UNIVERSITÉ DU QUÉBEC À MONTRÉAL

DÉTECTION DE DÉFAUTS DE PROGRAMMES JAVA

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

JOHNNY TSHEKE SHELE

JANVIER 2014

À Godelive.

REMERCIEMENTS

J'aimerais d'abord remercier particulièrement le professeur Guy Tremblay, mon directeur de recherche pour son encadrement, sa pédagogie, sa disponibilité et son soutien sous des formes diverses. Je le remercie également de manière spécifique pour le soutien financier qu'il m'a accordé. Sur ce registre de support financier, j'aimerais aussi remercier la professeure Naouel Moha pour avoir contribué à l'appariement qui m'a permis de bénéficier d'une bourse FARE. Je remercie également le professeur Abdellatif Obaid pour ses conseils et les différentes charges d'auxiliaire d'enseignement et de recherche qu'il m'a confiées. Je remercie tous les professeurs, chargés de cours et employées du département d'informatique avec une pensée spéciale pour ceux qui m'ont confié des charges d'auxiliaire d'enseignement. Les charges de correcteur pour le cours INF1120 (Programmation I) m'ont particulièrement permis d'utiliser *Oto* et d'examiner attentivement les codes sources des programmes des étudiants. Cela cadrait avec mes recherches dans le cadre du présent mémoire. Je ne saurais finir cette section sans remercier spécifiquement le professeur Etienne M. Gagnon, le concepteur de *SableCC* et *ObjectMacro*. Sa rigueur, ses exigences et son encadrement dans le cadre du cours de compilation ont sans doute été déterminant pour le développement du logiciel présenté dans ce mémoire.

Par la suite, j'aimerais remercier, d'une manière remarquable, mes enfants : Brian et Godelive. Je n'oublierai pas combien vous avez été particulièrement formidable durant la session d'hiver 2011. Votre soutien a été capital pour me permettre de poursuivre cette maîtrise. Trouvez ici l'expression de la plus grande satisfaction qu'un père puisse exprimer à ses enfants. Mes remerciements vont aussi à ma famille, aux amis et connaissances pour avoir été à mes cotés durant ces études. Je remercie nommément Benoit Okondalole-Wosha pour son soutien. J'aimerais aussi exprimer une gratitude à Jeannine Kawe, la mère de mes enfants, pour tout soutien direct ou indirect.

Pour finir, j'aimerais remercier les professeurs Esteban Zimányi de l'Université Libre de Bruxelles, Pierre Manneback de l'Université de Mons et Marie-Ange Remiche de l'Université de Namur pour avoir signé les lettres de recommandations en vue de poursuivre des études aux cycles supérieurs à l'UQAM.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

RÉSUMÉ

L'évaluation d'un programme dans le cadre d'un cours d'introduction à l'informatique est une tâche difficile qu'il faut compléter dans un délai précis. Au delà du fonctionnement, il faut aussi vérifier la qualité du code source du programme. Par la qualité du programme, on entend le fait que le code source se conforme aux standards et conventions généralement admis comme un bon style de programmation. Dans la pratique, la vérification de la qualité ne se préoccupe pas des erreurs détectables par le compilateur car le programme à analyser est supposé exempt de ces erreurs. On vérifie plutôt le non-respect des conventions et standards. On qualifie alors de défaut, une instruction ou partie du code source qui ne respecte pas les exigences du style de programmation considéré. L'évaluation de la qualité se fait donc en détectant les défauts pertinents.

Plusieurs institutions d'enseignement se sont lancées dans des projets divers pour mettre en place des outils d'évaluation automatique des programmes des étudiants. Le département d'informatique de l'Université du Québec à Montréal dispose d'un outil appelé *Oto*. Cet outil permet de corriger les aspects fonctionnels des programmes. Il lui manque la capacité d'évaluer la qualité du code source. *Oto* est néanmoins capable de faire appel à une application externe s'exécutant en ligne de commande et utilisant les flux standards d'entrée/sortie.

Dans ce mémoire, nous présentons *DDPJ*, *Détection de Défauts dans les Programmes Java*, notre outil de détection des défauts de programmes Java. Son fonctionnement est basé sur l'utilisation de métriques. Dans sa conception, chaque défaut est associé à une métrique. La détection des défauts sur le code source se fait donc en calculant les métriques associées. La spécification des défauts à détecter se fait tout simplement en passant la liste des métriques associées en arguments à *DDPJ*. Le programme *DDPJ* fournira alors un rapport avec les valeurs des métriques demandées. Au besoin, *DDPJ* peut aussi fournir un rapport détaillé avec les numéros de lignes des défauts dans le code source. *DDPJ* est développé en Java en utilisant *SableCC* et *ObjectMacro*. Il est portable et peut s'exécuter seul en ligne de commande ou intégré dans n'importe quel programme capable d'exécuter une application externe en ligne de commande.

INTRODUCTION

De nos jours, les cours d'introduction à la programmation attirent souvent un nombre important d'étudiants. Cela est particulièrement vrai dans l'enseignement supérieur en général et dans les universités en particulier. Ces étudiants font plusieurs travaux de programmation que l'enseignant doit corriger. Par enseignant, nous entendons le responsable du cours ou toute personne participant à l'évaluation des travaux de programmation. À l'université, ce pourrait être un professeur, un chargé de cours, un auxiliaire d'enseignement ou un correcteur.

Une des particularités dans la correction des programmes informatiques est qu'il faut non seulement vérifier le bon fonctionnement (compilation, exécution, résultats) du programme, mais il faut aussi examiner la qualité du code source. Par la qualité du code, il faut comprendre le fait que le code source se conforme à certains standards ou conventions généralement admis comme un bon style de programmation. Dans l'idéal, on aimerait non seulement avoir un programme informatique qui donne de bons résultats mais aussi que ce programme ne soit pas encombré avec du code inutile et qu'on puisse le maintenir ou le faire évoluer facilement.

La maintenance et l'évolution du code sont importantes dans la vie d'une application informatique. En effet, on est souvent amené à modifier le code source d'un programme informatique pour corriger, déboguer ou améliorer son fonctionnement ou encore pour supprimer ou ajouter certaines fonctionnalités. Modifier ou déboguer un programme informatique n'est pas facile. D'après Brian W. Kernighan et P. J. Plauger, ce serait même l'aspect le plus difficile de la programmation :

> *The problem with obscure code is that debugging and modification become much more difficult, and these are already the hardest aspects of computer*

programming (Kernighan et Plauger, 1978).

Il est donc capital d'avoir l'habitude d'écrire du bon code qu'on pourrait déboguer ou maintenir facilement. Pour y arriver, les enseignants veillent à ce que les futurs programmeurs s'imprègnent des bonnes pratiques dès le début de leur apprentissage. Le problème est que la vérification du style nécessite de parcourir le code source pour l'examiner et de s'assurer que les standards sont respectés. Le nombre souvent élevé d'étudiants inscrits aux premiers cours de programmation et le rythme des travaux laissent peu de temps au correcteur pour vérifier la qualité du code de chaque programme de manière adéquate dans le délai imparti.

À la difficulté d'évaluation de la qualité d'un code source s'ajoute une complication supplémentaire et spécifique à la correction des travaux des étudiants : l'équité. En effet, au bout du compte, le travail évalué doit être sanctionné par une note. Il faut donc, non seulement pouvoir évaluer la qualité de code d'un programme étudiant mais aussi pouvoir le noter de manière équitable par rapport aux autres travaux. Il faut donc être en mesure de s'assurer que des travaux de qualités équivalentes obtiennent la même note. Sur le plan pédagogique, il faut aussi donner des indications précises sur les faiblesses ou les défauts du code source afin de permettre à l'étudiant de s'améliorer dans son apprentissage. Du point de vue fonctionnement, si on met les mêmes entrées aux différents programmes, on s'attend à avoir des résultats identiques. Il suffit donc de vérifier, pour chaque programme, si on a les résultats attendus ou non. Quant aux aspects de qualité, le problème est tout autre. En principe, le code source de chaque programme est unique. Même pour des programmes qui font la même chose, les variables, fonctions, boucles et autres éléments des codes sources peuvent différer. On ne peut donc pas vérifier la qualité du code comme on vérifie habituellement le fonctionnement d'un programme.

Comme on peut s'en rendre compte, la correction des programmes informatiques des étudiants des cours d'initiation à la programmation est une tâche difficile qu'il faut accomplir au mieux possible dans un délai précis. Pour alléger cette tâche, plusieurs institutions d'enseignement se sont lancées dans différents projets pour développer des outils de correction automatique. À ce jour, la plupart des outils sont surtout en mesure

de corriger le fonctionnement en vérifiant si le programme analysé donne bien les résultats attendus. En ce qui concerne le style de programmation, des outils existent également mais ne sont pas appropriés pour la correction automatique des travaux des apprentis programmeurs. Cela s'explique notamment par la difficulté de déterminer les aspects de qualité à vérifier, la complication de les comptabiliser et de les repérer sur le code source, ainsi que la nécessité de rapporter le résultat de l'analyse tant sous forme chiffrée pour faciliter l'attribution d'une note que sous forme détaillée pour permettre à l'étudiant d'identifier facilement ses erreurs.

À l'Université du Québec à Montréal (UQAM), le département d'informatique dispose d'un outil de correction automatique appelé *Oto*. Cet outil permet de corriger les travaux de programmation du point de vue fonctionnement. Il lui manque toutefois la capacité d'analyser la qualité du code source. C'est pour répondre à ce besoin que nous présentons, dans ce mémoire, un outil de détection de défauts des programmes Java, outil que nous appelons *DDPJ, Détection de Défauts dans les Programmes Java*. L'accent est mis sur les défauts que l'on rencontre typiquement dans les travaux des étudiants des premiers cours de programmation. Notre outil est portable et peut être intégré facilement dans n'importe quel logiciel de correction automatique capable d'exécuter un programme externe de type «filtre» — donc qui reçoit les données et émets ses résultats par l'intermédiaire des flux d'entrée/sortie standards.

Bien que cette première version de notre logiciel soit développée pour analyser le code source Java, sa conception prévoit plusieurs possibilités d'extension tant au niveau des langages de programmation pris en charge qu'au niveau des éléments du style de programmation à examiner. Comme on s'intéresse principalement à la qualité du code, nous ne considérons donc pas les erreurs détectables par les compilateurs. Notre outil se positionne après la compilation. Le programme à analyser est présumé compilable et sans erreur de compilation. Ce choix s'explique par le fait qu'habituellement, un programme qui ne compile pas n'est pas corrigé et qu'on lui attribue directement une note de 0. La correction automatique de la qualité du code n'est donc pertinente que pour les programmes qui, au minimum, compilent sans problème et sans erreur de syntaxe.

L'outil de détection de défauts que nous présentons dans ce mémoire est premièrement conçu pour une utilisation dans le cadre de l'enseignement. Cette affirmation ne change en rien le fait que ce logiciel pourrait aussi être pertinent dans d'autres contextes. Pour faciliter la comptabilisation des violations des standards et conventions, nous qualifions de défaut toute instruction ou partie du code qui ne respecte pas certaines règles. À chacun de ces défauts, nous associons une métrique. Le problème revient donc à calculer les valeurs des métriques dans le code source à analyser. En rapportant les valeurs des métriques à l'issue de l'analyse du code source, on peut alors attribuer une note automatiquement. Une version détaillée du rapport contiendra aussi les informations nécessaires (numéro de ligne, par exemple) pour permettre de retrouver facilement les défauts rapportés dans le code source.

Dans la suite de ce mémoire, nous aborderons, au chapitre 1, quelques principaux outils utilisés actuellement pour la correction automatique. Ils seront abordés dans une perspective de détection des défauts de qualité ou d'évaluation de style de programmation. Ce sera aussi une occasion pour positionner notre outil par rapport à d'autres outils. On verra notamment pourquoi il était nécessaire de se lancer dans la conception et le développement d'un tel logiciel. Le chapitre 2 présentera les défauts que l'on trouve typiquement dans les programmes des étudiants. Un grand nombre de ces défauts sont repris dans des standards comme la convention *Sun*. De nombreux autres sont rapportés par des enseignants suite aux constats faits aux cours des années en corrigeant les travaux. Ceci nous conduira au chapitre 3 où nous passerons en revue les principaux logiciels utilisés pour la mise en oeuvre de *DDPJ*. En ce qui concerne la stratégie de détection des défauts, ce sera le sujet du chapitre 4. Il y sera présentée une stratégie globale pour la détection des défauts et, pour certains d'entre eux, nous présenterons une stratégie spécifique. Ces stratégies seront parfois accompagnées des algorithmes que l'on pourrait implémenter dans n'importe quel langage de programmation. Ces stratégies et algorithmes ont été mis en oeuvre concrètement dans le logiciel *DDPJ* qui sera présenté au chapitre 5. Dans le chapitre 6, nous montrons quelques exemples d'utilisation de *DDPJ* pour la correction des travaux en groupe. Il y a, en particulier, la correction des travaux

réellement rendus par les étudiants de l'UQAM, dans le cadre d'un travail pratique du cours de Programmation I (INF1120). Ces exemples montreront aussi la manière d'utiliser *DDPJ* comme une application externe à partir d'*Oto*. Celà est particulièrement utile dans la masure où *DDPJ* n'est pas encore intégré dans *Oto* sous forme de module spécifique. Finalement, nous allons conclure en donnant aussi quelques suggestions pour des recherches futures. Quant aux annexes, elles contiendront principalement le manuel d'utilisation du logiciel et les principaux fichiers de configuration de *DDPJ*. On trouvera aussi, dans les annexes, quelques exemples supplémentaires d'utilisation de *DDPJ*.

CHAPITRE I

OUTILS DE CORRECTION DE TRAVAUX DE PROGRAMMATION ET DE DÉTECTIONS DE DÉFAUTS

Dans ce chapitre, nous décrivons quelques outils de détection des défauts dans les codes sources. Nous allons particulièrement nous intéresser aux logiciels utilisés pour la correction des travaux des étudiants. En ce qui concerne les aspects historiques de ces outils, nous référons le lecteur aux documents indiqués dans la bibliographie (Guérin, 2005; Helmick, 2007; Tremblay et Labonté, 2003).

1.1 Oto

Dans cette section nous présentons brièvement *Oto* et la possibilité de lui intégrer d'autres applications.

1.1.1 Présentation générale

Oto est un outil de remise, de correction et de vérification (Tremblay et al., 2008) des travaux de programmation développé à l'Université du Québec à Montréal (UQAM). *Oto* est écrit en Ruby et a été conçu par Frédéric Guérin dans le cadre de son mémoire de maîtrise en informatique (Guérin, 2005). Bien que la version de base soit une application Unix utilisable en mode ligne de commande, il existe aussi une version Web (Oto, 2005). De plus, des améliorations importantes au niveau du langage de script ont aussi été apportées par Paul Lessard dans le cadre de son mémoire de maîtrise en

informatique (Lessard, 2010).

En ce qui concerne la remise ou la soumission des travaux, l'enseignant doit d'abord initialiser le système en créant une boîte de remise des travaux avec les paramètres appropriés. Il transmet par la suite les informations utiles aux étudiants (nom de la boîte). Les étudiants peuvent alors remettre les travaux au moyen de la ligne de commande ou via une interface Web. Les travaux remis sont sauvegardés dans un répertoire précis, contenant tous les travaux dans une boîte et remis par un groupe d'étudiants.

Une fois la date de remise atteinte, l'enseignant peut alors obtenir les travaux remis — en les important dans son espace personnel — puis peut procéder à la correction. Pour ce faire, il doit tout d'abord écrire un script contenant les commandes à exécuter (compilation, exécution de tests, commande Unix, etc.) pour procéder à la correction des travaux, ainsi que les arguments à fournir au programme. Le script spécifie aussi, par le biais de diverses formes de tests, les résultats attendus par l'exécution du programme.

1.1.2 Modules d'extension

Dès le départ, *Oto* a été conçu pour être extensible à l'aide de modules d'extension. Un tel module est un sous-programme intégré à *Oto* et effectuant une tâche précise. *Oto* dispose de divers modules de base, notamment, `Javac`, `JUnit` (Guérin, 2005). De façon générale, les modules d'extension viennent ajouter d'autres fonctionnalités à *Oto*, servant généralement de *proxy* pour des programmes externes, s'exécutant de façon typique en mode ligne de commande et utilisant les flux d'entrée/sortie standards.

Une façon simple de développer une application à intégrer à *Oto* est donc de prévoir une exécution en ligne de commande et telle que les résultats (*output*) soient produits sur la sortie standard. De la sorte, *Oto* peut récupérer le résultat pour le traiter et, possiblement, l'intégrer dans un rapport qui sera retourné à l'utilisateur.

1.2 Automark++

Automark++ est un outil d'évaluation de la qualité de programmes Java d'étudiants (Jubair et Khair, 2004). C'est une extension d'Automark, proposé initialement par Redish et Smyth (Redish et Smyth, 1986), qui a été faite à l'Université de Jordanie par Jubair et Khair. Comme pour la plupart des outils de correction, il a besoin d'une solution type que doit fournir le correcteur ou l'enseignant.

Pour des projets plus élaborés et pour lesquels il n'est pas facile de développer une solution type, Automark++ procède à l'évaluation en calculant un certain nombre de métriques. La qualité du projet évalué est déterminée en appliquant certaines formules aux valeurs des métriques. Le poids ou l'importance de chaque métrique est déjà fixé dans l'outil. Plusieurs métriques calculées n'ont de sens que pour un projet avec plusieurs classes ou méthodes. C'est notamment le cas des métriques suivantes :

- DIT (*Depth of Inheritance Tree*) : la profondeur de l'héritage entre une classe et la racine.
- TCN (*Total Number of Children*) : le nombre d'enfants direct d'une classe.
- TNIM (*Total Number of Inherited Methods*) : le nombre total de méthodes héritées par une classe.
- TNFC (*Total Number of Function Calls*) : le nombre total d'appels de fonctions.

1.3 GAME

GAME (Generic Automated Marking Environment) est un logiciel de correction automatique développé à l'Université de Griffith et basé sur un autre outil plus ancien appelé *C-Marker* (Blumenstein et al., 2004). Pour évaluer un programme, *GAME* examine quelques aspects de qualité et des résultats d'exécution de tests. Du point de vue qualité, *GAME* calcule essentiellement les métriques suivantes (Blumenstein et al., 2004) :

- Le nombre de blocs de commentaires par rapport au nombre de fonctions.

- Le nombre de déclarations valides de variables globales et locales.
- Le nombre de *constantes magiques* (voir section 2.2.2).

Une deuxième partie de l'évaluation consiste à examiner les résultats de l'exécution du programme de l'étudiant selon l'une ou l'autre des deux stratégies suivantes :

Mot-clé : vérifier la présence d'un mot-clé dans le résultat de l'exécution

Mots-clés ordonnés : vérifier la présence d'une liste ordonnée de mots-clés dans le résultat de l'exécution

Dans tous les cas, tout ce qui n'est pas mot-clé est ignoré au cours de l'analyse du résultat. Cette approche permet de pouvoir accepter plusieurs solutions possibles et, surtout, évite de rejeter a priori une solution tout simplement parce que le programme de l'étudiant retourne une réponse qui n'est pas syntaxiquement identique à ce qui est prévu dans le test. A titre d'illustration, si on s'attend à ce que le programme affiche "5" en résultat et que le programme de l'étudiant affiche "réponse = 5", GAME va pouvoir accepter ce résultat alors que d'autres outils vont le rejeter.

1.4 PMD

PMD est un outil d'analyse de la qualité de code source (PMD, 2009). Il supporte notamment le langage Java. On peut l'utiliser en ligne de commande ou l'intégrer dans un outil de développement comme Eclipse, BlueJ, NetBeans, JBuilder, etc. *PMD* prend en charge plusieurs défauts mais, dans la pratique, on ne va souvent s'intéresser qu'à une liste limitée de défauts pour l'analyse d'un code source donné. Il faut donc préciser dans un fichier de configuration (*ruleset*) les défauts à vérifier. Dans ce fichier, les entrées (*rules*) correspondent aux défauts à vérifier au moment de l'analyse du code source.

PMD est un bon outil pour analyser un code source mais il ne se prête pas encore tout à fait à la correction automatique. Il y a aussi des limitations qui n'incitent pas à l'intégrer directement dans un outil de correction automatique. En effet, dans un contexte d'évaluation de travaux d'étudiants et d'attribution de notes, *PMD* a les désavantages suivants :

- Il ne calcule pas les métriques de la plupart des défauts qu'il détecte, ce qui nécessite d'effectuer un traitement sur le rapport fourni pour calculer ces métriques.
- Les règles (*rules*) de vérification des défauts s'écrivent en Java ou XPath, ce qui peut nécessiter un apprentissage supplémentaire non négligeable. En effet, dans un cas comme dans l'autre, il faut bien avoir en tête la structure d'arbre syntaxique abstrait (AST) construit par *JavaCC*. Dans le cas particulier des cours d'initiation à la programmation, il y a souvent plusieurs travaux à corriger et, pour chaque travail, on pourrait avoir des défauts spécifiques à vérifier. Il faudrait donc, idéalement, avoir au moins un ensemble de règles (*ruleset*) pour chaque travail à corriger. La difficulté vient du fait que les règles (*rules*), et en particulier les positionnements des éléments dans l'arbre AST, doivent être sans faute. Une erreur dans une règle aurait pour conséquence une mauvaise correction avec des risques de récrimination de la part des étudiants.

Comme nous venons de le voir, dans un contexte d'apprentissage, *PMD* est intéressant si on l'intègre dans les outils de développement. Il permettrait aux étudiants de vérifier eux mêmes les défauts et les bons styles de programmation. Quant à la correction automatique, il ne s'y prête pas encore tout à fait. L'absence de calcul des métriques et les contraintes d'écriture des règles ne facilitent pas non plus l'intégration complète de *PMD* dans un outil de correction automatique.

1.5 Checkstyle

Checkstyle est un outil pour aider les programmeurs à écrire des codes source respectant certains standards (Burn, 2001). C'est notamment le cas de la convention *Sun* (Sun Microsystems, 1999). Pour limiter la vérification aux défauts pertinents au programme à analyser, *Checkstyle* permet d'écrire un fichier de configuration XML. L'écriture du fichier de configuration est plus simple qu'avec *PMD*. Le rapport produit indique les numéros des lignes des défauts détectés. Il n'y a par contre qu'un nombre limité des métriques. Ce sont essentiellement des métriques de complexité et de couplage.

Pour les défauts plus élémentaires comme ceux sur les noms de constantes ou de variables, on peut en détecter certains, mais les métriques associées ne semblent pas avoir été implémentées.

Le fait de ne pas fournir directement les valeurs des métriques correspondant aux défauts pertinents ne facilite pas l'intégration dans un outil de correction automatique qui aurait besoin de ces valeurs pour attribuer des notes aux étudiants. L'écriture du fichier de configuration exige aussi de bien connaître les règles syntaxiques de XML.

1.6 JMT

JMT (Java Measurement Tool) est un outil graphique développé à l'Université de Magdeburg et qui permet de faire une analyse statique des classes et des relations entre des classes Java (Kolbe, 2002). Il calcule plusieurs métriques et affiche le résultat sur une interface graphique. Il ne génère pas de rapport et n'indique pas non plus les numéros de lignes des éléments mesurés sur le code source.

