UNIVERSITÉ CATHOLIQUE DE LOUVAIN

ÉCOLE POLYTECHNIQUE DE LOUVAIN

Vers une analyse automatique de l'acquisition de compétences de programmation en Java

Mémoire présenté en vue de l'obtention du grade de
Master 60 crédits en sciences informatiques

Auteur:
Bastien BODART

Promoteur:
Pr. Olivier BONAVENTURE

Lecteurs:
Pr. Charles PECHEUR
Fabien DUCHÊNE

26 mai 2014

Table des matières

Chapitre 1

Introduction

Le cours LSINF1101 "Introduction à la programmation" est le premier cours de programmation que reçoivent les futurs ingénieurs et informaticiens lors de leur formation à l'Université catholique de Louvain. Celui-ci, en plus des séances magistrales, est organisé en différentes missions, chacune d'entre elles portant sur une partie précise de la matière et permettant aux étudiants de mettre en pratique des concepts fondamentaux de la programmation.

Ainsi, lors de chacune de ces missions, les étudiants, répartis au sein de différents groupes sont amenés à soumettre des fichiers sources de classes Java qui doivent répondre à des spécifications bien précises.

Le but du présent travail sera double :

- mettre en place un système d'analyse afin d'une part de fournir un outil pratique au corps enseignant permettant la correction automatique de ces fichiers parfois nombreux et d'autre part de récolter de précieuses données sur les erreurs commises par les étudiants telles que leur type, leur fréquence et leur importance
- utiliser ces données afin d'identifier les concepts clés dont la compréhension pose le plus de problèmes aux étudiants et de là, imaginer de futures missions plus ciblées vers ces mêmes concepts

Afin de parvenir à ce but, plusieurs sujets devront être abordés.
Nous commencerons par un état de l'art, qui tentera, au travers d'une liste qui vu l'ampleur du sujet se devra d'être non-exhaustive, de donner un aperçu des différentes solutions existantes et outils à disposition dans les domaines de l'analyse de code, de test de programmes et d'enseignement du langage Java.

Nous poursuivrons par une présentation du logiciel d'analyse automatique des fichiers des étudiants développé spécialement dans le cadre de ce travail et baptisé MissonAnalyzer. Nous nous attarderons sur ses fonctionnalités ainsi que sur les choix d'implémentation et problèmes rencontrés.

Nous entrerons ensuite dans le vif du sujet avec un passage en revue des différentes missions et des concepts sous-jacents auxquels elles se rapportent. Cependant, la configuration des missions fait que certaines d'entre elles ne seront pas testées car soit les spécifications fournies produisent un résultat qui n'est que très difficilement testable voire pas du tout, soit le concept de la mission n'est pas assez important pour que l'on s'y attarde. S'ensuivra une analyse des données récoltées lors de la correction de ces missions. Cette analyse, en fournissant une vision globale, devrait permettre de mettre au jour les difficultés principales rencontrées par les étudiants.

Pour terminer, muni du savoir acquis précédemment, nous pourrons nous lancer dans des propositions d'élaboration de nouveaux exercices et dans la modification des missions existantes afin de palier aux insuffisances identifiées. Ces propositions pourront le cas échéant servir de base lors d'une future mise à jour de la matière et de la pédagogie du cours.

Chapitre 2

État de l'art

Une rapide recherche permet de se rendre compte qu'il existe une multitude d'environnements de test et d'outils d'analyse de code, pour Java ou pour d'autres langages. Pour une grande part, ces logiciels, API ou autres frameworks sont des projets collaboratifs sous licence libre mais il existe aussi des solutions professionnelles plus évoluées à destination des entreprises, regroupant en un seul produit toute une série d'outils ayant pour vocation l'écriture de code de qualité, la maintenance de celui-ci et surtout la mise en place de suites de tests afin de s'assurer d'une correction la plus élevée possible.

Ces logiciels sont pour la plupart statiques dans le sens où ils sont voués à fournir au programmeur une appréciation de l'ensemble de son travail une fois celui-ci réalisé. Il existe néanmoins certains environnements plus dynamiques, à vocation purement académique, dans lequel le programmeur apprenti se voit soumettre des séries d'exercices, séparées par thématiques, pour lesquelles il obtient une correction immédiate accompagnée d'indications sur ses éventuelles erreurs.

On peut donc dégager trois catégories d'outils permettant d'analyser le travail d'un programmeur débutant à savoir, les analyseurs de code source, qui, en se basant sur des règles et des conventions d'écriture, fournissent une série de recommandations à appliquer afin de produire du code efficace et optimal, les frameworks et environnements de test, qui permettent de vérifier la correction du code en testant les résultats produits, globalement ou point par point au moyen de tests unitaires, et enfin les environnements interactifs permettant un apprentissage progressif des compétences requises en programmation. Nous allons maintenant présenter quelques-uns des outils existants dans ces trois catégories, en particulier ceux dont nous aurons besoin pour la mise en place de notre solution d'analyse.

2.1 Outils d'analyse de code

2.1.1 Xlint

`Xlint` est une des options de *javac*, le compilateur du langage Java[1]. Cette option permet de lever des avertissements, warnings en anglais, lors de la compilation de code Java. Ceux-ci sont alors affichés sur la sortie standard.

De nombreux types d'avertissement sont disponibles, l'activation de ceux-ci se faisant par l'option `-Xlint:name` où `name` est le nom du type d'avertissement. L'option `-Xlint:all` permet de tous les activer, cette pratique est recommandée par Oracle pour toute utilisation du compilateur.

Voici une liste non-exhaustive des types d'avertissement les plus susceptibles d'être rencontrés dans le cas qui nous occupe :

cast
Présence de conversions de type (casting) inutiles ou redondantes

deprecation
Utilisation d'éléments dépréciés

divzero
Présence d'une division par l'entier 0

empty
Présence d'une expression vide après un `if`

fallthrough
Présence d'un case dans un `switch`, autre que le dernier, qui ne se termine pas par `break`

finally
Présence d'une clause `finally` qui ne pourra être exécutée

rawtypes
Utilisation de types bruts

static
Problèmes lors de l'utilisation du modificateur `static`

try
Problèmes lors de l'utilisation de blocs `try`. Notamment quand des ressources ne sont pas utilisées

2.1.2 PMD

PMD[2] est un analyseur de code source supportant les langages Java, Javascript, XML et XSL. Il est cependant le plus utilisé pour Java.

C'est un logiciel sous licence libre, écrit en Java et développé à partir de 2002 avec le soutien de la DARPA [1], Defense Advanced Research Project Agency. Le programme en est actuellement à la version 5.1.1, publiée le 27 avril 2014. Le

1. `http://www.darpa.mil/`

projet, hébergé sur SourceForge, est maintenu par deux dizaines de développeurs et dix fois plus de contributeurs.

Le logiciel, qui comprend plusieurs librairies, est utilisable sous forme de plugin pour différents IDE tels que Eclipse, NetBeans ou BlueJ, au sein des outils Ant ou Maven, ainsi que sous forme d'exécutable en ligne de commande, tant sous Windows que les systèmes UNIX. Après son analyse d'un fichier ou d'un ensemble de fichiers ou dossiers, le logiciel peut fournir les résultats sous diverses formes comme du texte sur la sortie standard, un rapport HTML ou XML.

L'analyse est basée sur différents ensembles de règles pour lesquels le code source est testé. Ces règles sont en fait des règles syntaxiques qui doivent être remplies sous peine de soulever une erreur de violation. Différents ensembles de règles sont fournis avec le logiciel, chacun traitant d'un domaine particulier. L'utilisateur peut aussi créer lui même de nouvelles règles ou de nouveaux ensembles, s'il possède des connaissances en analyse syntaxique. Les ensembles destinés à Java sont désignés par un nom de la forme `java-name` où `name` est un mot ou ensemble de mot résumant les règles à appliquer.

Voici quelques ensembles de règles qui peuvent s'avérer utiles dans notre cas :

Basic
: Règles de bases concernant des bonnes pratiques qu'idéalement il conviendrait de suivre lors de l'écriture de code Java. Par exemple : éviter des `if` testant des conditions toujours vraies ou toujours fausses, éviter de vérifier un `null` après une autre vérification, éviter de multiples opérateurs unaires, etc

Code Size
: Règles permettant de vérifier que la taille des éléments du code conservent un taille acceptable. Par exemple : les méthodes trop longues, trop de paramètres pour une méthode, trop de champs dans une classe, etc

Comments
: Règles concernant les commentaires telles que des commentaires trop longs

Controversial
: Règles jugées controversées, donc pas forcément obligatoires, de pratiques à éviter. Par exemple : une assignation de `null`, un constructeur inutile, une variable locale finale

Design
: Règles permettant de remarquer le code non-optimal tel que la non-fermeture de ressources, l'absence de cas `default` dans un `switch` ou l'utilisation de `.equals(null)`

Empty Code
: Règles qui vérifient que le code ne contient pas de passages vides comme des méthodes vides, des `catch` vides, des blocs vides, etc

Naming
: Règles qui vérifient l'usage des noms et identifiants afin qu'ils ne soient pas trop longs, trop courts et qu'ils respectent les conventions

Optimization
Règles vérifiant l'optimisation du code comme l'utilisation préférentielle de paramètres de méthodes finaux, l'utilisation d'`ArrayList` au lieu de `Vector`, l'utilisation de `asList`, etc

Strict Exceptions
Règles pour la bonne mise en œuvre des exceptions comme éviter le `catch` d'objets `Throwable`, éviter la perte d'information, éviter de lancer une exception dans un bloc `finally`, etc

String and StringBuffer
Règles pour l'utilisation des `String` et des `StringBuffer` telles qu'éviter d'instancier des `String`, d'éviter de répéter des littéraux, d'appeler `toString` sur un `String`, etc

Unnecessary
Règles pour éviter l'utilisation de code inutile comme les conversions temporaires, les parenthèses ou les `return` inutiles

Unused Code
Règles pour éviter l'emploi de code inutilisé comme les variables locales non-lues ou les méthodes privées jamais appelées

Le fonctionnement du logiciel PMD est globalement le suivant :
- Un ou des ensembles de règles ainsi qu'un ou des noms de fichiers ou dossiers sont passés en arguments à PMD
- PMD fournit ensuite un `InputStream` du ou des fichiers à un parser généré par JavaCC et reçoit en retour une référence à un Abstract Syntax Tree (AST)
- L'AST est passé à la table des symboles qui crée les portées, trouve les déclarations et les utilisations des variables
- Chaque règle traverse ensuite l'AST et vérifie l'existence de problèmes
- Un rapport est rempli des violations de règles et est ensuite imprimé sous la forme choisie

On peut également noter que PMD inclus un outil de détection de copie de code appelé CPD, pour copy-paste-detector. Celui-ci fonctionne pour Java, C, C++, C#, PHP, Ruby, Fortran et JavaScript. Il est plutôt destiné à vérifier la présence de code copié dans les gros projets mais pourrait éventuellement servir à une détection du plagiat entre les étudiants.

2.1.3 Checkstyle

Checkstyle[3] est un logiciel libre sous licence GNU Lesser General Public License, hébergé sur SourceForge, actuellement développé par une demi-douzaine de personnes. La version 5.7 a été publiée le 3 février 2014.

Sous bien des aspects, Checkstyle est fort semblable à PMD. Il permet en effet d'analyser du code source Java par rapport à des ensembles de règles fort variés

concernant la duplication de code, les commentaires, les importations, etc. Comme PMD, il permet l'écriture de règles personnalisées.

Cet outil se focalise néanmoins davantage sur le respect de conventions d'écriture de code telles que l'ordre des modificateurs, la présence de virgules et d'espaces, l'indentation ou encore la présence d'accolades. Ces conventions sont généralement basées sur les conventions classiques du langage Java [2] mais peuvent être modifiées par l'utilisateur afin d'imposer ses propres règles.

L'utilisation de Checkstyle peut se faire grâce à un plugin pour plusieurs IDE tels que Eclipse, NetBeans, BlueJ ou Emacs, par ligne de commande avec ensuite génération d'un rapport en texte brut ou XML ou au moyen d'une tâche Ant. Cette dernière option est souvent utilisée lors de la construction de projets volumineux, ce qui permet de garder une cohérence dans le code des différentes classes, chose particulièrement utile si des personnes différentes avec leurs propres habitudes sont impliquées.

2.1.4 FindBugs™

Finbugs[4] est également un logiciel libre sous licence GNU Lesser General Public License, hébergé sur SourceForge et développé au sein de l'Université du Maryland [3]. La dernière version, 2.0.3, a été publiée le 22 novembre 2013.

C'est un outil permettant la découverte de bugs, non pas dans le code source, mais directement dans le bytecode Java en utilisant le principe de l'analyse statique, c'est à dire une simple lecture et non une exécution. Le programme va en fait chercher dans le bytecode des "bug patterns", qui sont des morceaux de code considérés comme erronés. Néanmoins, cette analyse étant parfois imprécise, il se peut que le programme rapporte des erreurs qui n'en sont pas.

Le logiciel est fourni sous forme de fichier `.jar` et peut être utilisé en ligne de commande pour fournir un rapport des fichiers analysés ou par le biais d'une interface graphique fort pratique car permettant d'afficher les erreurs trouvées dans le code source correspondant au bytecode. Il peut aussi être utilisé sous forme de plugin pour Eclipse ou avec une tâche Ant.

2. `http://www.oracle.com/technetwork/java/index-135089.html`
3. `http://www.umd.edu/`

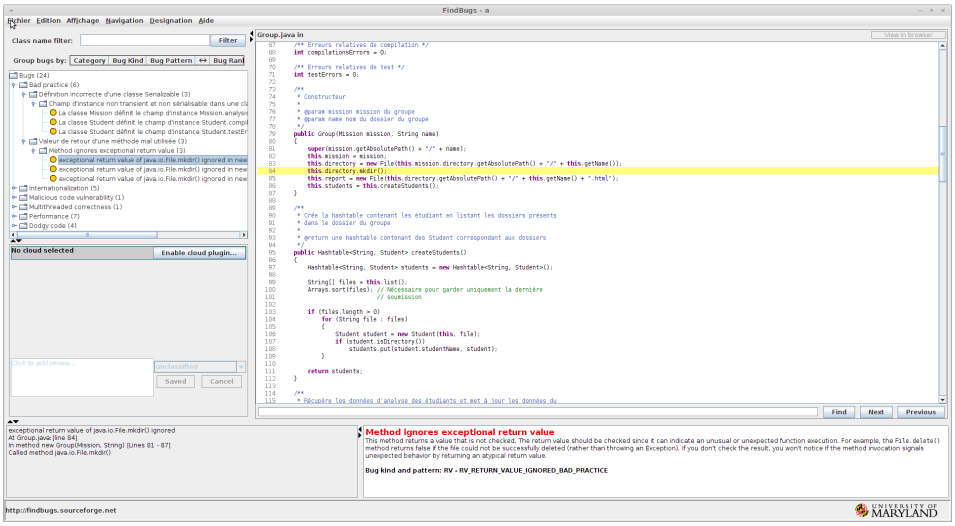


FIGURE 2.1 – Interface graphique de FindBugs

2.1.5 JavaPathFinder (JPF)

JPF[5] est un logiciel open source sous licence NASA Open source Agreement 1.3 et développé par le NASA Ames Research Center[4]. La dernière version, la 6ème, date du 30 novembre 2010.

C'est un autre outil d'analyse de bytecode Java mais, au lieu de simplement lire le code, il l'exécute de toutes les façons possibles afin de détecter d'éventuelles erreurs. JPF fonctionne en fait comme une machine virtuelle Java qui va parcourir tous les états potentiels du programme à tester afin de faire apparaître le cas échéant les exceptions non gérées, les situations de deadlock ou encore les situations de compétition. C'est avant tout un outil pour tester les programmes concurrents qui utilise le model checking et les principes des machines à état.

L'outil est hautement paramétrable par l'utilisateur et s'exécute au moyen d'une ligne de commande mais peut aussi être appelé depuis une autre application Java.

2.2 Frameworks et environnements de test

2.2.1 JUnit

JUnit[6] est un projet open source hébergé sur GitHub qui se présente sous forme d'une simple librairie. Le succès de JUnit fait qu'il est généralement inclus avec de nombreux IDE.

Framework de test par excellence, JUnit permet la création de tests unitaires à appliquer à un programme pour vérifier sa correction. Pour cela, JUnit utilise des assertions, qui comparent un résultat attendu à un résultat obtenu et vérifient

4. `http://www.nasa.gov/centers/ames/home/index.html`

si ceux-ci sont équivalents. La version 4.12, dernière en date, a modifié la manière d'écriture des tests et fait maintenant grand usage des annotations dont certaines en particuliers, les `@Rule`, pourront se révéler utiles dans notre problème :

ErrorCollector
Permet de continuer l'exécution d'un test même après une erreur d'assertion

TestName
Permet d'accéder au nom du test actuel

Timeout
Permet de définir une limite supérieure au temps d'exécution d'un test

2.2.2 TestNG

TestNG[7, 8] est basé sur JUnit 3 et est une alternative à celui-ci. Il est disponible sous forme d'exécutable et sous forme de plugin pour divers IDE. Il peut aussi être lancé par une tâche Ant et produire des rapports de test.

Il introduit de nouveaux concepts tels que l'utilisation de paramètres et de données pour les tests, les annotations, le multithreading, etc. Néanmoins une grande majorité de ses fonctionnalités ont été intégrées dans JUnit 4 quelques mois après sa sortie, ce qui a un peu éclipsé ce framework.

