KOPI Cross-Referencer

Approche méthodologique

Bertrand Blanc

Abdelaziz Boushabi

Hicham Idrissi Khamlichi

06 juin 2001

Abstract

In the second semester of ESSI II, we had to realize a project. We choosed to work with M. Graf on one titled KOPI cross–referencer. Java developpers have encountered problems to pack and to distribute their products. They have written a lot of packages which could depend on each other. Thanks to our product kXr they need not themselves to choose packages to include any more. Thus, kXr counts the number of classes and members encountered. Used packages are the ones which have been visited at least one time. Then, our project will help a lot of developpers who ask themselves these questions about dependances between packages.

In order to develop as fast as possible with the best chances to reach our objectives, we choosed to adopt a quality approach. It describes point by point each step in order to be as clear as possible in the life cycle of the software. Main points could be represented by key words: what, how, why, do it, test and use.

Table des matières

Table des figures

Chapitre 1

Présentation

Introduction

Le second semestre d'ESSI II propose aux étudiants de réaliser un *Projet* sur un sujet proposé et encadré par un enseignant. Il a pour but d'introduire le stage de fin d'année s'effectuant en entreprise pour sensibiliser d'une part les étudiants aux demandes de l'Entreprise et d'autre part appliquer la théorie et les langages appris dans les différents modules de l'année.

Quel programmeur JAVA ne s'est jamais demandé si toutes les classes ou méthodes qu'il a développées, étaient toutes utiles ? De même, pour distribuer un logiciel en archive JAR, tous les packages nécessaires ont-ils été inclus ; ou bien sont-ils tous utilisés ? Pour essayer de répondre à cette demande, un projet officiel GNU a été proposé, soutenu par la société DMS.

Intéressés par ces questions, nous avons choisi de travailler sur **un système de références croisées** prenant en entrée un ensemble de fichiers de classe dont les fichiers sources peuvent être écrits autant en JAVA qu'en SCHEME ou autre. Monsieur Graf, représentant de la société DMS, fut notre encadreur et nous soutaint vivement tout au long de nos choix.

Le but du projet est de fournir une solution à cette demande. Le travail se découpe en deux phases maîtresses :

- Analyse du contexte et de l'existant en fonction du cahier des charges, des spécifications et des contraintes. A l'issue de cette analyse, nous avons proposé une solution implantable.
- Réalisation des outils. Le but est de mettre en oeuvre les différentes étapes nécessaires à la production de ceux-ci.

1.1 Motivation

1.1.1 Que proposons-nous ?

L'objectif de notre projet est de concevoir un logiciel qui indiquera le nombre de points de **référencement** d'un package, interface ou classe ainsi que de ses membres (champs, méthodes) dans un ensemble de fichiers de classe obtenus par compilation en code JVM de fichiers *Java, Ada, Scheme, ...* De ceci découle deux principales questions :

1. quel est l'intérêt de connaître le référencement d'une classe ?
2. pourquoi travailler sur le code compilé et non sur le code source ?

1.1.2 Quelles sont les utilités

Pour exposer l'utilité de notre projet, nous allons répondre aux deux questions évoquées dans le paragraphe précédent :

1.1.3 Pourquoi connaître les référencements ?

Il est important de connaître le référencement d'une classe et ce pour au moins deux raisons :

1. supposons que l'on ait des dizaines de fichiers contenant plusieurs centaines voire milliers de lignes de codes et que l'on souhaite modifier ou bien même supprimer une méthode d'une certiane classe. Il est dans ce cas indispensable de connaître à quel endroit des autres fichiers cette méthode est utilisée pour pouvoir tenir compte des conséquences de cette modification. Notre outil va donc permettre à l'utilisateur d'éviter de chercher ligne par ligne et fichier par fichier l'emploi de cette fonction.

2. supposons maintenant que l'on ait défini plusieurs packages et qu'ils soient interdépendants c'est-à-dire que certains packages utilisent des classes définies dans d'autres packages. Supposons qu'un client n'ait besoin que des services offerts par une compilation de deux packages par exemple. Le problème est alors de savoir si ces deux packages peuvent être fournis seuls au client ou si nous devons lui fournir en plus d'autres packages auquels se réfèrent les deux premiers. C'est là qu'intervient donc l'utilité de notre outil qui va pouvoir détermier si les deux premiers packages font référence à d'autres packages ou non.

1.1.4 Pourquoi travailler sur le code compilé ?

L'intérêt de travailler sur le code compilé est d'augmenter le nombre de clients potentiels. En effet, il existe des compilateurs utilisant la *machine virtuelle Java* qui permettent de transformer n'importe quel langage (tel que Scheme, ...) en du code *Java* **compilé**. On pourra donc permettre aux programmeurs d'autres langages utilisant la *machine virtuelle Java* d'employer notre produit alors que si l'on restraignait notre outil au simple code **source** *Java*, on aurait par conséquent restreint notre clientèle aux seuls programmeurs *Java*.

Notre outil est indispensable ; chaque programmeur — utilisant la *machine virtuelle Java*— doit impérativement l'avoir fourni dans sa suite logicielle de développement.

1.2 Assurance qualité

Le plan d'assurance qualité, présenté en Annexe A, permet d'assurer que le travail que nous avons réalisé entre dans le cycle de vie du produit. Il décrit les phases successives allant de l'analyse à l'intégration en précisant les procédures opératoires de chaque étape. En outre, son rôle est de :

i. décrire les étapes de développement à suivre

ii. indiquer les procédures à appliquer en cas d'erreur

iii. indiquer les conditions de passage ainsi que les documents à fournir entre chaque étape

Notre travail s'appuie bien évidemment sur une série de critères, de souhaits et de contraintes requis par le client, ici M. Graf, et définis dans le cahier des charges (Annexe B).

1.3 Analyse et conception

L'analyse de l'existant a montré qu'un logiciel de référencement, présenté en Annexe C, existe. Il prend cependant en entrée un ensemble de fichiers sources écrits en JAVA. Notre approche, axée sur le byte-code JVM, permet en outre de s'abstraire de ce langage.

D'après les contraintes dues à la spécificité du byte-code et à l'utilisation de la bibliothèque **classfile** proposée par le projet **KOPI** de M. Graf, deux axes de conception sont apparus. L'un proposant une approche top-down et l'autre plutôt bottom-up :

- Partie externe : elle propose une analyse et la conception du système externe, qui s'appuira sur une couche intermédiaire de plus haut niveau que la bibliothèque **classfile**
- Partie interne : elle présente cette couche intermédiaire permettant une vue de plus haut niveau du byte-code.

L'annexec C présente le dossier d'analyse et de conception de la partie externe du système. En outre, les entrées ont été précisement déterminées :

i. Un ensemble de fichiers de classe
ii. Un ensemble de méthodes initiatrices permettant l'accession aux fichiers
iii. Une vue qui assure la cohérence de l'initiation sur l'ensemble des fichiers

1.4 Implantation, Intégration et Tests

Les choix d'implantation — tels que les diagrammes UML, les schémas composites ou fabriques abstraites — de la partie externe du système sont com-

mentés en Annexe D. Ceux de la partie interne sont présentés en Annexe E. Le langage d'expression fut JAVA car il offrait tout d'abord le style de programmation orientée objet requis par nos choix et de plus un compilateur[1] nous était fourni par **KOPI**.

L'intégration des deux parties est le point phare du projet car il détermine le suivi des contraintes définies par tous les documents réalisés jusqu'alors. Cependant, la question tant attendu arrive : est-ce que ça marche ?

Les tests[2] déterminés pendant la phase de création du cahier des charges vont nous permettre de répondre à cette question. Bien entendu, ils montrent que le prototype réalisé peut marcher. Néanmoins, une série de tests de plus grande envergure — avec détermination d'un protocole de benchmarking — est nécessaire.

Conclusion

Ce travail a tout d'abord abouti à la création d'un outil de référencement croisé. Cet outil et plus précisement cette approche fondée sur les fichiers de classe est nouvelle et prend place dans une niche qui n'est pas encore très bien fournie. Il présente le gros avantage de reposer sur deux standards de leur domaines respectif : le langage JAVA pour l'expression de sa partie implantatoire et le byte-code JVM pour ses entrées. Les informaticiens ont enfin à leur disposition un outil — faisant apparaitre les méthodes parasites ou les packages à inclure — simple, puissant et adapté à leur besoin.

Le respect des consignes imposées par le plan d'assurance qualité ont amené à la réussite de la maquette de notre produit. Les critères majeurs de modularité, réutilisabilité et gradabilité en font un produit évolutif permettant d'ajouter de nouvelles fonctionnalités telles que la gestion des archives JAR. Evidemment,

[1] kjc
[2] Annexe F

le processus avant-déploiement demande un peu plus de travail de l'ordre de quelques semaines, permettant la validation de l'outil. Nous espérons, maintenant, que notre respect des normes de qualité permette l'évolution de notre projet et non sa disparition ou sa refonte.

Sur un plan plus personnel, nous avons appris de très nombreuses notions durant ce projet. La plus importante est la notion de travail personnel et d'auto-apprentissage. Nous avons aussi évolué dans le domaine de la gestion de projet. Cela a été une nécessité pour éviter de finir sur un échec. Nous connaissons mieux les enjeux et les techniques qui entourent ce domaine.

Annexe A

Plan d'assurance qualité

A.1 But, domaine d'application et responsabilité

Ce plan d'assurance qualité concerne le développement d'un logiciel permettant de déterminer les références croisées d'un ensemble de fichiers respectant le langage du bytecode de la *machine virtuelle Java*. Ce logiciel est développé dans le cadre du projet de second semestre en seconde année de l'ESSI.

La réalisation de ce plan d'assurance qualité est assurée par M. Boushabi. Le suivi de ce plan sera réalisé par celui-ci et il prend donc comme responsabilité de le suivre et de le faire suivre à la lettre et dans ses moindres détails.

Au cours de différentes discussions avec M. Graf, nous sommes arrivés à dégager des facteurs principaux de qualité auxquels le présent projet devra répondre. Ces derniers sont l'évolutivité et la maintenabilité. Ces deux facteurs impliquent donc une série de critères qui pourront permettre, grâce à une métrique de juger de la qualité du produit final. Ces critères seront décrits plus loin dans ce plan.

Afin de présenter une bonne flexibilité, ce plan qualité, qui doit être validé par le tuteur de ce projet qui est M. Graf, contient une procédure d'évolution décrite ci-après :

- identification du paragraphe du plan d'assurance qualité à modifier.
- rédaction du texte correspondant à l'évolution.
- validation de ce nouveau texte par le responsable de la qualité.

– ajout de ce texte sous forme d'un addendum au plan qualité.

En cas de non application de ce plan, il est prévu de suivre la procédure suivante :

– identification des points qui ne sont pas respectés.
– étude par les responsables de la qualité des moyens à mettre en œuvre pour remédier à ce non respect.
– s'il est possible d'y remédier alors présentation de la solution au tuteur.
– s'il est impossible d'y remédier, ou si le tuteur ne considère pas la procédure comme viable, alors le responsable de la qualité devra joindre qu dossier de réalisation une feuille de non respect de la qualité précisant le ou les points qui ne sont pas respectés, les raisons de ce non respect et la ou les conséquences sur le produit final.
– si une solution a été trouvée pour remédier au problème alors mise en œuvre de cette solution de rédaction d'une fiche d'incident de la qualité précisant la nature de la non application et le remède employé.