1.7 Eclipse Metrics

Eclipse Metrics est un module (*plugin*) de la plateforme de développement *Eclipse* (voir section 3.1) et qui permet d'analyser les dépendances et de calculer les métriques d'un code source Java (Sauer, 2002). Comme pour de nombreux autres outils, les métriques calculées mesurent surtout les concepts plus avancés comme le couplage, l'héritage, la complexité, etc. Certains de ces concepts ne sont d'ailleurs pas abordés dans les cours d'initiation à la programmation.

1.8 AutoGrader

AutoGrader est un outil de correction de programmes Java développé à l'Université de Miami (Helmick, 2007). Pour la vérification fonctionnelle, *AutoGrader* exécute des tests unitaires pour vérifier les résultats. Pour faciliter cette tâche et, surtout, uniformiser les entrées et sorties des fonctions à vérifier dans les programmes des étudiants,

l'enseignant fournit une *interface* Java avec l'énoncé. Les étudiants doivent implémenter cette interface. Pour une évaluation donnée, la classe Java doit porter un même nom pour tous les étudiants.

Pour vérifier le style de programmation (qualité), l'auteur (Helmick, 2007) indique qu'ils utilisent *PMD* (PMD, 2009). Il suggère aussi la possibilité d'utiliser *Checkstyle* (Burn, 2001).

Il convient de préciser qu'*AutoGrader* est essentiellement un outil pour uniformiser les entrées et sorties des fonctions en vue de vérifier leur fonctionnement avec des tests unitaires. Pour chaque test, le résultat indiquera s'il y a eu succès ou échec.

1.9 HoGG

HoGG (Homework Generation and Grading) est un logiciel de correction automatique de travaux de programmation développé à l'Université Rutgers (Morris, 2003). Dans la pratique, le logiciel corrige les programmes Java des devoirs de cours d'introduction à la programmation. Le principe de fonctionnement de HoGG est le suivant (Morris, 2003) :

- Le devoir est énoncé sous forme d'un ensemble de spécifications. Notamment, ce peut être un découpage de l'énoncé en fonctions effectuant chacune une tâche précise.
- L'enseignant développe un test spécifique à chaque spécification.
- Les étudiants sont supposés soumettre des programmes qui compilent. Un programme qui ne compile pas obtient la note 0.
- HoGG va tester chaque spécification (exemple : chaque fonction) séparément et mettre le résultat (succès ou échec) dans un vecteur.

Le programme de l'étudiant et ses résultats sont enregistrés dans une base de données. De ces enregistrements, on génère plusieurs rapports dont :

- Un rapport envoyé à l'étudiant par courriel et contenant une description de chacun des tests effectués et le vecteur des résultats de ces tests.

- Un autre rapport envoyé à l'assistant par courriel et contenant la liste des étudiants ainsi que la note et le vecteur des résultats de chacun.

Dans la pratique, les auteurs indiquent que *HoGG* peut aboutir à des erreurs d'évaluation. C'est notamment le cas si le nom de la fonction n'est pas écrit exactement comme prévu dans le test. Les auteurs de l'article (Morris, 2003) ont d'ailleurs relevé tout un ensemble de situations potentiellement problématiques et ils ont proposé quelques solutions.

Malgré l'utilisation concrète de *HoGG* à grande échelle à l'université de Rutgers, on peut tout de même se demander si c'est une solution que l'on pourrait utiliser aisément partout. En effet, il faut prévoir une période d'adaptation et d'acquisition de l'expertise nécessaire pour développer des énoncés et des tests minimisant les erreurs de correction. Malgré leur expérience, les auteurs s'attendent à une charge de travail de 8 personnes-heures (*person-hours*) pour développer les spécifications et les tests nécessaires pour un devoir. Pour la correction d'un tel devoir, ils s'attendent à des erreurs de correction de l'ordre de 5 %. Pour utiliser *HoGG*, il faut aussi prévoir la gestion de ces erreurs de correction.

Le système est utile pour la vérification fonctionnelle et la vérification de l'exactitude des résultats mais n'est pas adapté pour vérifier la qualité du code. En effet, l'article (Morris, 2003) évoque des pistes de travaux futurs pour la détection de plagiat mais, a priori, rien ne semble prévu pour analyser le code source lui-même.

1.10 Positionnement de DDPJ

Le logiciel de détection de défauts que nous présentons dans ce mémoire s'appuie sur des calculs de métriques pour analyser la qualité du programme. Étant donné l'objectif initial d'intégrer, à terme, notre logiciel dans *Oto*, nous ne nous occupons pas des vérifications fonctionnelles. Nous concentrons nos efforts sur certaines vérifications de style et de structure. Au lieu d'écrire un script ou un fichier de configuration pour chaque évaluation, notre logiciel propose tout simplement de passer, en ligne de commande, les

noms des métriques correspondant à ce qu'on veut vérifier. Ceci réduit considérablement la tâche de l'enseignant et lui permet de faire plus rapidement les vérifications qui lui semblent pertinentes.

Contrairement aux autres outils, notre logiciel introduit un concept de seuil et de bornes pour les métriques à rapporter pour l'évaluation. En effet, lors d'une correction traditionnelle par un humain, l'enseignant peut estimer qu'un nombre limité d'occurrences d'un certain défaut peut être acceptable. C'est notamment le cas des «nombres magiques». Le correcteur peut juger acceptable le fait d'avoir une ou deux constantes en dur dans un code d'un certain volume. Ce n'est pas pour autant qu'il veuille que l'on utilise les constantes en dur partout. Dans la plupart des outils actuels, une telle situation est difficile à gérer. Notre logiciel permet de spécifier des bornes pour ne pas signaler de problème tant que les valeurs des métriques respectent un certain intervalle. Cette approche permet, dans le cas de la correction, de ne pas sanctionner ce que le correcteur considère comme acceptable.

Dans le rapport détaillé par contre, toutes les occurrences seront indiquées avec leur emplacement dans le code source. Ceci permet à l'étudiant de voir tout de même ses erreurs mêmes si ces dernières n'ont pas été pénalisées dans l'évaluation. Précisons que dans une perspective de correction automatique, la note à attribuer sera calculée en fonction des valeurs des métriques. C'est pour cette raison que l'on ne signale que les valeurs qui sortent de l'intervalle admissible. Si le correcteur ne précise pas les bornes d'intervalle alors toutes les occurrences du défaut seront signalées.

Contrairement à *PMD* et autres outils de vérification de la qualité du code, notre logiciel permet d'obtenir tant le rapport détaillé avec des indications des emplacements des défauts que le calcul des métriques associées. Le calcul des métriques est particulièrement important pour la correction automatique car il permet de chiffrer l'ampleur des défauts. Contrairement à *PMD* qui exige des compétences avancées pour définir les règles à vérifier, notre logiciel ne nécessite que de passer les noms des métriques en arguments. Le correcteur peut donc vérifier plus rapidement et à volonté tout ce qu'il veut.

Toutefois, le correcteur doit évidemment connaître et comprendre le rôle de chacune de ces métriques.

JMT et *Eclipse Metrics* calculent bien plusieurs métriques mais ils ne sont pas adaptés pour la correction automatique car ils fonctionnent principalement en mode interactif avec par l'intermédiaire d'une interface graphique. Il est par conséquent plus compliqué de les exécuter automatiquement sur plusieurs programmes distincts à analyser. Bien qu'il existe des stratégies [1] pour exécuter le *plugin Metrics* en ligne de commande avec `ant`, cela nécessite d'organiser le code source sous forme de projet *Eclipse*.

Dans cette première version, notre outil se focalise sur les défauts souvent présents dans les programmes écrits par des programmeurs débutants. Ce sont surtout ces défauts qui intéressent les enseignants des cours d'initiation à la programmation. Les métriques plus avancées comme celles calculant la complexité ou le couplage et que l'on retrouve dans des outils comme *Checkstyle* sont intéressantes mais ne sont pas forcément adaptées aux travaux d'initiation à la programmation. En effet, dans les travaux des apprentis programmeurs, il n'y a souvent qu'une seule classe et voire même une seule méthode, `main()`. La préoccupation de l'enseignant est donc souvent de vouloir vérifier l'assimilation, par les étudiants, de certains concepts bien précis comme ceux associés aux métriques implémentées dans notre outil (voir annexe A.3).

1. Voir `metrics.sourceforge.net`

CHAPITRE II

RÉPERTOIRE DES DÉFAUTS TYPIQUES

Dans ce chapitre, on dresse une liste des défauts et «instructions dangereuses» que l'on rencontre typiquement dans les codes des programmeurs débutants. Ces défauts ne sont généralement pas détectés par les compilateurs habituels. Ce sont souvent des codes qui sont syntaxiquement corrects, mais qui peuvent conduire à des problèmes tels que :

- Des erreurs au moment de l'exécution (*runtime error*)
- Des résultats erronés
- Un comportement différent de ce que voulait le programmeur
- Une mauvaise qualité du code
- Des difficultés de maintenance

Pour chacun des défauts énumérés dans ce chapitre, on donne une description et un exemple. Ce chapitre s'inspire du travail présenté dans le cours de compilation que nous avons suivi à l'été 2011 (Tsheke, 2011). Dans la suite de ce chapitre, les défauts seront regroupés par catégories selon ce qu'ils affectent directement dans le programme.

2.1 Catégorie des variables

Cette section présente les défauts spécifiques aux variables.

2.1.1 Initialisation inutile

Description du défaut

La description de ce défaut a été proposée par le professeur Guy Tremblay. L'initialisation inutile consiste à

- initialiser une variable dans une méthode (éventuellement `main`)
- ne pas du tout l'utiliser ou la toute première utilisation après initialisation est une affectation.
- les variables des boucles «for» ne sont pas concernées.
- la première utilisation ne se fait pas dans un bloc conditionnel.

L'initialisation était donc inutile parce qu'on aurait pu initialiser la variable à sa première utilisation et que la valeur initiale affectée n'a jamais été utilisée.

Exemple du défaut : Voir listing 2.1.

Listing 2.1 Initialisation Inutile

```
int a = 5; //initialisation inutile
... //code ne contenant pas la variable a
a = 0; //pas dans un bloc conditionnel: Initialiser ici!
```

2.1.2 Nom de variable non significatif

Description du défaut

La description suivante a été faite en collaboration avec la professeure Naouel Moha. Ce défaut consiste à avoir un nom de variable qui n'a aucun sens par rapport au contexte ou qui ne suggère aucune signification particulière. Dans le cas d'un langage orienté objet comme Java, on pourrait suggérer que les noms des variables de type non primitif puissent avoir un préfixe commun avec ce type. On pourrait aussi déconseiller les noms des variables d'instance ou de classe (static) composés d'une seule lettre.

Exemple du défaut : Voir listing 2.2.

Listing 2.2 Nom de variable non significatif

```
class Essai{
        private int i; //nom de variable d'instance non significatif
        void exemple(){
                Person voiture = new Person();//nom de variable locale ↙
                    ↳ non significatif
        }
}
```

2.1.3 Déclarations multiples sur une seule ligne

Description du défaut

Ceci n'est pas vraiment un défaut mais une recommandation de la convention Sun (Sun Microsystems, 1999) pour faciliter les commentaires des variables. C'est une question de qualité du code et de lisibilité.

Exemple du défaut : Voir listing 2.3.

Listing 2.3 Déclarations multiples sur une ligne

```
int i,j; // 2 variables declarees sur une seule ligne
```

2.1.4 Variable locale non initialisée à la déclaration

Description du défaut

Ceci est encore une recommandation de Sun (Sun Microsystems, 1999) pour que les variables locales soient initialisées là où elles sont déclarées, sauf si la valeur initiale dépend d'un calcul. On exclut aussi le cas où la valeur initiale dépendrait d'une certaine condition (`if`, `while`, etc.). On attire l'attention du lecteur sur le fait qu'il n'y a pas de

contradiction avec le défaut de la section 2.1.1. En effet, si la variable est initialisée à la déclaration et que la première utilisation suivant la déclaration est une affectation, alors on a deux cas de figure :

- La valeur affectée après la déclaration provient d'un calcul ou de la vérification de certaines conditions : dans ce cas, la présente section autorise de ne pas initialiser la variable à la déclaration. En gardant seulement l'affectation suivant la déclaration, on respecte les deux conditions.
- Dans le cas contraire, il suffit de déclarer la variable à sa première utilisation ou de supprimer l'affectation et d'initialiser la variable avec la valeur appropriée à la déclaration.

Exemple du défaut : Voir listing 2.4.

Listing 2.4 Variable non initialisée à la déclaration

```
int i; // declaration sans initialisation
int j = 1; // i pas encore initialisee
 i = j++; //on pourrait declarer i avec initialisation ici
```

2.1.5 Variable globale masquée par une variable locale

Description du défaut

Ici, c'est encore un problème de qualité, selon la convention Sun (Sun Microsystems, 1999). Ce défaut consiste à déclarer une variable locale qui porte exactement le même nom qu'une variable (globale) déjà déclarée au niveau supérieur. Cela peut apporter une certaine confusion et affecter la qualité du code.

Listing 2.5 Variable globale masquée par une variable locale

```
class Essai{
        int prix = 0;//variable globale
        void exemple(){
                int prix = 5;//variable locale qui masque la variable ↙
                    ↳ globale
        }
}
```

Exemple du défaut : Voir listing 2.5.

2.1.6 Utilisation d'une variable d'instance initialisée sous condition

Description du défaut

Ce défaut (Skevoulis et Xiaoping, 2000; Xiaoping et al., 1999) consiste à utiliser une variable de classe ou d'instance initialisée sous condition, en dehors du bloc d'initialisation. La variable pourrait alors retourner la valeur par défaut du système (`null`, 0, etc.), ce qui ne correspond pas nécessairement au souhait du programmeur. Dans certains cas, cela pourrait engendrer une erreur d'exécution (*runtime error*) ou un résultat erroné.

Exemple du défaut : Voir listing 2.6.

2.1.7 Variable de classe non initialisée

Description du défaut

Ce défaut (Xiaoping et Skevoulis, 1999) consiste à déclarer une variable de classe sans l'initialiser. Cela entraîne souvent des erreurs d'exécution ou des résultats erronés.

Listing 2.6 Variable d'instance initialisée sous condition (Skevoulis et Xiaoping, 2000)

```
public class Essai {
        //Rappel: pas de "static" dans la declaration de variable d'instance
        String s2;
        int i;
        void  message(){
                int j = 0;
                if(j > 0){
                        s2 = "Blabla";
                        i = 10;
                }
                //on quite le bloc d'initialisation conditionnelle
                System.out.println("s2 = " + s2 + " et i = "+ i);
        }
        ...
}
```

Listing 2.7 Variable de classe non initialisée

```
public class Essai {
        static int nombre;//declaration de variable de classe non initialisee
        //Rappel: "static" dans la declaration indique une variable de classe
        ...
}
```

Exemple du défaut : Voir listing 2.7.

2.1.8 Variable déclarée hors du bloc d'utilisation

Description du défaut

La spécification de ce défaut est une combinaison des propositions du professeur Tremblay et des recommandations des emplacements des déclarations de variables faites dans la convention Sun (Sun Microsystems, 1999). Il est recommandé de déclarer les variables au *début* du bloc *le plus interne* tout en préservant la sémantique de leurs points d'utilisation. Ceci permet d'éviter la confusion en indiquant clairement les variables spécifiques au bloc.

Exemple du défaut : Voir listing 2.8.

Listing 2.8 Variable déclarée hors du bloc d'utilisation

```
int i = 0;
int j = 0; // j declaree hors son bloc d'utilisation
if(i < MAX){// seul bloc d'utilisation de j
        ...
        j = i;
        ...
}
```

2.2 Catégorie des constantes

Cette section comporte des défauts spécifiques aux constantes.

2.2.1 Nom de constante de classe non en majuscule

Description du défaut

Ceci est un défaut de style (Sun Microsystems, 1999) par rapport à la convention Java de Sun qui veut que les noms des constantes des classes soient en lettres majuscules et les mots séparés par un soulignement (_).

Exemple du défaut : Voir listing 2.9.

Listing 2.9 Nom de constante de classe non en majuscule

```
public class Essai {
        static final int TailleMax = 10;
        //on devrait avoir TAILLE_MAX=10;
        ...
}
```

2.2.2 Utilisation de nombres magiques

Description du défaut

Les nombres littéraux ne devraient pas être codés directement (Sun Microsystems, 1999) mais sous forme de constantes. Les seules exceptions sont les nombres -1,0 et 1 que l'on pourrait trouver dans les compteurs (boucle).

Exemple du défaut : Voir listing 2.10.

Listing 2.10 Utilisation de nombres magiques

```
int i = 0;
while(i < 10){ //utilisation du nombre litteral 10
        ...
}
```

2.3 Catégorie des opérateurs

Cette catégorie est constituée de défauts spécifiques aux opérateurs.

2.3.1 Confondre l'affectation et l'égalité

Description du défaut

Ce défaut (M874, 2003; Liang, 2008; Hristova et al., 2003) arrive souvent quand on ne fait pas attention que l'égalité est "==" et qu'un simple "=" est une affectation.

Exemple du défaut : Voir listing 2.11.

Listing 2.11 Confondre l'affectation et l'égalité

```
boolean b = false;
boolean c = true;
...
if(b = c){ // = confondue avec ==
        System.out.println("egalite");
}
```

2.3.2 Opérateur de préfixe ou suffixe dans une affectation

Description du défaut

Ce défaut est rapporté dans une référence bibliographique (M874, 2003) et par la professeure Louise Laforest. Il consiste à affecter à une variable une autre variable qui a elle même un opérateur préfixe ou suffixe. Souvent le programmeur débutant ignore d'une part que les valeurs des deux variables vont être modifiées et, d'autre part, l'ordre dans lequel ces modifications vont intervenir. Ce qui donne parfois lieu à un comportement non désiré.

Exemple du défaut : Voir listing 2.12.

Listing 2.12 Opérateur de préfixe ou suffixe dans une affectation

```
int i = 0;
int j = ++i; //apres execution i devient 1 et j devient 1
j = i++; //apres execution j devient 1 et i devient 2
i = i++;// i devient 2 puis est ensuite incremente et devient 3
```

2.3.3 Chaînes des caractères comparées avec "==" au lieu de ".equals"

Description du défaut

Ce défaut consiste à comparer les chaînes des caractères avec "==" au lieu de ".equals". En effet, en Java, les chaines sont des objets de la classe `java.lang.String` (Ziring, 2011) et "==" compare seulement les adresses mémoires (Hristova et al., 2003).

Exemple du défaut : Voir listing 2.13.

Listing 2.13 Chaines des caractères comparées avec "==" au lieu de ".equals" (Ziring, 2011)

```
if (args[0] == "-a"){
        optionsAll = true;
}
```

2.4 Catégorie des structures de contrôle

Dans cette section, on aborde des défauts propres aux structures de contrôle.

2.4.1 Modification de la variable d'itération d'une boucle `for`

Description du défaut

Ce défaut consiste à affecter une valeur à la variable d'itération d'une boucle `for` à l'intérieur même de la boucle. Cela peut être dangereux et, dans certains cas, pourrait même donner lieu à une boucle infinie.

Exemple du défaut : Voir listing 2.14.

Listing 2.14 Modification de la variable d'itération d'une boucle `for`

```
for(int i = 0;i < MAX;i++){
        ...//code eventuel ne modifiant pas i
        i = 2; //modification de la variable d'iteration dans la boucle "for"
        ...//code eventuel ne modifiant pas i
}
```

2.4.2 `If` inutile

Description du défaut

Ce défaut consiste à faire un test `if` puis de ne rien en faire ou d'avoir un `else` qui ne fait rien. Il peut se présenter sous plusieurs formes. Parfois, il arrive suite à un point virgule (;) qui vient séparer le `if` de l'accolade ouvrante de son bloc (Liang, 2008). Il peut aussi arriver parce que le programmeur a commenté tout le code du bloc mais a oublié le `if` (ou le `else`) ne servant plus à rien.

Listing 2.15 `If` inutile (Liang, 2008)

```
boolean done = false;
if (done); //le point-virgule separe le if de son bloc
{
        System.out.println("done");
}
```

Exemple du défaut : Voir listing 2.15.

2.4.3 `If` sans accolades

Description du défaut

Pour faciliter l'ajout éventuel d'autres instructions, la convention Sun (Sun Microsystems, 1999) suggère de mettre le corps de l'instruction `if` entre accolades même s'il n'y a qu'une seule instruction. Ce défaut se distingue du *`if` inutile* (2.4.2) par le fait que son corps n'est pas vide.

Exemple du défaut : Voir listing 2.16.

Listing 2.16 `If` sans accolades

```
if(true)
        System.out.println("ceci devrait etre entre des accolades");
```

2.4.4 Certains `while` problématiques

Description du défaut

Le but cherché ici est d'attirer l'attention du programmeur sur des boucles `while` dont certaines indications font craindre une boucle infinie. Dans ce travail, on va s'intéresser aux boucles `while` dont aucune variable de l'expression (condition) n'est modifiée dans la boucle.

Exemple du défaut : Voir listing 2.17.

Listing 2.17 Certains `while` problématiques

```
while(i < 5)
{
        ...//bloc ne modifiant pas la variable i
}
//une autre forme
while(i<5);//le point virgule separe le while du corps de la boucle
{
        ...
        i++;
        ...
}
```

2.4.5 `While` potentiellement convertible à `for`

Description du défaut

Ce défaut est le fait d'avoir une boucle `while` dont le nombre d'itérations peut être déterminé avant même l'exécution. C'est souvent quand la boucle évolue en fonction d'une seule variable incrémentée (décrémentée) par une constante à chaque itération. Une telle boucle pourrait donc être remplacée par une boucle `for`. Cette description a été proposée par le professeur Tremblay et complétée avec les suggestions du professeur Gagnon.

Listing 2.18 `While` potentiellement convertible à `for`

```
while (i<5) {
        ...//pas de modification de i avant incrementation
        i = i + 1; //modification de la variable d iteration
        ...//pas de i apres incrementation
}
```

Exemple du défaut : Voir listing 2.18.

2.4.6 For sans corps de la boucle

Description du défaut

Ce défaut (Hristova et al., 2003) arrive lorsqu'un point-virgule vient séparer le `for` du corps de la boucle. Ce peut aussi être le cas quand le programmeur commente ou supprime tout le code du corps.

Exemple du défaut : Voir listing 2.19.

Listing 2.19 Instruction `for` sans corps de boucle

```
for(int i = 0;i < 5;i++); {
        ...//Corps de la boucle separe de "for" par un point-virgule
}
//un autre exemple
for(int i = 0;i < 5;i++){
        //Corps de la boucle sans code
}
```

2.4.7 Oubli de `break` après un `case` dans une instruction `switch`

Description du défaut

Ce défaut (Liang, 2008; M874, 2003) arrive fréquemment quand on ignore les effets de bord de `case`. De nombreux débutants ne réalisent pas que dans un `switch`, à la fin d'un `case`, c'est le `case` suivant qui sera exécuté s'il n'y a pas d'instruction `break`.

Exemple du défaut : Voir listing 2.20.

Listing 2.20 Oubli de `break` après un `case` dans une instruction `switch`

```
char ch = 'a';
switch (ch) {
case: 'a': System.out.print(ch);
case: 'b': System.out.print(ch);
case: 'c': System.out.print(ch);
}
```

2.5 Catégorie des tableaux et des chaînes de caractères

Cette catégorie regroupe les défauts spécifiques aux tableaux et aux chaînes de caractères.

