MASTER EN SCIENCES INFORMATIQUES

Visualisation pour la compréhension de programmes avec un outil metacase

Maes, Philip

Award date:
2014

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2013–2014

Visualisations pour la compréhension de programmes avec un outil metaCASE

Philip Maes

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Vincent Englebert

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

RÉSUMÉ

Avec la complexité croissante des logiciels ; le vieillissement de ces derniers qui implique de la maintenance ; les problèmes de la documentation qui doit être maintenue à jour ; cela couplé au fait que durant toute la durée de vie d'un logiciel, les contributeurs ne sont pas les mêmes, emportant ainsi une partie de la connaissance avec eux ; des outils sont nécessaire afin de supporter l'informaticien dans sa découverte et sa compréhension des programmes.

Ce mémoire se propose d'étudier la manière dont ce processus de compréhension se déroule et d'apporter une approche qui se concentre principalement sur les aspects dynamiques des programmes. Cette approche nécessite de capturer certaines données et de les visualiser, de la manière la plus efficiente possible, afin que l'informaticien puisse répondre aux questions qu'ils se posent. Cette visualisation est proposée au moyen d'un outil metaCASE.

Mots clés

Analyse dynamique, metaCASE, visualisation logicielle, compréhension de programmes, métriques dynamiques

ABSTRACT

With the increasing complexity of softwares ; their aging that implies they need maintenance ; problems with the documentation that must be maintained ; this coupled to the fact that throughout the lifetime of a software, contributors won't be the same, taking a part of the knowledge with them when they are leaving ; for all these reasons, tools are needed to support computer scientists in their discovery and understanding of softwares.

This thesis proposes to study how this process takes place and to elaborate an approach that focuses on the dynamic aspects of softwares. This approach requires some data to be captured and then to visualize them in the most efficient way possible, so that the computer scientist is able to answer to his questions. This visualization is integrated in a metaCASE tool.

Keywords

Dynamic analysis, metaCASE, software visualization, softwares understanding, dynamic metrics

REMERCIEMENTS

Je tiens tout particulièrement à remercier le Professeur Vincent Englebert qui m'a été d'une aide et d'un soutien inimaginable. Sa passion et sa ferveur m'ont tout simplement ébloui. La citation se trouvant sur la page suivante décrit parfaitement l'état d'esprit qui était le mien lorsque j'assistais à son cours. Je lui suis infiniment reconnaissant.

Je remercie bien évidemment l'ensemble des professeurs et du personnel de l'Université de Namur pour la transmission de leur savoir et leur professionnalisme. Merci aussi à mes camarades de classe avec qui on s'est beaucoup amusés.

Merci à toute ma famille proche avec laquelle on a su traverser et surmonter les moments difficiles. À ma mère, je ne pourrai jamais la remercier assez pour tout ce qu'elle a fait et ce qu'elle fait pour moi. Je ne sais que trop peu la chance que j'ai de pouvoir bénéficier de son amour, sa fidélité et de son dévouement.

Toutes mes pensées vont à mes grands-parents maternels qui, je l'espère, seraient fiers de ce que je deviens.

« Tu as le regard d'un homme prêt à croire tout ce qu'il voit parce qu'il s'attend à s'éveiller à tout instant. »

Morpheus, The Matrix.

TABLE DES MATIÈRES

TABLE DES ILLUSTRATIONS

GLOSSAIRE

Bytecode java	Code Java binaire regroupant des instructions exécutables par une JVM.
Classloader	Charge de manière dynamique les classes nécessaires au bon fonctionnement d'une application dans la JVM.
Design Pattern	Il s'agit d'une « bonne manière » de résoudre un problème souvent rencontré. Le patron de conception est indépendant de tout langage et décrit des procédés pour solutionner un problème récurrent.
DSL	*Domain-specific Language*, est un petit langage qui se concentre sur un domaine applicatif particulier.
DSML	*Domain-specific Modeling Language*, DSL permettant de spécifier un programme directement à partir de concepts du domaine cible.
Framework	Composant architectural « prêt à utiliser ». Fournit une structure complète réutilisable.
JAR	Java ARchive, fichier ZIP contenant un ensemble de classes Java.
Java	Langage de programmation orienté objet de haut-niveau mis au point par Sun Microsystems, racheté par Oracle.
JDK	*Java Development Kit*, package contenant l'implémentation de Java ainsi que des outils à destination des développeurs ; typiquement un compilateur Java des outils de profilage, etc.
JVM	*Java Virtual Machine*, interprète et exécute du bytecode Java.
Package	Au sens POO, il s'agit d'un espace de nom qui regroupe un ensemble de classes. Une analogie parlante est de considérer les packages comme des répertoires différents permettant d'organiser ses fichiers.
POJO	*Plain Old Java Object*, signifie qu'un objet donné est un objet Java ordinaire.
Programmation Orientée Objet ou POO	Paradigme de programmation informatique. Un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne, un livre, etc. Il possède des caractéristiques (attributs) et des comportements (méthodes). Il s'agit donc de représenter, de manière plus ou moins formelle, ces objets et les relations entre ces objets.

INTRODUCTION

Nous sommes souvent amenés en tant que développeurs à reprendre du code d'autres personnes ou de soi-même, à travailler sur des projets qui ont déjà une certaine durée de vie. Les informaticiens doivent alors comprendre ce programme, se créer une logique de compréhension qui va s'affiner et se préciser avec le temps. Mais en attendant, comment faire en sorte d'avoir des informations intéressantes et utiles à la compréhension d'un logiciel ? Comment rendre le temps passé à essayer de comprendre un logiciel le plus efficace possible ? Parfois ils peuvent chercher un certain temps dans la mauvaise direction, ou tout simplement, il peut se passer quelque chose qui leur échappe, mais de très simple.

Cette démarche de découverte et de compréhension d'un nouveau programme se doit d'être la plus structurée possible, il convient de réfléchir à la meilleure approche mais aussi de s'auto-observer et d'être critique afin d'améliorer ce processus. Il est aussi très enrichissant de pouvoir observer les autres, même les accompagner dans ce cheminement, cheminement qui peut être qualifié « d'élaboration mentale d'une carte de compréhension ».
En effet, ce processus peut être matérialisé comme étant une carte du monde, vide et noire au début. De par les expériences précédentes, un informaticien peut commencer par avoir quelques points déjà présents sur cette carte et mêmes quelques liaisons entre ces points. Généralement, il se fait présenter le programme, les points centraux lui sont expliqués et c'est donc tout naturellement par ceux-ci que son exploration va commencer. Car il s'agit bien de cela, une exploration, même une aventure durant laquelle il va se balader et rencontrer des structures diverses, suivre des flux, bref l'analogie avec la carte semble tout à fait adéquate : le monde serait alors le système à découvrir, les continents seraient les sous-systèmes, les pays, les blocs fonctionnels et pour finir, les villes et les routes, les éléments et leurs liens, ces-derniers pouvant être de différentes natures : des appels de méthodes, de l'héritage, une dépendance autre, etc.
Au fur et à mesure de l'exploration, l'informaticien remplit cette carte mentale et surtout, le plus important, les liens se font entre les divers éléments. Mais cela requiert patience, curiosité, organisation et par-dessus tout, du temps, plus ou moins en fonction de l'expérience et de la complexité de la tâche. Des liens durables ne sont le fruit que d'un travail régulier, répété et se font surtout quand les éléments sont bien définis, il n'y a pas de miracle.

Dans les conditions d'un projet réel, ce processus de compréhension d'un programme demande beaucoup de temps, et il est rapporté selon des études, que plus de 60% de l'effort de la conception logicielle est affecté à la compréhension du logiciel ([Cornelissen et al., 2009]). Ce n'est pas juste lors de la découverte mais aussi lors du travail en équipe : plusieurs personnes participent au projet et le logiciel grossit. Chacun ne suit pas avec précision ce que les autres membres de l'équipe font, cela demanderait trop de temps. Donc même en cours de développement, un développeur est soumis à ce processus, on pourrait dire qu'il l'est constamment.

Une question que chacun pourrait immédiatement se poser, et à juste titre, c'est pourquoi devons-nous à chaque fois réapprendre, (re-)découvrir ? Cela veut-il dire que quelque chose a été perdu ou est manquant ?

Durant la vie d'un logiciel, des personnes vont partir et venir, les technologies vont changer, il va y avoir un tas de modifications. Il existe, à l'heure actuelle, des logiciels en fonctionnement qui ont plus de 20 ans ! Prenons l'exemple d'un logiciel comptable qui a 20 ans : le nombre de modifications pour rester à jour par rapport aux lois, le nombre de personnes qui ont participé à son élaboration et à sa maintenance, les erreurs qu'il a fallu corriger, etc. Il se passe énormément de choses en 20 ans.

Par rapport au début de l'informatique et de l'ingénierie logicielle, les systèmes sont devenus beaucoup plus gros et plus complexes. Il n'est même généralement plus question de logiciel seul, mais de logiciels faisant partie d'un système où ils cohabitent et communiquent. Il n'était, et n'est, pas aisé de comprendre un logiciel créé par d'autres ou même par soi-même quelques temps après, alors comprendre sa place dans un système, les interactions avec les autres éléments du système sont des facteurs amplifiant la difficulté de compréhension.

Une constatation importante à mettre en avant est le temps et l'énergie consacrés à la phase de maintenance d'un logiciel ([Wu et al., 2000], [Cornelissen et al., 2009]). De premier abord la conception, le développement et la mise en production d'un logiciel pourraient être perçues comme les phases les plus coûteuses et consommatrices de temps, mais la réalité est plus nuancée. Faire évoluer, corriger, maintenir, améliorer sont autant de tâches faisant parties de la maintenance et représentant, au final, les activités principales. Comme évoqué ci-avant, les informaticiens qui ont conçu le logiciel ne sont généralement pas ceux qui s'occupent de le maintenir. Dès lors, pour n'importe qui arrivé en phase de maintenance, devoir s'approprier le logiciel et donc n'en avoir qu'une compréhension partielle, ensuite corriger des erreurs ou ajouter une nouvelle fonctionnalité peut avoir des conséquences insoupçonnées. Le changement peut avoir des effets de bord inattendus ou tout simplement, pas anticipés.

Le moyen privilégié de garder la connaissance est de documenter le logiciel et tout ce qui est fait en utilisant divers formalismes, divers supports. En théorie c'est bien, mais en pratique il faut que cette documentation soit complète et surtout à jour, maintenue, ce qui n'est pas souvent le cas. Cela entraîne au fil du temps, une perte de la connaissance. Les informaticiens n'en ont alors que la vue haut niveau ainsi que, en fonction des préoccupations de chacun, une vue plus ou moins détaillée mais localisée de certaines parties.
La seule documentation concernant le logiciel qui reste tout le temps à jour est le code source lui-même. Mais il faut se rendre compte que, et on en conviendra aisément, lire le code source et partir de là pour accomplir cette tâche de compréhension n'est pas le moyen le plus rapide. Les technologies utilisées et non maîtrisées à ce moment-là pourraient même freiner la tâche. C'est pour cela que beaucoup ont pensé pouvoir répondre à cette problématique avec des outils qui partaient du code pour générer cette documentation ([Rohr et al., 2008]).

L'objet du travail est donc de proposer un outil d'aide à la compréhension de programmes : il ne s'agit pas ici d'offrir une vue générale, mais bien de pouvoir fournir des informations sur des parties localisées du programme. Le but est que l'outil puisse aider un développeur dans sa compréhension, son exploration, l'élaboration de sa carte mentale en lui permettant de répondre à des questions comme : quels sont les types effectifs des paramètres reçus par cette méthode ? Pourquoi la méthode de telle classe est appelée alors qu'il était attendu que ça soit celle d'une classe fils ? Et bien d'autres.
Il s'agit donc de fournir une vue localisée et assez détaillée. Le mot *assez* a son importance car il s'agit de ne pas submerger l'utilisateur d'informations, ni de lui en fournir trop peu. Il convient de lui présenter des informations utiles pour sa compréhension. Le moyen utilisé pour la présentation des informations devient alors un autre défi.

Cela étant défini et clair, il faut alors débuter le voyage, comprendre la problématique ainsi que ses tenants et aboutissants. La première étape du travail a donc consisté à faire un état des lieux afin d'acquérir les outils et déceler les concepts clés. La « *Partie I : État de l'art »* présente les points essentiels les points essentiels qui constituent les trois parties importantes de ce cheminement: quelles sont les informations disponibles et intéressantes dans le cadre de la compréhension d'un programme (les métriques), comment récolter ces informations (l'analyse) et la troisième, comment afficher ces informations (la visualisation).

Nous avons fait le choix important de se concentrer sur les aspects dits « dynamiques » d'un logiciel : ces aspects apportent nombre d'informations à propos d'une ou de plusieurs exécution(s) d'un logiciel, de montrer comment les fonctionnalités sont réalisées, bref ils permettent de « rendre visible » l'exécution, comme si la boîte noire que constitue un programme se transformait en boîte en verre au travers de laquelle il est possible de voir le fonctionnement interne du programme en temps réel. Les aspects dits « statiques » concernent principalement ce qui a trait au code source. La justification de ce choix est qu'il y a une abondance de travaux et de littérature concernant les aspects statiques. À contrario, les aspects dynamiques sont moins représentés car plus difficiles à obtenir et à décrire.

La deuxième partie du travail, « *Partie II : Réalisation »*, se propose de décrire l'approche utilisée, d'exposer les éléments techniques et d'introduire les nouveaux concepts, mais surtout, d'expliquer les choix qui ont été faits. Le but n'est pas de présenter une solution révolutionnaire ni complète mais bien de poser les bases, d'ouvrir la voie à de nouvelles possibilités et à de futurs développements. Une grosse part de cette partie est dédiée à l'élaboration de l'analyseur dynamique qui va se charger de récolter et consolider les diverses données. Ensuite, une visualisation est proposée et décrite répondant à la problématique de compréhension localisée efficace.

Une contrainte importante du travail a été l'utilisation d'un outil *metaCASE*, appelé MetaDONE[1], qui est développé dans le cadre des activités de l'Université de Namur[2], sous la responsabilité de Vincent Englebert[3]. Un outil metaCASE permet de concevoir et de générer des outils CASE. Ces derniers permettent la création de modèles. Il s'agit en effet d'une contrainte car la visualisation est dès lors cadrée, et il convient donc de prendre en compte les possibilités et les restrictions inhérentes à ce genre d'outils : pour donner un exemple de contrainte simple, une visualisation en trois dimensions n'est pas faisable.

[1] http://www.metadone.be
[2] https://www.unamur.be
[3] http://directory.unamur.be/staff/englebev

PARTIE I : ÉTAT DE L'ART

1. Métriques dynamiques

Ce chapitre va expliquer les différences entre métriques statiques et dynamiques en même temps que donner un aperçu des préoccupations que tentent de résoudre ces métriques. Un ensemble de caractéristiques va être présenté dont le but est de définir ce qu'une métrique dynamique doit respecter afin d'être considérée comme acceptable. Ensuite diverses métriques dynamiques vont être expliquées, ainsi que, dans certains cas, la manière de les obtenir. Ces métriques sont regroupées par catégorie, chacune représentant un aspect à étudier.

1.1 Introduction

La plupart des métriques sont calculées à partir d'une analyse statique du code source des applications ([Caserta, 2012]). Ces métriques sont dites statiques et sont les plus répandues. La première métrique statique, et certainement la plus connue et la plus utilisée, est celle qui compte le nombre de lignes de code. Avec le paradigme orienté objet, une telle métrique a perdu de son intérêt et de sa pertinence, ceci étant notamment dû au fait que la réutilisation est mise en avant et que des fonctionnalités se mettent en place de manière dynamique.

Ces métriques statiques sont par ailleurs insuffisantes quand il s'agit d'évaluer le comportement dynamique d'un logiciel [Singh et al., 2013], ceci étant dû notamment au fait que le comportement d'une application n'est pas seulement influencé par sa complexité mais aussi par son environnement d'exécution.

Il convient de constater qu'il n'y a pas énormément d'études au sujet des métriques dynamiques. Pour illustrer ce propos, voici une phrase écrite dans [Yacoub et al., 1999] : *« Actuellement, au meilleur de notre connaissance, il n'y a aucune documentation disponible concernant le couplage au niveau objet et concernant les métriques dynamiques. »*. Depuis lors, il n'y a pas eu beaucoup d'études à ce sujet, la plupart de ces dernières se limitant à exprimer des métriques concernant le couplage et la complexité dynamique. Parmi les plus intéressantes et les plus citées, on peut mentionner par ordre chronologique [Yacoub et al., 1999], [Arisholm, 2004], [Hassoun et al., 2004]. Il y en a bien évidemment d'autres mais la majorité traitent du couplage ou définissent des métriques qui au final reviennent sur le couplage.

Une étude est à mettre en avant [Dufour et al., 2003] et se propose d'établir une série de métriques dynamiques ainsi que de fournir des critères quant à la pertinence ou non d'une métrique. C'est de cette étude que nous allons partir et dont nous allons reprendre les grands thèmes. Pour chaque catégorie de métrique, un exercice de mise à jour va être tenté en se

basant sur des études plus récentes. Il est intéressant de noter que la plupart des articles du domaine citent [Dufour et al., 2003], ce qui en fait une sorte de référence.

De plus, un travail similaire, et même plus approfondi car il a été réalisé dans le cadre d'une thèse de doctorat, a déjà été fait dans [Caserta, 2012] mais dans lequel il manque parfois d'explications.

1.2 Caractéristiques principales

Il convient avant tout de définir ce qu'est une métrique. La définition la plus concise est donnée par [Wikipedia, 2014a] :

« *Une métrique logicielle est une compilation de mesures issues des propriétés techniques ou fonctionnelles d'un logiciel.* »

Afin d'être utile, une métrique se doit d'être concise et informative. Concise, dans le sens où, pour l'aspect qu'il faut évaluer, elle doit définir un petit nombre de mesures qui doivent pouvoir en établir le profil et permettre de le quantifier. Informative, à savoir que la métrique concernée doit représenter une caractéristique pertinente d'un programme et permettre de différencier des programmes au comportement différent. Les métriques se doivent donc de fournir des valeurs claires et mesurables afin qu'elles puissent être comparées.

Outre les deux caractéristiques évoquées dans le paragraphe précédent, [Dufour et al., 2003] établit une liste, non exhaustive, de quatre qualités qu'une métrique dynamique devrait posséder :

- non ambiguïté : une métrique ambigüe serait difficilement informative et comparable. L'exemple donné et très explicite est à nouveau celui de la ligne de code. Comment comparer deux programmes écrits dans deux langages différents ? Faut-il compter les commentaires ? ;
- dynamique : il est évidemment essentiel qu'une métrique dynamique mesure un aspect du programme qui ne peut être obtenu que lorsque ce dernier est en exécution. Nous pouvons d'avance, à la vue de cette caractéristique, en partie comprendre pourquoi les métriques dynamiques sont plus difficiles à obtenir que les métriques statiques ;
- robuste : l'idée ici est qu'une métrique définisse un aspect assez précis et isolé. Lorsqu'un programme est en exécution, un aspect peut dépendre de plusieurs facteurs. Si un de ces facteurs change, il faut que les mesures concernées changent proportionnellement. Si ce n'est pas le cas, la métrique en question est mal définie, trop large ;
- indépendante de la machine : dès lors que les métriques dynamiques concernent le comportement d'un programme, il semble essentiel que ces métriques ne soient pas influencées par le fait que l'action de mesure se fasse sur une plateforme différente.