A.2 Documents applicables et documents de référence

A.2.1 Documents applicables

– API Java du site java.sun.com

A.2.2 Documents de référence

– **Java Language Specification**[1] du site java.sun.com par James Gosling, Bill Joy et Guy Steele.
– **Java Virtual Machine Specification**[2] du site java.sun.com par Tim Lindholm et Frank Yellin.
– API du package **at.dms.classfile**

[1] que l'on notera désormais *JLS*; cf. Abrévéations
[2] que l'on notera désormais *JVMS*; cf. Abrévéations

A.3 Terminologie

A.3.1 Abrévéations

- *JLS* (*Java Language Specification*) désigne les spécifications du langage *Java JDK 2.*
- *JVMS* (*Java Virtual Machine Specification*) désigne les spécifications de la machine virtuelle de *Java*, version 2.
- *API* est ...

A.4 Organisation

A.4.1 Intervenants du projet

Ce projet est réalisé par le trinôme composé par Bertrand Blanc, Abdelaziz Boushabi et Hicham Idrissi Khamlichi[3] Il est encadré et tutorié par Docteur Thomas Graf, enseignant à l'Ecole Supérieire en Sciences Informatique.

A.4.2 Rôles et missions

M. Graf a pour rôle d'encadrer et de guider ses élèves dans leur choix. Ses conseils interviennent à deux niveaux :

- Au niveau de la définition de l'outil de *cross-referencer* ainsi que de son mode de fonctionnement.
- Au niveau de la conception globale du projet, car son recul vis-à-vis de celui-ci est plus important et ses remarques forcément pertinentes.

Il peut intervenir quand il est solicité par ses élèves ou bien quand il ressent que le projet ne va plus dans le sens où il l'entend. Il doit exprimr ses besoins dans pendant la conception du cahier des charges.

Trinôme Blanc-Boushabi-Idrissi sont les personnes chargées de la réalisation du projet. Ils doivent produire les outils et toute la documentation nécessaire relativement à ce plan d'assurance qualité. Ils doivent donc concevoir le programme, mettre le programme en service et produire toute la documenta-

[3]le trinôme sera désormais désigné par Blanc-Boushabi-Idrissi.

tion nécessaire à la compréhension du développement et du fonctionnement du produit final. Ils doivent tenir compte des remarques de M. Graf.

A.4.3 Liens hiérarchiques et fonctionnels

Le trinôme Blanc-Boushabi-Idrssi est hiérarchiquement lié à M. Graf. par le fait qu'il est son encadreur. Il doit donc leur apporter ses avancements et tenir compte de leurs remarques et suggestions. Il est important qu'il exprime clairement au trinôme les objectifs à atteindre.

Le lien fonctionnel est plus évident : le trinôme Blanc-Boushabi-Idrssi a besoin de l'expérience et du savoir de M. Graf.

A.5 Démarche de développement

Cette section fait référnce au plan de développement (Figure A.1).

A.5.1 Analyse préalable

C'est la première phase dans le développement, elle est démarée dès le début du projet.

Activités

Nous allons axer nos recherches sur deux domaines distincts

Analyse de l'existant : cette phase a pour but de chercher ce qui a déjà été fait dans les domaines qui nous intéressent. Des recherches seront effectuées portant sur les outils qui pourraient répondre, en partie ou totalement, au problème posé par conception des systèmes de références croisées. Cf. la partie en annexe :*Analyse de l'existant.*

Analyse des besoins : demande d'informations complémentaires auprès de M. Graf concernant les besoins relatifs à ce système de références croisées. Enfin, nous définirons un utilisateur type du système de réferences croisées.

Points d'entrée nécéssaires

Le sujet des références croisées doit être rédigé et par conséquent fixé.

Documents réalisés

Il sera édité un document portant sur les besoins et les contraintes relatifs à la définition et à la conception du système de référencement. Ce document sera inclus au cahier des charges. Les résultats portant sur l'analyse de l'existant seront joints au présent document sous forme d'une annexe.

Condition de passage à la phase suivante

La phase suivante sera accessible si les conditions suivantes sont remplies :

1. Les besoins sont analysés.
2. Les documents décrits précédemment sont réalisés.
3. Les documents d'analyse et des contraintes à inclure dans le cahier des charges doivent être validés par le tuteur.

A.5.2 Mise en œuvre du cahier des charges

Cette phase commence apès la phase d'analyse préalable.

Activités

Les contraintes et les besoins ayant déjà été analysés lors de la phase précédente, le travail de cette phase se trouve quelque peu réduit. Il s'agit principalement de rédiger avec le concours de M. Graf, un plan de recette concernant l'ensemble logiciel.

Points d'entrée

Les documents d'anlyse des besoins et des contraintes réalisés lors de la phase précédente.

Documents réalisés

Un plan de recette qui se présentera, étant donné que le sytème de références croisées n'est pas encore défini, sous la forme de description de problèmes que le système est sensé résoudre. A la fin de cette phase, le cahier des charges sera terminé et rédigé. Il sera inclus au présent document.

Condition de passage à la phase suivante

Il est nécéssaire que le cahier des charges soit rédigé et validé par le tuteur avant de passer à la phase suivante.

A.5.3 Approches de l'anlyse conceptuelle

Cette phase prend place juste après la phase de création du cahier des charges.

Activités

Au cours de l'analyse conceptuelle du système, nous avons pu adopter deux types d'approches.

1. L'approche ***descendante*** : on décompose le système en sous-systèmes, chacun d'eux pouvant être ensuite redécomposée jusqu'à l'obtention de modules programmables "simplement". Le système serait donc représenté comme une boîte noire renfermant d'autres boîtes noires auxquelles sont associées des entrées et des sorties connues. Dans le cas présent, il s'agira de déterminer quelles sont les entrées et sorties du système de référencement et de définir ses sous-parties ainsi que leurs interactions.
2. L'approche ***ascendante*** : on part de modules déjà existants que l'on essaie de composer entre eux. Dans le cas présent, il s'agira de se servir des modules du package `at.dms.classfile` pour pouvoir manipuler et extraire les informations nécessaires des fichiers compilés.

Ces deux approches nous ont donc conduits à élaborer l'analyse de la partie interne et de la partie externe du système de façon parallèle dans un premier temps puis de façon complémentaire dans un second temps. Il s'agira donc d'identifier les modules autonomes et de décrire leurs interactions.

Points d'entrée nécéssaires

Cette partie dépend du cahier des charges.

Documents réalisés

Un document qui identifiera les modules déduites des approches ainsi que leurs interactions.

Conditions de passage à la phase suivante

Cette phase ne sera achevée que lorsque le rôle et les interactions des modules auront été cernés.

A.5.4 Partie interne du système

Spécification interne du système

Cette phase prend place juste après la phase concernant les approches d'analyse conceptuelle.

Activités

Il s'agit de définir la spécification interne du sytème ainsi que sa fonctionnalité, c'est-à-dire la manière d'extraire les informations contenues dans les fichiers compilés qui seront nécessaires au fonctionnement du système.

Points d'entrée nécéssaires

Cette partie dépend du résultat de l'approche ascendante.

Documents réalisés

Une annexe sera rédigée et donnera les spécifications internes du système.

Conditions de passage à la phase suivante

La phase suivante ne sera accessible que lorsque ces spécifications seront clairement définies.

Définition de la structure adoptée

Cette phase suit la spécification de la spécification de la partie interne.

Activités

Au cours de cette période, on spécifiera l'architecture de la partie interne qui sera adoptée pour répondre aux spécifications correspondantes. On établira également l'API des fonctions permettant d'obtenir cette structure. Des tests seront effectués pour vérifier la validité de la structure.

Points d'entrée nécéssaires

Il faut avoir à sa disposition toutes les spécifications de la phase précédente.

Documents réalisés

Un document sera réalisé expliquant les choix retenus dans cette architecture ainsi que les avantages et les inconvénients de celle-ci. On fournira également l'API des fonctions.

Conditions de passage à la phase suivante

Pour le passage à la phase suivante, l'architecture de la partie interne devra être définie. Elle devra avoir été soumise à critique auprès du groupe et de l'encadreur afin que des remarques puissent être prises en compte. L'API devra quant à elle être complète et commentée.

Implémentation de la partie interne

Cette phase suit la phase de définition de la structure adoptée.

Activités

Il s'agit d'implanter les modules en suivant l'API déterminée dans la précédente phase. On testera bien que les modules implantés respectent bien leurs fonctionalités.

Points d'entrée nécéssaires

Tous les documents produits dans la définition de la partie interne y compris l'API.

Documents réalisés

Pour chaque module implanté, une documentation devra être fournie précisant le rôle de chaque élément du programme.

Conditions de passage à la phase suivante

La phase suivante ne sera accessible que tous les modules seront implantés, toute la documentation sera produite et que tous les tests auront été réalisés.

A.5.5 Partie externe du système

Spécification externe du système

Cette phase prend place juste après la phase concernant les approches d'analyse conceptuelle.

Activités

Il s'agit de produire compte tenu du cahier des charges, les spécifications externes du système de référencement. Nous allons donc décrire les fonctionnalités du système.

Points d'entrée nécéssaires

Cette partie dépend du résultat de l'approche descendante.

Documents réalisés

Une annexe sera rédigée et donnera les spécifications externes du système. Ces spécification seront figés pour tout le reste du projet.

Conditions de passage à la phase suivante

Les spécifications du programme seront présentées aux tuteurs et devront être soumises à critique. Si ces spécifications ne se révellent pas satisfaisantes alors le délégué devra revenir sur celles-ci pour en proposer d'autres. On ne pourra clôturer cette phase que si les spécifications sont validées par l'encadreur.

Spécifications des composants du système

Cette phase commence après la spécification du système.

Activités

après la spécification du système, on décompose le système en sous-parties et on détermine leurs fonctionnlités mutuelles au sein du système. Après cette décomposition, des tests seront effectués afin de déterminer si la décomposition permet d'analyser correctement les différents aspects de programme.

Points d'entrée nécéssaires

Les spécifications du système.

Documents réalisés

Un document expliquant les choix, leurs avantages et leurs inconvénients, notamment en ce qui concerne l'évolutivité du système, sera attaché à celui-ci. La liste des fonctionnalités de chaque composant de notre système feront l'objet d'une annexe à laquelle se réfèrera le document précédent. Ces documents contiendront des figures pour présenter l'architecture. Chacun fera l'objet d'un chapitre.

Conditions de passage à la phase suivante

Cette phase ne s'achèvera que lorsque la fonctionnalité des diférents composants aura été définie. Il devra avoir été présenté au tuteur qui devra le critiquer. Les remarques devront être étudiées afin de définir si quelque chose peut-être ajouté ou supprimé. Après ces éventuelles modifications, la prochaine phase pourra débuter.

Description de l'algorithme

Cette phase suit la phase de spécifications des composants du système.

Activités

Il s'agit de définir un algorithme qui permettra de résoudre les références. Cet algorithme devra être d'une complexité raisonnable et devra tenir compte de la taille possible des fichiers qui pourront être analysés. Des tests seront réalisés pour vérifier la validité de l'algorithme.

Points d'entrée nécéssaires

Les spécifications des composants du système.

Documents réalisés

Un document expliquant les choix pris pour cet algorithme ainsi que ses avantages et inconvénients. On ajoutera également au document un descriptif du principe de l'algorithme

Conditions de passage à la phase suivante

Pour passer à la phase suivante, le principe de l'algorithme devra être défini. Il devra être, en outre, avoir été soumis à critique auprès du tuteur afin que des remarques puissent être émises pour la dernière fois. Ces remarques devront être discutées.