2.5.1 Risque de dépassement de l'indice du tableau

Description du défaut

Cette erreur (Liang, 2008) arrive fréquemment. Souvent, les débutants ne réalisent pas que les indices des tableaux commençant à 0 se terminent à l'indice `nombre d'éléments - 1`.

Listing 2.21 Risque de dépassement de l'indice du tableau

```
for (int i = 0; i <= x.length(); i++){//depassement lorsque i =x.length()
        System.out.print(x[i]);
}
```

Exemple du défaut : Voir listing 2.21.

2.5.2 Appeler `length()` sans initialiser la chaîne

Description du défaut

Ce défaut (Skevoulis et Xiaoping, 2000) consiste à faire appel à la méthode `length()` d'une chaîne de caractères (`String`) qui n'a pas été initialisée. Cela n'est pas détecté par le compilateur mais génère une erreur au moment de l'exécution (*Runtime error*).

Exemple du défaut : Voir listing 2.22.

Listing 2.22 Appeler `length()` sans initialiser la chaîne (inspiré de (Skevoulis et Xiaoping, 2000))

```
public class Essai {
        String s2;
        void  message(){
                System.out.println("Message");
                if(s2.length() > 0){ //appel de s2.length() sans initialiser s2
                        System.out.println("haha");
                }
        }
        ...
}
```

2.5.3 Allouer un tableau avec un nombre négatif ou nul d'éléments

Description du défaut

Ce défaut (Xiaoping et Skevoulis, 1999) consiste à déclarer un tableau avec une taille inférieure ou égale à zéro. Cette erreur n'est malheureusement pas détectée par le compilateur. Elle donne lieu à une erreur d'exécution.

Exemple du défaut : Voir listing 2.23.

Listing 2.23 Allouer un tableau avec un nombre négatif ou nul d'éléments

```
public class Essai {
        void  message() {
                int[] tab = new int[-10];//taille de tableau <=0
        }
        ...
}
```

2.6 Catégorie des fonctions/méthodes

Dans cette section, on présente des défauts spécifiques aux fonctions ou méthodes.

2.6.1 Paramètre non utilisé dans une méthode

Description du défaut

Ce défaut (Tremblay, 2011) consiste à déclarer un paramètre dans l'entête d'une méthode et de ne pas du tout l'utiliser dans le corps de cette méthode.

Listing 2.24 Paramètre non utilisé dans une méthode

```
void exemple(int param){
        System.out.println("param n est jamais utilise dans la methode");
}
```

Exemple du défaut : Voir listing 2.24.

2.6.2 Trop de paramètres

Description du défaut

Pour la bonne qualité, il est souhaitable, dans des conditions normales, de ne pas avoir plus de 7 paramètres dans une méthode (Tremblay, 2011). Dans la deuxième édition de son livre *Code Complete*, Steve McConnell justifie ce nombre en s'inspirant des expériences psychologiques de Miller de 1956. Ces expériences ont montré que généralement les gens ne peuvent pas mémoriser une trace de plus d'environ sept (7) blocs d'information à la fois.

Exemple du défaut : Voir listing 2.25.

Listing 2.25 Trop de paramètres

```
void exemple(int param1, boolean param2, int param3, int param4,
                int param5, int param6, String param7, int param8){
        ...//corps de la methode. param8 est de trop
}
```

2.7 Catégorie des autres défauts

Dans cette catégorie on trouve des défauts dont la description ne cadre pas tout à fait à aucune des catégories spécifiques énumérées précédemment.

2.7.1 Nom de classe Java différent du nom de fichier

Description du défaut

Ce défaut (Ziring, 2011) consiste à donner à une classe Java, un nom qui n'est pas identique (avec sensibilité à la casse) au nom (sans tenir compte de l'extension .java) du fichier la contenant. Il convient de préciser que certains compilateurs détectent ce défaut mais d'autres l'ignorent ou le tolèrent.

Exemple du défaut : Voir listing 2.26.

Listing 2.26 Nom de classe Java différent du nom de fichier

```
//fichier essai.java avec e minuscule
public class Essai {
        //nom de la classe avec E majuscule
        ...//code de la classe
}
```

2.7.2 Déclarer un constructeur comme une méthode

Description du défaut

Ce défaut (Ziring, 2011) arrive lorsque, par mégarde, on donne un type de retour (éventuellement `void`) à un constructeur. Le compilateur le traite alors comme une méthode et non plus comme un constructeur.

Listing 2.27 Déclarer un constructeur comme une méthode

```
public class Essai {
        public void Essai() {
                //le constructeur devient une methode a cause de "void"
                ...
        }
}
```

Exemple du défaut : Voir listing 2.27.

2.7.3 Caractère spécial dans un identifiant

Description du défaut

Ce défaut consiste à utiliser un caractère spécial (accent, etc.) dans un identifiant (nom de variable, fonction, etc.). En théorie, la plupart de ces caractères sont autorisés mais, en pratique, ils peuvent conduire à des complications inutiles. C'est notamment le cas si on édite le code source avec des éditeurs n'utilisant pas le même jeux des caractères. Dans ces conditions, les caractères spéciaux ou accentués peuvent se transformer et conduire à une situation indésirable. Plusieurs enseignants dont Fabien Michel[1] et Nicolas Van Zeebroeck[2] déconseillent d'ailleurs l'utilisation de ces caractères dans les identifiants.

Listing 2.28 Caractère spécial dans un identifiant

```
public class Essai {
        String donnée = "DONNEE";
        //on recommande donnee="DONNEE";
        ...
}
```

Exemple du défaut : Voir listing 2.28.

2.7.4 Variable d'instance non initialisée dans un constructeur

Description du défaut

Ce défaut (Xiaoping et al., 1999) consiste à déclarer une variable d'instance et ne l'initialiser ni à la déclaration ni dans un constructeur donné. En effet, pour les classes sans constructeur, la référence à une variable d'instance non initialisée à la déclaration risque de produire une erreur (*Exception*). Ce même comportement s'observe dans le cas d'un objet créé avec un constructeur n'initialisant pas une variable donnée non initialisée à la déclaration.

Exemple du défaut : Voir listing 2.29.

2.7.5 Null affecté à une variable de classe

Description du défaut

Ce défaut (Xiaoping et al., 1999) survient quand on affecte `null` à une variable de classe. Comme tous les objets de la classe partagent les variables de classe (*static*), le fait d'affecter `null` à une variable de classe peut engendrer des erreurs dans l'une ou

---

1. http://www2.lirmm.fr/~fmichel/ens/java/cours/java_basics.pdf

2. http://cs.ulb.ac.be/public/_media/teaching/infoh301/courspolytechjava2011-1.pdf

Listing 2.29 Variable d'instance non initialisée dans un constructeur (exemple inspiré de (Xiaoping et al., 1999))

```
public class Essai {
        protected String nom; //variable d'instance -- pas static
        protected String prenom; //variable d'instance -- pas static
        //constructeur
        public Essai() {
                nom = "nom";
                //prenom n est pas initialisé
        }
        ...
}
```

l'autre instance (objet) de la classe.

Exemple du défaut : Voir listing 2.30.

Listing 2.30 Null affecté à une variable de classe

```
public class Essai {
        static String nom = "nom";//déclaration de variable de classe initialisée
        //Rappel: "static" dans la declaration indique une variable de classe
        static void  message() {
                nom = null;//variable de classe mise à "null"
        }
        ...
}
```

2.8 Défauts détectés par notre outil

La version actuelle de notre outil ne détecte pas encore tous les défauts énumérés dans ce chapitre. Le lecteur trouvera les informations sur ceux qui sont détectés actuellement au chapitre 4. Une liste plus précise de défauts détectés par notre outil est donnée à l'annexe A.3.

CHAPITRE III

OUTILS UTILISÉS POUR LA MISE EN OEUVRE DE DDPJ

Dans ce chapitre, nous présentons les principaux outils que nous avons utilisés pour le développement de *DDPJ*, notre logiciel de détection des défauts. Nous donnons aussi quelques explications sur ce qui a motivé les choix.

3.1 Plateforme de développement

Pour le développement de notre application, nous avons choisi d'utiliser la plateforme `eclipse` (voir `http://www.eclipse.org/`). `Eclipse` est un IDE (*Integrated Development Environment*) qui permet le développement dans plusieurs langages de programmation et facilite l'intégration de bibliothèques. Il facilite aussi la création de l'exécutable de l'application développée. Un autre avantage important d'`eclipse` est l'*autocompletion* qui assiste à la programmation.

En comparaison avec les autres IDE populaires comme `netbeans` [1] ou le framework `msdn` [2], `eclipse` offre les avantages suivants :

- `Eclipse` fonctionne sur les trois systèmes d'exploitation les plus répandus : Linux/Unix, Windows et Mac. Ce n'est pas le cas de `msdn`, par exemple, qui ne fonctionne que sous Windows.

---

1. `http://netbeans.org/`

2. `http://msdn.microsoft.com/`

- Face à **Netbeans** qui est aussi multi systèmes d'exploitation, une étude (Wang, Baik et Devanbu, 2011) qui a comparé les deux plateformes a suggéré qu'un projet **Eclipse** serait plus facile à maintenir et à déboguer.

En tenant compte des raisons présentées ci-dessus, nous avons choisi de développer sur la plateforme **eclipse**. De la sorte, le projet (code source) pourrait facilement être importé pour l'extension, l'évolution ou la maintenance éventuelle du logiciel.

3.2 SableCC

SableCC est un outil utilisé pour développer des compilateurs (SableCC, 2000). En fonction d'une grammaire donnée, il permet, notamment, de générer un programme spécial appelé *parseur*. Un *parseur* est un analyseur qui permet de transformer le code source d'un programme en une structure d'arbre (Arusoaie et Vicol, 2012) plus facile à manipuler.

Une de ces structures en arbre souvent utilisée est l'arbre syntaxique abstrait (AST : *Abstract Syntax Tree*) (Gagnon, 1998; Wikipedia, 2008). Dans cette représentation, on ne représente que les éléments essentiels. Ainsi, il n'y aura par exemple pas de noeud spécifique pour les signes de ponctuation délimitant les règles de la grammaire (Wikipedia, 2008).

Une façon d'analyser le code source d'un programme consiste à parcourir l'arbre AST de sa représentation et à examiner la situation dans chaque noeud. Pour parcourir un AST, on utilise souvent des programmes conçus spécifiquement à cet effet et qu'on appelle *visiteurs*. Les classes générées par SableCC permettent d'écrire facilement des *visiteurs* pour parcourir l'arbre en profondeur d'abord (*depth first search*) et accéder aux différents éléments du programme à examiner (Gagnon, 1998).

Le concept de base de SableCC a été initialement présenté et développé par Étienne-M. Gagnon dans son mémoire de maîtrise (Gagnon, 1998). Actuellement, SableCC est disponible sous forme de logiciel libre (SableCC, 2000).

3.2.1 Grammaire SableCC

Une grammaire définit les règles d'écriture d'un langage donné. Dans cette sous section, nous n'allons pas entrer en profondeur dans les règles d'écriture d'une grammaire SableCC mais nous voulons donner au lecteur une idée générale pour mieux comprendre la suite de ce document. Des explications plus détaillées sont disponibles sur le site de SableCC (SableCC, 2000). Comme le montre le listing 3.1, une grammaire SableCC est composée de sections `Helpers`, `Tokens`, `Ignored Tokens` et `Productions`.

3.2.2 Compilation de la grammaire

Pour compiler une grammaire SableCC, on peut utiliser une commande telle que la suivante[3] :

```
java -jar sablecc.jar -d src grammaire.sablecc
```

La compilation de la grammaire génère un paquetage des classes Java avec les quatre sous paquetages suivants :

- `analysis`
- `lexer`
- `node`
- `parser`

Nous n'entrerons pas dans les détails de ces paquetages car cela sortirait du cadre du présent travail. Le lecteur intéressé pourra consulter les références données en bibliographie (Bergmann, 2007; SableCC, 2000; Gagnon, 1998; Gagnon et Hendren, 1998). Nous allons cependant utiliser les classes Java de ces paquetages pour développer les

---

3. Dans cette commande, `src` est le répertoire de destination où sera placé le paquetage (*package*) des fichiers générés. Dans la version 3.2 que nous avons utilisée, le nom du paquetage est spécifié dans la grammaire SableCC comme le montre la ligne 4 du listing 3.1. Il est tout à fait possible que dans les prochaines versions, le nom du paquetage soit spécifié en ligne de commande de compilation et non dans le fichier de la grammaire comme c'est le cas maintenant.

Listing 3.1 Extrait de la grammaire SableCC de Java 1.5 : `extraitJava-1.5.sablecc`

```
1  /* Extrait de la grammaire Java 1.5.  Version complete sur sablecc.org
2  * Dans cet extrait "//..." indique qu'on saute une partie
3  */
4  Package sabccom.java_1_5;
5  Helpers
6
7    unicode_input_character = [0..0xffff];
8   // ... on saute une partie
9  Tokens
10
11   // separators
12   at = '@';
13   //...
14   // identifier
15   identifier = java_letter java_letter_or_digit*;
16    //...
17 Ignored Tokens
18
19   white_space,
20   //...
21 Productions
22
23   compilation_unit =
24     package_declaration? [import_declarations]:import_declaration* ↙
       ↳ [type_declarations]:type_declaration*;
25     //...
26   method_declarator =
27     identifier l_par formal_parameter_list? r_par [dims]:dim*;
28   //...
29   variable_declarator =
30     {simple}      identifier [dims]:dim* |
31     {initializer} identifier [dims]:dim* assign variable_initializer;
32   //...
```

visiteurs en vue d'analyser les codes sources. Il convient de préciser que les classes du sous paquetage `node` sont générées en fonction des définitions de la partie `Productions` de la grammaire.

3.2.3 Exemple d'un visiteur

Dans cette sous-section, nous allons montrer comment parcourir avec un visiteur un code source préalablement transformé en structure d'arbre par un analyseur syntaxique SableCC.

Compilons la grammaire Java 1.5 disponible sur le site Internet de SableCC (SableCC, 2000) et considérons le code source du listing 3.6 pour notre analyse. Supposons que nous voulons trouver les numéros des lignes où sont initialisées les variables ainsi que les lignes où sont déclarées les méthodes.

Pour examiner ce code source, nous utilisons le programme du listing 3.2. Les éléments clés pour comprendre le fonctionnement de ce programme sont les suivants :

- La lecture du fichier de code source à analyser se fait à la ligne 14.
- L'analyse lexicale se fait à la ligne 15. Ici, la séquence de caractères est transformée en une séquence de jetons (*tokens*). Les jetons sont des chaînes de caractères élémentaires comme les identificateurs, les constantes, etc. (Bergmann, 2007).
- L'analyseur syntaxique est instanciée à la ligne 16 et transforme le code source en structure d'arbre (AST) à la ligne 17.
- À la ligne 19, on instancie le visiteur du listing 3.3 qui sera utilisé à la ligne 20 pour visiter l'arbre généré par l'analyseur syntaxique.

Rappelons que l'analyseur lexical (`Lexer`) et l'analyseur syntaxique ou parseur (`Parser`) sont générés par SableCC à la suite de la compilation de la grammaire. Nous référons encore une fois le lecteur au livre de Bergmann (Bergmann, 2007) ainsi qu'aux manuels de référence de SableCC (SableCC, 2000) pour avoir des informations détaillées sur ces programmes.

Un programme visiteur est essentiellement composé des méthodes à appliquer

Listing 3.2 Exemple appels analyseurs et visiteur : Mainsc.java

```
1  package sabccom;
2  import java.io.BufferedReader;
3  import java.io.FileReader;
4  import java.io.PushbackReader;
5  import sabccom.java_1_5.lexer.Lexer;
6  import sabccom.java_1_5.node.Node;
7  import sabccom.java_1_5.parser.Parser;
8  import sabccom.visitors.*;
9  public class Mainsc {
10         public static void main(String[] args) {
11                 System.out.println("main pour Visiteur ");
12                 try {
13                         String filename="src/sabccom/Main.java";
14                         FileReader in = new FileReader(filename);
15                         Lexer lexer = new Lexer(new PushbackReader(new ↙
                             ↳ BufferedReader(in), 1020));
16                         Parser parser = new Parser(lexer);
17                         Node tree = parser.parse();
18                         in.close();
19                         Visiteur visiteur = new Visiteur();
20                         tree.apply(visiteur); // effectuer la visite
21                 }catch(Exception e){
22                         System.out.println(e.toString());
23                 }
24         }
25 }
```

Listing 3.3 Exemple d'un visiteur : `Visiteur.java`

```
1  package sabccom.visitors;
2  import sabccom.java_1_5.*;
3  import sabccom.java_1_5.analysis.*;
4  import sabccom.java_1_5.node.*;
5  public class Visiteur extends DepthFirstAdapter {
6
7          public Visiteur(){
8                  //mettre des initialisations eventuelles
9          }
10         private void visit(Node node) {
11                 if (node != null) {
12                         node.apply(this);
13                 }
14         }
15         @Override
16         public void caseStart(Start node) {
17                 System.out.println("debut de la visite");
18                 this.visit(node.getPCompilationUnit());
19         }
20         @Override
21         public void caseAInitializerVariableDeclarator(AInitializerVariableDeclarator ↙
                           ↳ node){
22                 String identifiant = node.getIdentifier().getText();
23                 int ligne = node.getIdentifier().getLine();
24                 System.out.println("variable: " + identifiant + " initialisee a la ↙
                           ↳ ligne: " + ligne);
25         }
26         @Override
27         public void caseAMethodDeclarator(AMethodDeclarator node){
28                 String identifiant = node.getIdentifier().getText();
29                 int ligne = node.getIdentifier().getLine();
30                 System.out.println("methode: " + identifiant + " declaree a la ligne: ↙
                           ↳ " + ligne);
31         }
32 }
```

selon la nature du noeud. Habituellement, on n'implémente que des méthodes associées aux noeuds cherchés et traités durant la visite. Les autres noeuds sont ignorés au cours de la visite.

Pour notre visiteur du listing 3.3, nous avons les méthodes suivantes :

- `caseStart` : Méthode spéciale qui est appelée au début de la visite d'un arbre du code source et qui s'applique au noeud racine.
- `visit` : Normalement, avec un visiteur, le passage d'un noeud à l'autre se fait de manière implicite. Dans certains cas, on aimerait visiter explicitement un noeud ou un sous arbre à un moment précis. C'est ce qu'on fait dans la méthode `visit`.
- À la ligne 21, on traite le cas de la déclaration avec initialisation d'une variable. On mettra donc dans cette méthode tout le traitement spécifique à la présence de ce cas de figure dans le code source à analyser.
- La méthode à la ligne 27 fait le traitement requis lorsqu'on rencontre une déclaration d'une méthode dans le code source à analyser.

Chacune des méthodes dont le nom commence par le préfixe `case` est normalement associé à un type de noeud dont la classe se trouve dans le paquetage `node` généré suite à la compilation de la grammaire (voir section 3.2.2).

En utilisant le visiteur du listing 3.3, nous obtenons les résultats du listing 3.4 suite à la visite du code source du listing 3.6.

3.2.4 Motivations du choix de SableCC

Malgré une documentation relativement limitée par rapport à d'autres outils comme *JavaCC*, nous avons choisi SableCC, notamment pour les raisons suivantes :

- La simplicité des règles syntaxiques d'écriture de la grammaire. A cela, il convient d'ajouter le fait que certaines grammaires sont disponibles sur le site de SableCC (`http://sablecc.org/`). C'est notamment le cas de la grammaire Java.
- La séparation du code généré avec le code écrit par le programmeur (Gagnon et Hendren, 1998), ce qui permet de recompiler facilement la grammaire au besoin.

Listing 3.4 Résultat de la visite du code du listing 3.6 avec le visiteur du listing 3.3

```
main pour Visiteur
debut de la visite
variable: MAX initialisee a la ligne: 7
variable: DIVISEUR_TROIS initialisee a la ligne: 8
variable: DIVISEUR_QUATRE initialisee a la ligne: 9
variable: DIVISEUR_CINQ initialisee a la ligne: 10
methode: afficherListes declaree a la ligne: 11
variable: mProgram initialisee a la ligne: 12
variable: multiplesDeTrois initialisee a la ligne: 13
variable: multiplesDeQuatre initialisee a la ligne: 14
variable: multiplesDeCinq initialisee a la ligne: 15
variable: i initialisee a la ligne: 16
methode: main declaree a la ligne: 29
variable: mObj initialisee a la ligne: 30
variable: min initialisee a la ligne: 31
variable: max initialisee a la ligne: 32
```

- La facilité de manipulation de l'AST généré et d'écriture des *visiteurs* pour le parcourir. Ceci permet d'accéder sans trop de complications aux différents éléments du programme à analyser.
- La facilité de le combiner avec *ObjectMacro* (voir la section suivante) pour le développement des applications affichant les résultats selon un gabarit précis.
- La vérification, par l'analyseur syntaxique généré, des erreurs de syntaxe détectables par le compilateur. Ceci permet de se concentrer sur les défauts non pris en charge par le compilateur.
- L'expertise antérieure que nous avions de cet outil. En effet, dans le passé, nous avons eu à travailler avec SableCC dans le cadre de certains travaux (Tsheke, 2011) au cours de notre scolarité. Nous avions alors pu expérimenter sa puissance et sa facilité d'utilisation.

3.3 ObjectMacro

ObjectMacro est une bibliothèque permettant de générer du texte formaté au moment de l'exécution de l'application (ObjectMacro, 2011). En pratique, il permet de créer un gabarit d'affichage des résultats. Dans cette section, nous allons donner un bref aperçu de la pertinence et de l'utilisation d'`ObjectMacro`. Le lecteur trouvera des informations détaillées dans les références données en bibliographie (SableCC, 2000; Bergmann, 2007).

Pour simplifier les explications et faciliter le lien entre la théorie et la pratique, nous allons présenter *ObjectMacro* à l'aide d'un exemple concret.

3.3.1 Problème d'affichage

On veut parcourir *séquentiellement* les nombres entiers de MIN à MAX avec *une seule boucle* et afficher *dans l'ordre*, les trois listes suivantes :

- liste des nombres multiples de 3
- liste des nombres multiples de 4
- liste des nombres multiples de 5

Concrètement, *tous* les nombres de chaque liste doivent être affichés avant la liste suivante. Si un nombre satisfait aux conditions de plusieurs listes, il devra figurer dans chacune de ces listes. C'est par exemple le cas de 12 qui est à la fois multiple de 3 et de 4.

3.3.2 Création du gabarit d'affichage

Il est difficile de résoudre le problème présenté ci-haut en utilisant les méthodes simples d'affichage (`print` ou `println`) sans manipuler explicitement des listes de nombres. Nous allons montrer comment l'utilisation d'*ObjectMacro* permet de le faire de façon relativement simple, en effectuant les affichages à l'aide d'un gabarit précis.

