
TP COUVERTURE – PARTIE I + PARTIE II

Numéro de Référence
#UNIVPM001

(Document de 15 pages)

Résumé

Ce rapport présentera le résultat de nos travaux sur la première partie du TP Couverture. Il synthétisera sous forme d'un compte-rendu notre expérience de tests avec l'outil Eclemma.

Mots Clés

Couverture de code, Tests unitaires, Junit, Eclemma, TP2 Partie 1, FSIL.

Université Aix-Marseille
Centre Informatique
Luminy
13009 MARSEILLE CEDEX 13

SECTION DES REDACTEURS

Nom	Prénom	Contribution
MESTRALLET	Alexis	
RISCH	Philippe	

CONTACTS

Nom	Prénom	Email	Fonction
MESTRALLET	Alexis	Mestrallet.alexis@gmail.com	Etudiant FSI
RISCH	Philippe	Philippe.13009@yahoo.fr	Etudiant FSI

HISTORIQUE DES MODIFICATIONS

Modifications	Date	Version	Approbateur de la diffusion
Ajout du rapport de Test PARTIE II	12/10/2017	2.0	

TABLE DES MATIERES

1. PARTIE I : Introduction au couverture de code	4
1.1. Un Code Simple.....	4
1.2. Un Code Inconnu.....	4
1.2.1. Recommandation d'écriture de test unitaire	4
1.2.2. Utilisation d'eclEmma.....	4
1.2.3. Test du constructeur de la classe StringArray	4
1.2.4. Ajout de tests supplémentaires.....	4
1.2.5. Correction de la classe StringArray	5
PARTIE II : Rapport de Test Shop.java (Magasin.java).....	5
1.1. Introduction.....	5
1.1.1. Présentation	5
1.1.2. Spécifications.....	5
1.2. Tests	6
1.2.1. Conception de Test.....	6
1.2.2. Plan de Test	6
1.2.3. Cas de Test.....	7
1.2.3.1. Recherche Dichotomique	7
1.2.3.2. Article	7
1.2.3.3. Environnement de Test	7
1.2.3.4. Journal de Test	7
1.2.3.5. Incident de tests	8
1.2.3.6. Remarques.....	8
1.2.4. Packaging unit Test.....	8

1. PARTIE I : INTRODUCTION AU COUVERTURE DE CODE

Dans un premier temps nous avons consulté rapidement la documentation sur le site d'EclEmma, après quoi nous avons construit notre projet en récupérant les fichiers sources du TP précédent.

1.1. Un Code Simple

Nous avons donc lancé la série de tests de couverture de code sur le fichier PartialCoverTest avec EclEmma et obtenu les résultats suivants :

- 68.0% de couverture pour la classe PartialCover.java
- 94.5% de couverture pour la classe de test associée

Nous avons alors implémenté de nouveaux tests afin d'améliorer la couverture de code (i.e parcourir le maximum de branche sur le diagramme de flots résultant). Nous avons ainsi ajouté un test d'égalité ainsi qu'un test de supériorité. Ce qui nous a amené à une couverture de code plus complète : 97.0% pour les tests. En revanche, le taux de couverture de code de la classe n'a pas changé, ce qui nous permet d'affirmer qu'il était impossible de parcourir l'ensemble des branches du diagramme de flots résultant de ce programme.

1.2. Un Code Inconnu

Nous avons ensuite lancé une série de test de couverture de code sur un code inconnu (écrit par un tiers). Le but est de debugger le code source à l'aide de test unitaire et de test de couverture de code afin de nous assurer que nous testons bien le maximum de code.

1.2.1. Recommandation d'écriture de test unitaire

On remarque que certain des tests n'ont pas été nommé correctement. En effet les conventions de test veulent que le nommage des fonctions de tests soient explicites.

1.2.2. Utilisation d'eclEmma

Après utilisation d'EclEmma sur le code, nous obtenons les résultats suivant :

- Test : 100 %
- Classe : 52,5 %

On remarque que EclEmma se base bien sur les test unitaires. Cette outil ne permet pas de déterminer la couverture de code maximale possible pour une classe.

1.2.3. Test du constructeur de la classe StringArray

Après ajout de notre test nous obtenons une couverture de code de 90,1 %.

1.2.4. Ajout de tests supplémentaires

Nous avons ensuite ajouté de nouveaux tests afin de parcourir des instructions non couvertes par les tests précédents.

Ainsi nous nous sommes rendu compte qu'il restait une branche non parcourue, celle du tableau vide et que le programme était buggé aux endroits précédemment non couverts par les tests.

