Master 1 ALMA

Projet de Vérification et tests

EMF Project

Étudiants :
Coraline MARIE et Vincent RAVENEAU

Intervenant :
Gerson SUNYÉ

16 décembre 2013

Sommaire

1 Introduction

La création d'un logiciel, quel qu'il soit, n'est jamais sûre. Le code source peut contenir des fautes, des bugs ou des défauts, sans même que le programmeur ne les remarque. Fort heureusement, il existe aujourd'hui des méthodes efficaces pour contrer ces failles, et renforcer la fiabilité du code source, comme par exemple les tests unitaires.

Le module *Vérification et Tests* que nous avons étudié en Master Alma à l'Université de Nantes, nous a apporté un ensemble de techniques utilisables sur la majorité des langages informatique. Pour parfaire cet apprentissage, nous avons travaillé sur la restructuration et l'amélioration de l'EMF (Eclipse Modeling Framework), un logiciel libre développé pour et par Eclipse.

Ce rapport présente donc une application directe de ce que nous a appris le module Vérification et tests, au travers de deux grandes étapes de travail. Pour la première, nous avons tout d'abord dû restructurer une partie du code source de l'EMF en un projet Maven. Puis nous avons dû évaluer les tests déjà présent dans ce code. Pour la seconde étape nous avons dû améliorer autant que possible la qualité de la classe URI, afin d'augmenter sa fiabilité.

2 Restructuration et évaluation

2.1 Réorganisation avec Maven

L'un des objectifs pédagogiques du projet de vérification et tests, était de nous apprendre à utiliser un nouvel outil : Maven.

2.1.1 Qu'est ce que Maven ?

Maven (ou Apache Maven) est un logiciel libre, qui permet la gestion et la production automatique de projets liés au langage de programmation Java. Il s'agit d'un outil, qui aide à la conception de logiciels à partir du code source, en optimisant les tâches et en garantissant le bon ordre de fabrication.

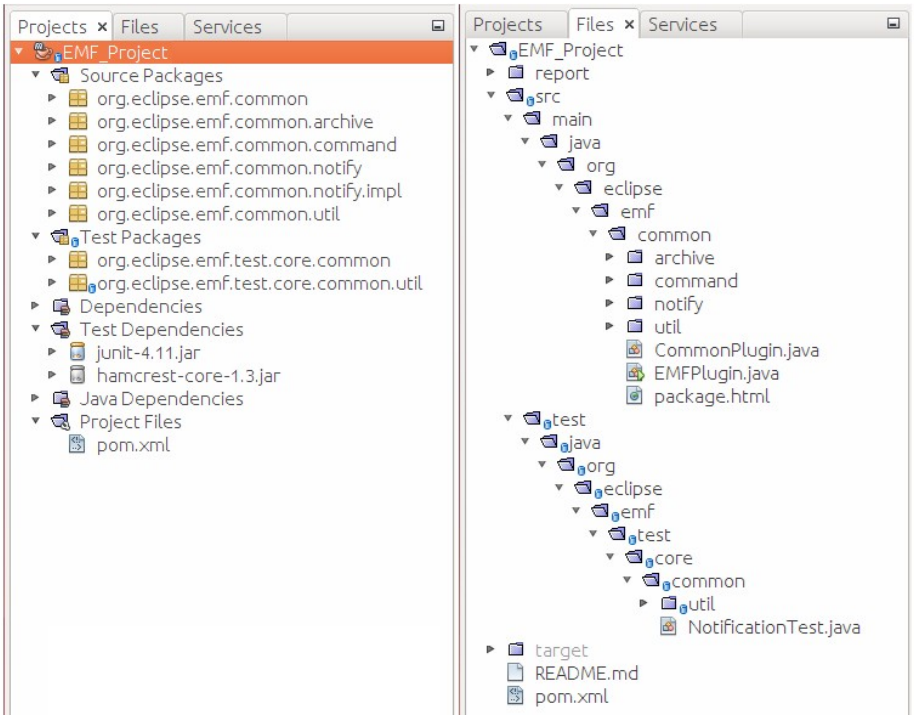
2.1.2 Conversion du code original

Le code source original du projet EMF est disponible sur gitHub. Il est donc téléchargeable en archive, et modifiable par n'importe qui. Cependant, l'archive que nous avons récupérée fournit un code source conditionné pour l'environnement de développement Eclipse, et cet IDE nous a été fortement déconseillé.

Ainsi, pour répondre à la demande du cahier des charges de ne pas utiliser Eclipse, nous avons choisi un nouvel IDE compatible avec Maven pour travailler. Ce choix s'est porté sur l'IDE NetBeans, qui permet de supporter plusieurs langages, et possédant un plugin de gestion de Maven. Cependant, le code source original devait être au préalable adapté, avant de pouvoir le travailler.