Description de l'architecture

Cette phase ne pourra débuter que lorsque la description de l'algorithme sera achevée et que la définition de la structure adoptée au niveau de la partie interne sera terminée.

Activités

Chaque composant du système identifié lors de la conception devra être structuré dans la syntaxe du langage Java. Il en sera de même pour l'algorithme. Des tests seront effectués pour vérifier que les fonctionnalités ont pu

être préservées.

Points d'entrée nécéssaires

- Les composants du systèmes doivent être définis.
- L'algorithme doit être déterminé.
- La définition de la structure adoptée au niveau de la partie interne doit être terminée.

Documents réalisés

Un document expliquera les choix retenus concernant les structures offertes par le langage Java, en indiquant les avantages et inconvénients de ces choix. Ce document décrira également les structures offertes par le langage Java –spécifiées au format UML– qui ont été retenues.

Conditions de passage à la phase suivante

Pour passer à la phase suivante, l'architecture devra être définie. Elle sera soumise à critque auprès de l'encadreur dans le but d'apporter d'éventuelles améliorations. Après ces éventuelles mises à jour, la prochaine phase pourra débuter.

Implémentation de la stucture

Cette phase suit la phase de définition de l'architecture du système.

Activités

Il s'agit d'implanter l'architecture du système et de ses composants. Il faudra définir un plan de recette, une écriture, une vérification et une documentation du code.

Points d'entrée nécéssaires

Tous les documents produits dans la définition de l'architecture serviront de base à ce travail.

Documents réalisés

Pour chaque module implanté, une documentation devra être fournie.

Conditions de passage à la phase suivante

La phase suivante ne sera accessible qu'au moment où tous les modules seront implantés, toute la documentation sera produite et où ils auront tous été testés.

A.5.6 Intégration et mise en service du logiciel

Cette phase débute quand tous les modules sont implantés, documentés et vérifiés.

Activités

Intégration de chaque module pour produire le système de référencement croisé. Tests sur le système. Debogage. Vérification du fonctionnement sur au moins un exemple réel. Présentation du résultat à l'encadreur.

Points d'entrée nécéssaires

Tous les modules implantés.

Documents réalisés

Le système de référencement sera décrit au travers de son manuel d'utilisation, joint au présent document. Enfin, des notes à l'attention des programmeurs

décriront les éventuelles particularités du code et les manières de faire évoluer le programme.

Condition de fin de projet

Toute la documentation devra être produite et le logiciel mis en place et prêt à fonctionner. Le produit final devra être montré au tuteur. Le présent document sera bouclé.

Toutes les étapes du développement du logiciel sont récapitulées par la figure A.1 en fin de ce présent document.

A.6 Gestion des modifications

A.6.1 Modifications dûes à des erreurs

Les procédures à suivre sont différentes selon le cas . Voici la description des procédures à suivre en cas d'erreur au niveau de :

Spécifications des programmes : c'est un type d'erreur assez grave, deux cas peuvent se présenter. Si l'erreur est découverte avant la phase d'écriture des modules, alors on peut encore y revenir si une décision est prise en ce sens dans une réunion du trinôme. Si l'erreur est découverte après la phase de conception de l'architecture alors aucune modification ne sera permise. Dans tous les cas, une fiche de détection d'erreur devra être jointe au présent document ; si une solution a été mise en œuvre alors elle fera partie de cette fiche. Cette fiche montrera les conséquences de cette erreur sur le produit final.

A.6.2 Modifications pour évolution

Comme nous l'avons décrit précédemment, les spécifications du système de références croisées sont figées juste après le cahier des charges. Une évolution proposée après cette phase sera refusée sauf si lévolution est mineure[4] et alors

[4]Une évolution sera qualifiée de mineure si elle n'entraîne pas de modifications dans les spécifications du système.

l'évolution sera mise en œuvre ainsi que la rédaction d'une fiche évolutive qui la décrira. Cette fiche devra identifier le point qui a subit une évolution, préciser si cette évolution concerne une fonction existante ou entraîne une création, et enfin décrire cette évolution (choix, avantages, inconvénients).

A.7 Méthodes, outils et règles

A.7.1 Méthodes et outils

On ne va imposer que trois outils :

1. Les programmes utiliseront le package **at.dms.classfile** qui permet de manipuler les fichiers de classe d'extension `.class`.
2. Chaque document sera rédigé sous la forme d'un fichier LaTeX 2_ε.
3. Les fichiers devront être édités à l'aide du logiciel *XEmacs*.

A.7.2 Règles et normes à respecter

Les seules règles et normes que nous allons imposer dans ce plan d'assurance qualité concernent l'implémentation. Tout d'abord, le langage utilisé pour implanter le système de référencement sera le langage *Java*. Chaque classe fera l'objet d'un fichier `.java`. Il y aura, dans les fichiers d'implémentation au moins un commentaire toutes les cinq lignes. Chaque entête de fonctions devra être remplies et devra respecter les conventions de code Java[5]. La `JavaDoc` devra être générée. Chaque fonction citée devra être commentée par les informations suivantes :

- description du rôle de la fonction
- description des paramètres en entrée
- description des valeurs retournées

On indiquera également le rôle de chacune des classes ainsi que des attributs.

[5]cf. le site `java.sun.com` à ce sujet

A.7.3 Reproduction, livraison

Ce logiciel est sous license GPL[6]. La distribution de ce logiciel se fera sous la forme d'une archive *.tar.gz*[7] qi est un format très répandu sur les systèmes *UNIX*. Cette archive contiendra les sources et le logiciel devra donc être compilé grâce à un '*Makefile*' qui sera obtenu à l'aide d'un fichier '*configure*'. Un fichier *README* devra être présent dans l'archive, il expliquera succintement l'installation du logiciel ainsi que son mode d'utilisation.

A.8 Suivi de l'application du plan qualité

Nous allons ici décrire les moyens mis en œuvre par le responsable du plan qualité (cf. §1) pour maîtriser la qualité tout au long du cycle de production.

Le responsable vérifiera par lui-même si toutes les conditions de passages entre deux phases sont respectées. Il vérifiera pour chaque module implanté s'il correspond à la norme décrite en §7.2. Il contrôlera la qualité de tout document produit qui doit s'intégrer au présent document ; pour ce faire, il vérifiera que le document remplit les critères énoncés dans la description des documents produits de la phase auquel il se rapporte (§5). La vérification sera donc constante et ainsi elle permet de penser que le produit final respectera ce plan qualité.

Pour donner encore plus de crédibilité au suivi de ce plan qualité, l'encadreur est invité, quand il le désire à vérifier son suivi en demandant des comptes au responsable de la qualité. Comme nous l'avons décrit dans la section 1, certains critères permettent d'obtenir les facteurs de qualité attendus. Nous allons donc décrire ici les différentes procédures à suivre pour vérifier le suivi de ces critères. Au vu des résultats observés quand on applique une des procédures suivantes, nous pouvons savoir si oui ou non la qualité est respectée.

Traçabilité : L'encadreur choisira un point particulier de ce plan et le responsable devra être capable de leur fournir tous les documents et/ou produits qui ont déjà dû être réalisé à ce point.

[6] cf. la publicaton GPL en annexe.

[7] Cette archive est obtenue avec le logiciel *tar* et compressée avec le logiciel *gzip*.

Réutisabilité : L'encadreur, ou les membres du trinôme, devront vérifier certains modules implantés. Ces derniers devront être autonome. Il sera observée la plus grande attention sur la qualité de l'éventuelle encapsulation des méthodes qui sont des marqueurs clairs de réutisabilité.

Modularité : Les auditeurs pourront vérifier que l'implémentation se fait en modules. Des modules qui doivent être documentés et, dans la mesure du possible, vérifiés. Le projet final doit être composé de différents modules compilés séparément.

Lisibilité : On pourra mesurer la lisibilité en étudiant tout d'abord l'indentation des fichiers des programmes. Ensuite, la densité de commentaires devra être en accord avec celle préconisée en 7.2, mais on pourra par ailleurs vérifier leur sémantique et donc leur utilité.

Expansibilité : L'expansibilité peut s'observer à deux niveaux différents. Dans un premier temps, au niveau de chaque module ; chacun devra permettre d'être améliorés en ayant le moins de répercutions possibles sur le reste du programme. Ensuite, au niveau de l'application dans son intégrité. Il s'agira de la faculté de l'architecture logicielle à être améliorée en rajoutant plus de fonctionnalités.

Cohérence : La cohérence se fera par construction. C'est-à-dire que puisque toutes les étapes jusqu'à celle de la conception, sont validées par l'encadreur, on peut être assuré que l'ensemble des deux modules sera cohérent. On pourra aussi vérifier que tous les documents produits sont aussi cohérents.

Si une défaillance est décelée alors le responsable de la qualité devra tout mettre en œuvre pour combler ce manque, quitte à stopper le déroulement du processus de développement ; il sera de plus rédigé une fiche de défaillance ponctuelle de la qualité qui sera joint à ce document dans une annexe.

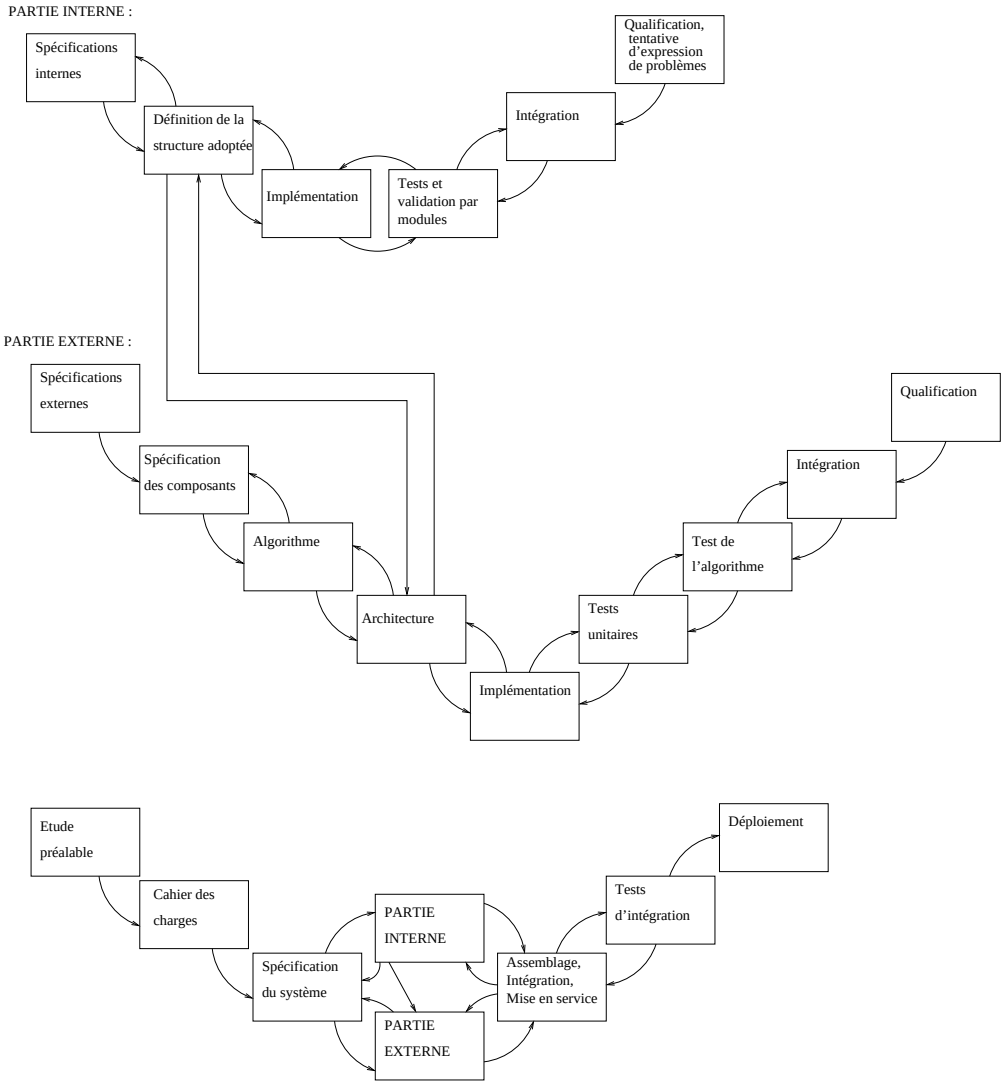

FIG. A.1 – Plan de développement du logiciel

A.9 Extensions

Les exemples d'extension ci-après ne sont cités qu'à titre indicatif et ne feront pas partie du cadre de notre projet. Les nouvelles fonctionnalités que l'on peut ajouter à la partie obtenue précédemment qui est le noyau sont :

Amélioration de l'interface : On pourrait réfléchir à une présentation des résultats qui soit conviviale. On pourrait également proposer au client plusieur types d'affichage des résultats tel que des tableaux ou des graphes par exemple.