Pour écrire un gabarit *ObjectMacro*, il suffit de connaître quelques règles simples :

Listing 3.5 Exemple de gabarit *ObjectMacro* : `macroexemple-template.objectmacro`

```
1  $macro: program$
2  Affichage selon gabarit
3  ------------------------
4  $macro: multipledetrois(mult3)$
5  $(mult3)
6  $end: multipledetrois$
7  $comment: Cette ligne est en commentaire --$
8  $macro: multipledecinq(mult5)$
9  $(mult5)
10 $end: multipledecinq$
11 $comment: ----------------------$
12 $macro: multipledequatre(mult4)$
13 $(mult4)
14 $end: multipledequatre$
15 $comment: ---bloc des multiples de trois------$
16 $macro: blockmultiplesdetrois$
17
18 Liste des nombres multiples de 3
19 --------------------------------
20 $expand: multipledetrois$
21 $end: blockmultiplesdetrois$
22 $macro: blockmultiplesdequatre$
23
24 Liste des nombres multiples de 4
25 --------------------------------
26 $expand: multipledequatre$
27 $end: blockmultiplesdequatre$
28 $macro: blockmultiplesdecinq$
29
30 Liste des nombres multiples de 5
31 --------------------------------
32 $expand: multipledecinq$
33 $end: blockmultiplesdecinq$
34
35 $end: program$
```

- Une macro s'ouvre ou se déclare en mettant `$macro: nomMacro$` sur une nouvelle ligne. Dans cette déclaration, `nomMacro` est le nom de la macro. La ligne 1 du listing 3.5 déclare la macro `program`.
- La fermeture se fait avec le `$end: nomMacro$` sur une nouvelle ligne. La macro `program` se ferme à la dernière ligne du listing précédent.
- À l'intérieur de la macro, on formate le texte tel qu'on voudrait qu'il s'affiche. Dans le cas du listing 3.5, la macro `program` contiendra tout le rapport.
- Chaque macro définie à l'intérieur d'une autre spécifie une partie de texte qui *peut* s'afficher à cet endroit précis. C'est le cas de la macro `blockmultiplesdetrois` à l'intérieur de `program`. En principe, l'ordre d'apparition des macros déclarées directement à l'intérieur d'une autre détermine l'ordre d'affichage. Le listing précédent montre que la liste des multiples de 4 s'affichera avant la liste des multiples de 5.
- Si une macro `macroFils` est déjà déclarée en dehors de la macro `macroParent`, on peut inclure son texte en mettant `$expand: macroFils$` à l'emplacement désiré. À l'intérieur de la macro `macroParent`, l'ordre d'apparition des macros déclarées directement et des macros incluses avec `expand` détermine l'ordre d'affichage. Dans le listing précédent, on veut que le `blockmultiplesdetrois` contienne des instances de `multipledetrois`. C'est l'astuce qui permet de regrouper l'affichage de certaines informations.
- Pour passer un argument à une macro, on le définit comme à la ligne 4 (`mult3`). Cet argument peut, par la suite, être affiché en le mettant entre parenthèses précédé du signe de `$` comme à la ligne 5. L'argument de la macro est le texte qui sera obtenu au cours de l'exécution du programme. C'est notamment le cas des résultats des calculs ou des opérations.
- On peut ajouter des commentaires de la manière suivante sur une nouvelle ligne `$comment: un commentaire ici --$` La ligne 7 du gabarit du listing 3.5 montre un exemple de commentaire.
- Les lignes blanches servent à faciliter la lecture en ajoutant une ligne blanche à l'affichage.

Comme on vient de le voir dans cet exemple, tout le contenu du gabarit d'affichage de l'application doit être placé à l'intérieur de la macro `program`.

3.3.3 Compilation du gabarit

Après avoir créé un gabarit et avant de pouvoir l'utiliser, il faut d'abord le compiler. Pour compiler une macro `template.objectmacro` (gabarit) avec *ObjectMacro*, on utilise la commande suivante :

```
java -jar objectmacro.jar -d src -p nomPackage template.objectmacro
```

L'option `p` permet de spécifier le nom du paquetage qui sera généré puis placé dans le répertoire indiqué, `src` dans le cas présent.

La compilation va générer un paquetage Java avec plusieurs classes à raison d'une classe par macro déclarée. Les noms des classes correspondent aux noms des macros avec la première lettre en majuscule et préfixés de la lettre `M`. Dans le cas précis de notre gabarit du listing 3.5, nous utilisons la commande suivante.

```
java -jar objectmacro.jar -d src -p sabccom.macros macroexemple-template.objectmacro
```

Nous obtenons alors le paquetage Java `sabccom.macros` contenant les classes suivantes :

- `MProgram`,
- `MBlockmultiplesdetrois`,
- `MBlockmultiplesdequatre`,
- `MBlockmultiplesdecinq`,
- `MMultipledetrois`,
- `MMultipledequatre`,
- `MMultipledecinq`.

3.3.4 Utilisation du gabarit

Une fois compilé, le gabarit ObjectMacro est maintenant prêt pour utilisation. Écrivons à présent un programme pour résoudre notre problème de la section 3.3.1.

Listing 3.6 Exemple d'utilisation du gabarit objectMacro : `Main.java`

```
6  public class Main {
7          final static int MAX = 15;
8          final static int DIVISEUR_TROIS = 3;
9          final static int DIVISEUR_QUATRE = 4;
10         final static int DIVISEUR_CINQ = 5;
11         private void afficherListes(int min,int max) {
12                 MProgram mProgram = new MProgram();
13                 MBlockmultiplesdetrois multiplesDeTrois = ↙
                           ↳ mProgram.newBlockmultiplesdetrois();
14                 MBlockmultiplesdequatre multiplesDeQuatre = ↙
                           ↳ mProgram.newBlockmultiplesdequatre();
15                 MBlockmultiplesdecinq multiplesDeCinq = ↙
                           ↳ mProgram.newBlockmultiplesdecinq();
16                 for(int i=min; i<=max; i++) {
17                         if(i % DIVISEUR_TROIS == 0) {
18                                 multiplesDeTrois.newMultipledetrois( "" + i );
19                         }
20                         if(i % DIVISEUR_QUATRE == 0) {
21                                 multiplesDeQuatre.newMultipledequatre( "" + i );
22                         }
23                         if(i % DIVISEUR_CINQ == 0) {
24                                 multiplesDeCinq.newMultipledecinq( "" + i );
25                         }
26                 }
27                 System.out.println(mProgram.toString());
28         }
29         public static void main( String[] args ) {
30                 Main mObj = new Main();
31                 int min = 1;
32                 int max =  MAX;
33                 mObj.afficherListes( min, max );
34         }
35 }
```

Considérons la fonction `afficherListes` du listing 3.6. Dans cette fonction, l'instanciation de la classe `MProgram` (ligne 12) initialise le gabarit d'affichage. Les lignes 13, 14 et 15 créent les blocs d'affichage des listes des multiples de 3, 4 et 5 respectivement. Les lignes 18, 21 et 24 ajoutent chacune un élément dans le bloc d'affichage approprié en passant la valeur de l'élément (`i`) en argument. Précisons que les arguments de la macro sont de type `String`. Quand on a fini de mettre tous les éléments dans les blocs d'affichage appropriés de l'objet `mProgram`, on peut maintenant afficher le résultat avec une simple instruction `println` comme à la ligne 27 du listing 3.6. L'exécution du programme de ce listing avec `MIN=1` et `MAX=15` produit l'affichage du listing 3.7, ce qui correspond bien au résultat désiré.

3.3.5 Motivation du choix d'objectMacro

Comme nous venons de l'illustrer à l'aide d'un exemple, nous avons choisi ObjectMacro pour les raisons suivantes :

- Il est facile de séparer la présentation des résultats avec la programmation de l'application proprement dite. De cette manière, on peut modifier le formatage des résultats sans forcément toucher à l'application et vice-versa.
- Les règles d'écriture des macro sont simples.
- La compilation de la macro génère un paquetage Java facile à intégrer dans l'application.
- La structure orientée objet des classes générées permet de produire efficacement des textes imbriqués et/ou répétés.

3.4 Combinaison SableCC et ObjectMacro

La nécessité d'utiliser à la fois SableCC et ObjectMacro vient du fait que nous voulons examiner un code source et afficher le résultat selon un gabarit précis. Si on regarde le listing 3.4 par exemple, on voit qu'on a tous les résultats mais l'affichage est pêle-mêle. Ce qui peut être particulièrement pénible dans l'hypothèse de la correction de travaux. Il serait souhaitable de regrouper les informations des variables dans une liste

Listing 3.7 Affichage avec un gabarit objectMacro : `exempleAffichageOM.text`

```
Affichage selon gabarit
-----------------------

Liste des nombres multiples de 3
--------------------------------
3
6
9
12
15

Liste des nombres multiples de 4
--------------------------------
4
8
12

Liste des nombres multiples de 5
--------------------------------
5
10
15
```

et celles des méthodes dans une autre, comme dans le listing 3.7. Cela apporterait plus de clarté dans les résultats des analyses effectuées. C'est ce que nous avons fait dans notre outil de détection des défauts.

CHAPITRE IV

STRATÉGIES DE DÉTECTION DES DÉFAUTS

Dans ce chapitre, nous expliquons les stratégies utilisées pour détecter les défauts avec l'outil que nous avons développé. Compte tenu du nombre (plusieurs dizaines) élevé de défauts détectés, nous ne détaillerons pas ici la stratégie pour chaque défaut séparément. Nous présentons plutôt les principales stratégies et nous indiquerons quelques défauts détectés par ces stratégies.

4.1 Stratégie globale

L'idéal poursuivi dans la stratégie globale est de trouver un moyen de détecter toutes les occurrences du défaut cherché. Pour cela, nous choisissons de baser la détection des défauts sur la grammaire même du langage de programmation. En l'occurrence, Java, pour cette version de l'application.

Le lecteur se souviendra que les défauts cherchés sont ceux qui ne sont pas détectables par un compilateur ordinaire. C'est-à-dire qu'on cherche du code syntaxiquement correct mais qu'on ne veut pas voir apparaître pour une raison ou une autre.

Le fait de baser notre détection sur la grammaire du langage de programmation nous apporte les assurances suivantes :

- Le code cherché ne peut se trouver qu'aux endroits prévus par la grammaire et pas ailleurs. Autrement, le code ne compilerait pas.
- On parcourt tous les endroits où le code est susceptible de se trouver car ces

endroits sont précisés dans la grammaire.

A l'aide des outils (voir chap. 3) utilisés pour le développement, on parcourt le code source avec un visiteur (comme à la section 3.2.3) à la recherche du défaut, ce qui garantit de trouver toutes les occurrences du défaut recherché.

4.2 Association défaut-métrique

Pour faciliter la détection, une métrique est associée à chaque défaut. La détection revient donc à analyser le code source fourni comme argument à notre logiciel et de calculer la valeur de la métrique. Cette approche permet aussi d'obtenir le résultat de la détection sous forme d'une valeur numérique pouvant être utilisée, le cas échéant, pour la correction du programme analysé.

Dans certains cas, on pourra aussi demander d'avoir un résultat plus détaillé (numéro de ligne, etc.) sur la détection des défauts. Le lecteur trouvera plus d'informations sur le fonctionnement du programme dans le chapitre 5.

4.3 Défauts liés à des instructions spécifiques

Dans le cas particulier de la correction de travaux d'étudiants, il arrive que le professeur mette des restrictions à l'usage de certaines instructions. Dans ces conditions, l'utilisation d'une instruction interdite constitue un défaut.

Pour détecter un tel défaut, on se base sur la définition de l'instruction dans la grammaire. En parcourant le code source, dès que l'on rencontre une forme de cette instruction, on incrémente la valeur de la métrique associée (voir section 4.2). Au besoin, on retient aussi certains détails comme la ligne où se trouve l'instruction et toutes les informations permettant de localiser facilement ce défaut.

Cette stratégie est notamment utilisée pour détecter les opérateurs ou instructions suivants (liste non exhaustive) :

Opérateurs : `++`, `--`, `+=`, `-=`, `/=`, `*=`

Structures : `if`, `while`, `for`, `switch`, `do`

Déclarations : variables, constantes, méthodes

4.4 Défauts liés aux constantes

Dans les défauts liés aux constantes, il convient de distinguer ceux relatifs aux identifiants de ceux qui se rapportent aux valeurs proprement dites, comme les constantes numériques.

4.4.1 Identificateurs de constantes

Pour détecter les défauts liés aux identificateurs des constantes, nous utilisons une stratégie simple. Nous parcourons les déclarations des variables et nous regardons celles qui sont déclarées comme des constantes (`final` en Java). A chacun des identifiants de ces constantes, nous effectuons les vérifications supplémentaires et spécifiques au défaut que nous cherchons à détecter. Si ces conditions supplémentaires sont satisfaites, alors le défaut est détecté et la valeur de la métrique correspondante est incrémentée. Sinon, il n'y a pas défaut.

4.4.2 Détection du défaut de nom de constante

Pour détecter le défaut (voir 2.2.1) dû au fait que le nom de constante de classe ne soit pas entièrement en majuscule, il suffit d'utiliser une expression régulière pour voir si le nom de la constante :

- Commence par une lettre majuscule (`A`, ..., `Z`) ou un soulignement (`_`) ;
- Ne contient que des lettres majuscules, soulignement et des chiffres (0, ..., 9) ;
- Ne contient aucun autre caractère.

4.4.3 Valeurs constantes numériques

Parfois, il est exigé que les valeurs constantes respectent certaines conditions spécifiques. C'est le cas des valeurs numériques. Souvent, on exige que des valeurs numériques ne puissent apparaître qu'à certains endroits spécifiques.

Dans le cas des «constantes magiques» par exemple, on exige que tout nombre numérique autre que 0 et 1 soit assigné à une variable constante (`final`). C'est cette variable qui sera, par la suite, utilisée dans le code.

Pour détecter les défauts des constantes numériques, on parcourt le programme partout où elles ne sont pas sensées se retrouver. Dans le cas des nombres magiques, c'est ailleurs que dans une déclaration de constante. Si on trouve une valeur numérique, on vérifie si sa valeur est autorisée à cet endroit (0 ou 1). Sinon, le défaut est détecté et la métrique correspondante incrémentée.

4.5 Défauts liés aux variables

Dans cette section, nous présentons la stratégie de détection de quelques défauts spécifiques aux variables.

4.5.1 Initialisation inutile

Ici on présente un peu plus en détail la détection de défaut d'initialisation inutile (voir section 2.1.1). A première vue, cela semble simple à détecter. Toutefois, la situation se complique lorsqu'il faut tenir compte de la portée des variables et des exécutions possibles. Les stratégies de détection expliquées ci-après ne concernent que les variables locales déclarées dans une méthode.

Portée des variables

L'idée de la détection de ce défaut est d'examiner toute la portée de la variable et de s'assurer que quelle que soit l'exécution, le premier usage de la variable après initialisation soit une affectation. S'il y a au moins une exécution normale possible qui ne respecte pas cette exigence, alors on ne peut pas signaler le défaut. En effet, l'initialisation pourrait, en particulier, être utile pour cette dernière exécution.

Dans la pratique, on applique l'algorithme 4.1 au programme à analyser, lequel réfère ensuite aux algorithmes 4.2, 4.3 et AlgoInitInutileBlocSpecial.

Algorithme 4.1 Détection de l'initialisation inutile

précondition: *noeud* est le noeud initial de l'arbre syntaxique abstrait (AST) du programme à analyser

postcondition: Tout l'arbre AST est analysé

```
procedure INITINUTILEDETECT(NoeudRacine noeud)
    pour tout methodes définies faire
        noeudMeth ← <noeud bloc de la méthode>
        INITINUTILEDETECTVISITERBLOC(noeudMeth)
    fin pour
fin procedure
```

Exclusion des variables globales

Pour pouvoir respecter les exigences énoncées plus haut, il est important de pouvoir examiner toutes exécutions possibles et analyser la séquences des instructions.

En ce qui concerne les variables globales ou les attributs (propriétés), la situation pourrait devenir complexe voire même impossible à déterminer puisqu'il faudrait examiner toutes les exécutions possibles. En effet, les variables globales peuvent êtres utilisées dans n'importe quelle méthode. Il faut donc non seulement s'assurer de la séquence d'exécution des instructions dans chaque méthode, mais aussi examiner l'ordre

Algorithme 4.2 Détection de l'initialisation inutile dans bloc

précondition: *noeud* est un noeud bloc (méthode ou bloc conditionnel) de l'arbre syntaxique abstrait (AST) du programme à analyser. *varsInitialisees* est une variable globale.

postcondition: Tout le sous arbre de racine *noeud* est visité.

procedure INITINUTILEDETECTVISITERBLOC(NoeudBloc *noeud*)
  varsInitialisees ← {}
  tant que Tout le sous-arbre de *noeud* n'est pas visité **faire**
    Tester l'instruction suivante dans le bloc
    si Initialisation d'une variable v **alors**
      $varsInitialisees \leftarrow varsInitialisees \cup \{v\}$
    sinon si Usage d'une variable v (PAS une affectation) **alors**
      $varsInitialisees \leftarrow varsInitialisees - \{v\}$
    sinon si Instruction conditionnelle **alors**
      noeudInst ← <noeud de l'instruction conditionnelle>
      TRAITERINSTRUCTIONCONDITIONNELLE(*noeudInst*)
    sinon si Affectation à une variable v **alors**
      Signaler le défaut si $v \in varsInitialisees$
      $varsInitialisees \leftarrow varsInitialisees - \{v\}$
    fin si
  fin tant que
fin procedure

d'exécution des méthodes. L'impossibilité d'examiner toutes les exécutions vient du fait qu'en Java, les attributs et les méthodes peuvent aussi être accédés ou appelés en dehors de la classe.

Considérons l'exemple du listing 4.1. Si on ne regarde que ce qui se passe dans la classe `A`, on risque de conclure que l'initialisation de `variableGlobale` est inutile. En observant ce qu'il y a dans la classe `B`, on s'aperçoit vite que l'initialisation de `variableGlobale` n'est pas inutile dans `A`. En effet, on accède à la variable sans qu'il y ait exécution d'aucune méthode de `A`.

Comme on le voit, pour trancher les cas de variables globales, il faut non seulement regarder ce qui se passe dans la classe mais aussi les appels provenant de l'extérieur. Nous avons considéré que chercher à vérifier le respect d'une telle contrainte sortirait du cadre du présent travail.

En tenant compte des différents cas de figure semblables à ce qu'on vient de voir ci-dessus, nous choisissons donc de ne pas considérer les variables globales ou les attributs dans la détection de ce défaut. Nous ne traitons donc que les variables locales déclarées dans une méthode.

Traitement des blocs conditionnels

Pour détecter le défaut d'initialisation inutile en présence de blocs conditionnels, il convient de répartir les variables en deux groupes selon leur portée :

- Les variables déclarées dans le bloc et dont la portée se limite au bloc ;
- Les variables déclarées avant le bloc et dont la portée continue au delà du bloc.

Pour le premier groupe, la détection se limite à regarder ce qui se passe dans le bloc. Pour le deuxième groupe, il faut non seulement regarder ce qui se passe dans le bloc conditionnel mais aussi ce qui se passe après le bloc conditionnel. En effet, pour une exécution donnée, il se peut que le bloc conditionnel ne s'exécute pas. Le traitement des blocs conditionnels est formalisé dans les algorithmes 4.3 et 4.4.

Listing 4.1 Initialisation Inutile : problématique des variables globales

```
class A {
        public int variableGlobale = 1; //initialisation
        ...
        void methodeA() {
                variableGlobale = 0;
        }
        void methodeB() {
                variableGlobale = 0;
        }
}

class B {
        ...
        A a = new A();
        System.out.println("La variable globale = " + a.variableGlobale);
        ...
}
```

Algorithme 4.3 Initialisation Inutile : Traiter l'instruction conditionnelle

précondition: *noeud* est un noeud instruction conditionnelle de l'arbre syntaxique abstrait (AST) du programme à analyser.

postcondition: Les initialisations inutiles à l'intérieur du bloc sont détectées.

procedure TRAITERINSTRUCTIONCONDITIONNELLE(NoeudInstrCond *noeud*)
Visiter la condition de l'instruction courante
<Vérifier la nature de *noeud*> :
si Une instruction "IF" **alors**
noeudCond ← <le corps du bloc "if">
VISITERBLOCCONDITIONNEL(*noeudCond*)
tant que Il y a des "elsif" ou "else" **faire**
noeudCond ← <le corps du bloc "elseif" ou "else" suivant>
VISITERBLOCCONDITIONNEL(*noeudCond*)
fin tant que
sinon si Une instruction "while" **alors**
noeudCond ← <le corps de l'instruction "while">
VISITERBLOCCONDITIONNEL(*noeudCond*)
sinon si Une instruction "for" **alors**
noeudCond ← <le corps de l'instruction "for">
VISITERBLOCCONDITIONNEL(*noeudCond*)
sinon si Une instruction "switch" **alors**
tant que tous les "case" ne sont pas traités **faire**
noeudCond ← <le corps du "case" suivant>
VISITERBLOCCONDITIONNEL(*noeudCond*)
fin tant que
sinon si Une instruction "catch" **alors**
noeudCond ← <le corps du "catch">
VISITERBLOCCONDITIONNEL(*noeudCond*)
fin si
fin procedure

Algorithme 4.4 Initialisation inutile : Traiter le bloc conditionnel

précondition: *varsInitialisees* et *varsAffecteesDansBloc* sont des variables globales

postcondition: Les variables initialisées avant le bloc et dont la première utilisation n'est pas une affectation sont supprimées de *varsInitialisees* qui contient les candidats à l'initialisation inutile.

procedure VISITERBLOCCONDITIONNEL(NoeudBlocConditionnel noeud)

copieVI ← *varsInitialisees*

Enregistrer les autres accumulateurs éventuels

varsInitialisees ← {}

Ré-initialiser les autres accumulateurs éventuels

INITINUTILEDETECTVISITERBLOC(noeud)

varsInitialisees ← *copieVI*

Recopier les valeurs des autres accumulateurs éventuels enregistrés précédemment

copieVAB ← *varsAffecteesDansBloc*

varsAffecteesDansBloc ← {}

tant que Pas la fin du bloc conditionnel **faire**

Visiter l'instruction suivante

si Affectation d'une variable v **alors**

si $v \in$ *varsInitialisees* **alors**

varsAffecteesDansBloc ← *varsAffecteesDansBloc* $\cup \{v\}$

varsInitialisees ← *varsInitialisees* $- \{v\}$

fin si

fin si

si Usage d'une variable v **alors**

varsInitialisees ← *varsInitialisees* $- \{v\}$

fin si

fin tant que

varsInitialisees ← *varsInitialisees* $\cup$ *varsAffecteesDansBloc*

varsAffecteesDansBloc ← *copieVAB*

fin procedure

La fusion des listes effectuée à la fin de l'algorithme 4.4 permet de vérifier si les variables initialisées avant le bloc conditionnel et dont la première utilisation dans le bloc est une affectation auront encore une affectation comme première utilisation après le bloc conditionnel. Même dans le cas d'une instruction "If", on analyse tant la partie "Then" que les parties "Else" et "Else If" éventuelles. De cette manière on s'assure de couvrir toutes les exécutions possibles.

4.6 Défauts sur les structures de contrôle

Dans cette partie, nous abordons les défauts liés à certaines structures de contrôle. On expliquera en détails la stratégie de détection de quelques défauts. Nous donnerons aussi les algorithmes que nous proposons d'utiliser.