2.2.3 Maveryx

Maveryx[9] est un environnement d'automatisation de tests qui permet de pratiquer ceux-ci à l'échelle de toute une application. Bien qu'open source et sous licence GPL, ce logiciel est édité et maintenu par une société du même nom qui propose aussi une version plus évolué mais payante.

Cet outil est destiné principalement aux applications faisant usage d'interfaces graphiques et permet de tester ces dernières en profondeur. La dernière version, 1.4, est disponible sous forme d'exécutable ou de plugin pour Eclipse.

2.2.4 Jtest

Jtest[10] est une solution complète de tests permettant aussi bien de faire de l'analyse statique de code que des tests unitaires ou de la détection d'erreur d'exécution. Il s'agit vraiment d'un outil clé en main destiné aux entreprises qui garanti un suivi constant du développement de leurs programmes. Jtest fait notamment appel à JUnit pour les tests unitaires et fourni une mesure de la couverture de code de ceux-ci.

Développé par la société Parasoft, ce logiciel propriétaire est compatible avec nombre d'IDE et les quatre principaux OS.

2.3 Programmation interactive

2.3.1 CodeLab

CodeLab[11] est une plate-forme d'exercices de programmation interactive accessible sur le Web. Développé par la société Turingscraft depuis 2002, ce système pédagogique a été et est toujours utilisé par plusieurs centaines d'institutions et des milliers d'étudiants l'on employé. Il propose des exercices dans divers langages tels que Python, Java, C++, C et d'autres encore. C'est un logiciel propriétaire qui offre aux enseignants des solutions d'apprentissage personnalisées à destinations de leurs étudiants qui sont encouragés à s'en servir activement pour leur enseignement.

Une des vitrines les plus connues de cet outil est *myprogramminglab*, anciennement *mycodemate* (fermé en 2012), la plate-forme d'apprentissage de l'éditeur Pearson, maison d'édition majeure. Un code d'accès de 12 mois à cet outil est fourni dans de nombreux livres qu'elle propose.

Les exercices, triés par chapitre du livre auquel ils se rapportent, sont présentés sous la forme de questions ou de petits problèmes que l'étudiant doit résoudre. Le cas échéant, une zone de texte où insérer du code est disponible pour fournir une solution. Après la soumission, le code est corrigé et l'étudiant reçoit des indications sur les erreurs commises, comme par exemple une erreur de compilation ou une erreur de résultat, une faute de frappe, une faute de syntaxe...

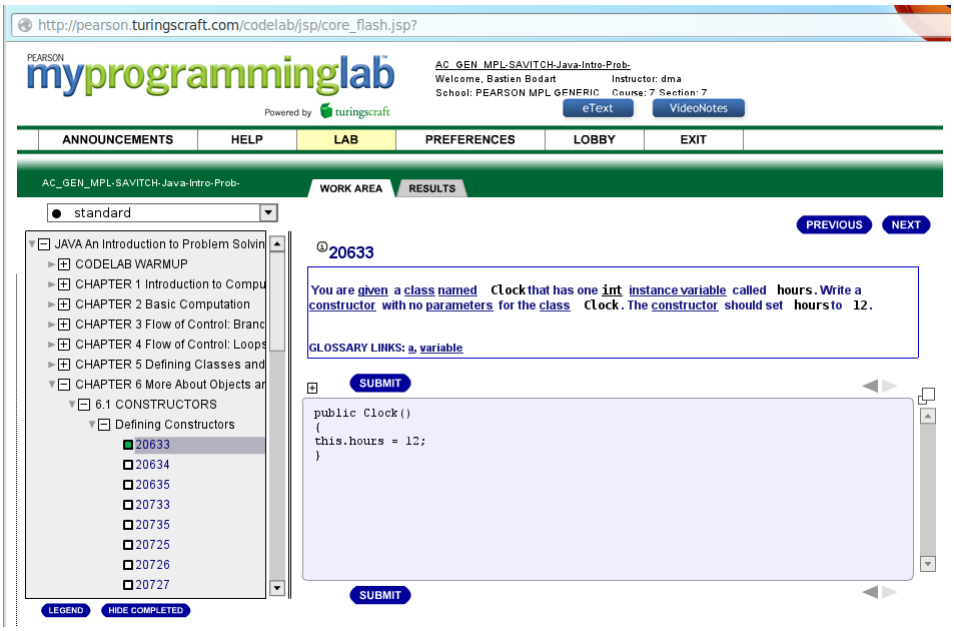


FIGURE 2.2 – Interface de myprogramminglab

2.3.2 Pythia

Pythia[12, 13] est une plate-forme d'apprentissage de programmation en ligne développée par le pôle d'ingénierie informatique de l'Université catholique de Louvain, toujours en version d'évaluation.

Basée sur le même principe que CodeLab, elle fourni une série d'exercices aux étudiants sous forme de listes de tâches à accomplir. Celles-ci comprennent généralement un contexte, une méthode à implémenter et une zone dans laquelle l'étudiant doit écrire son code. Ce code est ensuite copié dans une classe et exécuté dans une sandbox. Le résultat et un feedback sont ensuite présentés à l'étudiant.

2.3.3 Code school, code academy et Hurricane Electric courses

Bien que ne proposant pas d'apprentissage du langage Java car plutôt centrés sur l'apprentissage des langages et techniques du Web, ces sites[14, 15, 16] méritent d'être soulignés.

Ils proposent en effet des cours et exercices de programmation sur des langages tels que Ruby et Python. Ceux-ci ayant l'avantage d'être interprétés, ces sites proposent une série d'exercices simples tels que de la manipulation de `String` ou autres concepts de base de la programmation que l'élève peut exécuter dans une console. Des demandes ont été émises du côté des utilisateurs pour des exercices en Java, pour le moment restées sans réponse.

2.4 Bilan des outils disponibles

PMD et Checkstyle possèdent globalement les mêmes fonctionnalités, effectuent plus ou moins le même genre d'analyse et peuvent s'utiliser en combinaison avec Xlint. JPF et FindBugs sont des outils plus pointus, travaillant directement sur le bytecode, l'un avec exécution et l'autre sans, et permettant la recherche de bugs moins évidents. Ils ne seront sans doute pas nécessaires dans notre analyse.

JUnit et TestNG sont eux aussi fort semblables et proposent à peu de choses près les mêmes services, JUnit jouit cependant d'une plus grande notoriété et d'un meilleur développement, ce qui lui confère un avantage certain. Maveryx et Jtest sont des solutions à utiliser sur une plus vaste échelle telle qu'une application entière voire plus. Bien que très performante, il sera inutile de mettre en place une telle machinerie dans nos tests.

Les outils de programmation interactive n'auront pas non plus leur place dans l'analyse de code que nous envisageons. En revanche, il pourrait s'avérer intéressant de les intégrer directement dans la pédagogie d'enseignement, ce qui procurerait l'avantage de pouvoir fournir un retour direct à l'étudiant. CodeLab étant payant, il semble préférable d'utiliser Pythia pour autant que cette solution soit développée et maintenue.

Chapitre 3

MissionAnalyzer

Afin d'être en mesure de tester la correction du code soumis par les étudiants, c'est-à-dire le compiler et l'exécuter pour en vérifier les résultats, nous avons créé un outil spécialement dédié à cette tâche. En plus d'exécuter le code, celui-ci lancera éventuellement une analyse à l'aide de PMD et fournira des rapports qui permettront aux enseignants de tirer des conclusions sur la qualité du travail des étudiants.

Le logiciel sous forme de fichier `.jar` ainsi que les fichiers sources sont disponibles en accès libre sur GitHub[17].

3.1 Architecture

MissionAnalyzer a été écrit en utilisant le langage Java version SE 7, plus précisément à l'aide du JDK1.7.0_51. Ce choix a été fortement déterminé par le fait que le code à tester soit en Java également, ce qui va permettre d'utiliser certains outils et utilitaires prévus pour ce langage. L'ensemble du code du programme comporte environ 1800 lignes de code réparties en cinq classes. Les classes de test comportent en moyenne 200 lignes pour chaque mission.

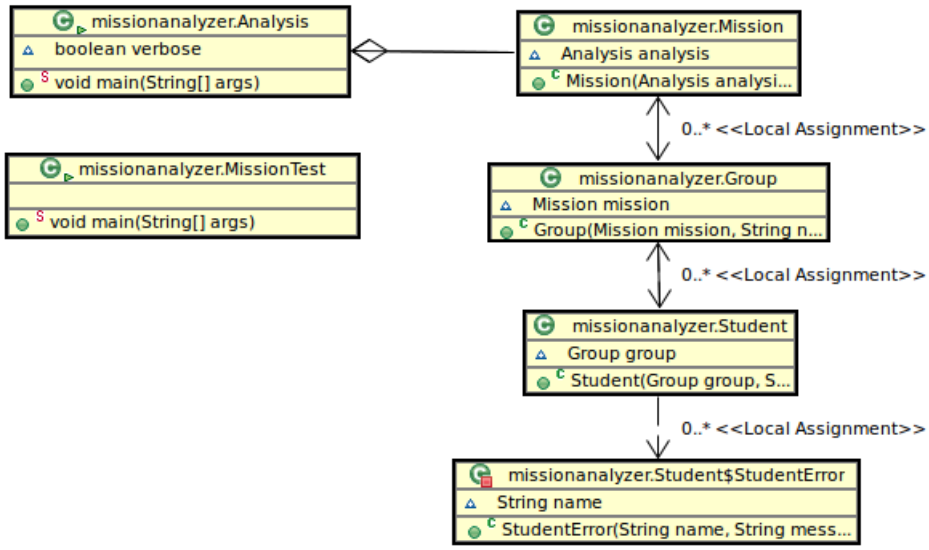


FIGURE 3.1 – Diagramme de classe

3.1.1 Classes

Toutes les classes font partie du package missionanalyzer. La figure 3.1 représente l'organisation de celles-ci sous forme de diagramme UML.

Analysis

La classe principale. Elle s'occupe de parser les paramètres reçus en arguments à l'aide de la librairie **commons-cli**. Elle crée ensuite un objet `Mission` et lance l'analyse sur celui-ci.

Mission

Cette classe représente une mission. Elle étend la classe `File` car elle représente aussi le dossier de la mission contenant les soumissions.

Elle crée un nouveau dossier "analysis" dans le dossier qu'elle représente. Celui-ci contiendra les rapports d'analyse. Elle crée ensuite les objets `Group` et lance un thread pour chaque étudiant de chaque groupe. Une fois tous les threads terminés, elle récupère les données auprès des objets `Group` et écrit les rapports généraux pour la mission aux formats HTML et JSON dans le dossier "analysis". Le format HTML a pour but une lecture directe des résultats dans un navigateur tandis que le format JSON est destiné à l'intégration de ceux-ci dans une éventuelle application externe utilisant Javascript.

Group

Cette classe représente un des groupes de travail dans lesquels les étudiants sont répartis en début d'année. Elle étend également la classe `File` car elle représente le dossier du groupe contenant les soumissions.

Elle crée un dossier au nom du groupe dans le dossier "analysis" puis crée les objets `Student` en prenant bien garde de n'en créer qu'un seul par étudiant, seule la soumission la plus récente étant conservée. Elle récupère les données des étudiants à la fin des analyses et écrit des rapports de groupe au format HTML et JSON.

Student

Cette classe représente un étudiant et étend la classe `File` également car elle représente aussi le dossier contenant les fichiers soumis par l'étudiant. De plus, elle implémente la classe `Runnable`. C'est en effet elle qui est appelée par chaque thread. Elle crée aussi un dossier au nom de l'étudiant dans le dossier d'analyse du groupe.

Elle effectue le travail d'analyse du code, la compilation et l'exécution des tests. Elle garde une trace des erreurs rencontrées et du résultat des différentes étapes. Elle écrit ses rapports à la fin de son exécution.

MissionTest

Cette classe n'est pas liée aux autres, c'est elle qui est exécutée par la classe `Student` avec le nom de la classe de test en argument. Elle lance les tests de la classe JUnit en question et récupère les éventuelles erreurs qu'elle imprime dans le flux d'erreur standard, seule façon de renvoyer les données collectées au programme comme nous le verrons plus loin.

3.1.2 Classes de test

Des classes de test sont fournies avec le logiciel. Ces classes étaient originellement prévues pour les missions de l'année 2012-2013, elles ont été remaniés lors de l'année 2013-2014. Tout changement dans les missions lors d'années ultérieures doit conduire à une vérification et une éventuelle réécriture de ces classes. Les pratiques à suivre pour l'écriture de ces classes sont développées à la section 3.5.2.

3.2 Objectifs

Comme signalé dans l'introduction, le but du logiciel d'analyse est double :

- permettre la correction des soumissions des étudiants de manière automatisée
- récupérer des données exploitables quant à la nature des erreurs rencontrées lors de cette correction

Les erreurs récupérées peuvent concerner divers domaines comme la compilation, les tests unitaires réalisés ou les avertissements générés lors de l'analyse du code source.

3.2.1 Sujet d'analyse

Les données analysées sont les soumissions des étudiants effectuées lors du premier quadrimestre 2012-2013. Il y a en tout onze missions, malheureusement pas toutes testables comme nous le verrons. Pour chacune des missions l'étant, nous allons effectuer une analyse grâce au logiciel et produire des rapports pour chaque groupe et chaque étudiant.

3.3 Principe de fonctionnement

Le logiciel a besoin de trois éléments externes pour fonctionner :

- Une dossier de mission contenant les données à analyser
- Un fichier de configuration propre à la mission à tester, à remplir par l'utilisateur. Celui-ci doit contenir le nom des fichiers obligatoires pour cette mission, c'est à dire ceux contenant le code de l'étudiant. Il doit aussi contenir une liste des noms des groupes à tester. Il est en effet possible de ne tester qu'un seul ou une partie des groupes de la mission. Il doit de plus contenir les éventuelles redirections à substituer aux adresses URL réelles des fichiers si l'exécution a lieu sur un serveur
- Une classe de test, elle aussi propre à la mission à tester, que l'utilisateur doit écrire. Idéalement, cette classe devrait contenir au moins un test par méthode à tester, le nom du test étant identique à celui de la méthode. Il est impératif que cette classe soit une classe de test JUnit 4

Le programme crée un dossier "analysis" dans le dossier de la mission. Celui-ci contiendra un dossier pour chaque groupe testé et dans ceux-ci un dossier pour chaque étudiant, leur arborescence sera identique à celle des données de la mission. Ces dossiers contiendront les rapports d'analyse ainsi que les éventuels fichiers contenant le flux des sorties standard et d'erreur.

Le fichier de configuration de mission est ensuite lu et l'analyse commence. Le logiciel parcourt le dossier de la mission selon l'algorithme de parcours en profondeur (depth first) et lance un nouveau thread d'analyse pour chaque étudiant rencontré, dans la limite d'un nombre maximal de thread fixé au préalable. Nous rappelons qu'en cas de multiples soumissions, seule la plus récente sera analysée. Chaque thread effectuera alors les opérations décrites ci-après.

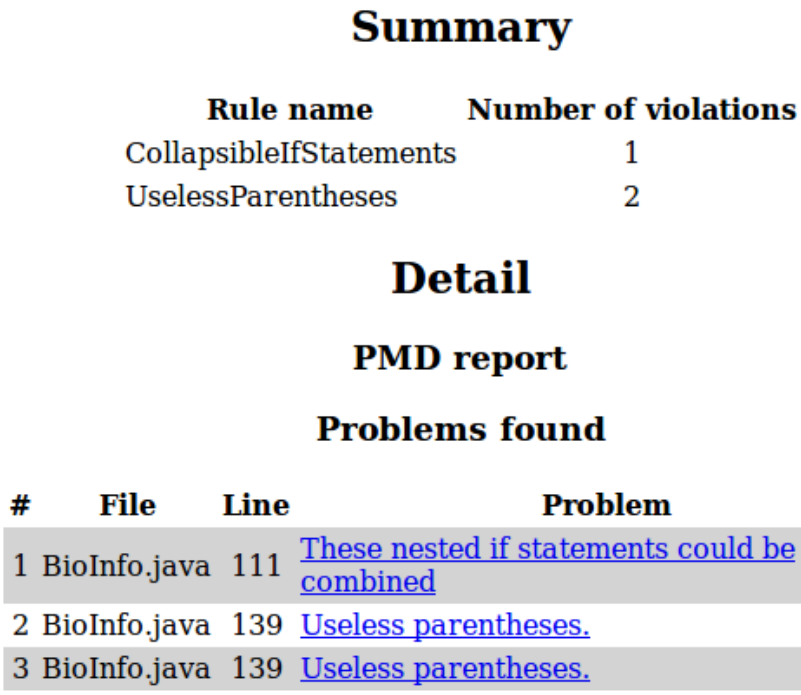
3.3.1 Analyse de code

Il est possible, si besoin est, d'effectuer une analyse du code source grâce à PMD [1]. Cette analyse sera effectué sur base des ensembles de règles définis au préalable par l'utilisateur ou à défaut sur l'ensemble **Basic** et portera sur tous les fichiers présents dans le dossier de l'étudiant. Deux rapports seront alors créés dans le dossier d'analyse de l'étudiant, un au format HTML, l'autre XML, contenant les violations aux règles rencontrées. Il faut noter que tous les fichiers présents dans le dossier seront analysés, même ceux fournis par les enseignants. Il serait dès lors sans doute judicieux de s'assurer que ceux-ci ne génèrent pas d'erreur lors de cette phase.