Options de référencement : On pourrait offrir à l'utilisateur la possibilité d'affiner ses recherches ou au contraire de les généraliser.

Pour la réalisation de ces extensions, on adopte un schéma incrémental représenté par la figure A.2, c'est-à-dire que l'on part du noyau qui est opérationel dans le but de l'améliorer en plusieurs étapes.

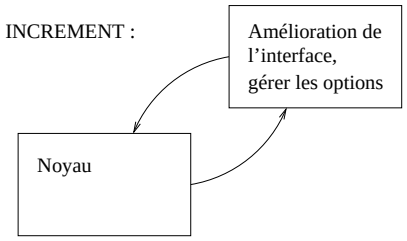

FIG. A.2 – Extension du logiciel

Annexe B

Cahier des charges

Durant la phase d'analyse préalable, nous avons pu identifier les besoins et contraintes liés à chaque partie du projet. Ces derniers seront à la base de tout notre travail. Ils vont nous être extrêmement utiles lors des phases de conception.

B.1 Besoins

Le principal besoin qui a été à l'origine du lancement de ce projet est la facilité à repérer les points de référencements d'un fichier de classe Java.

Il doit donc permettre de vérifier facilement le point de référencement d'un membre, c'est-à-dire que le résultat doit permettre de connaître directement le référencement. L'autre besoin est qu'il soit informatif, c'est-à-dire qu'il n'a pas à modifier les entrées ; dans notre cas, les fichiers de classe ne doivent pas être modifiés ou supprimés. L'outil doit pouvoir traiter des fichiers de taille énorme.

B.2 Contraintes

La principale contrainte liée à l'outil est qu'il ne peut être utilisé que sur des fichiers de **classe Java**. L'outil n'est donc pas sensé analyser du code source Java ou un code source écrit dans un autre langage. L'outil doit également être monoutilisateur. Il est également monotâche : l'outil ne traitera qu'un seul problème à la fois. Il est indépendant dans le sens qu'aucun autre logiciel n'est

indispensable à son bon fonctionnement.

B.3 Utilisateur type

L'outil doit pouvoir être utilisé par un programmeur utilisant la *machine virtuelle Java*, c'est-à-dire qu'un programmeur d'un quelconque langage qui se sert d'un compilateur transformant son code en du code Java compilé par l'intermédiaire de la *JVM*. Ce logiciel n'est donc pas destiné ni au grand public ni à des informaticiens ne manipulant pas de code compilé Java. Cet outil a une fin purement technique et constitue un accessoire au programmeur Java.

B.4 Plan de recette

cf. la partie **étude de cas**.

B.5 Conditions relatives à la qualité

Dans le plan d'assurance qualité que vous trouverez dans l'annexe A, nous décrivons les principaux facteurs de qualité qui seront garant de celle de ce projet. Ces facteurs peuvent être considérés comme des contraintes. Les procédures décrites en ? ? nous permettent d'assurer que ces facteurs seront respectés dans le produit final. En particulier, toutes les contraintes relatives à l'évolutivité du système y sont décrites.

Annexe C

Analyse de l'existant

Pour une étude prélable de notre projet, nous nous sommes interessé à l'étude d'outils déjà existants pour avoir une première approche. Nous allons pour cela étudier le fonctionnement d'un outil de référencement en Java : le *JavaXRef*.

C.1 Présentation

C.1.1 But

Le but de ce logiciel est de pouvoir déterminer à quel endroit du code source *Java* les champs, méthodes et classes ont été utilisés.

C.1.2 Les *définitions* et les *références*

Un programme Java est constitué de *définitions* ainsi que de *références* :lorsqu'on écrit un programme, on définit des structures tel que les classes, méthodes et variables. Ces définitions utilisent d'autres structures appelées *référnces*. Cet outil permet donc de décrire ces relations en analysant syntaxiquement le code source.

C.2 Fonctionnalités

Voici les fonctionalités du logiciel :

– l'outil est utilisé pour la syntaxe *Java*.

- prend en entrée une liste de fichiers ou répertoires à analyser.
- parcourt dans chaque fichier source (**.java**) les définitions ainsi que les références.
- construit une table de symboles où sont stockées toutes les définitions ainsi que les références à ces définitions.
- produit un fichier représentant la table des symboles.
- on ignore les fichiers compilés (**.class**).

C.3 Inconvénients

- on ne parcourt pas les packages d'origine comme les classes **java.util.Vector** => on ne peut pas indiquer les définitions de ces packages ainsi que l'endroit où ils sont référencés dans le code source.
- //problème des surcharges de méthodes.

C.4 Les Entrées et sorties

C.4.1 les entrées

- le code source.

C.4.2 les sorties

- la table des symboles.

C.5 La stratégie adoptée

- premier parcours -> on repère les définitions ainsi que les références.
- à la fin -> on résoud le référencement.

C.6 Les outils utilisés

- le langage : **Java**.
- l'analyseur lexical : **ANTLR** (parser generator).

– l'analyseur syntaxique : codé en *Java* qui aura pour action de gérer la table des symboles.

C.7 Architecture de la table des symboles

– on enregistre les symboles dans des vecteurs.
– des piles permettent de gérer les portées pour savoir lors d'un référencement de quelle définition il s'agit.

C.8 Comparaison avec notre approche

– Cette méthode analyse toutes les définitons alors que la notre débute l'anlyse à partir d'un point d'entrée et repère de façon récursive les rf́rences auxquelles on fait appel.
– Cette méthode permet de référencer les variables ainsi que les instances de classe alors que la notre ne référence pas les variables locales.
– Cette méthode analyse uniquement le code source Java alors que le notre analyse le bytecode de la JVM. Les programmeurs manipulant des compilateurs faisant appel à la JVM pourront se servir de notre outil.

Annexe D

Partie externe : analyse et conception

D.1 Le système de l'extérieur

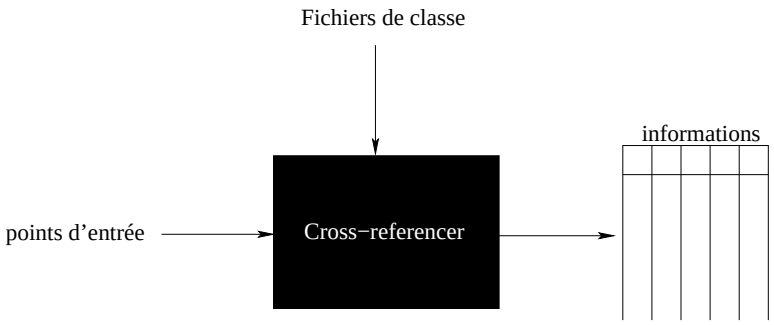

FIG. D.1 – Vue d'ensemble du cross-referencer

Le système de références croisées permet, tel que le suggère la figure D.1, d'obtenir des informations sur les références d'un ensemble de classes passées en entrée, fonction d'un ensemble initiateur contenant une série de points d'entrée.

Nous déterminerons dans cette partie les agents extérieurs pris en entrée par le système. Les informations obtenues donnent lieu à une autre analyse[1]. Deux points seront traités ici :

[1] Etude de cas d'Abdelaziz Boushabi

1. Les points d'entrée
2. L'ensemble des fichiers de classe

Les fichiers de classe sont obtenus indépendamment du langage utilisé par le programmeur —java, scheme, ...— avec, cependant, le point commun que le byte–code soit lexicalement, syntaxiquement et sémantiquement corrects. Nous pourrons néanmoins utiliser la dualité entre la spécification JAVA et celle de la JVM[2] de manière à réaliser la conception du produit en fonction d'un sous–ensemble de la JLS[3]. Cette approche reste en outre correcte vu qu'elle est axée sur *le concept objet* commun à JAVA et à la JVM[4] utilisant *les modificateurs d'accès*, *l'encapsulation* et le type de *classe*.

D.1.1 Les points d'entrée

Que sont les points d'entrée ? Les points d'entrée sont un ensemble de méthodes initiatrices d'un programme mettant en jeux une ou plusieurs classes. Nous noterons $\mathfrak{R}$ cet ensemble de méthodes.

A titre d'exemple, la fonction *main* permet de lancer un programme. Ainsi, elle peut être un très bon point d'entrée permettant de compter toutes les classes, méthodes et attributs ayant été utilisés. Cependant, les bibliothèques ne possèdent a priori aucune fonction *main*, ainsi il serait agréable de soit d'une part pouvoir les préciser, soit d'autre part pouvoir les déterminer de façon automatique.

Les points d'entrée se trouvent dans les fichiers constitués de classes, d'interfaces, d'attributs et de méthodes. Ainsi, qui peut être candidat pour être un point d'entrée ?

Classes et interfaces

Définir le rôle des classes ou interfaces est crucial afin de déterminer les potentiels points d'entrée.

[2] **J**ava **V**irtual **M**achine [2]
[3] **J**ava **L**angage **S**pecifications [1]
[4] JVMS §2 : *Java programming Language Concept*

Les modificateurs nous donnent le moyen d'effectivement les déterminer (JVMS §4.1) :

`ACC_PUBLIC`	:	Peut être accédée depuis l'extérieur du *package*
`ACC_FINAL`	:	Aucune *sous–classe* n'est autorisé
`ACC_SUPER`	:	Traitement des *méthodes* des *super–classes*
`ACC_INTERFACE`	:	C'est une *interface*, pas une *classe*
`ACC_ABSTRACT`	:	La *classe* ne peut pas être instanciée

Les modificateurs `ACC_FINAL`, `ACC_SUPER`, `ACC_INTERFACE` et `ACC_ABSTRACT` ne jouent aucun rôle sur la visibilité d'une classe. Ainsi, les points d'entrée possibles seront déterminés en fonction du modificateur `ACC_PUBLIC`. Si la classe ou interface est *public*, alors elle peut être accédée sans restriction dans et depuis l'extérieur du package.

Ainsi, toutes les classes ou interfaces publiques sont de potentiels points d'entrée dans le package depuis l'extérieur. Celles qui ne le sont pas peuvent permettre l'initiation uniquement à l'intérieur du package les contenant.

Les membres

Les modificateurs[5] `ACC_PUBLIC`, `ACC_PROTECTED` des membres nous permettent de déterminer un point d'entrée possible. En effet, lorsque le point d'entrée est l'appel d'une méthode depuis un autre package, alors nécessairement le modificateur doit être *public*. Cependant, une classe peut en hériter et, de ce fait, tout membre *public* ou *protected* peut être accédé.

