UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie électrique et de génie informatique

GÉNÉRATION DE SQUELETTES DES CONTRATS DE CLASSES ET DES TESTS UNITAIRES EN JAVA

Mémoire de maîtrise
Spécialité : génie électrique

Cheick Ismael MAIGA

Jury : Ruben GONZALEZ-RUBIO (directeur)
Wael SULEIMAN (rapporteur)
Stefan BRUDA

Sherbrooke (Québec) Canada Juillet 2016

À mon père, ma mère, et à mon oncle Yassiah Bissiri, merci pour les encouragements et les différents soutiens.

RÉSUMÉ

Le logiciel est devenu omniprésent dans nos vies de sorte qu'on le retrouve dans plusieurs domaines de la vie courante. Cependant cette omniprésence, n'est pas sans conséquences. Les bogues de logiciel peuvent causer de vrais désastres, économiques, écologiques voire sanitaires. Vu la forte omniprésente du logiciel dans nos vies, le fonctionnement de nos sociétés dépend fortement de sa qualité.

La programmation par contrat a pour but de produire des logiciels fiables, c'est-à-dire corrects et robustes. En effet, ce paradigme de programmation vise à introduire des assertions qui sont des spécifications de services. Ces spécifications représentent une forme de contrat. Les contrats définissent les responsabilités entre le client et le fournisseur. Le respect des contrats permet de garantir que le logiciel ne fait ni plus ni moins que ce que l'on attend de lui qu'il fasse.

Le test unitaire est un test qui permet de s'assurer du bon fonctionnement d'une partie précise d'un logiciel. C'est un test dont la vérification se fait en exécutant une petite unité de code. En somme, un test unitaire est un code qui exécute de manière indirecte le code d'une classe pour vérifier que le code fonctionne bien.

L'outil Génération de Squelettes des Contrats de classes et des tests unitaires (GACTUS) permet la génération automatique de squelettes de contrats de classes et celles des classes des tests unitaires d'un projet Java. La génération automatique du code source permet d'obtenir un code uniforme. GACTUS est un plug-in pour l'environnement de développement Eclipse écrit en Java. L'objectif principal de GACTUS est de faciliter la réalisation de logiciel de qualité grâce à la génération automatique des squelettes de contrats de classe et celui des tests unitaires et aussi d'accroître la productivité des développeurs. Pour faciliter son utilisation, GACTUS dispose d'une interface graphique permettant de guider l'utilisateur.

Mots-clés : génie logiciel, qualité du logiciel, test unitaire, contrat de classe, Java, plugin

REMERCIEMENTS

J'adresse tout d'abord mes remerciements aux membres de ma famille qui m'ont admirablement aidé tout au long de mes études ; ces quelques lignes sont insuffisantes pour exprimer ma reconnaissance pour tous les sacrifices consentis et soutenus durant tout ce parcours universitaire. Puisse Dieu leurs accorder longue vie afin que je puisse à mon tour les combler et leur témoigner de tout mon amour au nom des liens solides qui nous unissent.

Je tiens à remercier mon directeur, pour les remarques et ses conseils constructifs qu'il ma prodigué tout au long de mon projet. Une telle expérience n'aurait pu avoir lieu, ni avoir de réussite, tant sur le contenu, que sur la forme, la présentation du projet et le rapport si Monsieur GONZALEZ-RUBIO, ne m'avait pas encadré, et proposé le sujet. Merci du fond du coeur.

Que les membres du jury trouvent ici l'expression de toute ma reconnaissance pour avoir accepté de juger ce travail.

Je ne saurai terminer sans remercier ma famille, la famille SARAMBE, BELEM, COMPAORE, BISSIRI pour l'aide et les soutiens multiformes dans la réalisation de ce projet.

TABLE DES MATIÈRES

1 INTRODUCTION **1**
- 1.1 Mise en contexte et problématique . . . 1
- 1.2 Définition du projet de recherche . . . 2
- 1.3 Objectifs du projet . . . 3
- 1.4 Organisation du mémoire . . . 3

2 DÉFINITION DE LA QUALITÉ DU LOGICIEL **5**

3 ÉTAT DE L'ART **11**
- 3.1 Les tests de logiciel . . . 11
  - 3.1.1 Test unitaire . . . 13
- 3.2 Programmation par contrat (Design by contrat) . . . 17
  - 3.2.1 Programmation par contrat en Java avec C4J (Contrat for Java) . . . 20
- 3.3 Analyse du code Java . . . 23
  - 3.3.1 Java Development Tools (JDT) . . . 26
- 3.4 Résumé de l'état de l'art . . . 28

4 LE LOGICIEL GACTUS **31**
- 4.1 Description du plug-in GACTUS . . . 31
- 4.2 Analyse des besoins . . . 32
  - 4.2.1 Contraintes générales . . . 32
  - 4.2.2 Analyse des contraintes particulières d'une classe de test imposées par JUnit 4 . . . 33
  - 4.2.3 Analyse des contraintes imposées par C4J . . . 34
- 4.3 Conception générale du plug-in GACTUS . . . 35
  - 4.3.1 Les commandes sous Eclipse RCP . . . 35
  - 4.3.2 Description du scénario de génération de code source . . . 39
  - 4.3.3 Description de l'architecture du plug-in (GACTUS) . . . 41
- 4.4 Conception des fonctionnalités de GACTUS . . . 42
  - 4.4.1 Analyse d'un projet Java . . . 42
  - 4.4.2 Conception de la fonctionnalité de génération de squelette de code source . . . 46
- 4.5 Description des tests unitaires . . . 52
  - 4.5.1 Description du fonctionnement de l'exécution d'une classe de test sous Eclipse RCP . . . 52
  - 4.5.2 Solution pour exécuter les tests unitaires de Plug-in Eclipse . . . 52
  - 4.5.3 Description des tests unitaires du code source de GACTUS . . . 54
- 4.6 Description des contrats de classes . . . 54
- 4.7 Analyse des métriques du code source du logiciel GACTUS . . . 55
- 4.8 Utilisation de GACTUS . . . 55

5 DISCUSSION **59**
5.1 Les bonnes décisions 59
5.2 Discussion sur la programmation par contrat 59
5.3 Discussion : Est-ce qu'il faut tester unitairement les contrats de classe . . 61

6 CONCLUSION **63**
6.1 Récapitulatif 63
6.2 Contribution 63
6.3 Travaux futurs 64

A Installation du plug-in GACTUS **67**
A.1 Comment installer GACTUS 67
A.1.1 Installation manuelle 67
A.1.2 Installation à partir de Eclipse Marketplace 67

B Code source de squelette de classes généré avec GACTUS **71**
B.1 Classe Personne 71

C Code source d'un handler du plugin GACTUS **77**

D Code source de quelques classes de tests unitaires du plug-in GACTUS **79**

E Code source d'un contrat de classe du plug-in GACTUS **85**

LISTE DES RÉFÉRENCES **91**

LISTE DES FIGURES

2.1 Modèle de qualité de McCall 7
2.2 Modèle de Boehm 8

3.1 Diagramme de classe du framework JUnit 16
3.2 Interface d'exécution d'une classe de test avec JUnit 17
3.3 Exemple de structure d'un projet C4J 21
3.4 Schéma d'une chaîne de compilation classique 24
3.5 Schéma de compilation multi-source multi-cible 25
3.6 Java modèle vue 28

4.1 Fenêtre du menu « dependencies » du fichier « plugin.xml ». 32
4.2 Menu extension du fichier plugin.xml 36
4.3 Fenêtre d'ajout d'une nouvelle extension 37
4.4 Description d'une commande 38
4.5 Scenario du processus de génération 40
4.6 Diagramme de classe du scénario de génération 40
4.7 Architecture du plug-in GACTUS 41
4.8 Diagramme de classes du package « com.udes.gactus.model.core » 42
4.9 Description parseur et visiteur de code source 46
4.10 Diagramme de classe de la fonctionnalité de génération 47
4.11 Diagramme de classe de la vue 51
4.12 Couverture du code des classses de tests 55
4.13 Métriques du plug-in GACTUS 56
4.14 Vue de GACTUS et de ses sous-menus 57
4.15 Boîte de dialogue de confirmation de la génération 57

A.1 GACTUS : fenêtre d'installation 68
A.2 Boîte de dialogue « Add repository » 69
A.3 GACTUS : installation suite 69
A.4 GACTUS : installation en cours 70
A.5 Boîte de dialogue « Detail software installation » 70

LISTE DES TABLEAUX

3.1 Obligations et bénéfices du client et du fournisseur d'un service 18
3.2 Tableau représentatif des classes du Modèle Java 28

LISTE DES ACRONYMES

Acronyme	Définition
AGL	Atelier de Génie Logiciel
AST	Abstract Syntax Tree
C4J	Contrat for Java
DBC	Design By Contract
GACTUS	Génération Automatique de Contrat de classe et Test unitaire Université de Sherbrooke
GOF	Gang of Four
IEEE	Institute of Electrical and Electronics Engineers
JDT	Java Development Tools
MVC	Modèl Vue Controleur
SWT	Standard Widget Toolkit
VKSI	Verein der Karlsruher Software-Ingenieure
XML	EXtensible Markup Language

CHAPITRE 1
INTRODUCTION

1.1 Mise en contexte et problématique

De nos jours, on ne peut pas se passer du logiciel. En effet, on retrouve le logiciel dans plusieurs domaines de la vie courante. C'est le cas entre autres des domaines du transport, du commerce, des communications, de la médecine, des finances ou des loisirs. Ce qui fait que nous sommes en perpétuel contact avec des lignes de codes. Conduire une automobile Mercedes-Benz série classe S, c'est entrer en contact avec plus de 20 millions de lignes de code provenant uniquement de la voiture [10]. Le logiciel est donc *omniprésent* dans nos vies.

Cette omniprésence n'est pas sans conséquences. En effet, les erreurs logicielles communément appelées bogue peuvent causer de vrais désastres, économiques ou sanitaires. Un exemple de bogue parmi les plus célèbres est l'explosion de la première fusée Ariane 5 en 1996, qui a couté plus d'un demi-milliard de dollars américains. La cause du crash : « la panne des systèmes informatiques de bord ». La cause réelle de cette panne étant une conversion d'un nombre flottant codé sur 64 bits en une valeur entière sur 16 bits, ce qui a provoqué une exception, car le nombre n'était pas représentable sur 16 bits [19]. Le logiciel étant omniprésent dans nos vies, le fonctionnement de nos sociétés dépend fortement de son bon fonctionnement donc, de sa qualité. Au vu des désastres que peut causer une panne de logiciel, une question s'impose à nous : en quoi consiste la qualité du logiciel ?

Dans la littérature, plusieurs définitions ont été proposées afin d'appréhender la notion de qualité dans le logiciel. Bertrand Meyer, dans son ouvrage « Conception et programmation orientée objet » [25], définit la qualité du logiciel comme un compromis entre plusieurs objectifs. La résultante de la combinaison de plusieurs facteurs externes et internes. Les facteurs externes, perceptibles aux utilisateurs et clients, devraient être distingués des facteurs internes (exemples : lisibilité du code source, la réutilisabilité du code source, l'extensibilité) perceptibles aux concepteurs et développeurs. Pour Meyer, seuls les facteurs externes comptent, mais ils ne peuvent être obtenus qu'à l'aide des facteurs internes. Parmi les facteurs internes, les plus importants sont : la correction et la robustesse. « **La correction** est la capacité que possède un produit logiciel à mener à bien sa tâche, tel qu'elle a été définie par sa spécification. Ce facteur est le facteur de qualité essentiel, car si

un système ne fait pas ce qu'il est supposé faire, tout le reste (rapidité, interface utilisateur agréable ou autres) compte peu. **La robustesse** quant à elle, est la capacité qu'offrent des systèmes logiciels de réagir de manière appropriée en présence de conditions anormales. La robustesse complète la correction. En effet, la correction concerne le comportement d'un logiciel dans les cas qui sont conformes à ses spécifications ; la robustesse caractérise ce qui se passe en dehors de cette spécification » [25].

Depuis la création du génie logiciel, plusieurs méthodologies et outils ont été proposés en vue de construire des logiciels de qualité. Parmi ces méthodologies, il convient de mentionner *le test de logiciel* et *la programmation par contrat*. En Java, *JUnit* [5] est une référence pour faire des tests unitaires et *C4J* [7] permet de faire de la programmation par contrat.

1.2 Définition du projet de recherche

En se basant sur la définition donnée par Bertrant Meyer sur la qualité du logiciel, on peut retenir qu'un logiciel sera dit de qualité si ce dernier est robuste et correct. Ainsi donc, les développeurs devront tenir compte de ces deux facteurs lors du développement des logiciels. Mais comment peut-on vérifier si le logiciel est correct ou robuste ?

D'après G. Tremblay [30] la propriété qu'un logiciel soit correct ou non est une propriété qui est *relative*. En effet, on peut dire d'un logiciel qu'il est correct uniquement en faisant référence à la spécification qu'il doit satisfaire *donc relativement*. Si le comportement du composant logiciel est celui décrit par sa spécification, alors le logiciel est correct, autrement il est incorrect.

La programmation par contrat est une méthode de construction de logiciel qui conçoit les composantes d'un système de telle façon qu'elles coopèrent sur la base de contrats qui sont des spécifications définies de manière précise. Les contrats permettent de s'assurer de la correction et de la robustesse du logiciel. Les tests unitaires permettent de s'assurer du bon fonctionnement des différentes fonctionnalités du logiciel et de détecter de potentiels bogues donc, d'assurer de la réutilisabilité et de l'extensibilité. Si les tests unitaires ou la programmation par contrat sont réputés pour donner de très bons résultats dans le processus de création de logiciels, ces deux méthodologies et leurs outils sont moins utilisés surtout la programmation par contrat et pour la plupart du temps ces deux méthodologies sont utilisées distinctement. Et pour cause, leur utilisation demande un niveau de connaissances avancées en programmation de la part du développeur. Après une analyse de la littérature existante en la matière, nous avons remarqué qu'en combinant ces

deux concepts l'on pourrait accroître considérablement la qualité dans les logiciels. Une approche pratique a permis aussi d'en venir à la même conclusion. Il en ressort donc que la combinaison de ces méthodologies pourrait accroitre la qualité dans les logiciels. C'est à partir de cette observation que les questions de recherche suivantes ont été établies :

- *Comment réaliser un outil capable de générer des squelettes de classes de test unitaires et des squelettes de contrats de classes en utilisant JUnit et C4J ?*
- *Quel sera l'apport de ce nouvel outil quand il sera utilisé par les développeurs ?*
- *Comment vulgariser l'utilisation de cette nouvelle méthodologie ?*

1.3 Objectifs du projet

Pour ce projet, l'objectif global peut être défini comme suit : *Créer un logiciel permettant de générer automatiquement les squelettes des classes de tests unitaires et celui des contrats de classe afin d'améliorer la qualité des logiciels et accélérer la productivité des développeurs.*

L'issue de ce projet est de développer un logiciel sous forme de plug-in pour l'environnement de développement Eclipse. Ce plug-in permettra au développeur de gagner du temps pendant le développement du logiciel, car le développeur pourra générer les squelettes des tests unitaires et les squelettes des contrats de classe des différentes classes de son projet. Dans la réalisation du logiciel, il serait inclus des tests et des contrats afin de garantir la qualité du logiciel, bin entendus ces tests et contrats seront réalisés sans l'aide de l'outil.

En plus de cela, le logiciel qui sera produit sera de qualité dans la mesure où les tests unitaires permettront de s'assurer du bon fonctionnement du code source. Les contrats de classe quant à eux permettront au logiciel d'être robuste et correct donc fiable par la présence des spécifications introduites.

1.4 Organisation du mémoire

Dans ce mémoire, nous exposons les travaux qui furent réalisés pour l'élaboration d'un logiciel capable de générer automatiquement les squelettes des classes de tests unitaires et les squelettes des contrats de classe. Le mémoire commencé par une introduction qui situe le contexte, la problématique, puis la définition du projet de recherche ainsi que l'objectif du mémoire. Par la suite, des définitions sur la qualité du logiciel sont introduites. Ensuite, le chapitre de « l'État de l'art » est une synthèse des avancées recensées dans la littérature

qui ont été utilisées dans ce projet. À la suite du chapitre de l'État de l'art vient le chapitre « Le logiciel GACTUS » présentant les concepts théoriques ayant mené à la réalisation du logiciel. Le chapitre « Discussion » est une évaluation de certaines décisions prises en cours de projet. Le mémoire se termine par le chapitre « Conclusion ».

CHAPITRE 2

DÉFINITION DE LA QUALITÉ DU LOGICIEL

Ce chapitre a pour but de présenter l'un des concepts clés de ce mémoire qui est ***la qualité du logiciel***. Afin de faciliter la compréhension et la lecture du mémoire tout en offrant une idée globale du projet, il est important de donner une définition de la qualité du logiciel.

La définition de la qualité a toujours été sujet d'un vif débat. La raison pour laquelle le concept de qualité est si controversé est que l'on ne parvient pas à se mettre d'accord sur une signification. L'on se retrouve le plus souvent face à des normes.

Selon la norme ISO 9001 : 2000, la qualité est : *« Aptitude d'un ensemble de caractéristiques intrinsèques à satisfaire des exigences »*. Dans cette norme, la qualité peut se voir comme la satisfaction aux exigences d'un client. Mais si la qualité d'un produit se définit ainsi d'une manière globale qu'en est-il pour la qualité dans le logiciel ?

De nombreuses définitions et modèles ont été proposés pour définir la qualité du logiciel. Carter Jones [8] définit la qualité du logiciel selon trois bases :

- **La qualité fonctionnelle** : logiciel qui contient un faible taux de défauts et un haut niveau de satisfaction utilisateur. Le logiciel doit remplir toutes les exigences utilisateur et adhérer aux standards internationaux.

- **La qualité structurelle** : logiciel qui présente une robuste architecture et peut opérer comme un environnement multicouche sans panne ou performance dégradée. Le logiciel présente un faible niveau de complexité cyclomatique. La complexité cyclomatique (Cyclomatic complexity) est la mesure du nombre de chemins linéaires indépendants qu'il est possible d'emprunter dans une fonction ou méthode [23].

- **La qualité esthétique** : logiciel qui a de l'élégance, de l'ergonomie, avec des commandes faciles à utiliser, des interfaces et des écrans interactifs et des sorties bien présentées.

Stephen H. Kan définit la qualité du logiciel dans son ouvrage « Metrics and models in software quality engineering » comme la « conformité aux exigences » [20].

La norme ISO-9126 définit des caractéristiques et sous caractéristiques relatives à la qualité logicielle. Voici quelques caractéristiques :

- **Fonctionnalité** : le logiciel fonctionne selon les demandes du client, dans l'environnement du client.
- **Fiabilité** : capacité du logiciel à continuer de fonctionner correctement dans un contexte particulier (exemple : suite à une faute, perte de connexion imprévue, etc.)
- **Utilisabilité** : le logiciel est facile à utiliser et à apprendre.
- **Maintenabilité** : capacité à modifier facilement le logiciel (nouvelles fonctionnalités, réusinage[1], etc).
- **Testabilité** : c'est une sous caractéristique de la maintenabilité. C'est la capacité à tester facilement le logiciel. Cela permet de faire des ajouts au logiciel, de le modifier de façon sécuritaire et conséquemment, d'accélérer la maintenabilité. Ce point est particulièrement important pour les développeurs, car c'est ce qui leur permet de s'assurer que leur travail fonctionne comme demandé et sans bogue. Pour le client, c'est une façon de s'assurer que le produit livré correspond bien à ses attentes. De plus, les tests sont très utiles pour éviter une augmentation importante du coût des changements qu'il y aura à faire avec le temps.

Parmi les modèles proposés, on peut retenir celui de McCall et de Boehm. Nous donnerons une description brève des deux modèles.

En 1977 dans un rapport technique McCall introduit une définition hiérarchique des facteurs qui influent sur la qualité des logiciels [9]. Le modèle proposé par McCall est considéré comme étant le premier modèle qui a été publié. La figure 2.1[2] présente ce modèle de qualité. Chaque facteur de qualité sur le côté gauche de la figure représente un aspect de qualité qui n'est pas directement mesurable. Sur le côté droit sont représentées les propriétés mesurables qui peuvent être évaluées afin de quantifier la qualité en termes de facteurs.

Cela a permis d'identifier deux groupes de facteurs à savoir les facteurs internes et les facteurs externes. D'après Bertrand Meyer, tous les facteurs de qualité sont importants, mais ceux qui sont primordiaux sont :

[1]Le réusinage (refactoring) : consiste à apporter des améliorations à un programme dans le but de ralentir la dégradation due aux changements. Le but est d'améliorer la structure, réduire la complexité ou améliorer la lisibilité du code.

[2]Figure traduit en français du modèle initial

- Correction et robustesse : le terme fiabilité est utilisé pour désigner ses deux facteurs.
- Extensibilité et réutilisabilité : l'extensibilité est la facilité d'adaptation des produits logiciels aux changements de spécification et la réutilisabilité est la capacité des éléments logiciels à servir à la construction de plusieurs applications [25]. La modularité est employée pour désigner ses deux facteurs.

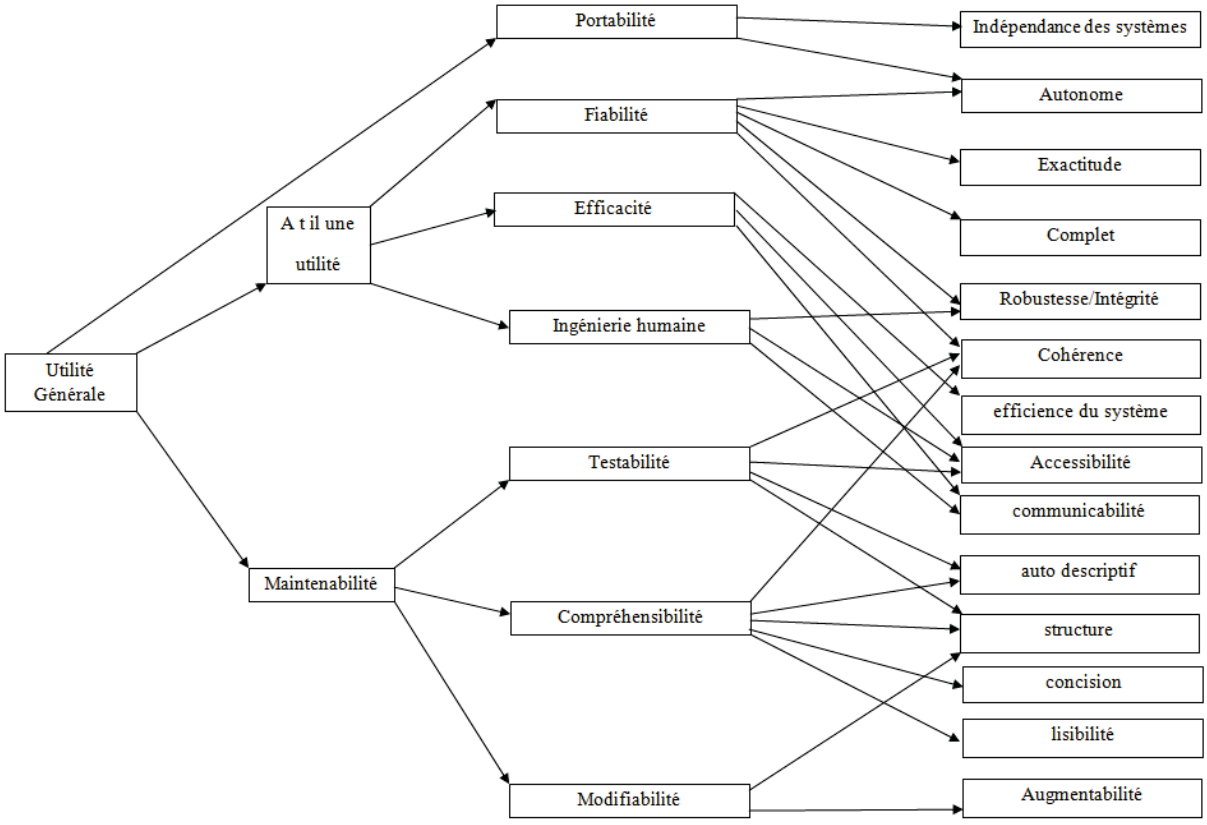

Figure 2.1 Modèle de qualité de McCall[3]

Le modèle de Boehm [6] [11] [13] et de ses collègues (voir figure 2.2[4]) définit qualitativement la qualité du logiciel comme un ensemble d'attribut et de mesures. C'est un modèle hiérarchique structuré autour de caractéristiques de haut niveau, de niveau intermédiaire et des caractéristiques primitives. Chacun des caractéristiques contribue au niveau de la qualité global. Pour Boehm et ses collègues, la caractéristique principale de la qualité est ce qu'ils définissent comme « l'utilité générale » du logiciel. Cette utilité représente les caractéristiques de haut niveau qui sont les exigences de haut niveau de base de l'utilisation réelle auxquelles l'évaluation de la qualité de logiciel pourrait être soumise. Les

[3]Source version originale : `https://msritse2012.files.wordpress.com/2013/01/mccalls-11-quality-factor-hierarchy.jpg`

[4]figure 2.2 est une traduction en français du modèle initial

caractéristiques de haut niveau portent sur trois questions principales que l'utilisateur du logiciel doit se poser :

- Est-ce utilitaire : comment (simple, fiable, efficace) pourrais je bien l'utiliser tel quel ?
- Maintenabilité : combien est-il facile à comprendre, modifier et tester de nouveau ?
- Portabilité : puis-je encore l'utiliser si je change d'environnement ?