1.3 Catégories

Taille

Les métriques qui concernent la taille d'un programme ont déjà été mentionnées, elles sont les plus simples à obtenir et les plus communes, à la différence que dans le cas qui nous occupe, le programme doit être en exécution afin de pouvoir estimer sa taille relative. D'ailleurs ce ne sont plus les lignes de code qui servent d'unité de mesure principale, mais comme le propose [Dufour et al., 2003], le nombre de classes chargées, le nombre d'instructions effectivement chargées ou exécutées, ou encore celles qui sont le plus souvent exécutées.

Il y a un élément important à prendre en compte, répondant à la caractéristique de robustesse précédemment évoquée, qui est le fait qu'il ne faut prendre en compte que les classes chargées spécifiques au programme. En effet, avec Java par exemple, la JVM charge par défaut un ensemble de classes quel que soit le programme chargé. Il n'est donc pas pertinent de prendre en compte ces classes pour les métriques liées à la taille d'un programme. Attention que pour d'autres métriques cela a du sens de prendre en compte ces classes-là.

Structures de données

Les structures de données et les types primitifs sont bien évidemment un point d'intérêt central. C'est une catégorie de métriques intéressantes qui permet de mettre en avant la manière dont l'application est structurée et qui permet aussi d'apporter des métriques permettant d'effectuer des optimisations substantielles.

Nous pouvons notamment citer des métriques comme le nombre de types primitifs utilisés par rapport aux objets, les types et structures les plus utilisés, le nombre de références vers un même objet, etc.

Cela permet aussi de catégoriser une application comme étant soit utilisatrice intensive de tableaux, de pointeurs ou encore de types à virgule flottante (ce qui dénote qu'il y aurait beaucoup de calculs scientifiques).

Complexité à l'exécution

Dans [Dufour et al., 2003], il s'agit d'une sous-catégorie de celle concernant la taille d'une application, sous-catégorie qui se nomme « *Structure* ». De plus, dans cet article, elle est un peu sous représentée mais il est fait mention du fait qu'au moment auquel l'article a été écrit, cette partie n'avait pas pu être finalisée. Dans [Caserta, 2012], il en est fait une catégorie à part entière, nommée « *Complexité* », ce qui de premier abord nous paraît tout à fait justifié car c'est un sujet très actuel et des plus difficiles à maîtriser.

Malgré cela, la raison pour laquelle [Dufour et al., 2003] n'a pas explicitement évoqué la complexité est, selon moi, que toutes les métriques permettent de décrire et d'expliquer la complexité. Il s'agit en fait de la catégorie racine. La complexité est générique et s'explique par plusieurs aspects ; ce qui nous amène à dire qu'avoir une catégorie concernant la complexité viole en fait certaines des caractéristiques essentielles que doivent posséder les métriques et qui ont été décrites précédemment. Il semblerait que le terme soit donc trop générique et erroné dans ce contexte.

Dans [Caserta, 2012], il est question de « *complexité à l'exécution* » dans le texte, ce qui semble tout à fait à propos et c'est la raison pour laquelle cette catégorie a été renommée de cette manière.

Ces métriques sont obtenues en comptant les instructions qui changent le flux d'exécution, à savoir toutes les structures de contrôles (if-then-else, les boucles,…) et les instructions de branchements comme les appels de méthode par exemple. Pour un programme Java ou s'exécutant sur la JVM, il s'agit donc de compter le nombre de bytecodes de contrôle exécutés lors de l'exécution. Cela reflète bien la complexité dynamique d'un programme. Il est aussi intéressant d'avoir le nombre de bytecodes de contrôle qui ont effectivement changé la direction du flux, ceux par lesquels le flux d'exécution est passé.

Ce type de métriques est particulièrement intéressant pour les développeurs car il permet de comprendre et d'améliorer directement le code du programme.

Couplage

Comme déjà introduit, il s'agit du type de métriques le plus étudié. À nouveau, la définition la plus simple et explicite est donnée par [Wikipedia, 2013a] :

« *Le couplage est une métrique indiquant le niveau d'interaction entre deux ou plusieurs composants logiciels (...). Deux composants sont dits couplés s'ils échangent de l'information.* »

Un point intéressant à noter ici est que l'analyse statique ne permet pas en réalité d'avoir une idée précise du couplage. En effet, dans les applications orientées objet, la liaison dynamique (dynamic binding) est souvent employée et ne permet dès lors pas de pouvoir mesurer efficacement le couplage en analysant juste le code source. La liaison dynamique est un mécanisme par lequel le lien avec la méthode appelée n'est pas créé pendant la compilation, mais en lieu et place, la méthode est recherchée par son nom à l'exécution.

Parmi les travaux majeurs effectués dans ce domaine, à savoir [Briand, 1999], [Yacoub et al., 1999], [Arisholm, 2004], [Hassoun et al., 2004], c'est [Arisholm, 2004] qui propose les meilleures manières de mesurer le couplage avec :

- les messages dynamiques : il faut compter le nombre total de messages distincts envoyés depuis (reçus par) un objet vers (depuis) d'autres objets ;
- les invocations de méthodes distinctes : il faut compter le nombre de méthodes distinctes invoquées par chaque méthode, et ceci, pour chaque objet ;
- les classes distinctes : il faut compter le nombre de classes différentes qu'une méthode utilise depuis un objet.

Ces résultats étant à chaque fois agrégés pour tous les objets de chaque classe.

Nous voyons ici trois métriques mais qui en fait, en fonction du point du vue, celui qui appelle ou celui qui est appelé, et du niveau de granularité, objet ou classe, font qu'il y a en tout douze métriques.

Polymorphisme

Le polymorphisme est une fonctionnalité importante du concept orienté-objet. Pour rappel, le polymorphisme est ([Wikipedia, 2014b]) :

« *…, le polymorphisme est l'idée de permettre à un même code d'être utilisé avec différents types, ce qui permet des implémentations plus abstraites et générales.* »

En Java, le polymorphisme est dit par héritage, ce qui permet de redéfinir une méthode dans les sous-classes (qui héritent donc) d'une classe ; la méthode effectivement appelée sera choisie en fonction du site d'appel et du type polymorphique de l'objet sur lequel l'appel est fait. Par exemple, à nouveau en Java, cela est faisable facilement car ces appels prennent la forme, au niveau du bytecode, d'instructions *invokeVirtual* et *invokeInterface*. Les appels à des méthodes statiques ne sont pas pris en compte car ces dernières ne sont pas affectées par le polymorphisme.

Les métriques pour cette catégorie sont divisées en trois parties:

- ici sont mesurées les instructions potentiellement polymorphiques, cela ne dit rien sur le fait que les polymorphismes soient réalisés ou non : en d'autres mots, que soit la méthode cible ou le type du receveur change à l'exécution. Il est notamment possible de compter le nombre d'instructions d'appels exécutées et aussi avoir une estimation de la densité d'appels dans l'application en divisant la première mesure par rapport au nombre total d'instructions exécutées ;
- l'intérêt va être ici mis sur la mesure du polymorphisme du receveur : pour chaque appel, il va être compté le nombre de types différents que le receveur peut avoir. En plus de cette mesure, on peut compter le nombre d'appels qui vont donner lieu effectivement à un changement du type du receveur ;
- l'intérêt va être mis ici sur la mesure des méthodes réellement ciblées : les mesures sont similaires à celles pour le receveur sauf qu'ici il est compté le nombre de méthodes potentiellement ciblées.

Dans [Caserta, 2012], une autre métrique est définie, métrique qui concerne plus les performances de la JVM que le polymorphisme directement. Elle a été placée dans cette catégorie car elle utilise le polymorphisme comme base pour mesurer la quantité de résolutions dynamiques que la JVM doit faire. En effet, ce n'est pas une opération sans coût que de retrouver la méthode effectivement ciblée.

Ces métriques sont utiles pour mesurer à quel point une application est « orientée objet » ou encore de voir si des améliorations dans les arbres d'héritage seraient profitables ou non pour les performances.

Concurrence

Dans [Dufour et al., 2003], le terme utilisé est « *Concurrency and Locking* » dont la traduction en français est moins explicite et qui met le « blocage » au même niveau que la concurrence alors qu'il en est un des composants. Dans [Caserta, 2012], c'est appelé « *Métriques liées aux fils d'exécutions* », ce qui est trop réducteur pour un nom de catégorie, catégorie pour laquelle on s'attendrait alors à n'avoir que des métriques qui concernent effectivement les fils d'exécution comme le nombre de fils d'exécution, leurs durées de vie moyennes, ainsi de suite. De plus, il est inapproprié de parler de fil d'exécution qui a une connotation trop concrète, déjà spécialisée alors qu'à nouveau, c'est un composant du domaine de la concurrence.

C'est un type de métriques des plus importants dans un contexte de programmation concurrentielle car c'est un aspect exclusivement dynamique. De manière statique, il n'est pas aisé d'avoir des métriques précises concernant l'efficacité d'une application dans un environnement concurrentiel ; cela ne peut se mesurer qu'à l'exécution. C'est donc un des points forts de cette technique.

Avec le développement des processeurs multi-cœurs et le besoin toujours plus accru de puissance de calcul, la programmation concurrentielle se développe de plus en plus et permet de profiter pleinement de ces avancées. Mais elle engendre aussi certains impératifs, comme la synchronisation ou le partage de ressources, qui, s'ils ne sont pas maîtrisés, ont des conséquences négatives. C'est pourquoi il est important de pouvoir se rendre compte de la réelle efficacité d'une application faisant usage de ce style de programmation. Ces métriques peuvent en outre servir pour effectuer des optimisations.

Cette catégorie sera divisée en deux grandes parties, à savoir les fils d'exécutions (threads) et la synchronisation.

Tout d'abord en ce qui concerne les fils d'exécutions :

- il convient tout d'abord de savoir si une application utilise plus qu'un fil d'exécution à la fois. Ce n'est pas aisé à déterminer car ce n'est pas parce qu'il y a plusieurs fils d'exécution qu'ils vont être exécutés de manière concurrente. Une approximation, mais qui donne quand même une idée de l'utilisation de la concurrence qui est faite par l'application, est possible en comptant le nombre maximum de fils d'exécution qui sont simultanément dans un état ACTIVE, le fil d'exécution est en cours de traitement, ou RUNNABLE, le fil d'exécution est prêt à travailler ;
- afin d'indiquer s'il y a plus ou moins d'exécutions concurrentes qui se déroulent, il est opportun de mesurer la quantité de code qui est exécutée pendant qu'un autre fil d'exécution est en cours d'exécution. Pour mesurer ceci, il est nécessaire d'avoir recours à la mesure définie au point précédent mais à un instant *t*.

Concernant la synchronisation, la meilleure manière de voir s'il en est fait usage et à quel point, est de mesurer le nombre de fois que des opérations de verrouillage (lock) sont effectuées. Ces verrouillages sont conceptualisés par le fait d'entrer dans des blocs dits synchronisés (synchronized) et matérialisés au niveau bytecode par les instructions *monitorenter* et *monitorend*. Il vient donc, de manière assez évidente, que mesurer le nombre d'entrées dans ces blocs, mais aussi de comparer le nombre d'entrées et le nombre de sorties afin de repérer les impasses (deadlocks). À nouveau, pour avoir une idée de la densité des blocs de synchronisation dans l'application, on peut mesurer, au niveau des bytecodes, le nombre de ces blocs pour mille instructions exécutées par exemple.

Afin de se rendre compte de l'utilisation qui est faite pour chaque bloc synchronisé, il peut être mesuré le nombre de fois qu'un bloc spécifique a été activé. Cela permet de repérer quels sont les points dits chauds.

Il est à noter que ces mesures ne respectent pas totalement la caractéristique d'indépendance de la machine car le nombre de cœurs peut différer de l'une à l'autre. De manière idéale, il faudrait évidemment effectuer ces mesures sur des machines disposant du même nombre de cœurs.

1.4 Conclusion

Les métriques dynamiques sont importantes et un sujet très vaste, sujet pour lequel il y a encore beaucoup à faire. Cela est notamment dû au fait qu'il est laborieux d'acquérir les données permettant de calculer les métriques dynamiques. Nous pouvons également mettre en avant le fait que ce sont souvent les mêmes sources qui sont citées dans les différents travaux relatifs à ce domaine. Il convient vraiment de retenir [Dufour et al., 2003] pour l'effort fait quant à l'élaboration de critères permettant de juger si une métrique est acceptable ou pas, ainsi que pour la proposition de plusieurs métriques, classées selon des catégories plus ou moins bien définies.

Les métriques dynamiques sont en petit nombre mais leur utilité est importante : elles permettent d'éclaircir les aspects dynamiques d'un logiciel, on pourrait même dire qu'elles offrent de la visibilité aux aspects, qu'elles les font parler.

Le domaine dans lequel cette approche a sa plus grande valeur est celui qui concerne la concurrence : il est extrêmement difficile d'avoir un comportement déterministe pour des applications exploitant la programmation concurrentielle. Avec les métriques dynamiques, il est tout à fait possible d'obtenir des mesures qui permettent de guider l'implémentation, de comprendre ce qui se passe réellement à l'exécution, sous réserve quand même que ça ne peut être représentatif de tous les comportements possibles.

2. Analyse dynamique

Il va être présenté ce qui est attendu de l'analyse dynamique. Comme pour les métriques, une explication des différences entre analyse statique et dynamique sera faite, en présentant les avantages et inconvénients de chacune des techniques. La problématique de l'influence de l'observateur sur l'objet observé sera exposée. Ensuite, différentes approches afin de mettre en place l'analyse dynamique vont être abordées.

2.1 Introduction

L'analyse est la partie du processus qui va permettre de récolter les informations nécessaires à la visualisation. Il s'agit d'une étape cruciale, à ne pas sous-estimer ; d'elle seule dépendra la qualité de ce qui pourra être proposé. Un outil mal conçu pourrait ne pas offrir assez de possibilités quant aux thèmes qu'il est possible d'analyser et ainsi voir très vite son utilité fortement réduite.

De manière succincte et systématique, il va être vu dans le présent chapitre, les forces et faiblesses de l'analyse dynamique ainsi que les différentes approches qu'il est possible d'adopter.

Il a été fait le choix de rester le plus générique possible ainsi que d'éviter les détails trop techniques ou liés à un environnement particulier, ceci malgré le fait qu'on soit dans un environnement Java. Comme déjà expliqué dans l'introduction de cette partie, le but est d'avoir les clés qui permettent de prendre les bonnes décisions. Par contre, pour faciliter la compréhension et pour illustrer les propos, cela sera le cas.

Il est à rajouter que l'analyse dynamique a nécessité beaucoup de travail et que dès lors, un chapitre conséquent lui est consacré dans la « *Partie II: Réalisation* », ce qui peut expliquer aussi que le présent chapitre s'attache à rester à un niveau plus général.

2.2 Dynamique contre statique

Dans [Ball, 1999], l'analyse dynamique est définie comme « *l'analyse des propriétés d'un programme à l'exécution* ». L'analyse statique analyse le code source d'un programme afin de prouver que ce dernier respecte certaines propriétés et ce, pour toutes les exécutions. L'analyse dynamique, elle, fournit des informations utiles concernant le comportement d'un programme et permet de détecter des violations de certaines propriétés désirées, ceci valant pour une ou plusieurs exécutions.

Les principaux atouts de l'analyse dynamique sont :

- la précision de l'information : les outils permettant de collecter les informations ou d'enregistrer certains aspects dynamiques du programme à examiner peuvent être réglés de manière précise afin de ne récolter que les informations effectivement nécessaires dans le but de répondre à un problème particulier ;
- la dépendance par rapport aux entrées (inputs) : c'est aussi un défaut et une qualité de l'analyse dynamique, cette dernière sera directement impactée par le moindre changement dans les entrées qu'on fournit au programme mais aussi ce que le programme produit (outputs). Il est donc très facile de pouvoir tester plein de cas car il suffit de changer ces entrées ;
- la mesure du temps : l'analyse dynamique peut fournir des mesures concernant le temps ; combien de temps met le programme pour analyser tel fichier, combien de temps il prend pour démarrer, pour effectuer telle ou telle fonctionnalité ? L'analyse statique en est tout bonnement incapable ;
- la subtilité : un autre atout qui n'apparaît pas dans la littérature est le fait que seule une analyse à l'exécution peut révéler certains bugs ou erreurs subtils de conception, de configuration, de design ou encore de logique introduits par l'humain ou par les technologies utilisées ainsi que leurs interactions. Typiquement les cas concernant la concurrence ou encore deux frameworks qui se tolèrent mal l'un l'autre. Comme déjà expliqué lors de la présentation des métriques, tout ce qui concerne les dépendances dynamiques, injection de dépendances, le polymorphisme, etc.

Outre ce qui vient d'être évoqué, et le fait d'obtenir des métriques concernant la qualité d'un programme, un intérêt majeur de l'analyse dynamique est de savoir où le code peut être optimisé de manière significative.

Dans [Arisholm, 2004], il est expliqué que l'analyse dynamique est généralement considérée comme plus compliquée car la collecte de données d'un programme en exécution est plus difficile à mettre en œuvre, surtout si les sources ne sont pas disponibles, et provoque une détérioration relative au niveau des performances. En effet, du fait des techniques utilisées afin de récolter l'information, il est presque toujours nécessaire de rajouter du code, donc de modifier le programme existant. Ce code va de plus généralement devoir sauvegarder les informations récoltées, peut-être les manipuler directement ou même déjà les consolider, ce qui va entraîner un coût à l'exécution au niveau des performances et ralentir le programme.

Il n'est pas évident de couvrir tous les aspects dynamiques d'un programme car il faut pour cela utiliser l'entièreté de ses fonctionnalités.

Un troisième point négatif concernant l'analyse dynamique, est le fait qu'elle n'est pas faisable tant que le programme à analyser n'est pas achevé ou bien, n'est pas pertinente trop tôt lors du développement du programme. Au contraire de l'analyse statique qui peut presque directement fournir des informations utiles concernant la qualité et l'état d'avancement du programme ainsi que d'autres métriques.

Voici une comparaison qui résume bien les différences entre l'analyse dynamique et statique. Cette comparaison est trouvée dans [Gupta et al., 2008] :

TABLEAU 1 - COMPARAISON ENTRE L'ANALYSE DYNAMIQUE ET L'ANALYSE STATIQUE.

Analyse dynamique	Analyse statique
Analyse centrée sur les stimuli que le programme reçoit	Analyse centrée sur le programme
Rend compte du comportement du programme	Rend principalement compte de la structure du programme
Plus précise	Comparativement moins exacte
Le fait de mesurer à l'exécution a un coût non négligeable au niveau des performances	L'activité de mesure se fait à un coût moindre
Gère l'héritage, la liaison dynamique, le code non-utilisé	A des manquements concernant la prise en compte des aspects orientés objet
Les résultats obtenus sont dépendants du scénario utilisé et de la manière dont il est réalisé	Il y a beaucoup moins de perturbations possibles dues
Processus assez lent	Processus beaucoup plus rapide
Processus relativement complexe	Processus relativement facile
Génère une très grande quantité de données	Génère relativement moins de données à traiter

Ces deux techniques sont donc complémentaires sur plusieurs points :

- l'exhaustivité : l'analyse dynamique rend compte de ce qui est observé à l'exécution et dépend donc fortement des jeux d'exécutions. Si dans un de ces jeux il est décidé de ne pas utiliser une fonctionnalité, cette dernière ne fera pas partie de l'analyse. À contrario, l'analyse statique analyse l'ensemble du programme. Aucune des deux analyses ne peut à elle seule trouver toutes les erreurs ;
- l'étendue : à l'exécution, il peut apparaître des dépendances indétectables autrement qu'avec une analyse statique, cela étant dû au fait de l'enchainement des appels de méthodes, des instanciations d'objets successives et autres. De plus, l'analyse statique va pouvoir étudier, par définition, plus de cas car elle va prendre en compte tous les chemins d'exécution possibles, pas juste ceux qui sont invoqués durant l'exécution.