1.2.5. Correction de la classe StringArray

Une fois le code couvert à 100%, nous avons pu passer au debuggage du constructeur. Nous avons ainsi trouvé trois problèmes majeurs :

- Une fois le tableau trié, la recherche d'éléments dupliqués ne couvrait pas l'ensemble du tableau, donc tous les éléments dupliqués n'étaient pas trouvés.
- Le code levait une exception : `OutOfBoundsException`.
- Il manquait l'ajout du dernier élément du tableau list dans le tableau uniques.

PARTIE II : RAPPORT DE TEST SHOP.JAVA (MAGASIN.JAVA)

1.1. Introduction

1.1.1. Présentation

Ce rapport contient la documentation des tests effectués pour la classe `Article` ainsi que pour les différentes implémentations de la recherche dichotomique du TP2 de Fiabilité Logicielle. Les catégories de tests contenues dans ce document incluent les tests de fonctionnalité et de conformité, afin de déterminer si la méthode de recherche est valide ou non, i.e est fonctionnelle et répond aux spécifications.

1.1.2. Spécifications

Les différentes méthodes de recherche dichotomiques fournies permettent d'effectuer une recherche d'article de manière dichotomique dans un tableau d'article **stock**. Le tableau **stock** contient un nombre **n** d'articles triés respectivement par prix puis par numéro de nomenclature croissants. Les méthodes de recherche dichotomique fournies sont définies de la manière suivante :

```
public Boolean searchDicho(Article article);
```

Type de Test	Approche
Unitaire	<p>Les tests consistent en un ensemble d'assertions sur les méthodes.</p> <p>Les tests sont tous effectués en Boîte Blanche, exceptés les tests de notre implémentation de la recherche dichotomique, qui ont été réalisés en Boîte Noire.</p>

1.2. Tests

1.2.1. Conception de Test

Notre approche des Tests pour la classe Article à été classique. Nous avons commencé par prendre connaissance de la classe et de ses méthodes, celles-ci étant relativement simples, nous avons implémentés nos tests sur tous les cas possibles en tâchant de couvrir le maximum de code de la classe.

Pour les méthodes searchDicho de la classe Shop en revanche, nous avons décidé de commencer par simuler le fonctionnement des méthodes sur le papier, avec un tableau stock d'article simple pour commencer. Nous nous sommes rapidement rendu compte que certaines implémentations bouclaient à l'infini, aussi avons nous décidé de mettre en place une règle de limitation de temps pour les tests respectifs. Puis nous nous avons complexifié les articles de notre tableau stock afin de couvrir tous les cas de tests, même après avoir obtenu ce qui nous semble être la couverture de code maximale. Enfin, nous avons pu tester les méthodes que nous avons implémentées directement avec notre suite de Test.

1.2.2. Plan de Test

Ce plan de test couvre l'ensemble des méthodes de recherches dichotomiques proposées ainsi que celles que nous avons implémentées. De même, il couvre l'ensemble de la classe Article.

- Classes à tester :

Module	Classes	Code Testé
main.java	Magasin.java Article.java	Méthodes <code>searchDichoI(Article article)</code> Toute la classe

Nous avons décidé des fonctionnalités à tester d'après les spécifications et après une étude du code soumis.

1.2.3. Cas de Test

1.2.3.1. Recherche Dichotomique

Nous testons les fonctionnalités suivantes pour la recherche dichotomique :

- La méthode renvoie true si elle trouve l'article dans le stock.
- La méthode renvoie false si elle ne trouve pas l'article dans le stock.
- La méthode renvoie false si elle trouve un article du même prix mais qui n'a pas le même nom dans le stock.
- La méthode renvoie false si elle trouve un article du même prix et du même nom mais qui n'a pas le bon numéro dans le stock.
- La méthode renvoie false si elle trouve un article du même nom et avec le bon numéro mais qui n'a pas le même prix dans le stock.
- La méthode parvient à trouver l'article dans le si celui-ci est à l'indice 0 du tableau stock.
- La méthode parvient à trouver l'article dans le si celui-ci est à l'indice **n** du tableau stock.

1.2.3.2. Article

Nous testons les fonctionnalités suivantes pour la classe Article :

- La levée effective de l'exception **AssertionError** dans le constructeur pour tous les cas possiblement erronés.
- Le bon fonctionnement du constructeur.
- Les getters
- La méthode **isEqual(Article article)** sur tous les paramètres d'**Article**
- La méthode **smallerThan(Article article)** sur tous les paramètres d'**Article**

1.2.3.3. Environnement de Test

Nous avons effectué nos tests dans les environnements suivants,

- Linux Mint 18 Sarah
- Ubuntu 16.04

1.2.3.4. Journal de Test

Cf. Tableau 1 dans Liste des Tableaux.