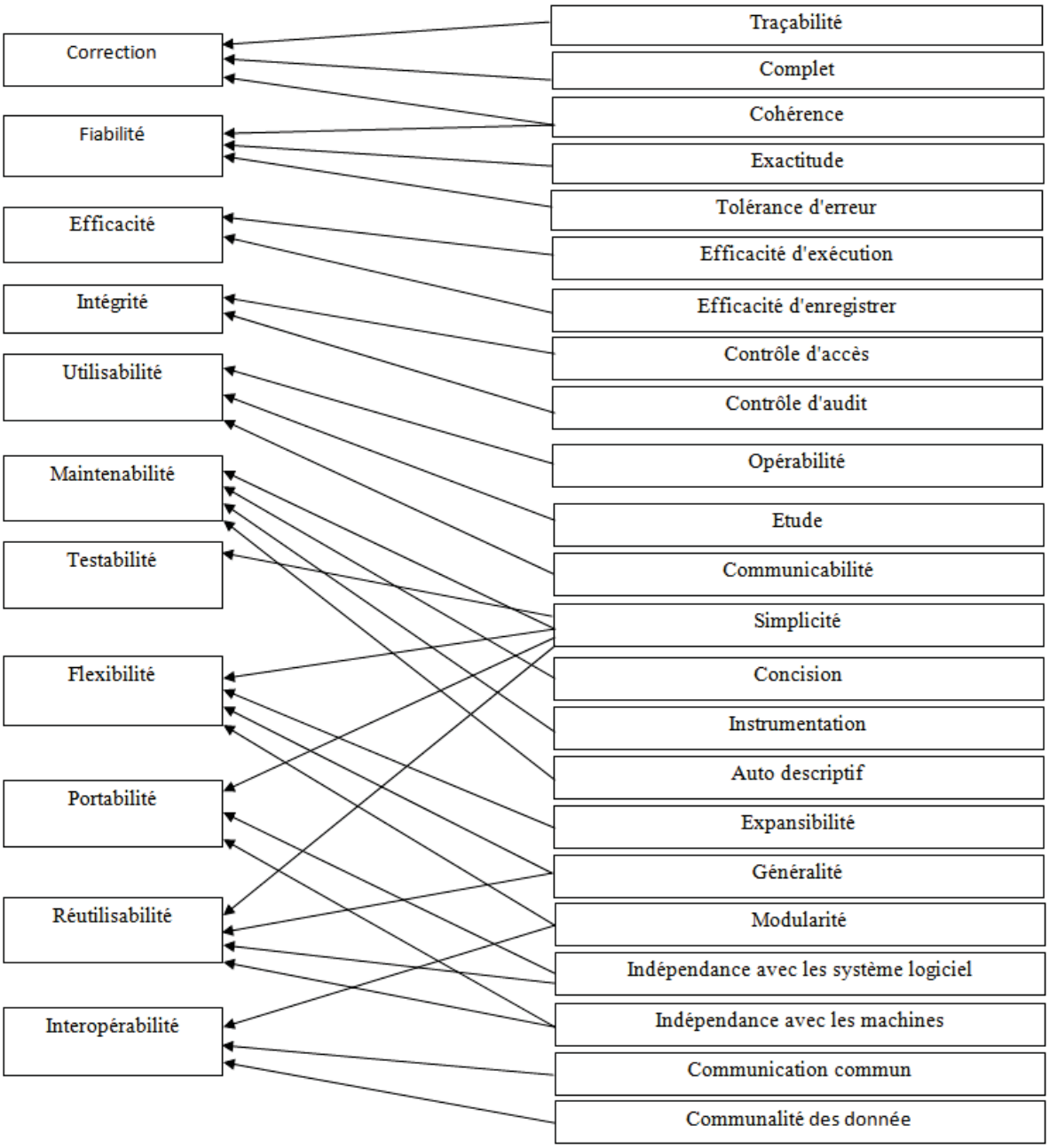

Figure 2.2 Modèle de Boehm[5]

À travers ses multiples définitions, on peut constater qu'il n'est pas possible de donner une définition unique à la qualité du logiciel. En effet, elle est multiforme et dépend du point de vue adopté c'est-à-dire que selon que nous soyons fournisseur ou client. Pour ce mémoire, nous nous intéressons plus au côté fournisseur c'est-à-dire les concepteurs et développeurs de logiciel. Dans ce mémoire, l'interêt est porté aux quatre facteurs primordiaux à savoir la correction, la robustesse, l'extensibilité et la réutilisabilité réuni sous le terme fiabilité pour les deux premiers et modularité pour les deux autres.

[5]Source modèle de Boehm : [6]

CHAPITRE 3
ÉTAT DE L'ART

3.1 Les tests de logiciel

Une des grandes préoccupations des créateurs de logiciels est d'être certains que leurs applications informatiques fonctionnent correctement dans toutes les situations possibles. Le test de logiciel constitue une façon de vérifier qu'un système informatique fonctionne bien. Selon la norme IEEE 829-1998 *« un test est un ensemble de cas à tester (état de l'objet à tester avant exécution du test, actions ou données en entrée, valeurs ou observations attendues, et état de l'objet après exécution), éventuellement accompagnés d'une procédure d'exécution (séquence d'actions à exécuter). Il est lié à un objectif »* [2]. Les tests sont généralement effectués sur deux niveaux :

- Structurel : le test est fait au niveau du code source.
- Fonctionnel : ce sont les tests portés sur les fonctionnalités de bas, comme de haut niveau.

Les tests de logiciel sont en général divisés selon deux grandes approches à savoir les tests en boîte noire et les tests en boîte blanche.

« Un test en boîte noire est un test basé sur les spécifications, et n'examine un système ou une unité que de l'extérieur, c'est-à-dire par le biais de son interface publique » [22]. Ces types de tests sont mis en place, sans avoir connaissance de la manière dont l'objet a été implémenté. Dans cette approche, le code source n'est pas accessible ou disponible. Les tests de cette catégorie sont centrés sur l'établissement des conditions d'entrées afin de vérifier la fonctionnalité du logiciel. Dans cette technique de tests, les erreurs suivantes sont les plus recherchées :

- Fonctionnalités erronées ou manquantes.
- Erreur d'assignation initiale et de terminaison.
- Erreurs d'interface.
- Erreurs dans les structures de données.
- Problèmes dans la performance.

En somme les tests boîte noire n'utilisent aucune information concernant la structure interne d'un composant du système à tester. En général, uns test boîte noire nécessite les étapes suivantes [14] :

- Créer un plan de test basé sur les spécifications de l'unité.
- Créer des jeux de test permettant de tester les spécifications.
- Appliquer les cas de tests.
- Vérifier que les sorties sont conformes.

Parmi les tests à boîte noire, on peut citer les tests suivants :

- Les tests fonctionnels permettent de valider une fonctionnalité. Au niveau de l'implémentation, ces tests valident les responsabilités des méthodes ou des fonctionnalités fournies par une classe. Par exemple, vérifier qu'un objet « connexion » permet effectivement d'établir une connexion/déconnexion par l'intermédiaire de ces méthodes. Du point de vue applicatif, ces tests valident des fonctionnalités de haut niveau qui seront par la suite utilisées par un utilisateur. Ces tests sont généralement effectués à l'aide de logiciel permettant d'enregistrer des manipulations utilisateurs et de les rejouer.
- Les tests de montée en charge permettent de vérifier la robustesse d'une application. C'est-à-dire de tester sa capacité de traitement, ses temps de réponse, etc. Par exemple, sur une application client/serveur, on testera les temps de réponse suivant la quantité d'entités connectées. En effet, un client privilégiera une application ayant des temps de réponse respectables, que ce soit lors d'une utilisation avec une faible charge ou non.
- Le test de non-régression consiste à vérifier si la nouvelle version de l'unité a été corrigée et si la modification n'a pas généré d'effets de bords. Le principe consiste à tester la nouvelle version de l'unité avec le jeu de test précédent. Les tests de non-régression sont des tests qui sont effectués implicitement. Ces tests sont effectués par l'intermédiaire des autres tests, qui sont exécutés et ré-exécutés, pour valider que les fonctions déjà développées soient toujours fonctionnelles lors d'une nouvelle version.
- Les tests unitaires (voir la section « Test unitaire » du présent chapitre).

Contrairement au type de test boîte noire, un test boîte blanche nécessite de connaître en détail la structure du programme. *« C'est un test basé sur une analyse de la structure interne du composant ou du système »*. Le principal bénéfice est de pouvoir tester les

différents chemins logiques pris par le code. Il vérifie si le code est robuste en contrôlant son comportement avec des cas de tests inattendus. L'implémentation de la classe doit être connue. Le but de ce test est d'exécuter chaque branche du code avec différentes conditions d'entrée afin de détecter tous les comportements anormaux. Pour faire ce type de test, il est primordial de comprendre le code source du composant à tester. Dans les tests de cette catégorie, on y trouve les tests suivants :

- Les tests d'intégrations vérifient la cohésion des interfaces des modules et valident leurs communications. Lorsqu'une application de grande envergure est développée, plusieurs parties de celle-ci sont développées séparément, puis regroupées pour fournir les fonctionnalités attendues. La phase où les modules sont regroupés est appelée la phase d'intégration et consiste à tester la communication entre les modules.
- Les tests de performance appartiennent à la famille des tests de robustesse et permettent de tester la qualité du code développé. En effet, un exemple de test de performance peut consister à vérifier les temps de réponse d'une application/module/méthode, suivant un chargement de données plus ou moins conséquent.
- Les tests unitaires qui est l'objet de la section suivante.

3.1.1 Test unitaire

Un test unitaire est un type de test utilisé pour tester les unités (les méthodes ou les fonctions) en isolation. Il permet de s'assurer que chaque unité fonctionne bien avant l'intégration avec d'autres unités. De plus, il permet de s'assurer que les unités existantes fonctionnent toujours après une modification ou un ajout. *C'est ce qui s'appelle la vérification de la régression.* Ainsi, quand il y a un problème, nous pouvons identifier rapidement quelle unité est la cause du problème sans avoir à chercher dans tous les chemins d'appels de toute l'application. Cela permet de construire l'application sur des composantes solides et testées individuellement avant de les intégrer.

Le test unitaire est un composant essentiel du processus de développement. Il augmente la qualité du code produit et réduit les temps de développement. Ces résultats sont obtenus grâce à deux techniques. La première est liée au fait que le test est réalisé au niveau de l'objet. De ce fait, les chances de générer les cas de test pouvant provoquer des erreurs, et assurant une couverture de 100% sont plus grandes. La seconde est que le code est testé dès sa création. Ceci simplifie la recherche et la correction d'éventuelles erreurs. Cette détection précoce des erreurs conduit à une réduction du temps de développement et donc des coûts, car le temps passé pour trouver un bogue et les ressources utilisées sont moindres

(des données statistiques indiquent que 2/3 des bogues problématiques en fin d'intégration auraient pu être détectés par un test unitaire) [14].

Finalement, un test unitaire opère à un niveau très détaillé et permet de tester chaque méthode et comportement dans le détail et en isolation. Concrètement, un test unitaire est en quelque sorte un programme qui exécute le code d'un autre programme pour s'assurer que celui-ci est correct. Les tests unitaires voient le jour, avec l'environnement de test sUnit[1] crée en octobre 1994 par Kent Beck.

Les tests unitaires connaîtront un grand succès avec l'arrivée du framework JUnit ; créé par Kent Beck et Erich Gamma. La popularité de JUnit entraînera la création de plusieurs frameworks d'automatisations de tests unitaires pour d'autres langages. L'ensemble des frameworks se nomme xUnit dont « x » représente le plus souvent l'initiale du langage de programmation. Un framework d'automatisation doit répondre à un certain nombre d'exigences qui sont les suivantes [22] :

- Le langage de spécification des tests est le langage de programmation lui-même.
- Le code de l'application et le code des tests doivent pouvoir être séparés.
- L'exécution et la vérification d'un cas de test sont indépendantes de l'exécution et de la vérification des autres cas de tests (description d'un test à exécuter, organiser en fonction d'un objectif de test prédéfini).
- Les cas de tests[2] peuvent être regroupés librement dans des suites de tests.
- Le succès ou l'échec de l'exécution d'un test doit apparaître au premier coup d'oeil.

Un cas de test (test case) est un ensemble composé de trois objets :

- Un état (ou contexte) de départ
- Un état (ou contexte) d'arrivée
- Un oracle, c'est-à-dire un outil qui va prédire l'état d'arrivée en fonction de l'état de départ et comparer le résultat théorique et le résultat pratique.

Une suite de tests est un ensemble de cas de tests exécutés et examinés conjointement.

[1] sUnit. logiciel de test pour le langage Smalltalk.

[2] Cas de test : la spécification d'un test inclut l'objet cible, les entrées, les sorties attendues, ainsi que le contexte et les effets de bord du cas de test [22].

JUnit

JUnit est un framework (voir figure 3.1) de test pour Java. Il définit deux types de fichiers (les cas de tests et les suites de tests). Il propose un cadre pour le développement des tests unitaires. La description des différentes classes donnée par Johannees Link dans son ouvrage « Tests unitaitres en Java. Les tests au coeur du développement » [22] est la suivante :

- TestResult : c'est le récipient dans lequel se déversent tous les résultats d'un passage de tests. Toutes les classes qui implémentent l'interface *TestCase*, *TestSuite* et *TestDecorator* sont dotées d'une instance de la classe *TestResult* dans leur méthode *« run() »*.

- TestListener : un TestListener peut être enregistré dans un *TestResult* par le biais de *addListener()*, pour être informé sur *le démarrage*, *la fin*, *l'échec* et *l'erreur* d'un test. Cette possibilité est exploitée par exemple par les trois classes *TestRunner* pour être au courant de l'exécution des tests.

- TestDecorator : cette classe sert de *super classe* pour de nombreuses extensions de framework de test. Elle implémente le modèle Decorator[3], qui autorise l'utilisation de plusieurs extensions simultanées.

- AssertionFailledError : c'est une exception levée lors de l'échec d'un appel *« assert »*, capturée par des instances de *TestCase* et finalement transmise à un *TestResult* pour être enregistrée.

- Assert : c'est la *super classe* de *TestCase* et elle accueille toutes les variantes de la méthode *Assert*, qui de surcroît sont toujours statiques. Ainsi d'autres classes (par exemple les classes *TestDecorator*) peuvent elles aussi utiliser la fonctionnalité *« assert »*.

- TestCase : c'est la classe définissant les cas de tests.

- TestSuite : cette classe définit les suites de tests. Elle permet de regrouper plusieurs cas de tests et les exécuter en même temps.

[3]Decorateur (Decorator) est un patron de conception qui permet d'ajouter des fonctionnalités nouvelles à une classe de façon dynamique sans impacter les classes qui l'utilisent ou en héritent [15].

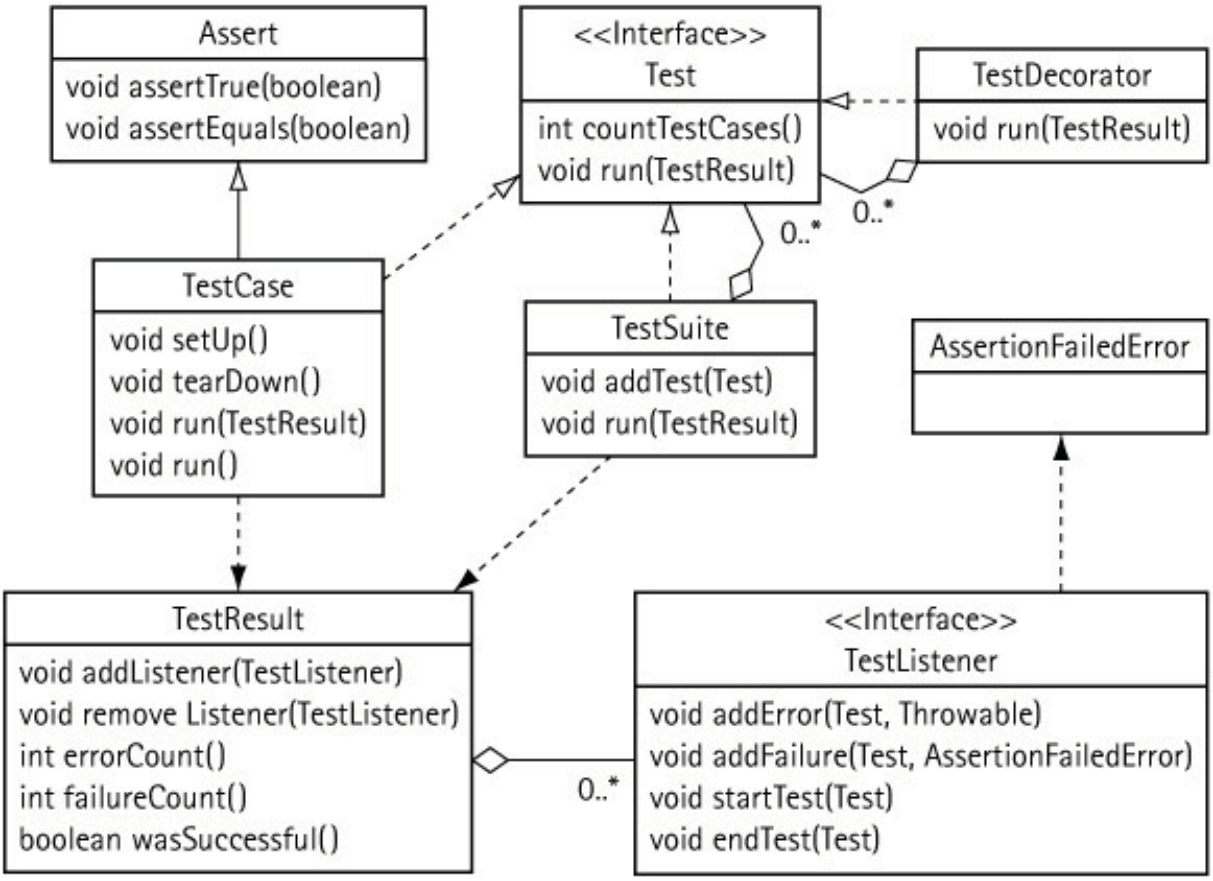

Figure 3.1 Diagramme de classe du framework JUnit

JUnit permet l'automatisation des tests unitaires ainsi que leur exécution. Il est considéré aujourd'hui comme un outil standard et a été intégré dans la majorité des environnements de développement de logiciels Java. Il est à sa cinquième version [27].

Afin de concevoir de très bons tests avec JUnit, il faut respecter quelques bonnes pratiques pour avoir de très bons résultats [4] [5] [28]. Johannes Link fait les recommandations suivantes [22] :

1. Le nom du test doit décrire la fonctionnalité testée.
2. La longueur de la méthode de test doit être aussi réduite que possible.
3. Le code de test doit contenir le moins possible d'instructions logiques.
4. Les résultats attendus doivent être indiqués sous forme de constantes définies à l'avance et non être calculés dans le test.
5. Les données de test et les résultats attendus doivent être proches que possible.

6. Les exceptions, qui peuvent aller jusqu'à la méthode de test et représenter une erreur ne doit pas être capturées.

7. Les valeurs limites : une manière d'augmenter l'efficacité des tests consiste à tester plus fortement les frontières d'une plage de valeurs autorisée, car c'est à ce niveau que se produisent les erreurs les plus fréquentes.

L'écriture des fichiers de tests et leurs exécutions manuelles deviennent pénibles si le projet est volumineux d'où l'importance de l'automatisation [32] [22] [3] [38] [37]. En somme, JUnit permet la génération des fichiers de tests, l'exécution des tests unitaires et offre une interface graphique (voir figure 3.2) permettant de voir le résultat de l'exécution. Dans la figure 3.2 la ligne verte indique que tous les tests « passent ».

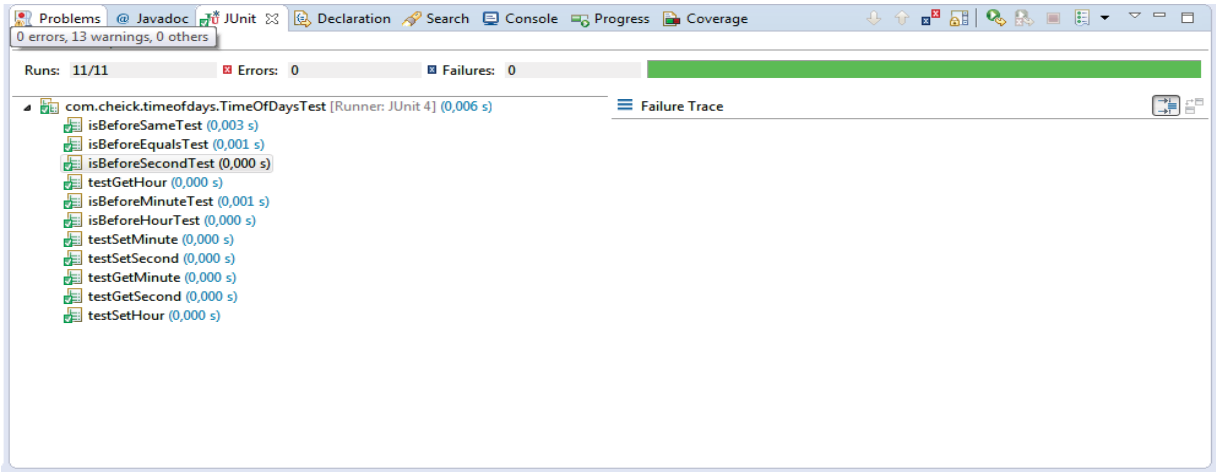

Figure 3.2 Interface d'exécution d'une classe de test avec JUnit

3.2 Programmation par contrat (Design by contrat)

En informatique, les méthodes formelles sont des techniques permettant de raisonner rigoureusement à l'aide de logique mathématique sur des programmes informatiques ou du matériel électronique afin de démontrer leur validité par rapport à une certaine spécification [36].

La présence d'une spécification est l'élément-clé pour le développement de logiciels fiables et corrects. Ainsi, on ne peut pas vérifier la fiabilité ou la correction d'un logiciel sans se référer à la spécification qu'il doit satisfaire. Le logiciel est fiable si son comportement est celui décrit par sa spécification. Dans le cas contraire, on peut dire qu'il n'est pas fiable.

La programmation par contrat ou « Design by Contrat (DBC) » permet de construire des logiciels fiables. Un contrat est un ensemble de conditions précises qui régissent les relations entre une classe fournisseur et ses clients. Le contrat d'une classe contient les contrats individuels des méthodes visibles de la classe, représentée par des préconditions et des postconditions et les propriétés globales de la classe, représentées par l'invariant de la classe.

« Une précondition est une assertion attachée à une méthode et qui doit être garantie par chaque client à chaque appel de la méthode ». Elle fait partie du contrat qui gouverne la méthode et elle exprime les contraintes que doit vérifier une méthode pour fonctionner correctement.

« Une postcondition est une assertion attachée à une méthode et qui doit être garantie par le corps de la méthode au moment du retour de n'importe lequel des appels à cette méthode si la précondition était vérifiée au début de l'appel ». Elle exprime les propriétés de l'état qui résulte de l'exécution d'une méthode.

« L'invariant d'une classe est une assertion qui doit être vérifiée lors de la création de chaque instance d'une classe et préservée par chaque méthode exportée de la classe, de sorte qu'elle sera vérifiée par toutes les instances de la classe observable de l'extérieur ». Dans la programmation par contrat, toute violation à une assertion à l'exécution est la manifestation d'un bogue dans le logiciel. Meyer définit les règles de violation suivante :

- une violation de précondition est la manifestation d'un bogue chez le client.
- une violation de postcondition est la manifestation d'un bogue chez le fournisseur.