Afin d'optimiser notre travail, nous n'avons pas converti la totalité de l'archive téléchargée, mais juste la partie du code intéressante pour le projet. Malgré la facilité de conversion de l'architecture, cette tâche fût relativement fastidieuse. Certaines erreurs de package et de plugin sont apparues, ce qui nous a pris du temps à résoudre. De plus, tous les tests du projet étant mis dans un unique dossier, il nous a fallu identifier ceux qui étaient liés au code sur lequel nous avions à travailler.

Voici l'arborescence obtenue après la conversion du code source original en projet Netbeans/Maven :



2.1.3 Bugs et résolution

Pour gérer les bibliothèques et les dépendances d'un projet, Maven utilise un fichier `pom.xml`, qui décrit l'ensemble des besoins de l'application. Lors de la conversion du code source initial vers notre projet Maven, plusieurs erreurs liées à ce fichier sont apparues, car il manquait des dépendances :

- JUnit : framework de test unitaire.
- Core Runtime : framework (de Eclipse RCP).
- EMF ecore et EMF ecore xmi.

2.2 Passage des tests à JUnit 4.11

Il a d'abord été nécessaire de mettre les tests existants à jour, afin qu'ils puissent être exécutés par la version 4.11 de JUnit. En effet, ceux ci étaient basés sur une version obsolète de JUnit. Cette partie n'a pas demandé énormément de travail, étant donné qu'elle a principalement consisté à supprimer les inclusions de l'ancienne version de JUnit afin de les remplacer par les récentes. Il a également été nécessaire de rajouter l'annotation `@Test` avant chaque méthode de test unitaire, afin que l'appel à JUnit puisse les identifier clairement.

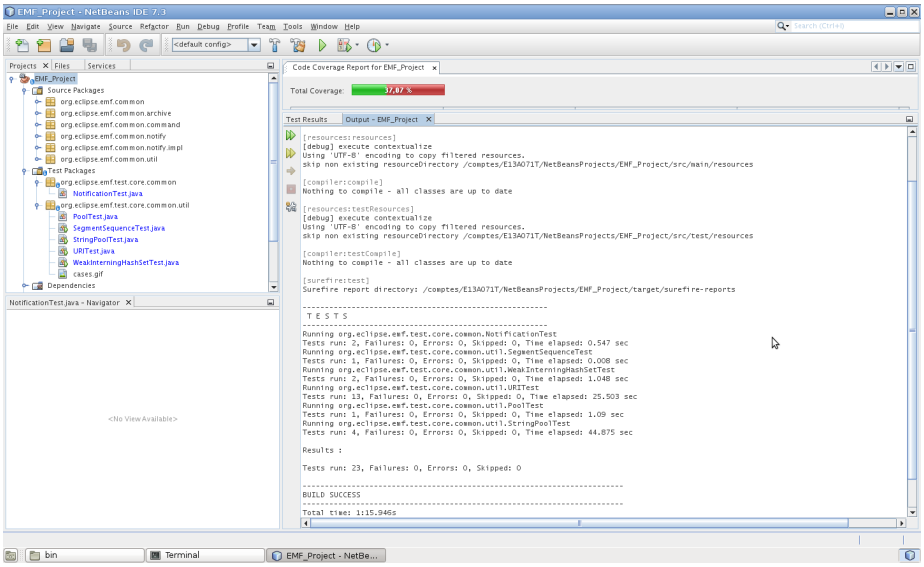
2.3 Evaluation de la qualité des tests

Afin d'évaluer la qualité des tests présents, nous avons procédé en trois étapes :

2.3.1 Exécution des tests

Nous avons tout d'abord exécuté les tests, afin de voir s'ils échouaient ou réussissaient, ainsi que leur nombre.

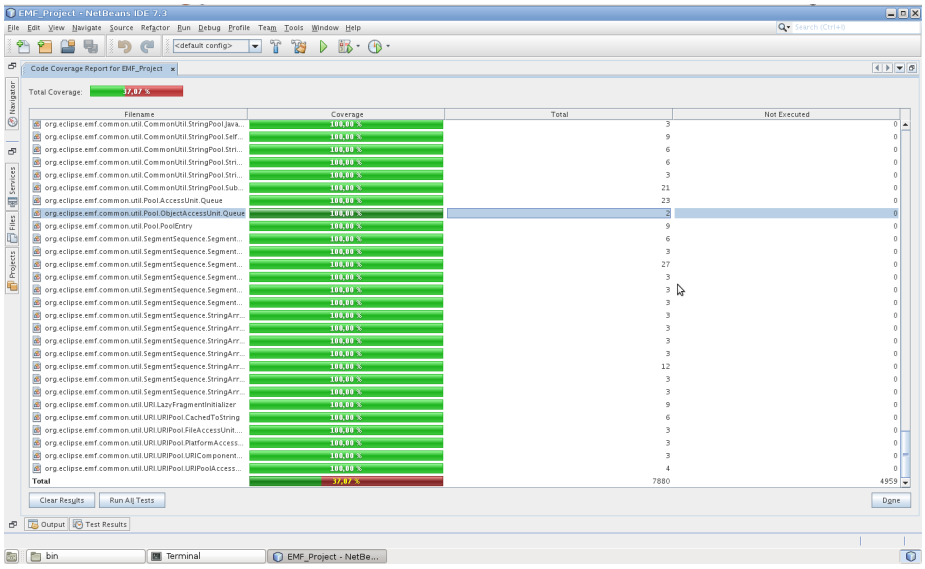
Comme le montre l'image suivante, les 23 tests exécutés sont tous positifs, ce qui est une bonne chose.