4.6.1 `While` convertible à `for`

Ici, la question est de savoir si la boucle `while` présente pourrait être remplacée par une boucle `for` équivalente. Le lecteur trouvera les détails sur ce défaut à la section 2.4.5.

Dans la plupart des cas, pour pouvoir remplacer une boucle `while` par une boucle `for` équivalente, il faut, au minimum, que les conditions suivantes soient réunies :

- Avoir au moins une variable dont la valeur est évaluée à chaque itération, dans la condition de la boucle.
- Cette variable est incrémentée ou décrémentée par une constante à chaque itération.
- La variable ne peut plus être utilisée dans le code qui suit statiquement l'incrémentation/décrémentation.

Choix de la variable d'itération

Dans les cas relativement simples, la variable d'itération candidate devrait apparaître dans la condition de la boucle `while`. À chaque itération, une valeur est assignée à

la variable d'itération une et une seule fois. Ces exigences tiennent du fait que la variable d'itération de `for` est modifiée une et une seule fois à chaque itération.

Incrément/décrément de la variable d'itération

À chaque itération, la variable d'itération doit être incrémentée ou décrémentée de manière constante. Cette contrainte vient du fait que, habituellement, la variable d'itération de la boucle `for` augmente ou diminue d'une valeur constante. En d'autres mots, on pourrait dire que la valeur de la variable d'itération évolue sous forme d'une progression arithmétique ($t_{n+1} = t_n + r$) de raison r, ce qui impose que r soit une constante.

Une affectation d'une variable candidate à devenir variable d'itération de `for` ne peut donc être que sous forme d'une progression arithmétique. Tout autre cas exclurait la variable à devenir variable d'itération. En particulier, il est exclu d'avoir une affectation dont une valeur proviendrait de l'appel d'une fonction. Cette exclusion s'explique par le fait que les valeurs retournées par une fonction peuvent varier à chaque appel. Dans cette dernière hypothèse, il serait plus difficile, pour ne pas dire impossible, de trouver une boucle `for` équivalente.

Utilisation de la variable d'itération après incrément/décrément

Une fois incrémentée ou décrémentée, la variable d'itération ne devrait plus apparaître dans le code, jusqu'à la fin du corps de la boucle. Toute nouvelle apparition exclurait la variable de la possibilité d'être la variable d'itération. Cette exigence permet de garantir que rien ne soit affecté par la nouvelle valeur de la variable d'itération avant la prochaine itération. En effet, dans une boucle `for`, la variable d'itération n'est incrémentée ou décrémentée qu'à la fin d'une itération.

4.6.2 Modification de la variable d'itération de `for`

Cette section donne la stratégie de détection du défaut de modification de la variable d'itération d'une boucle `for` (voir 2.4.1). La stratégie que nous proposons consiste

simplement en ce qui suit :

- Établir une liste des variables d'itération
- Parcourir toutes les affectations à l'intérieur de la boucle
- Signaler le défaut si on trouve une affectation ou une modification d'une variable d'itération.

En présence de ce défaut, il est souhaitable que le programmeur s'interroge pour savoir si l'utilisation d'une boucle `while` ne serait pas plus appropriée.

4.7 Défauts liés aux blocs

Dans cette section, nous abordons les défauts spécifiques au corps (blocs) des structures de contrôle.

4.7.1 `while`, `if`, `for`, ... vide

Il arrive parfois que, par mégarde, une instruction `while`, `if`, `for` ou une autre soit séparée de son corps par un point virgule. Dans ces conditions, on aboutit en réalité à une instruction avec un corps vide. Il arrive aussi que le corps d'une de ces instructions soit composé uniquement des accolades ouvrante et fermante (`{}`), sans aucune instruction à l'intérieur. Les espaces blancs éventuels n'étant pas considérés, ceci donne un résultat identique à un corps vide.

La détection de ce type de défaut se fait donc en vérifiant si le corps de l'instruction est vide ou est constitué uniquement de deux accolades ouvrante et fermante. A la demande, on signale à l'utilisateur ces cas pour qu'il s'assure que c'est bien une situation voulue et non accidentelle.

4.7.2 `while`, `if`, `for` sans accolades

Dans certains cas, le corps d'une instruction `while`, `if`, `for` ou autre peut être constitué d'une seule instruction. La bonne pratique suggère que dans une telle situation, on mette quand même cette instruction entre les accolades ouvrante et fermante pour

faciliter l'extension éventuelle du bloc dans le futur. Une exception est tout de même admise dans le cas de `else if`.

Pour détecter ce défaut, nous procédons de la manière suivante.

- Vérifier d'abord que le corps de l'instruction n'est pas vide (au sens de la section 4.7.1).
- Si cette première condition est rencontrée, vérifier ensuite que le corps commence par une accolade ouvrante ({) et se termine avec une accolade fermante (}).

Si la première condition est rencontrée et pas la deuxième, alors on signale le défaut. Remarquons que si le corps du bloc est vide, ce défaut ne sera pas signalé car cela correspond à un autre défaut traité à la section 4.7.1.

4.8 Défauts liés aux méthodes

Cette section présente la stratégie de détection de certains défauts spécifiques aux méthodes ou aux constructeurs.

4.8.1 Nombre de paramètres

Dans certains cas, on a besoin de limiter ou déterminer le nombre de paramètres dans une méthode ou un constructeur. On chercherait, par exemple, à savoir s'il y a des méthodes ou des constructeurs déclarés avec un nombre d'arguments supérieur, inférieur ou égal à certaines valeurs `min` et `max` données. Par défaut, `min`=0 et `max`=7. Cela correspond respectivement aux cas où au minimum il n'y a pas d'argument et au maximum suggéré dans la référence donnée en bibliographie (Tremblay, 2011).

La stratégie de détection de ces défauts consiste à compter les arguments déclarés puis à comparer le nombre obtenu avec la borne spécifiée. Le défaut sera signalé si la borne `max` est dépassée ou si la borne `min` n'est pas atteinte.

4.8.2 Nombre de méthodes ou des constructeurs déclarés

Dans le cas particulier de la correction, les enseignants aimeraient parfois vérifier si un certain nombre de constructeurs ou de méthodes ont été déclarés. Pour cette vérification, on se sert des règles de la grammaire et on comptabilise la présence de toutes les déclarations des méthodes ou des constructeurs. Dans la pratique, cela consiste à compter les noeuds de l'AST correspondant aux déclarations cherchées.

CHAPITRE V

MISE EN OEUVRE DE DDPJ

Dans ce chapitre, nous présentons *DDPJ*, le logiciel développé dans le cadre de notre projet de maîtrise, et ce du point de vue de sa conception. Comme pour toute application, nous devions lui trouver un nom. Le souhait était d'avoir un nom qui soit en lui-même porteur du concept. En accord avec le professeur Tremblay, nous avons retenu le nom suivant :

– **DDPJ : Détection des Défauts des Programmes Java.**

Le mot Java est utilisé ici parce que la première version est développée en Java pour des programmes écrits en Java.

L'approche utilisée pour la présentation de ce chapitre est d'aborder la structure et l'organisation interne du logiciel sans pour autant entrer dans les détails du code source. Avant cela, regardons d'abord quelques exigences principales à respecter.

5.1 Quelques exigences principales

Cette section présente les exigences principales que le logiciel développé devait respecter.

5.1.1 La post-compilation

Dans ce travail, nous avons voulu développer un logiciel capable de détecter les défauts de qualité que le compilateur ordinaire ne détecte pas. De par ce fait, l'exécution

de ce logiciel se situe donc au niveau post-compilation. Le logiciel pourra faire l'hypothèse que le code passé en argument est exempt de toute erreur syntaxique.

Le logiciel devra offrir la possibilité de localiser rapidement et facilement les défauts détectés. Pour cela, il devra fonctionner comme un compilateur capable de générer non pas un exécutable (code octet) mais un message texte avec des informations pertinentes à la demande de l'utilisateur.

Le message éventuel retourné en résultat dépendra des options et arguments passés par l'utilisateur. En effet, ceux-ci déterminent et expriment les besoins de l'utilisateur.

5.1.2 Exécution en mode ligne de commande

Un des souhaits exprimés pour ce logiciel est de pouvoir servir à la correction de travaux d'étudiants. Il devrait particulièrement aider à évaluer les travaux du point de vue qualité de code.

On aimerait aussi que le logiciel soit un outil d'aide à l'apprentissage de la programmation de qualité. L'utilisateur pourrait donc l'exécuter lui-même pour évaluer son programme et obtenir des indications, voire même des suggestions, sur la liste des défauts possibles.

Plus spécifiquement, nous prévoyons aussi l'intégration de ce logiciel dans *Oto* (Tremblay et al., 2008), un outil d'aide à la correction des travaux de programmation. Notre logiciel sera appelé directement à partir du script *Oto*.

Pour satisfaire à ces différentes exigences et s'accommoder plus facilement à l'automatisation des certaines tâches comme la correction, nous avons estimé que notre logiciel devrait être exécutable en ligne de commande, en utilisant les flux d'entrée/sortie standards.

5.1.3 Prise en charge de langues multiples

Comme nous l'avons dit à la section 4.2, on aimerait associer chaque défaut à une métrique. La métrique sera identifiée par un acronyme. Notre souhait est que l'acronyme ait un sens par rapport à la description ou au nom de la métrique. Cette exigence a pour but de faciliter l'utilisation du logiciel et minimiser le risque de passer en ligne de commande un sigle différent de la métrique que l'on veut calculer.

Nous avons voulu qu'il soit possible de donner à chaque métrique un nom et une description dans la langue de l'utilisateur en modifiant tout simplement un fichier de configuration. En effet, à terme, la liste de métriques pourrait devenir longue. Il faut donc donner aux utilisateurs qui le souhaitent la possibilité d'utiliser les sigles qui leur semblent les plus appropriés. Une telle option permettrait l'utilisation du logiciel quasiment partout à travers le monde.

A titre d'illustration, on aimerait qu'il soit possible d'adapter facilement et rapidement ce logiciel pour une utilisation dans un des contextes suivants :

- Utilisation d'une liste de métriques dans une autre langue (Anglais, Allemand, Néerlandais, Italien, Suédois, etc.).
- Utilisation des sigles des métriques différents de ce qui est proposé à la base. Ainsi, quelqu'un pourrait choisir de numéroter les métriques et les appeler MET1, MET2, MET3, ..., tout en gardant la description en français.
- Utilisation de sigles synonymes à un ou plusieurs sigles existants.
- Etc.

Ce logiciel ayant principalement des objectifs pédagogiques, il nous semble important que les acronymes et les messages soient facilement compréhensibles pour les utilisateurs.

5.1.4 Facilité d'évolution et d'extension

Comme tout autre logiciel, nous sommes persuadés que notre logiciel aura besoin d'évoluer pour s'accommoder aux nouveaux besoins et à l'évolution technologique. L'évo-

lution, c'est aussi des améliorations possibles des certaines choses qui sont implémentées dans la version actuelle.

Compte tenu du sujet traité dans ce logiciel, certaines évolutions et extensions sont d'ailleurs prévisibles. C'est notamment le cas de ce que nous illustrons ci-après.

Évolutions prévisibles

Parmi les évolutions prévisibles, il y a lieu de citer :

- Une évolution liée au langage de programmation proprement dit. En effet, si la grammaire du langage pour lequel on veut détecter les défauts évolue, il serait normal que l'implémentation du logiciel évolue aussi pour se conformer aux spécifications de la nouvelle grammaire. Remarquons ici que les compilateurs aussi évoluent constamment dans ce sens.
- Une évolution liée aux nouvelles exigences. Comme il est possible que les conventions et exigences de bonnes pratiques (Sun Microsystems, 1999) de programmation évoluent, il est souhaitable qu'il soit possible de faire évoluer facilement le logiciel en vue de refléter ces nouvelles pratiques.

Extensions prévisibles

Compte tenu du contexte particulier du développement de ce logiciel, plusieurs extensions sont possibles et même prévisibles. Parmi ces extensions, nous pouvons citer :

- L'implémentation d'autres défauts. En effet, dans le cadre de ce mémoire, nous avons développé la détection de quelques dizaines de défauts. Il en reste tout de même plusieurs non implémentés dans la liste que nous avons proposée au chapitre 2. Cette liste de défauts, étant elle même non exhaustive, est par conséquent extensible.
- La prise en charge d'autres langages de programmation (C,C++, ...).

5.2 Langage et grammaire

Nous avons développé notre application en Java. En effet, en prenant en compte la puissance de Java et des outils (voir chap. 3) que nous avons choisis, il nous a semblé naturel de développer dans ce langage.

Nous avons également choisi de commencer la détection des défauts dans le langage Java parce que c'est ce qui est utilisé pour les premiers cours de programmation à l'UQAM. C'est en particulier le cas du cours INF1120 Programmation I.

Pour cette première version, nous avons utilisé la grammaire de Java 1.5. Nous l'avons téléchargée à partir du site internet de SableCC (SableCC, 2000) puis nous avons adapté le nom du paquetage pour bien l'intégrer dans notre application.

5.3 Structure du logiciel

Dans cette section, nous expliquons la manière dont nous avons structuré notre logiciel. Nous y expliquons les avantages et les limites de certains choix que nous avons faits, ainsi que les configurations des fichiers systèmes du logiciel.

5.3.1 Structure générale et flux de données

La structure générale de *DDPJ* est illustrée dans le diagramme de flux de données de la figure 5.1. Les **analyseurs** de cette figure ont été générés par SableCC au cours de la compilation de la grammaire. Quant à l'**Afficheur de rapport**, nous l'avons obtenu en compilant le gabarit d'affichage avec ObjectMacro.

À la demande de l'utilisateur, les analyseurs transforment le code source à examiner en un arbre syntaxique abstrait (AST). Cet arbre est ensuite parcouru par le module de détection pour chercher les défauts spécifiés dans la demande de l'utilisateur. Les résultats et les rapports partiels sont progressivement accumulés dans l'**Afficheur de rapport** qui produira un rapport final de la détection à la fin de l'exécution du pro-

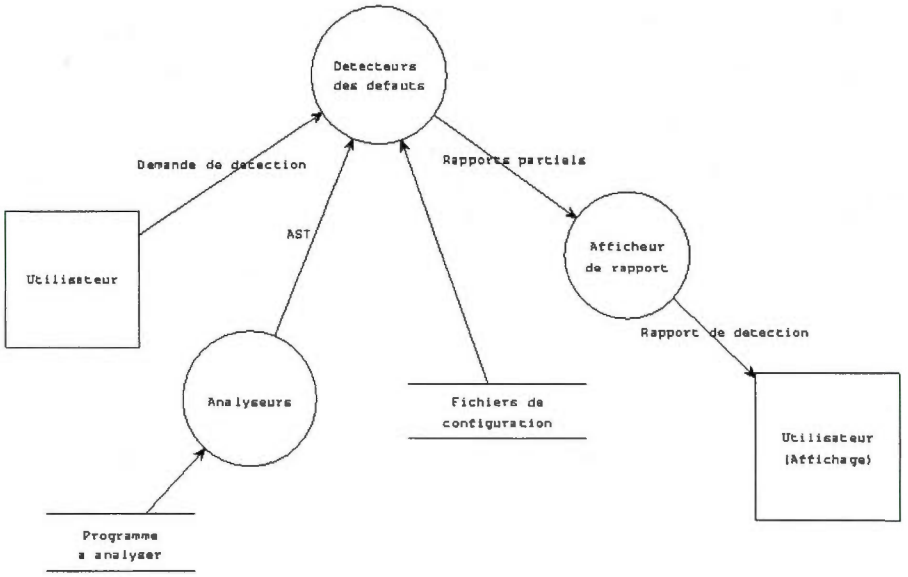

Figure 5.1 Diagramme de flux de données de DDPJ

gramme. Dans la suite de cette section, nous détaillerons un peu plus la structure de *DDPJ*. Les détails du fonctionnement sont abordés à la section 5.4.

5.3.2 Organisation du code source

Nous avons organisé le code source en projet eclipse Java nommé `DDPJ`. A l'intérieur du projet, les fichiers sont regroupés dans les paquetages et dossiers selon leurs particularités. Nous avons notamment :

- un paquetage des opérateurs de comparaisons
- un paquetage des classes implémentant la détection proprement dite des défauts
- un paquetage des tests unitaires
- un répertoire des fichiers de configuration
- un paquetage des classes effectuant certains traitements spécifiques et nécessaires à la détection des défauts
- quatre paquetages générés par SableCC (voir la section 3.2.2) suite à la compilation de la grammaire

- un paquetage généré par ObjectMacro (voir section 3.3.3) suite à la compilation du gabarit d'affichage (présentation) des textes

Les interactions entre les modules sont illustrées à la figure 5.2.

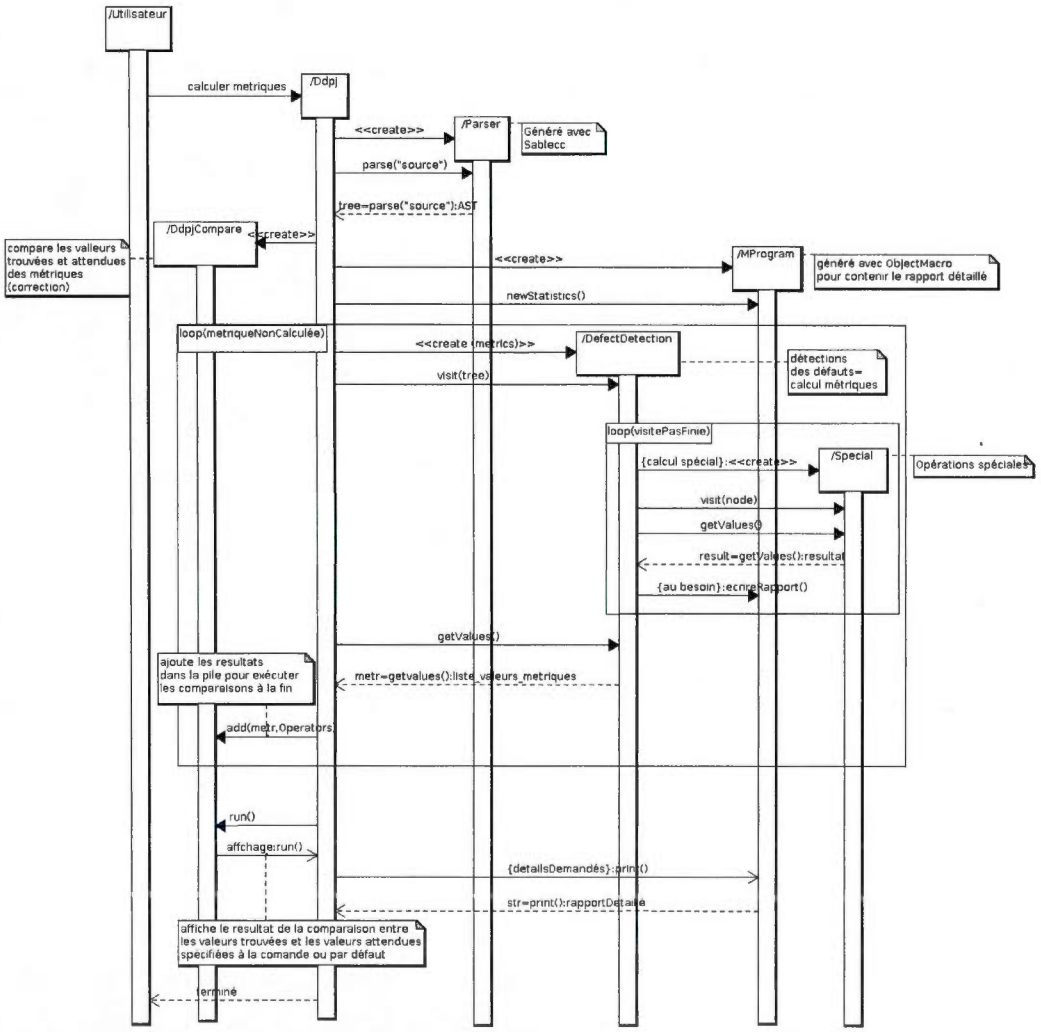

Figure 5.2 Diagramme de séquence du programme DDPJ

5.3.3 Associer un défaut à la classe qui la détecte

La structure de notre application est modulaire. De cette façon, des modifications éventuelles ne nécessitent de toucher qu'une portion du logiciel. Comme nous avons opté pour une approche associant chaque défaut à une métrique, nous avons des classes qui calculent ces métriques. Comme la liste des métriques et des classes est longue, il nous faut déterminer, au moment de l'exécution, les classes utiles pour les métriques demandées. De cette manière, on évite d'exécuter inutilement les autres classes. Il nous faut aussi pouvoir étendre facilement la liste des métriques ou changer le nom de la classe qui calcule une métrique sans spécialement toucher à d'autres parties du code source.

Pour résoudre ce problème, nous avons opté pour un fichier de configuration associant chaque métrique à la classe qui la calcule. Un exemple est présenté dans le listing 5.1.

Listing 5.1 Exemple de quelques entrées dans le fichier `ddpjConfig.properties`

```
1 # ---nom interne des metriques et classes les calculant
2 NDMMP=TooManyParametersInMethodDetection
3 NWI=WhileConvertibleToFor
4 NEWS=WhileConvertibleToFor
5 NWWB=WhileConvertibleToFor
6 NWCF=WhileConvertibleToFor
7 NVD=VariablesDeclarationDetection
```

Dans ce listing, chaque ligne est composée du nom de la métrique séparé du nom de la classe par le signe d'égalité. Ces combinaisons sont faites de la manière suivante :

- Le nom de la métrique repris à gauche est le nom système. C'est le nom utilisé en interne par l'application. En principe, ce nom n'est ni connu, ni vu par l'utilisateur. Il est vrai que dans certains cas, il peut y avoir coïncidence ou ressemblance avec le nom réellement manipulé par l'utilisateur. Pour calculer cette métrique, l'utilisateur manipulera le nom spécifié dans le fichier de configuration expliqué à la section 5.3.5. Cette approche permet d'adapter le nom

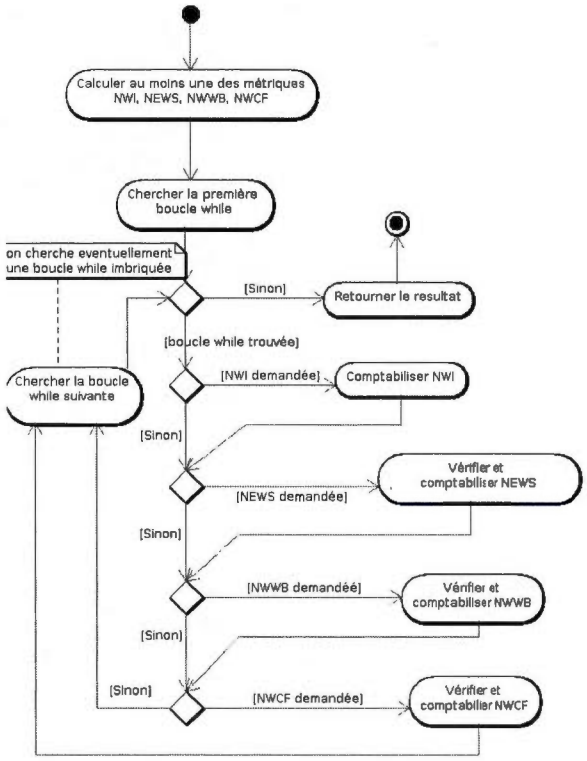

Figure 5.3 Diagramme d'activités calculant plusieurs métriques dans une même classe Java

de la métrique à l'utilisateur sans toucher à l'application.