« Si vous promettez d'appeler [une méthode] avec une précondition vérifiée, en retour, je promets de délivrer un état final dans lequel la postcondition est vérifiée » [25].

Le concept permet d'avoir des logiciels fiables si les règles introduites sont respectées par les deux parties. Ses règles permettent de responsabiliser le client et le fournisseur.

	Obligations	**Bénéfices**
Client	Doit satisfaire la précondition	Assure que la postcondition sera satisfaite
Fournisseur	Doit satisfaire la postcondition	Assure que la précondition sera satisfaite

Tableau 3.1 Obligations et bénéfices du client et du fournisseur d'un service

On peut donc résumer le concept de la programmation par contrat par ceci :

{Appelant A} Programme Prog {Appellé E}

Cela peut être traduit par : le **programme Prog** exige de son **appelant A** au respect du contrat défini par l'**appelé E** afin que son exécution se termine bien avec un résultat correct. La programmation par contrat s'appuie sur la logique de Hoare [16] pour prouver le facteur de correction dans le logiciel.

La logique de Hoare est un système formel permettant de raisonner sur la correction des logiciels en utilisant un formalisme logique. La vérification de la correction se base sur les triplets de logique d'Hoare défini par :

{P} C {Q} où **P** et **Q** sont des prédicats et **C** est un programme.

Le prédicat **P** est appelé précondition, il décrit l'ensemble des états en entrée de la portion du programme **C**. Le prédicat **Q**, nommé postcondition, caractérise quant à lui un ensemble d'états après transformation par le code **C**. On distingue deux types de correction d'un triplet de Hoare :

- **Correction partielle** : Le triplet de Hoare suivant **{P} C {Q}** est vrai si pour tout état initial vérifiant **P**, si l'exécution du programme **C** se termine, alors **Q** est vraie après l'exécution de **C**. On dit que le programme **C** est partiellement correct par rapport à **P** et **Q**.
- **Correction totale** : Le triplet de Hoare suivant **{P} C {Q}** est vrai si pour tout état initial de **C** vérifiant **P**, **C** se termine et **Q** est vraie après l'exécution de **C**. On dit que le programme **C** est totalement correct par rapport à **P** et à **Q**.

Afin d'écrire un bon contrat, Richard Mitchell et Jim McKim définissent six principes[26] qui sont :

- Principe 1 : séparer les requêtes des commandes. Les requêtes retournent des résultats, mais ne modifient pas la propriété des objets. Par contre, les commandes modifient les objets, mais ne retournent pas de résultat.
- Principe 2 : séparer les requêtes de base des requêtes dérivées.
- Principe 3 : pour chaque requête dérivée, écrire la postcondition qui spécifie quel résultat sera retournée en terme d'un ou de plusieurs requêtes de base.
- Principe 4 : pour chaque commande, écrire la postcondition qui spécifie la valeur de chaque requête de base.
- Principe 5 : pour chaque commande, décider d'une suite de précondition.
- Principe 6 : écrire l'invariant pour définir les propriétés qui resteront inchangées.

Si le concept de programmation par contrat est intégré au langage Eiffel [24], en Java, il faut trouver une aide externe. Au cours des dernières années, plusieurs méthodes et outils ont été proposés pour soutenir la programmation par contrat en Java.

3.2.1 Programmation par contrat en Java avec C4J (Contrat for Java)

C4J ou Contrat for Java a été développé en mars 2006 par Jonas Bergström. Cette version a été utilisée dans le cours de « La programmation orientée objet avec Java » pour enseigner près de 1800 étudiants par Hagen Buchawad à l'Institut de technologie de Karlsruhe. Vu l'importance du projet, une seconde génération de C4J voit le jour et est soutenue par plusieurs ingénieurs. En juillet 2012, l'association de génie logiciel de Karlsruhe (VKSI) en Allemagne accepte de prendre la maintenance du projet ainsi que son hébergement. En octobre 2012 la version C4J 6.0 a été rendue stable par VKSI [17].

L'objectif premier de C4J est de faciliter l'écriture des contrats de classes. Pour cela C4J offre un framework aux développeurs leur permettant d'ajouter facilement les contrats de classe dans leur programme. Les contrats de classe sont des classes Java à part entière. Ils permettent de définir les préconditions, les postconditions et les invariants. C4J permet de définir des méthodes pures qui sont des méthodes sans effet de bord à l'aide d'assertion « @Pure ou @PureTarget ». C4J offre aussi la possibilité d'accéder aux anciennes valeurs des attributs dans les postconditions avec l'assertion « @old ». C4J dispose également de quatre fichiers de configuration qui permettent par exemple d'activer ou de désactiver des contrats, de définir une réaction suite à une violation de contrat et autres. C4J dispose d'un plug-in pour l'environnement de développement Eclipse et d'un JAR (Java archive) pour les autres outils de développement. Avec ce plug-in il est possible de générer un projet C4J. La structure d'un projet C4J (voir la figure 3.3) est la suivante :

- src/contract/java : c'est dans ce dossier que le développeur met les fichiers sources des contrats.
- src/main/java : ce dossier contient les fichiers sources des classes.
- src/main/resources : ce dossier contient les fichiers de configuration de C4J.
- src/test/java : contient les fichiers de tests des classes.

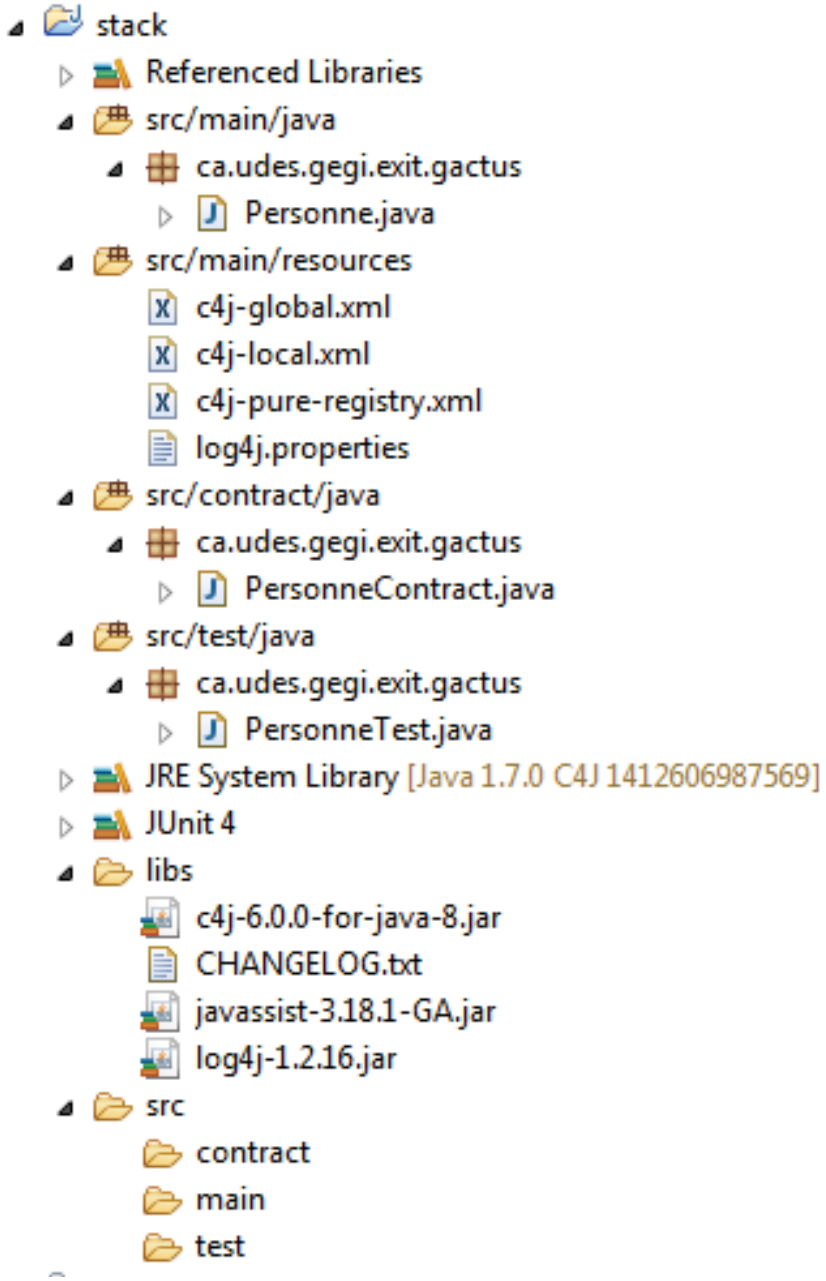

Figure 3.3 Exemple de structure d'un projet C4J

Il est possible de convertir un projet initial Java en un projet C4J. Dans un cas pratique, pour vérifier l'état d'un contrat avec C4J, le développeur doit créer une classe ayant une méthode « main » permettant d'exécuter un programme Java. En exécutant le programme si une classe ayant un contrat définie est appélée, C4J imprime l'état du contrat dans la

console de l'éditeur. Pour afficher l'exécution des contrats, C4J fait appel à la librairie Log4J[4].

Un exemple de contrat de classe est donné en annexe E.

[4]Log4j : Log4j est une bibliothèque de journalisation des logs.

3.3 Analyse du code Java

Dans le cadre du développement du logiciel GACTUS, une étude sur l'analyse du code du programme Java a été faite. Mais avant de présenter Java Development Tools (JDT) qui permet de manipuler le code source Java, voici des rappels concernant l'analyse de code source d'une manière générale.

Avant d'extraire les informations pertinentes d'un programme, il est important, dans le cas d'un code source, de vérifier que ce dernier respecte bien les règles de syntaxe du langage utilisé pour l'écrire. L'analyse du code [1] est une technique utilisée dans le processus de compilations des programmes. Un compilateur est un programme informatique qui transforme un code source écrit dans un langage de programmation (le langage source) en un autre langage informatique (le langage cible) [35] (voir les deux figures 3.4 et 3.5). Le compilateur doit aussi vérifier que le programme a un certain sens et signaler les erreurs qu'il détecte. Il y a deux parties dans la compilation : l'analyse du code et la synthèse. Dans le cas de cette recherche, on s'intéresse uniquement à l'analyse du code.

L'analyse du code partitionne le programme source en ces constituants et en crée une représentation intermédiaire. Elle consiste à lire le code afin de comprendre la structure, la syntaxe et la sémantique d'un programme. Pendant l'analyse du code, les opérations spécifiées par le programme source sont déterminées et conservées dans une structure hiérarchique appelée arbre de syntaxe abstrait (AST). L'AST est un arbre qui présente tous les détails (caractéristiques) d'un code source.

L'analyse du code permet de savoir comment isoler le code en petits morceaux appelés jetons (tokens). De ces jetons, il est possible de construire l'arbre[5] représentant le programme. Enfin, il est possible de trouver les informations nécessaires pour générer les squelettes de classes de tests et de contrats. La génération de codes sources est une opération permettant de générer automatiquement du code source. Son but est d'automatiser la production de codes sources répétitifs afin de minimiser les risques d'erreurs et de permettre au programmeur de se concentrer sur l'écriture de codes à plus grande valeur ajoutée.

[5] Arbre : structure de données

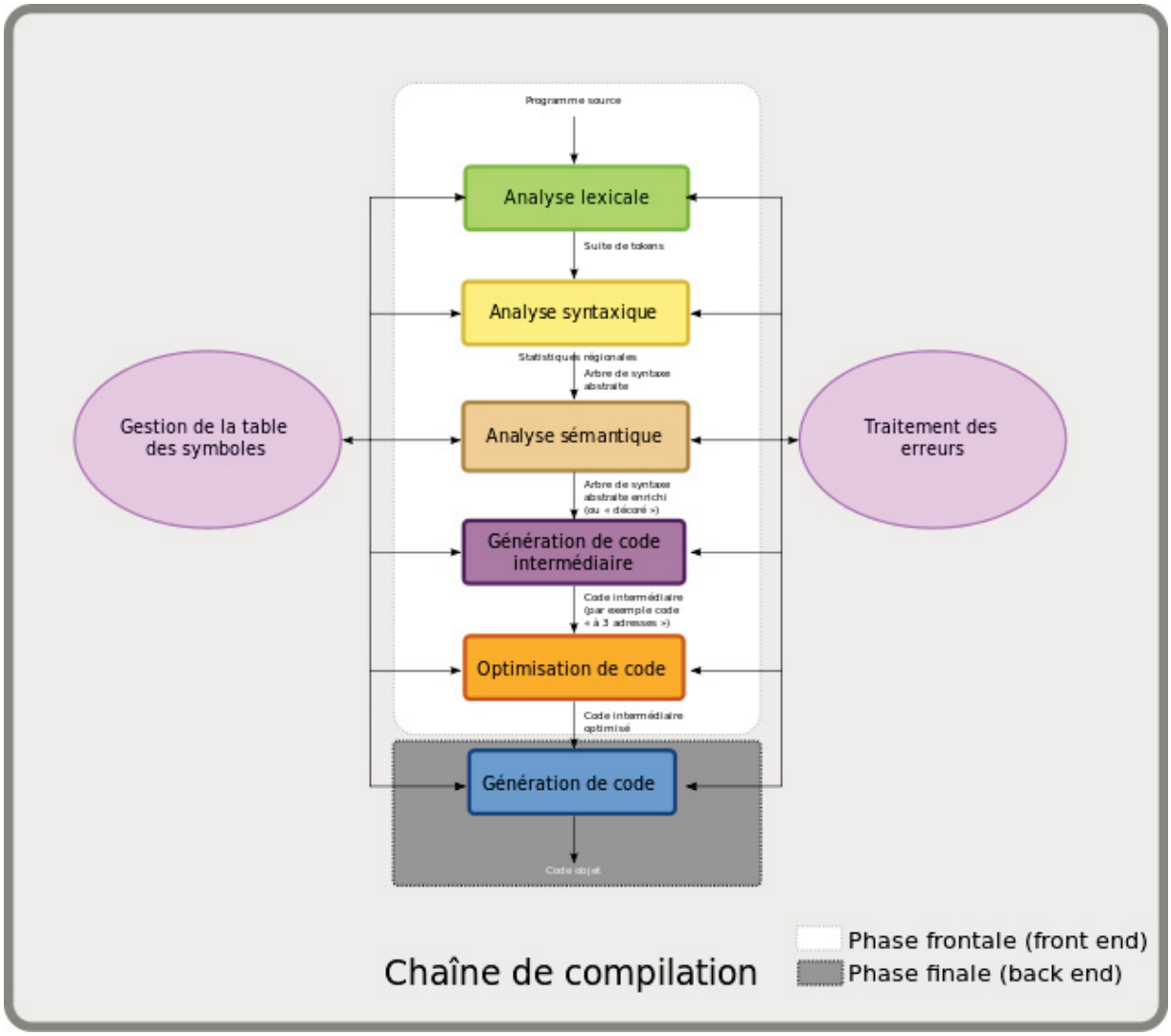

Figure 3.4 Schéma d'une chaîne de compilation classique

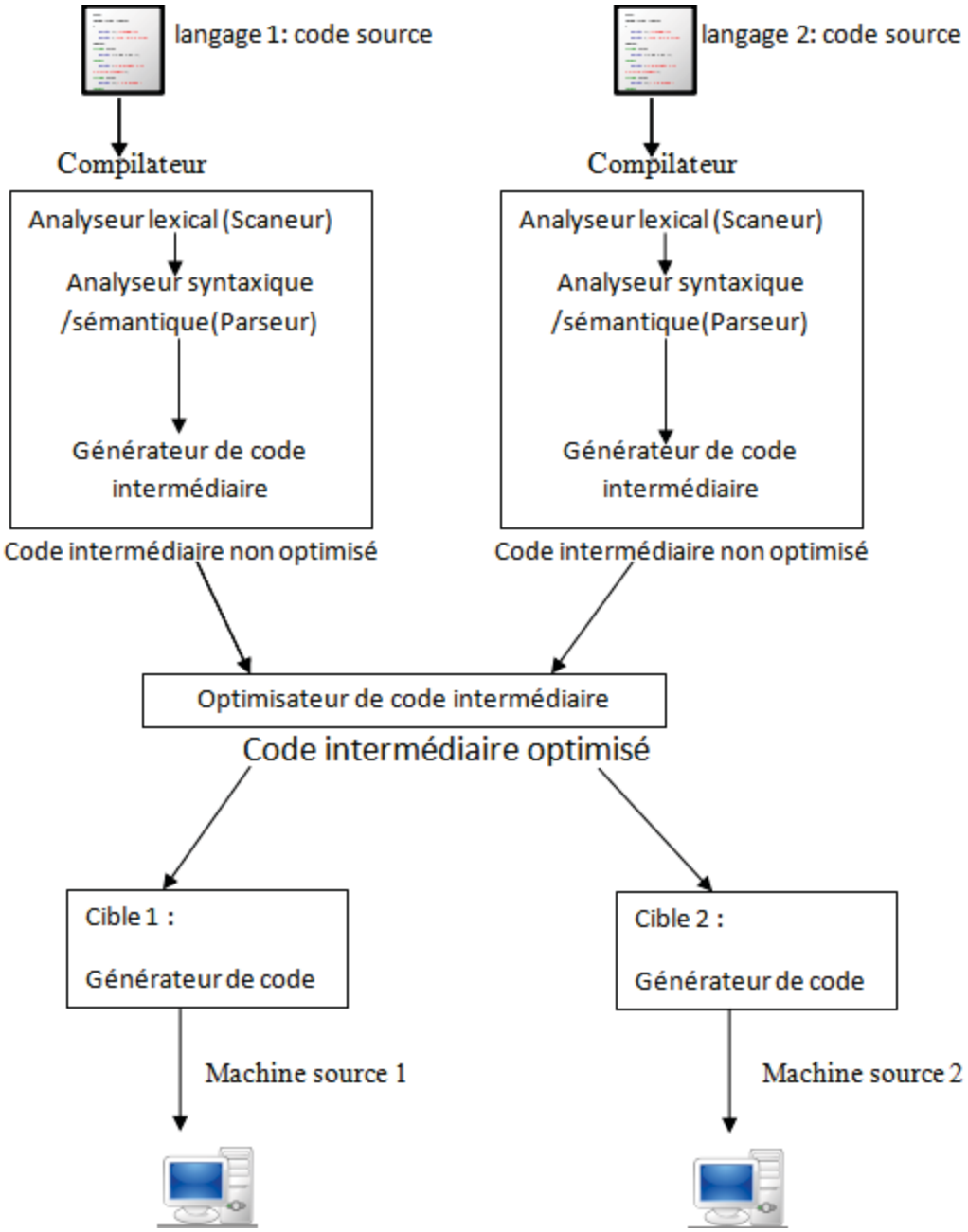

Figure 3.5 Schéma de compilation multi-source multi-cible

3.3.1 Java Development Tools (JDT)

Eclipse dispose d'un outil intégré appelé Java Development Tools (JDT) [18]. Java development tool fournit un environnement de développement pour le langage Java. Il comprend plusieurs APIs et fournit :

- les perspectives « Java » et « Navigation Java »
- les vues « Packages » et « Hierarchie »
- les éditeurs « Java » et « Scrapbook »
- les assistants : pour créer de nouveaux projets, packages, classes, interfaces...

C'est un plug-in incorporé à Eclipse, puissant, extensible. Son extensibilité se traduit par la possibilité d'ajouter de nouvelles fonctionnalités à l'environnement de développement Java. Il fournit également des APIs[6] permettant l'accès et la manipulation du code source, l'ajout de nouvelles actions aux différents vues et éditeurs de l'environnement de développement Java, etc [12] [21]. JDT permet l'accès au code source de deux manières à savoir le Java modèle (Java Model) et l'abstract Syntax Tree (AST) [33] [31].

- Java Modèle est l'ensemble des classes qui modélisent les objets associés à la création, l'édition et la construction d'un programme Java (figure 3.6).
- L'AST est un framework pour de nombreux outils de l'environnement de développement Eclipse. L'AST donne la structure d'un code source dans une forme d'arbre. Cet arbre est plus pratique pour analyser et modifier dynamiquement du code source.

Les classes du modèle Java sont définies dans **« org.eclipse.jdt.core »**, elles implémentent des comportements Java spécifiques pour les ressources et décomposent ces dernières en éléments du modèle. Le package **« org.eclipse.jdt.core »** définit les classes qui modélisent les éléments dont un programme Java est constitué. JDT utilise un modèle objet « en mémoire » pour représenter la structure d'un programme Java. Cette structure est dérivée du chemin de classe du projet. Ce modèle est hiérarchique. Les éléments d'un programme peuvent être décomposés en éléments enfants [18]. Le tableau ci-dessous répertorie les différentes d'éléments Java.

[6] API : Application programming interface ou interface de programmation ensemble de classe et de méthode permettant un logiciel d'offrir ses services à un autre logiciel [34]

Éléments	Description
IJavaModel	Représente l'élément Java racine, qui correspond à l'espace de travail. Il s'agit du parent de tous les projets de nature Java. Il vous permet également d'accéder à ceux qui ne sont pas de nature Java.
IJavaProject	Représente un projet Java dans l'espace de travail (enfant de IJavaModel).
IPackageFragmentRoot	Représente un ensemble de fragments de package et établit le lien entre ces fragments et une ressource sous-jacente qui peuvent être un dossier, un fichier JAR ou un fichier ZIP (enfant de IJavaProject).
IPackageFragment	Représente la partie de l'espace de travail qui correspond à un package entier ou à une portion de ce package (Enfant de IPackageFragmentRoot).
ICompilationUnit	Représente un fichier source Java (.java) (enfant de IPackageFragment).
IPackageDeclaration	Représente une déclaration de package dans une unité de compilation (enfant de ICompilationUnit).
IImportContainer	Représente l'ensemble de déclarations d'importation de packages dans une unité de compilation (enfant de ICompilationUnit).
IImportDeclaration	Représente une déclaration d'importation de packages particulière (enfant de IImportContainer).
IType	Représente soit un type source à l'intérieur d'une unité de compilation, soit un type binaire à l'intérieur d'un fichier classe.
IField	Représente un champ à l'intérieur d'un type (enfant de IType).
IMethod	Représente une méthode ou un constructeur à l'intérieur d'un type (enfant de IType).
IInitializer	Représente un initialiseur statique ou d'instance à l'intérieur d'un type (enfant de IType).
IClassFile	Représente un type compilé (binaire) (Élément enfant de IPackageFragment).

ITypeParameter	Représente un paramètre de type (n'étant pas un enfant d'élément Java, vous pouvez l'obtenir à l'aide de IType.getTypeParameter (String) ou IMethod.getTypeParameter (String))
ILocalVariable	Représente une variable locale dans une méthode ou un initialiseur (n'étant pas un enfant d'élément Java, vous pouvez l'obtenir à l'aide de ICodeAssist.codeSelect (int, int))

Tableau 3.2 Tableau représentatif des classes du Modèle Java

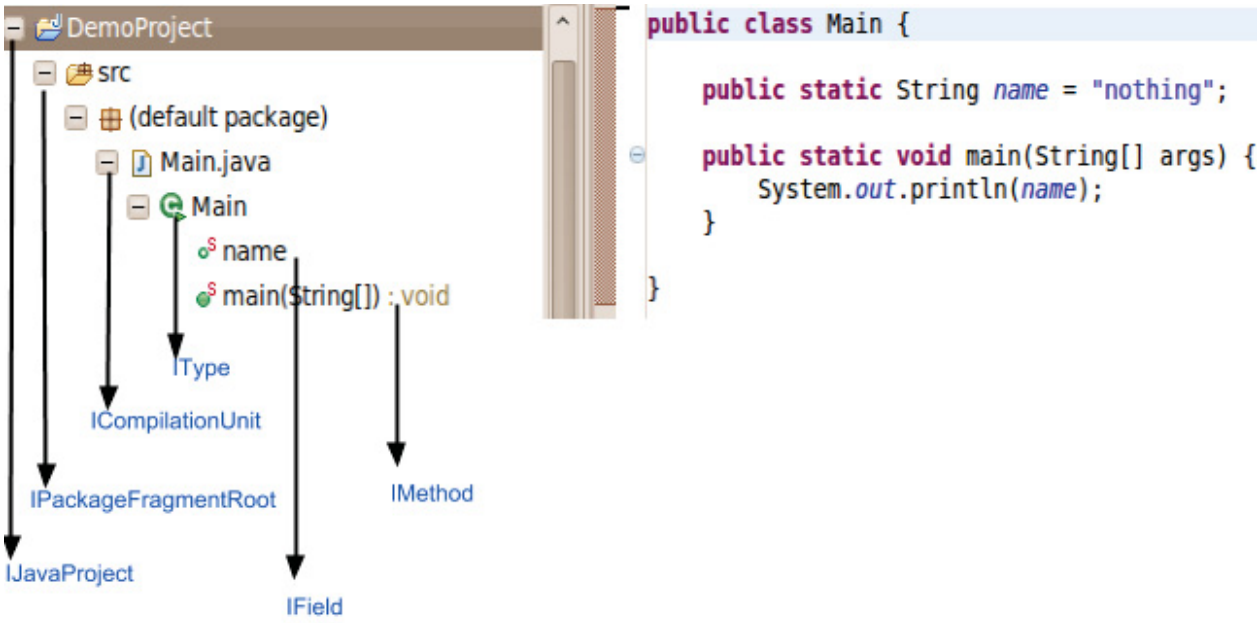

Figure 3.6 Java modèle vue

3.4 Résumé de l'état de l'art

L'état de l'art avait pour but de faire une étude de la littérature des concepts clés c'est-à-dire les tests unitaires, la programmation par contrats afin de prendre connaissance des avancées sur ces concepts dans le domaine du génie logiciel. Une étude de l'outil JDT a été faite afin de comprendre la manipulation de code source Java sous Eclipse.

