

Projet Test et Vérification : **Test URI**

Sarra BOUTAHLIL
Alexis DONNART

Décembre 2016

Résumé

Dans le cadre du cours de Test et Vérification de Master 1 ALMA 2016-2017 dispensé à l'Université de Nantes par S. Gerson, il nous a été demandé d'évaluer et d'améliorer la qualité de la classe URI, qui fait partie du projet EMF (Eclipse Modeling Framework), dans le langage Java.

Table des matières

1	Analyse statique	3
1.1	Erreurs de syntaxe	3
1.2	Bugs trouvés	3
1.3	Évolutions à prévoir	3
2	Tests	4
2.1	Tests unitaires	4
2.2	Analyse des tests	4
3	Refactoring	5
4	Conclusion	6

1 Analyse statique

Première étape du test de la classe URI : l'analyse statique. On va s'attacher à analyser la syntaxe et les erreurs éventuelles que l'on pourrait trouver sans exécuter le code.

1.1 Erreurs de syntaxe

Nous utilisons dans un premier temps l'outil *PMD* pour analyser la syntaxe et les bonnes pratiques du code.

De nombreuses corrections ont été apporté grâce à l'analyse de PMD.

- **Parenthèses inutiles** mauvaise pratique la plus commune trouvé dans la classe URI, facilement corrigé en supprimant les parenthèses.
- **Blocs vides** de nombreux blocs étaient vides : if, catch, ... Nous avons simplement supprimé les blocs qui pouvaient l'être, et ajouté des logs d'erreur dans ceux qui ne pouvaient pas l'être.

En conclusion, des mauvaises pratiques facilement corrigés. Nous réutiliserons cet outil tout au lieu du test de la classe URI pour nous assurer de toujours respecter les bonnes pratiques de syntaxe.

1.2 Bugs trouvés

Nous avons ensuite utilisé *FindBugs* pour tenter de détecter des bugs dans la syntaxe. Nous en avons détecté et corrigé plus de 40. Parmi les plus courants :

- Utilisation de l'opérateur `==` plutôt que *equals* ce qui pouvait conduire à des erreurs de comparaison.
- Utilisation d'opérateurs *synchronized* sans réels utilités.
- Comparaison entre deux références du même objet : erreur de la part du développeur.

D'autres erreurs moins graves ont également été détecté grâce à cet outil, mais de nombreuses erreurs ont pu être corrigé avant même l'exécution.

1.3 Évolutions à prévoir

Dans l'analyse statique du code, nous avons pu approcher la classe URI et sa structure interne. Nous avons détecté de nombreux problèmes de conception.

La classe URI est une *classe dieu*, elle contrôle tout et englobe toutes les autres classes et fonctionnalités. Il faudra découper cette classe et en extraire toutes les classes internes (6).

De même, certaines des classes internes de URI possèdent elles aussi de nombreuses classes internes. On extraira également ces classes.

De très nombreuses fonctionnalités de la classe URI ne sont pas implémentées et ne font que renvoyer *true*, *this* ou *null*. Nous veillerons à tenir compte dans les tests de cette absence d'implémentation.

2 Tests

Dans un deuxième temps, nous allons tester la classe URI de façon dynamique puis nous évaluerons la qualité des tests.

2.1 Tests unitaires

Nous allons tenter de valider la classe URI à l'aide de test unitaires réalisés avec *JUnit*.

La classe URI n'est qu'en partie testable du fait de la visibilité et de la contrôlabilité de ses états. Toutes les classes publiques et testables s'élèvent au compte de 43. Parmi toutes ces fonctions, nous dénombrons plusieurs cas de tests, pour un total de 100 tests.

Outre ces cas de tests, certaines fonctions sous tests ne correspondent pas à la spécification ou plantent (*NullPointerException*) quelque soit les cas testés, nous avons donc décidé de les annotés *@Ignore*.

Nous avons également décidé d'annoter *@Ignore* les fonctions accessible mais utilisant des fichiers, l'utilisation de fichier étant difficilement contrôlable et testable.

2.2 Analyse des tests

L'écriture des jeux de test finie, nous allons maintenant s'assurer de leur qualité en regardant la couverture et la mutation.

Couverture du code

Pour assurer la couverture du code, nous utiliserons le plugin *EclEmma* de Eclipse. Celui ci se charge de vérifier que toutes les branches du code sont couvertes par nos tests. Nous obtenons un score de couverture 71.5% [1].






Element	Coverage	Covered Instruc	Missed Instructi	Total Instructions
▼ URI	 37,3 %	6 769	11 370	18 139
▼ src/main/java	 32,0 %	5 030	10 676	15 706
▸ org.eclipse.emf.common.util	 32,0 %	5 030	10 676	15 706
▼ src/test/java	 71,5 %	1 739	694	2 433
▸ org.eclipse.emf.common.util	 71,5 %	1 739	694	2 433

FIGURE 1 – Score de couverture de la suite de test de la classe URI

Ce score s'explique pour la majorité par l'absence de contrôlabilité sur tous les états du système. Notamment l'absence d'implémentation de nombreuses fonctions (qui renvoient toujours les même valeurs) ne nous permet pas de tester toutes les conditions, et ce quelque soit les valeurs d'entrées.

Analyse de mutation

Dans un deuxième temps, nous allons muter le code pour s'assurer de la qualité de nos tests. Nous utilisons pour cela l'outil *PIT* disponible sous Eclipse pour l'analyse de mutation.

Les tests de la classe URI obtiennent un score de 17%, qui s'explique encore une fois par l'absence de contrôlabilité de l'état interne des fonctions, mais également par l'absence ou l'incohérence de la spécification des fonctions.

3 Refactoring

Avec les jeux de tests fonctionnels et leur qualité assuré, nous pouvons effectuer des modifications pour assurer une bonne conception. Nous utiliserons pour cela les outils de refactoring, dont ceux mis à disposition par Eclipse. Chaque refactoring impliquera une réexécution des tests unitaires ainsi que des outils de test statique (PMD, FindBugs).

Nous avons identifié précédemment les principaux problèmes de la classe URI à savoir qu'elle possède trop de comportement et trop de classes internes. Pour résoudre ce problème, nous créerons un nouveau type indépendant à partir de chaque classe interne. Ainsi les classes *URIPool*, *Opaque*, *Fragment*, *Hierarchical* et *LazyFragmentInitializer* deviennent des classes à visibilité package indépendantes.

De ces classes, nous constatons encore la présence d'une *classe dieu* : *URIPool*. Nous allons donc en extraire toutes ses classes internes. Les classes *AccessUnitBase*, *StringAccessUnit*, *PlatformAccessUnit*, *FileAccessUnit*, *ComponentAccessUnit* et *CachedToString* deviennent également des classes indépendantes.

Nous constatons que dans toutes les classes implémentant *AccessUnit*, une classe interne *Queue*, presque similaire à toutes ces classes est présente. Nous allons donc extraire ces classes, les renommer spécifiquement du nom de la classe dont elle sont issues (*ComponentQueue*, ...) et en extraire les fonctions communes dans une super classe *Queue*.

En dernier, nous voulions extraire toutes les données *final static* de la classe URI vers une classe de données, mais ce processus perdait la référence de certaines données, donc nous avons laissé les données tels quels.

4 Conclusion

La classe URI a été développé de façon approximative : non finie, des erreurs, du code qui ne respecte pas l'usage, ... Grâce aux outils d'analyse statique et dynamique, nous avons pu valider le fonctionnement de certaines partie de cette classe, ou au contraire les invalider. Ces analyses ont finalement permis d'améliorer la conception de la classe à l'aide du Refactoring.