Néanmoins, le modificateur *static* permet de spécifier que le membre — appelé membre de classe (variable de classe ou méthode de classe) — sera figé lors de l'initialisation de la classe pour toutes les instances de cette classe (JLS §8.3.1, §12.4.1).

Ainsi, les membres *static* de classes sont des points d'entrées possibles, alors que la classe n'est peut-être pas instanciable.

[5] JVMS §4.5, 4.6

Synthèse

Les points d'entrée sont des méthodes publiques qui permettent de lancer un traitement sur un ensemble de classes. Néanmoins, une méthode publique d'une classe non publique peut être un point d'entrée si et seulement si l'une des assertions suivantes est vérifiée :

- le système est lancé sur un autre package : la classe non publique hérite d'une classe publique ;
- le système est lancé sur le package lui-même.

Soit $\mathfrak{D}$, un ensemble de classes et $\mathfrak{R}$ un ensemble de points d'entrée spécifié. Tous les éléments de $\mathfrak{R}$ sont des méthodes de classes de $\mathfrak{D}$. En effet, s'ils ne le sont pas, alors, nécessairement, aucune classe de $\mathfrak{D}$ n'est utile car aucun moyen permettant d'y entrer n'existe. Nous pouvons alors définir une relation d'appartenance entre une méthode — et plus généralement un membre — et la classe le contenant. Nous noterons $\ltimes$ cette relation.

$$\forall\, x \in \mathfrak{R},\ \exists\, y \in \mathfrak{D}\ x \ltimes y \wedge \exists\, y' \in \mathfrak{D}\backslash\{y\}\ x \ltimes y' \Rightarrow y = y'$$

D.1.2 Les fichiers de classe

La seconde entrée du système est un ensemble de fichiers de classe. Pour des raisons de commodités, nous accepterons aussi en entrée des packages qui sont des ensembles de fichiers de classe. Il sera de plus primordial dans le cas de packages de préciser si l'analyse doit être faite à l'intérieur du package ou depuis l'extérieur. De cette façon, les méthodes initiatrices de $\mathfrak{R}$ pourraient connaitre leur mode opératoire en cas de conflit d'accession. Afin de résoudre ce problème, nous allons définir un troisième paramètre d'entrée : la **vue**.

Les vues

En fonction de l'endroit où l'on se positionne — à l'intérieur du package ou à l'extérieur —, la vue que nous avons de l'ensemble du système de classes est tout à fait différente. La figure D.3 symbolise la vision globale d'un package

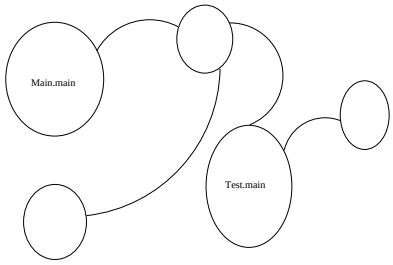

FIG. D.2 – A l'intérieur du package ex1

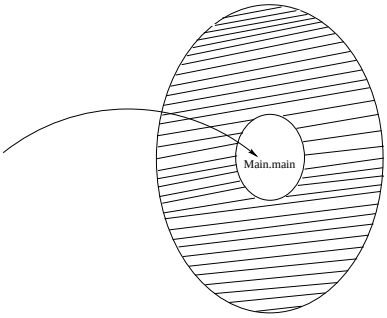

FIG. D.3 – A l'extérieur du package ex1

faisant apparaitre les classes publiques, alors que la figure D.2 montre l'intérieur du package. Dans un cas, $\mathfrak{R}$ sera limité à {ex1.Main.main} et dans le second à {ex1.Main.main, ex1.Test.main}.

Soit $\mathfrak{C}$, un ensemble de classes composant des packages $p_0 \ \ldots, \ p_n, n \in \mathbb{N}$. Nous noterons $\mathfrak{C}$ cet ensemble observé depuis l'extérieur et $\mathfrak{C}_{p_i,\ldots,\ p_{i+k}}$ le même ensemble observé depuis l'intérieur des packages $p_i, \ldots, \ p_{i+k}$.

Les fichiers

Soit $\mathfrak{D}$, l'ensemble des fichiers de classe composé de fichiers de classe c ou de packages p, eux même composés de fichiers de classe. Ainsi,

$$\mathfrak{D} \ = \ \bigcup_i p_i \cup \bigcup_i c_i \wedge \forall i, \ p_i \ = \ \bigcup_j c_j$$

De plus, ce domaine borne l'espace de recherche du cross-referencer. Nous pouvons, comme le montre la figure D.4, déterminer un périmètre clos.

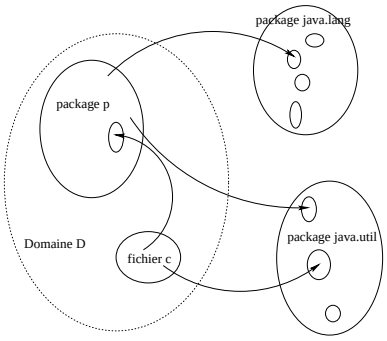

FIG. D.4 – Domaine de recherche

Les figures D.2 et D.4 montre une inter-dépendance entre tous les fichiers représentés dans les sources JAVA par les directives *import*. Il existe donc des relations de dépendance entre les fichiers notée $\overset{imp}{\prec}$. Notons Dom_c, l'ensemble des classes dont dépend le fichier c :

$$Dom_c \ = \ \{c'/c \overset{imp}{\prec} c'\}$$

Nous pouvons restreindre les ensembles de dépendance des classes en fonction de $\mathfrak{D}$:

$$\forall\, x \ \in \ \mathfrak{D} \ Dom_x^{\sim} = \ \{c \ \in \ \mathfrak{D} \ / \ x \overset{imp}{\prec} \ c\} \ = \ Dom_x \ \cap \ \mathfrak{D}$$

D.1.3 Les instructions

Les fichiers sont constitués de membres[6]. Les méthodes sont constituées de suite d'instructions accédant ou modifiant des variables référencées, appelant des méthodes ou définissant de nouvelles variables référencées. Dans notre système, nous considèrerons les attributs de type primitif de la même façon que nous considérons les références. Le but du cross-referencer est de compter le nombre de points d'accession aux membres d'une classe ou interface.

Avec ces éléments, nous pourrons, par exemple, déterminer les attributs, méthodes, classes, interfaces ou packages parasites pour une application donnée.

Notons **Def(c, m)** l'ensemble des définitions ou d'appels de méthodes de la méthode m de la classe c, et **Ref(c, m)** le référencement du membre m de la

[6]champs, méthodes, interfaces ou classes internes (JLS §8.1.5)

classe *c*.

Précisons que nous ne faisons pas un profiler, car une référence appelée dans une boucle ne sera comptée qu'une seule fois. De même, nous ne faisons aucune analyse de flots, ie par exemple, toutes les références du corps d'une méthode seront déterminées indépendament des branches conditionnelles.

Nous avons désormais toutes les indications nécessaires afin d'entrer dans le système $\mathfrak{S}$ à proprement parler.

D.2 Le système de l'intérieur

D.2.1 Récapitulatif des données

Les données

Le système $\mathfrak{S}$ a trois entrées

a. La vue *vue* qui permet de se positionner à l'intérieur ou à l'extérieur de packages

b. L'ensemble $\mathfrak{D}_{vue}$ composé de packages et de fichiers de classe

c. L'ensemble $\mathfrak{R}$ composé de méthodes étant les points d'entrée dans les fichiers de classe

Nous avons de plus défini deux relations

i. La relation $\overset{imp}{\prec}$: x utilise y se note x $\overset{imp}{\prec}$ y

ii. La relation $\ltimes$: x est membre de la classe ou de l'interface y se note x $\ltimes$ y

Nous utiliserons aussi des ensembles

- **Def(c, x)** qui est l'ensemble des accessions à des variables membre ou à des appels de méthodes de la méthode c.x (méthode x de la classe ou interface c)
- Dom_x est l'ensemble des classes que la classe ou l'interface x utilise
- $Dom_x^{\sim}$ est l'ensemble des classes que la classe ou l'interface x utilise à l'intérieur du domaine $\mathfrak{D}$

Finalement, le prédicat **Ref(c, x)** nous permettra de référencer le membre x de la classe c.

Les relations

Les relations suivantes sont nécessaires afin d'entrer dans le système correctement.

1. Tout élément de la racine est un point d'entrée d'une classe de $\mathfrak{D}$:

$$\forall\, x \in \mathfrak{R},\ \exists\, y \in \mathfrak{D}\ x \ltimes y$$

2. Nous précisons l'assertion précédente avec les vues :

$vue = \{p_0,\ \ldots\ ,\ p_k\}$

$\Rightarrow\ \forall\, x \in \mathfrak{R},\ \exists\, y \in \mathfrak{D}_{p_0,\ \ldots\ ,\ p_k}\ x \ltimes y$

$\Rightarrow\ \exists\, i \in I_k,\ \exists\, j \in p_i\ /\ x \ltimes i.j \vee \exists\, y \in \mathfrak{D} - vue,\ x \ltimes y$

3. Nous déterminons les domaines de chaque classe :

$$\forall\, x \in \mathfrak{D},\ Dom_x^{\sim} = \{\ y \in \mathfrak{D}\ /\ x \overset{imp}{\prec} y\ \} = Dom_x \cap \mathfrak{D}$$

4. Pour chaque fichier de classe, il existe un ensemble de membre qui référencent ou font appel à des méthodes d'autres classes du domaine de recherche :

$$\forall\, x \in \mathfrak{D},\ \exists\, \mathfrak{E}_x,\ \forall\, y \in \mathfrak{E},\ y \ltimes x \wedge \forall\, z \in Def(y,\ x),\ \exists\, x' \in \mathfrak{D},\ z \ltimes c'$$

D.2.2 Proposition d'algorithme

Premier algorithme

L'algorithme suivant permet de remplir les champs des membres appelés de façon récursive grâce à l'opérateur $\widehat{\mathfrak{S}}$ défini ci-dessous :

$\forall\, r \in \mathfrak{R},\ \widehat{\mathfrak{S}}(r)$

$\widehat{\mathfrak{S}}(x)\ \Rightarrow$

$\exists\, c \in \mathfrak{D}_{vue},\ x \ltimes c,\ Ref(c,\ x)$

$\wedge \forall\, d \in Def(c,\, x),\ \widehat{\mathfrak{S}}(d)$

Cependant, l'héritage n'est pas pris en compte ici, ni la différence entre les les différents membres de la classe. Nous devons ainsi ajouter quelques précisions supplémentaires.