Les tests de logiciel permettent de vérifier que le code fonctionne bien. Ils sont regroupés en deux grandes catégories à savoir les tests boîtes blanches et les tests boîtes noires.

Les tests boîtes noires n'utilisent aucune information concernant la structure interne d'un composant du système à tester par contre les tests boîtes blanches nécessites de connaitre en détail la structure du programme. Il existe plusieurs types de tests en fonction de la catégorie. Parmi eux, il y'a les tests unitaires qui permettent de tester unitairement une unité afin de s'assurer de son bon fonctionnement. En programmation orientée objet, une unité est une classe. JUnit est un framework de tests unitaires permettant de faire des tests unitaires en Java. Il dispose d'annotations et un environnement d'exécution permettant de voir l'exécution. Si un test n'est pas bien écrit cela peu causer des erreurs.

La programmation par contrat élaboré par Bertrand Meyer en s'appuyant sur la logique d'Hoare permet d'assurer de la correction dans le logiciel. À l'aide de spécifications définies, on peut facilement savoir si le logiciel est correct ou non. La programmation par contrat permet de responsabiliser le client (utilisateur) et le fournisseur (développeur) du logiciel. Elle consiste à définir des contrats de classe. Un contrat de classe contient des préconditons, des postconditions et un invariant de classe. La précondition est une condition qui doit être respectée au début de l'exécution d'une méthode. La postcondition est une condition qui doit être vérifiée après exécution de la méthode. L'invariant de classe quant à lui est une propriété qui doit être toujours vérifiée. Le non-respect de la précondition est une manifestation d'un bogue chez le client (appelant de la méthode). Un bogue se présente au niveau du fournisseur si la postcondition est violée. Pour définir un bon contrat de classe, il faut respecter les six principes identifiés par Richard Mitchell et Jim McKim dans le livre « Design by Contract, by example ». En Java, C4J est un framework facilitant la création de contrats de classe pour un projet Java. C4J permet de définir des préconditions, des postcondions, des invariants. C4J dispose de quatre fichiers de configuration permettant de configurer les contrats de classes dans le projet.

L'analyse de code source est une étape importante lors de la compilation. Elle permet de comprendre la structure, la syntaxe et la sémantique d'un programme. L'analyse du code permet d'isoler le code en petits morceaux et de construire l'arbre syntaxique. Avec cet arbre il est possible retrouver toutes les informations nécessaires à la génération de nouveau code source. Java Developpment tools (JDT) permet d'analyser un code d'un programme Java. JDT permet d'accéder à un programme Java de deux manières : sous forme d'arbre (AST) et sous forme de Java modèle. JDT permet de manipuler le code source d'un projet Java sous Eclipse.

CHAPITRE 4

LE LOGICIEL GACTUS

Ce chapitre décrit le plug-in GACTUS et l'analyse effectuée pour la réalisation, la conception du plug-in GACTUS ainsi que la phase de développement. À la fin, il présente comment utiliser le plug-in. La réalisation d'un plug-in Eclipse se fait avec l'environnement de développement Eclipse RCP (Rich Client Platform).

4.1 Description du plug-in GACTUS

Avant de décrire le plug-in GACTUS, il serait judicieux de décrire un plug-in Eclipse d'une manière générale.

Un plug-in Eclipse est un programme informatique conçu pour ajouter de nouvelles fonctionnalités. Le plug-in Eclipse est un fichier JAR (Java Archive) contenant, en plus de ses classes Java, deux fichiers manifestes à savoir « META-INF/MANISFEST.MF » et « plugin.xml ».

Le fichier « MANIFEST.MF » est exploité par le noyau d'Eclipse, pour obtenir des informations sur le plug-in qui serviront notamment à gérer le cycle de vie du plug-in et ses relations avec les autres plug-ins.

Le fichier « plugin.xml » est propre à Eclipse, il sert à concrétiser les fonctionnalités d'extensibilité d'Eclipse. Par ce fichier, des plug-ins vont déclarer des points d'extension et d'autres plug-ins vont se brancher sur ces points d'extensions. Ce fichier est créé à chaque création d'un projet « Plug-in Project ». Si le plug-in pour bien fonctionner dépend de d'autres plug-ins, l'ajout des plug-ins de dépendance se fait par ce fichier à l'aide du bouton « add » de son menu « dependencies ». La figure 4.1 montre les différents plug-in dont GACTUS dépend.

Les fonctionnalités que GACTUS apporte à Eclipse sont les suivantes :

- Génération de squelette de contrat de classe adaptée au traitement de C4J (voir structure d'un contrat de classe en annexe B).
- Génération de squelette des cas de tests d'une classe de test adaptée au traitement de JUnit (voir structure d'une classe de test en annexe B).

- Création et configuration automatique des fichiers de configuration (lorsque nécessaire).
- Ajout de librairies C4J et de JUnit parmi les librairies du projet (lorsque nécessaire).
- Restructuration d'une classe existante après génération de son contrat de classe.
- Restructuration d'un projet Java avec la création de dossiers tels que « contrat », « tests », « resource » (lorsque nécessaire).

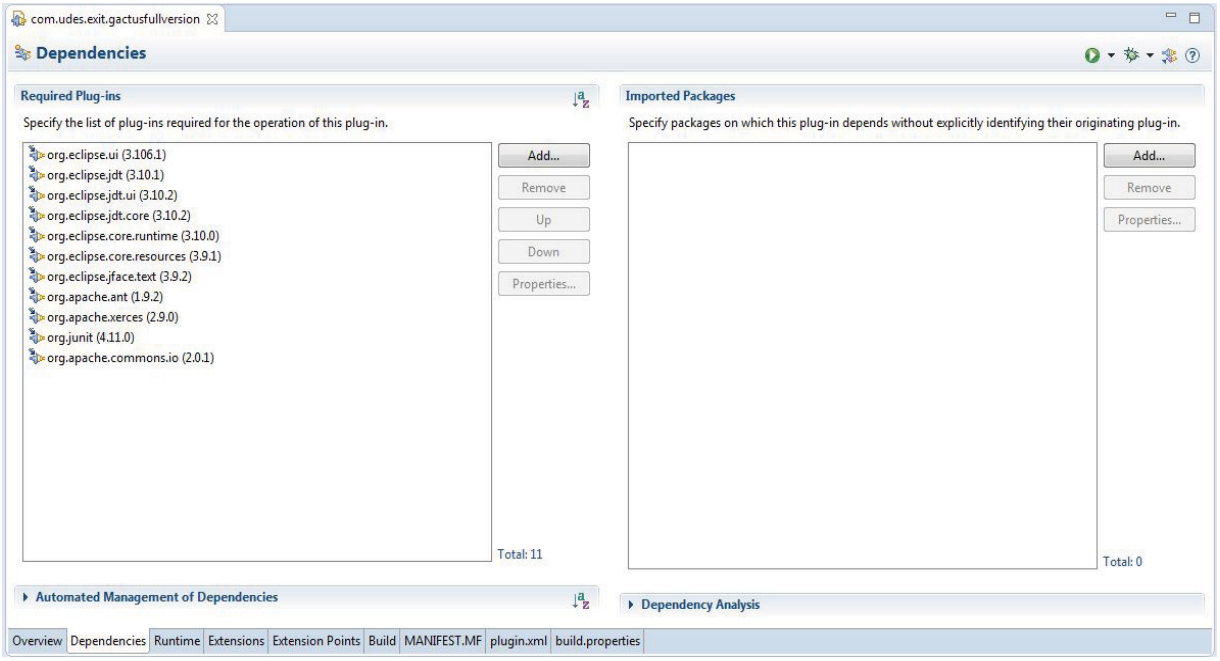

Figure 4.1 Fenêtre du menu « dependencies » du fichier « plugin.xml ».

4.2 Analyse des besoins

Comme tout projet de développement de logiciel, une analyse a été faite afin de déceler les besoins et les contraintes pour la réalisation du logiciel GACTUS. Dans cette section, nous donnerons d'abord les contraintes générales qui en découlent. Ensuite, nous donnerons des contraintes imposées par JUnit et par C4J.

4.2.1 Contraintes générales

Les contraintes générales liées à ce présent projet sont les suivantes :

- Le code généré devrait être du code Java.

- La génération devrait être uniquement pour du code Java.
- La génération est pour les classes situées dans le dossier « src » d'un projet Java.
- La génération du code devrait être possible pour une classe ou une interface.
- Il devrait être possible de générer du code pour l'ensemble des classes d'un paquet, ou pour toutes les classes du dossier « src » d'un projet.
- Le code généré des cas de tests devrait être conforme à celui de JUnit.
- Le code des contrats de classes devrait respecter C4J.
- JUnit (version 4) et C4J devront figurer parmi les librairies du projet.
- Le projet devrait avoir un dossier nommé « contrat » qui contiendra tous les contrats de classes.
- Le projet devrait avoir un dossier source nommé « tests » qui contiendra toutes les classes de tests unitaires.
- Le projet devrait avoir un dossier nommé « resource ». Ce dossier contiendra tous les fichiers de configuration.
- Afin de faciliter l'utilisation, le plug-in devrait disposer d'une interface graphique simple et conviviale.

4.2.2 Analyse des contraintes particulières d'une classe de test imposées par JUnit 4

Une classe de test est une classe publique Java. Le nom de cette classe est composé du nom de la classe à tester suivi de la chaine « Test ». Cette classe doit contenir au moins un cas de test. Un cas de test est une méthode dont le nom commence par « test » suivi d'une chaine de caractère (en général le nom de la méthode à tester). Un cas de test doit avoir les caractéristiques suivantes :

- Il doit être déclaré « public ».
- Le type de retour doit être « void ».
- Il ne doit pas avoir de paramètres.
- Il doit être annoté par « @Test ».

Le squelette de classe de test unitaire généré par GACTUS devrait respecter ces contraintes. En plus de cela, chaque cas de test de test doit contenir ces deux contraintes :

- `«fail("not yet implements")»` : cette fonction fait échouer le cas de test. La raison est de permettre au pour que le développeur d'écrire le code source nécessaire pour faire passer le test.
- `«//TODO: Please add your test case source code here»` : cette chaine de caractère est un commentaire invitant le développeur à ajouter le code source du cas de test.
- toutes les méthodes de la classe dont le squelette de classe de test unitaire a été généré devront avoir un cas de test dans le squelette de classe de test unitaire.

En annexe B se trouvent une classe et le squelette de sa classe de test unitaire généré par GACTUS.

4.2.3 Analyse des contraintes imposées par C4J

Avec C4J, un contrat de classe peut être à l'interne de la classe où le contrat devrait s'appliquer. Dans ce cas, la classe et son contrat se trouvent dans un même fichier. Le contrat peut être aussi dans un autre fichier, c'est-à-dire être une classe externe à la classe où il devrait s'appliquer. Avec GACTUS, nous optons pour une classe externe afin d'avoir une bonne organisation du projet. Les contraintes suivantes devront être respectées :

- Un contrat de classe doit être une classe Java.
- Le nom du contrat de classe est le nom de classe où le contrat s'applique suivi de « Contract ».
- Le contrat de classe doit être annoté par « @Contract ».
- Elle doit étendre ou implémenter la classe sur laquelle le contrat s'applique.
- La méthode définissant l'invariant de classe doit être annotée par « @ClassInvariant ».
- Chaque méthode définissant un contrat doit définir la précondition et la postcondition à l'aide des structures de contrôle *(if(preCondition()) et if(postCondition()))*.
- Le contrat de classe doit avoir un attribut qui est une instance de la classe sur laquelle le contrat s'applique.

- Cet attribut doit avoir l'annotation « @Target ».
- La classe où le contrat s'applique doit être annotée par « @contractReference ».
- Les getters dans le contrat de classe doivent ignorer le retour de fonction à l'aide de l'instruction « return ignored() ».
- Les méthodes de type « void » et les getters de la classe où le contrat s'applique doivent être annotés par « @Pure ».
- Les méthodes du contrat de classe doivent avoir l'annotation « @Override » et avoir la même signature que la méthode dont elles héritent.

En annexe B se trouve le squelette de contrat de classe généré par GACTUS et aussi un modèle de fichiers de configuration de C4J.

4.3 Conception générale du plug-in GACTUS

4.3.1 Les commandes sous Eclipse RCP

La contribution d'un plug-in à Eclpise via l'interface graphique se fait à travers des commandes. En effet Eclipse, utilise le patron de conception « commande » pour définir un nouveau comportement pour son interface graphique. Une commande sous Eclipse déclare un nouveau comportement. Pour définir et utiliser une commande, on doit déclarer ces trois extensions qui sont :

- « org.eclipse.ui.command » : incluse pour déclarer la description du comportement de la commande.
- « org.eclipse.eclipse.handler » : cette extension sert de point de connexion entre la commande et les classes qui seront appelées lors de l'exécution de la commande. Le comportement de la commande est défini via cette extension. Lors de l'appel de la commande, c'est la classe du « handler » qui est exécutée.
- « org.eclipse.ui.menu » : utilisé pour définir l'emplacement et comment la commande sera incluse dans l'interface graphique.

L'ajout d'une extension se fait par le menu « extension » du fichier « plugin.xml » (figure 4.2). Pour ajouter une nouvelle extension, il faut cliquer sur le bouton « Add », une fenêtre apparait avec une liste d'extension. Il reste à choisir la bonne extension. Il est possible de filtrer les extensions en donnant un début du nom de l'extension dans la zone de filtre

de la fenêtre d'ajout de nouvelles extensions (figure 4.3). Une fois que l'extension a été ajoutée au projet, pour l'utiliser dans une classe, il faut l'importer à l'aide du mot clé Java « import » suivi du nom de l'extenion (exemple : `import org.eclipse.ui.*`).

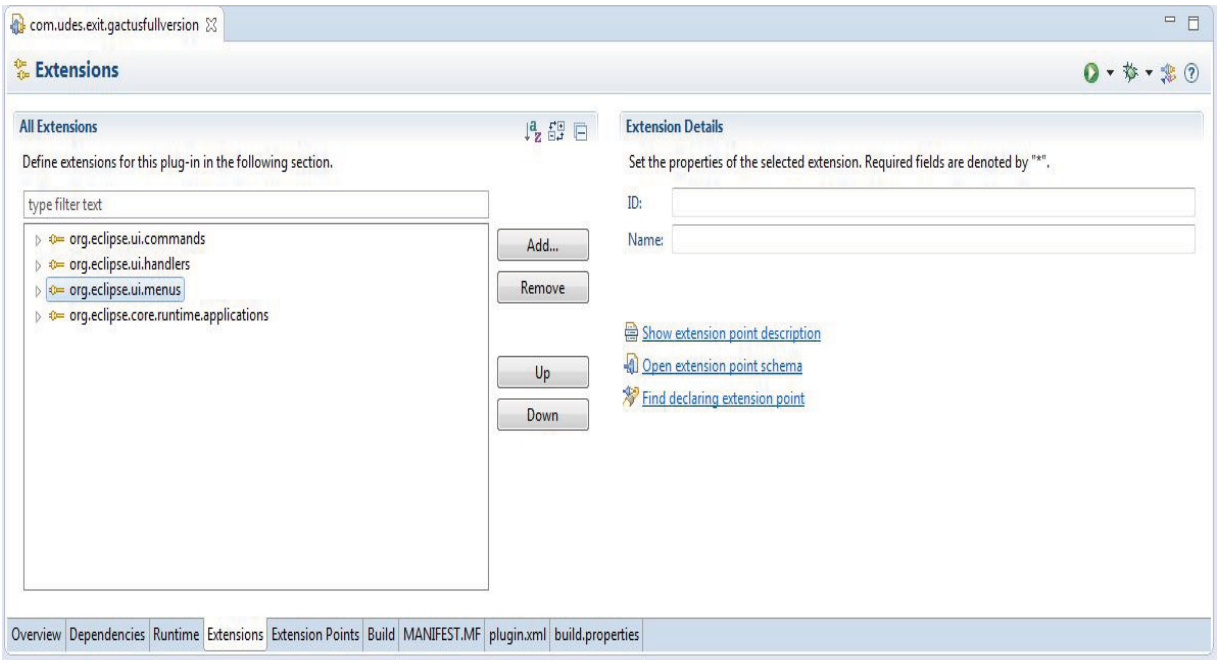

Figure 4.2 Menu extension du fichier plugin.xml

Les commandes sont regroupées par catégorie selon leurs apports. Le fonctionnement d'une commande peut se résumer à la figure 4.4.

GACTUS ajoute trois nouveaux comportements à l'interface graphique d'Eclipse. Ces trois comportements sont définis sous forme de commandes qui sont :

- « Generate Test and contract » : commande pour la génération du squelette des classes de test et le squelette de classe de contrat.
- « Generate Contract » : commande pour la génération unique du squelette de contrat de classe.
- « Generate Test » : commande pour la génération unique du squelette de classe de test.

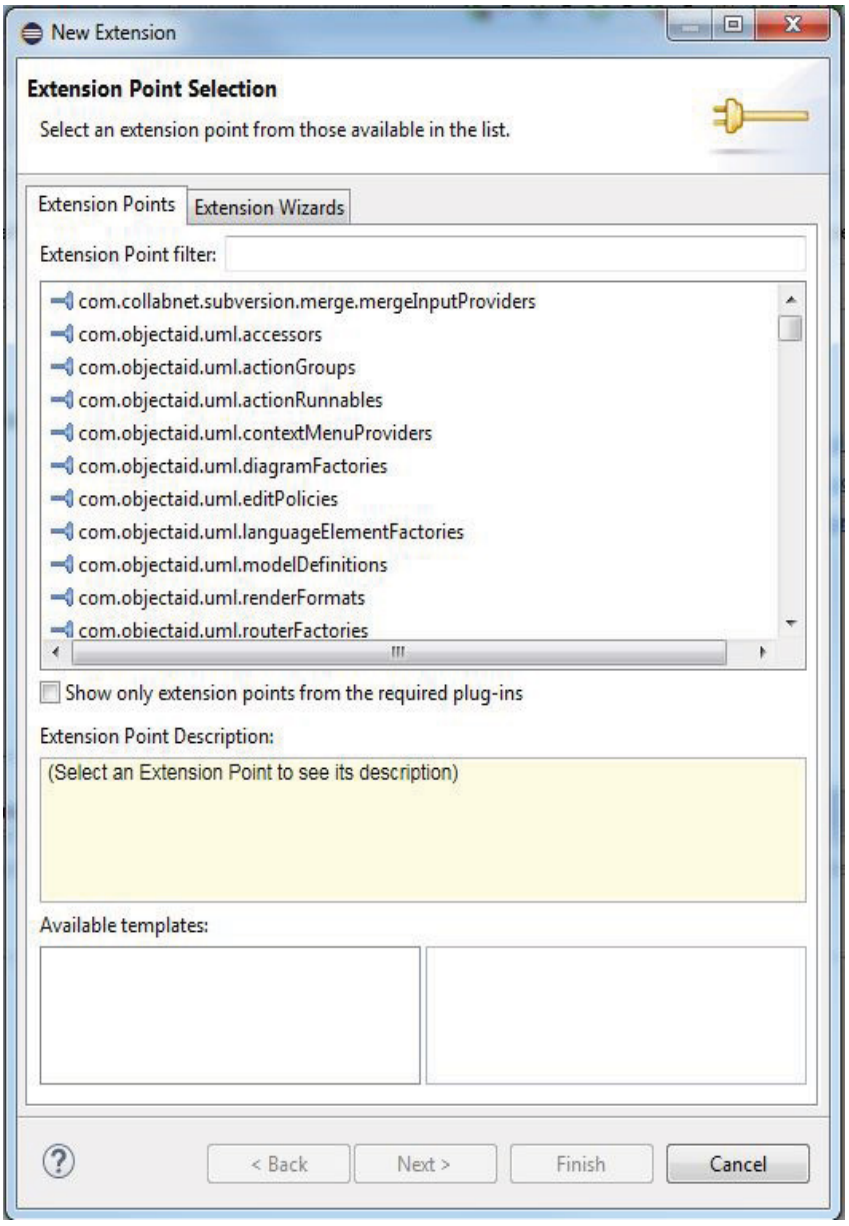

Figure 4.3 Fenêtre d'ajout d'une nouvelle extension

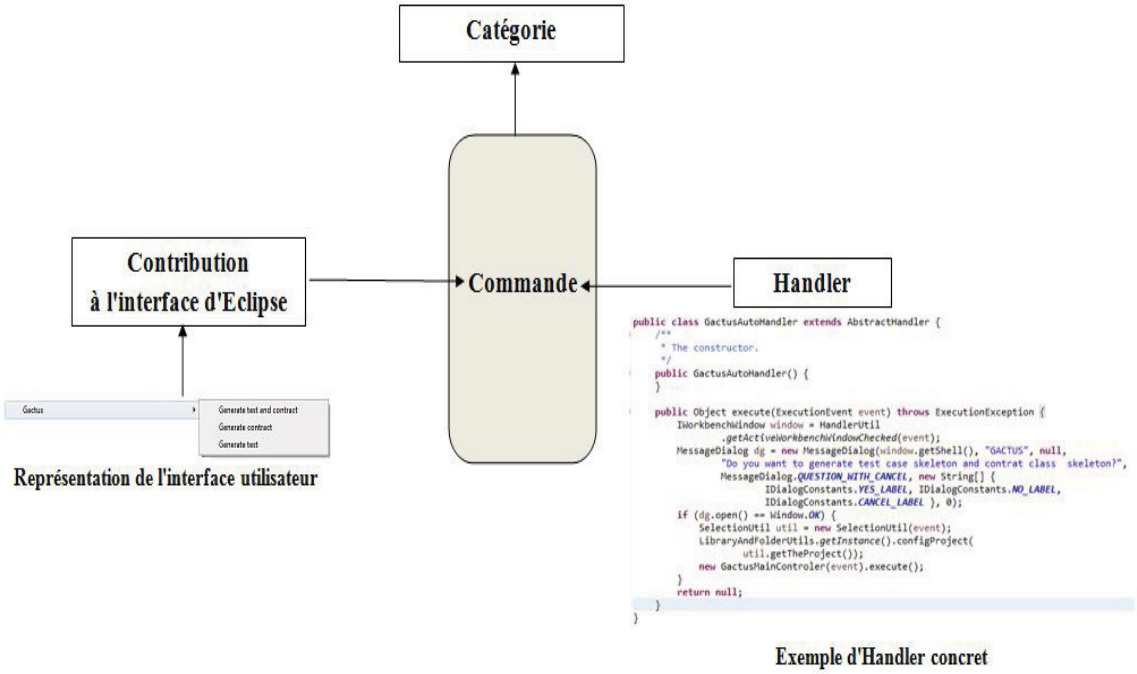

Figure 4.4 Description d'une commande

4.3.2 Description du scénario de génération de code source

D'une manière générale, le processus de la génération de squelette se fait selon le scénario suivant :

1. Clic droit sur l'interface graphique d'Eclipse précisément sur un projet (classe ou package) : cette action fait appel à l'extension « menu contribution » définie dans « org.eclipse.ui.menu » afin d'afficher le menu.

2. Clic sur l'un des sous-menu : cette action déclenchera la commande associée au sous-menu. Dans cette action c'est l'extension « org.eclipse.ui.command » qui est appelée.

3. Exécution du « handler » associé à la commande : dans cette action l'handler est exécuté à l'aide de sa méthode « execute ». Dans cette méthode, le contrôleur de GACTUS (classe GactusMainControler) est appelé. Ce dernier se charge d'appeler le modèle et la vue exacts pour la réalisation de l'action.