Cette opération est facultative et bien qu'instructive, ne nous paraît pas d'une importance critique car les règles qu'elle permet de vérifier concernent plutôt de bonnes pratiques à adopter et non la correction du code fourni et n'a de ce fait pas grand intérêt pour un examinateur cherchant à vérifier l'acquisition de concepts par un étudiant.

Néanmoins, un tel rapport peut être d'une grande utilité si il est fourni à l'étudiant lui-même avec la correction de son travail, surtout lors des premières missions, cela lui permettra de remarquer ses fautes et d'ainsi acquérir sur le long terme les bonnes pratiques en question. Il est en effet possible d'obtenir un rapport sous forme d'un simple fichier HTML qu'il serait facile de distribuer (voir Figure 3.2). Pour ce faire, nous recommandons principalement l'utilisation des ensembles **Basic**, **Design**, **Naming** et **Optimization**.

1. Voir section 2.1.2



Summary

Rule name	Number of violations
CollapsibleIfStatements	1
UselessParentheses	2

Detail

PMD report

Problems found

#	File	Line	Problem
1	BioInfo.java	111	These nested if statements could be combined
2	BioInfo.java	139	Useless parentheses.
3	BioInfo.java	139	Useless parentheses.

FIGURE 3.2 – Exemple de rapport d'analyse de code par PMD

3.3.2 Compilation

Dans le cas où les fichiers obligatoires se trouvent bien dans le dossier de l'étudiant, la compilation de ceux-ci sera tentée. Les erreurs éventuelles du compilateur sont récupérées et gardées en mémoire. Si la compilation est réussie, on lance la compilation de la classe de test. Encore une fois les erreurs du compilateur sont récupérées. En cas de réussite, le fichier `.class` est créé dans le dossier de l'étudiant. Il faut noter que la compilation des fichiers de l'étudiant se fait avec l'option `-Xlint:all` [2], ce qui aura pour effet de faire passer les warnings générés de cette façon pour des erreurs, qui seront écrites dans les rapports d'analyses, mais n'empêchera pas la compilation des fichiers.

3.3.3 Exécution

Si la compilation s'est déroulée sans erreur, on lance alors l'exécution de la classe de test dans un nouveau processus et le thread courant attend le retour du test. Les erreurs d'assertion sont elles récupérées grâce à un parser qui lit le flux d'erreur standard tandis que le flux de sortie standard éventuel est écrit dans un fichier dans le dossier de l'étudiant. Un gestionnaire d'exécution permet de vérifier si celle-ci s'est bien déroulée. Il se pourrait que des erreurs ne soient pas reconnues comme

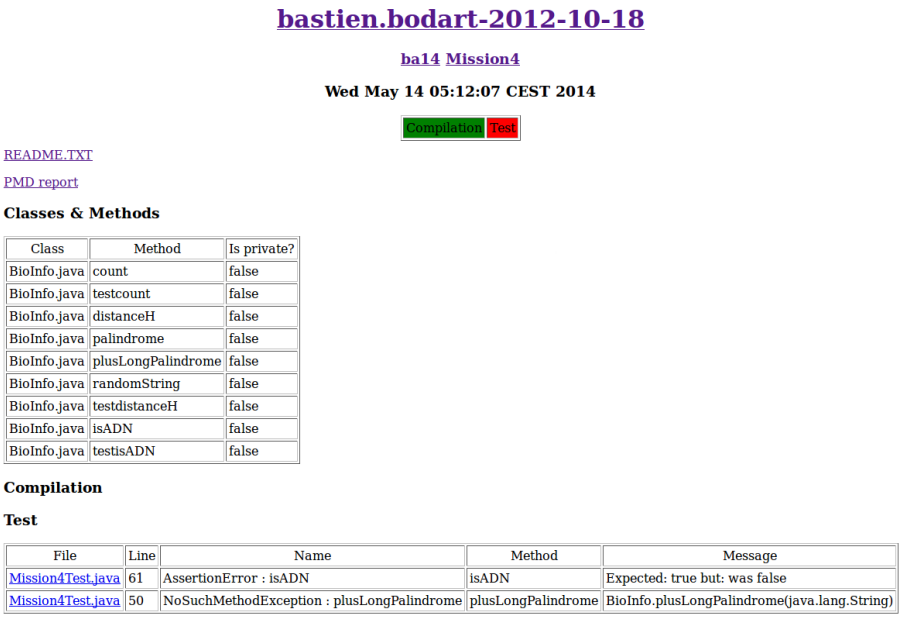
2. Voir section 2.1.1

étant des erreurs dues au test, elles seront alors écrites dans un fichier d'erreur. Ce cas de figure ne devrait cependant pas se produire dans une situation normale.

3.3.4 Rapports

Quels que soient les résultats des opérations précédentes, deux rapports d'analyse sont ensuite créés dans le dossier de l'étudiant. Chacun reprend la liste des fichiers manquants, la liste des classes et méthodes présentes, la liste des erreurs de compilation et la liste des erreurs de test (voir Figure 3.3). Une fois que l'analyse a été effectuée pour chaque étudiant, deux autre rapports sont créés pour chaque groupe ainsi que deux rapports globaux pour la mission. Ceux-ci reprennent également les listes des erreurs rencontrées avec une mesure de leurs occurrences absolues et de leur occurrences relatives, c'est à dire du nombre d'étudiants ayant commis ce type d'erreur.

Pour autant qu'il soit présent dans le dossier du fichier `.jar`, un fichier Javascript `sorttable.js` [3] sera copié dans les dossiers contenant les rapports des groupes et des missions. Celui-ci permet le tri des entrées dans les tableaux d'erreurs.



bastien.bodart-2012-10-18

ba14 Mission4

Wed May 14 05:12:07 CEST 2014

Compilation Test

README.TXT

PMD report

Classes & Methods

Class	Method	Is private?
BioInfo.java	count	false
BioInfo.java	testcount	false
BioInfo.java	distanceH	false
BioInfo.java	palindrome	false
BioInfo.java	plusLongPalindrome	false
BioInfo.java	randomString	false
BioInfo.java	testdistanceH	false
BioInfo.java	isADN	false
BioInfo.java	testisADN	false

Compilation

Test

File	Line	Name	Method	Message
Mission4Test.java	61	AssertionError : isADN	isADN	Expected: true but: was false
Mission4Test.java	50	NoSuchMethodException : plusLongPalindrome	plusLongPalindrome	BioInfo.plusLongPalindrome(java.lang.String)

FIGURE 3.3 – Exemple de rapport d'analyse HTML d'un étudiant

3. `http://www.kryogenix.org/code/browser/sorttable/`

3.4 Développement

3.4.1 Choix d'implémentation

Librairies

Afin de profiter des facilités qu'elles procurent ou des fonctionnalités nécessaires qu'elles implémentent, les librairies suivantes sont utilisées par le programme:

commons-cli
: Permet un traitement efficace et rapide des arguments passés en ligne de commande

commons-exec
: Fournit un cadre pour l'exécution de processus externes avec un gestionnaire d'erreur et un watchdog qui permet l'arrêt du processus au bout d'un certain temps

commons-lang
: Fournit des outils supplémentaires pour la manipulation des classes de bases de Java

junit
: Framework pour les classes de test

hamcrest-all
: Fournit des matchers qui seront utilisés lors des tests

gagawa
: Fournit des outils pour la création de fichiers HTML au moyen d'objets

gson
: Fournit des outils pour la création de fichiers JSON

pmd
: Fournit les outils pour lancer l'analyse PMD dans l'application

Les références de ces librairies se trouvent en annexe. De plus, PMD lui même a besoin de bien plus de librairies pour fonctionner, la liste et références de celles-ci se trouvent en annexe également.

Analyseur de code

Le choix d'un analyseur de code s'est porté sur PMD, décrit en section 2.1.2. On peut justifier ce choix par plusieurs critères :

- La génération de rapport est automatique et peut se faire sous format HTML et XML
- L'analyse peut se faire directement depuis l'application au moyen d'une librairie. Ce qui permet de faire gagner pas mal de temps car il n'est pas nécessaire de lancer un nouveau processus

– Le large choix de règles fournies et la possibilité d'en écrire de nouvelles soi-même

Checkstyle aurait pu servir lors de cette analyse mais son orientation trop portée sur les conventions nous a fait préférer PMD.

Compilateur

Deux options étaient disponibles pour la compilation des fichiers : soit lancer un nouveau processus en se servant de la commande *javac*, soit utiliser le compilateur directement au sein de l'application grâce au `ToolProvider`. Ce dernier permet d'accéder au compilateur Java présent dans le système pas le biais de l'interface `JavaCompiler`. Il est alors possible, moyennant l'utilisation d'un file manager et la création de tâches, d'effectuer la compilation de fichiers `.java` sans lancer de nouveaux processus. Cette dernière option est de loin la plus avantageuse pour deux raisons principales.

Primo, la création d'une nouveau processus est une tâche qui demande plus de temps et de ressources qu'un appel au sein d'une application. Sans compter que la compilation elle-même prend plus de temps également. Nous avons constaté que, toutes choses étant égales par ailleurs, l'utilisation de l'interface `JavaCompiler` divise par quatre le temps nécessaire à la compilation.

Secundo, cette interface permet l'utilisation d'un `DiagnosticCollector`, qui va nous permettre de récupérer les erreurs générées par le compilateur bien plus facilement que par la lecture d'un flux.

Exécuteur

Le logique voudrait que l'exécution de la classe de test se face dans l'application principale mais malheureusement, cela ne peut pas être le cas. En effet, le test du code de l'étudiant, qui n'a pas été vérifié préalablement, peut mener à des situations de boucles infinies qui auraient pour effet de bloquer le logiciel. Dès lors, il s'impose de lancer les tests soit dans un autre thread soit dans un processus et d'y adjoindre un timer afin de pouvoir l'interrompre au bout d'un temps prédéterminé si l'exécution ne s'est pas terminée.

Malheureusement, la solution du thread n'est pas envisageable car la façon dont Java gère l'arrêt de ceux-ci ne permet pas d'en récupérer un qui exécute une boucle infinie. En effet, celle-ci est basée sur la méthode `interrupt`, qui lève une exception d'interruption mais ne stoppe pas l'exécution. La méthode dépréciée `stop` quant à elle n'arrête pas non plus un thread en état de boucle infinie car celui-ci n'est tout simplement pas disponible pour gérer son propre arrêt. Nous pouvons donc dire qu'il n'existe pas de moyen propre et efficace pour stopper un thread sans la coopération de celui-ci.

Reste la solution du processus, lancé avec l'aide de la libraire **commons-exec** qui elle-même utilise la classe `Runtime`. Grâce à cette librairie, nous pouvons

invoquer un objet `Watchdog` qui va veiller à ce que le processus ne dépasse pas un certain temps d'exécution, faute de quoi il sera arrêté grâce à la méthode `destroy`. Une gestionnaire de flux permet la récupération des erreurs sur la sortie d'erreur standard et un gestionnaire d'exécution pourra nous dire si le processus s'est bien terminé ou non.

Sécurité

Il est important de noter que l'exécution du code de l'étudiant se fait sans aucune vérification de sécurité. Il est peu probable que celui-ci soit malveillant volontairement mais il se pourrait qu'il le soit par inadvertance ou une erreur de l'étudiant. Bien que nous n'ayons pas utilisé cette option dans nos tests, il pourrait être préférable, dans une optique de totale sécurité d'utiliser le `SecurityManager` Java [4] avec un fichier `policy` défini par l'utilisateur du logiciel en fonction de ses craintes pour la sécurité du système.

Multithreading et performance

Il nous est vite apparu que l'utilisation de plusieurs threads pour traiter autant de données pourrait se révéler fort profitable.

Le programme utilise par défaut quatre threads principaux pour traiter les analyses des étudiants. Après différents tests de performance, effectués sur un processeur Intel Core i5 3570k, les données récoltées et présentées dans la figure 3.4 permettent de dire qu'il s'agit du nombre de threads le plus plus raisonnable fournissant une bonne performance, tout du moins sur une processeur quadri-coeur. Il semble d'ailleurs logique qu'un tel processeur produise un meilleur résultat avec un tel nombre.

4. `http://docs.oracle.com/javase/tutorial/security/tour2/step2.html`

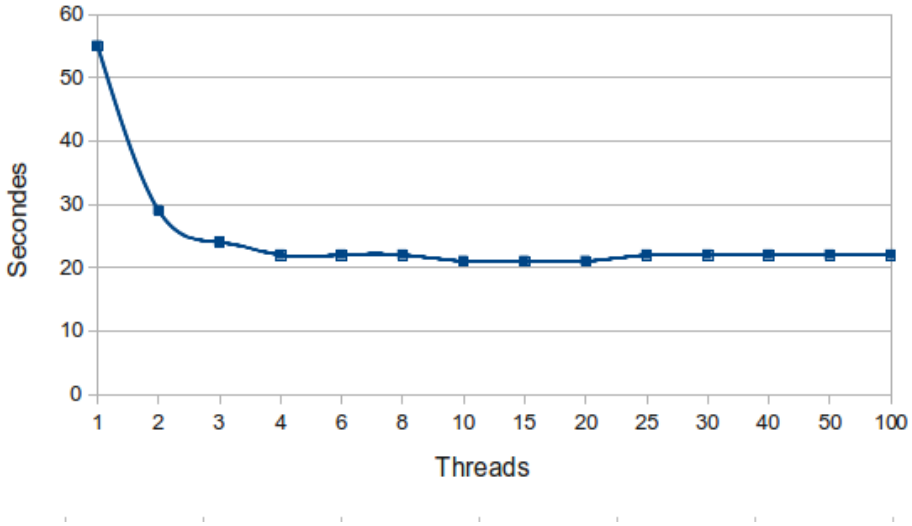


FIGURE 3.4 – Mesure du temps d'exécution du logiciel sur la mission 4 en fonction du nombre de threads

Lors de la phase de production du logiciel durant le premier quadrimestre 2013-2014, l'exécution n'a jamais pris plus de quelques minutes. Celle-ci s'effectuait sur un serveur dont les ressources étaient variables.

Collecte des résultats

La collecte de résultat se fait en deux temps.

Premièrement lors de l'analyse que chaque thread effectue pour chaque étudiant. Lors de celle-ci, les erreurs de compilation et d'exécution sont enregistrées sous forme d'objet `StudentError`, une classe interne à `Student`.

Deuxièmement, après l'exécution de tous les threads, quand les groupes parcourent tous leurs étudiants pour récolter leurs erreurs et ensuite quand la mission parcourt tous les groupes pour faire de même.

La raison pour laquelle la collecte des données des groupes se fait à la fin de tous les threads peut s'expliquer par le fait qu'une autre façon de faire n'aurait pas donné des résultats optimaux. En effet, on aurait pu faire en sorte que chaque étudiant transmette ses données à son groupe à la fin de son analyse, mais la concurrence des threads fait que l'on aurait du opérer une synchronisation de ceux-ci, ce qui dégraderait la performance. Il aurait aussi été possible de récolter les données une fois que tous les threads d'un groupe aient terminé, mais cela aurait dans ce cas provoqué un temps d'attente important si un thread ne terminait pas, empêchant le démarrage des threads du groupe suivant.

Au final, récupérer les données après toutes les analyses n'est pas si handicapant, il s'agit juste de parcourir une deuxième fois l'arbre des groupes et des étudiants.

Rapports

Le premier format des rapports est le HTML. C'est en effet un format pratique, portable et facile à lire.

L'avantage de ce format est aussi qu'il est possible d'établir des liens entre les rapports. Dès lors, le rapport de mission contient des liens vers les rapports des groupes et des étudiants, ceux des groupes vers le rapport de mission et ceux des étudiants et ceux des étudiants vers le rapport de la mission et de leur groupe. Ce dernier contient aussi des liens vers les fichiers de sortie ou d'erreur éventuels, vers le rapport PMD et vers le dossier de l'étudiant.

L'autre format est le JSON (JavaScript Object Notation) qui est un format de données facilement exploitable pour une application Web écrite en JavaScript. [5]

Les rapports de mission contiennent les pourcentages de réussite des étudiants et les erreurs qu'ils ont commises. Ils contiennent aussi les pourcentages de réussite au sein des groupes, ce qui permet de remarquer ceux en difficulté et qui auraient besoin d'une aide pédagogique supplémentaire.

Encapsulation

Le programme n'utilise absolument pas les principes de l'encapsulation, tous les champs et toutes les méthodes étant publics. Seule `StudentError`, une classe interne, est privée. Ce choix est motivé par plusieurs raisons telles que la lourdeur d'utilisation des accesseurs et le fait que ce logiciel n'a pas pour vocation d'être intensément modifié dans le futur ni d'être utilisé par un grand nombre de personnes.

3.4.2 Problèmes rencontrés

Deux problèmes sérieux se sont présentés lors de la réalisation de ce logiciel. Ces problèmes sont dûs à ce qui semble être des bugs de Java et de PMD. Un des deux problèmes ne peut pas être résolu à ce jour mais peut être contourné. Un troisième concernait l'encodage des fichiers à compiler mais a pu être résolu tandis qu'un quatrième concernant l'utilisation de PMD avec *cron* est toujours présent.