Second algorithme

Nous précisons de nouveaux prédicats :

- x *est une superclasse ou superinterface* de y sera noté $y \overset{inh}{\prec} x$
- x *est un champ* de y sera noté $x \overset{f}{\ltimes} y$
- x *est une méthode* de y sera noté $x \overset{m}{\ltimes} y$
- x *est une classe interne* de y sera noté $x \overset{inner}{\ltimes} y$

Ainsi, l'algorithme précédent peut être complété en :

$\forall\, r \in \mathfrak{R},\ \widehat{\mathfrak{S}}(r)$

$\widehat{\mathfrak{S}}(x) \Rightarrow$

$\exists\, c \in \mathfrak{D}_{vue},\ x \ltimes c,\ Ref(c,\, x)$

// `héritage d'une classe -- abstraite ou pas -- ou interface, interne ou pas`

$\wedge \forall\, s \in \mathfrak{D}_{vue},\ c \overset{inh}{\prec} s \wedge x \ltimes s \Rightarrow Ref(s,\, x)$

$\wedge$ (// `Les membres`

// `Le membre est un champ`

$x \overset{f}{\ltimes} c$

// `Le membre est une méthode`

$\vee\ x \overset{m}{\ltimes} c \Rightarrow \forall\, d \in Def(c,\, x),\ \widehat{\mathfrak{S}}(d)$

// `Le membre est une classe interne ou une interface`

$\vee \exists\, c' \in \mathfrak{D}_{Vue}\ /\ c \overset{inner}{\ltimes} c' \Rightarrow Ref(c',\, c)$

)

Complexitée

Chargement de la table

Le remplissage de table de référencements et de celle des appels est linéaire en le nombre d'instruction de la somme des lignes de byte-code JVM contenu dans les fichiers de classe de $\mathfrak{D}_{Vue}$.

Traitement des données

Le traitement des données rendant le nombre de points de référencement par application de l'algorithme est obtenu de même de façon linéaire. Nous cherchons, en effet, le nombre de points de référencement avec en informations supplémentaires le fichier source et le numéro de la ligne où ils se trouvent.

```
1: methode(){
2:     call methode2;
3:     call field1;
   }
```

Supposons qu'il existe un chemin arrivant en 1, alors nécessairement tout les appels du corps de la méthode sont de potentiels points de référencement. De plus, quelque soit le chemin arrivant au point 1, alors les points de référencement seront identiques et déjà déterminés. Il est de ce fait inutile de traiter plusieurs fois une même donnée. Pour cette raison, lorsque les points auront été déterminés, il est inutile de garder en mémoire les appels les marquant : ils seront simplement détruits.

Cette propriété permet d'exhiber un **invariant**. A chaque appel de l'algorithme $\widehat{\mathfrak{S}}$ sur un point α :

i. L'appel α référence le membre de classe cible

ii. Ce point est détruit après traitement

iii. Le nombre de ligne est réduit d'une unité

D.2.3 Architecture

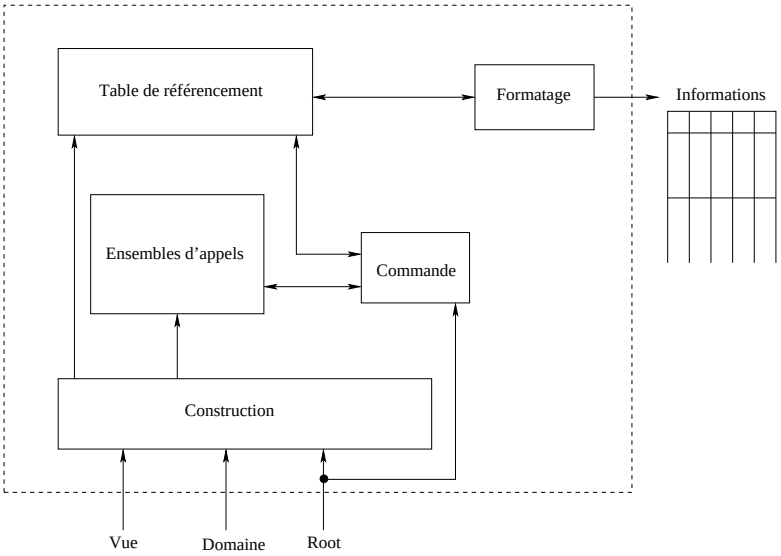

FIG. D.5 – Architecture du cross-referencer

La figure D.5 présente l'architecture générale su système :

1. La table de référencement : elle contient toutes les classes et les membres les composants avec leur nombre d'appels
2. Les ensembles d'appels : la liste des appels à des membres de classes des méthodes de chaque classe
3. La construction : elle permet de construire et d'initialiser la table de référencement et les ensembles d'appels en fonction des entrées
4. Le formatage : il construit les informations requises par l'utilisateur d'après la table de référencement
5. La commande : c'est le coeur du système qui remplit la table de référencement en fonction des ensembles d'appels et des initiateurs de l'ensemble **Root**

La table de référencement

Cette structure comporte la liste de toutes les classes du domaine $\mathfrak{D}$ en fonction de la vue. De même, tous les membres (champs, méthodes, classes

internes, interfaces) seront listés pour chaque classe ou interface. Une API devra proposer toutes les fonctions d'accession nécessaire pour l'ajout d'un appel à un membre et pour la lecture de la structure pour la restitution des données.

Les ensembles d'appels

Cette structure contient la liste des classes du domaine $\mathfrak{D}$ en fonction de la vue et toutes les méthodes et méthodes de classes internes les composants. Chaque méthode contient la liste des membres de classe appellés.

La construction

Ce module est le plus proche du code JVM. L'API du *classfile KOPI*[7] permettra d'accéder au byte-code et de remplir les structures de la table de référencement et des ensembles d'appels. L'architecture interne de la construction est spécifiée par Hicham Idrissi Khamlichi.

Le formatage

Le formatage des données de la table de référencement dépend des besoins du client. Ainsi, nous devons obligatoirement avoir la possibilité de spécifier différent type de formatage en passant par exemple par un design patter de *fabrique abstraite*.

La commande

Elle est l'implantation de l'algorithme proposé ci-dessus. Précisons une fois de plus que le cross-referencer n'est pas :

un profileur il ne simule pas l'exécution du code et le passage dans les boucles

un optimiseur il ne détecte pas le code parasite et donc certains membres qui ne seraient pas appelés

un simulateur il ne détecte pas en fonction de jeux d'entrée pour quel chemin opter

[7] copyleft DMS : si vous voulez l'acheter, DMS se réserve le droit de le mettre en copyright

un analyseur il ne fait aucune vérification de visibilité

D.2.4 Diagrammes UML

La structure de données

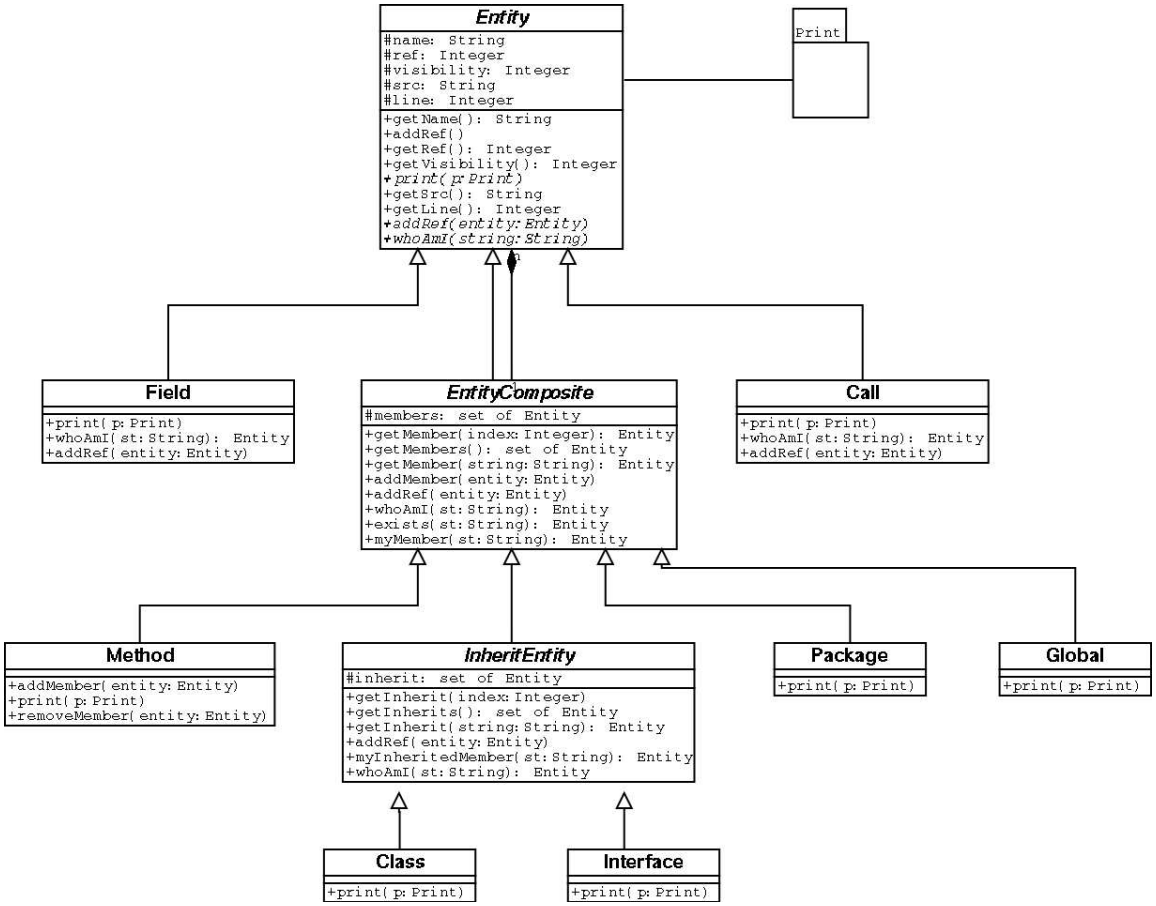

FIG. D.6 – Diagramme de classe de la structure de données

Le diagramme de classes D.6 présente l'architecture du cœur du système. Le schéma composite mis en œuvre permet le référencement automatique d'une *entité* en fonction de son type effectif :

- Package
- Classe
- Interface
- Méthode

– Champ

De même, la *visibilité* d'une *entité* et l'*ajout d'un membre* dépendent du type effectif de cette *entité*. Ainsi, les deux tables — celle des référencements et celle des appels — de la figure D.5 peuvent se fusionner en une seule représentée ici.

Ainsi, la *table des appels* et la *table des référencements* peuvent se fusionner une une seule *table* utilisant le package défini par le schéma D.6.

Le formatage

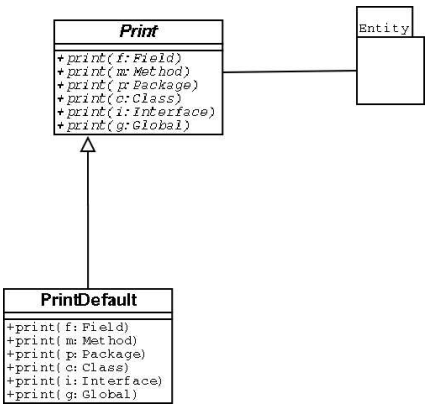

FIG. D.7 – Diagramme de classe du formatage des données

Le diagramme D.7 présente la *fabrique abstraite*, ou plutôt son adaptation, permettant le formatage des données. L'utilisateur peut passer en paramètre de l'affichage le *style* de formatage en écrivant les fonctions abstraites de la classe **Print**.

L'algorithme

Le schéma D.8 montre la simplicité de liaison de la classe implantant notre algorithme avec le reste du système. Cette approche *classe abstraite* permettra d'implanter un nouvel algorithme de meilleure complexité ci celui-ci en $\bigcirc(n)$ (n, le nombre de lignes) n'était pas suffisament performant.