2.3.2 Couverture du code

Nous avons ensuite utilisé Cobertura, un outils de couverture de code permettant de mettre en évidence la proportion du code effectivement exécutée lors de l'appel des tests unitaires.

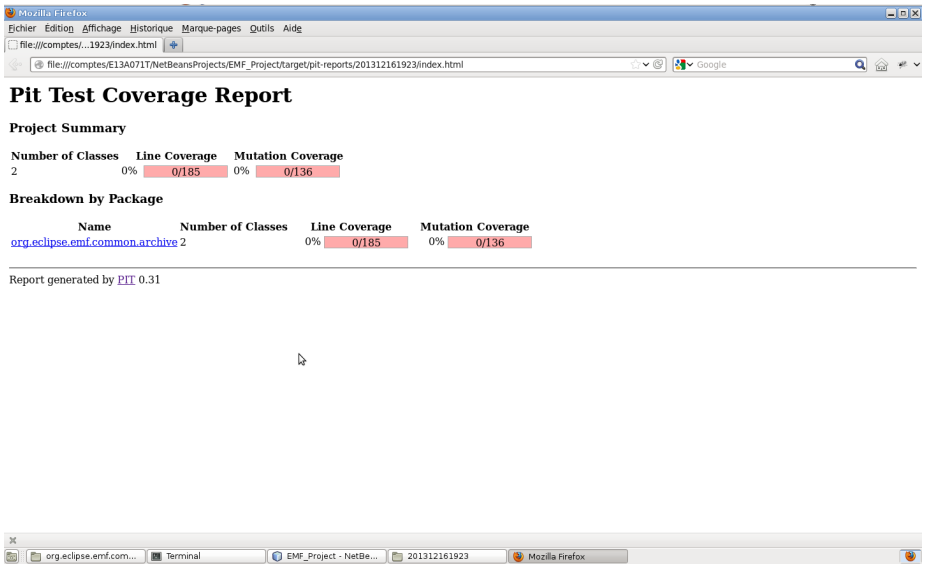
Comme le montre l'image suivante, uniquement 37% du code est testé. Même si ce pourcentage est faible, il est intéressant de remarquer que la majorité des classes sont soit complètement couvertes, soit pas du tout (très peu de classes ont une couverture partielle).



2.3.3 Analyse de mutation

Nous avons enfin utilisé Pit, un outils d'analyse de mutation. Il permet de générer des mutants afin de tester si de légères modifications du code permettent de tromper la suite de tests, ou s'ils sont détectés.

Malheureusement, très peu de ces mutants sont détectés, et la majorité des rapports d'exécution de Pit sont similaires à celui présent sur l'image suivante.



2.4 Conclusion sur la partie 1

Suite à ces observations, on peut conclure que la suite de tests actuellement présente sur le projet n'est ni exhaustive (elle est même très loin de l'être au vu

des résultats de couverture du code), ni vraiment efficace, comme le montrent les résultats de l'analyse de mutation. Il est donc nécessaire d'améliorer celle-ci, d'une part en étendant la couverture effectuée, et d'autre part en implémentant plus de cas de test afin de mieux identifier les erreurs induites par les mutants.

3 Amélioration de URI

Il nous était ensuite demander de travailler sur une unique classe, la classe URI du package `org/eclipse/emf/common/util`, afin d'améliorer sa qualité, ainsi que celle de ses tests. Malheureusement, nous n'avons pas pu effectuer cette partie, faute de temps.

3.1 La classe URI

Après avoir terminé la première partie du projet, c'est à dire réorganiser le dossier pour Maven et réparer les erreurs de compatibilités, nous avons chercher les défauts de conceptions de la classe URI. Malheureusement nous avons manqué de temps pour mener à bien cette étape, et nous n'avons pas trouvé de défaut pertinent pour la suite de notre travail. Cette classe n'a donc pas été modifiée.

4 Conclusion

Bien que nous n'ayons pas totalement terminé ce travail, le projet de Vérification et Tests reste un projet d'apprentissage. Nous avons appris de nouvelles notions, comme l'utilisation de Maven et de son système d'architecture. Nous avons également amélioré nos compétences en informatique en travaillant sur le code d'autres programmeurs, et en le retravaillant.

Ce projet nous a permis d'utiliser nos nouvelles connaissances étudiées en *Vérification et Tests*, mais pas seulement. Nous nous sommes servis pour ce projet d'autres notions vues dans d'autres modules d'enseignement, comme par exemple *Concepts et Outils de Développement*, ou *Génie Logiciel*.