Process.destroy()

Ce problème semble être un bug de Java. Il apparait que la méthode `destroy`, qui a pour fonction de tuer un processus, ne soit pas totalement fiable lorsqu'il s'agit de tuer un processus juste après l'avoir créé si ce processus est en situation de boucle infinie et a été lancé par un thread.

5. Une telle application a été développée et mise en production lors de l'année 2013-2014 par Thibault Sottiaux (thibault.sottiaux@student.uclouvain.be)

Voici un simple code permettant de recréer le problème:

```
for(int i = 0; i < 100; i++){
  new Thread(new Runnable(){
    public void run(){
      try{
         Process p = Runtime.getRuntime()
           .exec(new String[]{"java", "InfiniteLoop"});
         p.destroy();
      }
      catch(IOException | InterruptedException e){
         e.printStackTrace();
      }
    }
  }).start();
}
```

Comme on peut le voir, ce code crée cent threads et chacun lance dans un processus un programme Java appelé `InfiniteLoop`. Celui-ci est juste, comme son nom l'indique, une boucle infinie. Ensuite le processus est détruit aussitôt.

Il s'avère que parfois un nombre aléatoire de processus ne sont pas détruits et continuent leur exécution de `InfiniteLoop` après la terminaison du programme principal. Ceci constituant une violation des spécifications de la méthode `destroy`, un bug a été soumis à Oracle [6] et s'est vu attribuer le numéro 9005842, mais est toujours sans réponse à l'heure actuelle. Après différents tests, on peut remarquer que le nombre de processus survivants diminue si un temps d'attente est introduit entre la création et la destruction. Plus grand ce temps, plus rares les processus survivants. Il existe aussi une relation avec le nombre de threads ; plus ce nombre est grand, plus il y aura de processus survivants.

Ce problème peut survenir dans notre programme quand le timer du `Watchdog` pour l'exécution est trop court et que le nombre de threads utilisé est trop élevé. C'est pourquoi il est recommandé de ne pas fixer le timer en-dessous d'une valeur de 5000 millisecondes.

PMD.run()

Le deuxième problème sérieux provenait de PMD.

Si on s'en réfère à l'API de ce programme [7], il est possible de l'appeler en utilisant la méthode statique `run`. Cependant, un appel simultané de cette méthode par deux threads concurrents causait une levée d'exception. Nous en avons donc conclu que cette méthode statique ne devait sans doute pas l'être entièrement et

6. `http://bugreport.java.com/bugreport`
7. `http://pmd.sourceforge.net/pmd-5.1.1/apidocs/net/sourceforge/pmd/PMD.html`

qu'une ressource partagée était la cause de cette erreur. Un bug a été soumis à l'équipe de PMD.

Ce problème a d'abord été contourné par la synchronisation des threads lors de l'exécution de cette méthode, au détriment de la performance. La version 5.1 du logiciel a corrigé le problème, qui était bien du à une ressource partagée. [8]

Encodage des fichiers

La compilation était parfois impossible à cause de l'encodage des fichiers qui était différent de l'UTF-8. Le problème a été résolu, le programme prenant maintenant en compte l'encodage du fichier à compiler en utilisant l'option `-encoding` de *javac*.

PMD et cron

Lors de la mise en production, il est apparu que PMD ne fonctionnait pas correctement lors d'un lancement du programme avec *cron* bien que tout fonctionnait parfaitement sans ce dernier. En effet, le logiciel bloquait si un fichier Java du dossier à analyser commençait par une minuscule. La seule parade trouvée à ce jour a été d'annuler l'analyse par PMD pour ces dossiers.

3.5 Classes de test

3.5.1 Framework

L'utilisation de JUnit 4 comme framework de test nous a paru comme la plus cohérente et la plus efficace.

Cela permet tout d'abord l'utilisation de la méthode `runClasses` pour lancer tous les tests d'un seul coup et la récupération des erreurs produites dans le résultat obtenu par cet appel.

L'annotation `@Before` est aussi très utile afin d'instancier les différents objets dont les tests ont besoin. Néanmoins l'annotation `@Timeout`, elle, ne sera pas employé car encore une fois, celle-ci fait appel à `interrupt`, qui comme nous l'avons déjà mentionné, ne permet pas de récupérer les threads coincés dans des boucles infinies.

Pour être certain de l'utilisation de JUnit 4, il faut s'assurer que la classe possède bien l'annotation `@RunWith(JUnit4.class)`.

3.5.2 Recommandations

Réflexion

Toute référence à un fichier d'étudiant dans la classe de test peut générer une erreur de compilation quand ce fichier n'existe pas, ne contient pas les bonnes mé-

8. `http://sourceforge.net/p/pmd/bugs/1124/`

thodes ou autre. C'est pourquoi il est préférable d'utiliser la réflexion pour invoquer les méthodes à tester. Avec cette façon de procéder, la classe de test sera compilée à tous les coups. Il sera de plus possible d'outrepasser un éventuel modificateur private sur une méthode.

Présence de méthode

Il faudrait idéalement toujours tester la présence des méthodes requises ainsi que les types et nombres des paramètres et le type éventuel du retour. La réflexion permet de faire ce test, qui lèvera une exception en cas d'absence de méthode adéquate.

ErrorCollector

Idéalement, il faut utiliser un `ErrorCollector`, qui va collecter les erreurs d'assertion au fur et à mesure de l'avancée des tests et permettre de continuer ceux-ci sans être arrêté à la première erreur. L'utilisation de ce collecteur implique l'utilisation de matchers de la libraire **hamcrest** avec la méthode `checkThat`.

Mesure de temps ou analyse de complexité

Une mesure de temps peut être effectuée à n'importe quel endroit en utilisant la méthode `getCurrentThreadCpuTime` sur un objet `ThreadMXBean`. Il est possible, après avoir fait ces calculs dans la classe de test, de récolter les informations en les imprimant sur la sortie standard. Il suffit de faire précéder une première ligne contenant un label ou un nom de méthode par le caractère "$" et de la faire suivre par une deuxième qui contient une donnée intéressante comme le temps d'exécution ou la complexité de l'algorithme. Ces données seront ensuite intégrées aux rapports.

Rappelons que la mesure de complexité ne peut se faire que par approximation suivant le temps d'exécution, le calcul précis étant impossible sans analyse du code.

Tests et sous-tests

Dans notre cas, l'analyse doit aller plus loin que de simples tests unitaires. C'est pourquoi il est utile de séparer les tests d'une méthode en différents sous-tests qui permettront de vérifier la présence d'erreur dans un cas précis comme une chaîne vide en paramètre ou des données importantes dans la première ou dernière case d'un tableau. Un tel sous-test ne doit être pris en compte que si le test de la méthode principale est réussi et il faut bien sûr éviter un maximum que les cas testés par plusieurs sous-tests soient redondants ou empiètent les uns sur les autres.

Le logiciel permet de créer de tels sous-tests, en nommant une méthode de test `test_sousTest` où `test` est le nom d'une méthode de test existante tandis que `sousTest` est un nom quelconque. Le résultat de cette méthode de sous-test ne sera pris en compte que si le test principal a réussi et ne sera pas comptabilisé

comme erreur dans le résultat de celui-ci. Cela permet de raffiner le type d'erreur que l'on peut détecter d'une manière plus simple que le passage en revue des messages d'erreur.

Pour utiliser cette fonctionnalité, il faut faire précéder la classe de test de l'annotation `@FixMethodOrder(MethodSorters.NAME_ASCENDING)` afin d'exécuter les tests dans l'ordre alphabétique.

Exceptions et erreurs

Étant donné qu'il nous est impossible de prévoir comment va se passer l'exécution du code d'un étudiant, toute méthode de test devrait être définie comme lançant un objet Throwable afin que celui-ci soit capté par le logiciel et écrit dans le fichier d'erreur de l'étudiant.

3.6 Bilan de l'outil

Avec ce logiciel, nous serons donc en mesure d'obtenir de manière automatique, sous forme de rapports pouvant aisément être parcourus, de précieuses informations sur le travail des étudiants, que ce soit leur capacité à écrire du code fonctionnel, correct et de qualité ou le respect des consignes qui leurs ont étés attribuées.

Notons que bien que ce logiciel soit utilisé dans le cadre d'un cours d'informatique de base, il peut être utilisé pour tester n'importe quel code Java pourvu qu'on se donne la peine d'écrire les classes de test adéquates et que l'on respecte l'arborescence dans les dossiers et fichiers à tester.

Chapitre 4

Analyse des missions

Comme mentionné dans l'introduction, le cours LSINF1101 est divisé tout au long de l'année en plusieurs missions[18] qui couvrent chacune un aspect précis de la programmation, généralement au rythme d'une par semaine. Répartis en groupes, les étudiants sont invités, après le cours magistral expliquant la matière et une séance d'exercices, à fournir des classes Java répondant aux spécifications fournies. Pour l'année 2012-2013, 444 étudiants étaient renseignés comme inscrits au cours. Cependant le nombre de soumissions est largement inférieur, les étudiants étant autorisés à travailler à plusieurs.

Pour chaque mission, nous allons tenter de dégager les concepts-clés qu'elle tente d'enseigner, nous passerons ensuite en revue les consignes fournies pour déterminer quel travail l'étudiant doit fournir et ainsi pouvoir mettre en place des tests pertinents.

Nous étudierons ensuite les résultats de l'analyse concernant la compilation et plus particulièrement les tests. Pour ce faire, l'outil d'analyse nous fournira l'ensemble des erreurs rencontrées avec leurs nombres d'occurrences absolues et relatives. En ce qui concerne les erreurs de compilation, la plupart sont dues à des erreurs de syntaxe et ne sont malheureusement pas remarquées par les étudiants. Ce genre d'erreur ne porte pas à conséquence et il se pourrait donc que nous les passions sous silence.

De même nous ne nous attarderons pas sur le résultat des temps d'exécution, les étudiants ayant déjà pas mal de difficultés à produire du code correct, vérifier en plus qu'il soit optimal n'est pas vraiment révélateur des leurs capacités.

Précision sur les données présentées dans les tableaux :

- *Absolue* représente le nombre d'occurrences absolues d'une erreur, c'est à dire le nombre de fois où cette erreur est apparue
- *Relative* représente le nombre d'étudiants ayant commis cette erreur
- *Proportion* représente le pourcentage du total d'étudiants ayant commis cette erreur. Ce total est celui du nombre d'étudiants de la mission quand il s'agit d'une erreur de compilation. Mais il s'agit du total d'étudiants ayant réussi la compilation quand il s'agit d'une erreur de test

4.1 Missions 1 et 2

La première des missions porte sur la familiarisation avec le langage Java et une première approche des variables, des boucles et des méthodes d'entrée/sortie. Il est demandé aux étudiants de compléter la méthode `main` d'une classe pour que celle-ci affiche les valeurs de 1 à 10.

La deuxième mission quant à elle continue dans la même voie. Il est cette fois demandé de compléter la méthode `main` afin de calculer les solutions d'équations diophantiennes paramétrées par des entrées au clavier.

Ces deux missions fournissant uniquement un résultat sur la sortie standard et demandant une interaction de l'utilisateur ne sont pas très faciles à tester automatiquement avec des tests unitaires. De plus les concepts couverts sont des plus basiques. C'est pourquoi nous commençons notre analyse approfondie des erreurs par la mission 3.

La seule analyse réalisable pour les missions une et deux serait une analyse du code grâce à PMD qui permettrait, comme dit plus haut, de vérifier que l'étudiant met en œuvre de bonnes pratiques de programmation. Cette analyse, ainsi que les erreurs du compilateur, devrait permettre à l'enseignant de facilement identifier les problèmes rencontrés lors de l'écriture même du code.

4.2 Mission 3 : Méthodes

4.2.1 Concepts clés

Dans cette mission, les étudiants sont amenés à créer des méthodes qui renvoient des résultats simples en fonction des arguments. Ces méthodes effectuent de simples opérations arithmétiques. Le but recherché est donc clairement d'initier l'étudiant au concept des méthodes et de la division en sous-problèmes, des types de variables et des opérations arithmétiques de base.

4.2.2 Consignes

Il est demandé à l'étudiant de fournir une classe `LibMath.java` après avoir implémenté plusieurs méthodes :

average
: Prend trois réels en argument et renvoie la moyenne de ceux-ci

maximum
: Prend trois réels en argument et renvoie la plus grande valeur parmi ceux-ci

minimum
: Prend trois réels en argument et renvoie la plus petite valeur parmi ceux-ci

median
: Prend trois réels en argument et renvoie la valeur médiane de ceux-ci

Une dernière méthode, non nommée, leur est demandée, celle-ci doit vérifier si un nombre entier positif donné par l'utilisateur est sublime. La classe `LibMath` contient des méthodes de test fournies pour chacune des méthodes leur permettant de vérifier leur bon fonctionnement.

4.2.3 Tests

Les tests vont donc porter sur les quatre premières méthodes, la dernière, sans nom défini et demandant une interaction n'étant pas testable automatiquement.

Pour chacune des méthodes, nous testerons diverses combinaisons de paramètres positifs, négatifs ou nuls. Nous testerons aussi avec des combinaisons où deux ou trois paramètres sont égaux dans un sous-test pour chaque méthode.

Il s'agit également de vérifier si les méthodes existent bel et bien et que les types des paramètres et du retour sont corrects (`double`).

4.2.4 Résultats et interprétation

Voici les résultats globaux de la mission :

Groupes	Étudiants	Compilation réussie	Test réussi
19	220	184 (83,6%)	92 (41,8%)

On remarque que parmi les étudiants ayant réussi la compilation, seulement la moitié d'entre eux ont réussi le test de leurs méthodes.

Compilation

Voici une liste des principales erreurs rencontrées :

	Absolue	Relative	Proportion
cannot find symbol	436	25	11,36%
missing return statement	3	3	1,36%
method minimum in class LibMath cannot be applied to given types	10	2	0,91%
method maximum in class LibMath cannot be applied to given types	5	1	0,45%
method average in class LibMath cannot be applied to given types	5	1	0,45%

– *cannot find symbol* représente une bonne part des erreurs commises par les étudiants. Celle-ci dénote généralement qu'au moins une des méthodes n'a pas été créée alors qu'elle est appelée dans la classe de l'étudiant

– *missing return statement* indique que trois étudiants n'ont pas compris, ou oublié, que les méthodes doivent renvoyer un résultat. Toutes les méthodes doivent ici renvoyer un `double`
– *cannot be applied to given types* dénote dans ce cas ci que les méthodes créées n'emploient pas les bons types ou le bon nombre de paramètres. Toutes les méthodes doivent ici prendre trois `double` en arguments

Test

Nous allons séparer les erreurs d'exécution des erreurs d'assertion pour plus de clarté. Voici une liste des erreurs d'exécution rencontrées lors du test :

	Absolue	Relative	Proportion
NoSuchMethodException : median	13	13	11,41%
NoSuchMethodException : maximum	12	13	7,07%
NoSuchMethodException : average	12	12	6,52%
NoSuchMethodException : minimum	11	11	6,52%
NoClassDefFoundError	4	1	0,54%

– *No such method exception* indique que soit une des méthodes n'existe pas soit n'a pas les bons paramètres soit n'a pas le bon type de retour. La présence d'une telle erreur dans le test est la cause de l'appel par réflexion. Parmi celles-ci, `median` est la plus souvent manquante
– *No class def found* indique que la définition de classe n'a pas pu être chargée. C'est généralement la cause d'une déclaration de package non respectée

Voici maintenant un liste des erreurs d'assertion :

	Absolue	Relative	Proportion
AssertionError : median_equal	172	52	28,26%
AssertionError : median	62	21	11,41%
AssertionError : minimum_equal	10	10	5,43%
AssertionError : maximum_equal	13	9	4,89%
AssertionError : minimum	10	2	1,09%
AssertionError : maximum	2	1	0,54%
AssertionError : average_equal	6	1	0,54%

– *Assertion error* indique bien sûr que les tests sur la méthode n'ont pas réussi

- *average* est bien assimilée par les étudiants, il s'agit uniquement d'additionner trois nombres et de les diviser par 3. Seulement un étudiant pose problème dans le cas où des paramètres sont égaux, ce qui est assez curieux et montre qu'il n'a sans doute pas compris comment calculer une moyenne
- *minimum* et *maximum* posent plus de problèmes, la recherche de ceux-ci faisant d'habitude appel à des instructions conditionnelles if-else. Il semble que l'utilisation de celles-ci ne soit pas parfaitement maîtrisée. Il est intéressant de remarquer qu'il y a beaucoup plus d'erreurs lorsque des paramètres sont égaux. Cela laisse à penser que les étudiants utilisent les opérateurs strictement plus grand (ou plus petit) et n'envisagent pas ce cas de figure
- *median* est le gros point noir de la mission puisque 4 étudiants sur 10 n'ont pas une méthode correcte. Plus encore que `minimum` et `maximum`, `median` fait appel aux instructions `if-else`, sans doute imbriquées, et nécessite un algorithme plus complexe. Il semble donc que cette complexité soit problématique pour beaucoup et que ce problème soit trop complexe. Les étudiants ne pensent peut-être pas à réutiliser leurs méthodes précédentes pour se faciliter la tâche. Il se peut aussi cependant que le concept mathématique d'une médiane échappe à certains ou que la définition qu'ils en possèdent ne soit pas correcte. Cela semble être le cas car bon nombre d'entre eux n'ont pas un résultat correct quand des paramètres sont égaux

Résumé

Hormis les erreurs de syntaxe et les méthodes non implémentées, la compilation se passe plutôt bien. On peut remarquer de petites erreurs sur le type de retour ou les paramètres, erreurs que l'on retrouve aussi dans les tests d'ailleurs.