Les « handlers » associés aux différentes commandes de GACTUS sont les suivants :

- « GactusContratOnlyHandler » : cette classe (handler) est lancée lorsque la commande « Generate Contract » est activée, plus précisément quand l'utilisateur clique sur le sous-menu « Generate Contract ». Ce handler en s'exécutant va lancer la génération des squelettes des contrats de classe en utilisant la classe « GactusMainControler »

- « GactusTestOnlyHandlers » : représente le handler associé à la commande « Generate Test ». L'exécution de sa méthode « execute » entrainera la génération des squelettes des classes de tests.

- « GactusAutoHandler » : ce handler est associé à la commande « Generate Test and Contract ». Une action sur le sous-menu entrainera le lancement de ce handler. L'exécution de sa méthode « execute » permet la génération des squelettes des contrats de classes et celui des classes de tests.

Lors de la création d'un nouveau projet « Plug-in Project » avec le template « Hello, Words command » avec Eclipse RPC, un handler par défaut est généré. Le handler contient une méthode « execute ». Un modèle de classe handler du plug-in GACTUS se trouve en annexe C.

Le processus de génération de code source peut être résumé par la figure 4.5.

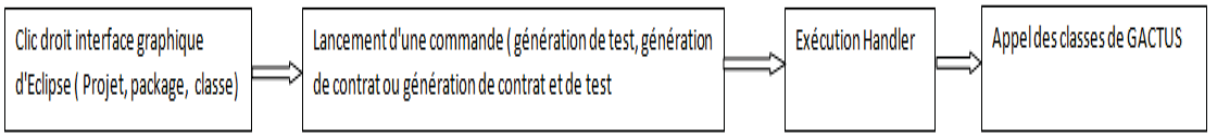

Figure 4.5 Scenario du processus de génération

Le diagramme de classe de la figure 4.6 est celui associé au processus de génération. Les classes se terminant par « Handler » sont les « handlers » et se trouvent dans le package « com.udes.gactus.handlers » du code source de GACTUS.

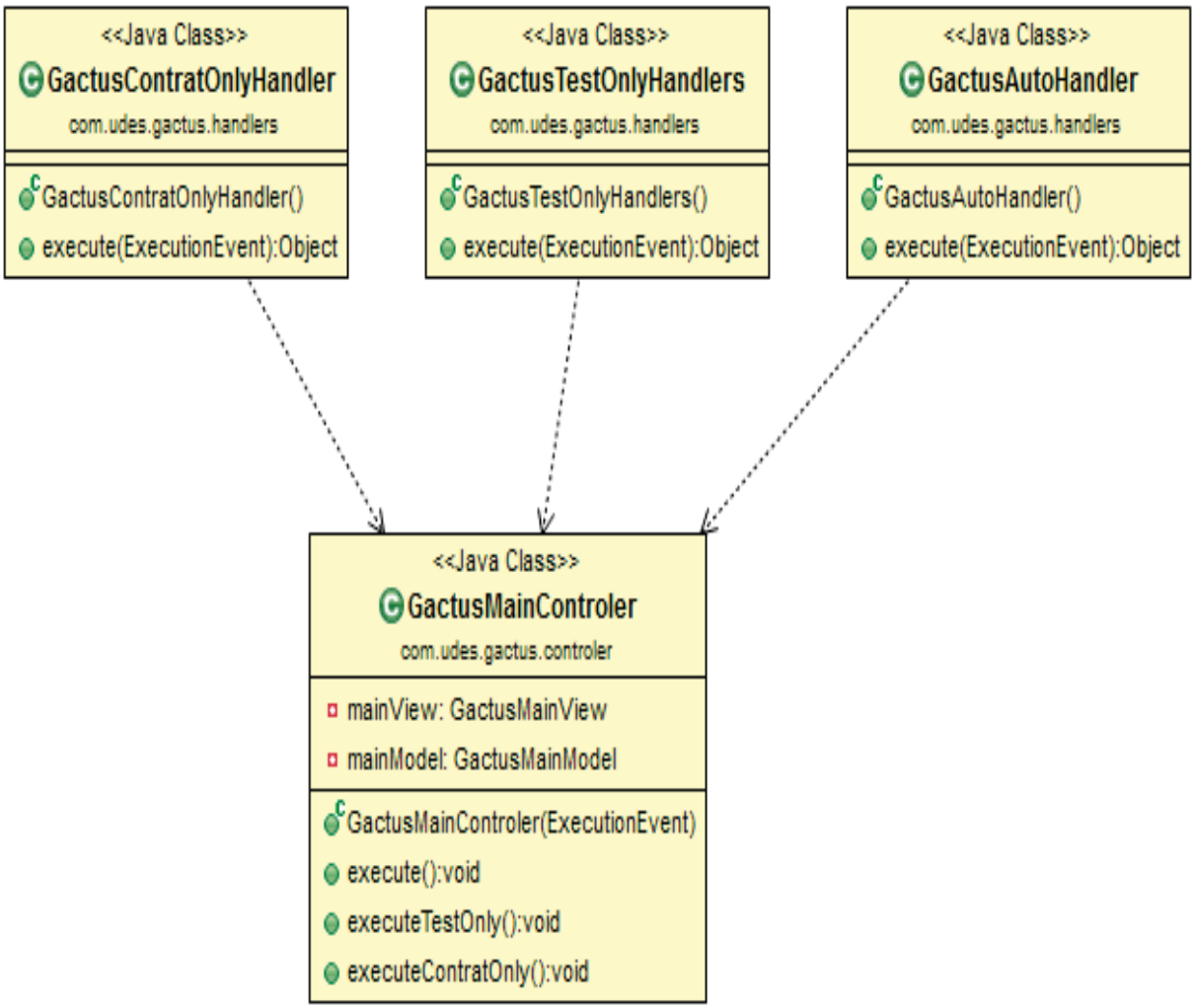

Figure 4.6 Diagramme de classe du scénario de génération

4.3.3 Description de l'architecture du plug-in (GACTUS)

Dans le code source du plug-in GACTUS (voir figure 4.7), le modèle MVC (Modèle Vue Controleur) est utilisé. Le choix de cette architecture s'explique par le fait qu'elle nous permet de séparer la logique de la présentation. Dans cette architecture, le modèle permet de gérer toute la partie logique du plug-in. Le modèle fait appel à l'API JDT d'Eclpise pour l'analyse ainsi que la génération des squelettes de classes de tests et des squelettes des contrats de classes.

La Vue permet de gérer l'interface utilisateur du plug-in. Elle utilise le plug-in JFace d'Eclipse. JFace est un plug-in encapsulant de nombreuses opérations de base qui facilitent le développement des interfaces graphiques. JFace repose sur SWT (Standard Widget Toolkit) qui est la bibliothèque de composants d'interface utilisateur standard utilisée par Eclipse et développée par IBM. Afin d'utiliser JFace, il a été ajouté comme packages parmi les packages dont le plug-in GACtUS depend.

Le module contrôleur quant à lui, en fonction de la requête faite par l'utilisateur, se chargera d'appeler le modèle et la vue adéquate pour la réalisation de la tâche demandée. Il est activé par les handlers.

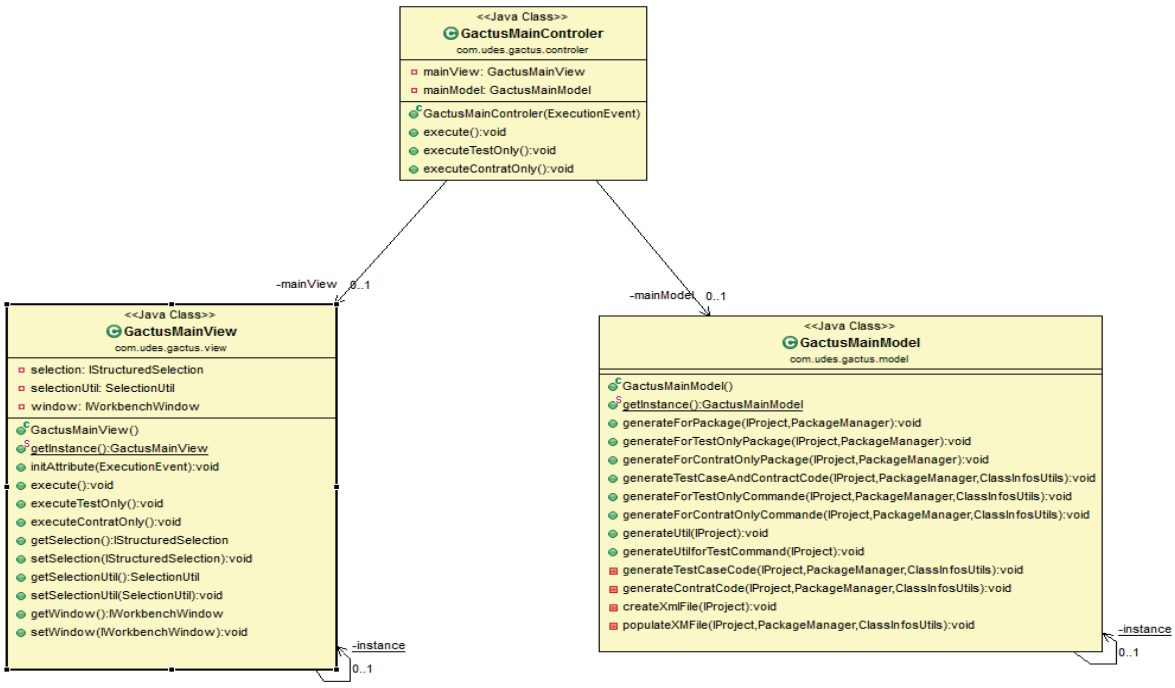

Figure 4.7 Architecture du plug-in GACTUS

4.4 Conception des fonctionnalités de GACTUS

4.4.1 Analyse d'un projet Java

Pour générer du code il faut faire une analyse afin trouver les informations telles que le nom des classes, la liste des packages, des méthodes... Dans le code source de GACTUS, l'analyse d'un projet Java est faite par les classes du package « com.udes.gactus.model.core ». L'analyse se fait en utilisant l'API JDT d'Eclipse. Le diagramme de classe de la figure 4.8 est celui des classes du package « com.udes.gactus.model.core ».

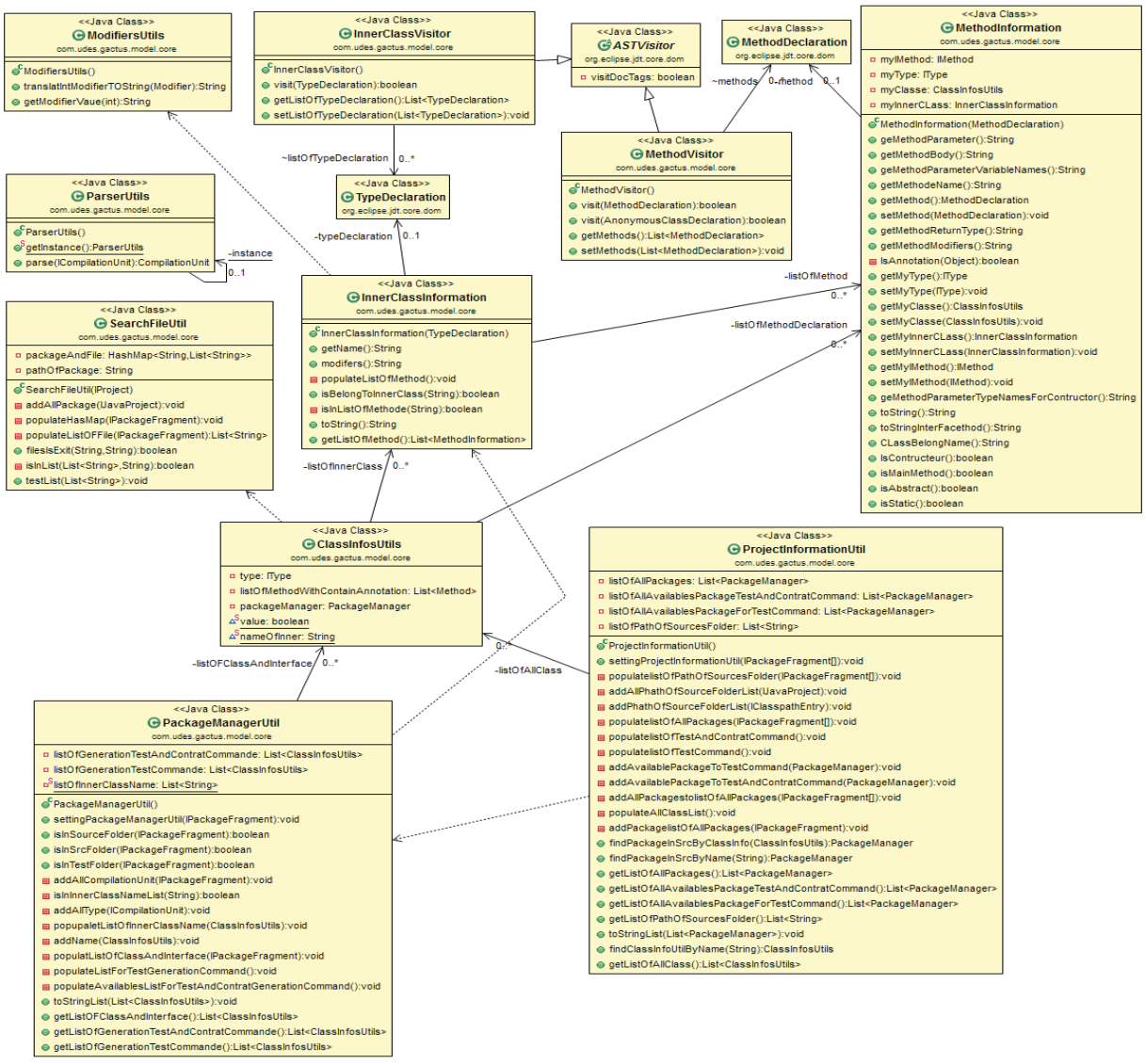

Figure 4.8 Diagramme de classes du package « com.udes.gactus.model.core »

Dans ce diagramme, les classes suivantes sont les plus importantes :

1. `ParserUTils` : cette classe définit une méthode qui permet de parser un fichier (classe Java). Elle prend une unité de compilation et construit l'AST associé à cette unité. L'unité de compilation représente une classe Java. La structure de cette classe est la suivante :

```
package com.udes.gactus.model.core;
import org.eclipse.jdt.core.*;
/**
 * @author Cheick Ismael MAIGA
 * @Dec: Parse a Java File
 */
public class ParserUtils {
   private static ParserUtils instance = null;
   public static ParserUtils getInstance() {
      if (instance == null) {
         instance = new ParserUtils();
      }
      return instance;
   }
/**
 * @param unit
 * @return CompilationUnit
 * @throws JavaModelException
 */
   public CompilationUnit parse(ICompilationUnit unit)
         throws JavaModelException {
      ASTParser parser = ASTParser.newParser(AST.JLS8);
      parser.setKind(ASTParser.K_COMPILATION_UNIT);
      parser.setSource(unit);
      parser.setResolveBindings(true);
      return (CompilationUnit) parser.createAST(null); // parse
   }
}
```

2. `MethodeVisitor` : cette classe héritant de la classe « `ASTVisitor` » de « JDT Core », son rôle est de visiter toutes les méthodes de la classe. Avec cette visite l'on peut avoir les informations suivantes sur une méthode :

   - Modificateur
   - Nom de la méthode
   - La liste des paramètres
   - Le type de retour
   - Corps de la méthode

```
package com.udes.gactus.model.core;

import org.eclipse.jdt.core.dom.*;
/**
 * @author Cheick Ismael MAIGA
 */
public class MethodVisitor extends ASTVisitor {
   List<MethodDeclaration> methods = new
       ArrayList<MethodDeclaration>();

   public boolean visit(MethodDeclaration node) {
      methods.add(node);
      return super.visit(node);
   }
   public boolean visit(AnonymousClassDeclaration
       anonyomousClassDeclaration) {
      return true;
   }
   public List<MethodDeclaration> getMethods() {
      return methods;
   }
   public void setMethods(List<MethodDeclaration> methods) {
      this.methods = methods;
   }
}
```

3. `InnerClassVisitor` : pour visiter une classe interne, nous avons conçu cette classe. Elle hérite aussi de la classe « `ASTVisitor` »

```
package com.udes.gactus.model.core;

import java.util.ArrayList;
import java.util.List;
import org.eclipse.jdt.core.dom.*;
/**
 * @author Cheick Ismael MAIGA
 */
public class InnerClassVisitor extends ASTVisitor {

   List<TypeDeclaration> listOfTypeDeclaration = new
       ArrayList<TypeDeclaration>();


   public boolean visit(TypeDeclaration node) {
      if (!node.isPackageMemberTypeDeclaration()) {
         listOfTypeDeclaration.add(node);
      }
      return super.visit(node);
   }


   public List<TypeDeclaration> getListOfTypeDeclaration() {
      return listOfTypeDeclaration;
   }


   public void setListOfTypeDeclaration(
         List<TypeDeclaration> listOfTypeDeclaration) {
      this.listOfTypeDeclaration = listOfTypeDeclaration;
   }
}
```

Le principe de l'analyse du code avec le parseur et l'ASTVisitor peut se résumer avec la figure 4.9

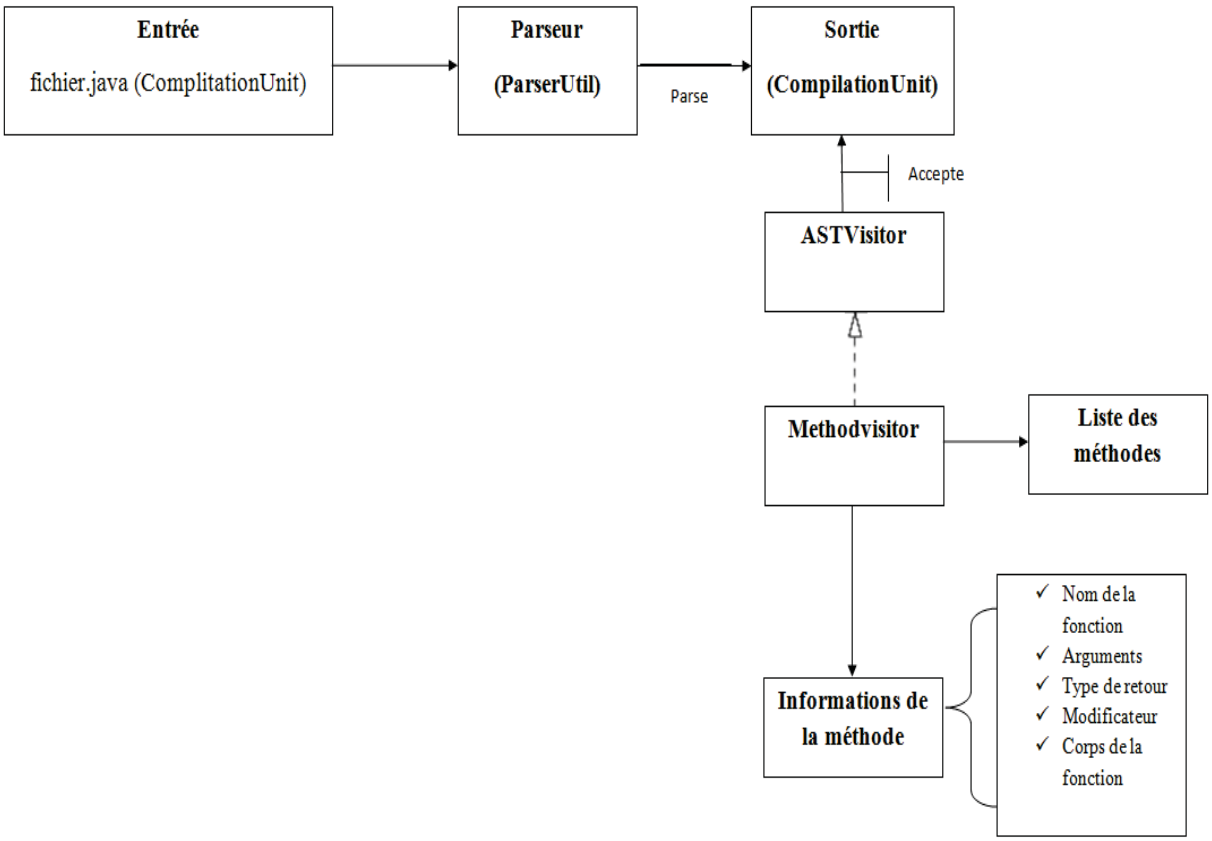

Figure 4.9 Description parseur et visiteur de code source

4.4.2 Conception de la fonctionnalité de génération de squelette de code source

Le scénario de génération de squelette de classe est la suivante : lorsqu'on reçoit une requête de génération, la classe « Generate » du package « com.udes.gactus.model.classandinterface » instancie un objet de type « GenerateContratClass » ou « GenerateTestCas » selon le type de la génération voulu. Ensuite, il appelle la méthode « generate ». Cette méthode fait la génération du squelette. La génération d'un contrat de classe diffère un peu de celle d'une classe de test. En effet pour un contrat, la génération peut être celle d'une interface ou d'une classe par contre pour le cas d'une classe de test ; la génération est uniquement pour les classes. Le diagramme de classe suivant est celui des classes permettant de faire la génération du squelette de classe. Ces classes appartiennent aux packages « com.udes.gactus.model.generatecontrat », « com.udes.gactus.model.generatetest »,

« com.udes.gactus.model.classandinterface ».