- Le nom placé à droite du signe d'égalité est celui de la classe qui calcule la métrique. Le lecteur remarquera qu'une même classe peut calculer plusieurs métriques. C'est notamment le cas des lignes 3 à 6 du listing 5.1.

Dans le listing 5.1 par exemple, l'interprétation donnée à la ligne 6 est que la métrique de nom système `NWCF` est calculée par la classe `WhileConvertibleToFor`.

5.3.4 Avantages et limites des classes calculant plusieurs métriques

L'intérêt d'avoir la possibilité de calculer plusieurs métriques dans une même classe est qu'en cas de demande de détection de plusieurs défauts associés, on puisse instancier la classe une seule fois. Comme nous utilisons les visiteurs, cela permet de faire, potentiellement, plusieurs calculs au cours d'une même visite, ce qui améliore la vitesse d'exécution globale de l'application.

En lisant le paragraphe précédent, le lecteur pourra se demander si l'application ne serait pas alors plus rapide en calculant toutes les métriques dans une même classe. La réponse à cette question est malheureusement négative.

En effet, lorsqu'on exécute une classe calculant plusieurs métriques, on doit vérifier à plusieurs étapes s'il est utile de calculer et/ou conserver les valeurs associées à telle ou telle autre métrique. Ces vérifications peuvent aussi ralentir le système si le nombre de métriques devient important.

Une autre question qui pourrait venir à l'esprit du lecteur serait de savoir pourquoi ne pas alors calculer une et une seule métrique par classe. La réponse à cette question est simple. Avec les visiteurs, calculer une métrique par classe est bon tant qu'on calcule des métriques qui sont très différentes. Par contre, si on demande à déterminer les valeurs des métriques qui ont des nombreuses similitudes dans les opérations de calcul, on va perdre du temps et dégrader la performance par rapport à l'application elle-même. En effet, pour chaque métrique, il va falloir instancier à nouveau la classe et refaire plus ou moins le même parcours que le visiteur précédent. En regardant le diagramme d'activités de la

figure 5.3, on voit que les étapes de recherche des boucles `while` doivent être exécutées quelle que soit la métrique de ce groupe à calculer. Le fait de regrouper ces métriques dans une même classe permet de minimiser ces opérations communes.

Il faut donc non seulement limiter le nombre de métriques par classe mais aussi les regrouper éventuellement en fonction de la similarité des opérations de calcul. Dans le listing 5.1 par exemple, on remarque que les métriques des lignes 3 à 6 se rapportent à l'instruction `while`.

Si on demande de calculer le nombre d'instructions `while` convertibles à une instruction `for` (ligne 6) par exemple, il faut passer par toutes les instruction `while`. Donc, en les comptant au besoin, on trouve directement la valeur de la métrique de la ligne 3. Arrivé au niveau de chaque instruction `while`, on peut en profiter pour voir :

- si le corps est vide (ligne 4),
- s'il n'y a pas d'accolades entourant l'instruction du corps (ligne 5),
- ou si l'instruction peut être convertie en boucle `for`.

Comme on le voit dans cette illustration (fig. 5.3), ce regroupement est intéressant car, de toute façon, même si on ne calcule qu'une seule de ces métriques, le parcours commun sera fait. Par contre, si on veut en calculer plusieurs, on ne fera le parcours qu'une seule fois.

5.3.5 Configuration des langues

Il est facile de configurer *DDPJ* pour ajouter ou utiliser une autre langue. Pour cela, il n'est pas nécessaire de toucher à l'implémentation proprement dite. Il suffit d'éditer et adapter les fichiers de configuration ou la macro du gabarit d'affichage.

Description des métriques

Pour utiliser les noms et la description des métriques dans une autre langue, il suffit de créer un fichier de propriétés (*properties*) Java dont le nom est composé du préfixe

`metrics-` auquel on ajoute les lettres de la langue. Ainsi, pour la langue française (`fr`), on aura le nom `metrics-fr.properties`.

Ce fichier doit être placé dans le sous-répertoire `ddpj/etc` du projet. Pour chaque métrique, il y aura deux entrées dans ce fichier comme le montre le listing 5.2 :

- La première entrée associe le nom de la métrique dans la langue à configurer à celui utilisé par le système en interne comme expliqué dans la section 5.3.3. C'est ce nom (à gauche) qui est manipulé par l'utilisateur.
- La deuxième entrée est une clé composée du nom de la métrique dans la langue à configurer et du suffixe `.DESC` (pour description) à laquelle on associe une description. C'est cette description qui sera affichée à l'utilisateur lorsqu'il voudra voir la liste des métriques implémentées.

Listing 5.2 Exemple d'une entrée dans le fichier `metrics-fr.properties`

```
1 NIWV=NEWS
2 NIWV.DESC=Nombre d'instructions while vides
```

Texte des résultats détaillés

Pour modifier le texte d'interface ou de détails des résultats, il faut éditer la macro se trouvant dans le répertoire `macros` puis la recompiler (voir 3.3.3).

Listing 5.3 Exemple d'une macro dans le fichier `template.objectmacro`

```
1 $macro: numberofemptywhilestatement(nbwst)$
2 Nombre de boucle "while" vides: $(nbwst)
3 $end: numberofemptywhilestatement$
```

Si nous considérons l'exemple du listing 5.3 ci-dessus, on peut modifier la ligne 2 sans toucher au texte entre parenthèses précédé d'un signe de $. Ce texte sera remplacé par la valeur de l'argument défini à la ligne 1 comme nous l'avons expliqué à la section 3.3.

Les lignes 1 et 3 correspondent respectivement à la déclaration du début et de fin de la macro. Nous suggérons de ne pas modifier ces lignes particulières car cela affecte

directement l'application. En effet, la déclaration de la macro à la ligne 1 correspond au nom de la classe et des arguments éventuels qui seront générés à la compilation. Cette classe est réellement utilisée dans l'application pour produire le texte de la ligne 2 à l'endroit voulu. Si on modifie le nom de la classe ou le nombre d'arguments, il faudra apporter des adaptations appropriées dans le code source de l'application. Un tel cas de figure est justement ce qu'on veut éviter dans le principe de la séparation de l'application et de la présentation.

5.4 Étapes principales d'une exécution

Dans cette section, nous allons expliquer les principes généraux du fonctionnement de notre logiciel. Il ne s'agit pas ici de donner le manuel d'utilisation que le lecteur trouvera à l'annexe A. Ici, nous montrons comment le logiciel extrait les informations utiles à partir de la ligne de commande, détermine les modules à exécuter puis retourne le résultat. Le lecteur trouvera une illustration de l'algorithme général à la figure 5.4.

Lorsque l'utilisateur lance la commande, l'application extrait la liste des métriques à calculer et les options éventuelles. On cherche par la suite à établir une correspondance entre les noms des métriques passées en arguments et les noms systèmes. Pour cela, il faut d'abord déterminer la langue que l'utilisateur veut utiliser. Si l'option langue n'est pas présente ou a une valeur non connue par le système, la langue par défaut est considérée. Dans la version actuelle, la langue par défaut est le français.

Une fois que la langue est déterminée, on va consulter le fichier de configuration approprié, comme expliqué à la section 5.3.5, pour déterminer les noms systèmes des métriques. Avec la liste des noms systèmes, on va consulter le fichier de configuration `ddpjConfig.properties`, tel qu'expliqué à la section 5.3.3, pour déterminer la liste des classes Java à exécuter.

Au moment de l'exécution, chaque classe reçoit une liste des métriques à calculer accompagnées des options éventuelles. Parmi les métriques que la classe peut calculer, elle ne calcule réellement au cours de l'exécution que celles qui lui sont demandées.

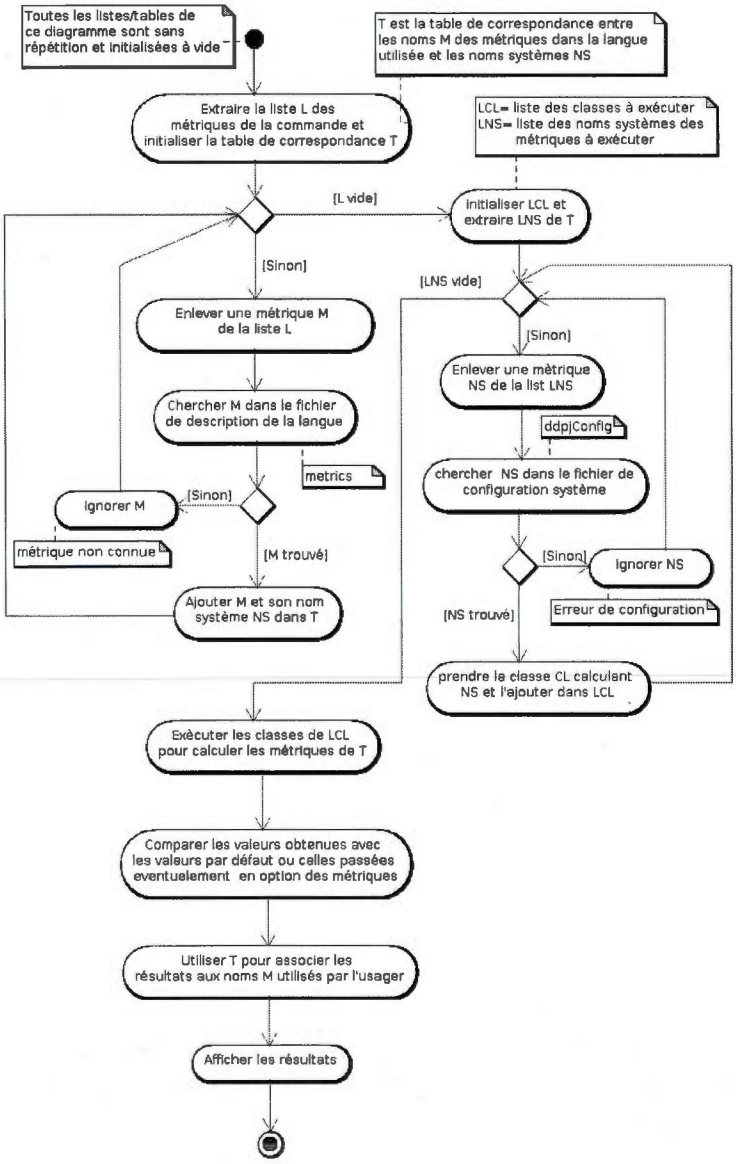

Figure 5.4 Diagramme d'activités principal pour le calcul des métriques

Le rapport est initialisé par le programme principal avant l'exécution des classes. Pendant le calcul des métriques, on remplit le rapport progressivement avec les résultats partiels en faisant appel aux classes appropriées des macros `Objectmacro` (ObjectMacro, 2011).

Une fois l'exécution terminée, la classe principale affiche les résultats et le rapport éventuel à l'utilisateur en utilisant les noms des métriques qu'il a utilisés. Les noms systèmes ne sont pas affichés. L'affichage du rapport dépendra aussi du fait que l'utilisateur a demandé un rapport détaillé ou non.

5.5 Module de correction

Un des buts poursuivis dans ce travail est de développer un outil pour la correction du point de vue qualité du code source. Pour cela, au delà de la détection des défauts, le logiciel doit aussi répondre à certaines questions précises. Ces questions sont formalisées sous forme de comparaison entre les valeurs trouvées et les valeurs attendues par le correcteur (enseignant).

Pour la plupart des métriques, l'utilisateur peut préciser à la ligne de commande si la valeur attendue doit être inférieure, égale ou supérieure à une valeur numérique précisée (un exemple d'utilisation se trouve à l'annexe A.1.3). Lorsque le logiciel calcule la métrique pour le code source, il compare ensuite la valeur obtenue pour cette métrique avec celle attendue en fonction de l'opérateur spécifié. Si la condition est respectée, rien n'est signalé. Sinon, on retourne un message d'erreur indiquant, entre autre, la valeur trouvée pour la métrique. À titre d'exemple, si l'utilisateur (correcteur) exige qu'on ne peut pas tolérer plus de trois nombres magiques, le logiciel va calculer cette métrique. Si le résultat est supérieur à trois alors un message d'erreur est renvoyé pour signaler que le code source ne respecte pas les conditions ; sinon aucun message n'est retourné. Dans ce cas précis, si on ne trouve que deux nombres magiques par exemple, on ne signalera rien.

Précisons que ce module de correction n'affiche pas en soit les détails de la détec-

tion. Il appartient à l'utilisateur de demander le détail en précisant l'option appropriée à la ligne de commande. Le lecteur trouvera les options disponibles à l'annexe A.

Pour pouvoir vérifier si les valeurs trouvées pour les métriques sont conformes aux valeurs attendues, nous avons conçu les fonctions appropriées et implémentant les opérateurs de comparaison des nombres : $<, \leq, =, \geq, >$. À chacun des opérateurs correspond une fonction qui reçoit en argument le nom de la métrique et les deux valeurs à comparer, puis qui affiche un message adéquat en cas de non respect de la condition.

5.6 Tests unitaires et tests de validation

Pour valider les résultats des calculs, nous avons développé un module de tests unitaires. Ce module nous a permis de valider particulièrement les valeurs numériques des métriques.

Pour valider les rapports produits en résultats, nous avons développé des tests de niveau système mettant en œuvre des gabarits des textes attendus et non attendus. Nous en profitons ici pour souligner la collaboration du professeur Guy Tremblay dans le développement et la validation de ce type de tests spécifiques.

Le principe de ces tests de niveau système est simple. Pour chaque cas, on a un sous répertoire contenant les fichiers suivants.

- Un fichier (exemple) de code source (Java) à tester.
- Un script d'exécution du programme `DDPJ` avec les arguments et les options nécessaires pour tester le code source.
- Un fichier texte contenant le texte attendu. L'ordre d'apparition des éléments est important dans ce fichier. En effet, les lignes de ce fichier doivent non seulement apparaître dans le résultat mais aussi doivent être rencontrées dans l'ordre spécifié. Sinon, il y a erreur.
- Un autre fichier texte contenant du texte non attendu. Ici, l'ordre n'a pas d'importance. La seule exigence est que les lignes de ce texte *ne doivent pas apparaître dans le résultat.*

Pour faire un test, on exécute le script Ruby qui va exécuter le programme *DDPJ* avec le code source du cas de test puis comparer les textes attendus et non attendus avec le résultat produit. Notons qu'en pratique, le fichier du texte attendu ne contient pas tout le texte attendu en résultat, mais uniquement les éléments clés qui doivent absolument apparaître. C'est le cas par exemple des indications sur les numéros des lignes où on détecte les défauts qui ne sont pas spécifiés dans les résultats attendus — parce que trop dépendants de la mise en page. Dans le fichier de texte non attendu, on va par exemple mettre les indications erronées des numéros de lignes ou des indications de résultats susceptibles d'apparaître en cas d'erreur dans le résultat.

5.7 Compilation, distribution et installation de DDPJ

Cette section explique comment compiler le code source de *DDPJ* pour obtenir un exécutable. On y trouve aussi des explications sur la façon de distribuer et d'installer cet exécutable.

5.7.1 Comment compiler DDPJ

Pour distribuer le logiciel *DDPJ*, il faut d'abord le compiler. Pour cela, nous avons écrit un fichier Ant `ddpj.xml` se trouvant dans le répertoire racine du projet eclipse. Ce fichier Ant permet d'automatiser la compilation en exécutant toutes les opérations nécessaires dans l'ordre approprié pour produire un exécutable `ddpj.jar`. À titre de rappel, la compilation d'un fichier Ant se fait au moyen de la commande `ant`. Dans le cas de *DDPJ*, la commande à exécuter est la suivante [1] :

```
ant -f ddpj.xml
```

La configuration actuelle de `ddpj.xml` permet notamment d'effectuer les opérations suivantes au cours de la compilation :

– Nettoyer les répertoires en supprimant les exécutables et les paquetages générés

1. Le programmeur se souviendra que, par défaut, si on ne précise pas le nom du fichier avec l'option `-f`, Ant cherchera le fichier `build.xml`.

par SableCC et ObjectMacro ;

- Recompiler la grammaire avec SableCC (génère notamment le paquetage `parser`) ;
- Recompiler le gabarit d'affichage (template) avec ObjectMacro (génère le package `macros`) ;
- Recompiler les sources `ddpj` ;
- Créer un exécutable `ddpj.jar` ;
- Inclure les bibliothèques nécessaires dans l'exécutable.

5.7.2 Distribution et installation de DDPJ

La distribution de *DDPJ* se fait simplement en transmettant le fichier `ddpj.jar`. En ce qui concerne l'installation, il suffit de placer le fichier `ddpj.jar` dans le répertoire approprié où il doit être accessible en exécution. Comme c'est une application Java, il devrait normalement s'exécuter sur la plupart des plateformes sans problème. L'exigence majeure est bien sûr d'avoir Java installé sur la machine. En ce qui concerne le développement et les tests, nous l'avons fait avec le Java de Sun dont la version minimum était la version 1.6.

CHAPITRE VI

EXEMPLES D'UTILISATION DE DDPJ POUR LA CORRECTION DES TRAVAUX EN GROUPE

Dans ce chapitre, nous allons voir deux exemples d'utilisation de *DDPJ* pour évaluer la qualité de programmes. Nous mettrons l'accent sur la correction de travaux d'étudiants en groupe. Pour cela, nous allons nous servir d'*Oto* et appeler *DDPJ* comme une application externe. Le lecteur qui s'intéresse à l'utilisation de *DDPJ* sur un seul fichier, de manière isolée, trouvera quelques exemples supplémentaires à l'annexe B.

6.1 Correction de travaux en groupe et rapport aux étudiants

Dans cette section, nous montrons l'utilisation de *DDPJ* dans la correction de travaux en groupe. Nous mettons l'accent sur le rapport détaillé qui doit être généré et envoyé aux étudiants. Pour simplifier les explications, nous allons nous servir d'un exemple de problème type. Pour ce problème, nous allons considérer un exemple de solution donnée puis nous utiliserons *Oto* pour corriger les travaux en groupe.

Supposons que le problème consiste à déterminer si une année est bissextile ou non. Pour rappel, une année est bissextile si elle est divisible par 400 ou divisible par 4 et non divisible par 100. Faisons l'hypothèse que l'énoncé du problème précise qu'on ne doit utiliser ni une boucle `for`, ni une boucle `while` pour résoudre ce problème.

Considérons un groupe de travaux sur ce problème et cherchons à faire une correction automatique en groupe avec *Oto*. N'étant pas encore intégré à *Oto*, *DDPJ* doit

être appelé comme une application externe utilisant les flux standards d'entrées/sorties.

Commençons par écrire le script *Oto* du listing 6.1 puis activer une évaluation avec la commande suivante :

```
oto activer_eval bissextiles script-anneesbissextiles.oto  \
                                                ExempleAnneesBissextiles.java
```

Comme l'indique la commande, l'évaluation s'appelle `bissextiles` et utilise le script *Oto* `script-anneesbissextiles.oto` du listing 6.1. Dans ce script, la ligne 3 fait appel à *DDPJ* en précisant les métriques à calculer. Ici, le correcteur a aussi voulu vérifier l'usage des constantes magiques (NCMG) [1]. Dans cette évaluation, nous ne nous intéressons qu'à la qualité du code source. Focalisons notre attention sur le travail du listing 6.2 et lançons la correction automatique en exécutant la commande suivante :

```
oto script-anneesbissextiles.oto *.tp_oto ddpj.jar
```

Oto va appeler *DDPJ* pour vérifier la qualité. Par la suite, *Oto* rassemble les rapports de tous les différents travaux individuels dans un seul rapport global dont nous reprenons un extrait au listing 6.3. Cet extrait correspond au travail du listing 6.2. En observant ce rapport, on voit qu'aucune boucle `for` ou `while` n'a été utilisée, mais il y a trois constantes magiques. C'est ce rapport qui sera envoyé à l'étudiant car il contient suffisamment d'information pour l'identification des défauts.

Listing 6.1 Exemple de script *Oto* utilisant *DDPJ* : `script_anneesbissextiles.oto`

```
1 groupe.each( :tri => :equipe, :fichiers_auxiliaires => :copier )  do |tp|
2     #verification de la qualite du code avec ddpj
3    tp['Verification de la qualite du code source avec ddpj']=`java -jar ddpj.jar -d ↙
                          ↳ ExempleAnneesBissextiles.java NIFO NIW NCMG`
5 end
7 puts  produire_rapport_complet( groupe )
```

1. La liste des métriques est reprise à l'annexe A.3

Listing 6.2 Exemple d'un travail à corriger en groupe avec *DDPJ*

```
1  import java.io.*;
2  public class ExempleAnneesBissextiles {
3          /* determine si une annee est bissextile:
4           * Divisible par 4 et non par 100 ou divisible par 400
5           */
6          public static void main(String[] args) {
7                  if( args.length <= 0 ) {
8                          System.out.println("Precisez annee svp!");
9                          System.exit( -1 );
10                 }
11                 try {
12                         int annee=Integer.parseInt(args[0]);
13                         if(( annee % 400 == 0 ) || (( annee % 4 == 0 ) && ( annee%100 ↙
                             ↳ != 0 ))) {
14                                 System.out.println("OUI");
15                         } else {
16                                 System.out.println("NON");
17                         }
18                 }catch( Exception e ) {
19                         System.out.println( "Erreur. Entrer un nombre entier svp! " + ↙
                             ↳ e.toString());
20                 }
21         }
22 }
```

Listing 6.3 Extrait de rapport détaillé de correction *Oto* et *DDPJ*

```
**************************************************************************
                Rapport complet de correction Oto
Nom de l'evaluation: script-anneesbissextiles.oto
En date du:  2013-07-24 a 15:29
**************************************************************************
**************************************************************************
TRAVAIL:    tremblay+2013.06.14.20.38.21.917949+DURN27018400.tp_oto
Equipe:     DURN27018400
Depot:      2013-06-14 a 20:38
Deposeur:   tremblay
Nom:        ag391244
Courriel:   ???
RESULTATS:
  Verification de la qualite du code source avec ddpj:
      NIW = 0
      NCMG = 3
      NIFO = 0
      Rapport detaille
      ----------------
      Nombre de constantes magiques: 3
      Nombre d'instruction "for": 0
      Nombre d'instructions "while" : 0
      Resultats de la detection
      -------------------------
      Detection des constantes magiques
      ---------------------------------
      Constante magique: 400
      Ligne numero: 13
      Constante magique: 4
      Ligne numero: 13
      Constante magique: 100
      Ligne numero: 13
      Detection des instructions "for"
      ------------------------------------------------
      Detection d'instructions "while"
      --------------------------------
```

6.2 Correction du TP1 du cours INF1120-41

Dans cette section, nous procédons à la correction des programmes Java remis par les étudiants du cours de Programmation I, groupe 41 (INF1120-41), dans le cadre du premier travail pratique (TP) de la session de l'automne 2011. Par la suite, nous analysons brièvement les défauts détectés.