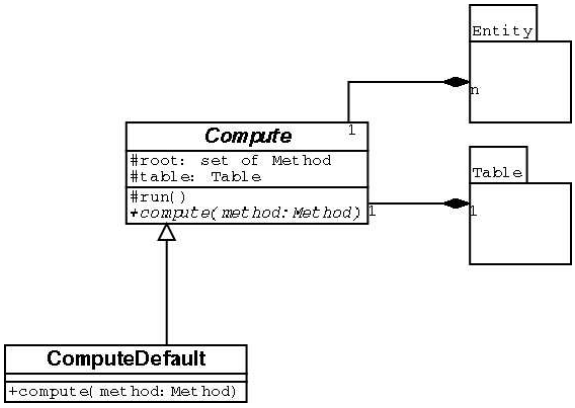

FIG. D.8 – Diagramme UML de l'algorithme

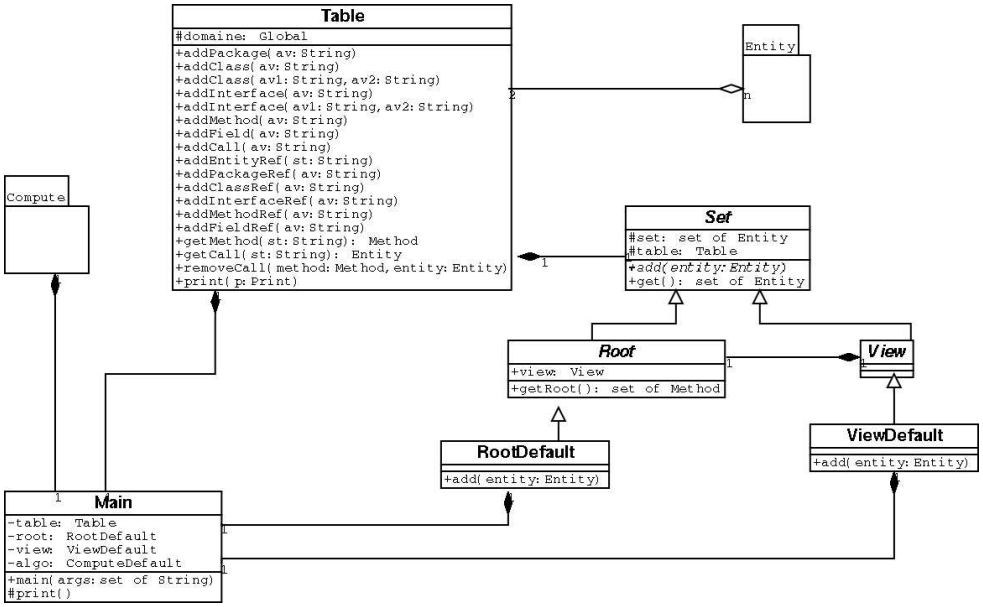

FIG. D.9 – Diagramme de classe du système en général

Système général

Le diagramme D.9 met en relation les packages nécessaires et propose la fonction initiatrice.

Annexe E

Partie interne : conception

Le sytème prend en entrée un ensemble de fichiers de classes. Qu'est ce qu'il y a dans un fichier de classe ? Y'a -t- il toutes les informations dont on a besoin ? De quoi a -t- on besoin exactement ?

E.1 Analyse des fichiers de classes

Un fichier de classe definie la représentation d'une et une seule classe ou interface à travers la structure :

- un identificateur de la classe.
- la version du fichier de classe.
- le nombre d'entrées dans la table *constantPool* décrie ci-dessous.
- la table *constantPool* contient des structures représentants le nom de la classe, de la super classe, les noms des variables et d'autres constantes liées à la structure du fichier de classe et de ses sous-structures.
- les flags d'acces :
  - public
  - final
  - super
  - interface
  - abstract

- une reférence sur une table de *constantPool* représentant la classe ou l'interface définie dans ce fichier.
- une reférence sur une table de *constantPool*représentant la super classe de la classe définie dans ce fichier.
- le nombre d'interfaces.
- une table contenant des référence sur des tables de *constantPool* représentant les interfaces de cette classe ou interface.
- le nombre de variables.
- une table de structures contenant la description complète de chaque variable.
- le nombre de methodes.
- une table de structures contenant une description complète de chaque methode de cette classe ou interface.
- le nombre d'attributs.
- une table d'attributs.

E.2 Analyse des besoins

Cette A.P.I. est d'une utilisation interne. Nous allons l'adapter, alors, à notre application, sans perdre tout de même, de généralité.

E.2.1 Classe et interfaces

De notre point de vue, une classe est définie par :

- les modificateurs
- son nom
- le nom de sa super classe
- ses inner classes
- ses interfaces
- ses membres

les modificateurs

Une classe peut être declarée public, final, interface ou abstract.

les nom de la classe et de la super classe

Si la classe est dans un package de nom qualifié *P*, alors, le nom qualifié de la classe est *P.identifier*, où *identifier* est l'identificateur de la classe. Si la classe est dans un package non nomé, alors, le nom de la classe est *identifier*.

le nom du package

Le nom qualifié q'un package qui est un sous-package, est composé du nom qualifié du package où il se trouve suivi d'un ('.') puis le nom du sous-package.

les inner Classes

Une classe interne est avant tout une classe interne. Elle représenté par une structure similaire à celle d'une classe.

les membres

- membres declaré dans la classe.
- membres herités d'une super classe.
- membres herités d'une super classe.

E.2.2 Variables

Une variable peut être déclarée public, protected ou private. Ainsi que static, transient, volatile ou final. Une variable sera noté par un *descriptor*, sous forme d'une chaine de caractères pouvant être généré par la grammaire suivante :

FieldDescriptor : FieldType

ComponentType : FieldType

FieldType : BaseType | ObjectType | ArrayType

BaseType : B|C|D|F|I|J|S|Z

ObjectType : L <classname> ;

ArrayType : [ComponentType

<classname> représente le nom qualifié d'une classe ou d'une interface.

BaseType	Type	Intérpretation
B	byte	signed byte
C	char	Unicode character
D	double	double-precision floating-point value
F	float	single-precision floating-point value
I	int	integer
J	long	long integer
L<classname> ;	reference	an instance of class <classname>
S	short	signed short
Z	boolean	true or false
[	reference	one array dimension

TAB. E.1 – Les intérpretations des type des champs

Par exemple *double d*[][][]*est*[[[*D*.

E.2.3 Methodes

Une méthode est notée par un *MethodDescriptor* qui peut être généré par la grammaire :

MethodDescriptor : (ParameterDescriptor*) ReturnDescriptor

ParameterDescriptor : FieldType

ReturnDescriptor : FieldType | V

Le caractère V indique que la methode ne returne rien (son type de retour est void).

Par exemple le descripteur de la methode *Object mymethod(int i, double d, Thread t)* est (`IDLjava/lang/Thread;`)`Ljava/lang/Object;`

E.2.4 Diagramme UML

Le diagramme UML E.1 présente la structure de l'API proposée et impalantée.

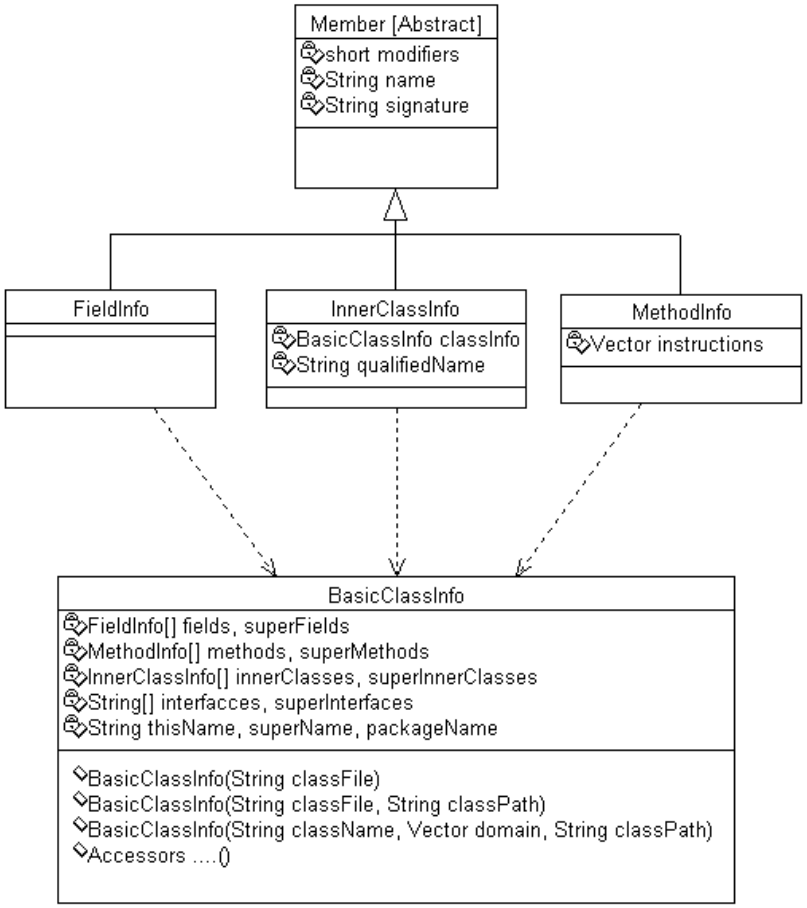

FIG. E.1 – Diagramme UML de l'A.P.I.

Annexe F

Plans de recettes

F.1 Premier exemple

Nous allons analyser un exemple concret dans le but d'illustrer notre procédure.

On considère un package où nous avons défini 2 classes déclarées en public.

```
package pac ;
public class Try extends Try2 {
   private int y ;
   public Try() { System.out.println("it's Try constructor") ;}
   public static void stat1(Try tr) { }
   public static void stat2(Try tr) { tr.modify() ; }
   public void modify(){ }
   public void meth1(int x) {
      Other o = new Other() ;
      this.meth2(o) ;
      SubOther so = new SubOther() ;
      this.meth2(so) ;
      y = 12 ;
   }
}
```

```
public class Try2 {
   public Try2(){};
   public void meth2( Other ot) {
      ot.getOther() ;
      System.out.printl("it's meth2") ;
   }
}
public class Other {
   int t ;
   public Other(){} ;
   public int getOther(){ t = 10 ; return t ;}
}
public class SubOther extends Other {
   public SubOther(){} ;
}
public class Test {
   public static void main(String[] args){
      Try e = new Try() ;
      int x = 6 ;
      e.meth1(x) ;
      stat1(e) ;
      stat2(e) ;
      Try2 e2 = new Try2() ;
   }
}
```

Chacune des classes précédentes sera définies dans un fichier d'extension `.java`. Ces fichiers seront compilés à l'aide de la commande *javac* et nous obtiendrons des fichiers d'extension `.class` (qui sont les fichiers compilés).

F.1.1 Entrées

1. Le domaine :

   Nous choisirons comme domaine l'ensemble des classes du package `pac` ainsi que l'ensemble des fichiers compilés (il faut préciser package et fichiers de classe selon létude de conception de M. Blanc[1])

2. La vue :

   On choisira une vue interne au package c'est-à-dire qu'on pourra avoir accès à toutes les classes publiques ou non du package.

3. L'ensemble des points d'entrée :

   Le point d'entrée sera la métode `main` de la classe `Test`

Nous rappelons que nous devons pouvoir indiquer pour chaque champ et méthode de classe, classe interne et interface le nombre de fois qu'ils ont étés référencés en partant des points d'entrée.

F.1.2 Sortie

Le tableau F.1 présente les résultats attendus.

F.1.3 Obtention des résultats

Le système est initié au début de la méthode `main`. A chaque fois que l'on rencontre un champ , une méthode ou une instance de classe, on incrémente de 1 le référencement de l'élément correspondant, puis on réitère le procédé de façon récursive sur le corps des méthodes. Par exemple : nous rencontrons une instanciation de la classe `Try`. On doit donc incrémenter de 1 le nombre de références des constructeurs `Try.Try()`, `Try.Try2()` et `Try2.Try2()` ainsi que de l'attribut `Try.y`.