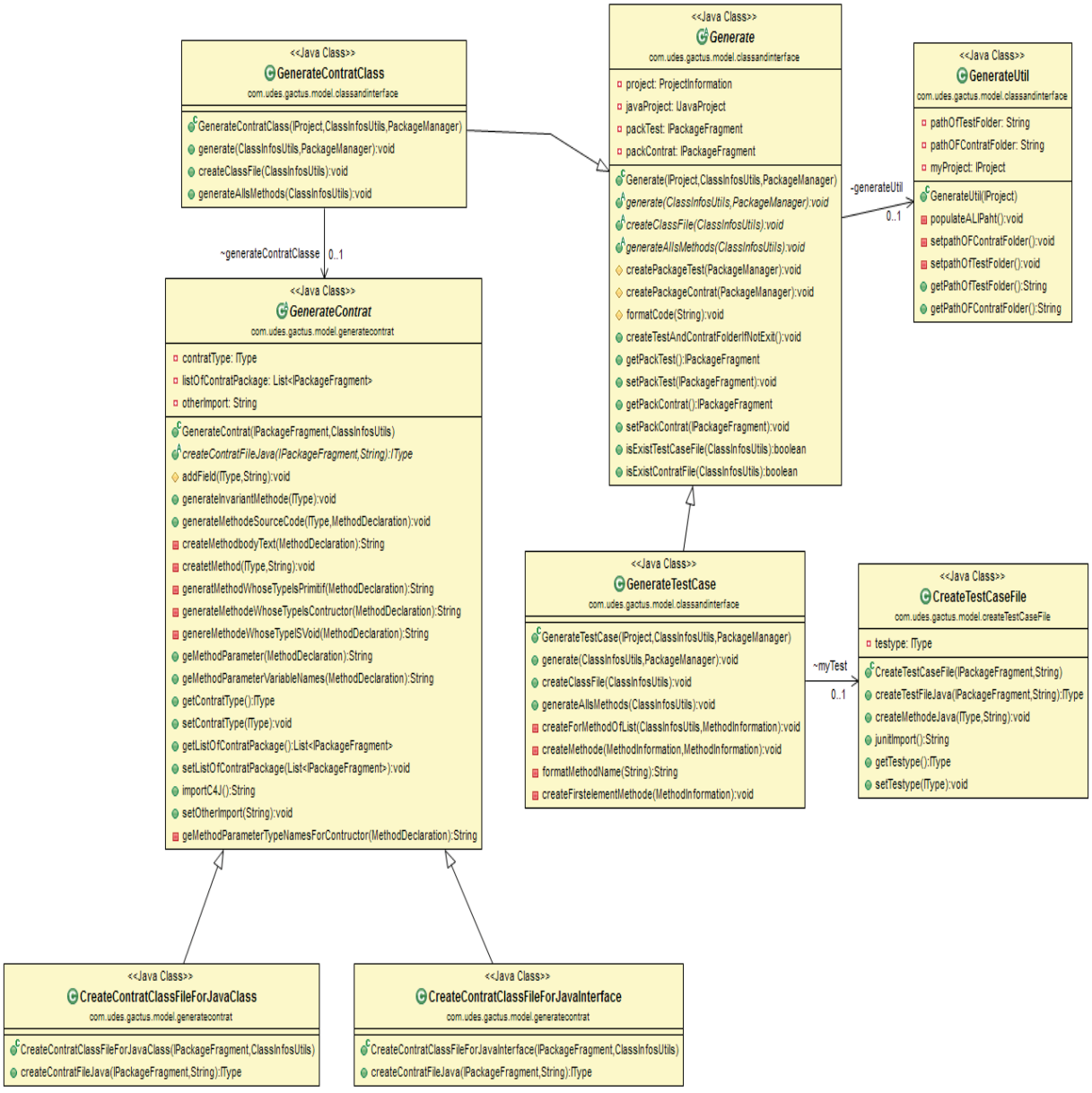

Figure 4.10 Diagramme de classe de la fonctionnalité de génération

Description du scénario de génération de classe de test

Les classes concernées par la génération de squelette de classe de test sont « `GenerateTestCase` » et « `CreateTestCaseFile` ». La classe « `GenerateTestCase` » hérite de la classe « `Generate` ». Elle redéfinit la méthode « `generate` ». Dans cette méthode on fait appel à la méthode « `createPackageTest` » qui est chargée de créer le package de la classe de test et aussi la

méthode « createClassFile ». Dans la méthode « createClassFile », nous créons le fichier Java c'est-à-dire fichier avec l'extension « .java » grâce à la création d'un objet de type « CreateTestCaseFile ». Une instanciation de la classe « CreateTestCaseFile » crée un fichier Java vide sans méthode, mais qui contient les imports de package de Junit (Les imports de package permettent d'utiliser les méthodes définies dans JUnit). Une fois le fichier de test crée, nous créons tous les cas de tests à l'aide de la méthode « generateAllMethode ». Pour créer un nouveau cas de méthode, nous faisons appel à la méthode « createMethodeJava » de la classe « CreateTestCaseFile ». Cette méthode prend deux paramètres : le nom de la méthode et le fichier Java dans lequel la méthode devrait être créée. À l'aide d'une chaine de caractère, nous créons la structure de la méthode. Pour ajouter la méthode dans le fichier Java, nous utilisons la méthode « createMethod » définie par la classe « IType » de l'api « core » de JDT. La structure de la méthode « createMethodeJava » est la suivante :

```
public void createMethodeJava(IType type, String methodeName)
      throws JavaModelException {
   String nameofMethodeConcatTest = methodeName + "()";
   String text = "@Test \n public void " + nameofMethodeConcatTest
         + "{"
         + "\t\t// TODO: add your test case code here \n"
         + " \n\t\t fail(\"Not yet implemented\")";
         + \n}";
   if (type != null)
      type.createMethod(text, null, true, null);
}
```

Le scénario de génération d'une classe de test peut être résumé à l'aide de l'algorithme 1

Description du scénario de génération de squelette de contrat de classe

Le principe de la génération de squelette de contrat de classe est similaire à celui d'une classe de test. À la seule différence que dans le cas de la génération du squelette de contrat de classe, la génération peut-être celle d'une interface ou celle d'une classe. Dans le cas d'une interface, le contrat de classe devrait implémenter l'interface et celui d'une classe, il devrait étendre la classe. Selon le cas, c'est la classe « ContratClassFileForJavaClass » ou « ContratClassFileForJavaInterface ». Comme pour la classe de test, la classe « GenerateContratClass » héritant de la classe « Generate » redéfinit la méthode « generate » qui se chargera d'appeler les méthodes « createPakageContrat »

« `createClassFile` » pour création du package du contrat et la création du fichier Java. Mais contrairement à la génération de la classe de test, la méthode « `createClassFile` » va vérifier si la génération se fait pour une classe ou une interface. Le code source de cette méthode est la suivante :

```
@Override
   public void createClassFile(ClassInfosUtils myClass) {
      if (myClass.isClass() == true) {
         generateContratClasse = new CreateContratClassFileForJavaClass(
               getPackContrat(), myClass);
         generateAllsMethods(myClass);
      }
      if (myClass.isInterface() == true) {
         generateContratClasse = new
            CreateContratClassFileForJavaInterface(
              getPackContrat(), myClass);
         generateAllsMethods(myClass);
      }
   }
```

Result: Scénario de génération de squelette de classe de test lors de l'appel de la méthode « generate »
Début;
appel de la méthode « `createPackageTest` » ▷ cette méthode crée un package qui contiendra la classe de test;
appel de la méthode « `createClassFile` » ▷ cette méthode va créer le fichier java;
while *Liste des méthodes non vides* **do**
  lire la méthode courante;
  if *méthode non vides* **then**
    appel de la méthode « createMethode »;
  else
    passer à la méthode suivante;
  end
end
Fin

Algorithm 1: Algorithme de la génération de squelette de classe de test

Conception de l'interface utilisateur

Pour faciliter l'utilisation de GACTUS, l'interface graphique est simple d'utilisation. L'interface graphique est un menu comportant trois sous-menus. Les trois sous-menus exécutent chacune une des fonctionnalités proposées par GACTUS. Les classes gérant la vue se trouvent dans le package « com.udes.gactus.view ». Ces classes utilisent l'API « JFace » et l'API « UI » d'Eclipse. La classe « `SelectionUtil` » à l'aide de ses méthodes « `config` » et « `setData` » permet d'avoir les informations de l'action de l'utilisateur. Ces informations seront transmises à la classe « `MainView` » qui se chargera d'appeler la bonne vue pour le traitement de la requête. L'appel d'une vue se fait à l'aide de l'une des méthodes « `execute` », « `executeTestOnly` », « `executeContratOnly` ». Chacune de ces méthodes est une instance un objet de type « `ExecuteTestAndContratCommandUtil` », « `ExecuteContratCommandUtil` », « `ExecuteContratCommandUtil` ». Chacune de ces classes définit des méthodes permettant l'exécution d'un type de génération. L'architecture de GACTUS se reposant sur le design MVC, l'exécution se traduit par un message envoyé au contrôleur qui se chargera d'appeler le modèle pour la réalisation. Le diagramme de classe (figure 4.11) suivant est celui des classes de la vue.

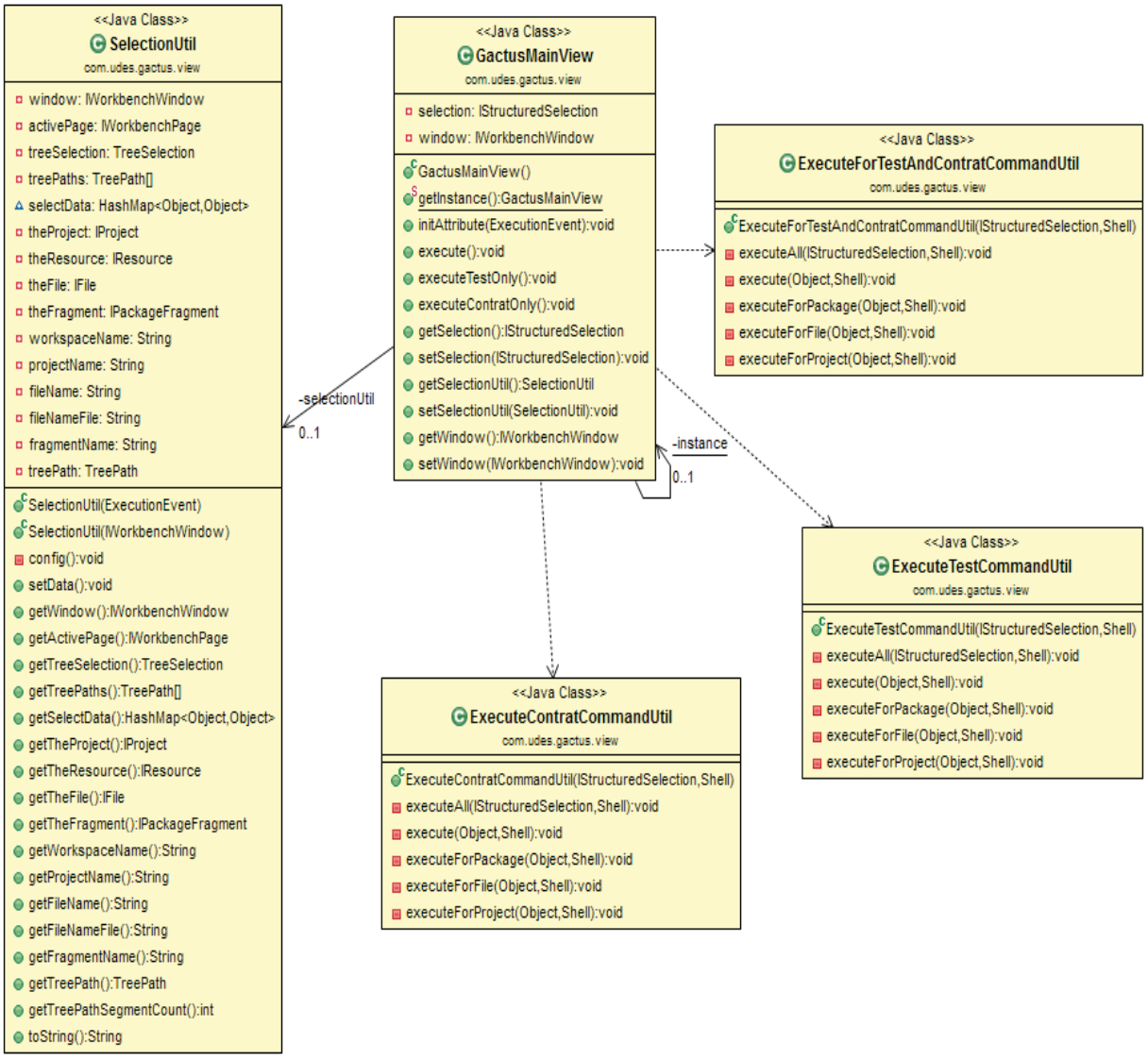

Figure 4.11 Diagramme de classe de la vue

4.5 Description des tests unitaires

4.5.1 Description du fonctionnement de l'exécution d'une classe de test sous Eclipse RCP

Sur Eclipse RCP, l'exécution d'une classe de test unitaire se fait à l'aide du sous-menu « JUnit Plugin Test » du menu « Run as » après un clic droit sur la classe de test unitaire. À chaque exécution, une nouvelle instance d'Eclipse RCP se lance avec la création d'un nouveau répertoire de travail nommé « junit-workspace ». La nouvelle instance d'Eclipse RCP se ferme après quelques secondes d'exécution. En plus de cela, elle ne tient pas compte des projets existant dans le répertoire de travail « junit-workspace ». Cela veut dire que même si on copie un projet existant dans le répertoire « junit-workspace », au lancement de la nouvelle instance, ce projet ne sera pas visible dans l'explorateur de projets. Aussi, vu que l'instance se referme seule, il n'est pas possible d'importer un projet existant dans son répertoire de travail, ni crée un nouveau projet à partir de l'assistance de création de projets. Cela constitue un grand handicap, pour exécuter les classes de tests unitaires de GACTUS. En effet, pour exécuter GACTUS, l'on doit avoir accès à l'un projet disponible dans l'explorateur de projet.

4.5.2 Solution pour exécuter les tests unitaires de Plug-in Eclipse

Pour résoudre le problème d'exécution des classes de tests unitaires, nous avons procédé de la manière suivante :

1. Créer un dossier nommé « F4Test » dans laquelle nous avons copié un projet Java.
2. Créer une classe nommée « ConfigureTestUtil » : dans cette classe nous utilisons l'API JDT pour créer un espace de travail Eclipse. Dans cet espace de travail, nous créons un nouveau projet Java en l'initialisant avec le projet copié préalablement dans le dossier « F4Test ». La structure de la classe « ConfigureTestUtil » est la suivante :

```
public class ConfigureTestUtil {
   private IProject project;
   private static ConfigureTestUtil instance = null;
   private ProjectInformation projectInformation;
   private ConfigureTestUtil() {
      if (instance == null) {
```

```
            instance = new ConfigureTestUtil();
            initSetup();
        }
    }
    public static ConfigureTestUtil getInstance() {
        return instance;
    }
    private void initSetup() {
        Bundle bundle = Activator.getDefault().getBundle();
        IPath path = new Path("F4Test/underDev/projetForTest");
        URL setupUrl = FileLocator.find(bundle, path,
            Collections.EMPTY_MAP);
        File setupFile;
        try {
            setupFile = new
                File(FileLocator.toFileURL(setupUrl).toURI());
            String baseDirectory = setupFile.getAbsolutePath();
            String projectName = "projetForTest";
            IWorkspace workspace = ResourcesPlugin.getWorkspace();
            IWorkspaceRoot root = workspace.getRoot();
            importProject(baseDirectory, projectName);
            IProject[] projectList = root.getProjects();
            project = projectList[projectList.length - 1];
            projectInformation = new ProjectInformation(project);
        } catch (URISyntaxException | IOException e | CoreException e)
            {
            System.out.println(e.getMessage());
            e.printStackTrace();
        }
    }
    private static void importProject(final String baseDirectory,
            final String projectName) throws CoreException {
        IProjectDescription description =
            ResourcesPlugin.getWorkspace()
                .loadProjectDescription(new Path(baseDirectory +
                    "/.project"));
```

```
        IProject project = ResourcesPlugin.getWorkspace().getRoot()
              .getProject(description.getName());
        project.create(description, null);
        project.open(null);
    }
}
```

4.5.3 Description des tests unitaires du code source de GACTUS

L'ensemble des classes de tests se trouve dans le dossier « tests » et l'exécution de l'ensemble des tests se fait dans la classe « AllTests ». Cette classe est une suite de test. Le taux de couverture des classes du module modèle est de 86,2% (voir figure 4.12). Il faut noter que nous avons dû désactiver certains cas de tests surtout pour la génération des codes sources. En effet il n'est pas possible d'exécuter tous les cas de génération en même temps vu que l'utilisateur devrait toujours faire un choix parmi les trois options qui lui sont offerts. Lors de l'exécution de la classe de tests nous pouvons soit exécuter le cas de test lançant la commande de génération de contrat de classe et celui des tests unitaires ou la commande lançant uniquement la génération des contrats de classes et la commande lançant la génération des tests unitaires.

Nous n'avons pas pu tester les classes de la vue, car l'interface graphique de GACTUS se lance via le menu contextuel et cela à partir de l'explorateur de projet d'Eclipse. Des modèles de quelques classes de tests unitaires du code source de GACTUS se trouvent dans l'annexe D.

4.6 Description des contrats de classes

Pour assurer la correction de GACTUS, nous avons élaboré des contrats sur les principales classes. Environ 70% des classes du code source de GACTUS ont des contrats de classes. Les contrats de classes du code source de GACTUS sont sous le format C4J et sont présents dans le dossier « contracts » du projet GACTUS. Les spécifications introduites dans les contrats sont celles définies par les contraintes de la section « Anaylse ». Afin de vérifier le bon fonctionnement des contrats de classes, dans les classes de tests nous avons écrit des cas de tests ne respectant pas les spécifications. Lors de l'exécution, nous avons constaté que ces cas de tests ne passaient pas les tests. Un modèle de contrat de classe du logiciel GACTUS est donné dans l'annexe E.

tests	86,2 %	1 616	258	1 874
com.udes.gactus.model	70,8 %	364	150	514
GactusautoProjectModelTest.java	54,2 %	117	99	216
GactusautoPackageModelTest.java	74,6 %	129	44	173
GacutAutoClassAndInterfaceModelTest.j	90,2 %	37	4	41
XMLPopulateUtilTest.java	96,4 %	81	3	84
com.udes.gactus.test.util	65,9 %	137	71	208
com.udes.gactus.mdole.core	96,4 %	715	27	742
com.udes.gactus.model.generatexml	96,6 %	112	4	116
com.udes.gactus.main.test	0,0 %	0	3	3
com.udes.gactus.model.classandinterface	96,1 %	73	3	76
com.udes.gactus.analyse	100,0 %	26	0	26
com.udes.gactus.util	100,0 %	189	0	189

Figure 4.12 Couverture du code des classses de tests

4.7 Analyse des métriques du code source du logiciel GACTUS

Les métriques du plug-in GACTUS sont définies par la figure 4.13. Dans cette figure, on peut remarquer que le nombre total de lignes de code est 6496, le nombre de classes est 74. Le nombre total de paquets est 21. *À noter que l'avertissement de la complexité cyclomatique de McCabe est dû à une utilisation d'une classe de l'API JDT.*

4.8 Utilisation de GACTUS

GACTUS dispose d'une interface graphique très simple d'utilisation. Pour utiliser GACTUS, sélectionnez le projet ou le package ou le fichier (.java) dont vous voulez générer le contrat de classe ou le test unitaire ou les deux, puis faites menu contextuel (clic droit). Dans la liste vous verrez GACTUS (figure 4.15).

Il vous reste à choisir votre type de génération. Selon le type de choix, une boite de dialogue comme celle de la figure 4.15 suivante devrait apparaître, vous demandant de confirmer la génération ou d'annuler.

Metric	Total	Mean	Std. Dev.	Maxim...	Resource causing Max...	Method
Number of Parameters (avg/max per method)		0,571	0,827	4	/ca.udes.exit.gactusaut...	getconfirgurationElements
Number of Static Attributes (avg/max per type)	17	0,23	0,508	2	/ca.udes.exit.gactusaut...	
Efferent Coupling (avg/max per packageFragment)		3,476	2,771	12	/ca.udes.exit.gactusaut...	
Specialization Index (avg/max per type)		0,036	0,216	1,81	/ca.udes.exit.gactusaut...	
Number of Classes (avg/max per packageFragment)	74	3,524	2,839	12	/ca.udes.exit.gactusaut...	
Number of Attributes (avg/max per type)	158	2,135	2,663	15	/ca.udes.exit.gactusaut...	
Abstractness (avg/max per packageFragment)		0,041	0,104	0,333	/ca.udes.exit.gactusaut...	
Normalized Distance (avg/max per packageFragment)		0,328	0,263	0,81	/ca.udes.exit.gactusaut...	
Number of Static Methods (avg/max per type)	18	0,243	0,515	2	/ca.udes.exit.gactusaut...	
Number of Interfaces (avg/max per packageFragment)	0	0	0	0	/ca.udes.exit.gactusaut...	
Total Lines of Code	6496					
Weighted methods per Class (avg/max per type)	1225	16,554	21,374	164	/ca.udes.exit.gactusaut...	
Number of Methods (avg/max per type)	651	8,797	8,885	50	/ca.udes.exit.gactusaut...	
Depth of Inheritance Tree (avg/max per type)		1,23	0,534	3	/ca.udes.exit.gactusaut...	
Number of Packages	21					
Instability (avg/max per packageFragment)		0,649	0,247	1	/ca.udes.exit.gactusaut...	
McCabe Cyclomatic Complexity (avg/max per method)		1,831	1,633	13	/ca.udes.exit.gactusaut...	getStatusForContratCase
Nested Block Depth (avg/max per method)		1,407	0,687	5	/ca.udes.exit.gactusaut...	PopulateLocalXML
Lack of Cohesion of Methods (avg/max per type)		0,287	0,321	0,937	/ca.udes.exit.gactusaut...	
Method Lines of Code (avg/max per method)	3893	5,819	8,522	138	/ca.udes.exit.gactusaut...	createC4JPureRegistryX...
Number of Overridden Methods (avg/max per type)	27	0,365	2,203	19	/ca.udes.exit.gactusaut...	
Afferent Coupling (avg/max per packageFragment)		4,048	7,786	34	/ca.udes.exit.gactusaut...	
Number of Children (avg/max per type)	7	0,095	0,408	2	/ca.udes.exit.gactusaut...	

Figure 4.13 Métriques du plug-in GACTUS

Si vous confirmez la génération, quelques secondes après vous devrez voir la structure de votre projet initial modifier avec la création d'un dossier « test », un dossier « contrat », un dossier « ressource ».

Le scénario suivant décrit comment se fait la génération d'un contrat de classe.

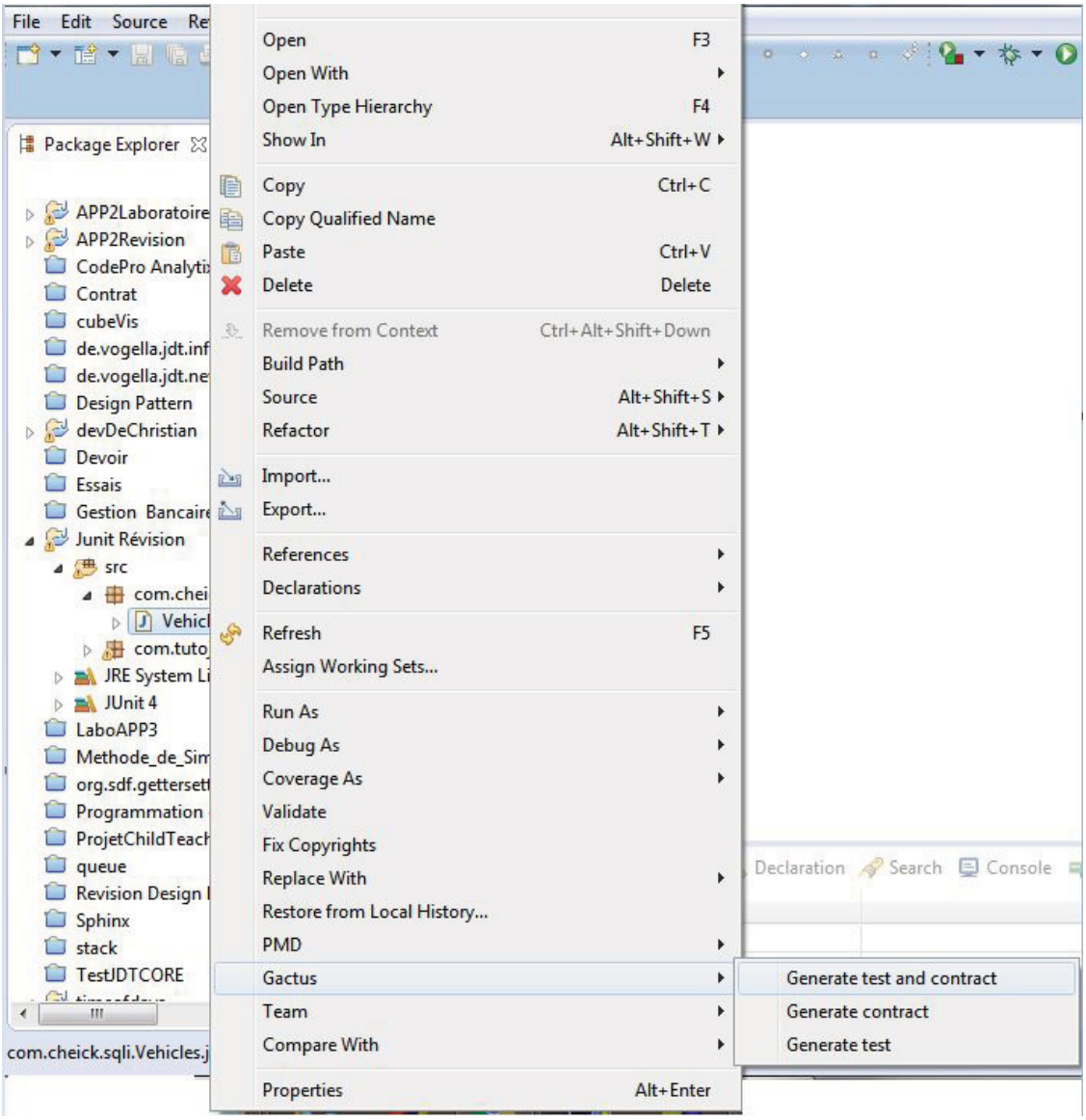

Figure 4.14 Vue de GACTUS et de ses sous-menus

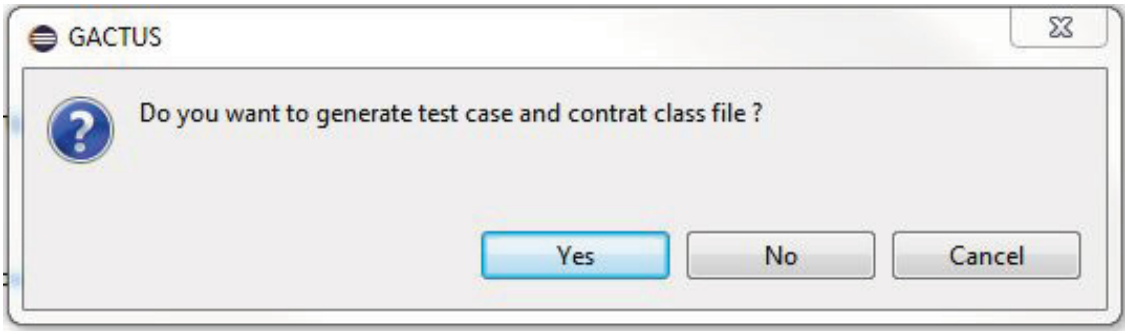

Figure 4.15 Boîte de dialogue de confirmation de la génération

CHAPITRE 5
DISCUSSION

5.1 Les bonnes décisions

Pour ce projet de recherche, plusieurs décisions ont été prises pour effectuer la recherche. Parmi les décisions, celles ayant eu plus d'effets positifs sur la recherche sont les suivantes :

- Le choix du langage Java : Java fait des langages de programmation les plus utilisés pour la réalisation des logiciels selon le classement Tiobe 2016 [29]. Vu sa popularité, ce fut une bonne décision de réaliser le logiciel pour ce langage de programmation, car le plug-in pourrait intéresser une grande communauté de développeurs. En plus, le logiciel étant un plug-in il s'intègre facilement aux différents environnements de développement.