Pour les tests, l'arithmétique semble bien maîtrisée, ce qui n'est pas complètement le cas de l'utilisation des instructions conditionnelles `if-else`. La complexité de l'algorithme de la méthode `median` est peut-être aussi à mettre en cause de même que la notion mathématique d'une médiane.

On peut aussi se demander comment des erreurs d'assertion peuvent avoir lieu, étant donné que des méthodes de test pour chaque méthode à implémenter étaient fournies. On peut douter, dès lors qu'on suppose ces méthodes correctes, de l'utilisation systématique de celles-ci par les étudiants.

En résumé, nous pouvons dégager les points problématiques suivants :
- respect des spécifications pour les arguments et les retours de méthodes
- utilisation des instructions conditionnelles `if-else`, imbriquées surtout
- complexité de l'algorithme pour le calcul d'une médiane
- concept mathématique d'une médiane
- prise en compte des cas particuliers (paramètres égaux)
- utilisation des méthodes de test fournies

4.3 Mission 4 : Strings

4.3.1 Concepts clés

Dans cette mission, les étudiants ont pour objectif de se familiariser avec les chaînes de caractères, c'est-à-dire les objets `String`. Ils doivent pour cela écrire des méthodes en rapport avec ces chaînes, que ce soit par la manipulation ou la création d'objets `String` ou la récolte d'information au sein de ceux-ci. Il leur est aussi demandé d'écrire des méthodes de test afin de vérifier la correction de leurs propres méthodes.

L'objet de cette mission est donc de fournir un premier contact avec les `String`, objets importants en programmation s'il en est, de poursuivre l'apprentissage de création de méthodes et d'inculquer l'importance d'effectuer des tests sur le code produit. La division en sous-problème est aussi abordée.

4.3.2 Consignes

Il est ici demandé à l'étudiant de rendre une classe `BioInfo.java` qui contiendra plusieurs méthodes :

isADN
: Prend une chaîne de caractères en argument et renvoie `true` si la chaîne contient uniquement des caractères a, t, c, g et `false` sinon

count
: Prend une chaîne de caractères et un caractère en arguments et renvoie un entier représentant le nombre d'occurrences du caractère dans la chaîne

distanceH
: Prend deux chaînes de caractères de même longueur en arguments et renvoie un entier représentant la distance de Hamming [1] entre ces deux chaînes

plusLongPalindrome
: Prend un chaîne de caractères en argument et renvoie une chaîne représentant le plus long palindrome contenu dans celle-ci. Il est spécifiquement conseillé à l'étudiant de diviser l'algorithme de cette méthode en sous-problème pris en charge par des méthodes plus simples

La classe `BioInfo` devra aussi contenir des méthodes de test pour ces méthodes.

4.3.3 Tests

Les tests porteront sur les quatre méthodes principales, pas sur les méthodes de test, celles-ci n'ayant pas de spécifications définies.

1. http://fr.wikipedia.org/wiki/Distance_de_Hamming

Pour chaque méthode, nous testerons avec divers caractères et/ou chaînes de caractères, de tailles diverses également. La base de ces méthodes étant de parcourir une chaîne de caractères, les paramètres devront être tels qu'ils permettent de détecter d'éventuelles erreurs qui pourraient survenir en début, milieu ou fin de la chaîne. Nous mettrons donc en place des sous-tests pour ces cas.

On peut remarquer que les spécifications données ne contiennent pas d'indications sur comment traiter les chaînes vides (sans caractères). Bien que celles-ci ne soient pas équivoques si on prend la peine de les analyser correctement et de les appliquer au pied de la lettre, il est probable que les étudiants ne pensent pas à prendre en compte ce cas limite ou ne savent tout simplement pas comment le traiter. Nous effectuerons donc un sous-test portant sur ce cas.

4.3.4 Résultats et interprétation

Voici les résultats globaux de la mission :

Groupes	Étudiants	Compilation réussie	Test réussi
19	188	176 (93,62%)	16 (8,51%)

On remarque que la compilation s'est améliorée par rapport à la mission précédente mais que le taux de réussite des tests est très bas, moins de dix pour cent des étudiants ayant fourni des méthodes correctes.

Compilation

Voici une liste des principales erreurs :

	Absolue	Relative	Proportion
cannot find symbol	5	3	1,60%
missing return statement	2	2	1,06%
(Warning) redundant cast to int	2	2	1,06%

- *cannot find symbol* est beaucoup moins présente que lors de la mission 3, sans doute cela est-il dû au fait qu'il n'y ait pas de méthodes de tests fournies dans la classe, donc pas d'appels à un élément inexistant
- *missing return statement* indique une méthode sans retour. Sans doute un oubli mais dénote quand même que la méthode n'a pas été testée
- *redundant cast to int* n'est pas une erreur mais un avertissement du compilateur, montrant un casting inutile

Test

Comme nous avons testé de nombreux cas particuliers, nous allons séparer les erreurs d'exécution des erreurs d'assertion. Voici une liste des erreurs d'exécution rencontrées :

	Absolue	Relative	Proportion
NoSuchMethodException : plusLongPalindrome	58	58	32,95%
NoSuchMethodException : count	24	24	13,64%
NoSuchMethodException : distanceH	15	15	8,52%
NoSuchMethodException : isADN	12	12	6,82%
StringIndexOutOfBounds Exception	12	8	4,55%
NoClassDefFoundError	4	1	0,57%

– *No such method exception* montre qu'une méthode appelée n'existe pas ou n'a pas les bons paramètres ou le bon type de retour. On peut supposer qu'après la mission 3 et l'initiation aux méthodes, ces deux derniers cas sont moins probables et on en déduit donc qu'une bonne part de ces erreurs sont dues à une méthode inexistante car non écrite par l'étudiant. Et de fait on remarque que la méthode la plus compliquée, `plusLongPalindrome`, est la méthode qui manque le plus, 58 étudiants ne l'ayant pas implémentée. Cela constitue près d'un tiers des étudiants, ce qui est relativement élevé. Certaines des autres méthodes sont également manquantes mais dans une plus faible mesure. On peut quand même faire une remarque intéressante concernant `count` qui bien qu'étant moins compliquée que `distanceH` est plus souvent absente que cette dernière
– *String index out of bounds exception* est une erreur classique de manipulation de `String` et de tableaux. L'étudiant a du mal à comprendre combien d'itérations doit effectuer sa boucle et un nombre trop grand génère cette erreur. Le concept difficile à appréhender est que l'index d'un `String` commence à 0 et que le dernier élément possède donc un index *longueur-1*. Ajouter à cela le test de relation d'ordre non-stricte et une erreur de ce type peut facilement survenir. Néanmoins, seulement 8 étudiants n'est pas dramatique et l'on aurait pu s'attendre à un plus grand nombre
– *No class def found* indique toujours une classe non trouvée, sans doute à cause d'une déclaration de package

Voici maintenant une liste des erreurs d'assertion pour les méthodes de tests et sous-tests :

	Absolue	Relative	Proportion
AssertionError : plusLongPalindrome_singleCharacter	41	41	23,30%
AssertionError : plusLongPalindrome	98	38	21,59%
AssertionError : isADN	16	16	9,09%
AssertionError : plusLongPalindrome_empty	15	15	8,52%
AssertionError : count	46	14	7,95%
AssertionError : plusLongPalindrome_end	10	10	5,68%
AssertionError : distanceH	18	6	3,41%
AssertionError : isADN_empty	4	4	2,27%
AssertionError : plusLongPalindrome_long	2	2	1,14%
AssertionError : isADN_first	2	2	1,14%
AssertionError : distanceH_first	2	1	0,57%
AssertionError : isADN_last	1	1	0,57%
AssertionError : plusLongPalindrome_beginning	1	1	0,57%

- *plusLongPalindrome* est de loin la méthode la moins bien réussie, ce qui semble logique vu sa difficulté. 38 étudiant ne réussissent pas le test de base de la méthode, soit plus de 20% d'entre-eux. Le décomposition en sous-problème n'a sans doute pas été bien utilisée de même qu'à l'évidence leur méthode de test.
  - *singleCharacter* est encore moins bien réussie. C'est un test où une chaîne ne contient pas de palindrome de plusieurs lettres. Mais un palindrome d'une seule lettre est un palindrome valide, chose que les étudiants ne doivent pas avoir bien assimilée. Cela montre une mauvaise lecture ou une mauvaise interprétation des spécifications
  - *empty* est plus ambiguë. Le plus long palindrome dans une chaîne vide devrait être une chaîne vide mais on peut laisser à l'étudiant le bénéfice du doute quant au retour à fournir pour un tel paramètre
  - *beginning* et *end* testent avec des chaînes contenant un palindrome en début ou fin de chaîne. On remarque que le dernier caractère est souvent plus oublié que le premier
  - *long* teste si il s'agit bien du plus long des palindromes de la chaîne qui est renvoyé, peu de problèmes de ce côté
- *isADN* pose aussi étonnamment des problèmes alors qu'elle est la plus simple des méthodes. 23 étudiants ne réussissent pas le test principal ou un des sous-tests ; *empty* : chaîne vide, *first* et *last* : mauvais caractère en première ou dernière position
- *count* est mieux maîtrisée, 8% des étudiants seulement ayant un résultat in-

correct
- *distanceH* pose peu de problèmes, 7 étudiants ne réussissant pas les tests

Résumé

En ce qui concerne la compilation, presque toutes les erreurs ont disparu. Ce changement peut s'expliquer par une combinaison de deux facteurs ; la capacité à écrire du code syntaxiquement correct que les étudiants commencent à acquérir et l'absence de méthodes de tests fournies avec la classe `LibMath`, ce qui évite des erreurs si des méthodes ne sont pas implémentées.

Pour les tests, on peut déplorer qu'une proportion importante d'étudiants n'aient pas rempli les objectifs fixés et rendu une implémentation, même incorrecte, des méthodes demandées. `PlusLongPalindrome` en particulier est évitée par un tiers des étudiants. On peut peut-être y voir un certain découragement devant une méthode dont l'algorithme est d'un niveau plus élevé que les autres méthodes.

Au niveau des erreurs commises, de nouveau `PlusLongPalindrome` en est très rarement exempte, particulièrement avec les cas de paramètres particuliers. `isADN` est aussi étrangement problématique. Il faut peut-être en chercher la raison dans la façon qu'ont les étudiants de tester les caractères de la chaîne et dans la façon dont ils gèrent la condition de retour. `count` a plus de succès de même que `distanceH` mais les échecs constatés, de même que ceux des autres méthodes trahissent le non emploi généralisé de méthodes de test par les étudiants.

Si nous résumons, nous pouvons donc désigner les points suivants comme posant problème :
- complexité de l'algorithme de `PlusLongPalindrome`
- découpage en sous-problèmes
- prise en compte des cas particuliers (chaîne vide)
- condition des boucles (premier et dernier caractères oubliés, index dépassé)
- gestion et condition du retour
- création et utilisation de méthodes de test

4.4 Mission 5 : Tableaux

4.4.1 Concepts clés

Dans cette mission, les étudiants sont amenés à se familiariser avec les tableaux et à effectuer des opérations de plus en plus complexes sur ceux-ci. Ils doivent pour cela écrire des méthodes permettant d'effectuer des opérations sur des images, représentées par des tableaux de deux dimensions contenant des luminances de pixels sous forme d'entiers de 0 à 255. Il leur est aussi spécifiquement demandé d'écrire des classes de test pour vérifier le bon fonctionnement de ces méthodes.

L'objectif de la mission est d'initier l'étudiant aux structures de données composées de tableaux (arrays en anglais), d'utiliser les compétences déjà acquises

dans la création de méthode et d'écrire des algorithmes complexes nécessitant une division en sous-problèmes.

Il est aussi question de programmation défensive avec l'utilisation d'assertion et d'optimisation du temps d'exécution des méthodes. Nous ne pourrons cependant pas tester ces caractéristiques, les tests que nous effectuons utilisant des paramètres respectant les spécifications et le temps d'exécution des méthodes étant une donnée trop vague pour être utilisée.

4.4.2 Consignes

Il est demandé à l'étudiant de soumettre une classe `ImageGray.java` contenant plusieurs méthodes :

brighten
: Prend un tableau 2D représentant une image en argument et renvoie un autre tableau 2D, copie de l'argument mais dont les entiers auront été multipliés par un certain facteur. Cette opération revient à rendre les pixels représentés plus lumineux et donc l'image plus claire

substract
: Prend deux tableaux 2D de même taille représentant des images et un entier définissant un seuil en arguments. Renvoie un nouveau tableau 2D qui est une combinaison des deux arguments ; il contient les pixels du premier moins les pixels du second si ceux-ci ont une différence de luminance en-dessous du seuil

contains
: Prend deux tableaux 2D représentant des images et un entier définissant un seuil en arguments. Détermine ensuite si le deuxième tableau, obligatoirement plus petit que le premier, est un sous-tableau du premier, dans la limite de différence de luminance définie par le seuil.

rescale
: Prend un tableau 2D représentant une image et deux entiers définissant une hauteur et une largeur en arguments. Renvoie un nouveau tableau 2D représentant une image copie de l'argument, de taille définie par les arguments, les données de l'image étant extrapolées depuis l'image d'origine

Des classes de test doivent aussi être écrites pour chacune de ces méthodes, que nous ne pourrons tester, celles-ci n'étant pas spécifiées.

4.4.3 Tests

Pour chaque méthode, nous testerons avec divers paramètres en ce qui concerne les tableaux. Nous mettrons aussi en place des sous-tests pour vérifier l'utilisation correcte du paramètre de seuil quand celui-ci est présent. Les méthodes devant aussi ne pas modifier les arguments, nous testerons cela également dans des sous-tests.

Afin de conserver la certitude de la correction des tests, tous les résultats attendus seront codés en dur dans la classe de test et non déterminé par un algorithme, celui-ci pouvant également se révéler erroné.

4.4.4 Résultats et interprétation

Voici les résultats globaux de la mission :

Groupes	Étudiants	Compilation réussie	Test réussi
19	165	155 (93,94%)	11 (6,67%)

On remarque que les résultats de compilation se situent au même niveau que la mission précédente, le taux d'erreur étant d'environ 7%. On peut commencer à envisager ce chiffre comme un moyenne du taux de réussite de compilation quand la classe est fournie.

La réussite des tests chute encore un peu à environ 7%. Un si bas niveau malgré l'utilisation des méthodes depuis deux missions déjà tend à montrer que la difficulté de cette mission réside dans le fait d'écrire des algorithmes corrects.

Compilation

Voici une liste des principales erreurs :

	Absolue	Relative	Proportion
(Warning) redundant cast to int	43	27	16,36%
cannot find symbol	15	5	3,03%
(Warning) redundant cast to double	5	5	3,03%
missing return statement	2	2	1,21%

- *redudant cast* apparaît suffisamment pour être souligné. Que ce soit avec `int` ou `double`, certains étudiants semblent avoir des problèmes pour savoir quel type de variable ils sont en train d'utiliser
- *cannot find symbol* et *missing return statement* sont toujours présentes mais dans des proportions négligeables

Test

Encore une fois, nous allons séparer les erreurs d'exécution des erreurs d'assertion pour plus de clarté. Voici une liste des erreurs d'exécution rencontrées :

	Absolue	Relative	Proportion
ArrayIndexOutOfBounds Exception	39	32	20,65%
ArithmeticException	3	3	1,94%
NoSuchMethodException : rescale	2	2	1,29%
NoSuchMethodException : contains	2	2	1,29%
NoSuchMethodException : brighten	2	2	1,29%
NoSuchMethodException : subtract	1	1	0,65%

– *Array index out of bounds* est une erreur classique lors de la manipulation de tableaux. Le parcours des éléments de ceux-ci se faisant principalement à l'aide de boucles `for`, la condition d'arrêt de celles-ci doit bien prendre garde à ne pas dépasser les index des tableaux et sous-tableaux. C'est souvent le cas lorsque les étudiants définissent la condition d'arrêt à l'aide d'une relation d'ordre non stricte ou pire, la définissent à l'aide de nombres plutôt qu'avec la longueur du tableau à parcourir. On peut par contre supposer que le concept d'index démarrant à 0 a été assimilé depuis la mission précédente
– *Arithmetic exception* se réfère généralement à une division par 0. C'est une erreur courante mais peu alarmante
– *No such method* montre qu'aucune méthode correspondant aux spécifications n'a été trouvée. Ce qui est assez curieux car le corps des méthodes était fourni dans le fichier squelette de la classe `ImageGray`

Voici maintenant une liste des erreurs d'assertion pour les tests et sous-tests :

	Absolue	Relative	Proportion
AssertionError : contains	207	106	68,39%
AssertionError : rescale	398	70	45,16%
AssertionError : subtract	88	32	20,65%
AssertionError : brighten_modify	20	20	12,90%
AssertionError : subtract_modify	38	19	12,26%
AssertionError : brighten	45	15	9,68%
AssertionError : contains_threshold	24	14	9,03%
AssertionError : subtract_threshold	11	5	3,23%
AssertionError : contains_equal	1	1	0,65%

- *contains* est manifestement la méthode posant le plus de difficultés. Celle-ci comportant l'algorithme la plus compliqué, cela n'a rien de surprenant. La division en sous-problème n'a sûrement pas été appliquée par un bon nombre des 106 étudiants n'ayant pas une méthode correcte. Cette manière de procéder est ici obligatoire vu le nombre de paramètres à prendre en compte pour fournir une solution correcte ; les tailles respectives des tableaux et l'obligation d'utiliser plusieurs boucles `for` imbriquées
- *rescale* est aussi difficilement réussie. Il faut peut-être prendre en compte la difficulté de comprendre correctement les spécifications. Toute personne ayant déjà manipulé une image comprend ce qui se passe au niveau visuel lorsque l'on modifie la taille d'une photo. C'est par contre beaucoup moins évident lorsque l'on s'intéresse à la mécanique derrière cette modification. Le rôle des facteurs d'étirement n'est peut-être pas évident pour tous les étudiants
- *subtract* n'est pas maîtrisée par un cinquième des étudiants. Cette méthode est pourtant plus simple que les deux précédentes mais le fait de créer un nouveau tableau en se basant sur deux autres semble poser problème
- *brighten_modify* et *subtract_modify* vérifient que les méthodes `brighten` et `subtract` ne modifient pas les paramètres. Ces arguments étant des objets non protégés, leur contenu peut être modifié par les méthodes, ce que les spécifications interdisent. Il est probable que plutôt que de renvoyer un nouveau tableau, les étudiants renvoient le tableau argument modifié
- *contains_threshold* et *subtract_threshold* vérifient la correction des méthodes `contains` et `subtract` lorsque le paramètre de seuil doit être utilisé, c'est-à-dire quand le deuxième tableau en argument n'est pas exactement un sous-tableau du premier. On teste dans ce cas avec un seuil supérieur à zéro. Certains étudiants semblent ne pas prendre en compte ce paramètre
- *contains_equal* vérifie qu'un sous-tableau est bien un de ses propres sous-tableaux. Ce qu'un étudiant n'a pas l'air d'avoir envisagé

Résumé

Pour la compilation, le niveau est le même que précédemment. Il faut toutefois prendre garde aux avertissements de casting redondants. Cela pourrait montrer que l'étudiant s'embrouille avec les types de données qu'il manipule et que par précaution, il utilise un `cast` pour s'assurer du fonctionnement de son code. Ce genre de comportement est bien sûr à éviter car, même si cela ne prête pas à conséquence dans ce cas-ci, il peut être dangereux de perdre la compréhension de son propre code.