1.2.3.5. Incident de tests

Nous avons testé les cinq différentes implémentations de la recherche dichotomique. Aucune d'entre elle ne passe tout les tests. En effet nous avons remarqué plusieurs problème dans leurs implémentations respectives, dont :

- La conditions d'arrêts
 - Boucles infinies
 - Variable found erroné : n'utilise pas toujours `isEqual(Article article)`
- Mauvaise initialisation des variables d'index du stock
 - `i != 0` ou `j != stock.length` -> parcours incomplet
- Comparaison de prix au lieu de la méthode `smallerThan`

Notre implémentation réunit finalement le code fonctionnel des cinq implémentations proposées.

1.2.3.6. Remarques

La taux de couverture de code de la classe `Article` ne peut être supérieur à 97.9 % bien que toute les méthodes soient couvertes à 100%. En effet, il semble impossible de couvrir l'ensemble des branches de l'assertion présente dans le constructeur. Nous avons remarqué qu'en la remplaçant par un if qui lève l'exception, le taux passe à 100%. Rien de plus de notable pour la classe `Article` et ses tests.

1.2.4. Packaging unit Test

Nous avons choisi dans notre convention d'adopter un packaging des tests en parallèle au code source, car d'après nous cette façon d'organiser à pour avantage de retrouver et exécuter l'ensemble des suites de test plus facilement. Aussi il nous parait plus clair de séparer les tests du code de production. Il s'agit de la convention adoptée par Maven.

Cependant ajouter les tests dans le même package que le code source permet de restreindre l'accès au package. En revanche ce type d'organisation peut s'avérer cauchemardesque lors de l'implémentation de projets conséquents.

BIBLIOGRAPHIE ET LIENS

<http://www.eclEmma.org>

ANNEXES




GLOSSAIRE
































INDEX

TABLES DES FIGURES

LISTE DES TABLEAUX

Tableau 1 : legende

Passed	
Failed	
ERROR	

Classe de Test	Méthode de Test	Résultat	Taux de Couverture de la méthode testée
ArticleTest	articleEmptyNameTest()		100 % (97.9%)
	articleInvalidTest()		
	articleNegativeNumberTest()		
	articleNegativePriceTest()		
	articleValideTest()		
	getNameTest()		100 %
	getNumberTest()		100 %
	getPriceTest()		100 %
	isEqualTest()		100 %
	smallerThanTest()		100 %
SearchDicho1Test	searchDicho1ElementInStock()		93.8 %
	searchDicho1ElementNotInStock()		
	searchDicho1ElementWrongName()		
	searchDicho1ElementWrongNumber()		
	searchDicho1ElementWrongPrice()		
	searchDicho1ElementMax()		
	searchDicho1ElementMin()		
SearchDicho2Test	searchDicho2ElementInStock()		100 %
	searchDicho2ElementNotInStock()		
	searchDicho2ElementWrongName()		
	searchDicho2ElementWrongNumber()		
	searchDicho2ElementWrongPrice()		
	searchDicho2ElementMax()		
	searchDicho2ElementMin()		
SearchDicho3Test	searchDicho3ElementInStock()		94.8 %
	searchDicho3ElementNotInStock()		
	searchDicho3ElementWrongName()		
	searchDicho3ElementWrongNumber()		
	searchDicho3ElementWrongPrice()		
	searchDicho3ElementMax()		
	searchDicho3ElementMin()		

SearchDicho4Test	searchDicho4ElementInStock()		100 %
	searchDicho4ElementNotInStock()		
	searchDicho4ElementWrongName()		
	searchDicho4ElementWrongNumber()		
	searchDicho4ElementWrongPrice()		
	searchDicho4ElementMax()		
	searchDicho4ElementMin()		
SearchDicho5Test	searchDicho5ElementInStock()		100 %
	searchDicho5ElementNotInStock()		
	searchDicho5ElementWrongName()		
	searchDicho5ElementWrongNumber()		
	searchDicho5ElementWrongPrice()		
	searchDicho5ElementMax()		
	searchDicho5ElementMin()		
SearchDichoEtuRecTest	searchDichoEtuRecTestElementInStock()		100 %
	searchDichoEtuRecTestElementNotInStock()		
	searchDichoEtuRecTestElementWrongName()		
	searchDichoEtuRecTestElementWrongNumber()		
	searchDichoEtuRecTestElementWrongPrice()		
	searchDichoEtuRecTestElementMax()		
	searchDichoEtuRecTestElementMin()		
SearchDichoEtuTest	searchDichoEtuTestElementInStock()		100 %
	searchDichoEtuTestElementNotInStock()		
	searchDichoEtuTestElementWrongName()		
	searchDichoEtuTestElementWrongNumber()		
	searchDichoEtuTestElementWrongPrice()		
	searchDichoEtuTestElementMax()		
	searchDichoEtuTestElementMin()		