Il est intéressant d'utiliser conjointement les analyses statiques et les analyses dynamiques afin d'avoir une analyse complète et précise d'un programme plutôt que partielle. Néanmoins, en fonction des propriétés qu'il est nécessaire d'identifier et de mesurer, il est possible de n'utiliser que l'analyse adéquate.

2.3 L'effet de l'observateur

Dans le point précédent, il est fait mention d'un désavantage concernant l'analyse dynamique et qui était le coût à l'exécution que peut induire la mise en place d'une telle technique.

Il y a, dans plusieurs disciplines scientifiques, un effet qui veut que le simple fait de l'observation provoque des changements sur le phénomène qui est observé. Le meilleur exemple pour illustrer cela appartient au champ de la physique et est l'expérience des fentes de Young ([Wikipedia, 2014c]) avec son interprétation quantique : à un niveau macroscopique, cette expérience met en évidence la nature ondulatoire de la lumière et la dualité onde-particule de la matière par l'apparition d'interférences pour la nature ondulatoire et d'impacts sur les écrans de détection pour la nature corpusculaire.

Mais lorsqu'on cherche à connaître par quelle fente exactement un quantum est passé, à un niveau microscopique donc, l'effet d'interférence est éliminé. En effet, pour observer le photon, il faut « illuminer » la sortie des deux fentes pour savoir par laquelle il sort ; cette lumière qui illumine les fentes contient en fait assez d'énergie pour perturber le photon observé et le faire changer de trajectoire.

Donc le simple fait d'avoir voulu mesurer, a provoqué un impact de la part de l'instrument de mesure sur le sujet étudié. Dans les sciences sociales par exemple, il a été remarqué que les personnes vont se comporter différemment quand elles sont observées.

En ingénierie logicielle, le phénomène porte le nom d' « effet de sonde » (probe effect) et se définit comme « *l'altération involontaire du comportement d'un système causée par la mesure du système.* » ([Wikipedia, 2013b]). Dans [Zaidman, 2006], ce phénomène est expliqué, pour l'analyse dynamique, par : du fait de l'ajout de code et de la modification du code original, le système pourrait être moins réactif ou même influencer les interactions entre threads. Pour illustrer les conséquences d'un manque de réactivité, on peut facilement imaginer un utilisateur, cliquer plusieurs fois sur un bouton du fait du manque de réaction de la part du système.

Ce n'est pas anodin, car l'analyse dynamique, étant dépendante des entrées et étudiant le comportement à l'exécution, pourrait en fait fausser les résultats du fait de sa présence. Il convient donc d'être très attentif à ce qui est fait afin d'être le moins invasif possible et avoir peu d'impacts. Particulièrement, le fait de pratiquer une analyse post-mortem, à savoir après exécution, permet de grandement réduire cet effet. Dès lors, à l'exécution, seul le traçage est effectué.

2.4 Approches

Maintenant que nous avons bien défini ce qu'est l'analyse dynamique, il nous reste à discuter de la partie la plus cruciale, à savoir sa mise en place afin qu'elle soit la plus efficace possible. L'analyse dynamique peut être réalisée en utilisant différentes approches.

Il va être ici principalement résumé [Gupta et al., 2008] et [Zaidman, 2006], le tout complété avec [Cornelissen et al., 2009], ce qui donne une liste des différentes approches utilisées afin d'effectuer l'analyse dynamique. Afin de garder cette partie conceptuelle, théorique et explicative, les détails techniques ne seront pas abordés en profondeur. Par contre, ils le seront dans la Partie 2 du présent document, partie qui concerne plus la réalisation.

Les profileurs (profilers)

Un profileur est typiquement utilisé pour étudier les besoins au niveau des performances et de la mémoire d'un programme. Un débogueur (debugger) est quant à lui généralement utilisé afin de passer au travers d'un programme, étape par étape, à un niveau très bas, afin de découvrir les raisons d'un comportement inattendu. Leur principe se base sur l'envoi d'événements, tels que les appels et retours de méthodes ou encore l'accès aux variables, à différents moments de l'exécution. Il suffit donc de capturer ces événements et soit de les stocker soit de les analyser et interpréter directement.

Les benchmarks sont normalement utilisés pour l'analyse à l'exécution de programme orienté objet. Dans ce cas-ci, le programme est considéré comme une boîte noire : les entrées sont fournies, les sorties sont reçues mais il n'y a aucune connaissance de ce qui se passe dans la boîte même si cela est possible. L'activité de benchmarking consiste donc en l'exécution du programme et en la capture de données reflétant ses performances.

En Java, cela peut se réaliser typiquement en instrumentant la JVM elle-même ou le code du programme afin de capturer les événements décrits plus haut et d'ajouter le comportement voulu afin de traiter ces événements. Pour plus de détails techniques, veuillez vous référer au chapitre consacré à ce sujet, « *Partie II, Instrumentation* ».

C'est une approche assez bas-niveau mais qui permet d'avoir accès à énormément d'informations utiles pour mener à bien l'analyse.

Approche orientée aspect

La programmation orientée aspect (POA) permet la modularisation de préoccupations transversales, tout comme la programmation orientée objet permet la modularisation de préoccupations communes. De manière grossière, cela permet d'insérer des bouts de code au commencement ou à la fin des méthodes par exemple ; dans ces bouts de code, on a alors accès au « contexte » d'exécution. La POA permet d'écrire des bouts de code de manière indépendante du programme à analyser : ce code sera « tissé » dans le programme en fonction de l'approche adoptée. À nouveau, pour les détails, veuillez vous référer au chapitre « *Programmation Orientée Aspect (POA)* », dans la « *Partie II* ».

Dans cette optique orientée aspect, le traçage de l'exécution, l'extraction des informations nécessaires à l'analyse dynamique peuvent être vus comme une préoccupation transversale. Deux avantages à cette approche sont qu'elle permet d'avoir un outil réutilisable pour analyser divers programmes, il suffit de le tisser ; deuxièmement, le code étant indépendant, il ne devrait pas avoir d'effets de bord.

Réécriture de l'AST

L'AST (abstract syntax tree ou arbre syntaxique abstrait en français) est « *un arbre dont les nœuds internes sont marqués par des opérateurs et dont les feuilles (ou nœuds externes) représentent les opérandes de ces opérateurs*. » ([Wikipedia, 2013c]). Il s'agit, de manière très résumée, de la représentation sous forme d'arbre des instructions qui composent un programme informatique.

Lorsque le code source d'un programme est chargé et que l'AST est construit, il est possible de modifier cette construction en se plaçant juste après l'étape de construction classique de chaque élément afin d'y apporter des modifications visant à permettre la capture des informations nécessaires à l'analyse.

Enveloppement de méthode (Method Wrapper)

Il s'agit tout simplement d'intercepter les appels aux méthodes existantes et d'ajouter de nouveaux comportements autour d'elles, à savoir avant l'entrée dans la méthode et après sa sortie. La méthode ne doit en aucun cas être impactée par cet ajout de fonctionnalités, cela doit être transparent pour elle. Le désavantage c'est qu'il faut avoir accès au code source. Cette fonctionnalité pourrait être dans le cas qui nous préoccupe, l'extraction d'informations pour l'analyse.

Dans le monde orienté objet, cela peut typiquement se faire grâce au patron de conception « Proxy », expliqué dans [Gamma et al., 1994] : ce patron a pour objectif la conception d'un objet qui se substitue à un autre objet (le sujet) et qui en contrôle l'accès. L'objet qui effectue la substitution (le proxy) possède la même interface que le sujet, ce qui rend cette substitution transparente vis-à-vis des clients (ceux qui interagissent avec le sujet). Donc un client envoie un appel de méthode, qui passe par le proxy qui effectue ce pourquoi il a été rajouté, proxy qui va ensuite déléguer l'appel vers le sujet. Une fois que le sujet a fini, l'appel revient vers le proxy, qui ensuite rend la main au client.

Il est à noter que certaines implémentations d'outils permettant la POA utilisent cette manière de faire pour mettre en place les aspects.

Approches ad-hoc

Tout ce qui vient d'être vu est structuré et éprouvé. Malgré cela, on voit bien que ce n'est pas bénin à mettre en place et dans certains cas très ponctuels ou très limités, il n'y a pas besoin de mettre en place toute cette machinerie. Pour ces besoins-là, Il peut être décidé d'avoir recours à des instrumentations manuelles ou un peu plus « artisanales », comme l'utilisation de scripts.

C'est bien entendu une solution à court-terme et à appliquer que pour des intérêts limités et réduits. Si ce n'est pas le cas, cela provoquera plus de problèmes que ça n'apportera de solutions, ceci pour les raisons évoquées tout au long des différentes approches : par exemple, la non possibilité de la solution à grossir (scale), grossissement qui provoquerait une très nette diminution des performances.

2.5 Conclusion

Nous avons vu notamment qu'une des caractéristiques est l'abondance des données qui seront récoltées. Un travail important à faire sera donc de porter une attention particulière à ne pas capturer des données inutiles ou juste parce qu'elles sont disponibles.

Ce travail sera aussi à faire pour une des conséquences de la mise en place de l'analyse dynamique et qui est son impact sur l'exécution du programme, car cela ne se fait pas de manière complètement transparente et gratuite. Le fait d'avoir accès aux propriétés dynamiques a un coût non négligeable. De toute manière, s'il est négligé, son coût n'en sera que plus élevé.

Il s'agit vraiment ici d'un travail de recherche de l'équilibre, tel un créateur de parfum mettant en place une nouvelle senteur: il faut un peu de ceci, un peu de cela. Si ce n'est pas bon, il convient d'effectuer des réajustements, ceci toujours dans l'optique de la recherche d'un équilibre entre tous ses constituants. Concrètement, c'est ce qui s'est passé lors de la réalisation de l'analyseur.

La variété de techniques à mettre en œuvre offre un éventail de possibilités qui devrait permettre à chacun de trouver celle qui convient le mieux à son besoin.

3. Visualisation

Ce chapitre va présenter les différents thèmes sous-jacents et à prendre en compte lors de la définition d'une visualisation. La problématique sera tout d'abord abordée et illustrée à l'aide des résultats d'études. Le processus de compréhension de programmes sera ensuite exposé ainsi que les différentes approches possibles. Après, sera décrit ce qui est recherché par la visualisation des programmes ainsi que les caractéristiques essentielles qu'elles se doivent d'avoir. Une analyse des métaphores utilisées dans l'informatique sera présentée. Finalement, une comparaison entre les rendus d'une visualisation en 2D et en 3D sera faite.

3.1 Introduction

Dans [Gershon et al., 1997], la visualisation y est définie comme : « *la visualisation lie les deux plus puissants systèmes de traitement de l'information connus à ce jour : l'esprit humain et l'ordinateur moderne.* » ainsi que par : « *la visualisation est plus qu'une méthode de calcul. Il s'agit du processus de transformation de l'information en une forme visuelle, permettant aux utilisateurs d'observer l'information.* ».

Un logiciel est souvent vu comme une boîte noire, dans laquelle il n'est habituellement pas possible de voir ce qui se passe à l'exécution. La lecture du source code n'est pas la manière la plus rapide pour comprendre un programme mais c'est bien lui qui spécifie le comportement du programme. La visualisation est généralement plus adéquate pour aider et supporter la démarche de compréhension.

Dans [Koschke, 2003], une enquête a été menée afin de savoir qui utilisait des solutions de visualisation, si cela était important et quelles formes devait-elle prendre. Il en résulte que 42% des participants ont répondu que cela était important, 40% ont jugé que cela était absolument nécessaire et le pourcentage restant comprend des réponses allant de « ça peut être utile » à ce n'est pas important. Cela représente quand même 82% de personnes qui trouvent cela important. Les profils ayant répondu à ce sondage étaient des personnes impliquées dans la maintenance logicielle, la rétroingénierie, la réingénierie et dans les processus de mesure, il s'agit donc de profils très ciblés. Quant à la forme que la visualisation prenait, ils sont une écrasante majorité à avoir répondu que c'était des graphes, viennent ensuite le texte formaté, les diagrammes UML, ainsi que d'autres moyens comme les matrices, les arbres, etc. Il est intéressant de constater que 59% d'entre eux sont satisfaits de cette visualisation, contre 22% qui estiment qu'elle ne répond pas entièrement à leurs besoins et 6% qui n'en sont pas satisfaits, le reste n'ayant pas d'avis.

Il y a un paradoxe intéressant à mettre en avant ici : la visualisation utilise abondamment les ordinateurs et joue un rôle important dans l'aide à la compréhension. C'est même une discipline à part entière, appelée « amplification de l'intelligence » ou « augmentation cognitive » ou encore « intelligence augmentée par les machines » ([Wikipedia, 2014d]), à ne pas confondre avec « l'intelligence artificielle » qui a un but différent. Les informaticiens ont développés des outils très sophistiqués afin d'apporter la visualisation pour diverses disciplines d'ingénierie,

scientifiques, pour l'imagerie médicale, etc. Le paradoxe est que ces mêmes informaticiens n'utilisent que très peu la visualisation pour les aider dans l'implémentation, la maintenance ou encore la compréhension de systèmes complexes. Dans [Diehl, 2007], il est expliqué que souvent les développeurs s'adaptent à la représentation fournie par la machine, plutôt que le contraire ; on peut directement penser à la programmation en langage machine notamment.

Une autre explication donnée dans [Reiss et al., 2003] résume bien la visualisation logicielle, et rejoint même ce qui était dit sur l'analyse : la visualisation logicielle n'a à présent pas été d'une grande aide quant à la compréhension des programmes. Les visualisations qui s'intéressent aux aspects statiques d'un logiciel ne fournissent qu'une vue limitée et ne disent rien sur les comportements dynamiques complexes. Les visualisations dynamiques sont trop coûteuses car complexes à mettre en place et surtout elles requièrent que l'exécution du programme se déroule dans un environnement stable, qui produit les données nécessaires à la visualisation et provoquent des ralentissements de l'exécution du programme. Cela ne donne donc pas envie aux programmeurs ou concepteurs de les utiliser.

Il va être ici présenté les constituants importants qui font qu'une visualisation est plus ou moins efficace. Il ne sera donc pas exposé une liste des outils et des différentes techniques de visualisation ; à ce sujet, il est plus intéressant de se référer au travail déjà effectué dans [Caserta, 2012] et [Storey et al., 2005] notamment. Bien entendu, certaines techniques seront évoquées, mais elles ne seront pas détaillées. Il est plus intéressant d'en comprendre les tenants et aboutissants ainsi que l'aide que la visualisation peut apporter à la compréhension des programmes.

3.2 Compréhension des programmes

Dans le cadre du présent travail, la visualisation a pour but ultime l'aide à la compréhension des programmes. Il convient donc d'expliquer ce que cette compréhension signifie ainsi que son fonctionnement. La compréhension des programmes est une activité menée par les ingénieurs logiciels afin d'essayer de comprendre un programme. Comprendre signifie, à ce niveau, comprendre le code. Dans quels buts ? Principalement pour assurer le processus de maintenance, pour déboguer ou même encore pour réutiliser du code existant.

Pourquoi est-ce devenue nécessaire et un sujet de recherche très actif ? Dans [Storey et al., 1997] plusieurs raisons sont évoquées, dont le fait que certains très vieux programmes sont toujours maintenus et sont devenus difficiles à comprendre, que les programmes ne sont plus développés par une équipe stable mais voient défiler plusieurs développeurs ayant des styles de programmation et des expériences variées, que les programmes utilisent beaucoup de technologies différentes et ont tendance à grossir en taille. Un autre point important est le fait que le code source d'un programme devient au fil du temps la seule documentation à jour et complète, sans pour autant être correcte par rapport aux spécifications originales du programme.

Beaucoup de stratégies ont été développées pour aider le programmeur dans son cheminement quant à la compréhension d'un programme mais aucune n'est parfaite : il n'y a en effet pas de technique ultime permettant de couvrir tous les aspects d'un programme. Il faudra que le programmeur compose avec ces techniques, en les limitant afin de ne pas être surchargé d'informations, afin de construire une représentation la plus adéquate et complète possible. Il

faut rajouter à cela que tous les programmeurs, humains avant tout, ont des comportements différents face à cette tâche et n'ont pas tous besoin des mêmes informations.

Dans [Storey et al., 1997] et [Dunsmore, 1998], une liste de modèles cognitifs pour la compréhension des programmes est établie et propose de décrire le comportement des programmeurs dans ce processus de compréhension :

- Approche « bottom up » : c'est l'approche la plus classique et la plus instinctive dans ce cadre. Il s'agit de partir du code source, de former mentalement des boîtes, des ensembles qui sont des abstractions de plus haut-niveau ayant plus de signification. Ces abstractions correspondent généralement aux différents blocs fonctionnels et sont eux-mêmes regroupés jusqu'à ce qu'une compréhension haut-niveau du programme soit formée. Un modèle, une représentation abstraite du programme est donc créée. Il a été remarqué que cette approche est presque toujours utilisée lorsque le code examiné est tout à fait nouveau pour le programmeur;
- Approche « top-down » : il s'agit ici de partir d'une certaine connaissance du domaine fonctionnel de l'application et de le relier au code source. Pour réaliser cela, des hypothèses à propos du programme sont faites et sont vérifiées ou rejetées par le code source.

  Contrairement à l'approche « bottom-up », cette approche est généralement utilisée quand le programme ou que le genre du programme est plus ou moins familier. Cette approche est plutôt utilisée par des programmeurs expérimentés qui reconnaissent plus facilement les différentes structures et parties du programme ainsi que les conventions généralement d'usage pour ce type de programme ;
- Approche systématique et selon les besoins : la partie systématique décrit le fait de parcourir le code en détails en étant particulièrement attentif aux structures de contrôle du flux ainsi qu'aux flux de données, afin d'obtenir une compréhension approfondie du code source. La partie « selon les besoins » utilise le même processus mais en se concentrant seulement sur les parties de code qui intéresse le programmeur. Ce type d'approche est difficile à mettre en place pour des gros systèmes et est plus sujet aux erreurs car moins précis ;
- Approche dite intégrée : il s'agit d'une combinaison des trois approches précédentes afin de construire un seul modèle. Il est proposé qu'une compréhension détaillée soit construite en passant d'une approche à une autre, ce qui permet que les niveaux d'abstractions différents se complètent l'un l'autre. Cette approche est utilisée quand le code source et le programme commence à devenir familier. Il s'agit donc d'une étape avancée dans le processus de compréhension car un recoupement est effectué en prenant en compte plusieurs niveaux.

À cela, il convient de rajouter un facteur très important dans le processus de compréhension et qui n'est autre que le temps. De par l'expérience, on se rend compte qu'il faut laisser le temps au cerveau de « digérer » toute cette information et surtout, qu'il puisse faire les liens entre les niveaux d'abstraction. Dans aucun des articles de recherche lus il n'en est mention. Les différentes techniques exposées sont là pour guider le programmeur afin que le temps qu'il passe dans sa recherche de la compréhension soit le plus efficace possible et surtout que son approche soit structurée.