L'instanciation de la variable `x` n'a aucun effet sur notre table de référencement car la classe `Integer` est défini dans un package se trouvant à l'extérieur

[1] `Conception et spécification` p.6, §2.1.1

Membres	**nb.réf.**	**Classe**	**interne**	**Interf**	**Champ**	**Méth**	**static**	**Hérité**
pac.Try		#						
int y	1				#			
Try()	1					#		
Try2()	1					#		#
void meth1(int)	1					#		
stat1	1					#	#	
stat2	1					#	#	
void meth2(Other)	2					#		#
pac.Try2		#						
Try2()	2					#		
void meth2(Other)	2					#		
pac.Other		#						
int t	4				#			
Other()	2					#		
int getOther()	2					#		
pac.SubOther		#						
SubOther()	1					#		
int t	2				#			#
int getOther()	1					#		#
Other()	1					#		#
pac.Test		#						

TAB. F.1 – Nombre de référencements des méthodes

de notre *domaine* considéré en entrée.

Puis on incrémente le référencement de la méthode `meth1()` ainsi que ceux des méthodes et champs auxquels elle fait appel.

F.1.4 Extensions possibles

Les futures extensions que l'on peut envisager consisterait à :

– indiquer à quel endroit le référencement a été effectué.

– indiquer le référencement des instances de classe (ou variables).

Suite à ces extentions, le tableau de référencement avec les entrées du premier exemple sera le tableau F.2 .

Les résultats concernant le référencement des instances de classe ainsi que des variables sont récapitulés dans le tableau F.3.

On remarque que l'instance `e2` de la classe `Try2` peut être supprimée car elle est définie mais jamais utilisée.

F.1.5 Représentation schématique

La figure F.1 représente les realtions entre les entrées et la sortie.

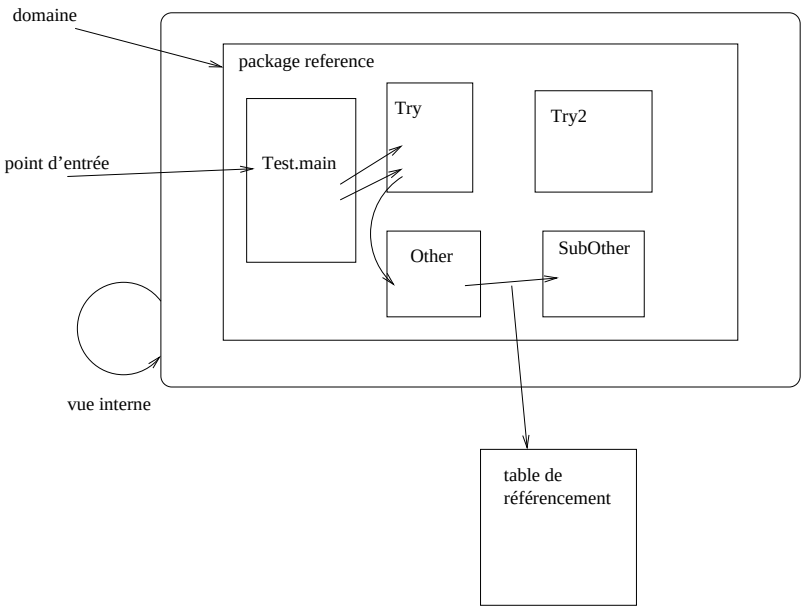

FIG. F.1 – Schéma général

Membres	**nb. réf.**	**Appels intermédiaires**
pac.Try		
int y	1	Test.main l1→Try.Try()
Try()	1	Test.main l1
Try2()	1	Test.main l1→Try.Try()
void meth1(int)	1	Test.main l3
void meth2(Other)	2	Test.main l3 → Try.meth1 l2 Test.main l3 → Try.meth1 l4
pac.Try2		
Try2()	2	Test.main l1→Try.Try(), Test.main l6
void meth2(Other)	2	Test.main l3 → Try.meth1 l1 Test.main l3 → Try.meth1 l3
pac.Other		
int t	4	Test.main l3→Try.meth1 l1→Try2.meth2 l1→Other.getOther l1 Test.main l3→Try.meth1 l3→Try2.meth2 l1→Other.getOther l1 Test.main l3→Try.meth1 l2→Try2.meth2 l1→Other.getOther l1 Test.main l3→Try.meth1 l4→Try2.meth2 l1→Other.getOther l1
Other()	2	Test.main l3 → Try.meth1 l1,3
int getOther()	2	Test.main l3 → Try.meth1 l2 → Try2.meth2 l1 Test.main l3 → Try.meth1 l4 → Try2.meth2 l1
pac.SubOther		
SubOther()	1	Test.main l3 → Try.meth1 l3
int t	2	Test.main l3 → Try.meth1 l3→Try2.meth2 l1 Test.main l3 → Try.meth1 l4→Try2.meth2 l1
int getOther()	1	Test.main l3 → Try.meth1 l4 → Try2.meth2 l1
Other()	1	Test.main l3 → Try.meth1 l1
pac.Test		

TAB. F.2 – Intermédiaires entre le point d'entrée et le membre qui est référencé

Instances	**nb. réf.**
pac.Try	
e	3
pac.Try2	
e2	1
pac.Other	
o	2
so	2
pac.SubOther	
so	2
pac.Test	

TAB. F.3 – Référencements des variables et instances de classe

F.2 Second exemple

Cet exemple illustre le cas de surcharge de méthodes.

F.2.1 Entrées

– *Domaine* :

```
package overload;
public class A {
   public void put(){System.out.println("it's A")}
}
public class B extends A {
   public void put(){System.out.println("it's B")}
}
public class Test {
   public static void main(String args[]) {
      B b = new B();
      b.put();
```

```
    }
}
```

- *Vue* : interne au package `overload`
- *Point d'entrée* : la méthode `Test.main`

F.2.2 Sortie

Membres	**nb. réf.**
class A	
A()	1
put()	0
class B	
A()	1
B()	1
A.put()	0
B.put()	1

TAB. F.4 – Surcharge de méthodes

F.3 Troisième exemple

Cet exmeple illustre le cas où on utlise une méthode comme point d'entrée :

F.3.1 Entrées

- *Domaine* :

```
package withmethod;
public class A {
   public void put(){this.hello()}
   public void hello(){ }
```

```
}
public class Test {
   public static void main(String args[]) {
      A a = new A();
      a.put();
      a.hello();
   }
}
```

– *Vue* : interne au package `withmethod`

– *Point d'entrée* : la méthode `A.put()`

L'utilisateur voudrait savoir quels sont les membres auxquels il faut prêter attention s'il désire modifier la méthode `A.put()`

F.3.2 Sortie

Membres	**nb. réf.**
class A	
A()	1
put()	1 //le point d'entrée lui même
hello()	1

TAB. F.5 – Autre point d'entrée

Conclusion : l'utilisateur devara faire attention au constructeur `A.A()` et à la méthode `A.hello()`.

F.4 Quatrième exemple

Cet exemple illustre le cas où une méthode fait référence à elle-même de façon directe (méthode récursive) ou indirecte. Considérons la situation suivante :

F.4.1 Entrées

– *Domaine* :

```
pulic class R {
   rec1(){ rec2() ; rec1() ;}
   rec2(){ rec3() ;}
   rec3(){ rec1() ;}
   main(){ rec1() ;}
}
```

– *Vue* : interne

– *Point d'entrée* : la méthode `R.main()`

La figure F.2 présente les référencements des différentes méthodes de la classe `R`.

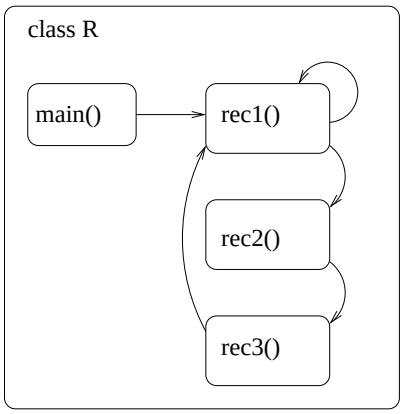

FIG. F.2 – Schéma des référencements

La méthode `R.main()` fait référence à la méthode `R.rec1()`. Cette dernière fait référence à elle même, donc on n'en tient pas compte. Elle fait aussi référence à elle-même en passant par `R.rec2()` et `R.rec3()` : on n'en tient également pas compte.

F.4.2 Sortie

Suite à cette étude, le résultat sera celui du tableau F.6.

Membres	**nb. réf.**
class R	
rec1()	1
rec2()	1
rec3()	1

TAB. F.6 – Référencement bouclé

Le référencement de `R.rec1()` n'est donc que de 1.

Annexe G

Rapports

G.1 Liens dynamiques

Les tests unitaires de la partie externe font apparaitre que les liens dynamiques peuvent être gérés en ajoutant, selon la nécessité, au corps des appels à cette fonction le membre appelé. Ainsi deux hypothèses sont possibles :

a. Ajouter à toutes les méthodes abstraites un tableau dons les cases pointent sur une paire <sous-classe, methode non abstraite>. Ainsi, en fonction du type de la variable, il serait possible d'ajouter au corps de la méthode appelante, la véritable méthode appelée.

b. Modifier la partie interne de sorte à implanter un système similaire à celui décrit dans le point précédant (ou autre) en simulant le run-time. La couche supérieure — en l'occurence la partie externe — accèderait à ces informations grâce à une simple primitive.

Il est cependant impossible dans l'état actuel et statique à ce niveau, du dossier de conception de vérifier un quelconque type. De ce fait, la première hypothèse semble difficilement valide. En outre, la partie interne peut aisément accéder aux information de bas niveau (byte-code JVM) ainsi, il serait envisageable d'intégrer ce processus dans cette partie.

G.2 Cohérences des données

Trois paramètres sont requis :

- Un ensemble de fichiers de classe
- Une vue
- Un ensemble méthodes initiatrices

Ainsi, les vérifications d'usage décrites dans le dossier d'analyse et de spécifications de la partie externe doivent être effectuées. A l'heure actuelle, ce n'est pas le cas. De ce fait, les informations passées en entrée doivent être valides. Il suffit désormais d'implanter ces tests décrits par les propriétés du dossier d'analyse.

G.3 Méthodes initiatrices

Passer les méthodes initiatrices en paramètre est à l'heure actuelle assez peu convivial, par exemple :

```
java cr.Main *.class -root=''cr/Main/main([java/lang/String;)V''
```

En effet, il s'agit de la solution la plus juste et la plus naturelle. Le système de cohérence des données pourrait permettre en fonction des paramètres de résoudre automatiquement `-root=main` en `cr/Main/main([java/lang/String;)V` lorsque aucune ambigüité ne peut survenir.

Bibliographie

[1] James Gosling Bill Joy Guy Steele Gilad Bracha. *TheJava*© Language Specification 2d edition. Addison–Wesley, 2000.

[2] Tim Lindholm Frank Yellin. *TheJava*© Virtual Machine Specification 2d edition. *SunMicrosystems*™, 1999.

[3] Bertrand BLANC Frédéric MACCARI. Plan d'assurnce qualité, 2000.

[4] Benjamin Bayart. Joli manuel pour LaTeX2. ESIEE, 1995.

[5] Marie-Claude GAUDEL. Précis de génie logiciel. MASSON, 1996.

[6] Anne-Marie HUGUES. Méthodes et Outils pour l'Assurance Qualité du Logiciel. ESSI, 2000.