- Le choix de JUnit : JUnit est connu comme étant la meilleure bibliothèque gratuite pour faire des tests unitaires en Java. Cela a fait de sorte qu'il soit d'office intégré dans la plupart des environnements de développements Java.

- Le choix de C4J : C4J est l'une des plus récentes bibliothèques permettant de faire la programmation par contrat. C4J propose une documentation bien détaillée facilitant son utilisation et son apprentissage. De plus grâce aux différents fichiers de configuration et de surcroit en XML il est plus facile d'activer ou de désactiver un contrat de classe.

- L'approche de développement du logiciel GACTUS : pour la réalisation du plugin, nous avons procéder par la réalisation d'un plugin simple afin de comprendre le fonctionnement et le développement d'un plug-in d'une manière générale. De même que la compréhension de l'API JDT. Cela a beaucoup facilité la réalisation des fonctionnalités de GACTUS.

5.2 Discussion sur la programmation par contrat

Avant d'entamer la discussion, il est primordial de définir les termes erreur, bogue et panne qui sont qualifiés de **malheurs logiciels** par Bertrand Meyer. Il donne la définition suivante à ces termes :

- Une erreur est une décision incorrecte prise durant le développement d'un système logiciel.

- Un défaut est une propriété d'un système logiciel qui peut induire le système à s'écarter du comportement attendu.

- Une panne est un événement se produisant dans un système logiciel et qui écarte celui-ci du comportement attendu lors d'une exécution.

Il en résulte de ces définitions, la relation causale suivante : « *Les pannes sont dues aux défauts, qui résultent d'erreurs* » [25]. En partant de cette relation causale, on peut conclure que l'*erreur* est la base des malheurs logiciels c'est-à-dire les décisions incorrectes prises durant le développement.

Le principe de la programmation par contrat est de s'assurer de la correction du programme grâce aux spécifications que l'on introduit. Une spécification n'est entre autres qu'une propriété qui doit être vérifiée. Un exemple de spécification serait que la racine carrée d'un nombre soit toujours positif. Ainsi pour un programme qui calcule la racine carrée d'un nombre, si le résultat est négatif alors dans ce cas on pourrait dire que le programme n'est pas correct, car la spécification n'est pas respectée. À l'aide d'un contrat, il serait possible de déterminer le responsable de l'erreur d'après les principes d'obligations et de bénéfices décrits par Bertrand Meyer [25].

Une spécification peut être vue comme une décision. Cette décision est prise soit par plusieurs personnes, dont le développeur, soit par le spécialiste d'analyse. Si une spécification est une décision et la décision est prise par un être humain, alors nous ne sommes pas à l'abri des erreurs donc pas à l'abri des pannes de logiciel. Dans le cadre de cette recherche, nous nous intéressons plus à l'écriture du code une fois la spécification déterminée c'est-à-dire la traduction de la spécification en code source.

L'écriture d'un code source demande des prises de décisions face à certaines situations. Donc pendant l'écriture du code source, si une mauvaise décision est prise, une erreur interviendrait et engendrerait alors une panne dans le logiciel. Comment s'assurer que le code des contrats de classe écrit par le développeur soit sans erreurs ? La seule manière pour s'assurer qu'un code source est fonctionne bien c'est de faire des tests unitaires. Ainsi l'on pourrait se demander s'il faut écrire des tests unitaires aussi pour les contrats de classe ?

5.3 Discussion : Est-ce qu'il faut tester unitairement les contrats de classe

D'une manière générale, il est conseillé de tester unitairement tout code source d'un programme. Ainsi donc les contrats de classes étant de code source, idéalement il serait important d'écrire des tests unitaires pour ce code source. Tester le code source des contrats de classe nous permet d'assurer le bon fonctionnement de la classe de contrat. En effet une erreur dans l'un des contrats des méthodes de la classe peut entraîner un mauvais fonctionnement du logiciel même si des tests ont été faits. Il faudrait donc tester les spécifications des contrats des méthodes de la classe contrat.

Cependant, il est impossible de tester une classe unitaire pour une classe de contrat de classe C4J. Le contrat de classe par défaut implémente ou étend une classe ou une interface. Le contrat de classe ne redéfinit pas les fonctionnalités des classes dont elle hérite ce qui fait que le test de la classe mère suffit. De plus, dans un contrat de classe les valeurs de retours des fonctions héritées sont ignorées. Dans les contrats des méthodes on introduit des conditions, ceux qui ne modifient pas les données de la classe mère. Ainsi lorsqu'on écrit une classe de test pour une classe mère, la classe de test s'applique aussi à celui du contrat de classe. De plus l'évalution se fait d'abord par celui des contrats de classe avant celle de la classe mère. Il n'est pas nécéssaire d'écrire une classe de test pour un contrat de la classe.

CHAPITRE 6

CONCLUSION

6.1 Récapitulatif

Le challenge pour les développeurs de logiciels n'est pas uniquement de savoir coder, mais de s'assurer que les logiciels qu'ils créent sont de meilleure qualité. En effet, le logiciel est devenu omniprésent dans nos vies si bien qu'une erreur dans le logiciel peut causer d'énormes catastrophes.

Vu l'importance du logiciel et sa forte omniprésence dans notre quotidien, ce projet de recherche avait pour but de proposer un outil aux développeurs de logiciels plus précisément celui de Java afin que ces derniers puissent développer des logiciels de meilleure qualité. Les objectifs principaux de ce mémoire ont été enumérés au chapitre 1.

La qualité dans le logiciel peut être vue sous plusieurs angles. Ainsi plusieurs définitions ont été proposées. Afin de bien appréhender la notion de la **qualité du logiciel**, nous avons élaboré le chapitre 2 à cet effet.

Au chapitre 3, les principaux concepts clés tels que les tests unitaires, la programmation par contrat et l'analyse du code source d'un programment ont été traités.

Les tests unitaires permettent aux développeurs d'assurer que le logiciel ne contient pas de potentiel bogue, par contre la programmation par contrat lui permet de s'assurer que le logiciel est correct c'est-à-dire ne fait plus ni moins que ce qu'on lui a demandé de faire. C'est une propriété primordiale qu'un logiciel devrait avoir. Le chapitre 4 montre comment s'y prendre pour faire des tests unitaires ou de la programmation par contrat voir les deux lors du développement d'un logiciel.

Le chapitre 5 décrit le développement du logiciel GACTUS.

6.2 Contribution

Nous proposons un logiciel innovant qui aide à produire un logiciel correct et fiable donc de meilleure qualité. En effet la correction et la fiabilité du logiciel sont les facteurs primor-

diaux pour un logiciel. Le logiciel que nous avons réalisé possède les propriétés suivantes faisant de lui un outil innovant :

- Facile d'installation : étant un plug-in il s'installe facilement sur les environnements de développement tel que Eclipse.

- Gain de temps : le gain en temps peut se voir à deux niveaux. Tout d'abord le plug-in génère du code source donc le développeur n'a plus qu'à compléter le code pour ses tests et ses contrats. En plus cela le plug-in fait une configuration automatique JUnit et C4J.

- Facile d'utilisation : pour utiliser le plug-in, il suffit de faire un clic droit sur l'élément sur lequel on désire générer la classe de test et celui du contrat de classe puis à l'aide du menu choisir le mode de génération.

En combinant JUnit et C4J, notre solution pourrait être l'unique outil qui prend en compte les tests unitaires et la programmation par contrat. En effet il existe des outils permettant soit de faire des tests unitaires ou de la programmation par contrat, mais pas les deux à la fois.

6.3 Travaux futurs

Pour parfaire plus le logiciel GACTUS, les travaux futurs pouvant dériver de ce projet de recherche sont les suivants :

- Ajouter toutes les dépendances du plug-in une fois durant l'installation. En effet pour l'instant l'utilisation du logiciel devrait aussi télécharger C4J et l'importer dans son projet comme librairie. Cela peut s'avérer pénible pour les utilisateurs non avertis ou aussi s'il arrivait que C4J d'une quelconque manière ne puisse être téléchargé, cela rendrait GACTUS inutilisable ou plus ou moins non disponible sa fonctionnalité qui permet de gérer la programmation par contrat. GACTUS fait l'ajout automatique de JUnit car JUnit est intégré dans la plupart des environnements de développement.

- L'autre point important serait d'ajouter l'annotation logique soit dans C4J, soit faire une extension de C4J dans GACTUS. En effet les opérateurs logiques permettent de renforcer l'invariant de classe. On trouve des annotations logiques dans certains plug-ins Java de programmation par contrat tel que icontrat, Jcontracts. Notons aussi que Eiffel possède ses annotations. Ses annotations sont entre autres « *exist* », « *forall* », « *implies* ». Une annotation de type « *exist* » est une annotation qui est

valable pour au moins un élément d'une collection. L'annotation « *implies* » traduit l'implication logique. Une annotation « *forall* » tient si une assertion est valable pour chacun des éléments de la collection. Pour l'instant un développeur doit coder ces annotations à la main

ANNEXE A
Installation du plug-in GACTUS

A.1 Comment installer GACTUS

GACTUS a été développé comme un plug-in pour l'environnement de développement Eclipse. Ainsi il existe deux manières (manuelle ou avec « Eclipse marketplace ») pour intégrer GACTUS à Eclipse.

A.1.1 Installation manuelle

Pour installer GACTUS manuellement il faut procéder de la manière suivante : tout d'abord, aller dans le menu help d'Eclipse et cliquer sur le sous-menu « install New software ». Une fenêtre apparaît comme la montre la figure A.3 ci-dessous.

Dans cette fenêtre, cliquer sur le bouton « Add ». Une boite de dialogue (voir figure A.2) s'ouvre. Cette boite de dialogue sert à renseigner le nom du plug-in dans notre cas il faut mettre « GACTUS » et aussi à renseigner l'emplacement du plug-in. Cliquer sur « Local » pour aller chercher le plug-in sur votre ordinateur. Une fois les deux champs de la boite de dialogue renseignés, vous devrez voir apparaître le plug-in comme sur la figure A.3. Si le plug-in n'apparaît pas, il faut décocher « Group items by category ».

Pour poursuivre l'installation, sélectionner GACTUS en cochant la case devant le nom du plug-in GACTUS puis cliquer sur le bouton « suivant ». La fenêtre qui apparait vous demande d'accepter ou de refuser la licence. Accepter et cliquer de nouveau sur le bouton « suivant ». Une fois la licence acceptée, le reste de l'installation se fait en cliquant sur le bouton suivant jusqu'au démarrage effectif de l'installation (figure A.4). Si l'installation se passe bien une boite de dialogue apparait vous demandant de redémarrer Eclipse. Pour finir l'installation, redémarrer Eclipse. Vérifier l'installation de GACTUS sur votre environnement de développement Eclipse en cliquant sur le menu « Help » et après le sous-menu « Installation Details », vous voir devrez GACTUS parmi les différents plug-ins installés.

A.1.2 Installation à partir de Eclipse Marketplace

Dans le menu « Help » sélectionnez le sous-menu « Eclipse Marketplace » une fenêtre, comme celle de la figure ci-dessous apparaît. Dans cette fenêtre écrivez GACTUS dans la zone de recherche « Find » et cliquez sur le bouton entrer. Vous devrez voir apparaître GACTUS avec un bouton « Install ». En cliquant sur ce bouton une deuxième fenêtre apparaît, cliquez sur le bouton « Confirm » pour commencer l'installation des dépendances du plug-in . Une fois les dépendances installées, une nouvelle fenêtre apparaît, il faut accepter la licence et cliquez sur le bouton « finish » pour lancer l'installation du

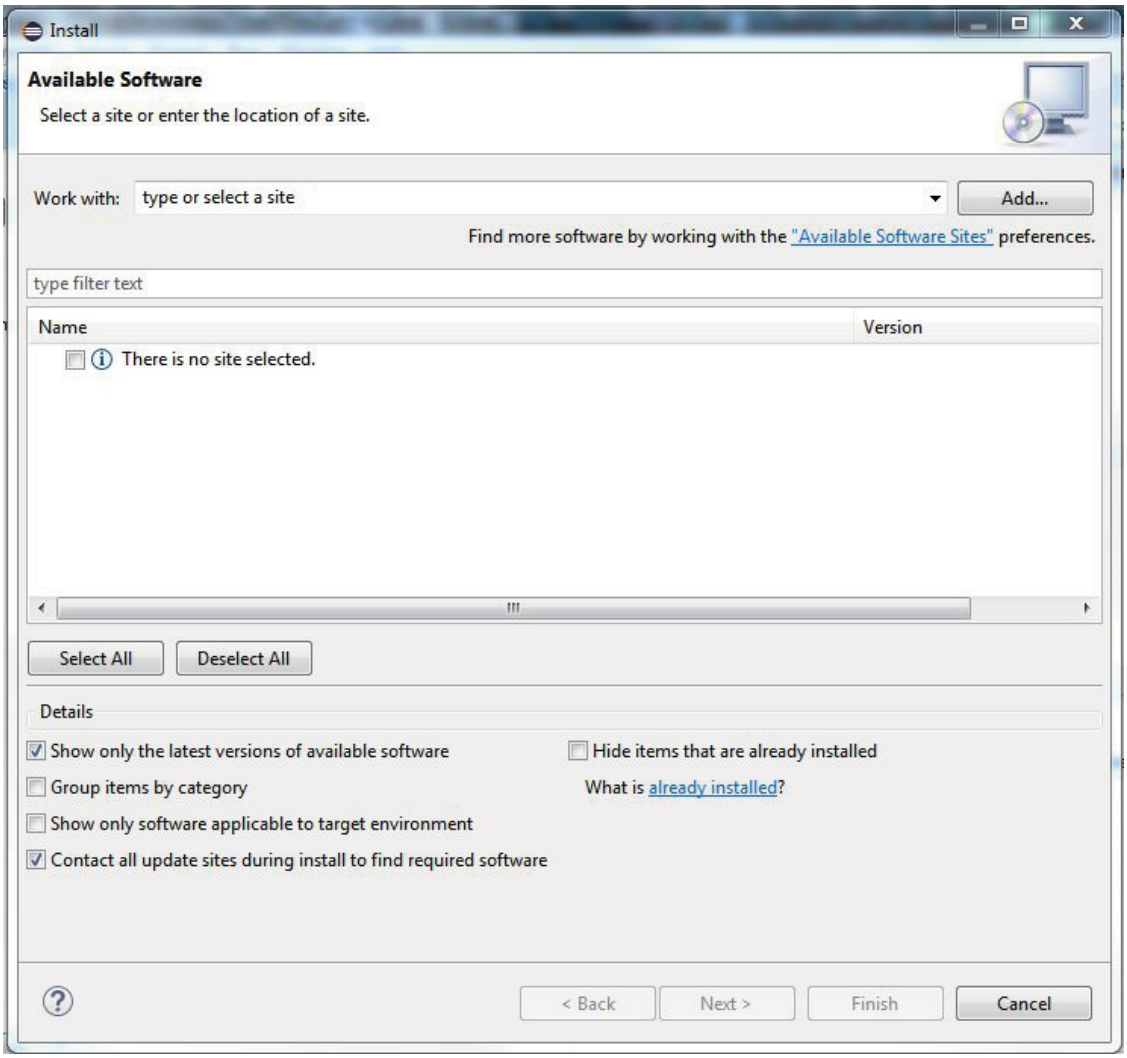

Figure A.1 GACTUS : fenêtre d'installation

plug-in. Si l'installation se passe bien, une boîte de dialogue apparaît de nouveau, vous demandant de redémarrer Eclipse. Cliquer sur « Ok » pour redémarrer Eclipse afin de finaliser l'installation.

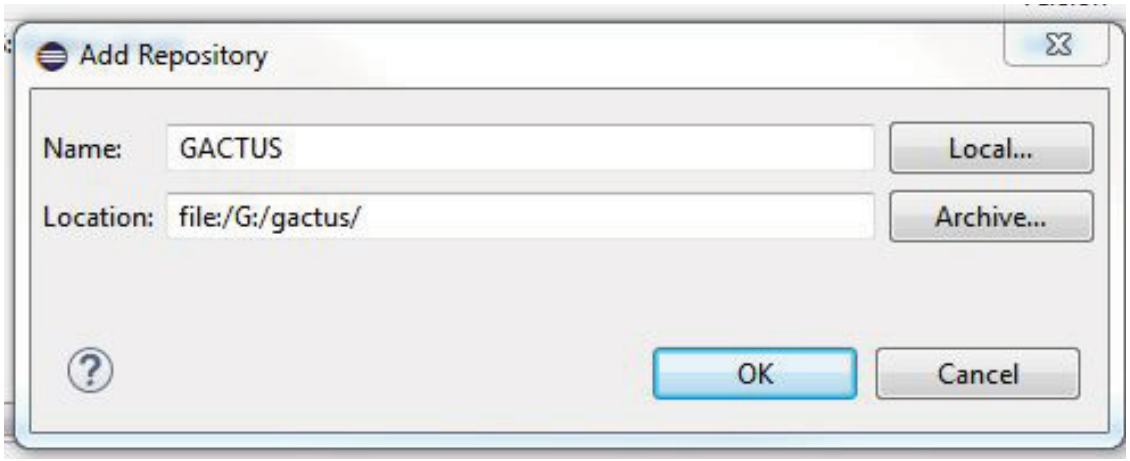

Figure A.2 Boîte de dialogue « Add repository »

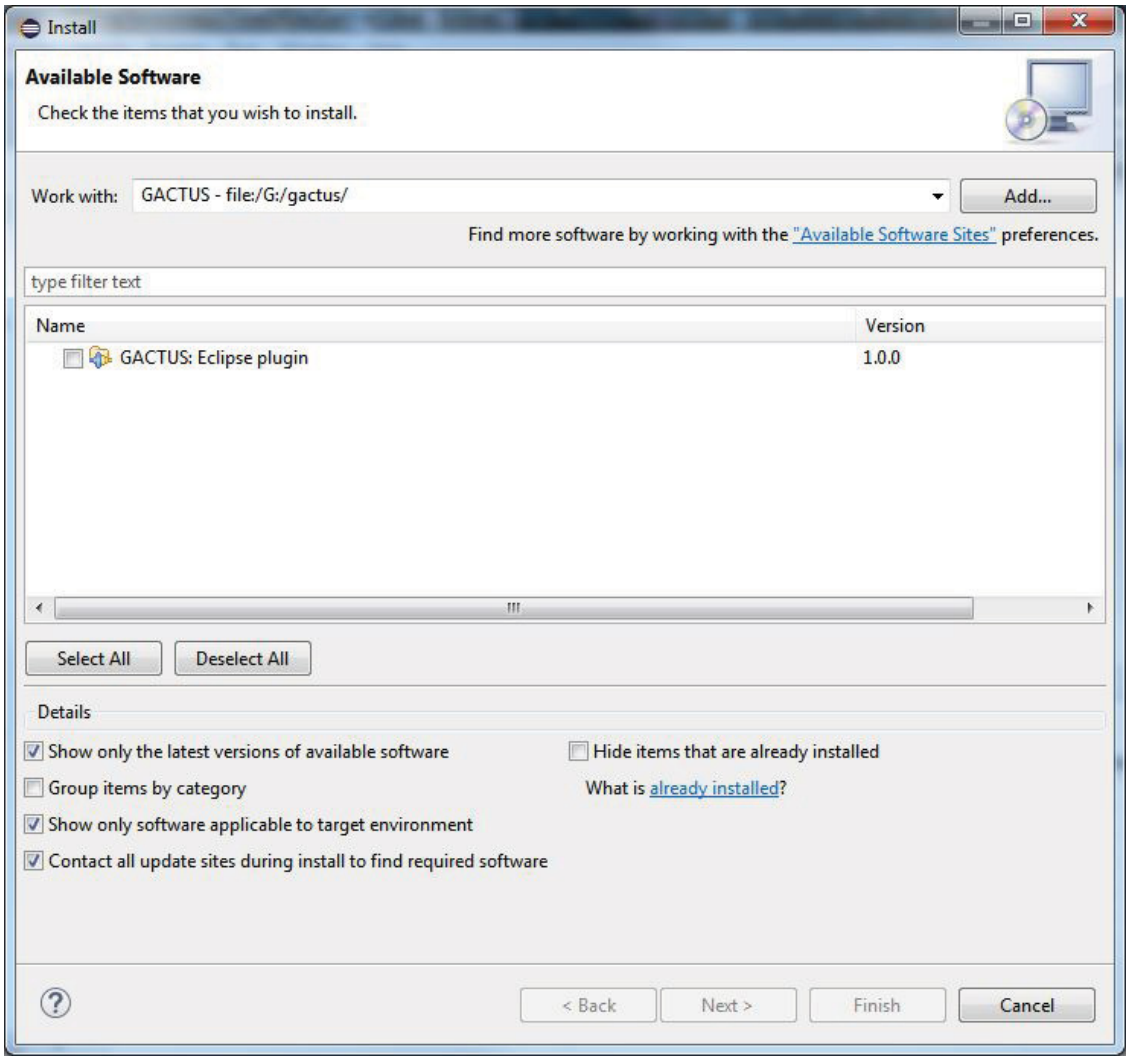

Figure A.3 GACTUS : installation suite

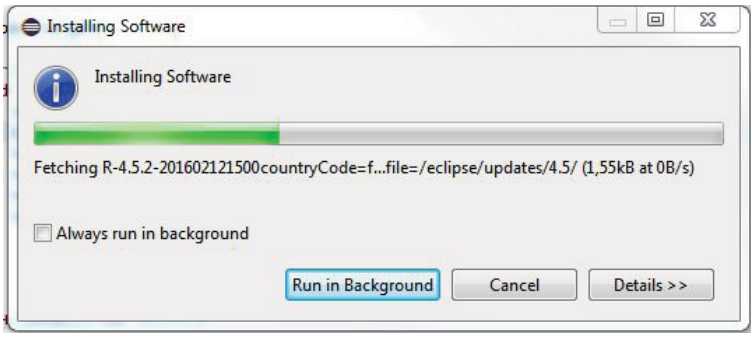

Figure A.4 GACTUS : installation en cours

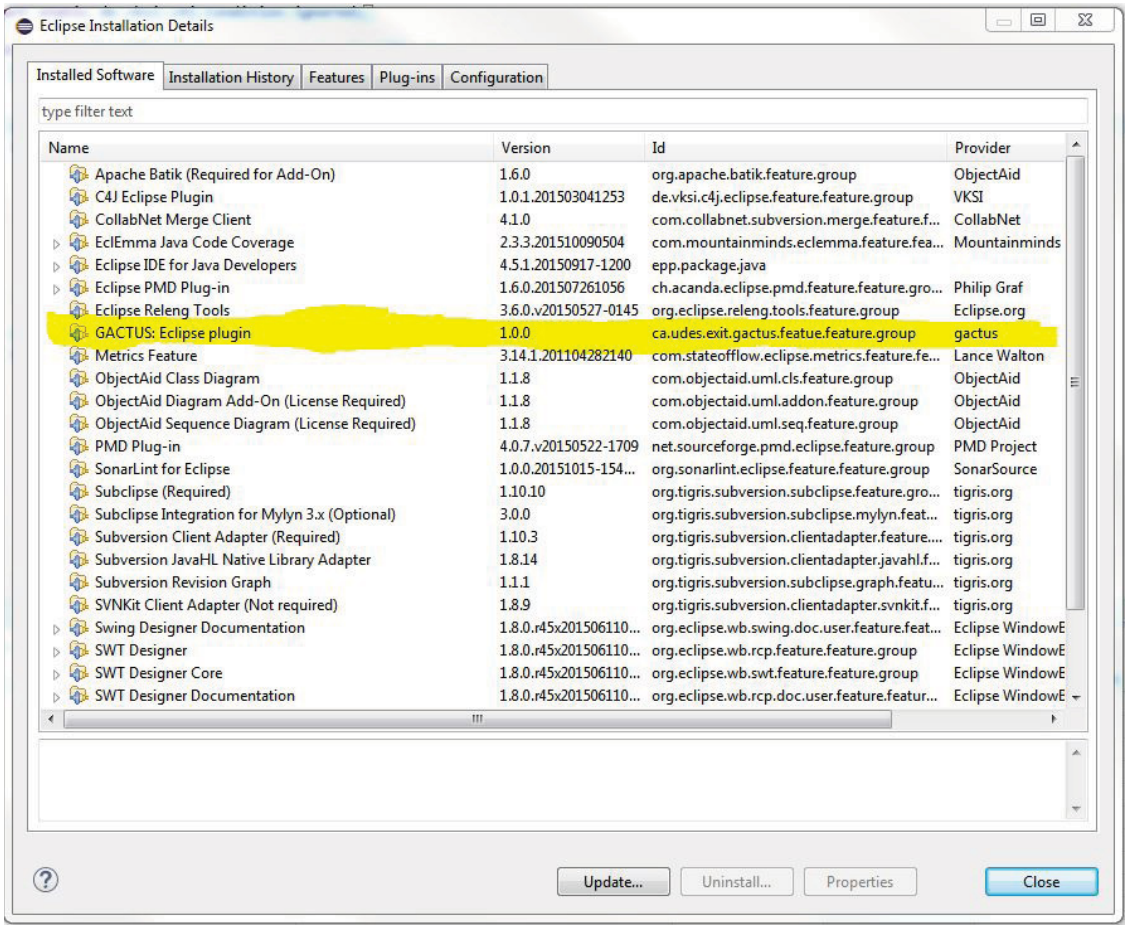

Figure A.5 Boîte de dialogue « Detail software installation »

ANNEXE B

Code source de squelette de classes généré avec GACTUS

B.1 Classe Personne

La classe Personne est une classe Java permettant de gérer le nom et le prénom d'une personne. Dans la suite, nous montrons les squelettes de classe de test unitaire et le squelette de contrat de classe générée avec GACTUS.