3.3 Visualisation des programmes

Dans l'ingénierie logicielle, il y a deux disciplines qui concerne la visualisation et qui sont souvent confondues l'une avec l'autre. Il s'agit de la « programmation visuelle » et de la « visualisation des programmes ». Il est question ici de cette dernière et concerne le fait d'utiliser des représentations graphiques afin de mettre en évidence certains aspects du code, ou en ce qui nous concerne, certains aspects d'un programme à l'exécution.

Une première définition intéressante est donnée par [Roman et al., 1992] : « *la visualisation des programmes est l'extraction d'information à propos de certains aspects d'un programme et leurs présentations sous une forme graphique.* ». Une seconde définition est donnée par [Petre et al., 1998] : « *il s'agit du fait d'essayer de trouver la simplicité dans un artefact complexe (par exemple plusieurs milliers de lignes de code), afin d'en produire une représentation sélective.* ».

C'est une discipline qui a commencé dans les années 1970 et qui au départ proposait une des formes les plus simples de visualisation d'un programme, à savoir le « pretty-printing », qui est la mise en forme plus claire du code source en améliorant sa disposition, en utilisant des couleurs et des polices de caractères différentes, etc. Le but recherché était la lisibilité avant tout.

Avec le développement des nouvelles technologies et l'apparition des interfaces graphiques, les diverses représentations ainsi que les outils se sont multipliés, devenant plus complexes et variés.

Comme mentionné dans [Dunsmore, 1998], cette discipline bénéficie encore aujourd'hui d'un attrait important et se concentre principalement sur :

- la maintenance des programmes : comme déjà introduit, du fait que les systèmes sont de plus en plus complexes, que les développeurs ne sont pas les mêmes tout au long de la vie de ces systèmes et que la documentation n'est pas toujours fiable, c'est un point sur lequel des efforts sont à fournir. Généralement, les techniques mises au point permettent une exploration du code source et de la structure interne d'un programme, code source qui est alors enrichi d'informations glanées statiquement ou dynamiquement;
- les systèmes distribués et parallèles : depuis plusieurs années, il y a une forte augmentation de l'utilisation de ce genre de systèmes, ce qui apporte son lot de problèmes et de concepts à maîtriser. Cela n'étant pas évident du tout, la visualisation est toute indiquée pour aider à cette tâche;
- la découverte de nouvelles métaphores : comme nous le verrons un peu plus loin, les métaphores sont largement utilisées par les humains et surtout dans l'ingénierie logicielle car elles permettent de mieux appréhender, de concrétiser, quelque chose d'abstrait.

Le but est donc d'afficher des informations abstraites et leurs relations d'une manière naturelle et intuitive. Parmi toutes les techniques existantes, ces dernières suivent généralement un des deux grands principes suivants, donnés par [Diehl, 2007] :

- l'exploration interactive : cette visualisation interactive permet à l'utilisateur d'avoir une vue d'ensemble, ensuite s'il le désire, de zoomer et de filtrer l'information pour avoir les détails quand il le souhaite ;

- la focalisation + le contexte : ici, une visualisation détaillée d'un aspect ou d'une partie est directement proposée (la focalisation) et est intégrée dans la visualisation du contexte. Cette technique fournit donc en même temps un aperçu et des détails.

3.4 Caractéristiques recherchées

Maintenant que nous en savons un peu plus sur ce qu'est la visualisation de programmes, il est peut-être temps de se demander ce qui peut constituer une bonne visualisation, ce que sont les propriétés qu'elle doit supporter et les caractéristiques qu'elle doit respecter.

Avant toute chose, il convient de savoir ce pourquoi on veut créer une visualisation, quelle va être sa signification. Une visualisation ne peut raisonnablement pas rendre compte de tous les aspects d'un programme. Pour cela, il est opportun de se poser cinq questions apportées par [Maletic et al., 2002] et répondant chacune à un élément permettant de résumer et de circonscrire ce qu'une visualisation apporte:

- ses tâches : pourquoi est-ce que cette visualisation est nécessaire ? ;
- son audience : qui va utiliser cette visualisation ? ;
- sa cible : quelle est la source de donnée à représenter ? ;
- sa représentation : comment va être représentée cette cible ? ;
- le support : avec quel support cette visualisation devrait-elle être représentée ?.

Nous allons ensuite évoquer des besoins qui ont été identifiés dans [Reiss et al., 2003]. Dans cet article, il est suggéré qu'il faut absolument tout faire pour que la visualisation proposée soit conviviale avant tout, même si cela se fait aux dépens des détails ou bien de la qualité de la visualisation. Ils estiment qu'il faut principalement :

- maximiser l'information affichée… : la visualisation doit permettre d'afficher autant d'informations que possible d'un coup. Cela permet au programmeur de ne pas constamment devoir ajuster, chercher ou naviguer dans la visualisation afin de trouver des informations utiles ;
- … tout en proposant une vue compacte : afin de proposer une utilisation pratique en combinaison avec d'autres outils, cette visualisation doit prendre le moins de place possible pour pouvoir être affichée en même temps que le code source par exemple. De cette manière, il y a une correspondance directe entre les deux.

Dans [Gračanin et al., 2005] et [Young et al., 1998], il est question cette fois-ci des qualités qui rendent une visualisation efficace et expressive. Il vous en est proposé ici une version fusionnée et épurée :

- Définir l'étendue de la visualisation : il convient bien entendu de mettre en avant un ou certains aspects d'un programme mais on ne peut décemment par tout représenter. Il faut donc bien définir ce qui est visé. Cela peut être la partie statique ou dynamique, les flux de données, les dépendances ou autres. Cela doit se faire très tôt dans le processus d'élaboration d'une visualisation ;
- Proposer une navigation simple : la visualisation se doit d'être structurée. L'utilisateur doit pouvoir comprendre ce qui est représenté et à quel niveau d'abstraction il est. Il doit pouvoir se déplacer dans cette visualisation sans pour autant s'y perdre ;

- Avoir une complexité visuelle faible : il faut évidemment tout faire pour que la visualisation soit la plus simple visuellement parlant pour qu'elle soit la plus efficace possible. Cela passe par une organisation de la répartition de l'information la plus adéquate possible. Il faut bien se rendre compte que ce point-ci dépend fortement de la complexité de ce qu'il faut visualiser ;
- Varier le niveau de détail : comme nous l'avons, lors du processus de compréhension nous ne sommes pas toujours intéressés par les mêmes informations. Il doit donc être possible de laisser le choix de cacher ou d'afficher certaines informations en fonction de l'intérêt de l'utilisateur ;
- Une certaine résilience au changement : pour un même programme, selon le point d'intérêt, une visualisation ne devrait pas être complètement différente en fonction qu'on ajoute certaines informations. Cela dans le but de ne pas désorienter l'utilisateur sous peine qu'il ait l'impression d'être face à quelque chose de complètement nouveau ;
- Éviter la surcharge d'information : il faut bien entendu être complet sans pour autant que cela rende la visualisation trop lourde ou bien même que cela écrase l'utilisateur ;
- Utilisation adéquate des métaphores visuelles : les métaphores permettent de présenter de l'information en utilisant des concepts familiers aux utilisateurs. Cela permet à ces derniers de mieux appréhender la visualisation. Il faut que cette métaphore visuelle soit consistante (on ne peut pas se servir de la même métaphore pour tout représenter) et riche en expressivité (autrement elle perd directement de son intérêt). Les métaphores visuelles seront abordées plus en détails dans les points suivants ;
- Permettre une certaine interactivité : dans le processus de compréhension, l'utilisateur doit pouvoir s'approprier la visualisation. Cette compréhension va évoluer et l'utilisateur doit pouvoir enrichir la visualisation avec ce qu'il apprend, pouvoir mettre des commentaires ;
- Un certain niveau d'automatisation : c'est un aspect important qui est souvent oublié, mais la visualisation doit pouvoir être générée automatiquement et ce le plus facilement possible. Il faut que l'intervention de l'utilisateur soit la plus rare et qu'elle ait le moins d'impact possible sur les résultats. Il faut donc bien veiller à avoir ce point en tête.

3.5 Les métaphores

Tout d'abord la définition d'une métaphore est la suivante : « *Emploi d'un terme concret pour exprimer une notion abstraite par substitution analogique, sans qu'il y ait d'élément introduisant formellement une comparaison.* »[4]. Le but d'une métaphore est donc d'évoquer des images mentales afin de mieux visualiser des concepts et d'exploiter des analogies pour mieux les comprendre.

L'informatique recourt souvent à des métaphores : il suffit de penser aux arbres, aux feuilles, aux queues, aux files, aux dossiers et tant d'autres. Le but de la visualisation logicielle est bien d'évoquer ces images qui permettent de mieux comprendre les logiciels. Trouver des métaphores va donc permettre non seulement de produire de meilleures visualisations mais

[4] http://www.larousse.fr/dictionnaires/francais/m%C3%A9taphore/50889

aussi d'améliorer la manière donc on communique à propos des logiciels. Une métaphore donne une forme aux différents aspects intangibles d'un logiciel.

Plus particulièrement, la métaphore visuelle va être utilisée dans ce domaine. Une définition intéressante est donnée par [Diehl, 2007] : « *une métaphore visuelle est une analogie qui est à la base d'une représentation graphique d'une entité ou un concept abstrait, et qui a pour but de transférer les propriétés de la représentation graphique vers celles de l'entité abstraite ou du concept.* ». Les métaphores visuelles affectent directement l'expressivité de la visualisation, pour autant que sa complexité visuelle n'impacte pas négativement l'efficacité de la visualisation.

Il convient néanmoins d'être particulièrement prudent quant à l'utilisation de métaphores car un problème qui survient régulièrement est le fait du transfert de propriétés de manière non intentionnelle. Nous nous retrouvons dès lors avec des concepts qui n'ont rien à voir avec le concept ou l'entité qu'on souhaite représenté.

Lors du choix d'une métaphore, il faut aussi être sûr que toutes les représentations visuelles qu'offre la métaphore suffisent à représenter tous les objets du domaine ou propriétés du sujet, et surtout, qu'elle puisse être utilisée de manière consistante. Souvent, la métaphore n'est pas assez riche et il va falloir combiner plusieurs métaphores afin que la visualisation désirée soit possible, ce qui engendre des incohérences, voir mêmes des incompréhensions.

Une bonne illustration de cela est la métaphore de la *fenêtre*, abondamment utilisée dans les interfaces utilisateur graphiques : ces entités n'en ont que la forme. Quand on pense à une fenêtre, on pense à pouvoir regarder au travers, à la fermer, à la couvrir avec un rideau. Mais essayez de déplacer votre fenêtre, de la minimiser, voir même de la superposer à une autre fenêtre. Vous surprenez vous souvent à lire du texte sur vos fenêtres ?

Un autre exemple, pour illustrer ici la consistance, est donné dans [Diehl, 2007] : dans Mac OS X[5], la métaphore de la *poubelle* est utilisée mais n'est pas consistante. Le fait de glisser un dossier dans la poubelle ne le supprime pas définitivement, il faut pour cela vider la poubelle. Maintenant glissez-y un disque dur et celui-ci est éjecté.

3.6 Rendu : 2D contre 3D

Les techniques de visualisation en 2D (deux dimensions) sont les plus courantes et les plus faciles à mettre en œuvre. Elles impliquent généralement l'utilisation de graphes ou d'arbres. Cette représentation souffre de lacunes lorsqu'il s'agit de présenter de manière claire des systèmes plus ou moins complexes car il faut alors afficher beaucoup de nœuds et d'arcs. Comme le support est plat, les éléments ont tendance à s'enchevêtrer et un travail cognitif supplémentaire est nécessaire afin de bien comprendre ce qui est représenté. Pour remédier à cela, il est généralement affiché des bouts du modèle, à charge de l'utilisateur de (dé)-zoomer et de naviguer dans le graphe. Il est aussi possible de superposer des couches d'informations : chaque couche représentant un aspect du programme. Vous comprendrez très vite qu'afficher toutes les couches rend la visualisation très lourde et qu'il est donc difficile d'avoir toute cette information en une fois. Ce qui est fréquemment mis en place pour palier à ce problème est

[5] https://www.apple.com/osx

l'utilisation de différentes vues du même modèle : chaque vue représente alors un aspect mais est affichée séparément des autres vues.

Le meilleur exemple pour illustrer cela semble être la technique appelée SHriMP (Simple Hierarchical Multi Perspective) décrite dans [Storey et al., 1997b]. Cette technique propose une approche intégrée combinant les fonctionnalités de zoom, de vue panoramique et même d'une fonctionnalité appelée « fisheye » qui permet, en fonction de la place du curseur, d'effectuer un zoom automatique du niveau de détails, tout cela afin de pouvoir supporter le plus de stratégies de compréhension possible.
Il est en outre possible d'appliquer des filtres sur le type d'information à afficher et d'avoir plusieurs vues en fonction du besoin. Les hiérarchies sont généralement affichées sur deux ou trois niveaux à la fois, le premier niveau étant représenté par un rectangle englobant les éléments du deuxième niveau et ainsi de suite. Ces derniers ne sont pas détaillés, il faut alors zoomer sur ceux-ci pour obtenir les détails. Comme d'habitude avec les graphes, les divers éléments sont reliés entre eux par des arcs, en utilisant un code couleur en fonction de la nature de ce que l'entité représente. Une particularité notable de cette technique est qu'elle met le code, en relation avec ce qui est observé, directement à disposition.

Voici ce que peut donner une implémentation de la technique de visualisation SHriMP. Il s'agit d'une implémentation proposée par le plugin Eclipse Creole[6]. Nous pouvons y apercevoir les différents éléments et fonctionnalités qui viennent d'être exposés :

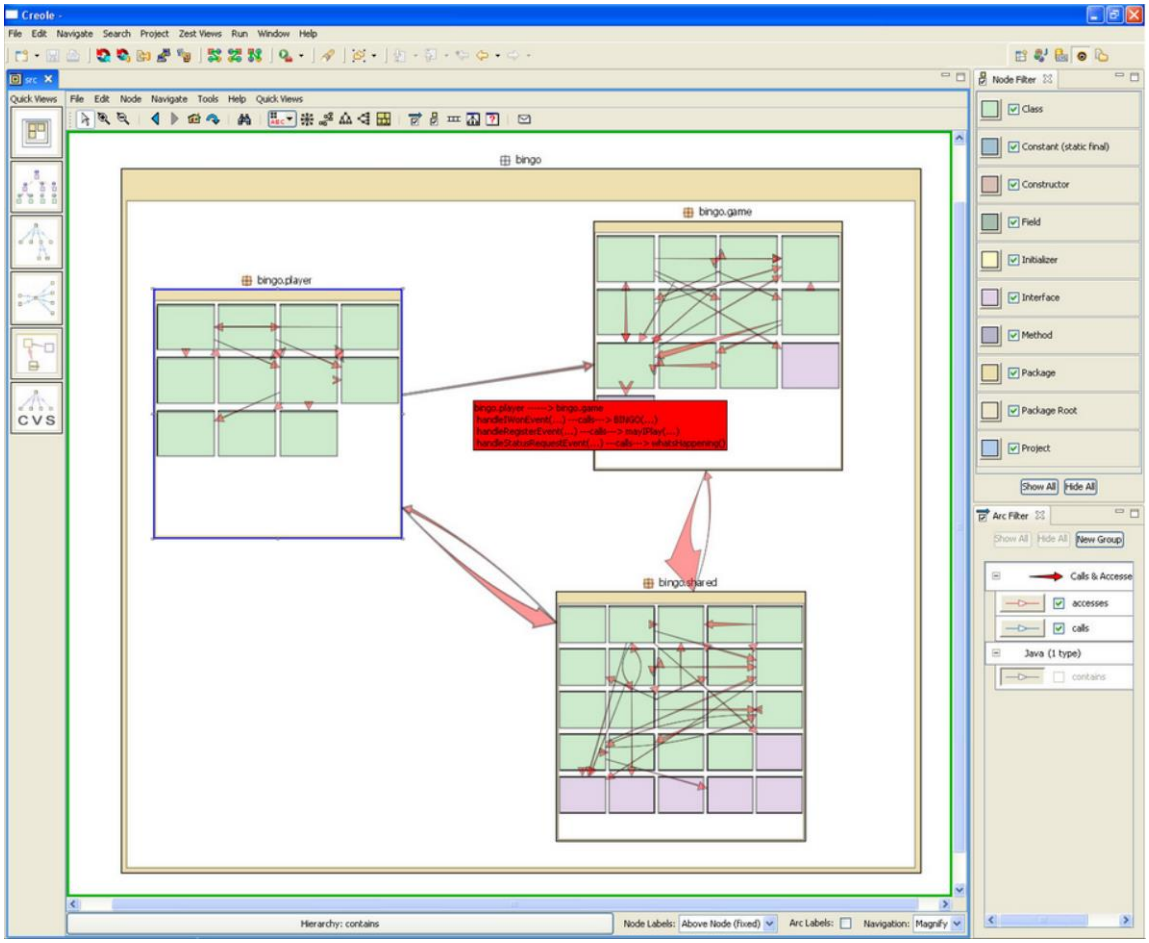


FIGURE 1 – VISUALISATION SHRIMP AVEC CREOLE.

[6] http://thechiselgroup.org/2003/07/06/creole

Un autre moyen de visualisation utilisé, et non des moindres, est la notation UML[7]. Les diagrammes UML sont souvent utilisés comme moyen de visualisation, notamment par les outils de rétroingénierie. UML présente le très gros avantage d'avoir un diagramme par aspect d'un programme ou d'un système qui doit être représenté. En outre, UML est facile à comprendre, peut être très expressif et est extensible : il est facilement possible d'ajouter des informations sur base des diagrammes existants. C'est cette approche qui a été choisie pour ce travail et qui sera présentée dans la Partie 2 concernant la réalisation.

Comme on vient de le voir, la visualisation en 2D provoque souvent une surcharge cognitive du fait de l'affichage de trop d'informations. Des chercheurs ont donc commencé à explorer d'autres voies et notamment celles qui visent à rajouter une dimension, amenant donc à utiliser la 3D (trois dimensions). La 3D offre l'avantage de pouvoir « donner du relief » aux éléments et surtout d'éviter les chevauchements.

Malgré le nombre élevé de techniques de visualisation en 3D et le fait que cela paraisse attirant voir même amusant, ces visualisations échouent à communiquer les informations importantes et donc à supporter le processus de compréhension d'un système ou d'un programme. Dans [Wettel et al., 2007], deux raisons majeures y sont évoquées et semblent assez compréhensibles : la première est le fait de la dimension supplémentaire : un réflexe naturel est de se dire qu'il faut en profiter pour rajouter de l'information. Malheureusement, cela amène le plus souvent à une surcharge de l'information et des problèmes de navigations. Une deuxième raison est qu'en 2D, le graphe est le moyen privilégié de représenter l'information. Ce qui arrive donc généralement est une transposition du graphe 2D dans un environnement 3D. Les nœuds et les arcs flottent alors et l'utilisateur peut naviguer, voler autour des parties intéressantes. Cette fonctionnalité ne semble pas naturelle et mène à la désorientation, ce qui fait naître une perception négative de la 3D à accomplir ce genre de tâche. De plus, cela n'est pas pratique à utiliser au quotidien. Il convient donc d'être prudent et de ne pas tomber dans ces travers.

Là où excelle la visualisation en 3D est la faculté à donner une vue d'ensemble assez claire et riche. Avec le bon angle de vue, la troisième dimension peut alors apporter un gain non-négligeable, permettant en un clin d'œil d'avoir une image beaucoup plus expressive sans pour autant surcharger le travail cognitif.

