, nous utilisons un plugin de `IntelIJ IDEA` qu'est `PMD` [2]. L'installation se fait comme pour `Metrics Reloaded`. Ci-dessous se trouvent les résultats de notre analyse du code avant et après intégration.

3.2.1 Explications des Bad Smells

Les Bad Smells sont des structures que l'on retrouve a différents endroits dans le code et qui suggèrent le besoins d'un refactoring.

- **Comment** : Conventions liées aux commentaires et qui sont gage de qualité.
  - (A1) **CommentSize** : Cible les commentaires qui font plus de 6 lignes et/ou dont les lignes sont de longueur supérieure à 80 caractères.
  - (A2) **CommentRequired** : Détermine le nombre de commentaire manquant ou inversement, qui ne devrait pas être présent.
    Ci-dessous, un tableau récapitulatif :

Type de commentaire	Requis
serial version UID	Non
Commenter les Enum	Oui
Commenter les méthodes protected	Oui
Commenter les méthodes public	Oui
Commenter les champs	Oui
Commenter les Class	Oui

  - (A3) **CommentDefaultAccessModifier** : Renvoie le nombre de commentaire(s) ne respectant pas la règle suivante : "les méthode, champs et classe imbriquée qui ont un modificateur d'accès par défaut doivent être documenter".
- **Controversial** : Ensemble des règles qui déterminent les structures considérées comme controversées.
  - (B1) **DataflowAnomalyAnalysis** : Détection d'anomalies concernant les variables locales, les anomalies détectées sont les suivantes : variable non initialisée, variable redéfinie immédiatement après son initialisation et définition inutilisée d'une variable.
  - (B2) **OnlyOneReturn** : Détection de plusieurs "return" dans une méthode (par convention, le point de retour d'une méthode doit être la dernière instruction de la méthode).
  - (B3) **CallSuperInConstructor** : Détection de l'absence de l'instruction "super()" au sein du constructeur d'une classe héritante.
  - (B4) **DefaultPackage** : Détection d'absence de modificateurs de visibilité lors d'une déclaration (si aucun modificateurs de visibilité n'est précisé lors de la déclaration d'un objet (méthode ou variable) alors cet objet n'est accessible que par les classes appartenant au même package de la classe contenant la déclaration de l'objet concerné).

2. `https://plugins.jetbrains.com/plugin/?id=1137`

    - (B5) **NullAssignment** : Détection d'une réassignation à "null" d'une variable local.
    - (B6) **AtLeastOneConstructor** : Détection d'une classe sans définition d'au moins un constructeur.
    - (B7) **UseConcurrentHashMap** : Détecte l'utilisation d'une HashMap dans un cadre multi-threadé (Java préconise l'utilisation d'HashTables qui évitent les problèmes de blocage en programmation concurrentielle).
    - (B8) **AvoidFinalLocalVariable** : Détection d'une variable local final.
- **Naming** : Ensemble des règles de bonnes pratiques liées aux noms et aux identifiants.
    - (C1) **ShortVariable** : Détection d'un nom de variable jugé trop court que pour être suffisamment explicit.
    - (C2) **AvoidFieldNameMatchingTypeName** : Détection d'une méthode/variable partant de le même nom que la class dans laquelle elle se trouve.
    - (C3) **ShortClassName** : Détection d'un nom de class jugé trop court que pour être suffisamment explicit.
    - (C4) **AbstractNaming** : Détection d'une abstract class dont le nom ne commence pas par "Abstract" (convention).
    - (C5) **LongVariable** : Détection d'un nom de variable jugé trop long que pour garder le code facilement lisible et compréhensible.
- **Optimizations** : Ensemble des règles concernant les optimisations générales des bonnes pratiques
    - (D1) **MethodeArgumentCouldBeFinal** : Détections des méthodes qui ne modifie pas la valeur de leur paramètres et dont les paramètres n'ont pas été déclarés final.
    - (D2) **LocalVariableCouldBeFinal** : Détection des variables locales qui ne sont jamais redéfinis.
    - (D3) **AvoidInstantiatingObjectsInLoops** : Détection d'objets instanciés au sein d'une boucle et qui pourrait l'être en dehors de la boucle.
    - (D4) **PrematureDeclaration** : Détection d'une variable définis bien avant le moment ou on l'utilise, cette définition peut être faite au moment ou on a besoins de la variable en question.
    - (D5) **RedundantFieldInitializer** : Détection d'un champ explicitement instancié par défaut (si on ne donne aucune information lors d'une instanciation, les valeurs pas défauts sont automatiquement utilisées).
    - (D6) **UseStringBufferForstringAppends** : Détection de plusieurs concaténation successives sur un même sting (cette façon a un impact sur les performance, en utilisant correctement un StringBuffer cette impacte peut être significativement réduit).
- **Sunsecure** : Ensemble des règles visant à vérifié le respect des préconisations de sécurité.
    - (E1) **ArrayIsStoredDirectly** : Détection d'un constructeur ou d'une méthode recevant un ArrayList en paramètre et qui est directement utiliser (Sun préconise de cloner l'ArrayList afin de prévenir toute modification inopiné de l'ArrayList en question).