Pour les erreurs d'exécutions, le problème majeur se situe au niveau des sorties d'index. Ce concept, bien que parfois peu naturel à cause d'un index de départ nul, est extrêmement important à acquérir pour la manipulation de tableaux. Pouvoir parcourir tous les éléments d'un tableau sans en oublier ni sortir de celui-ci est une compétence de base de tout programmeur.

En ce qui concerne les erreurs d'assertion, `contains` est à l'évidence un trop gros morceau pour 7 étudiants sur 10. La division en sous-problème, étape obligatoire pour l'écriture d'une telle méthode, n'est sans doute pas appliquée systématiquement. On peut aussi soupçonner que les étudiants ne prennent pas le temps d'analyser le problème avant de commencer à écrire leur code. L'analyse sur papier des données et spécifications est en effet primordiale dans ce cas. L'algorithme de cette méthode, bien que complexe devrait commencer à être à leur portée mais nécessite une bonne dose de réflexion.

Les autres méthodes sont aussi fort laborieuses mais à un niveau plus raisonnable, bien que le respect des spécifications ne soit pas encore tout le temps au rendez-vous, malgré l'encouragement à la programmation défensive. L'application de celle-ci doit se faire aussi bien pour les conditions `pre` que `post`, ce qui semble échapper à certains.

Enfin, un tel niveau d'échec laisse penser que soit les classes de test n'ont pas été écrites ou en tout cas mal écrites, soit simplement pas utilisées. L'importance de tests unitaires bien écrits et leur utilisation ne semble pas encore assimilées par beaucoup. Mais l'écriture de tests complets est peut-être elle-même trop compliquée et le fait que certains cas particuliers ne leur viennent pas à l'esprit lors de l'écriture de la méthode les empêchent sans doute de les intégrer dans les tests. Il pourrait être intéressant de fournir des classes de tests sous forme de bytecode, de façon à ne pas donner donner des réponses trop flagrantes mais à fournir des indications concernant la voie à suivre.

En résumé, nous pouvons souligner les points suivants :
- compréhension de son propre code
- analyse préalable du problème
- décomposition en sous-problèmes
- condition d'arrêt de boucle
- écriture de tests complets et utilisation de ceux-ci
- respect des spécifications et des conditions `post` en particulier

4.5 Mission 6 : Classes et objets

4.5.1 Concepts clés

Dans cette mission, les étudiants sont confrontés pour la première fois à l'écriture d'une classe entière. Ils doivent définir deux classes représentant des objets, l'une faisant référence à l'autre, ainsi qu'une troisième contenant le programme qui utilisera ces objets. Ces classes sont utilisées dans le cadre de la gestion d'un catalogue de musique.

Le but de la mission est donc d'inculquer les bases de la programmation orientée objet et quelques grands principes de celle-ci tels que les instances, les variables d'instance, les références ou encore les constructeurs.

Des squelettes de classes de test sont aussi fournis afin de leur permettre de développer des tests et de s'assurer de la correction de leurs classes.

4.5.2 Consignes

L'étudiant doit fournir trois classes qu'il doit créer lui même.

La première est une classe `Temps.java`, celle-ci représentant un temps, et doit contenir trois variables entières pour les heures, minutes et secondes. Elle doit de plus contenir un constructeur prenant trois entiers en arguments. Elle doit aussi contenir les méthodes suivantes :

toSecondes
: Renvoie le temps représenté sous fourme d'un entier, la somme des secondes que ce temps représente

delta
: Prend un temps *t* en argument et renvoie la différence de temps en secondes sous forme d'entier entre ce temps et le temps *t*. L'entier est positif si ce temps est plus grand

apres
: Prend un temps *t* en argument et renvoie un booléen définissant si ce temps est plus grand que le temps *t*

ajouter
: Prend un temps *t* en argument et modifie ce temps ci en lui ajoutant le temps *t*. La méthode doit corriger les variables minutes et secondes si celles-ci dépassent 59

toString
: Renvoie une représentation de ce temps sous forme de `String`. La forme de celui-ci est "HH:MM:SS"

La seconde classe est une classe `Chanson.java`, représentant une chanson du catalogue. Elle doit contenir trois variables ; un titre et un auteur sous forme de `String` et une durée sous forme d'un objet `Temps` défini par la classe précédente. Cette classe doit comprendre des méthodes pour renvoyer les variables d'instance (getters) et une méthode `toString`.

La troisième classe est une classe `Programme.java` contenant la méthode principale. Celle-ci lit des descriptifs de chansons depuis la console ou depuis un fichier et crée les objets correspondants.

4.5.3 Tests

Pour les deux classes `Temps` et `Chanson`, nous allons tester les méthodes requises avec différents paramètres ainsi que leur présence au moyen de sous-tests. Nous allons aussi vérifier la présence du constructeur pour la classe Temps et vérifier son bon fonctionnement avec le test principal.

Les spécifications de la classe `Programme` n'étant pas clairement établies, nous ne pourrons pas la tester. De plus les spécifications de la méthodes `toString` de la classe `Chanson` n'étant pas assez explicite, nous ne testerons pas non plus

cette méthode, les résultats risquant d'être trop erratiques. Le test de la même méthode dans la classe `Temps` devrait nous donner un aperçu convenable de la capacité des étudiants à écrire une telle méthode.

4.5.4 Résultats et interprétation

Voici les résultats globaux de la mission :

Groupes	Étudiants	Compilation réussie	Test réussi
21	165	123 (74,55%)	60 (36,36%)

On remarque que la compilation est moins bien réussie que précédemment cela étant sans doute à mettre sur le fait que les étudiants doivent écrire les classes de A à Z et plus seulement des méthodes dans des classes fournies.

Les tests sont mieux réussis, plus d'un tiers des étudiants ayant fourni un code correct. Il faut cependant souligner que cette mission ne comporte pas d'algorithmes très compliqués et qu'elle se concentre principalement sur des concepts.

Compilation

Les erreurs de compilation, bien que plus variées dans cette mission, sont cependant encore principalement composées d'erreurs de syntaxe que nous ne détaillerons pas ici. Voici donc une liste des erreurs nous paraissant intéressantes :

	Absolue	Relative	Proportion
(Warning) redundant cast to int	36	13	7,88%
cannot find symbol	56	10	6,06%
incompatible types	17	4	2,42%
(Warning) static method should be qualified by type name, java.lang.String, instead of by an expression	5	4	2,42%
bad operand types for binary operator '+'	2	2	1,21%
invalid method declaration ; return type required	1	1	0,61%
constructor Temps in class Temps cannot be applied to given types ;	1	1	0,61%
constructor Chanson in class Chanson cannot be applied to given types ;	1	1	0,61%

– *redundant cast* apparaît de nouveau mais en plus petite proportion que lors de la mission précédente mais on remarque que 13 étudiants ont toujours du mal à comprendre à quels types de variables ils ont affaire

– *cannot find symbol* est toujours présente mais est sûrement due à des erreurs de syntaxe
– *incompatible types* dénote généralement l'utilisation d'une méthode avec de mauvais paramètres
– *static method* montre que certains étudiants utilisent une méthode statique sur un objet `String`. Cela trahit sans doute une confusion quant au caractère statique des méthodes
– *bad operand* montre généralement l'utilisation de l'opérateur "+" avec des types incorrects. Dans la plupart des cas, l'étudiant essaye d'additionner des objets, espérant additionner leurs variables d'instances
– *invalid method declaration* indique que l'étudiant a mal défini l'en-tête de sa méthode en oubliant le type de retour. Il a peut-être aussi déclaré le constructeur avec un nom différent de celui de la classe

Test

De nouveau, nous séparerons les erreurs d'exécution des erreurs d'assertion. Voici la liste de erreurs d'exécution rencontrées :

	Absolue	Relative	Proportion
NoSuchMethodException : temps_toSecondes	20	20	16,26%
NoSuchMethodException : chanson_getters	11	11	8,94%
NullPointerException	10	10	8,13%
NoSuchMethodException : temps_delta	2	2	1,63%
NoSuchMethodException : temps_ajouter	1	1	0,81%
NoSuchMethodException : temps_toString	1	1	0,81%
StackOverflowError	2	1	0,81%

– *no such method* montre que les méthodes en question n'ont pas été implémentées. L'erreur pour la méthode `toSecondes` est plus une anomalie car ces étudiants ont nommé leur méthode `toSeconds`. C'est donc plus une erreur de respect des conventions de nommage
– *null pointer exception* est une erreur courante lors de l'utilisation d'objets. Dans le cas présent, celle-ci survient principalement lors du l'utilisation du constructeur de `Chanson`, quand un objet `Temps` égal à `null` est utilisé. Au lieu d'assigner celui-ci à leur variable d'instance, certains étudiants tentent de récupérer les informations qu'il contient, ce qui mène à cette erreur
– *stack overflow* montre généralement un récursion infinie. Dans le cas présent, un étudiant a défini ses méthodes `get` de telle façon qu'elles utilisent

un appel à elles-mêmes

Voici maintenant une liste des erreurs d'assertion :

	Absolue	Relative	Proportion
AssertionError : temps_delta	16	15	12,20%
AssertionError : temps_ajouter	17	9	7,32%
AssertionError : temps_toString	11	7	5,69%
AssertionError : temps_apres	9	5	4,07%
AssertionError : temps_toSecondes	2	1	0,81%
AssertionError : chanson_getters	2	1	0,81%

- *temps_delta* est une erreur la plupart du temps générée par un mauvais signe du retour. Il peut s'agir d'une mauvaise application des conventions mais il se peut aussi que les étudiants aient du mal à bien utiliser les références `this` à l'objet courant
- *temps_ajouter* pose quelques problèmes, pour la plupart causés par un mauvais report lors d'un dépassement de borne supérieure pour les minutes et secondes
- *temps_toString* n'est pas une erreur alarmante. Le fait que ce soit une erreur d'assertion montre que la méthode a été bien écrite mais que le résultat produit n'est pas celui attendu, ce qui peut être dû à des espaces ou juste à un non-respect des conventions
- *temps_après* pose peu de problèmes. C'est un algorithme de comparaison qui devrait normalement être facilement implémentable à ce niveau d'enseignement. Seul 5 étudiants en sont encore incapables
- *temps_toSecondes* est bien maîtrisée
- *chanson_getters* est un test portant sur toutes les méthodes `get` en une fois, vu leur simplicité. Cette façon d'accéder aux variables semble bien comprise

Résumé

Pour la compilation, on remarque bien plus d'erreurs que précédemment, sans doute dues à l'écriture complète des classes par les étudiants. Les castings redondants sont une nouvelle fois présents, trahissant une incertitude dans le chef des étudiants concernant le type de variables qu'ils manipulent. On remarque aussi une petite confusion avec l'utilisation des méthodes statiques, dont le concept semble échapper à certains.

Pour les erreurs d'exécution, la plus logique à trouver ici est celle du pointeur `null`. Cela montre que les étudiants ne protègent pas leurs méthodes contre la possibilité pour un objet d'être `null` quand celui-ci doit être utilisé. Cette une erreur classique que l'on peut rencontrer partout.

En ce qui concerne les erreurs d'assertion, celles-ci sont peu nombreuses mais rappelons-le, les fonctions qu'elles implémentent ne sont pas très compliquées. On

peut toujours remarquer que les spécifications ne sont parfois pas respectées. On peut aussi s'inquiéter que les étudiants sachent faire la différence entre un objet passé en paramètre et l'objet courant.

En résumé, les points suivants semblent d'importance :
- correction de la syntaxe d'une classe entière
- compréhension de son propre code
- différences entre méthodes statiques et non-statiques
- pointeur `null`
- respect des spécifications
- utilisation de `this`

4.6 Mission 7 : Extension et héritage

4.6.1 Concepts clés

Dans cette mission, les étudiants sont amenés à rencontrer des concepts tels que l'héritage et l'extension de classe. Ils doivent écrire des classes complètes étendant une classe fournie, redéfinir des méthodes et utiliser des outils comme `super`, `this`, la classe générique `Object` ou les méthode `instanceOf` et `equals`.

Le but de la mission est donc de donner un aperçu de ce qu'est l'héritage, qui est un concept de base du langage Java, et de mettre en pratique celui-ci avec un cas réel de logiciel de facturation simplifié.

Des classes à étendre ou à étoffer sont fournies, de même que des classes de test qui devront elles aussi être complétées.

4.6.2 Consignes

Il est demandé à l'étudiant de fournir trois nouvelles classes.

La première est la classe `ArticleReparation.java`, qui étend la classe `Article` fournie. Celle-ci représente un article d'une facture concernant des prestations de réparation. Elle doit contenir une variable d'instance `double`, représentant la durée du travail effectué et un constructeur prenant un `double` correspondant en argument.

Deux méthodes doivent être redéfinies :

getDescription
: Renvoie un `String` contenant un descriptif des prestations

getPrix
: Renvoie un `double` représentant le coût des prestations en fonction de la durée de celles-ci

La deuxième classe est la classe `Piece.java` qui représente une pièce à facturer. Elle doit contenir plusieurs variables d'instance, dont les plus importantes sont un `String` représentant la description de la pièce et un `double` représentant son prix, ainsi qu'un constructeur prenant ces deux variables en paramètres. Elle

doit aussi contenir des méthodes `get` permettant d'accéder à ces variables et doit redéfinir la méthode `equals` de la classe `Object` pour tester l'équivalence de deux objets `Piece`.

La troisième classe est la classe `ArticlePiece.java`, qui étend la classe `Article` fournie. Celle-ci représente un article d'une facture concernant une pièce. Elle doit contenir deux variables d'instance, un entier définissant le nombre de pièces et un objet `Pièce` référençant la pièce dont il est question. Elle doit également redéfinir les méthodes `getDescription` et `getPrix` et implémenter un troisième `getTVA` qui renvoie un taux de TVA en fonction d'une éventuelle réduction.

Enfin, la classe `Facture` doit être modifiée en lui adjoignant un numéro séquentiel unique et deux nouvelles méthodes :

getNombre
: Prend un objet `Pièce` en paramètre et renvoie un entier représentant le nombre de pièces identiques au paramètre dans la facture

printLivraison
: Permet d'imprimer une facture

4.6.3 Tests

Au fur et à mesure de l'avancement des missions, les spécifications se font de moins en moins précises, pour permettre à l'étudiant de prendre plus d'initiative et de l'obliger à réfléchir. Malheureusement, cela cause un effet négatif sur les tests car il est impossible ou du moins très compliqué d'effectuer un test sur une classe ou une méthode non-spécifiée.

Néanmoins, pour ce qui a été spécifié précisément, nous allons effectuer des tests au sein des trois classes de l'étudiant, vérifier la présence des constructeurs et des méthodes obligatoires et vérifier le retour de celles-ci. Nous vérifierons aussi que l'héritage est bien présent.

Les choses qu'il nous sera impossible de tester sont les retours des méthodes `getDescription`, qui ne sont pas spécifiés, le constructeur général de la classe `Piece`, l'ordre des paramètres n'étant pas spécifié, la méthode `getTVA`, les valeurs des variables d'instances de `Piece` n'étant pas définies par défaut et la méthode `printLivraison` dont l'output n'est pas spécifié non plus.