Nous proposons d'illustrer cela, et par la même occasion le point précédent qui parlait des métaphores, avec un exemple connu et éprouvé, celui de la métaphore de la ville, qui présente surtout la particularité de s'être adaptée et d'avoir pu exploiter ce qui vient d'être présenté.

3.6.1 Métaphore de la ville

C'est une métaphore qui a été abondamment étudiée comme le souligne [Caserta, 2012] page 96. L'idée n'est pas neuve et consiste à représenter des programmes orientés objet comme des villes habitables qu'il est possible d'explorer de manière intuitive comme si on se baladait dans une vraie ville.

[7] http://www.uml.org/

Métaphore fortement utilisée, elle l'est souvent de la mauvaise manière car les niveaux de granularité choisis ne sont pas souvent adaptés. Dans le paradigme orienté objet, deux des éléments principaux sont la notion de classes, et par la même, de packages dans lesquels sont rangées les classes. Ils semblent donc naturels d'utiliser ces deux concepts comme les deux niveaux de granularité possible. Il est possible d'aller jusqu'aux méthodes, aux propriétés, ce qui dépend de l'objectif recherché. Si c'est juste pour donner une vue d'ensemble et avoir un aperçu du design au niveau classes et packages, c'est suffisant. Il est alors possible de compléter cela par des outils plus orientés analyse du code source.

Sachant cela, il convient donc d'afficher les classes comme des bâtiments et les packages comme des quartiers de la ville. Concernant les packages, il s'agit de rectangles qui peuvent se superposer au fur et à mesure qu'on avance dans l'arborescence. Il y a donc un effet de niveau, qui forme la topologie, qui permet de décrire la profondeur. Chaque classe est aussi un rectangle et est située à l'intérieur du quartier auquel elle appartient, son package.

Il reste qu'il est encore possible d'utiliser des attributs de ces formes et de cette métaphore pour apporter d'autres informations, toujours en gardant en tête le fait de ne pas surcharger la visualisation. Cela tombe bien car il y a un type d'information intéressant, varié et facile à afficher : les métriques logicielles. Il est notamment possible d'utiliser la hauteur, la largeur et même la couleur d'un building pour représenter certaines métriques. Dans [Wettel et al., 2007] par exemple, ils ont fait le choix d'utiliser la largeur de la base d'un building pour représenter la quantité d'attributs qu'a une classe. La hauteur du building représente la quantité de méthodes qu'a une classe. Cela permet de directement repérer les classes importantes dans le système par exemple. La couleur peut être utilisée pour représenter l'intensité à laquelle les méthodes d'une classe sont sollicitées. Tout cela étant bien entendu dépendant des choix faits quant à la manière d'implémenter la métaphore. Dans la plupart des outils, ce sont des options configurables par l'utilisateur.

Au niveau de l'interactivité, il est possible d'obtenir des informations diverses en sélectionnant un quartier ou un bâtiment, même des informations agrégées quand plusieurs sont sélectionnés.

Il est à noter que dans la plupart des implémentations, la 3D n'est pas utilisée à son maximum pour cette métaphore. Pour éviter le problème de désorientation, il est fait le choix d'avoir une base plane, fixe, sur laquelle sont posés des éléments. Cela évite de devoir naviguer sur l'axe verticale ce qui pourrait devenir vite laborieux.

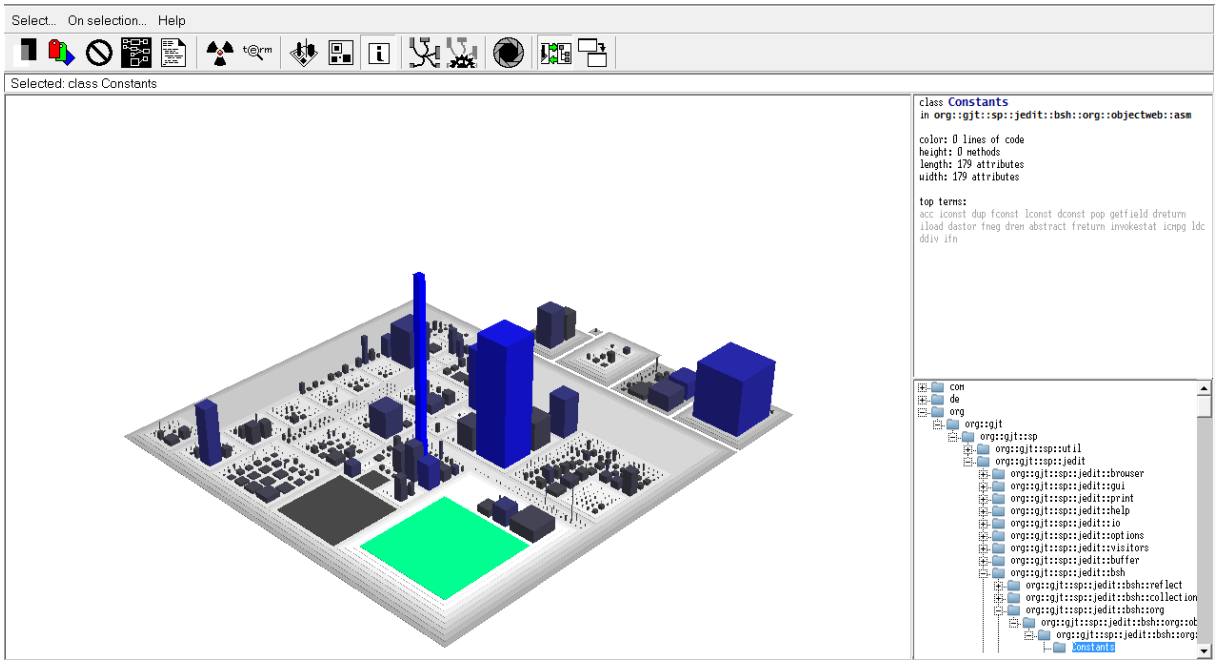


FIGURE 2 – VISUALISATION 3D DU PROGRAMME JEDIT À L'AIDE DE CODECITY ET DE LA MÉTAPHORE DE LA VILLE.

Voici ci-dessus une capture du programme CodeCity[8] , qui propose une application de la métaphore de la ville. La ville représente le programme jEdit[9] qui est un éditeur de texte écrit en Java. Le programme fait un peu moins de 100.000 lignes de code et est composé d'un package principal (le plus grand quartier) et de plus petits packages qui sont des librairies (les autres quartiers). Un rectangle correspondant à la classe *Constants* est sélectionné (en vert) et que dès lors des informations relatives à cette classe sont affichées en haut à droite. De plus, et ce automatiquement, sa place dans l'arborescence est affichée en bas à droite. Il faut aussi noter qu'il y a plusieurs classes qui comprennent beaucoup de méthodes (les bâtiments les plus hauts) et que deux classes, dont celle sélectionnée, contiennent beaucoup d'attributs.

[8] http://www.inf.usi.ch/phd/wettel/codecity.html
[9] http://www.jedit.org

3.7 Conclusion

Comme on a pu le voir, il s'agit d'un domaine extrêmement vaste qui fait appel à plusieurs disciplines. De même, les techniques disponibles sont nombreuses et ont chacun leurs forces et leurs faiblesses. Malgré cela, un défaut à la visualisation de ce qui se passe à l'exécution des programmes demeure: le fait qu'il soit difficile d'avoir l'ensemble des informations nécessaires avec une seule vue. À cet effet, il convient de combiner ces vues, qui sont donc complémentaires et présentent chacune un aspect, une réalité à la fois d'un programme.

Le processus de compréhension et de découverte n'est pas spécifique à notre discipline mais s'applique à notre quotidien. Il y a cependant des points importants dans la structure d'un programme qui peuvent être identifiés, par lesquels on peut démarrer et autour desquels il est possible de construire sa compréhension. Comme quand on défait une pelote de laine, il convient d'abord de trouver le début ou la fin et ensuite de remonter le fil. La démarche est analogue.

On a pu noter aussi le fait que, et ce dû à des raisons pratiques et historiques, nous tendons à utiliser les graphes de manière automatique pour les représentations d'un système. À nouveau, il s'agit ici en fait de simplement mettre en forme, faire apparaître sous une forme imagée, sans transformation quelconque, la structure sous-jacente. Par exemple, pour visualiser un graphe d'objets, nous faisons une transposition directe des nœuds et des arêtes, et la visualisation offerte se limite à cela. Nous avons pu voir qu'avec la métaphore de la ville, par exemple, il est possible d'avoir d'autres visualisations.

Une difficulté majeure dans la démarche de constitution d'une visualisation intéressante est de savoir quoi faire face à la quantité de données généralement disponibles concernant l'exécution d'un programme. On se rendra encore mieux compte de cela dans la « *Partie II: Réalisation* », car cela a été un problème récurrent avec lequel il a fallu faire preuve d'imagination et surtout d'un sens aigüe de la rationalisation: la cohérence et la parcimonie sont au cœur de cette démarche.

PARTIE II : RÉALISATION

Aperçu global

La solution imaginée et mise en place va maintenant être présentée. Cette solution doit faciliter la compréhension d'un logiciel et ce de manière localisée. Cela va se faire aux moyens d'informations récoltées lors de l'exécution du logiciel et ensuite, par la consolidation, l'interprétation et la présentation des informations récoltées de manière à ce qu'elles soient pertinentes et facilement assimilables.

Concrètement, l'information nécessaire sera collectée à l'aide d'un petit programme Java, un agent pour être plus précis, appelé AnaDONE. Ce dernier a la particularité de s'exécuter avant que le programme à analyser soit démarré, et même de récolter des informations sur son contenu ainsi que de modifier ce contenu afin de capturer certains événements. Ces informations seront écrites dans un fichier. Il faut ensuite pouvoir visualiser ces informations, ce qui se fera à l'aide de MetaDONE et du plug-in VizuDONE développé pour l'occasion, qui importera le fichier et affichera de manière structurée l'information.

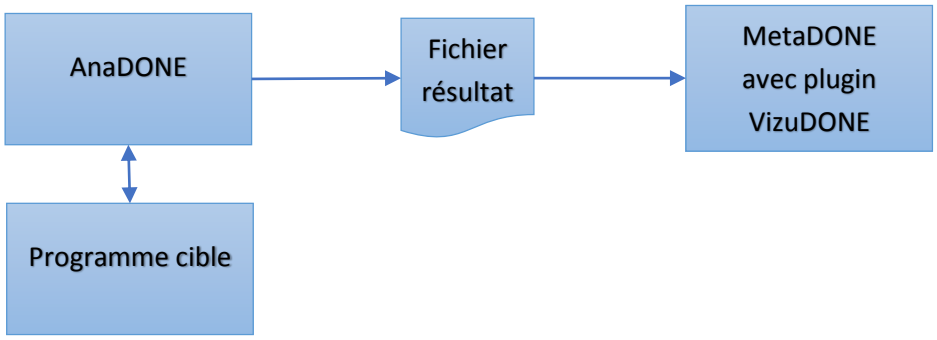


FIGURE 3 – SCHÉMA DES COMPOSANTS DE LA SOLUTION PROPOSÉE.

Un gros accent a été mis sur le fait que les différents composants de la solution proposée se doivent d'être performants et extensibles. En effet, cette solution doit pouvoir être enrichie lors de recherches futures. La performance est au centre de cette réalisation, qui peut aussi être vue comme une étude de faisabilité, car comme on a pu le noter dans la *Partie I : État de l'art*, c'est un défaut récurrent lorsque les aspects dynamiques sont étudiés, cela étant dû au fait de l'abondance des données.

De plus, il faut que les différentes étapes qui constituent le flux complet, à savoir exécuter l'analyse du programme cible et lancer la visualisation, nécessitent le moins d'opérations manuelles possibles. L'automatisation du processus doit donc être la plus avancée possible. Avec la solution proposée, il s'agit seulement de lancer l'analyse par l'intermédiaire d'AnaDONE et de charger le fichier résultat avec MetaDONE.

4. Analyse

Ce chapitre présente l'analyseur que nous avons élaboré, AnaDONE. Il permettra de récolter les informations pour notre visualisation. Un ensemble de techniques et de technologies va d'abord être présenté afin de pouvoir comprendre de quoi est constitué l'analyseur et ce qu'il est possible de faire. C'est ensuite au tour du fonctionnement même d'AnaDONE d'être exposé ainsi que des diverses fonctionnalités qui ont lui ont été ajoutées. Certaines problématiques inhérentes à ce genre de pratique, comme les cycles, seront expliquées. Une métrique dynamique sera présentée plus en détails ainsi que la manière de la calculer.

4.1 Introduction

Le but est de développer un outil capable de récolter l'information nécessaire à la compréhension d'un logiciel. À cette fin, AnaDONE a été mis en place et se veut le plus transparent possible afin d'éviter tout impact négatif sur l'exécution du programme cible, ainsi qu'extensible et configurable.

Il convient de ne pas oublier qu'il s'agit de l'élément crucial car c'est lui qui récolte et fournit les données nécessaires à la visualisation. En ne maîtrisant pas ce qui est fait, on risque d'avoir des données erronées.

Afin de ne pas déverser un flot trop important de nouveaux concepts et de technologies, une sélection d'éléments clés, afin que vous puissiez appréhender complètement ce qui est proposé, a été opérée ; éléments qui seront présentés de la manière la plus claire possible. Diverses illustrations et diagrammes viendront, autant que faire se peut, éclaircir et concrétiser ce qui sera abordé.

On pourra notamment constater qu'un soin important a été apporté quant à l'explication des choix faits et des solutions qui ont été retenues ou non. Le chemin qui a mené à la version d'AnaDONE ici présentée a été long et beaucoup de pistes ont été essayées. Pas mal d'obstacles ont été rencontrés, dont le plus important est la masse de données collectées, obstacles qui ont fait l'objet d'une attention particulière et dont les solutions ont été élaborées en ayant conscience des conséquences ou non qu'elles ont.

4.2 Instrumentation

L'instrumentation est la possibilité d'injecter du code dans un programme déjà compilé. Il est alors possible de modifier le code afin d'y introduire de nouvelles fonctionnalités comme le traçage d'événements ou la surveillance de l'exécution de méthodes. Cela peut se réaliser de deux manières différentes :

- statique : le bytecode d'une classe est modifié et le résultat est sauvegardé de manière permanente sur le disque, en plus de l'original;

- dynamique : le bytecode d'une classe est ici modifié en mémoire, juste avant qu'il ne soit chargé.

C'est l'instrumentation dynamique qui a été choisie pour plusieurs raisons :

- l'instrumentation est réalisée à chaque démarrage : bien que cela implique un temps de chargement plus long, cela permet de facilement modifier notre outil afin de changer les éléments à prendre en compte pour le traçage ;
- la transparence : l'accès au code source n'est pas nécessaire. En outre, l'instrumentation statique double tous les fichiers (originaux plus instrumentés). Souvent les programmes écrits en Java sont fournis sous forme d'archive jar ; il faudrait alors en extraire les fichiers *.class, les transformer, refaire une archive, etc. C'est fastidieux et peu répétitif ;
- l'efficacité : seules les classes réellement utilisées par l'application sont instrumentées. Cela permet d'être plus efficace. Avec l'instrumentation statique, ce n'est pas le cas, tout ce qui est possiblement utilisable est instrumenté. Cela peut donc avoir un impact substantiel sur les performances de l'application.

Dans [Gupta, 2008], il est expliqué où, en Java, l'instrumentation peut être effectuée. En Java, le code source est converti en bytecode, qui est une représentation intermédiaire indépendante de toute plateforme. Ce bytecode subit ensuite, à l'exécution, des conversions spécifiques à l'environnement dans lequel il est exécuté, pour être finalement exécuté sur la JVM. Cela offre trois possibilités : au niveau du code source, au niveau du bytecode et au niveau de la JVM.

Instrumentation de la JVM

Il y a plusieurs implémentations libres disponibles de la JVM ainsi que les accès à leurs codes sources. La difficulté ici est qu'il faut avoir une connaissance assez poussée du fonctionnement interne de la JVM, ce qui n'est pas courant. Par contre, on peut alors avoir accès à tous les aspects d'un programme Java en exécution. Autre atout de cette approche, c'est que le coût au niveau des performances est beaucoup moindre, car le tout est effectué par la JVM à un niveau plus bas, pour autant que cela soit optimisé. C'est une approche qui est utilisée pour les très gros projets ou dans les cas où un besoin intensif de l'instrumentation est fait. Un désavantage apparaît alors et est le fait qu'il faut suivre l'évolution de la JVM : quand une nouvelle version sort par exemple, il faut la retravailler à nouveau, ce qui demande du travail supplémentaire.

Instrumentation du bytecode

C'est la solution qui a été retenue dans le présent travail et qui est privilégiée dans la plupart des cas car extrêmement flexible. Il s'agit donc de manipuler le bytecode Java afin d'y insérer du code personnalisé qui va enregistrer les informations nécessaires à l'exécution. C'est la manière la plus simple de faire, qui ne demande pas de connaissance trop bas-niveau, mais qui provoque un léger, voir plus gros si c'est mal réalisé, coût à l'exécution. Si c'est l'instrumentation statique qui est choisie, il faut alors ré-instrumenter le code à chaque fois qu'il change. Dans le cas dynamique, cela se fait de manière automatique.

Utiliser JPDA

JPDA[10] (Java Platform Debugger Architecture) est un composant que Java possède par défaut, permettant de développer des applications de débogage pouvant s'intégrer avec différentes implémentations ayant cette fonctionnalité. C'est une approche qui permet d'obtenir facilement des informations spécifiques, même de bas-niveau, à propos du comportement à l'exécution d'un programme Java.

JPDA se compose de trois couches :

- JVM TI : Java VM Tool Interface définit les services de débogage que la JVM doit fournir. Cette interface fournit la notification d'événements concernant les opérations de bas-niveau de la JVM. Dès lors, un programme qui traite ces événements peut enregistrer des informations sur l'exécution d'un programme Java ;
- JDWP : Java Debug Wire Protocol qui définit comment la communication va se dérouler entre les processus du débogué et du débogueur. Il s'agit essentiellement de définir le format des informations et des requêtes transférées entre les deux. C'est ce composant qui permet au débogué et au débogueur de s'exécuter sur des VM séparées (même distantes) ;
- JDI : Java Debug Interface définit une interface haut-niveau disponible pour le langage Java et permettant d'écrire facilement des applications de débogage distantes.

Pour rendre tout cela plus concret, un exemple d'instrumentation vous est présenté ci-après. Il s'agit bien de mettre en lumière le résultat de l'instrumentation et se base donc sur un cas tout à fait fictif, qui ne sert que les besoins de l'explication.

Voici donc le code Java original des deux classes qui seront instrumentées:

```
public class ClassA {

    public ClassA() {
        //NOOP
    }

    public void methodA() {
        ClassB classB = new ClassB(" !");
        String result = classB.methodA("Hello");
        System.out.println(result);
    }

}
```

```
public class ClassB {

    private final String end;

    public ClassB(String end) {
        this.end = end;
    }

    public String methodA(String paramA) {
        return paramA + end;
    }

    public static void methodB() {
        //NOOP
    }

}
```

FIGURE 4 – CODE ORIGINAL SERVANT D'EXEMPLE POUR ILLUSTRER L'INSTRUMENTATION.