6.2.1 La correction du TP

Dans cette sous-section, nous montrons les différentes étapes de la correction du TP 1 et commentons par la suite le rapport généré et destiné à l'enseignant. Comme nous traitons les travaux rendus réellement par les étudiants, nous ne montrerons ici que des données anonymes, donc ne permettant pas d'identifier les étudiants. Pour cela, nous supprimerons toutes les lettres et le dernier chiffre composant le code permanent et le nom d'utilisateur (code MS) dans l'extrait du rapport repris dans le listing 6.5 et dans la table 6.1.

Pour corriger ces travaux en groupe, nous allons nous servir d'*Oto*. Pour cela, nous commençons par écrire le script du listing 6.4. Dans ce script, nous demandons d'analyser la qualité de chaque programme avec *DDPJ*. Nous voulons particulièrement vérifier les éléments suivants :

- Les constantes magiques : métrique `NCMG`
- Les variables non initialisées à la déclaration : métrique `NVNID`
- Les noms des constantes de classe pas entièrement en majuscules : métrique `NCNM`.

Le lecteur remarquera qu'ici, on ne souhaite pas que *DDPJ* génère un rapport détaillé. En effet, on n'a pas précisé l'option **-d**. Cela s'explique par le fait que nous voulons avoir un rapport pertinent pour le correcteur. On veut juste savoir le nombre de défauts pour attribuer une note appropriée.

Comme les étudiants ont remis [2] les TP avec *Oto* et que nous avons déjà récupéré les travaux et les avons placés dans le répertoire `TP1Gr41`, nous allons maintenant activer l'évaluation `tp1gr41` avec la commande suivante :

```
oto activer_eval tp1gr41 script-tp1gr41.oto Tp1Gr41.java
```

Par la suite, nous corrigons ce groupe de travaux au moyen de la commande suivante :

```
oto script-tp1gr41.oto TP1Gr41/*.tp_oto ddpj.jar
```

Dans cette commande, on passe `ddpj.jar` en argument parce que *DDPJ* n'est pas encore intégré [3] à *Oto* et, par conséquent, l'exécutable doit être copié dans le répertoire de correction. Ceci se fait automatiquement par *Oto* grâce à la directive spécifiée à la première ligne du script du listing 6.4.

Nous obtenons alors un rapport dont un extrait est repris dans le listing 6.5. Ce rapport est destiné à l'enseignant et ne contient donc pas les détails des emplacements des défauts dans le code source.

Listing 6.4 Script de correction du TP1 INF1120-41 Automne 2011 : `script-tp1gr41.oto`

```
groupe.each( :tri => :equipe, :fichiers_auxiliaires => :copier )  do |tp|
    # verification de la qualite du code avec ddpj
    # Correction du TP1 gr41 du cours INF1120, Automne 2011.
   tp['Verification de la qualite du code source avec ddpj']='java -jar ddpj.jar ↙
                        ↳ Tp1Gr41.java NCMG NVNID NCNM'

end

puts  produire_rapport_complet( groupe )
```

Dans le listing 6.5, nous illustrons le rapport concernant deux travaux :

---

2. Nous référons le lecteur à la documentation d'*Oto* (Oto, 2005) pour les détails sur la remise des travaux.

3. Ni même installé sur le serveur `zeta.labunix.uqam.ca` sur lequel nous avons corrigé.

Listing 6.5 Extrait de rapport de correction en groupe du TP1 INF1120-41 avec *Oto* et *DDPJ*

```
1  ************************************************************************
2                      Rapport complet de correction Oto
4  Nom de l'evaluation: script-tp1gr41.oto
5  En date du:  2013-07-26 a 10:43
9  ************************************************************************
11 ************************************************************************
13 TRAVAIL:    --29112-+2011.10.20.17.37.36.406277+----2660790-.tp_oto
15 Equipe:     ----2660790-
16 Depot:      2011-10-20 a 17:37
17 Deposeur:   --29112-
18 Nom:        etuds_gt
19 Courriel:   ???
22 RESULTATS:
24   Verification de la qualite du code source avec ddpj:
25       NCNM = 2
26       NCMG = 224
27       NVNID = 11
30 ************************************************************************
32 TRAVAIL:    --59124-+2011.10.21.19.58.25.196191+----0504730-.tp_oto
34 Equipe:     ----0504730-
35 Depot:      2011-10-21 a 19:58
36 Deposeur:   --59124-
37 Nom:        etuds_gt
38 Courriel:   ???
41 RESULTATS:
43   Verification de la qualite du code source avec ddpj:
44       NCNM = 2
45       NCMG = 5
46       NVNID = 0
```

- Le premier est un des pires travaux du point de vue des critères de qualité spécifiés. Il y a 224 constantes magiques, 11 variables non initialisées à la déclaration et 2 noms de constantes de classe pas entièrement en majuscules.
- Le deuxième travail est un des meilleurs. Il n'y a que 5 constantes magiques, aucune variable non initialisée à la déclaration et seulement 2 noms de constantes pas entièrement en majuscules.

La table 6.1 montre les résultats complets des défauts détectés dans les divers travaux. Avec le rapport généré, l'enseignant peut facilement et rapidement attribuer des notes appropriées et équitables sur le critère de qualité du code. Pour cela, il suffit d'associer une certaine pénalité au nombre de défauts détectés.

6.2.2 Analyse des défauts détectés sur l'ensemble des travaux

Dans cette sous section, nous analysons globalement les nombres des défauts détectés à la correction du TP 1 du cours INF1120-41 donné à l'automne 2011. Au total, on a corrigé 40 travaux. A l'issue de la correction, on a obtenu les résultats présentés dans la table 6.1. En observant ces résultats, on constate ce qui suit :

- Il y a de nombreuses constantes magiques (`NCMG`). Tous les travaux ont plusieurs instances de ce défaut. Ceci pourrait être une indication pour l'enseignant d'insister ou de revenir sur ce point précis pour permettre aux étudiants de mieux l'assimiler.
- Le nombre de variables non initialisées à la déclaration (`NVNID`) est limité . Il y a même un travail qui n'a aucune instance de ce défaut. Comme quasiment tous les étudiants ont commis ce défaut, ce pourrait aussi être pertinent que l'enseignant en dise un mot aux étudiants mais, sans nécessairement s'y attarder.
- La plupart des étudiants ont bien compris les règles de noms des constantes des classes (`NCNM`). Il n'y a que quelques travaux qui contiennent des instances de ce défaut. L'enseignant peut donc cibler les quelques étudiants qui ont commis un nombre important de ce défaut pour s'assurer qu'ils ont bien compris.

Tableau 6.1 Défauts détectés dans les travaux du TP 1 de INF1120-41 de l'automne 2011

Equipe	NCNM	NCMG	NVNID
----1106870-	0	28	1
----1912830-	0	32	13
----0902890-	0	42	7
----1204900-	0	46	28
----3003830-	0	54	12
----2405840-	4	25	6
----1351710-	2	24	10
----2004770-	0	71	23
----2660790-	2	224	11
----2405740-	0	33	15
----0901740-	0	64	13
----0861640-	0	24	7
----0504730-	2	5	0
----0355890-	0	17	13
----1709900-	0	33	10
----1601850-	0	42	13
----2006870-	2	40	14
----1803850-	0	41	13
----1006710-	0	14	2
----0105920-	0	17	10
----0706850-	24	32	3
----0102690-	2	80	28
----1602770-	0	46	7
----0508900-	0	28	31
----0506900-	0	14	17
----2901900-	2	27	9
----0553840-	0	31	24
----0856820-	0	76	9
----1153820-	12	27	26
----0212880-	0	40	6
----2309810-	0	30	9
----2501870-	0	83	27
----2459920-	0	37	9
----1504870-	0	80	7
----1062810-	0	38	13
----0502820-	0	19	9
----0112820-	2	46	19
----1202870-	0	20	5
----1502730-	0	43	16
----1210860-	0	32	26

Comme nous venons de le voir dans la correction de ce TP, au delà de la correction proprement dite, *DDPJ* combiné avec *Oto* pour la correction des travaux en groupe pourrait aider l'enseignant à voir plus facilement les aspects de qualité qui sont bien compris par les étudiants et ceux qui nécessitent une explication plus ciblée.

CONCLUSION

La correction automatique préoccupe plusieurs enseignants confrontés à un nombre important d'étudiants. On aimerait non seulement alléger la tâche du correcteur, mais aussi rendre la correction la plus équitable possible et pouvoir donner des nouvelles rapidement aux étudiants pour qu'ils en tiennent compte dans les travaux suivants. Si la correction automatique se prête relativement bien pour les questions avec une réponse précise, ce n'est pas le cas pour des questions où le texte de la réponse n'est pas connu d'avance mais nécessite d'être analysé. La manière d'analyser le texte de la réponse dépend du domaine de la matière traitée. Dans ce mémoire, nous nous sommes intéressés au code source Java. L'enjeu était de développer un outil automatique pour évaluer la qualité des codes sources des programmes des étudiants.

La finalité de l'évaluation du code source abordée dans ce mémoire était de corriger le travail d'un étudiant sur la base de certains critères d'appréciation précis. Un des défis était également de définir ces critères d'appréciation et d'identifier puis comptabiliser les éléments associés sur le code source. On voulait que tout cela soit intégré dans un processus complet de correction automatique. En effet, la correction d'un programme informatique nécessite d'évaluer les aspects fonctionnels et la qualité du code. Du point de vue fonctionnel, on vérifie que le programme donne des résultats attendus. Cet aspect n'a pas fait l'objet de ce mémoire car il y a déjà plusieurs outils qui s'en occupent. Nous ne nous sommes pas non plus préoccupés des erreurs détectables par le compilateur car les travaux à corriger automatiquement doivent au minimum compiler.

Une façon d'évaluer la qualité du code consiste à vérifier, au delà du fonctionnement, que les standards et les styles de programmation généralement admis comme des bonnes pratiques sont respectés. C'est notamment le cas de la convention *Sun*.

Il s'est posé la question de savoir s'il n'y avait pas déjà une application informatique qui permettait de corriger complètement des travaux de programmation Java de niveau débutant. Au chapitre 1, nous avons passé en revue un certain nombre d'outils existants. Nous avons vu que la plupart de ces outils se limitaient essentiellement à la correction automatique du fonctionnement du programme. Ceux qui abordaient la qualité du code se focalisaient sur des aspects trop avancés et pas vraiment applicables sur les programmes des débutants ou ne permettaient pas de déterminer facilement les styles à vérifier dans le code source. Les applications les mieux abouties pour l'évaluation de la qualité du code ne donnaient pas de rapport chiffré que l'on pourrait intégrer facilement dans un processus de correction automatique. Ils nécessitaient d'analyser le rapport fourni pour déduire les défauts détectés et le nombre d'instances de chacun.

Comme pour la plupart d'autres applications de correction automatique, la version actuelle d'*Oto*, un outil développé au département informatique de l'UQAM, ne permet pas d'analyser le qualité de code. Il y avait donc un besoin de le doter de cette fonctionnalité. Heureusement qu'*Oto* a été conçu avec une architecture modulaire qui permet l'intégration d'autres applications. N'ayant pas trouvé un autre outil qu'on pouvait intégrer facilement à *Oto* pour automatiser la correction de la qualité du code, nous avons conçu et développé *DDPJ* (Détection de Défauts des Programmes Java) sous la direction du professeur Tremblay.

Pour cela, nous avons qualifié de défaut toute partie du code qui ne respecte pas certains standards. Au chapitre 2, nous avons établi une liste non exhaustive de défauts typiques dans les programmes des cours d'initiation à la programmation. La plupart de ces défauts ont été illustrés avec des exemples concrets pour mieux comprendre comment ils se présentent dans le code source Java. Nous avons réparti les défauts répertoriés en plusieurs catégories selon les éléments du langage qu'ils affectent. Ceux qui ne correspondaient à aucune catégorie énumérée ont été regroupés et placés dans une catégorie à part.

Avec une liste de défauts, l'évaluation de la qualité du code revient à vérifier la présence de ces défauts dans le code source. La qualité du code est inversement proportionnelle au nombre de défauts. Il se peut aussi qu'on attribue une certaine pondération à chaque défaut pour considérer certains plus graves ou problématiques que d'autres. Sachant exactement ce qu'il faut pour évaluer la qualité du code, on pouvait maintenant envisager de corriger automatiquement les travaux de programmation Java.

Pour développer *DDPJ*, nous avons utilisé d'autres logiciels libres. C'est au chapitre 3 que nous avons présenté brièvement les principaux d'entre eux. On a notamment parlé de *SableCC*, un logiciel qui permet de développer des compilateurs. En effet, *DDPJ* recevant en entrée un code source et fournissant en sortie le résultat de la détection des défauts, peut être vu comme un traducteur. C'est un traducteur spécial qui ne produit pas un code machine mais un rapport texte formaté. C'est ce rapport qui va être récupéré par *Oto* pour l'appréciation de la qualité du code et pour générer le rapport qui sera retourné à l'utilisateur. Pour faciliter le traitement automatique, la compréhension et la lisibilité du rapport produit par *DDPJ*, nous avons utilisé *ObjectMacro* pour le formatage du texte. Cela nous a permis de produire un rapport où les éléments n'apparaissent pas spécialement en fonction de leurs séquences dans le code source, mais de manière groupée et structurée selon leur nature. Ainsi, toutes les occurrences d'un défaut seront présentées en bloc avec, pour chacune d'elles, les informations nécessaires permettant de la localiser dans le code source, ce qui rend le rapport plus lisible et compréhensible. La lisibilité est particulièrement importante pour le rapport à retourner aux étudiants pour leur permettre de bien comprendre les erreurs et s'améliorer par la suite.

La détection de défauts proprement dite a été abordée au chapitre 4. C'est dans ce chapitre que nous avons présenté les différentes stratégies utilisées ou proposées pour détecter les différents défauts. Étant dans un contexte d'enseignement, et plus particulièrement de correction, nous avions besoin de comptabiliser ou quantifier les défauts détectés. Cela permet de mesurer précisément l'ampleur des défauts et de déduire une appréciation de la qualité du code. Cela facilite aussi la comparaison de la qualité entre les codes sources des différents programmes. Ainsi, même en présence de programmes

donnant les mêmes résultats sur le plan fonctionnel, on peut mieux noter les étudiants qui écrivent des codes sources de meilleure qualité car ils auront moins de défauts détectés. Pour toute ces raisons, on a associé chaque défaut à une métrique. Le problème de détection revient donc à calculer les valeurs des métriques associées aux défauts pertinents pour le code source à analyser. Nous avons présenté une stratégie globale et quelques stratégies spécifiques pour certains groupes de défauts. Toutes ces stratégies s'appuient sur la grammaire même du langage de programmation. Il n'est donc pas nécessaire de définir une autre grammaire ni des règles supplémentaires pour détecter les défauts.

Toutes ces idées ont été mises en pratique dans le logiciel, *DDPJ*, présenté au chapitre 5. Nous avons notamment présenté les exigences principales fixées pour le logiciel, sa structure et son fonctionnement. Nous avons aussi abordé les tests unitaires et les tests de validation effectués pour s'assurer du bon fonctionnement de ce logiciel. La conception a été faite pour faciliter la maintenance et les extensions futures. Contrairement à d'autres outils qui nécessitent d'écrire un fichier de configuration pour déterminer les défauts à détecter, *DDPJ* ne nécessite que le passage des noms de métriques en ligne de commande. Il produit deux types de rapport : un rapport simple comportant essentiellement les valeurs des métriques et un rapport détaillé. C'est aussi une application portable qui se distribue et s'installe facilement. Il suffit de copier le fichier jar dans un répertoire accessible en exécution. Il peut être intégré dans n'importe quel outil de correction automatique capable d'exécuter une commande en ligne.

Au chapitre 6, nous montrons quelques exemples d'utilisation de *DDPJ*. Nous montrons, notamment, une utilisation dans le cadre de la correction d'un groupe de travaux. Nous avons illustré un exemple de correction des travaux d'étudiants remis dans le cadre du cours de programmation I (INF1120) de l'UQAM. Le manuel d'utilisation de *DDPJ*, les principaux fichiers de configuration et quelques exemples d'utilisation supplémentaires sont présentés dans les annexes.

La réalisation de ce mémoire a été intéressante et instructive car elle nous a permis d'approfondir nos connaissances en compilation et en qualité logiciel et aussi d'améliorer

notre style de programmation. Il a aussi été une occasion formidable de mieux comprendre la grammaire Java et de maîtriser davantage *SableCC* et *ObjectMacro*. Nous avons pu expérimenter concrètement qu'on pouvait faire avec *SableCC* des choses qui, jusqu'à présent, se faisaient essentiellement avec *JavaCC*. En effet, la plupart des outils d'analyse de qualité que nous avons trouvés utilisent *JavaCC*. A notre connaissance, *DDPJ* est le premier outil développé avec *SableCC* et *ObjectMacro* pour la détection de défauts des programmes Java, dans le contexte de la correction de travaux d'étudiants. Nous avons également approfondi nos connaissances sur les différentes mesures ou métriques que l'on peut appliquer à un code source d'un programme informatique en général, et Java en particulier. Nous avons montré qu'on pouvait automatiser la correction de la qualité du code en associant chaque défaut à une métrique, car le problème revient alors à vérifier les valeurs numériques prises par ces métriques.

En guise de perspectives, plusieurs points de ce mémoire peuvent être étendus dans des recherches futures. C'est notamment le cas de la liste des défauts. Il faut non seulement continuer à répertorier les défauts énumérés dans les standards et conventions, mais aussi recenser les défauts fréquemment constatés par les correcteurs des travaux d'étudiants. Il faut aussi chercher une stratégie plus efficace pour nommer les métriques associées aux défauts et éviter toute confusion lorsque la liste sera plus longue. Une piste à examiner serait d'établir et maintenir une liste numérotée des défauts. Les noms des métriques seraient alors en fonction de ces numéros, ce qui éviterait des mnémoniques qui peuvent devenir de plus en plus difficiles à distinguer au fur et à mesure que la liste croît. Une autre extension possible serait d'ajouter la prise en charge d'autres langages de programmation comme le C/C++, que l'on retrouve souvent dans les cours d'initiation à la programmation. Il est aussi imaginable de faire évoluer le concept pour une application dans un contexte de développement professionnel. Quelque soit le contexte d'utilisation, il serait souhaitable de faire évoluer régulièrement l'application en fonction de l'évolution des langages de programmation pris en charge.

Dans sa version actuelle, *DDPJ* peut être appelé à partir d'*Oto* comme une application externe. Une intégration complète sous forme d'un module *Oto* serait souhaitable.

APPENDICE A

MANUEL D'UTILISATION

Dans ce chapitre, nous présentons le manuel d'utilisation du logiciel *DDPJ*. Le lecteur y trouvera donc les explications des diverses commandes et des options qui leur sont associées.

A.1 Commande principale

Comme nous l'avons dit à la section 5.7, le programme DDPJ est distribué sous forme de fichier jar : `ddpj.jar`. L'usage se fait sous la forme générale suivante :

```
java -jar ddpj.jar [options] fichier [metrique...]
```

Dans cette commande, `fichier` est le nom (éventuellement avec le chemin d'accès) du fichier source dont on veut détecter les défauts. Dans cette version initiale, le fichier source est un fichier Java.

Précisons tout de suite que les options ont préséance sur les arguments, c'est à dire que le traitement du fichier source et le calcul des métriques éventuelles se fait en fonction des options passées à la ligne de commande. Nous donnons les détails des options à la section suivante.

A.1.1 Options

Comme dans l'usage traditionnelle, les options de `ddpj` sont indiquées par une lettre ou une chaîne de caractères précédée du caractère '-' (moins). Dans cette version

initiale, les principales options sont les suivantes :

–h : Affiche l'aide. Cette option a également deux synonymes : `--aide` et `--help`. En présence de cette option, aucun calcul de métrique ne se fait. Le système se contente d'afficher l'aide.

–d : Affichage le rapport détaillé de la détection. Autrement, le programme n'affichera pas de détail. Cette option est surtout intéressante quand on calcule au moins une métrique. On peut aussi utiliser le synonyme `--detail` pour avoir les détails.

–L : Affiche la liste des métriques disponibles dans la langue par défaut. Dans cette version initiale, la langue par défaut est le français.

–Lfr : Affiche la liste des métriques en français.

–Len : Affiche la liste des métriques en anglais.

A.1.2 Calculer les métriques

Pour calculer une métrique, il suffit de la passer en argument après le nom du fichier. A titre d'exemple, pour calculer le nombre d'instruction `IF` dans le fichier `Exemple1.java`, il suffit d'exécuter la commande suivante :

```
java -jar ddpj.jar Exemple1.java NIF
```

Dans ce cas, `DDPJ` retournera la valeur de la métrique demandée. On aura un affichage comme le suivant :

```
NIF = 4
```

A.1.3 Vérification des valeurs attendues

Il est possible de spécifier la valeur attendue d'une métrique ou un intervalle dans laquelle elle devrait se situer. Si, à l'issue de l'exécution, *DDPJ* trouve une valeur conforme à ce qui est attendu, il n'affichera rien. Sinon, il y aura un message d'erreur indiquant la valeur trouvée et ce qui est attendue. Pour préciser la valeur attendue ou son intervalle, il suffit de faire suivre le nom de la métrique d'un opérateur de comparaison puis de la valeur numérique. Les opérateurs de comparaison reconnus sont suivant :

< : Inférieure. La métrique doit être strictement inférieure à la valeur numérique précisée.

<= : Inférieure ou égale.

= : La métrique doit être exactement égale à la valeur numérique précisée.

>= : Supérieure ou égale

> : Strictement supérieure.

Si nous considérons l'exemple de la sous section précédente et que nous voulons préciser que la valeur de métrique `NIF` doit être inférieur ou égale à 2, nous exécuterons la commande suivante :

```
java -jar ddpj.jar Exemple1.java NIF<=2
```

Notons que sur le terminal, cette commande écrite ainsi pourrait provoquer une erreur car le caractère "<" a une signification particulière dans le *shell*. Pour éviter cela, il faut mettre toute la chaine composée de la métrique et ses arguments ainsi que les options éventuelles entre guillemets. Sur un terminal, la commande précédente serait donc plutôt écrite de la manière suivante :

```
java -jar ddpj.jar Exemple1.java "NIF<=2"
```

Pour ce cas précis du fichier `Exemple1.java`, *DDPJ* affichera ce qui suit :

```
Condition non satisfaite NIF: Valeur obtenue = 4; attendue: <= 2
```

Signalons que sur une même ligne de commande, on peut demander la vérification des valeurs attendues sur certaines métriques et demander tout simplement le calcul des valeurs prises par d'autres métriques. C'est le cas de la commande ci-après qui spécifie qu'il ne faut pas avoir plus de 2 instructions `if` et demande en même temps de savoir le nombre d'instructions `else`.

```
java -jar ddpj.jar Exemple1.java "NIF<=2" "NIE"
```

Le résultat produit sera alors les deux lignes suivantes :

```
NIE = 2
Condition non satisfaite NIF: Valeur obtenue = 4; attendue: <= 2
```

A.2 Passage des options aux métriques

Certaines métriques calculées avec *DDPJ* offrent des options pour préciser les conditions à vérifier. En l'absence de ces options en ligne de commande, les valeurs par défauts seront considérées. Pour spécifier une option à une métrique précise, il suffit de faire suivre le nom de la métrique du caractère ':' puis de l'option. A titre d'exemple, si on veux calculer le nombre de méthode déclarées avec trop de paramètres, on utilisera la métrique `NMDTP`. Une question qui subsiste est de déterminer le nombre à partir duquel on considère qu'il y a trop de paramètres. On peut répondre à cette question en précisant l'option `MAX`. Ainsi si on veut détecter les méthodes déclarées avec plus de 3 paramètres, on utilisera la commande ci-après :

```
java -jar ddpj.jar Exemple1.java "NMDTP:MAX=3"
```

DDPJ permet aussi de vérifier la valeur attendue tout en précisant les options. Pour vérifier s'il y a plus de 2 méthodes déclarées avec plus de 3 paramètres, on exécutera la commande suivante :

```
java -jar ddpj.jar Exemple1.java "NMDTP<=2"  "NMDTP:MAX=3"
```

A.3 Liste des métriques disponibles

Dans cette section, nous reprenons la liste (en français) des métriques actuellement disponibles.