4.6.4 Résultats et interprétation

Voici les résultats globaux de la mission :

Groupes	Étudiants	Compilation réussie	Test réussi
19	208	170 (81,73%)	42 (20,19%)

La compilation, bien qu'imparfaite, reste dans des niveaux acceptables, vu le nombre de classes à fournir et de méthodes à implémenter. De fait, malgré une

augmentation du code à produire, le taux de compilation est supérieur à celui de la mission précédente, ce qui démontre une amélioration de la capacité des étudiants à écrire du code syntaxiquement correct.

Les tests sont moins bien réussis cependant, seulement un étudiant sur cinq parvenant à fournir des résultats corrects, du moins pour ce qui a été testé, ce chiffre étant donc à prendre comme une borne supérieure.

Bien que les méthodes ne soient pas très compliquées au niveau des algorithmes, elles font appel à des concepts abstraits et nécessitent un compréhension certaine de la manière donc fonctionne le langage Java et des concepts des langages orientés objets. On peut donc sans doute voir dans ce taux de réussite médiocre, le fait qu'il s'agit d'une première rencontre des étudiants avec toute la mécanique de Java.

Compilation

Les erreurs de compilation, bien que peu nombreuses, sont ici extrêmement variées et il serait inutile des les citer toutes. Nous laisserons de côté les erreurs de syntaxe et nous concentrerons sur la liste des erreurs intéressantes, regroupées par catégories, sans nous préoccuper du nombre de leurs occurrences, celles-ci étant limitées à quelques étudiants pour chaque type d'erreur :

- *cannot find symbol* est toujours présente, les classes de tests fournies faisant appel à des éléments potentiellement manquants
- *static variable should be qualified by type name* est un avertissement quant à l'utilisation d'une méthode statique sur un objet, chose qui devrait être évitée
- *redundant cast* est encore présente, ce qui montre que les étudiants perdent parfois la notion des types
- *non-static method cannot be referenced from static context* apparaît quand une méthode non-statique est appelée dans un contexte statique. Elle montre que les principes des méthodes statiques ne sont pas encore bien compris par certains
- *method has been deprecated* est présente, ce qui montre que des étudiants utilisent des méthodes dépréciées, pour les objets `Date` en particulier
- *method cannot be applied to given types* montre un mauvais appel de méthode
- *has private access* est une erreur présente également. Il se peut donc que certains n'aient pas l'habitude d'utiliser les méthodes `get` pour accéder aux variables d'instance

Test

Encore une fois nous séparerons les erreurs d'exécution des erreurs d'assertions. Voici la liste des erreurs d'exécution :

	Absolue	Relative	Proportion
NoSuchMethodException : facture_getNombre	54	54	31,76%
NoSuchMethodException : piece_equals	45	45	26,47%
NoSuchMethodException : articleReparation	41	41	24,12%
NoSuchMethodException : articlePiece_getPrix	26	26	15,29%
NoSuchMethodException : articleReparation_getDescription	14	14	8,24%
NoSuchMethodException : articlePiece	13	13	7,65%
NoSuchMethodException : articlePiece_getDescription	12	12	7,06%
NullPointerException	12	9	5,29%
NoSuchMethodException : piece	9	9	5,29%
NoSuchMethodException : articleReparation_getPrix	8	8	4,71%
StackOverflowError	1	1	0,59%
IllegalArgumentException	1	1	0,59%

- *No such method* apparaît pour de nombreuses méthodes :
  - *facture getNombre* n'a pas été implémentée par 54 étudiants, sans doute à cause de sa difficulté. Néanmoins le test ayant besoin de constructeurs qui sont peut-être manquants, il se peut que ce chiffre élevé soit dû à cette absence
  - *piece equals* est également peu implémentée sans doute à cause de sa complexité elle aussi
  - *articleReparation*, *articlePiece* et *piece* montrent tous une absence de constructeur pour les classes correspondantes, ce qui est un non-respect des spécifications. On peut remarquer que certains étudiants ont créé leurs propres constructeurs sans faire appel à `super` et qu'ils recréent des variables d'instances déjà présentes dans la super-classe
  - les méthodes *getPrix* sont aussi relativement évitées bien qu'elles ne soient pas compliquées en-soi
  - les sous-tests *getDescription* montrent juste l'absence de la méthode
- *null pointer exception* montre que certains étudiants ne vérifient pas la valeur non-nulle des paramètres passés à leurs méthodes
- *stack overflow* montre un appel de méthode récursif à l'infini
- *illegal argument* montre juste un appel de méthode avec un mauvais argument

On fera néanmoins remarquer que l'absence de certaines méthodes peut aussi s'expliquer par le non-respect des spécifications ou des signatures de méthodes à redéfinir.

Voici maintenant la liste des erreurs d'assertion :

	Absolue	Relative	Proportion
AssertionError : facture_getNombre	51	51	30,00%
AssertionError : articlePiece_getPrix	18	18	10,59%
AssertionError : piece_equals	4	4	2,35%
AssertionError : articleReparation_getPrix	4	4	2,35%
AssertionError : articlePiece	2	2	1,18%
AssertionError : articleReparation	1	1	0,59%

- *facture get nombre* est la méthode la plus compliquée à réaliser, il est donc normal que les étudiants aient quelques difficultés. Celle-ci fait en effet appel à l'opérateur `instanceOf` qui est peut-être difficile à maîtriser
- les méthodes *getPrix* posent problème à certains. Le calcul peut être compliqué à cause du fait qu'il faille récupérer des données dans un objet extérieur
- *piece equals* est également problématique pour 4 étudiants. Cette méthode devant respecter la signature de la méthode `equals`, elle doit prendre un argument de type `Object`. Celui-ci doit alors subir un casting avant d'être comparable avec un objet `Piece`. Cette subtilité a peut-être échappé à certains
- *articlePiece* et *articleReparation* montrent que ces deux classes n'ont pas étendu la classe `Article` comme demandé

Résumé

Pour la compilation, les erreurs rencontrées étaient peu nombreuses mais de types extrêmement variés. En les parcourant, on peut cependant noter avant tout que le principe des méthodes statiques est parfois mal assimilé par certains étudiants. On pourra aussi remarquer que l'emploi de castings inutiles continue à être fréquent. On remarque aussi l'utilisation de méthodes dépréciées, qui bien qu'inutiles dans cette mission, montre que quelques étudiants font preuve d'initiative dans leur implémentation en utilisant des méthodes qu'ils découvrent ailleurs que dans ce cours. L'utilisation de `super` et des variables d'instances de la superclasse pose aussi parfois problème.

Pour les erreurs d'exécution, le grand problème est évidemment l'absence des méthodes ou constructeurs à tester. Cela peut trahir un non-respect des spécifications tout comme une trop grande difficulté dans l'implémentation. Les erreurs de

pointeur `null` montrent qu'il manque encore une vraie pensée "objets" dans le chef des étudiants.

En ce qui concerne les erreurs d'assertion, certaines méthodes se révèlent peut-être trop compliquées, comme quand il s'agit de parcourir un tableau pour effectuer un calcul sur les éléments d'une certaine classe uniquement. Certaines méthodes sont par contre plus réussies comme `equals`, ce qui est assez étonnant.

En résumé, nous pouvons dégager les points problématiques suivants :

- utilisation des méthodes statiques
- compréhension de son propre code
- respect des spécifications
- pointeur `null`
- complexité de la méthode `getNombre`, trop de concepts sont appliqués à la fois
- utilisation de `super` et des variables d'instance de la super-classe

4.7 Mission 8, 9 et 10

Les missions 8, 9 et 10, de par leurs spécifications et la forme des classes à tester, ne permettent pas de réaliser une analyse correcte qui pourra fournir des résultats dignes d'intérêt.

La mission 8 traite des interfaces graphiques, domaine qui nécessite avant tout un examen visuel. Les tests de ces classes ne seront donc que peu révélateurs.

Les missions 9 et 10 sont consacrées à l'utilisation d'un dictionnaire dans un programme utilisant des entrées/sorties à la console. De plus, une grande liberté est laissée à l'étudiant dans ses choix d'implémentation, ce qui fait que tous les programmes sont différents et donc non-testables automatiquement. Seule la compilation et la présence de certaines méthodes pourraient être vérifiées, ce qui est peu utile à ce niveau du cours.

4.8 Mission 11 : Structures chaînées

4.8.1 Concepts clés

Dans cette mission, les étudiants sont amenés à implémenter une interface qui fourni des méthodes pour la gestion d'une pile. La classe ainsi créée sera utilisée dans un programme plus vaste traitant de L-Systems. Les éléments de la pile sont des objets `State`, définis dans une classe fournie. La façon dont la pile est implémentée est laissée au choix de l'étudiant.

Les étudiants seront aussi amenés à utiliser JUnit pour la première fois par la création d'une classe de test JUnit 3 qui testera leur implémentation.

Le but de cette mission est donc de vérifier la compréhension du concept des interfaces et la capacité à gérer une structure de données complexe.

Ne pouvant pas tester le programme principal ni les classes de test, nous nous contenterons donc d'analyser la classe de l'étudiant.

4.8.2 Consignes

L'étudiant doit compléter une classe `Stack.java` qui lui est fournie, en implémentant les méthodes de l'interface `StackIF.java` et en ajoutant un constructeur sans arguments. Les méthodes à implémenter sont les suivantes :

push
: Prend un objet `State` non `null` en argument et le place au sommet de la pile

pop
: Renvoie l'objet `State` se trouvant au sommet de la pile et le retire de celle-ci. Si celle-ci est vide, une exception `EmptyStackException`, définie dans une classe fournie, est lancée

size
: Renvoie le nombre d'objets `State` dans la pile

isEmpty
: Renvoie `true` si la pile ne contient pas d'objets `State`, `false` sinon

peek
: Prend un entier compris entre 0 et la taille de la pile représentant un index et renvoie l'objet `State` présent dans la pile à cet index

4.8.3 Tests

Nous allons d'abord tester que la classe `Stack` implémente bien l'interface `StackIF`. Nous testerons ensuite la présence des méthodes et le bon fonctionnement de celles-ci. Pour les méthodes qui nécessitent que la pile ne soit pas vide, un sous-test sera réalisé, faisant appel à la méthode `push`. Le bon fonctionnement de cette dernière est donc nécessaire au test de la méthode. Nous testerons aussi qu'une exception est bien renvoyée par la méthode `pop` quand elle est utilisée sur une pile vide.

4.8.4 Résultats et interprétation

Voici les résultats globaux de la mission :

Groupes	Étudiants	Compilation réussie	Test réussi
19	161	158 (98,14%)	104 (64,60%)

La compilation est ici quasiment parfaite, ce qui est à la fois normal étant donné que cette mission est la dernière du cours et très réjouissant. On peut y voir l'influence de l'utilisation de JUnit qui nécessite une classe compilée pour fonctionner.

Les tests sont moins bien réussis, mais restent dans une proportion très raisonnable. L'implémentation d'interface ayant déjà été abordée dans la mission 8, on peut supposer que ce qui pose problème est la gestion d'une structure de données complexe.

Compilation

Seulement trois étudiants ont commis des erreurs de compilation. Nous allons nous contenter de les citer :

- *(Warning) found raw type* montre que le type générique ou le type `State` n'a pas été utilisé pour la création de la structure de données
- *unchecked conversion* montre une erreur de type pour une `LinkedList`
- *constructor EmptyStackException in class EmptyStackException cannot be applied to given types* montre que le constructeur de l'objet `EmptyStackException` n'a pas été utilisé correctement

Toutes ces erreurs sont mineures et ne portent pas à conséquence.

Test

Voici une liste des erreurs d'exécution rencontrées :

	Absolue	Relative	Proportion
NullPointerException	38	24	15,19%
EmptyStackException	7	7	4,43%
InstantiationException	1	1	0,63%
StackOverflowError	1	1	0,63%

- *null pointer* est ici assez courante. La structure de données n'étant certainement pas `null`, on peut supposer que l'erreur survient lors d'un `pop` ou d'un `peek`
- *empty stack* est l'exception normalement levée lors d'un `pop` sur une pile vide. Sa présence montre qu'elle a été lancée hors de ce contexte alors qu'elle n'aurait pas dû l'être
- *instantiation exception* montre une absence de constructeur sans argument pour la classe `Stack`
- *stack overflow* est ici causée par un constructeur s'appelant lui-même

Voici maintenant la liste des erreurs d'assertion :

	Absolue	Relative	Proportion
AssertionError : stack_pop	18	18	11,39%
AssertionError : stack_peek_push	17	17	10,76%
AssertionError : stack_isEmpty_push	5	5	3,16%
AssertionError : stack_size	5	5	3,16%
AssertionError : stack_size_push	3	3	1,90%
AssertionError : stack_pop_push	3	3	1,90%

- *stack pop* est la moins bien réussie, cela est dû au fait qu'il faille prévoir de retirer l'élément de la pile et de lancer une exception si celle-ci est vide, ce qui n'est pas toujours accompli
- *stack peek push* pose aussi des problèmes. C'est un sous-test où l'on vérifie la présence d'un élément à l'endroit prévu après l'y avoir poussé. On peut donc supposer que l'ordre d'insertion dans la pile n'est pas respecté
- *stack isEmpty push* renvoie qu'une liste est vide alors qu'elle ne l'est pas. Cela est dû à une mauvaise structure de donnée dont la taille n'est pas incrémentée à l'ajout d'un élément
- *stack size* est dû à une instanciation de pile non vide
- *stack size push* est aussi dû à une combinaison des deux causes précédentes
- *stack pop push* vient aussi du fait que la structure de données est mal implémentée

On rappellera que les tests utilisant `push` sont aussi soumis au bon fonctionnement de cette méthode.

Résumé

Pour la compilation, les erreurs sont négligeables. Leur nombre très bas est très certainement dû à l'obligation des étudiants de mettre en place une classe de tests unitaires JUnit, l'utilisation de celle-ci nécessitant une classe qui compile.

Pour ce qui est des erreurs d'exécution, on remarque surtout que les erreurs de pointeur `null` sont encore trop nombreuses. Le respect scrupuleux des spécifications devant éviter cela, il faut surtout y voir une mauvaise structure de données. La création de tests unitaires aurait du permettre de remarquer ces erreurs, on peut donc aussi supposer que tous les cas de figure n'ont pas été testés.

Pour les erreurs d'assertion, c'est principalement la forme de la structure de données, une liste chaînée, qui pose problème. Le choix leur étant laissé dans l'implémentation de celle-ci, certains ont utilisé une `LinkedList`, d'autres ont créé leur propre liste composée d'objets `Node`. Bien qu'il existe de nombreux moyens d'implémenter une liste chaînée, certains sont plus pratiques et moins compliqués que d'autres. Le choix d'un bon modèle a donc peut-être été difficile pour certains.

En résumé, nous pouvons dégager les points à problèmes suivants :
– pointeur `null`
– couverture des cas particuliers par les tests
– complexité d'une structure de donnée

Chapitre 5

Améliorations des missions

L'analyse effectuée au chapitre précédent a permis d'identifier et de mettre au jour différents concepts mal assimilés, situations problématiques ou états de fait à éviter. Nous allons maintenant nous efforcer d'apporter des suggestions d'améliorations à appliquer aux missions afin de permettre différentes choses :

- la réduction des problèmes d'assimilation de certains concepts par les étudiants
- l'augmentation des moyens mis en place pour éviter les erreurs
- la maximisation de la testabilité des délivrables de chaque mission

5.1 Tests unitaires et cas particuliers

Tout au long des missions, les étudiants ont été encouragés à tester leur code. Que ce soit par l'écriture de leurs propres classes de test ou grâce à des tests fournis. Néanmoins nous constatons que des erreurs subsistent malgré ces tests. Bien que l'on ne puisse pas faire grand chose si les étudiants ne prennent pas la peine d'écrire ou d'utiliser ces tests, il pourrait être bénéfique de mettre en place certaines pratiques.

5.1.1 Tests à écrire ou compléter

Dans beaucoup de missions, les classes de test sont fournies avec le package de la mission mais souvent l'étudiant doit les compléter lui-même avec des tests de son cru. Il arrive aussi que l'étudiant doive écrire ses propres classes entièrement.

Le problème survenant alors est que la plupart du temps l'étudiant écrira un test de sa méthode pour le cas le plus courant et il ne lui viendra pas à l'idée de tester des cas particuliers comme des paramètres vides, `null` ou du mauvais type.

Nous proposons donc de joindre à chaque mission des indications quant aux cas à tester. En faire une liste exhaustive serait sans doute trop évident. Il s'agirait donc de glisser subtilement des allusions à ces cas particuliers dans le texte ou

dans les exercices préliminaires que les étudiants réalisent en début de mission. Une question portant par exemple sur le résultat d'une méthode sur un objet `null` devrait sans doute permettre aux étudiants d'entrevoir la variété des cas possibles et d'élargir leur vision au-delà des cas où tout fonctionne normalement.

Dans la mesure du possible, nous proposons aussi que les tests soient écrits avant les classes principales, ce qui forcera l'étudiant à écrire des tests respectant les spécifications et à analyser le problème avant de se lancer dans l'écriture de son code.