Ce code va maintenant être instrumenté de manière telle que toutes les entrées dans les constructeurs, les entrées dans les méthodes, les sorties des méthodes ainsi que les appels de méthodes seront pris en compte. Pour chacun de ces événements, une série de paramètres plus ou moins utiles seront passés.

[10] http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/architecture.html

```java
public class ClassA {

    public ClassA() {
        EventTracerImpl.oO.notifyConstructor("ClassA");
        //NOOP
    }

    public void methodA() {
        EventTracerImpl.oO.notifyMethodIn("methodA");

        ClassB classB = new ClassB(" !");

        EventTracerImpl.oO.notifyMethodCall("methodA", classB, "Hello");
        String result = classB.methodA("Hello");

        EventTracerImpl.oO.notifyMethodCall("println", System.out, result);
        System.out.println(result);

        EventTracerImpl.oO.notifyMethodOut("methodA");
    }

}
```

```java
public class ClassB {

    private final String end;

    public ClassB(String end) {
        EventTracerImpl.oO.notifyConstructor("ClassA", end);
        this.end = end;
    }

    public String methodA(String paramA) {
        EventTracerImpl.oO.notifyMethodIn("methodA", paramA);

        EventTracerImpl.oO.notifyMethodOut("methodA");
        return paramA + end;
    }

    public static void methodB() {
        EventTracerImpl.oO.notifyMethodIn("methodB");
        //NOOP
        EventTracerImpl.oO.notifyMethodOut("methodB");
    }

}
```

FIGURE 5 – CODE INSTRUMENTÉ SERVANT D'EXEMPLE POUR ILLUSTRER L'INSTRUMENTATION.

4.3 Agent Java (JAgent)

Un agent Java permet d'instrumenter de manière dynamique du code Java. Plus précisément, c'est un composant logiciel qui fournit, à une application, les moyens d'effectuer de l'instrumentation. Dans ce contexte, il s'agit de la capacité de redéfinir le contenu d'une classe qui est chargée à l'exécution. Ce composant est disponible depuis la version 1.5 (2004) de Java.

Il se matérialise par une archive jar contenant :

- une classe java qui déclare une méthode avec pour signature :
  ```
  public static void premain(String agentArgs, Instrumentation inst)
  ```
  Il s'agit du point d'entrée de l'agent. Au chargement de la JVM, après que cette dernière soit initialisée et avant que l'application principale soit démarrée, cette méthode est appelée. Il est possible d'implémenter d'autres signatures mais ce n'est pas pertinent dans notre cas et ça ne ferait qu'alourdir inutilement l'explication ;
- un fichier *META-INF/MANIFEST.MF* qui doit avoir une entrée *Premain-Class* indiquant l'endroit où se trouve la classe contenant la méthode *premain* (voir point précédent).

Au démarrage du programme à instrumenter, on va pouvoir attacher un agent grâce à l'option *javaagent* qui permet d'indiquer où se trouve l'agent en question :

```
-javaagent:/cheminversagent/agent.jar
```

Afin de pouvoir passer des arguments à la méthode *premain*, on peut lui envoyer une chaîne de caractères. Cette chaîne est reçue au travers de la variable *agentArgs* (voir la signature de la méthode *premain*). S'il faut passer plusieurs arguments, il faut les délimiter pour ensuite couper le contenu d'*agentArgs* aux bons endroits et récupérer nos arguments :

```
-javaagent:/cheminversagent/agent.jar=arg1;arg2;arg3
```

Dans la signature de la méthode *premain,* elle reçoit en paramètre un objet de type *Instrumentation*[11]. Cette interface fournit les services nécessaires afin d'instrumenter du code Java. Ce qui est important dans *Instrumentation* est la méthode

```
void
addTransformer(ClassFileTransformer transformer);
```

qui permet d'enregistrer auprès de la JVM un objet qui implémente l'interface *ClassFileTransformer*. Cette interface possède une seule méthode à implémenter donc :

```
byte[]
transform(  ClassLoader         loader,
            String              className,
            Class<?>            classBeingRedefined,
            ProtectionDomain    protectionDomain,
            byte[]              classfileBuffer)
   throws IllegalClassFormatException;
```

C'est cette méthode qui va être appelée par la JVM à chaque fois qu'elle charge une définition de classe (en dehors de celles avec lesquelles les *ClassFileTransformer* enregistrés ont une dépendance). Au moment où le flux arrive dans la méthode *transform*, la classe n'est pas encore complètement chargée : on peut d'abord la transformer comme désiré, ou ne rien faire aussi, et ensuite retourner le résultat de nos changements, ou *null* si rien n'a été changé. Il faut retourner un tableau d'octets, tableau qui contient la définition de la classe transformée,

[11] http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html

sérialisée sous forme d'octets. Si *null* est renvoyé, pour la JVM, c'est comme s'il n'y a eu aucune transformation et donc la définition initiale est chargée.

Voici un bon résumé avec une explication plus visuelle et en séquence de ce qui vient d'être exposé :

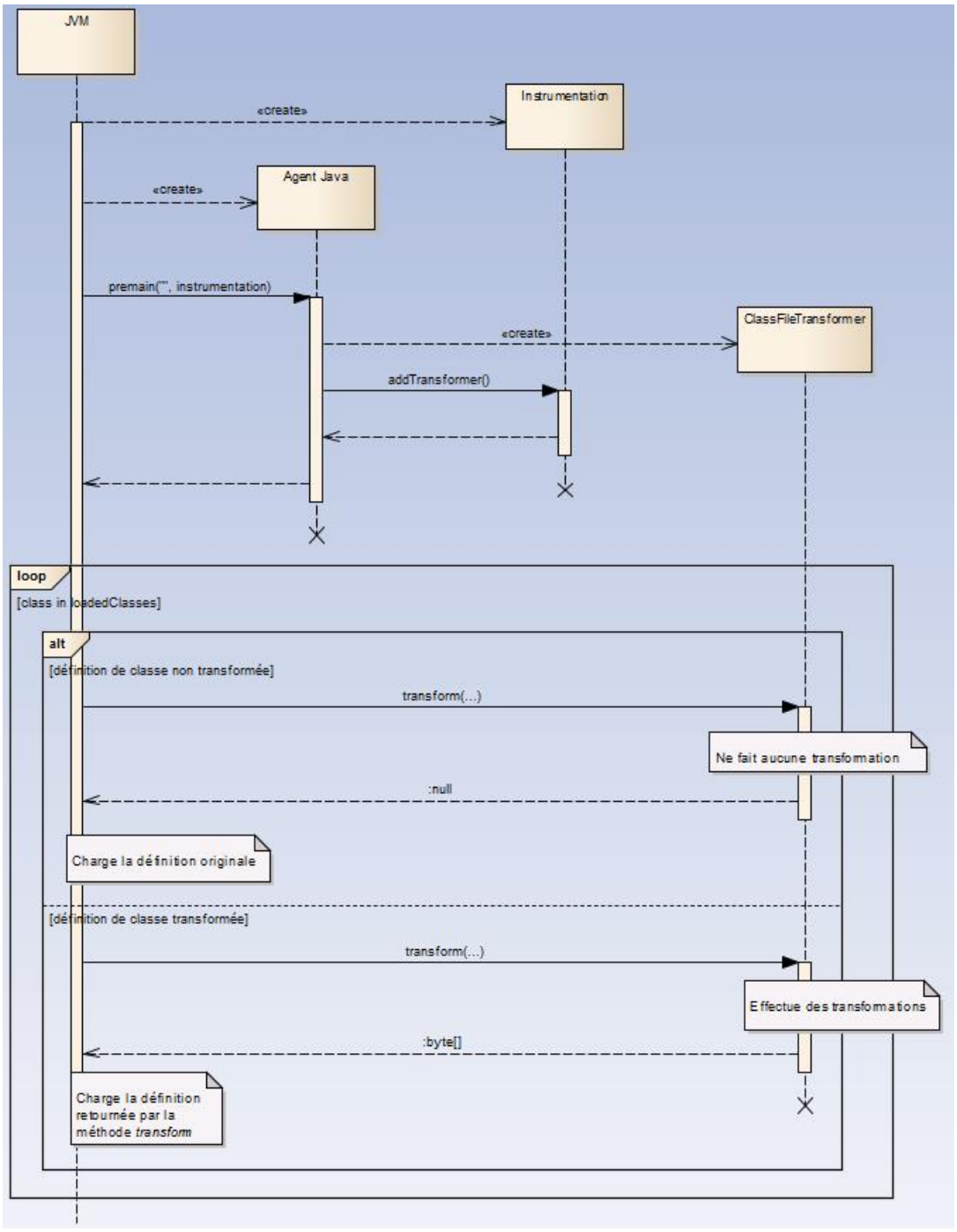


FIGURE 6 – DIAGRAMME DE SÉQUENCE EXPLIQUANT LE FONCTIONNEMENT D'UN AGENT JAVA ET DE SA RELATION AVEC CLASSFILETRANSFORMER.

4.4 Manipulation du bytecode

Plusieurs frameworks de manipulation du bytecode existent pour Java. Parmi les plus célèbres, ASM[12], Javassist[13] et BCEL[14]. Tous ces frameworks permettent de modifier des classes existantes ou de générer dynamiquement des classes, directement sous forme binaire.

ASM est trop bas-niveau, il implique la manipulation du bytecode en direct. Ce framework est réputé pour sa puissance et ses performances. Un point important à noter est qu'il requiert une certaine connaissance du bytecode Java. Si pour réaliser des choses simples, cela est très aisé, une fois qu'on veut aller un peu plus loin, il faut bien comprendre le fonctionnement du bytecode.
ASM possède un bon support dans Eclipse avec le plugin Bytecode Outline[15] qui permet notamment d'afficher le code version ASM à partir du bytecode d'une classe.

Il est intéressant de lire [Kuleshov, 2007] qui offre un bon aperçu, bien que basique, de ce qu'il est possible de faire avec ASM ainsi que des comparaisons au niveau des performances avec ses principaux rivaux. Cet article date un peu concernant ce dernier point mais permet de donner une idée.

Javassist permet de s'abstraire de cette manipulation du bytecode car il offre un ensemble de méthodes qui font le travail pour nous. Il est bien entendu toujours possible de manipuler le bytecode directement mais de manière moins poussée qu'avec ASM. Une particularité de Javassist est qu'il intègre un compilateur Java basique : cela permet à Javassist de compiler plus vite car il n'est pas nécessaire d'avoir toutes les possibilités du vrai compilateur Java. En effet, Javassist ne permet pas de tout modifier directement, aussi non il aurait été plus facile de manipuler le bytecode directement, mais comme introduit précédemment, il offre un mini-langage intermédiaire pour faire cela.

Javassist est le framework qui a été choisi principalement pour son abstraction du bytecode et sa facilité d'utilisation. Il semble être un bon compromis dans notre cas.

Un élément commun à ces outils d'instrumentation est l'utilisation du design pattern *Visitor* ; ce dernier construit une opération à réaliser sur les éléments d'un ensemble d'objets. De nouvelles opérations peuvent ainsi être ajoutées sans modifier les classes de ces objets. La structure des opérations possibles sur ces éléments est découplée et l'ajout de nouvelles opérations s'en trouve grandement facilité.

Comme il l'a déjà été expliqué brièvement dans la « *Partie 1* », chapitre « *Approches* », point « *Réécriture de l'AST »*, le code source d'un programme peut être représenté sous forme d'arbre dont les nœuds peuvent être des classes, des méthodes, des propriétés, des instructions, etc. Le pattern *Visitor* s'applique très bien dans ce cas et offre l'extensibilité nécessaire. Il suffit de visualiser le fait qu'en parcourant l'arbre, à chaque nœud, en fonction de ce qu'il représente, une méthode spécifique du *Visitor* peut être appelée.

Il va être tenté ici d'illustrer le pattern *Visitor* à l'aide d'un diagramme de séquence afin de se rendre compte de la mécanique du pattern. Le cas est très simple : il y a Visitor qui joue le rôle

[12] http://asm.ow2.org
[13] http://www.javassist.org
[14] http://commons.apache.org/proper/commons-bcel
[15] http://asm.ow2.org/eclipse/index.html

du … Visitor, ainsi que deux éléments *ClassElement* et *MethodElement* qui sont les constituant de la structure en forme d'arbre qu'il faut traverser.

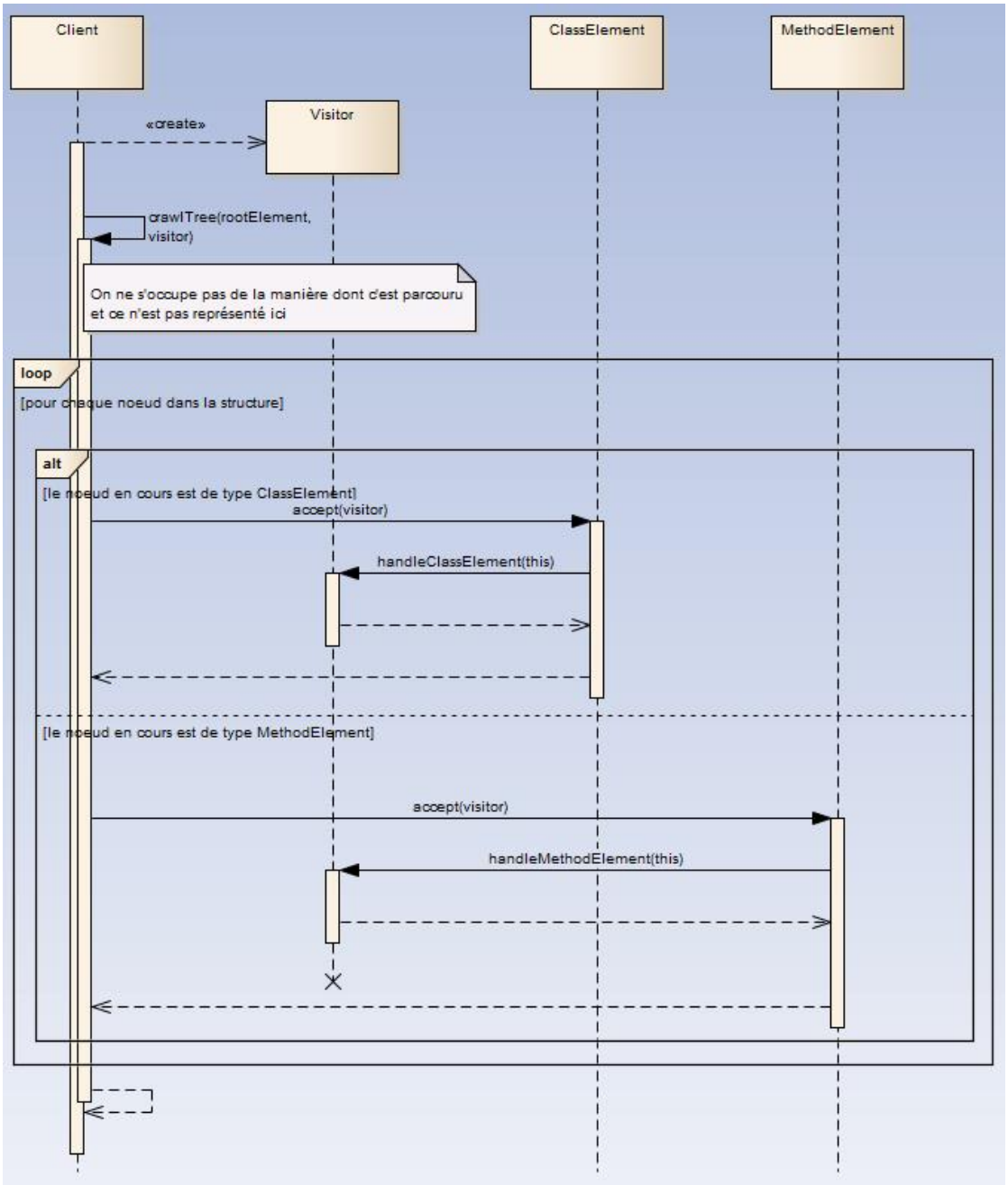


FIGURE 7 – DIAGRAMME DE SÉQUENCE ILLUSTRANT L'UTILISATION DU DESIGN PATTERN VISITOR PAR LES OUTILS DE MANIPULATION DE BYTECODE JAVA.

4.5 Programmation Orientée Aspect (POA)

La programmation orientée aspect (ou aspect-oriented programming, AOP, en anglais) est une méthodologie qui complète la Programmation orientée objet (POO). En POO, les applications sont organisées en classes et interfaces. Ces concepts sont bien adaptés pour le développement par séparation des préoccupations métiers c'est-à-dire qu'une classe s'occupera d'un sujet métier.

Mais il peut y avoir des préoccupations transversales c'est-à-dire des exigences qui concernent l'ensemble (ou une partie) des classes d'une application. Il est très fréquent dans les applications d'entreprise avec notamment la journalisation, l'authentification ou la validation. Et c'est ce qu'apporte la Programmation orientée aspect : elle permet la modularisation de ces préoccupations transversales.

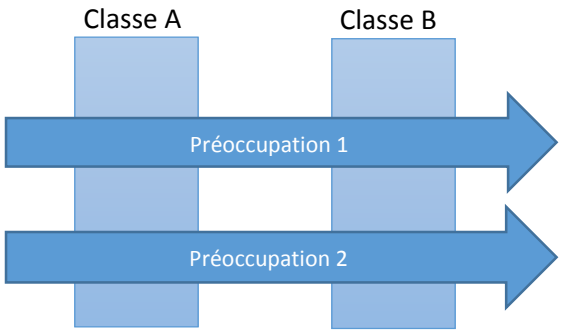


FIGURE 8 – REPRÉSENTATION CLASSIQUE DES PRÉOCCUPATIONS DITES TRANSVERSALES.

Plusieurs concepts à connaître :

- un aspect : correspond à la formalisation d'une préoccupation transversale ;
- le greffon (advice) : est le code définissant le comportement d'un aspect ;
- le point de coupe (pointcut) : désigne où un aspect doit être appliqué, intégré dans l'application. Un point peut être une méthode (ou un appel), une exception, un constructeur ou un attribut ;
- le point de jonction (join point) : spécifie quand il faut insérer le greffon par rapport à un point de coupe défini. Typiquement : avant, après, autour de ou à la place de la cible.

Pour mettre cela en œuvre, une technique appelée le « tissage » (weaving) est utilisée. Il s'agit du processus d'application des aspects aux objets cibles. Le tissage prend donc en entrée les aspects et une application (un ensemble de classes) et fournit une application dont le comportement (classes) est étendu par les aspects (voir partie de droite sur le schéma suivant). Cela entremêle (tisse) les classes et les aspects aux différents points de coup décrits.

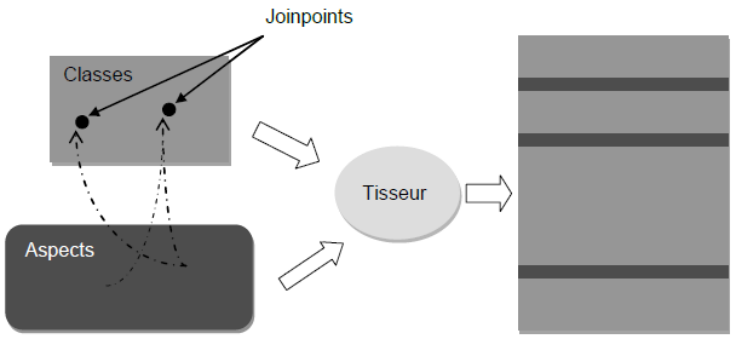


FIGURE 9 – ILLUSTRATION DU PHÉNOMÈNE DE TISSAGE DANS LA PROGRAMMATION ORIENTÉE ASPECT (POA).