NBHS = Nombre d'instructions `break` hors `switch`

NCD = Nombre de constructeurs déclarés

NCDTP = Nombre de constructeurs déclarés avec trop de paramètres

NCMG = Nombre de constantes magiques

NCNM = Nombre de constantes de classe pas entièrement en majuscules

NCSB = Nombre de `case` sans `break`

NDMV = Nombre de déclarations multiples de variables

NDV = Nombre de déclarations de variables

NESA = Nombre d'instructions `else` sans accolade

NETA = Nombre d'opérateurs étoile-assignation (`*=`)

NFML = Nombre d'instructions `for` avec modification de la variable d'itération dans le bloc

NIE = Nombre d'instructions `else`

NIEV = Nombre d'instructions `else` vides

NIF = Nombre d'instructions `if`

NIFO = Nombre d'instruction `for`

NII = Nombre d'initialisations inutiles de variables locales

NIS = Nombre d'instructions `switch`

NITV = Nombre d'instructions `then` vides

NIW = Nombre d'instructions `while`

NIWV = Nombre d'instructions `while` vides

NMA = Nombre d'opérateurs moins-assignation (`-=`)

NMD = Nombre de méthodes déclarées

NMDTP = Nombre de méthodes déclarées avec trop de paramètres

NMM = Nombre d'opérateurs moins-moins (`--`)

NPA = Nombre d'opérateurs plus-assignation (`+=`)

NPCA = Nombre d'opérateurs pourcent-assignation (`%=`)

NPP = Nombre d'opérateurs plus-plus (`++`)

NPTI = Nombre d'opérateurs point d'interrogation (`?`)

NSLA = Nombre d'opérateurs slash-assignation (`/=`)

NTSA = Nombre d'instructions `then` sans accolade

NVD = Nombre de variables déclarées

NVNID = Nombre de variables non initialisées à la déclaration

NWCF = Nombre d'instructions `while` potentiellement convertibles à un `for`

NWSA = Nombre d'instructions `while` sans accolade (non vides)

APPENDICE B

AUTRES EXEMPLES D'UTILISATION DE DDPJ AVEC UN RAPPORT DÉTAILLÉ

Dans ce chapitre, nous montrons deux exemples d'exécution de *DDPJ* avec des rapports détaillés. Nous allons illustrer le fait que même pour des métriques calculées dans la même classe, le rapport ne contiendra que les informations des métriques demandées.

B.1 Code souce

Considérons le code source source du listing B.1 dont on veut analyser la qualité avec *DDPJ*.

B.2 Premier exemple d'exécution avec résultats détaillés

Considérons le code source du listing B.1 et cherchons à détecter les défauts suivants :

- Les boucles `while` potentiellement convertibles en boucle `for`
- Les variables initialisées inutilement
- Les instructions `case` sans `break`
- Les instructions `break` hors `switch`

En consultant la liste des métriques disponibles à la section A.3, nous trouvons que ces défauts correspondent aux métriques NWCF, NII, NCSB, NBHS respectivement.

Listing B.1 Exemple de code source Java à examiner : fichier `Exemple1.java`

```
2  public class Exemple1 {
3          final static int DIX = 10;
4          final static int Huit = 8; //pas tout en majuscule
5          static int n = 20;
6          public static void main( String[] args ) {
7                  int i = 0;
8                  int j = 12;
9                  i= ( int )(Math.random() * 30);
10                 while( i < n ) {
11                         if( i < j ) {
12                                 j = 2 * i;
13                                 break;
14                         }
15                         else { }; //else vide
16                         j = j + 1;
17                         if( j > DIX ){
18                                 if( i < 16 ){
19                                         j = 2 * i;
20                                         System.out.println( "valeur i = " + i + " et ↙
                                   ↳ valeur de j = "+j );
21                                 }
22                                 else System.out.println("val j = " + j);// accolades!
23                         }
24                         i += 2;
25                 }
26                 System.out.println( "i = " + i );
27                 switch( i ) {
28                 case 5: j = j - 5; break;
29                 case DIX: j = j + 10; break;
30                 case 15: j = j * 15; //oubli de break
31                 case 20: j = ( int )( j/i ); break;
32                 default: i = Huit; j = 0;
33                 }
34                 System.out.println( "Val i = " + ( i + j ) );
35         }
36 }
```

Nous aimerions aussi avoir un rapport détaillé de cette détection, c'est-à-dire, avoir des indications précises sur les emplacements des défauts. Pour cela, nous exécutons la commande ci-après :

```
java -jar ddpj.jar -d Exemple1.java  NWCF NII NCSB NBHS
```

Le résultat obtenu est dans le listing B.2[1].

B.3 Deuxième exemple d'exécution avec résultats détaillés

Dans cette deuxième exécution avec des détails, nous voulons illustrer le fait que *DDPJ* ne fournit en rapport que les résultats des détections demandées. Même pour les métriques qui sont calculées dans une même classe, *DDPJ* n'affichera que les résultats demandés. En observant cet exemple et celui de la section précédente, on voit bien que *DDPJ* respecte le choix de l'utilisateur. C'est à l'utilisateur de déterminer les défauts à détecter. Ceci a l'avantage de ne fournir en rapport que ce qui intéresse l'utilisateur.

Prenons toujours le code du listing B.1 mais, cette fois-ci, intéressons nous aux défauts suivants :

- Les instructions `else` vides
- Les instructions `else` sans accolades
- L'instruction `break` en dehors de `switch`
- Nom de constante de classe pas entièrement en majuscule

En regardant la liste des métriques A.3, nous obtenons pour ces défauts les métriques NIEV, NESA, NBHS et NCNM respectivement. Exécutons maintenant *DDPJ* avec la commande suivante :

```
java -jar ddpj.jar -d Exemple1.java NIEV NESA  NBHS NCNM
```

Nous obtenons alors le résultat se trouvant dans le listing B.3.

---

1. Pour tenir sur une seule page, les lignes blanches ne sont pas affichées

Listing B.2 Exemple de résultats détaillés pour le programme du listing B.1

```
NWCF = 1
NII = 1
NCSB = 1
NBHS = 1
Rapport detaille
---------------
Nombre d'initialisations inutiles: 1
Nombre de "while" potentiellement convertibles en "for": 1
Nombre d'instructions "case" sans "break": 1
Nombre d'instructions "break" en dehors de "switch" : 1
Resultats de la detection
-------------------------
Detection de "while" convertibles en for"
-----------------------------------------
While potentiellement convertible a un for: i < n
Variable d'iteration: i
Ligne numero: 10
Suggestion: Verifier s'il ne faudrait pas plutot utiliser une boucle "for"
Detection des "case" sans "break"
---------------------------------
"case" sans break : case 15
Ligne numero: 30
Detection des "break" en dehors de "switch"
-------------------------------------------
break hors switch : break ;
Ligne numero: 13
Detection des initialisations inutiles de variables locales
-----------------------------------------------------------
Initialisation inutile: i = 0
Ligne numero: 7
Explication: Dans toutes les executions possibles, la variable i est redefinie sans ↙
                        ↳ avoir ete utilisee.
```

Listing B.3 Deuxième exemple de résultats détaillés pour le programme du listing B.1

```
NCNM = 1
NESA = 1
NBHS = 1
NIEV = 1
Rapport detaille
----------------
Nombre de noms de constantes pas toutes en majuscules: 1
Nombre de "else" vides: 1
Nombre de "else" sans accolade: 1
Nombre d'instructions "break" en dehors de "switch" : 1
Resultats de la detection
-------------------------
Detection des noms de constantes pas tout en majuscules
-------------------------------------------------------
Nom de constante pas tout en majuscules: Huit
Ligne numero: 4
Detection de "else" inutiles (else vide)
----------------------------------------
if (i < j )
(...)
else{ }
Ligne numero: 15
Probleme potentiel: "else" vide
Suggestion: Verifier si le "else" peut etre supprime.
Detection de "else" sans accolade
-------------------------------
if (i < 16 )
(...)
else (...)
Ligne numero: 22
Probleme potentiel: "else" sans accolade
Suggestion: Mettre l'instruction dans des accolades.
Detection des "break" en dehors de "switch"
-------------------------------------------
break hors switch : break ;
Ligne numero: 13
```

APPENDICE C

PRINCIPAUX FICHIERS DE CONFIGURATION

Le lecteur trouvera dans cette annexe les principaux fichiers de configuration utilisés dans cette première version de *DDPJ*.

C.1 Gabarit d'affichage ObjectMacro

Le listing C.1 montre quelques extraits du fichier du gabarit d'affichage (template) ObjectMacro utilisé actuellement pour la version française des rapports. Compte tenu de la longueur du gabarit, nous n'affichons dans ce listing que quelques échantillons des macros pour permettre au lecteur d'avoir une idée générale du découpage des sections du rapport produit par *DDPJ*.

C.2 Configuration système des métriques

Le lecteur trouvera dans le listing C.2 la correspondance entre les noms systèmes des métriques et les classes qui les calculent. Comme nous l'avons dit précédemment, en principe, les noms des métriques indiqués ici ne sont utilisés que par le système pour déterminer les classes à exécuter.

C.3 Les noms des métrique en français

Les listings C.3, C.4 et C.5 montrent la configuration et la description des métriques en Français.

Listing C.1 Extrait du gabarit d'affichage de DDPJ : `defectdetection-template-fr.objectmacro`

```
$macro: program $
Rapport detaille
----------------
$macro: statistics$
$macro: title$
Statistiques de detection des defauts
$end: title$
$comment: ---extait de la section des macros de statistiques----$
$macro: numberofmultiplevariabledeclaration(nbmvar)$
Nombre de declarations multiples de variables: $(nbmvar)
$end: numberofmultiplevariabledeclaration$
$comment: ----------------Fin Extrait-----------------------$
$end: statistics$
$expand: statistics$
Resultats de la detection
-------------------------
$comment: --extrait de la section des macros de blocs de rapport detaille---$
$macro: blockmultiplevariablesdeclaration$
Detection des declarations multiples de variables
-------------------------------------------------
$expand: multiplevariablesdeclaration$
$end: blockmultiplevariablesdeclaration$
$comment: ---- -- ----------------------$
$macro: multiplevariablesdeclaration(multidecl,mvarln)$
Variables declarees: $(multidecl)
Ligne numero: $(mvarln)
$end: multiplevariablesdeclaration$
$comment: ---- --Fin extrait----------------$
$end: program$
```

Listing C.2 Classes calculant les métriques `ddpjConfig.properties`

```
# ---nom interne des metriques et classes les implementant
NIF=NeedlessIfDetection
NIFE=NeedlessIfDetection
NETP=NeedlessIfDetection
NEEP=NeedlessIfDetection
NMGC=NonMagicConstantDetection
NUCC=NoneUpperCaseConstantNameDetection
NTWB=IfWithoutBracesDetection
NEWB=IfWithoutBracesDetection
NDC=TooManyParametersInMethodDetection
NDM=TooManyParametersInMethodDetection
NDCMP=TooManyParametersInMethodDetection
NDMMP=TooManyParametersInMethodDetection
NWI=WhileConvertibleToFor
NEWS=WhileConvertibleToFor
NWWB=WhileConvertibleToFor
NWCF=WhileConvertibleToFor
NVD=VariablesDeclarationDetection
NDV=VariablesDeclarationDetection
NMVD=VariablesDeclarationDetection
NSVD=VariablesDeclarationDetection
NCWB=CaseWithoutBreakDetection
NSS=CaseWithoutBreakDetection
NBOS=CaseWithoutBreakDetection
NPP=OperatorsDetection
NMM=OperatorsDetection
NPA=OperatorsDetection
NMA=OperatorsDetection
NPCA=OperatorsDetection
NSLA=OperatorsDetection
NSTA=OperatorsDetection
NQMK=OperatorsDetection
NNI=NeedlessInitializationDetection
NFI=ForLoopModificationInsideBlock
NFLA=ForLoopModificationInsideBlock
```

Listing C.3 Description et configuration des métriques en français : `metrics-fr.properties` (voir aussi les listings C.4 et C.5)

```
# -- description des metriques en francais
NIF=NIF
NIF.DESC=Nombre d'instructions if
#-----------
NIE=NIFE
NIE.DESC=Nombre d'instructions else
#-----------
NITV=NETP
NITV.DESC=Nombre d'instructions then vides
#-----------
NIEV=NEEP
NIEV.DESC=Nombre d'instructions else vides
#-----------
NCMG=NMGC
NCMG.DESC=Nombre de constantes magiques
#-----------
NCNM=NUCC
NCNM.DESC=Nombre de constantes de classe pas entierement en majuscules
#-----------
NTSA=NTWB
NTSA.DESC=Nombre d'instructions then sans accolade
#-----------
NESA=NEWB
NESA.DESC=Nombre d'instructions else sans accolade
#-------------
NCD=NDC
NCD.DESC=Nombre de constructeurs declares
#-------------
NMD=NDM
NMD.DESC=Nombre de methodes declarees
#-------------
NMDTP=NDMMP
NMDTP.DESC=Nombre de methodes declarees avec trop de parametres
#-------------
```

Listing C.4 Configuration des métriques en français : suite du listing C.3

```
NCDTP=NDCMP
NCDTP.DESC=Nombre de constructeurs declares avec trop de parametres
#-------------
NIW=NWI
NIW.DESC=Nombre d'instructions while
#-------------
NIWV=NEWS
NIWV.DESC=Nombre d'instructions while vides
#-------------
NWSA=NWWB
NWSA.DESC=Nombre d'instructions while sans accolade (non vides)
#-------------
NWCF=NWCF
NWCF.DESC=Nombre d'instructions while potentiellement convertibles a un for
#-------------
NDV=NVD
NDV.DESC=Nombre de declarations de variables
#-------------
NVD=NDV
NVD.DESC=Nombre de variables declarees
#-------------
NDMV=NMVD
NDMV.DESC=Nombre de declarations multiples de variables
#-------------
NVNID=NSVD
NVNID.DESC=Nombre de variables non initialisees a la declaration
#-------------
NCSB=NCWB
NCSB.DESC=Nombre de case sans break
#-------------
NIS=NSS
NIS.DESC=Nombre d'instructions switch
#-------------
NBHS=NBOS
NBHS.DESC=Nombre d'instructions break hors switch
#-------------
```

Listing C.5 Configuration des métriques en français : suite et fin du listing C.4

```
NPP=NPP
NPP.DESC=Nombre d'operateurs plus-plus (++)
#-------------
NMM=NMM
NMM.DESC=Nombre d'operateurs moins-moins (--)
#-------------
NPA=NPA
NPA.DESC=Nombre d'operateurs plus-assignation (+=)
#-------------
NMA=NMA
NMA.DESC=Nombre d'operateurs moins-assignation (-=)
#-------------
NPCA=NPCA
NPCA.DESC=Nombre d'operateurs pourcent-assignation (%=)
#-------------
NSLA=NSLA
NSLA.DESC=Nombre d'operateurs slash-assignation (/=)
#-------------
NETA=NSTA
NETA.DESC=Nombre d'operateurs etoile-assignation (*=)
#-------------
NPTI=NQMK
NPTI.DESC=Nombre d'operateurs point d'interrogation (?)
#-------------
NII=NNI
NII.DESC=Nombre d'initialisations inutiles de variables locales
#-------------
NIFO=NFI
NIFO.DESC=Nombre d'Instruction for
#-------------
NFML=NFLA
NFML.DESC=Nombre d'instructions for avec modification de la variable d'iteration dans ↙
                        ↳ le  bloc
```

BIBLIOGRAPHIE

Arusoaie, A., et D. I. Vicol. 2012. « Automating abstract syntax tree construction for context free grammars ». In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, p. 152–159.

Bergmann, S. D. 2007. *Compiler Design : Theory, Tools, and Examples Java Edition.* Rowan University. <`http://cs.rowan.edu/~bergmann/books`>.

Blumenstein, M., S. Green, A. Nguyen et V. Muthukkumarasamy. 2004. « GAME : a generic automated marking environment for programming assessment ». In *Information Technology : Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on.* T. 1, p. 212 – 216 Vol.1.

Burn, O. 2001. Checkstyle. Site Web. Dernière visite : 21.04.2013. <`http://checkstyle.sourceforge.net`>.

Gagnon, E. M. 1998. « SableCC, an object-oriented compiler framework ». Mémoire de maîtrise, School of Computer Science, McGill University, Montreal.

Gagnon, E. M., et L. Hendren. 1998. « SableCC, an object-oriented compiler framework ». In *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, p. 140 –154.

Guérin, F. 2005. « Oto, un outil générique et extensible pour corriger les travaux de programmation ». Mémoire de maîtrise, Université du Québec à Montréal (UQAM). <`http://oto.uqam.ca/documents/memoire.pdf`>.

Helmick, M. T. 2007. « Interface-based programming assignments and automatic grading of Java programs ». In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education.* Coll. « ITiCSE '07 », p. 63–67, New York, NY, USA. ACM. <`http://doi.acm.org/10.1145/1268784.1268805`>.

Hristova, M., A. Misra, M. Rutter et R. Mercuri. 2003. « Identifying and correcting Java programming errors for introductory computer science students ». *SIGCSE Bull.*, vol. 35, p. 153–156. <`http://doi.acm.org/10.1145/792548.611956`>.

Jubair, A. J., et E. S. Khair. 2004. « Automark++ : A case tool to automatically mark student Java programs ». *The International Arab Journal of Information Technology.* <`http://www.ccis2k.org/iajit/PDF/vol2.2,no.1/10-Jubair.pdf`>.

Kernighan, B. W., et P. J. Plauger. 1978. *The Elements of Programming Style.* McGraw-Hill Book company, 2e édition. ISBN 0-07-034207-5, chap. 1 p. 2.

Kolbe, C. 2002. Java measurement tool. Site Web. Dernière visite : 21.05.2013. <http://ivs.cs.uni-magdeburg.de/sw-eng/agruppe/forschung/tools/>.

Lessard, P. 2010. « Un langage spécifique au domaine pour l'outil de correction de travaux de programmation Oto ». Mémoire de maîtrise, Université du Québec à Montréal (UQAM). <http://oto.uqam.ca/documents/memoire-lessard.pdf>.

Liang, Y. D. 2008. Introduction to Java programming. Site Web. Dernière visite : 17.04.2013. <http://www.cs.armstrong.edu/liang/intro6e/intro6estudentFAQ.html>.

M874. 2003. Site Web. M874 Software development for networked applications using Java, une page web du cours., Dernière visite : 17.04.2013. <http://www.open.ac.uk/StudentWeb/m874/!synterr.htm>.

Morris, D. 2003. « Automatic grading of students' programming assignments : an interactive process and suite of programs ». In *Frontiers in Education, 2003. FIE 2003. 33rd Annual.* T. 3, p. S3F – 1–6 vol.3.

ObjectMacro. 2011. ObjectMacro. Site Web. Dernière visite : 01.02.2013. <http://sablecc.org/wiki/ObjectMacro/>.

Oto. 2005. Oto, un outil générique et extensible pour corriger les travaux de programmation. Site Web. Dernière visite : 16.04.2013. <http://oto.uqam.ca/>.

PMD. 2009. PMD. Site Web. Dernière visite : 21.04.2013. <http://pmd.sourceforge.net>.

Redish, K. A., et W. F. Smyth. 1986. « Program style analysis : a natural by-product of program compilation ». *Commun. ACM*, vol. 29, no. 2, p. 126–133. <http://doi.acm.org/10.1145/5657.5661>.

SableCC. 2000. SableCC. Site Web. Dernière visite : 30.01.2013. <http://sablecc.org/>.

Sauer, F. 2002. Eclipse metrics plugin. Site Web. Dernière visite : 21.05.2013. <http://sourceforge.net/projects/metrics>.

Skevoulis, S., et J. Xiaoping. 2000. « Generic invariant-based static analysis tool for detection of runtime errors in Java programs ». In *Technology of Object-Oriented Languages and Systems, 2000. TOOLS-Pacific 2000. Proceedings. 37th International Conference on*, p. 36 –44. <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=891356>.

Sun Microsystems, I. 1999. Code conventions for the Java programming language.

Site Web. Dernière visite : 17.04.2013. <http://www.oracle.com/technetwork/java/codeconv-138413.html>.

Tremblay, G. 2011. Style de programmation et qualité du code. Site Web. INF5171 Programmation concurrente et parallèle, Notes de cours de la session d'hiver. <http://www.labunix.uqam.ca/~tremblay/INF5171/Materiel/kiss-dry.pdf>.

Tremblay, G., F. Guérin, A. Pons et A. Salah. 2008. « Oto, a generic and extensible tool for marking programming assignments ». *Softw. Pract. Exper.*, vol. 38, p. 307–333. <http://portal.acm.org/citation.cfm?id=1345485.1345488>.

Tremblay, G., et E. Labonté. 2003. « Semi-automatic marking of Java programs using JUnit ». In *International Conference on Education and Information Systems : Technologies and Applications (EISTA'03)*, p. 42–47. http://www.info2.uqam.ca/~tremblay/TremblayLab03.pdf.

Tsheke, J. 2011. Détection des défauts dans les programmes. INF7441 — Compilation, Prof. Gagnon, E.M., Projet de session d'été.

Wang, X. O., E. Baik et P. T. Devanbu. 2011. « Operating system compatibility analysis of eclipse and netbeans based on bug data ». In *Proceedings of the 8th Working Conference on Mining Software Repositories*. Coll. « MSR '11 », p. 230–233, New York, NY, USA. ACM. <http://doi.acm.org.proxy.bibliotheques.uqam.ca:2048/10.1145/1985441.1985479>.

Wikipedia. 2008. Abstract syntax tree. Site Web. Dernière visite : 17.04.2013. <http://en.wikipedia.org/wiki/Abstract_syntax_tree>.

Xiaoping, J., S. Sawant, J. Zhou et S. Skevoulis. 1999. « Detecting null pointer violations in Java programs ». In *Computer Software and Applications Conference, 1999. COMPSAC '99. Proceedings. The Twenty-Third Annual International*, p. 456 – 461.

Xiaoping, J., et S. Skevoulis. 1999. « A generic approach of static analysis for detecting runtime errors in Java programs ». In *Computer Software and Applications Conference, 1999. COMPSAC '99. Proceedings. The Twenty-Third Annual International*, p. 67 –72.

Ziring, N. 2011. Welcome to the Java mistakes page! Site Web. Dernière visite : 17.04.2013. <http://users.erols.com/ziring/java-npm.html>.