3.2.2 Comparaison des résultats

Le tableau 3.2 a permis de mettre en avant une augmentation particulièrement marquée en ce qui concerne le manque de commentaire, le nombre de variable dont le nom est jugé trop court ainsi que le nombre de variables/arguments pouvant être final. Bien que nous soyons conscient de l'impact que cela peut avoir sur la compréhension du code, le facteur temps ne nous aura pas permis de régler chaque problème relevé ici. Nous noterons tout de même qu'une rapide analyse à l'aide de `InCode Helium` n'aura relevé aucun anti-pattern ce qui nous conforte dans l'idée d'un code qui, bien que pouvant être mieux documenté et plus abouti, reste le fruit d'un travail réfléchis et dont la structure reste propre.

Bad Smell	**Avant intégration**	**Après intégration**
(A1) **CommentSize**	58	82
(A2) **CommentRequired**	74	262
(A3) **CommentDefaultAccessModifier**	3	4
(B1) **DataflowAnomalyAnalysis**	56	108
(B2) **OnlyOneReturn**	29	61
(B3) **CallSuperInConstructor**	6	30
(B4) **DefaultPackage**	3	4
(B5) **NullAssignment**	1	3
(B6) **AtLeastOneConstructor**	12	22
(B7) **UseConcurrentHashMap**	2	2
(B8) **AvoidFinalLocalVariable**	1	2
(C1) **ShortVariable**	112	201
(C2) **AvoidFieldNameMatchingTypeName**	1	4
(C3) **ShortClassName**	6	7
(C4) **AbstractNaming**	5	8
(C5) **LongVariable**	9	16
(D1) **MethodeArgumentCouldBeFinal**	212	317
(D2) **LocalVariableCouldBeFinal**	195	289
(D3) **AvoidInstantiatingObjectsInLoops**	6	10
(D4) **PrematureDeclaration**	1	2
(D5) **RedundantFieldInitializer**	1	12
(D6) **UseStringBufferForstringAppends**	1	5
(E1) **ArrayIsStoredDirectly**	1	1

TABLE 3.2 – Comparaison des Bad Smells avant et après intégration des extensions.

3.3 Conclusion

L'analyse des métriques à mis en avant le fait que la proportion de ligne de code pour les tests a augmenté. C'est un bon indicateur du fait que l'on a testé les fonctionnalités ajouté. L'analyse des "Bad Smell" à mis en avant le fait que notre code peut encore être amélioré pour en augmenter la qualité.

4 Conclusion

Ce projet nous a permis d'utiliser les notions vues au cours de Software Evolution de façon concrète. D'autre part, nous avons aussi appris à utiliser la fonctionalité de "pull-request" de Github. Finalement, il a été intéressant de fusionner des fonctionnalités implémentées séparemment et pour lesquelles seules quelques instructions avaient été brièvement discutées.

4.1 Note

Par manque de temps, le code n'a pas été amélioré pendant l'analyse d'après intégration. De ce fait, les résultats sont cohérent avec les notions vues au cours (i.e dégradation de la qualité du code).