Il existe trois manières d'effectuer le tissage :

- à la compilation : le tissage est ici réalisé dans une étape de pré compilation ;
- au chargement : le tissage se fait quand les classes sont chargées. Les aspects sont ajoutés par instrumentation dynamique du code ;
- à l'exécution : les aspects peuvent être appliqués pendant que le programme est en exécution. À cette fin, il faut que cela soit supporté.

Bien que le mécanisme soit pratique pour insérer des éléments à des endroits très spécifiques, il n'est pas souvent le plus adéquat. Comme il l'est très justement mentionné dans [Chawla et al., 2004], la POA ne permet pas d'avoir accès à des éléments bas-niveau (comme le bloc de base) et surtout, cette technique peut provoquer une diminution importante des performances.

Pour le besoin principal de l'analyseur, à savoir injecter du code à certains endroits et avoir accès à des informations contextuelles précises ainsi que configurer les éléments à analyser, cette option n'a pas été retenue dans le cadre de ce travail.

4.6 Extraction et enregistrement

Maintenant que nous avons tous les éléments techniques nécessaires à la bonne compréhension de la solution mise en place, son fonctionnement général va être décrit. Dans la figure suivante, on peut observer les composants principaux. Le module développé et appelé AnaDONE se compose principalement de deux composants, à savoir l'agent Java (JAgent) et le traceur (EventTracer).

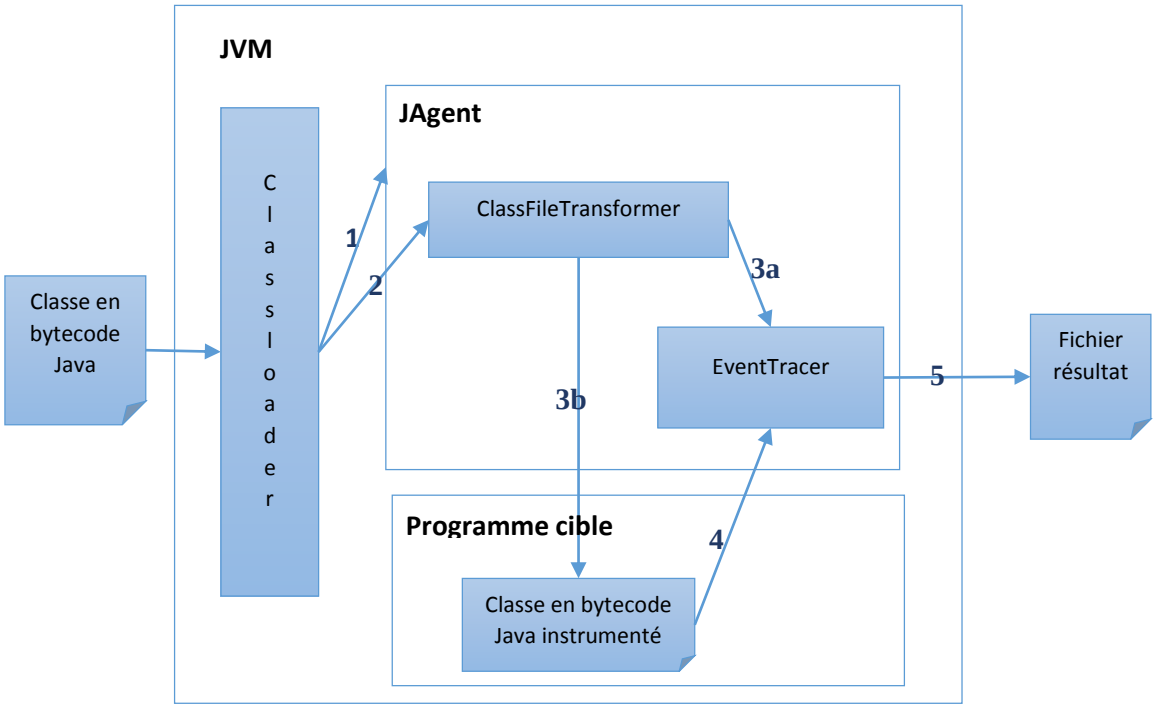


FIGURE 10 – SCHÉMA DES COMPOSANTS TECHNIQUES QUI CONSTITUENT ANADONE.

Au démarrage, le *classloader* de la JVM va donc charger toutes les classes dont il a besoin: les classes systèmes qui sont nécessaires à son fonctionnement, ensuite les classes du programme à exécuter et de ses dépendances. Les classes systèmes sont chargées avant que l'agent Java lui-même ne le soit. Après vient le tour des classes de l'agent Java et celles dont il a besoin pour fonctionner puis celles du le programme cible.

Une fois les classes systèmes chargées, le *classloader* va appeler notre agent (1) pour qu'il configure ce dont il a besoin : en gros, l'agent va enregistrer les *ClassFileTransformer* dont il a besoin pour capturer les informations nécessaires et bien entendu, opérer aux diverses instrumentations.

Cela fait, pour chaque classe de programme cible et de ses dépendances, le *classloader* va appeler les *ClassFileTransformer* enregistrés (2). Le résultat de cet appel sera le bytecode instrumenté de la classe ou rien dans le cas où il n'y a pas eu de modification. Il y a deux choses importantes qui sont effectuées ici : la récolte d'informations dites statiques (3a) et l'instrumentation (3b).

Lors de la récolte des informations statiques, il ne faut pas oublier qu'on est à l'exécution et que ce n'est pas le source code qui est analysé, mais bien le bytecode généré par le compilateur Java à partir du source code. C'est un peu trompeur et ambigü, mais on pourrait bien parler ici d'analyse statique-dynamique. Les informations récoltées ici sont très importantes pour la suite des opérations car elles vont servir de référentiel pour les consolidations, analyses et calculs qui seront effectués plus tard. Dans notre cas, ces informations sont les différentes classes qui doivent être analysées, leur hiérarchie d'héritage (pour chaque classe, la classe dont elle hérite éventuellement, ainsi que des interfaces qu'elles implémentent éventuellement) et tous les appels de méthodes statiques qui sont rencontrés dans les classes à analyser. Une partie de ces données sont envoyées à l'*EventTracer* (3b) pour qu'il les garde pour ses analyses ou qu'il les écrive dans le fichier de résultat.

L'autre tâche qui est réalisée, est l'instrumentation du bytecode (3a). Dans notre cas, la seule instrumentation nécessaire pour le moment, consiste à ajouter un morceau de code au tout début de chaque méthode faisant partie des classes à analyser. Ce morceau de code consiste en l'appel à notre *EventTracer* (4) afin qu'il enregistre les informations importantes sur cet appel, ces informations étant : le type effectif de la classe contenant la méthode appelée (équivalent en Java à *this.getClass()* ou juste le nom complet de la classe dans le cas d'une méthode statique), la ligne de la méthode appelée, les types effectifs de tous les paramètres reçus (ou *null* si le paramètre est *null*), ainsi que le contexte d'appel qui a mené à la méthode appelée (plus de détails à ce sujet seront donnés dans *4.9 Une nouvelle métrique : la distance d'appel*).

La dernière pièce, et non des moindres, qui n'a pas encore été décrite, est l'*EventTracer*. L'*EventTracer* tout seul ne fait rien. Il s'agit en fait d'une API à utiliser (comme on l'a vu ci-avant) pour que les informations nécessaires à l'analyse soit prises en compte et enregistrées si nécessaire. Comme on peut le voir sur le schéma, c'est un point central, et encore plus à l'exécution du programme cible : tous les événements capturés vont l'appeler. Un soin très particulier a donc été apporté à son élaboration et au fait de le rendre le plus possible transparent.

L'*EventTracer* en soit est une interface exposant des méthodes utilisables. L'implémentation est tout à fait libre, de sorte qu'il soit possible d'adapter le traceur réel en fonction des besoins. Dans le cas qui nous occupe, deux versions sont proposées, pour les deux le résultat étant le même, à savoir un fichier contenant le résultat de l'analyse et les différents enregistrements des captures d'événements, cela au format XML. La première implémentation était plutôt naïve et écrivait dans le fichier résultat de manière synchrone, ce qui provoquait certaines contentions. De plus, toutes les consolidations n'étaient pas possibles. Une deuxième implémentation a donc été réalisée, en tenant compte de l'expérience gagnée avec la première : écriture dans le fichier asynchrone, mécanismes de détection de cycle (voir *4.8 Cycles*), protection contre les appels concurrents, mémorisation dynamique des informations nécessaires aux analyses et possibilité de mettre l'enregistrement des données sur pause (voir *4.10 Décider de l'enregistrement*).

La capture d'écran ci-après est celle d'un fichier et montre le résultat de l'analyse d'un petit programme créé, avec quelques classes, des appels de méthodes, des cas intéressants avec du polymorphisme :

```
<root>
<class isinterface="false" isabstract="false">
    <name>demo1.ConcretInterfaceClass</name>
    <parent>demo1.AbstractClass</parent>                    1
    <interface>demo1.InterfaceDemo</interface>
</class>
...
<callsite>
    <called line="17">demo1.ConcretInterfaceClass.methodAbstract(int)</called>
    <caller line="30">demo1.Demo1.main(java.lang.String[])</caller>                    2
</callsite>
...
<method_in>
    <called line="18">demo1.AbstractClass.methodNonAbstract(java.lang.String)</called>
    <caller distance="1" line="22">demo1.ConcreteClass2.methodNonAbstract</caller>
    <stacktrace>
        <element line="22" index="1">demo1.ConcreteClass2.methodNonAbstract</element>
        <element line="41" index="2">demo1.Demo1.main</element>
    </stacktrace>
    <parameters>
    <parameter index="0">java.lang.String</parameter>                    3
    </parameters>
</method_in>
</root>
```

FIGURE 11 –FICHIER RÉSULTAT DE L'ANALYSE AVEC ANADONE.

Il n'y a aucun lien entre les trois éléments, ils ont été pris au hasard et sont là pour illustrer la structure du fichier résultat. Il y a trois types de bloc importants :

- en 1, le bloc *class* qui contient les informations sur les classes à prendre en compte pour l'analyse. Sont notamment retenus : son nom, son package, le fait que ça soit une interface ou non, que la classe soit abstraite ou non, ainsi que sa classe parent si elle en a une, et les interfaces qu'elles implémentent si il y en a ;
- en 2, le bloc *callsite* qui correspond à tous les appels de méthodes rencontrés (cf. *statique-dynamique* au début de ce point). Ici sont enregistrées les informations sur la méthode appelante (*caller*) et la méthode appelée (*called*). Ces informations sont les lignes de l'appel de méthode et de la méthode appelée, ainsi que les signatures complètes ;
- en 3, le bloc *method_in* qui est le résultat de la capture à l'exécution de l'événement correspondant à une méthode appelée. Il y a beaucoup d'informations ici, notamment comme pour le *callsite* les informations sur la méthode appelante et la méthode appelée (*called* et *caller*), mais aussi les types effectifs des paramètres reçus par la méthode appelée (*parameters*) avec leur ordre (index), ainsi que le contexte d'appel (*stacktrace*) avec pour chaque élément qui le compose, le nom de la méthode, son ordre dans le contexte et la ligne de l'appel effectué.

4.7 Configuration de l'agent

Il a été mis en place un mécanisme permettant de décrire, de manière aisée, les packages et classes à prendre en compte pour la phase de récolte de données et d'instrumentation, donc les classes à analyser.

L'approche ici est que par défaut rien n'est pris en compte ; aucun des packages, classes ou méthodes qui sont chargés par le classloader ne sont instrumentés par notre module. Tout ceci dépend de ce qu'il a été décidé d'instrumenter. Dans la figure suivante, on peut voir les deux

éléments *package.include* et *class.include* avec des valeurs respectives. Dans ce cas-là, toutes les classes du package *org.gjt.sp.jedit.help* et la classe spécifique *org.gjt.sp.jedit.View* seront prises en compte :

```
package.include=org.gjt.sp.jedit.help
class.include=org.gjt.sp.jedit.View
record.default=true
```

FIGURE 12 – DESCRIPTION DU FICHIER DE CONFIGURATION D'ANADONE.

Il faut savoir qu'au début c'était l'inverse qui avait été implémenté : par défaut tout était instrumenté, et il fallait spécifier les éléments à exclure. Nous nous sommes très vite retrouvés face à une masse énorme de données capturées et enregistrées, ce qui était tout bonnement ingérable. De plus, cela ne répondait pas à une des caractéristiques souhaitées pour la solution, à savoir qu'elle puisse répondre à un problème localisé.

La logique utilisée est très simple et est la même pour les deux cas : les différents noms sont séparés par des « *;* », ensuite pour chaque élément traversé correspondant au cas ciblé (package ou classe), une vérification est faite afin de s'assurer que le nom de cet élément commence par l'une des valeurs renseignées. Si c'est le cas, cet élément n'est pas instrumenté.

Pour des raisons de performances, cette vérification ne se fait pas en utilisant une boucle qui teste tous les noms un par un mais, par le biais d'expressions régulières qui sont construites au démarrage. Il faut se rendre compte que plusieurs milliers de classes sont chargées, classes qui résident dans des centaines de paquets et contenant plusieurs méthodes pour chacune d'entre elles. Cette fonctionnalité qui utiliserait une comparaison implémentée de manière grossière avec une boucle aurait de gros impacts négatifs sur les performances. Les deux expressions régulières créées pour l'exemple de la figure précédente sont dans l'ordre : *^((org/gjt/sp/jedit/help))(.*)* et *^((org.gjt.sp.jedit.View))(.*)* .

Tout cela se spécifie dans un fichier *.properties dont le chemin est passé à l'agent Java en utilisant le mécanisme décrit dans le point *4.3 Agent Java (JAgent)*, et permettant de passer un certain nombre de paramètres de type chaîne de caractères à l'agent. Exemple :

```
java -javaagent:myagent.jar=./jagent.properties ...
```

Il y a encore un élément de ce fichier de configuration qu'il reste à voir, l'élément *record.default* dans la figure précédente, mais pour lequel il faut introduire un nouveau concept, concept qui fait l'objet du point *4.10 Décider de l'enregistrement*.

Un autre point de personnalisation est la possibilité de dire où le résultat de l'analyse doit être stocké. Cela se fait au moyen d'un paramètre système que les applications Java peuvent retrouver et qui peut être spécifié en ligne de commande avec le reste des paramètres de la manière suivante :

```
java -javaagent:myagent.jar=./jagent.properties -Danalysis.out.path="C:\logs\analysis_results.log" ...
```

Si ce paramètre n'est pas spécifié, le fichier est créé dans le répertoire de travail courant de l'utilisateur.

Nous le voyons, cette approche permet d'avoir un contrôle assez fin sur ce qu'il faut instrumenter, sans être trop fastidieuse ou restrictive.

4.8 Cycles

Une problématique intéressante et récurrente avec ce genre d'outils est celle de « cycle ». En effet, notre analyseur utilise lui-même des classes et méthodes de la JVM qui exécute l'application cible et qui donc contient les éléments instrumentés. Il faut donc faire attention à ne pas analyser l'analyseur lui-même.

Un petit exemple simple permet d'illustrer ce propos : imaginons que la méthode *System.out.println(String msg)* , permettant d'afficher du texte sur le flux de sortie standard (la console en général), ait été instrumentée de telle manière qu'à chaque fois qu'un flux d'exécution y entre, la première chose qui soit faite est un appel à notre analyseur. Ce dernier va se charger d'écrire, sur le flux de sortie standard, le paramètre passé, à savoir le contenu de la variable *msg*.

Afin d'écrire sur le flux de sortie standard, notre analyseur va donc lui aussi utiliser la méthode *System.out.println(String msg)*. En entrant à nouveau dans cette méthode, cette fois-ci appelée par l'analyseur, la première chose qui va être faite va être d'appeler l'analyseur afin qu'il … OutOfMemoryError !

Pour résoudre ce problème, un système simple a été mis en place et consiste en l'utilisation de variables de type booléen, *locales* aux différents threads, permettant de savoir si dans la chaîne d'appel, le flux est passé ou non par une des méthodes de notre analyseur.

Donc dès qu'un thread entre dans une méthode de notre analyseur, un test est fait sur la valeur de la variable booléenne spécifique au thread, variable que nous appellerons « *reentrant* ». Cette variable booléenne peut avoir deux valeurs : *true* signifiant que le thread possède dans sa file d'appels de méthodes une des méthodes de l'analyseur et que donc, nous sommes en présence du fameux cas de cycle ; *false* dans le cas contraire. Si sa valeur de *reentrant* est égale à *true*, alors on sort directement de la méthode. Si elle est égale à *false*, sa valeur est mise à *true* et l'opération normale de la méthode est exécutée. Une fois cette opération terminée, la valeur de *reentrant* est mise à false. Ce mécanisme se base sur le principe dit des « *régions critiques* ».

Cela a été fait de cette manière pour éviter à tout prix d'avoir recours à des mécanismes de blocage plus lourds qui auraient eu un impact vraiment négatif sur les performances de la solution proposée.

Pour illustrer tout cela, imaginons un cas simple mais parlant dans lequel nous avons deux classes : *ClasseB* offre une méthode statique, *void displayThis(String msg)*, permettant d'afficher du texte dans la console. *ClasseA* utilise cette méthode. Le code de ces deux classes a été instrumenté de manière telle qu'à chaque entrée dans une méthode, cela est notifié à l'analyseur *Analyseur* au travers de la méthode *void notifyMethodIn(String nomMethode)*. Cette méthode va aussi utiliser la méthode *displayThis* afin de faire afficher le nom de la méthode passé en paramètre.

Voici un diagramme de séquence qui se propose de visualiser notre petit exemple de manière plus formelle :

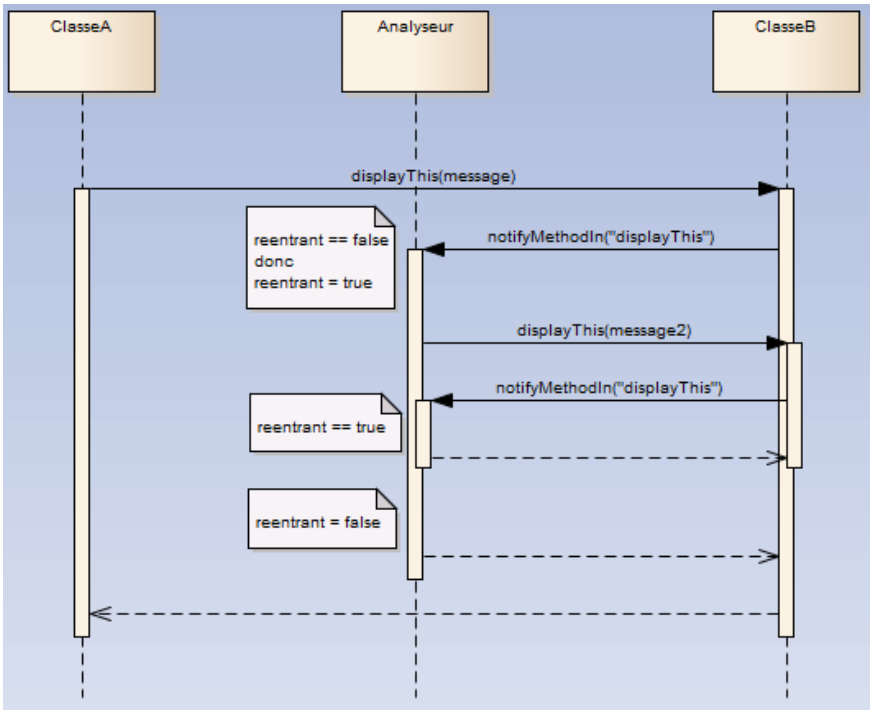


FIGURE 13 – DIAGRAMME DE SÉQUENCE DÉCRIVANT LA SOLUTION MISE EN PLACE POUR ÉVITER LES PROBLÈMES DE CYCLES.