5.1.2 Tests fournis

Dans d'autres missions, des classes de tests contenant tous les tests des méthodes sont fournies à l'étudiant. Bien que cela soit un outil d'une utilité indéniable, notamment car l'utilisation de celles-ci force l'étudiant à implémenter toutes les méthodes, nous doutons fort que les étudiants parcourent ces classes avant d'écrire leur code. Ils procèdent donc le plus souvent par essai/erreur. Nous proposons donc qu'une lecture et analyse de ces classes de test soit organisée préalablement à l'écriture de code.

Le fait que certaines missions ne reçoivent pas ce genre de classes peut laisser penser que le corps enseignant cherche à éviter de donner trop d'indications quant à la direction à suivre pour réussir cette mission. Lorsque c'est le cas, nous proposons alors de fournir non pas le code source des classes de test mais directement les classes compilées qui fourniront lors de leur exécution plus ou moins de renseignement quant à la faute commise. Cela permettra non seulement aux étudiants de pouvoir tester entièrement leur code et de s'assurer qu'ils respectent bien les spécifications.

5.1.3 JUnit

JUnit est le framework de test par excellence et son utilisation fait presque partie des standards de tout développeur Java. La facilité avec laquelle il est possible de faire des tests unitaires est encore renforcée par l'intégration des tests à l'IDE BlueJ, utilisé dans le cadre des missions. Néanmoins, sans l'utilisation de ce plugin, le test des classes est moins évident à réaliser ou en tout cas moins direct, ce qui peut pousser certains étudiants à passer outre l'étape des tests, comme nous avons pu le voir dans l'analyse des missions.

Nous recommandons dès lors que JUnit soit utilisé le plus tôt possible et que les classes de test fournies ou à créer suivent les spécifications de ce framework. Il nous semble en effet que l'introduction de JUnit à la dernière mission fait perdre son intérêt à l'outil, qui pourrait se révéler bien plus efficace s'il était utilisé plus tôt.

5.2 Spécifications

Nous l'avons vu au cours de notre analyse, les spécifications des missions sont souvent ignorées par les étudiants : signature des méthodes ou conventions de nommage non respectées, types de retour erronés, etc. Bien que parfois, ces erreurs ne posent pas de problème, comme dans le cas d'un nom incorrect, cela pose le problème de l'impossibilité d'effectuer des tests automatiques à l'échelle d'une mission au moyen de l'outil présenté ici par exemple.

Ces problèmes surviennent principalement au cours des premières missions, les étudiants n'ayant pas encore assimilé l'importance que des spécifications possèdent en programmation. Nous pouvons envisager plusieurs solutions :

- Fournir les squelettes des classes et des méthodes. Ceci aura pour effet d'aumoins éviter les problèmes de nommage et de signature
- Utiliser autant que possible l'implémentation d'interfaces et de classes abstraites. Néanmoins, cette solution est peu applicable dans les premières missions, ces concepts étant trop avancés
- Tester la présence des méthodes requises dans les tests fournis ou forcer les étudiants à écrire ceux-ci

Certaines missions comportent des spécifications vagues, qui ne comprennent pas de signatures ni de noms de méthodes. Bien que cela fournisse des instructions claires sur le travail à réaliser, les tests automatiques sont alors impossibles. Dans la mesure du possible, nous conseillons dans le cas de ces missions de fournir des spécifications plus explicites.

Si la mission comporte une partie où l'étudiant doit faire preuve d'initiative ou d'imagination, et que le champ libre lui est laissé pour une partie de l'implémentation, les tests automatiques ne sont pas non plus envisageables. Dans ce cas, nous conseillons de quand même spécifier explicitement des méthodes permettant de vérifier la correction du code de l'étudiant. Ces méthodes pourraient être des accesseurs ou des méthodes `toString` avec un format de sortie précis.

5.3 Concepts

5.3.1 Décomposition en sous-problèmes

Dans quelques cas, les algorithmes que les étudiants devaient réaliser se sont révélés trop ardus. On peut penser en particulier au parcours de tableaux ou de `String`. Le concept de décomposition en sous-problèmes n'est pas encore naturel chez les étudiants. Pour beaucoup d'entre-eux, l'analyse préalable du problème n'est pas assez poussée et ils s'attellent directement à la tâche de l'écriture du code. Il en résulte souvent des algorithmes erronés.

Sans vouloir entrer dans les détails des invariants, des conditions `pre-post` et autres, nous conseillons quand même de mettre en avant ce concept qui reviendra

sans cesse dans leurs travaux. Nous pensons qu'il faut forcer son utilisation, même si celle-ci passe inaperçue.

Dès lors nous conseillons si possible, qu'au lieu de demander un travail contenant trois méthodes n'ayant en rapport que les objets qu'elles traitent, de proposer d'écrire des méthodes ayant un lien entre elles. Ces méthodes feraient appel les unes aux autres et la réalisation d'une méthode Y demanderait d'avoir préalablement réalisé la méthode X. De cette façon, en construisant une batterie de méthodes, le problème final de la mission serait de résoudre un algorithme compliqué au moyen de ces méthodes précédentes, appliquant ainsi le concept des sous-problèmes.

5.3.2 Méthodes statiques

Le concept des méthodes statiques apparaît comme étant mal maîtrisé par certains étudiants. On a vu que pas mal d'entre eux avaient recours à l'appel de méthodes statiques sur des objets ou à l'appel de méthodes non statiques depuis un contexte statique.

Ceci n'est pas arrangé par le fait que les premières méthodes qui leur sont demandées soient statiques contrairement aux suivantes dès que le principe des objets est abordé. Cela peut se révéler générateur de confusion pour les étudiants.

La seule façon d'apporter un semblant d'amélioration à ce problème nous semble être l'intégration dans les exercices préliminaires des questions portant sur ce sujet. Des questions du genre "Comment faire un appel de telle méthode dans `main`" permettrait de donner une bonne vision du concept aux étudiants.

5.3.3 null

Les erreurs de pointeurs `null` sont malheureusement courantes dans tous les programmes Java. Ce sont bien sûr des erreurs d'exécution qu'il est impossible de prévoir sans un examen approfondi du code. Pour éviter ce genre d'erreurs, on peut envisager deux solutions :

- des spécifications précisant le comportement à adopter au cas où des paramètres ou des variables seraient `null` ou encore précisant dans les préconditions que le cas `null` ne doit pas être géré
- des méthodes prenant en compte le cas `null` et levant des exceptions ou générant des erreurs ou mieux, n'arrêtant pas l'exécution. Mais dans ce cas, toute la chaîne d'appels doit être prémunie contre `null`

Nous conseillons particulièrement la première approche. Les étudiants, n'étant en effet pas habitués à croiser `null`, n'ont en effet pas le réflexe d'imaginer quel comportement pourrait avoir leurs méthodes dans un cas pareil. Il faudrait donc préciser dans les spécifications des méthodes à implémenter les comportements de celles-ci en particulier pour le retour.

5.3.4 Types

Les analyses ont aussi montré une utilisation excessive et inutile du casting, particulièrement lors de la manipulation de variables de type primitif. Cela montre qu'à un moment dans leur raisonnement, des étudiants perdent le fil de ce qu'ils écrivent et en viennent à ne plus être certains des actions que leur code effectue. C'est évidemment quelque chose à éviter à tout prix.

Malheureusement, ce type d'erreur peut difficilement être corrigé par de l'écriture de code. Dès lors, encore une fois, nous proposons d'intégrer des questions spécifiques dans les exercices préliminaires. Des questions simples portant par exemple sur l'arithmétique dans Java et le type qui en résulte ou des questions plus poussées sur le casting d'objets.

Chapitre 6

Conclusion

Nous allons maintenant terminer par un bref rappel des résultats obtenus, leur comparaison aux objectifs fixés ainsi que par quelques idées de continuation du présent travail.

Résultats et objectifs

Nous avons développé un logiciel tout à fait fonctionnel qui a permis de récolter des données sur les capacités des étudiants à effectuer le travail qui leur est demandé. En plus d'avoir analysé les données de l'année 2012-2013, le logiciel a été mis en production durant l'année 2013-2014 avec des résultats concluants, après une brève période de mise en place. Il s'est révélé être un outil instructif aux dires des tuteurs et des enseignants l'ayant utilisé, ce qui constitue un but atteint parmi ceux qui étaient fixés.

Ce logiciel est maintenant mis à la disposition de l'équipe enseignante pour les années futures. Toute latitude est laissée à celle-ci pour son adaptation et son amélioration.

Les données récoltées avec ce logiciel ont permis de mettre au jour certaines lacunes dans les compétences des étudiants du cours LSINF1101 telles que la décomposition en sous-problèmes, les méthodes statiques, le mot-clé `null` ou les types de variables. Elles ont aussi permis de proposer des améliorations pour les futures missions à savoir une utilisation accrue des tests et des classes JUnit ainsi que des spécifications plus précises bien que permettant suffisamment de liberté dans l'implémentation. Ceci constitue un second but atteint.

Développements futurs possibles

Plusieurs améliorations pourraient s'avérer bénéfiques dans l'optique d'une analyse automatique plus poussée.

Tout d'abord, le logiciel pourrait être développé afin de tenir compte des rapports de PMD d'une manière globale. En effet, pour le moment, seuls des rapports individuels sont disponibles.

De même, on peut parfaitement imaginer l'intégration d'un autre outil d'analyse de code plus complet en ajout de PMD ou l'écriture de règles personnalisées avec celui-ci.

L'utilisation en conjonction avec une plate-forme d'apprentissage pourrait aussi s'avérer très intéressante, l'étudiant recevant en temps réel toute une analyse sur sa production.

On pourrait aussi penser à développer un parser qui permettrait de trouver, selon certains patterns prédéfinis, à quoi sont dues les erreurs à la compilation.

Dans une toute autre optique, il serait également possible d'effectuer de la détection de plagiat en comparant tous les codes soumis mais il nous semble que cela s'éloigne légèrement du but premier.

Mais il nous semble surtout que le plus important reste sans doute le développement régulier du logiciel pour qu'il s'accorde aux changements qu'il a lui même permis d'initier ainsi qu'aux désidératas de ses utilisateurs qui pourront ainsi obtenir toujours plus d'informations et d'analyses utiles à leur tâche d'enseignement de la programmation.

Bibliographie

[1] Javac
`http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/javac.html`

[2] PMD
`http://pmd.sourceforge.net`

[3] Checkstyle
`http://checkstyle.sourceforge.net`

[4] FindBugs
`http://findbugs.sourceforge.net`

[5] JPF
`http://babelfish.arc.nasa.gov/trac/jpf`

[6] JUnit
`http://junit.org`

[7] TestNG
`http://testng.org/doc/index.html`

[8] Suleiman H. Beust C., *Next Generation Java Testing : TestNG and Advanced Concepts*, Addison-Wesley Professional, 1st edition, 2007

[9] Maveryx
`http://www.maveryx.com`

[10] Jtest
`http://www.parasoft.com/jtest?itemid=14`

[11] CodeLab
`http://turingscraft.com`

[12] Pythia
`http://www.csited.be/pythia`

[13] Sébastien Combéfis, Vianney le Clément de Saint-Marcq. Teaching Programming and Algorithm Design with Pythia, a Web-Based Learning Platform. *Olympiads in Informatics*, 6:31-43, 2012

[14] Hurricane Electric
`http://code.he.net/index.php`

[15] Code school
http://www.codeschool.com

[16] Code Academy
http://www.codecademy.com

[17] MissionAnalyzer
https://github.com/basbodart/MissionAnalyzer

[18] Olivier Bonaventure, Charles Pecheur, Damien Leroy, Sébastien Combéfis, *Énoncés des séances d'exercices : Informatique 1*, École Polytechnique de Louvain, septembre 2012

Annexe A

Manuel d'utilisation du logiciel

A.1 Lancer l'analyse

Pour lancer le programme de d'analyse, il faut utiliser la commande *java -jar MissionAnalyzer.jar* suivie d'arguments.

Il y a trois arguments obligatoires :

-d *missionDirectory*
: *missionDirectory* correspondant au chemin vers le dossier de soumissions de la mission à analyser

-m *missionFile*
: *missionFile* correspondant au chemin vers le fichier de configuration de la mission

-t *testFile*
: *testFile* correspondant au chemin vers le fichier de la classe de test JUnit

Les autres arguments, facultatifs, sont :

-v
: Affiche de l'information sur la sortie standard lors de l'exécution

-p [*rulesets***]**
: Active l'utilisation de PMD. *rulesets* correspond aux ensembles de règles [1] à vérifier, séparés par une virgule. (Ruleset par défaut : `java-basic`)

-w *timer*
: Définit le temps en millisecondes au-delà duquel le `Watchdog` tuera le processus de test. *timer* correspond au temps en millisecondes. (Par défaut : 60000)

-e *charset*
: Défini l'encodage des fichiers Java à compiler. *charset* correspond au jeu de caractères utilisé. (Par défaut : ISO-8859-1)

1. Voir `http://pmd.sourceforge.net/pmd-5.1.1/rules/index.html`

-threads *n*
Défini le nombre de threads à utiliser simultanément. (Par défaut : 4)

A.2 Fichier de configuration

Ce fichier doit être un simple fichier de texte comprenant dans sa première partie le chemin vers le dossier de la mission puis sur la ligne suivante le chemin vers le dossier contenant la classe de test. Ceci est nécessaire si l'utilisation du logiciel se fait sur un serveur qui ne fait pas correspondre les chemins avec l'arborescence des répertoires. Si ce n'est pas le cas, void doit apparaitre sur les deux lignes.

Après une ligne vide, doit apparaitre la liste des noms des fichiers obligatoires pour la mission à raison de un par ligne, puis une ligne vide, puis une liste des noms des dossiers des groupes de la mission à analyser à raison de un par ligne également. Concrètement, il doit avoir la forme :

cheminVersLeDossierDeLaMission | void
cheminVersLeDossierDeLaClasseDeTest | void

fichier1.java
fichier2.java

groupe1
groupe2

A.3 Données à analyser

Les soumissions des étudiants doivent se présenter sous la forme d'un ensemble de dossiers et fichiers organisés selon l'arborescence suivante :

- MissionX
  - grpA
  - grpB
    - identifiant1-AAAA-MM-JJ
    - identifiant2-AAAA-MM-JJ
      - fichier
      - fichier

Le dossier principal est bien évidemment celui de la mission. Il a pour nom "Mission" suivi du nombre de la mission (de 1 à 11).

Dans celui-ci sont présents les dossiers des groupes de travail. Ils portent généralement un nom abrégé tiré du nom du local où les travaux pratiques du groupe ont lieu. Il existe aussi un groupe baptisé "noGroup" ainsi que, dans certaines mission, un groupe"staff" qu'évidemment nous n'allons pas tester.

Dans chaque dossier de groupe, on retrouve les dossiers de soumissions des étudiants. Leur identifiant est unique et consiste généralement en leur nom et pré-

nom séparés par un point. Dans certains cas cependant, le prénom ou le nom peut être remplacé par une initiale. Le nom de dossier se termine, après un tiret, par la date de soumission sous format AAAA-MM-JJ (année-mois-jour) qui est le format de norme ISO 8601 pour la date courante. On remarquera qu'il peut exister plusieurs dossiers de soumission pour un même étudiant, si tant est qu'il a soumis son travail à des dates différentes. La logique voudra bien sur que l'on analyse uniquement la soumission la plus récente, sensée contenir le travail le plus abouti, ce qui sera le cas.

Dans ces dossiers d'étudiants se trouvent différents fichiers, dont supposément au moins un contient le code écrit par l'étudiant. Ce fichier se doit de posséder un nom particulier, défini dans le fichier de configuration de la mission. On peut aussi retrouver d'autres fichiers dont certains sont nécessaires au fonctionnement du programme de l'étudiant.

Attention !

Si pour une raison quelconque, cette structure venait à être modifiée d'une façon ou d'une autre, le fonctionnement du logiciel d'analyse ne serait plus garanti. Des modifications devraient lui être apportées afin de l'adapter à la nouvelle structure. Les deux points critiques qui se doivent d'être respectés sont : l'arborescence Mission -> Groupe -> Étudiant -> Fichier et le format du nom de dossier des étudiants "nom-AAAA-MM-JJ"

A.4 Fichier sorttable.js et fichiers cachés

Le fichier `sorttable.js` doit se trouver dans le même dossier que le fichier `.jar` sous peine de créer une erreur. De même, la présence de fichiers ou dossiers cachés dans l'arborescence est susceptible de créer des erreurs, il faut donc s'assurer que ce ne soit pas le cas.

Annexe B

Liste et références des librairies nécessaires à MissionAnalyzer

- commons-cli, commons-exec, commons-lang
  `http://commons.apache.org`
- junit
  `http://junit.org`
- hamcrest-all
  `http://hamcrest.org`
- gagawa
  `http://code.google.com/p/gagawa`
- gson
  `http://code.google.com/p/google-gson`
- pmd
  `http://pmd.sourceforge.net`

Annexe C

Liste et références des librairies nécessaires à PMD

- ant, ant-launcher, ant-testutil
  http://ant.apache.org/antlibs/proper.html
- asm
  http://asm.ow2.org
- commons-io
  http://commons.apache.org/proper/commons-io
- dom4j
  http://dom4j.sourceforge.net
- javacc
  https://javacc.java.net
- jaxen
  http://jaxen.codehaus.org
- jcommander
  http://jcommander.org
- jdom
  http://www.jdom.org
- rhino
  https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino
- saxon, saxon-dom
  http://saxon.sourceforge.net
- xercesImpl, xml-apis, xmlParserAPIs
  http://xerces.apache.org
- xom
  http://www.xom.nu