Listing B.1 – Code source de la classe Personne

```
package ca.exitlab.gactus.seminaire;

import de.vksi.c4j.ContractReference;
import de.vksi.c4j.Pure;

@ContractReference(PersonneContract.class)

public class Personne {
   private String nom;
   private String prenom;

   /**
    * @param nom
    * @param prenom
    */
   public Personne(String nom, String prenom) {
      this.nom = nom;
      this.prenom = prenom;
   }

   /**
    * @param nom
    * the nom to set
    */
   public void setNom(String nom) {
      this.nom = nom;
   }

   /**
    * @param prenom
```

```
     * the prenom to set
     */
    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }

    @Pure
    public String getNom() {
        return nom;
    }

    @Pure
    public String getPrenom() {
        return prenom;
    }

}
```

Listing B.2 – Squelette de contrat de classe de la classe Personne générée par GACTUS

```
package ca.exitlab.gactus.seminaire;

import static de.vksi.c4j.Condition.ignored;
import static de.vksi.c4j.Condition.postCondition;
import static de.vksi.c4j.Condition.preCondition;
import de.vksi.c4j.ClassInvariant;
import de.vksi.c4j.Target;

public class PersonneContract extends Personne {

   @Target
   private Personne target;

   @ClassInvariant
   public void classInvariant() {
      // TODO : add invariant code
   }

   public PersonneContract(String nom, String prenom) {
      super(nom, prenom);
      if (preCondition()) {
         // TODO : add precondition code
      }
      if (postCondition()) {
         // TODO : add postcondition code
      }
   }

   @Override
   public String getNom() {
      return ignored();
   }

   @Override
   public void setNom(String nom) {
      if (preCondition()) {
         // TODO : add precondition code
      }
      if (postCondition()) {
         // TODO : add postcondition code
      }
   }
```

```
   @Override
   public String getPrenom() {
      return ignored();
   }

   @Override
   public void setPrenom(String prenom) {
      if (preCondition()) {
         // TODO : add precondition code
      }
      if (postCondition()) {
         // TODO : add postcondition code
      }
   }

}
```

Listing B.3 – Squelette de la classe de test unitaire de la classe Personne générée par GACTUS

```
package ca.exitlab.gactus.seminaire;

import static org.junit.Assert.fail;

import org.junit.Test;

public class PersonneTest {

   @Test
   public void testPersonne() {
      // TODO: please add your test cas code here!
      fail("Not yet implemented");
   }
   @Test
   public void testgetNom() {
      // TODO: please add your test cas code here!
      fail("Not yet implemented");
   }
   @Test
   public void testsetNom() {
      // TODO: please add your test cas code here!
      fail("Not yet implemented");
   }
   @Test
   public void testgetPrenom() {
      // TODO: please add your test cas code here!
      fail("Not yet implemented");
   }
   @Test
   public void testsetPrenom() {
      // TODO: please add your test cas code here!
      fail("Not yet implemented");
   }
}
```

Listing B.4 – Structure du fichier « c4j-gobal.xml »

```
<!-- Structure du fichier C4J-global.xml -->
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<c4j-global xmlns="http://c4j.vksi.de/c4j-global/">
   <write-transformed-classes directory="c4j-classes">false
   </write-transformed-classes>
   <contract-violation-action>
      <default>
         <log>false</log>
         <throw-error>true</throw-error>
      </default>
   </contract-violation-action>
   <contract-violation-action>
      <default>
         <log>false</log>
         <throw-error>true</throw-error>
      </default>
      <package name="ca.exitlab.gactus.seminaire">
         <log>false</log>
         <throw-error>true</throw-error>
      </package>
      <class name="ca.exitlab.gactus.seminaire.Personne">
         <log>false</log>
         <throw-error>true</throw-error>
      </class>
   </contract-violation-action>
</c4j-global>
```

Listing B.5 – Structure du fichier « c4j-local.xml »

```
<!-- Structure du fichier c4j-local.xml -->
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<c4j-local xmlns="http://c4j.vksi.de/c4j-local/">
   <configuration>
      <pure-validate>true</pure-validate>
      <pure-registry-import>c4j-pure-registry.xml</pure-registry-import>
   </configuration>
   <configuration>
      <root-package>ca.udes.gegi.exit.gactus</root-package>
      <pure-validate>true</pure-validate>
      <pure-registry-import>c4j-pure-registry.xml</pure-registry-import>
<contract-scan-package>ca.udes.gegi.exit.gactus</contract-scan-package>
   </configuration>
</c4j-local>
```

ANNEXE C

Code source d'un handler du plugin GACTUS

Le code source suivant est celui du handler de la génération du squelette de contrat de classe et du squelette de classe de tests unitaire.

Listing C.1 – Code source du handler GactusAutoHandler

```
package com.udes.gactus.handlers;
/**
 *GactusAutoHandler handler extends AbstractHandler, an IHandler base
    class.
 */
public class GactusAutoHandler extends AbstractHandler {

   public GactusAutoHandler() {}
   /**
    * the command has been executed, so extract extract the needed
       information
    * from the application context.
    */
   public Object execute(ExecutionEvent event) throws ExecutionException {
      IWorkbenchWindow window = HandlerUtil
            .getActiveWorkbenchWindowChecked(event);
      MessageDialog dg = new MessageDialog(window.getShell(), "GACTUS",
         null,
           "Do you want to generate test case skeleton and contrat class
              skeleton ?",
           MessageDialog.QUESTION_WITH_CANCEL, new String[] {
                IDialogConstants.YES_LABEL, IDialogConstants.NO_LABEL,
                IDialogConstants.CANCEL_LABEL }, 0);
      if (dg.open() == Window.OK) {
         SelectionUtil util = new SelectionUtil(event);
         LibraryAndFolderUtils.getInstance().configProject(
               util.getTheProject());
         new GactusMainControler(event).execute();
      }
      return null;
   }
}
```

ANNEXE D

Code source de quelques classes de tests unitaires du plug-in GACTUS

Classe permettant d'exécuter plusieurs classes de tests unitaires.

Listing D.1 – Code source de la suite de test

```
package com.udes.gactus.main.test;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;



@RunWith(Suite.class)
@SuiteClasses({

          LibraryAndFolderUtilsTest.class,
          ClassInfosUtilsTest.class,
           PackageManagerTest.class,
         ProjectInformationTest.class,
         SearchFileUtilTest.class,
           MethodInformationTest.class,
        GenerateXMLC4JLocalTest.class,
        GenerateXMLC4JPureRegistryTest.class,
        GenerateXMLC4JGlobalTest.class,
        AnalyseProjectTest.class,
        GacutAutoClassAndInterfaceModelTest.class,
        XMLPopulateUtilTest.class,
        ContratClassSourceCodeGenerateTest.class,

        ConfigureTestUtilTest.class,

        TestCaseClassSourceCodeGenerateTest.class,

})
public class AllTests {
```

```
}
```

Classe qui teste la classe classeInfosUtil.

Listing D.2 – Code source de la classe de test unitaire de la classe ClassInfosUtils

```
package com.udes.gactus.mdole.core;

import static org.junit.Assert.*;

public class ClassInfosUtilsTest {
   ConfigureTestUtil configureTest = ConfigureTestUtil.getInstance();
   private ClassInfosUtils classInfos;

   @Before
   public void setUp() throws Exception {
      classInfos = configureTest.getProjectInformation()
            .findPackageInSrcByName("ca.gactus.projetoftest")
            .findClassOrInterfaceByName("Person");
      classInfos.setPackageManager(configureTest.getProjectInformation()
            .findPackageInSrcByName("ca.gactus.projetoftest"));
      classInfos.PopulateListOFInnerClass();
      classInfos.PopulateListOfMethoDeclaration();
   }

   @Test
   public void testClassInfosUtils() {
      assertNotNull(classInfos);
   }

   @Test
   public void testGetClassName() {
      assertEquals("Person", classInfos.getClassName());
   }

   @Test
   public void testGetImport() {
      assertNotNull(classInfos.getImport());
   }

   @Test
   public void testGetSuperClass() {
      assertNull(classInfos.getSuperClass());
   }
```

```
@Test
public void testHasInnerClass() {
   assertFalse(classInfos.hasInnerClass());
}

@Test
public void testFindByName() {
   MethodInformation methodInfo = classInfos.findByName("getName");
   assertEquals("getName", methodInfo.getMethodeName());
}

@Test
public void testFindIMethodeByName() {
   IMethod iMethode = classInfos.findIMethodeByName("getName");
   assertEquals("getName", iMethode.getElementName());
}

@Test
public void testClassPackageName() {
   assertEquals("ca.gactus.projetoftest",
      classInfos.classPackageName());
}

@Test
public void testGetAnnotation() {
   assertNotEquals(0, classInfos.getAnnotation());
}

@Test
public void testGetType() {
   IType iType = classInfos.getType();
   assertNotNull(iType);
}

@Test
public void testGetListOfMethodDeclaration() {
   assertNotNull(classInfos.getListOfMethodDeclaration());
}

@Test
public void testToString() {
   assertNotNull(classInfos.toString());
}

@Test
```

```
public void testIsAbstract() {
   assertFalse(classInfos.isAbstract());
}

@Test
public void testIsEmpty() {
   assertFalse(classInfos.isEmpty());
}

@Test
public void testListOfMethodIsEmpty() {
   assertFalse(classInfos.listOfMethodIsEmpty());
}

@Test
public void testIsContainContratReference() {
   assertFalse(classInfos.isContainContratReference());
}

@Test
public void testIsClass() {
   assertTrue(classInfos.isClass());
}

@Test
public void testIsInterface() {
   assertFalse(classInfos.isInterface());
}

@Test
public void testToStringAllMethodsInformation() {
   assertTrue(classInfos.toStringAllMethodsInformation().contains(
         "getName"));
}

@Test
public void testToStringAllInnerClassInformation() {
   assertNotNull(classInfos.toStringAllInnerClassInformation());
   ;
}

@Test
public void testGetListOfMethodWithContainAnnotation() {
   assertEquals(0, classInfos.getListOfMethodWithContainAnnotation()
         .size());
```

```
}

@Test
public void testGetSource() {
   assertTrue(classInfos.getSource().contains("name"));
}

@Test
public void testGetListOfInnerClass() {
   assertEquals(0, classInfos.getListOfInnerClass().size());
}

@Test
public void testGetMyClassUnit() {
   ICompilationUnit iCompilatUnit = classInfos.getMyClassUnit();
   assertNotNull(iCompilatUnit);
}

@Test
public void testGetPackageManager() {
   PackageManager pck = classInfos.getPackageManager();
   assertEquals("ca.gactus.projetoftest", pck.getPackageName());
}

@Test
public void testGetTypeString() {
   assertEquals("class", classInfos.getTypeString());
}

@Test
public void testIsExitngTestCaseFile() {
   assertFalse(classInfos.isExitngTestCaseFile());
}

@Test
public void testIsExitngContractFile() {
   assertFalse(classInfos.isExitngContractFile());
}

@Test
public void testIsAvailable() {
   assertTrue(classInfos.isAvailable());
}
```

```
   @Test
   public void testIsAvailableForTestCommad() {
      assertTrue(classInfos.isAvailableForTestCommad());
   }

   @Test
   public void testIsEmptyInterface() {
      assertFalse(classInfos.isEmptyInterface());
      ;
   }

   @Test
   public void testGetStatus() {
      assertEquals(0, classInfos.getStatus());
   }

   @Test
   public void testGetStatusForTestCase() {
      assertEquals(0, classInfos.getStatusForTestCase());
   }

   @Test
   public void testGetStatusForContratCase() {
      assertEquals(0, classInfos.getStatusForContratCase());
   }

   @Test
   public void testIsAContractClass() {
      assertFalse(classInfos.isAContratFile());
   }
}
```

ANNEXE E

Code source d'un contrat de classe du plug-in GACTUS

Listing E.1 – Contrat de classe de la classe ProjectInformation

```
package com.udes.gactus.core;

public class ProjectInformationContract extends ProjectInformation {
   public ProjectInformationContract(IProject project) {
      // TODO Auto-generated constructor stub
      super(project);
   }

   @Target
   private ProjectInformation target;

   @ClassInvariant
   public void classInvariant() {
      assert target != null : " ProjectInformation must not null";
   }

   @Override
   public String getProjectName() {
      return ignored();
   }

   @Override
   public IProject getProject() {
      return ignored();
   }

   @Override
   public String getPathOfProject() {
      return ignored();
   }

   @Override
   public void setProject(IProject project) {
      if (preCondition()) {
         assert project != null : " project must not null";
```

```
    }
    if (postCondition()) {
       assert target.getProject() == project : "project is set";
    }
 }

 @Override
 public String getWorkspace() {
    return ignored();
 }

 @Override
 public PackageManager findPackageInSrcByName(String name) {
    if (preCondition()) {
       assert name != null && name != "" : "Package name must not null";
    }
    if (postCondition()) {

    }
    return ignored();
 }

 @Override
 public PackageManager findPackageInSrcByClassInfo(ClassInfosUtils
     myClass) {
    if (preCondition()) {
       assert myClass != null : "class must exist";
    }
    return ignored();
 }

 @Override
 public ClassInfosUtils findClassInfoUtilByName(String className) {
    if (preCondition()) {
       assert className != null && className != "" : "classe name must
          not null";
    }
    if (postCondition()) {

    }
    return ignored();
 }

 @Override
 public IPackageFragment[] getPackages() {
```

```
   return ignored();
}

@Override
public void setPackages(IPackageFragment[] packages) {
   if (preCondition()) {
      assert packages != null : "parameter must not null";
   }
   if (postCondition()) {
      assert target.getPackages() == packages : "target packages must
         change";

   }

}

@Override
public List<PackageManager> getListOfAllPackageInProject() {
   if (preCondition()) {

   }
   if (postCondition()) {
      assert target.getListOfAllPackageInProject() instanceof
         java.util.List : " resutl must a List";
   }
   return ignored();
}

@Override
public List<String> getListOfSourcesFolder() {
   if (preCondition()) {

   }
   if (postCondition()) {
      assert target.getListOfSourcesFolder() instanceof java.util.List
         : " resutl must a List";
   }
   return ignored();
}

@Override
public List<PackageManager>
    getListOfAllAvailablesPackageForTestCommade() {
   if (preCondition()) {
```

```
    }
    if (postCondition()) {
       assert target.getListOfAllAvailablesPackageForTestCommade()
          instanceof java.util.List : " resutl must a List";
    }
    return ignored();

 }

 @Override
 public List<PackageManager>
     getListOfAllAvailablesPackageForTestAndContratCommand() {
    if (preCondition()) {

    }
    if (postCondition()) {
       assert target
             .getListOfAllAvailablesPackageForTestAndContratCommand()
                instanceof java.util.List : " resutl must a List";
    }
    return ignored();

 }

 @Override
 public boolean isNotOpen() {
    if (preCondition()) {

    }
    if (postCondition()) {
       assert target.isNotOpen() == true || target.isNotOpen() == false
          : " resultat is boolean";
    }
    return ignored();
 }

 @Override
 public int getStatus() {
    if (preCondition()) {

    }
    if (postCondition()) {
       assert target.getStatus() == 0 || target.getStatus() == 1
             || target.getStatus() == 2 || target.getStatus() == 3 :
                "Status must 0 or 1 or 3 or 4";
```

```
      }
      return ignored();
   }

   @Override
   public int getStatusForTestGeneration() {
      if (preCondition()) {

      }
      if (postCondition()) {
         assert target.getStatusForTestGeneration() == 0
               || target.getStatusForTestGeneration() == 1
               || target.getStatusForTestGeneration() == 2
               || target.getStatusForTestGeneration() == 3 : "Status must
                  0 or 1 or 3 or 4";
      }
      return ignored();
   }

   @Override
   public IJavaProject getiJavaProjec() {
      return ignored();
   }

   @Override
   public void setiJavaProjec(IJavaProject iJavaProjec) {
      if (preCondition()) {
         assert iJavaProjec != null : "parameter must not null";
      }
      if (postCondition()) {
         assert target.getiJavaProjec() == iJavaProjec : " target attribut
            iJavaProject must set";
      }
   }
}
```

LISTE DES RÉFÉRENCES

[1] Appel, A. W. et Palsberg, J. (2002). *Modern compiler implementation in Java*, 2e édition. Cambridge University Press, Cambridge, UK ; New York, NY, USA, 512 p.

[2] association, I. S. (2016). *IEEE standards association*. `https://standards.ieee.org/findstds/standard/829-1998.html` (page consultée le 19 mai 2016).

[3] Bacchelli, A., Ciancarini, P. et Rossi, D. (2008). On the effectiveness of manual and automatic unit test generation. Dans *Software Engineering Advances, 2008. ICSEA '08. The Third International Conference on*. IEEE Computer Society, p. 252–257.

[4] Beck, K. (2002). *Test-Driven Development : By Example*, 1re édition. Addison-Wesley Professional, Boston, 240 p.

[5] Beck, K. et Gamma, E. (2002). *JUnit CookBook*. `http://junit.sourceforge.net/doc/cookbook/cookbook.htm` (page consultée le 23 octobre 2015).

[6] Boehm, B. W., Brown, J. R. et Lipow, M. (1976). Quantitative evaluation of software quality. Dans *Proceedings of the 2Nd International Conference on Software Engineering*. ICSE '76. IEEE Computer Society Press, Los Alamitos, CA, USA, p. 592–605.

[7] C4J (2011). *C4J Overview*. `http://c4j-team.github.io/C4J/index.html` (page consultée le 15 mai 1016).

[8] Capers, J. (2012). Software quality in 2012 : A survey of the state of the art. *Software Quality Groyp of New England*. `http://sqgne.org/presentations/2012-13/Jones-Sep-2012.pdf` (page consultée le 23 janvier 2016).

[9] Cavano, J. P. et McCall, J. A. (1978). A framework for the measurement of software quality. *SIGSOFT Softw. Eng. Notes*, volume 3, numéro 5, p. 133–139.

[10] Charette, R. N. (02 2009). *IEEE Spectrum*. `http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code` (page consultée le 18 janvier 2016).

[11] Côté, M.-O., Suryn, W. et Georgiadou, E. (2006). Software quality model requirements for software quality engineering. *ETS*. `http://profs.etsmtl.ca/wsuryn/research/SQE-Publ/Quality%20model_requirements.%20SQM2006.pdf` (page consultée le 30 janvier 2016).

[12] Doudoux, J. M. (2007). *Développons en Java avec Eclipse*. `http://www.jmdoudoux.fr/java/dejae/chap007.htm` (page consultée le 30 mars 2016).

[13] Fleming, I. (2010). *SQA Software Quality Assurance*. `http://www.sqa.net/softwarequalityattributes.html` (page consultée le 24 janvier 2016).

[14] Forys, A. (10 2001). *Journal du net*. `http://www.journaldunet.com/developpeur/tutoriel/out/011012_parasoft(3).shtml` (page consultée le 19 avril 2016).

[15] Freeman, E., Freeman, E., Bates, B. et Sierra, K. (2004). *Head first design patterns*, 1re édition. O'Reilly and Associates Inc, 654 p.

[16] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, volume 12, numéro 10, p. 576–580.

[17] Hollis, M., Bergström, J. et Particka, A. (2010). *sourceforge*. `http://c4j.sourceforge.net/` (page consultée le 10 octobre 2014).

[18] IBM (2014). *IBM*. `http://www-01.ibm.com/support/knowledgecenter/SSZND2_6.0.0/org.eclipse.jdt.doc.isv/guide/jdt_int.htm?cp=SSZND2_6.0.0%2F3-1-1-0&lang=fr` (page consultée le 15 aout 2015).

[19] Jazequel, J. M. et Meyer, B. (1997). Design by contract : the lessons of ariane. *Computer*, volume 30, numéro 1, p. 129–130.

[20] Kan, S. H. (2002). *Metrics and Models in Software Quality Engineering*, 2e édition. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 560 p.

[21] Kebir, S. (11 2014). *Developpez.com*. `http://skebir.developpez.com/tutoriels/eclipse/jdt/` (page consultée le 10 août 2015).

[22] Link, J. (2003). *Tests unitaires en Java : les tests au coeur du développement*. Dunod, Paris, 352 p.

[23] McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, volume SE-2, numéro 4, p. 308–320.

[24] Meyer, B. (1992). *Eiffel : the language*. Prentice-Hall, Inc., New York.

[25] Meyer, B. (2000). *Conception et programmation orientées objet*, 2e édition. Eyrolles, Paris, 1224 p.

[26] Richard, M. et Jim, M. (2002). Elementary principles of design by contract. Dans *Design by Contract, by Example*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, p. 1–167.

[27] Saff, D., Cooney, K., Birkner, S. et Philipp, M. (12 2014). *JUnit*. `http://junit.org/` (page consultée le 23 novembre 2015).

[28] Schneider, A. (21 Decembre 2000). *Javaworld*. `http://www.javaworld.com/article/2076265/testing-debugging/junit-best-practices.html#resources` (page consultée le 30 janvier 2016).

[29] Tiobe (07 2016). *Tiobe*. `http://www.tiobe.com/tiobe-index/` (page consultée le 25 juillet 2016).

[30] Tremblay, G. (2005). *labunix*. `http://www.labunix.uqam.ca/~tremblay/INF3135/Materiel/contrats+erreurs.pdf` (page consultée le 10 octobre 2014).

[31] Vogel, L. (08 2012). *vogella.com*. `http://www.vogella.com/tutorials/EclipseJDT/article.html` (page consultée le 3 juin 2015).

[32] Wahid, M. et Almalaise, A. (2011). Junit framework : An interactive approach for basic unit testing learning in software engineering. Dans *Engineering Education (ICEED), 2011 3rd International Congress on*. IEEE Computer Society, p. 159–164.

[33] Wang, X. (01 2011). *Eclipse JDT Tutorials*. `http://www.programcreek.com/2011/01/best-java-development-tooling-jdt-and-astparser-tutorials/` (page consultée le 20 decembre 2015).

[34] Watt, D. A. (2004). *Programming Language Design Concepts*. John Wiley & Sons, 492 p.

[35] Wikipedia (12 2014). *Wikipedia*. `http://fr.wikipedia.org/wiki/Compilateur` (page consultée le 4 avril 2016).

[36] Wikipedia (02 2015). *Wikipedia*. `http://fr.wikipedia.org/wiki/M%C3%A9thode_formelle_(informatique)` (page consultée le 13 avril 2015).

[37] Zhang, L., Marinov, D., Zhang, L. et Khurshid, S. (2011). An empirical study of junit test-suite reduction. Dans *Proceedings of the 2011 IEEE 22Nd International Symposium on Software Reliability Engineering*. ISSRE '11. IEEE Computer Society, Washington, DC, USA, p. 170–179.

[38] Zhu, H., Hall, P. A. V. et May, J. H. R. (1997). Software unit test coverage and adequacy. *ACM Comput. Surv.*, volume 29, numéro 4, p. 366–427.