Nous pouvons facilement imaginer la boucle infinie qu'il y aurait eu sans la mise en place du mécanisme décrit ci-avant.

4.9 La distance d'appel

Dans la littérature, nous avons identifié une métrique qui permet d'apporter une information intéressante : la distance d'appel. Cette métrique est attachée à une méthode et indique quelle est la position de son appeleur effectif dans l'arbre des appels qui ont mené jusqu'à elle. Lorsqu'une méthode *A* appelle une autre méthode *B*, A est l'appeleur et B l'appelé. L'arbre d'appel est alors : B, A (on remonte de l'appelé à l'appeleur).

L'intérêt de cette métrique est qu'elle peut être utile dans la détection de design patterns (un Proxy dynamique, un Decorator ([Wikipedia, 2014e]), etc) ou de mécanisme de modification du code source (Instrumentation dynamique, POA, etc). Bref, cela met en évidence le fait qu'il existe des déroutements, appliqués à l'exécution, entre l'appeleur et l'appelé. Cette métrique pourrait être très utile pour la compréhension d'un programme car elle met clairement en évidence un comportement purement dynamique et qui est difficilement appréhendable depuis le code source.

Pour obtenir cet arbre des appels en Java, il y a plusieurs manières de faire, la plus simple étant d'instancier une nouvelle exception et de récupérer tout l'arbre des appels ayant mené jusqu'où l'exception est créée. Au niveau du code, cela se traduit par :

```
StackTraceElement[] elements = new Exception().getStackTrace();
```

Dans cet arbre, l'élément à l'index 0 représente la méthode où l'exception a été instanciée. L'élément à l'index 1 est l'appeleur, celui à l'index 2 est l'appeleur de l'appeleur, etc.

L'objet *StackTraceElement* offre les propriétés suivantes :

```
private String declaringClass;
private String methodName;
private String fileName;
private int    lineNumber;
```

Il nous est dès lors possible de vérifier que l'appeleur est bien celui auquel on s'attendait. Si ce n'est pas le cas, c'est bien qu'il y a eu un changement et qu'un mécanisme externe a été mis en place.

En reprenant l'exemple du premier paragraphe du présent point, avec A l'appeleur et B l'appelé : imaginons qu'on ait mis en place de la POA qui ferait en sorte qu'avant chaque appel à B, on fasse d'abord un appel à C, pour ensuite appeler B. L'arbre des appels donnerait ceci : B, C, A. L'appeleur originel (A) au lieu de se trouver à l'index 1, se trouve maintenant à l'index 2.

Comment calculer cela ? Lors de la phase d'analyse et d'instrumentation qui se déroule pendant le chargement des classes, il faut garder, et ce pour chaque appel de méthode détecté, la trace de l'appeleur ainsi que de l'appelé. Lorsqu'une méthode instrumentée est appelée à l'exécution, l'appeleur effectif (celui à l'index 1) est récupéré avec la technique décrite plus haut et une vérification est faite pour s'assurer que pour la méthode appelée, l'appeleur est bien un de ceux prévus (parmi les traces gardées pendant l'analyse). Si ce n'est pas le cas, il faut refaire ce test mais cette fois-ci pour tous les éléments de l'arbre d'appels ayant un index strictement plus grand que un, du plus petit au plus grand, jusqu'à arriver à la fin. Si en chemin on rencontre l'appeleur prévu, on peut s'arrêter là, la distance d'appel sera égale à l'index de l'élément courant.

Une précaution importante à prendre ici est causée par le polymorphisme. Imaginons que nous ayons une classe A qui déclare une méthode *methodA*. Maintenant, rajoutons une classe B qui étend A et qui en plus redéfinit la méthode *methodA*. Nous avons dans une autre classe, une méthode qui a instancié B mais référence cette instance en utilisant une variable typée avec A. Un appel vers *methodA* est fait sur cet objet. Lors de l'analyse au tout début, l'analyseur ne peut pas aller au-delà de cela : pour lui l'appel ira vers A. Mais en fait, à l'exécution, l'appel peut très bien aller vers B.

Poussons encore l'exemple plus loin : imaginons maintenant que A ne déclare pas *methodA* et qu'il hérite d'une classe C qui elle déclare *methodA*. Pour l'analyse, à nouveau, l'appel de méthode ne sera jamais vu comme pouvant aller vers C, ce qui sera en fait le cas à l'exécution. C'est ici qu'est à prendre la précaution : quand un appel est détecté et que la trace de cet appel est gardée, il faut aussi créer une trace pour tous les héritiers de A, mais aussi pour toutes les classes dont A héritent, bref remonter l'arbre d'héritage, car potentiellement rien n'empêche à l'exécution qu'ils soient les objets sur lesquels la méthode est effectivement appelée. Cela implique donc aussi lors du chargement des classes de construire ces arbres d'héritage.

Il y a lieu de noter que cette distance d'appel peut varier pour chaque appel, même lors de plusieurs appels à la même méthode : cela peut être évidemment dû au polymorphisme mais aussi au fait qu'on ne sache pas exactement ce que les mécanismes intermédiaires font, ce qui pourrait résulter dans des routes différentes pour arriver enfin à la méthode désirée.

Pour résumer : quand la distance d'appel égale à 1, cela signifie que l'appeleur est bien celui qui était prévu ou un de ceux faisant partie de l'arbre d'héritage de l'appeleur. Dans le cas où la distance est plus grande ou égale à 2, c'est un indicateur qui signale qu'il y a eu un déroutement par rapport au plan initial.

4.10 Décider de l'enregistrement

Lorsque l'analyseur est en route, il se peut qu'on ne veuille pas tout le temps qu'il enregistre les événements qu'il capture. Néanmoins, pour fonctionner, l'analyseur a toujours besoin d'un minimum d'informations, notamment celles concernant les classes et leurs hiérarchies ainsi que les appels de méthodes détectés lors du chargement des classes par la JVM. Il se sert de ces informations pour faire ses consolidations et ses comparaisons. Mis à part cela, on pourrait tout à fait souhaiter mettre l'enregistrement en pause, puis la rétablir à un moment bien précis, ceci afin de ne pas polluer le fichier résultat avec des informations dont on n'a pas besoin. Il faut bien différencier ici la capture et l'enregistrement : la capture des événements dynamiques se fera quand même, mais l'enregistrement sera fonction de ce qu'on a décidé.

Le mécanisme mis en place permet à tout moment à l'utilisateur de décider d'activer la capture ou de la mettre sur pause. En pause, les informations dont l'analyseur a besoin pour son fonctionnement sont bien enregistrées mais les autres ne le sont plus. Les autres sont principalement les informations dynamiques comme le fait qu'une méthode ait été appelée, un constructeur ou autres, donc des événements dynamiques, ceci étant dépendant de l'instrumentation opérée. De plus, une option de configuration du fichier décrit dans *4.7 Configuration de l'agent* vous permet de spécifier si, par défaut, il faut enregistrer ou non. Il est envisageable que dès le départ il ne soit pas souhaitable d'enregistrer les informations des événements dynamiques, afin d'attendre un état bien précis et seulement commencer l'enregistrement.

Ce mécanisme est mis en place grâce à JMX[16] (Java Management eXtensions), une API pour la gestion et la surveillance d'applications, d'objets systèmes, de services et de la JVM elle-même. Ces ressources sont représentées par des objets qui suivent une convention et qui sont appelés MBean (Managed Bean). Ces MBeans sont alors gérables depuis un client JMX : un client est fourni par défaut avec le JDK est appelé JConsole[17], mais il y en a d'autres et on peut fabriquer le sien car il s'agit bien avant tout d'une API. Les détails ne seront pas abordés ici mais en pratique il s'agit d'un simple objet Java (ou POJO) qui doit implémenter une interface qui contient les opérations qu'il sera possible d'invoquer sur ce MBean. Ensuite, il faut demander au MBeanServer de prendre en charge notre MBean et de bien vouloir exposer les opérations décrites dans notre interface : à cet effet, il faut lui passer une instance de l'objet implémentant notre interface, ainsi qu'un nom et un chemin qui identifieront de manière unique le MBean.

[16] http://openjdk.java.net/groups/jmx/
[17] http://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html

Dans le cas qui nous occupe, une seule opération est exposée : *void doRecord(boolean record)*. Le paramètre à donner permet d'indiquer si on désire que les événements soient enregistrés.

La capture suivante montre dans JConsole le MBean exposé par AnaDONE, dont le chemin est *be.deeple.dynamicanalysis* et le nom *DynamicAnalysisJAgentMBean*. Nous pouvons y voir les opérations exposées dont celle qui nous intéresse, à savoir *doRecord*. Quand une opération est sélectionnée, à droite, on peut voir ses détails et notamment invoquer l'opération dans le cadre intitulé *Operation Invocation*, dans lequel il faut spécifier la valeur du paramètre à envoyer et ensuite cliquer sur le bouton avec le nom de l'opération.

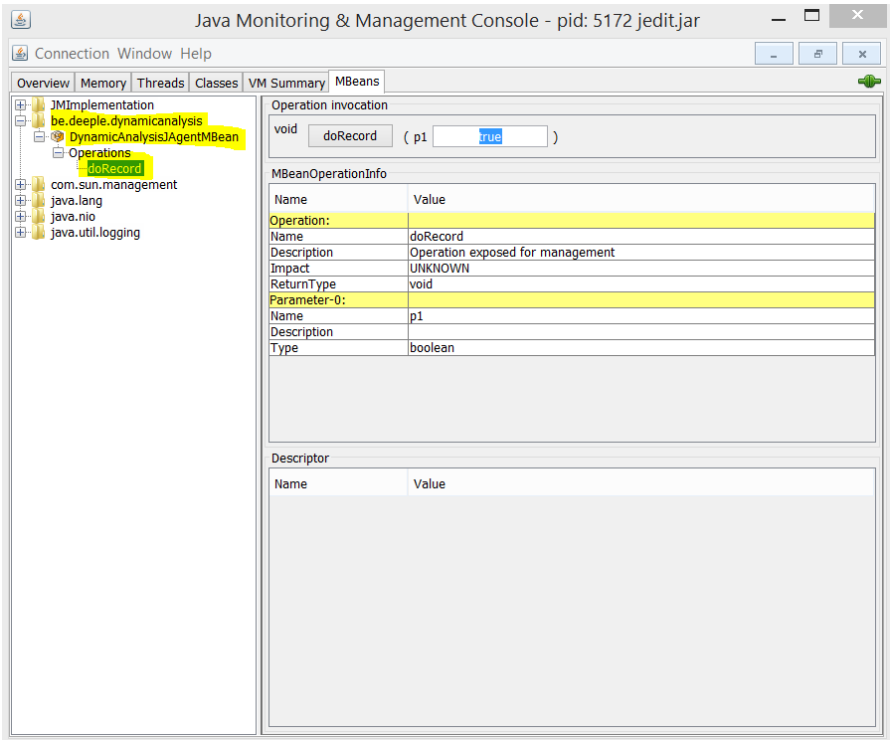


FIGURE 14 –CAPTURE D'ÉCRAN DE JCONSOLE PERMETTANT DE CONTROLER ANADONE.

Du côté d'AnaDONE, quand l'opération est invoquée, le paramètre va être sauvé dans une variable. Le mécanisme en place est simple : dans toutes les méthodes d'enregistrement des informations des événements capturés, avant d'enregistrer les informations, un test est fait sur cette variable, en fonction de quoi les informations sont enregistrées ou non.

4.11 Conclusion

Comme dit dans l'introduction, AnaDONE a nécessité un travail conséquent malgré le fait qu'il ne soit pas très grand, cela étant dû aux précautions à prendre car c'est extrêmement sensible. Il est important de bien comprendre les concepts en jeu.

Le lecteur attentif aura peut-être remarqué que les métriques et leurs calculs sont absents de la solution proposée. Ce choix s'est imposé car les métriques connues ne sont que peu pertinentes pour une aide à la compréhension localisée ; on pourrait bien entendu compter le nombre d'appels de méthode, le nombre de lignes de code ou autre, mais ce n'est pas indispensable. Et puis, en se basant sur les données récoltées, avec une analyse après coup, il est tout à fait

possible d'en extraire certaines. Malgré tout, pendant les diverses recherches, une nouvelle métrique a pu être identifiée, et serait tout à fait utile, il s'agit de la distance d'appel. Il faudrait notamment y consacrer plus de temps et trouver une manière plus efficace de l'obtenir.

Les buts principaux étant que la solution soit performante et extensible, tout ce qui est superflu, ou semble l'être, n'a pas été mis en place. Il s'agit en fait d'une base saine qu'il faut enrichir et faire évoluer. Tous les éléments sont là pour le permettre. Au niveau de la performance, même sur de gros programmes, les impacts constatés ne sont pas énormes et n'empêchent en rien le bon fonctionnement du programme cible.

Il est intéressant de noter qu'AnaDONE se place à un endroit qui permet de décider de la granularité de l'analyse désirée et des comportements à observer. Cela est rendu possible par l'utilisation d'un agent Java qui nous offre toutes les possibilités dont nous avons besoin pour cet exercice. Nous ne pouvons qu'insister sur le fait que beaucoup d'autres choses sont encore possibles et qu'il serait dommage de s'en priver. Les possibilités offertes par la JVM et ses API permettent vraiment d'aller dans le détail et de faire du sur-mesure. Il faut par contre s'atteler à trouver l'équilibre et maîtriser ce qui est fait.

5. Visualisation

Ce chapitre présente la visualisation qui a été créée, mise en place et intégrée dans MetaDONE. MetaDONE étant un outil metaCASE, une rapide introduction concernant les concepts d'outils CASE et metaCASE va être faite. Elle montrera principalement les raisons d'être de ces outils ainsi que leurs différences. Ensuite, c'est le fonctionnement de MetaDONE qui sera expliqué afin de montrer ses spécificités et en quoi il s'agit d'un candidat adéquat pour l'exercice de visualisation.

La visualisation proposée se base sur le diagramme de classe UML qui a été augmenté afin de prendre en compte certaines informations concernant les aspects dynamiques des programmes. On verra notamment qu'il n'est pas aisé d'avoir une visualisation claire et concise sans devoir faire un choix quant aux informations à afficher. Cette visualisation a été intégrée à MetaDONE par l'intermédiaire d'un plugin développé pour l'occasion et qui s'appelle VizuDONE. Ce dernier comprend la définition du méta-modèle, la logique de création du modèle concret ainsi que la logique d'affichage.

5.1 CASE et metaCASE

Les outils CASE (Computer-aided software engineering) ont été créés afin de solutionner les problèmes courants qui concernent la qualité, les coûts et la productivité, problèmes qui minent le développement logiciel. Leurs buts principaux est d'automatiser l'utilisation de modèle conceptuels, de règles, de notations basées sur les diagrammes. Le meilleur exemple concerne les outils CASE pour la modélisation orientée-objet qui sont généralement basés sur la méthodologie UML.

Bien qu'au début ils avaient pour principale cible la phase d'implémentation, et donc de supporter les développeurs dans leurs tâches, ils ont fini par être utilisés pour toutes les activités d'un projet, en allant de la capture des exigences, à la création de prototypes et bien d'autres.

Les outils CASE traditionnels permettent d'utiliser seulement un nombre limité de méthodes et qui plus est, fixé ([Isazadeh, 1997]) ; le grand défaut de ces outils est donc le manque de personnalisation. Il faut avoir en tête que chaque projet est différent, chaque domaine est différent, dès lors utiliser une méthode fixe ne permet pas de s'adapter aux différentes situations. Les méthodes sont pour la plupart du temps personnalisées et surtout, évoluent avec le temps. Il est souvent plus facile d'utiliser un stylo et une feuille de papier, afin d'être libérer des contraintes d'une méthode qui ne convient pas totalement, cela impliquant d'abandonner l'assistance que peut apporter un outil dédié à cela.

Un outil metaCASE permet de concevoir et de générer rapidement et facilement des outils CASE à partir d'une définition abstraite. Les outils metaCASE offrent la possibilité de capturer les spécifications d'une méthode, pour ensuite générer, de manière automatique, un outil CASE depuis ces spécifications ([Ebert et al., 1997]). Le mot automatique dans la phrase précédente est capital car il permet de couvrir les besoins d'évolution et de changements ; il suffit alors de modifier ou d'ajouter des spécifications et de régénérer l'outil CASE. De plus, il est possible

d'adapter entièrement la méthode aux besoins d'un projet ou d'une organisation, voir même d'en créer une nouvelle. Le résultat obtenu est donc une méthode et un outil qui l'intègre, tout cela adapté aux besoins.

Au niveau du fonctionnement, les outils CASE utilisent généralement une architecture à deux niveaux, tandis que les outils metaCASE utilisent trois niveaux ([MetaCase, 2004]) :

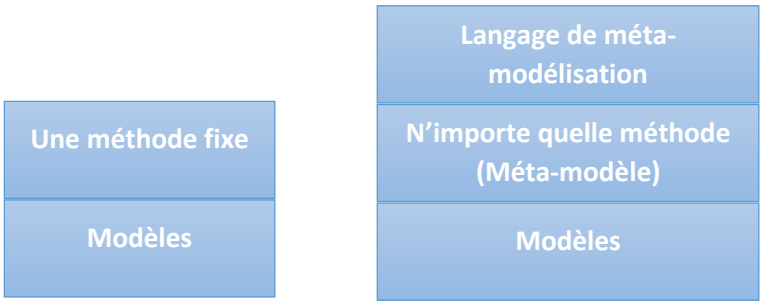


FIGURE 15 – ARCHITECTURE OUTILS CASE ET METACASE

Dans le cas des outils CASE, la méthode est écrite en dur dans l'outil, ce qui limite les modèles qu'il est possible de créer. Les difficultés de personnalisation pour modifier la méthode sont dès lors compréhensibles car il faut modifier l'outil.
Les outils metaCASE ajoutent un troisième niveau, ce qui permet d'enlever la limitation qu'ont les outils CASE : il est alors possible d'avoir des méthodes flexibles. Cela est rendu possible par l'utilisation de méta-modèles qui définissent les concepts, les notations et les règles d'une méthode, ils décrivent une méthode. Ces méta-modèles, au lieu d'être écrits en dur dans l'outil, sont en fait des éléments vivants, tout comme les modèles, et sont traités comme des données. Cela nous amène au troisième niveau, le langage de méta-modélisation qui permet de décrire et modifier le méta-modèle. C'est ce langage de méta-modélisation qui est alors écrit en dur dans l'application.

Les outils metaCASE pourraient être de très bons candidats afin de fournir une solution intégrée pour la visualisation de programmes car on retrouve les mêmes plaintes faites à l'encontre des outils CASE, que pour les outils de visualisation ([Wu et al., 2000]) : les outils proposent une seule méthode, le code source est généralement fermé et donc difficilement adaptable au besoin, le rendu n'est pas personnalisable,… Exactement les mêmes difficultés sont rencontrées avec les outils CASE. Dès lors, la solution ne serait-elle pas simplement d'utiliser les outils metaCASE ?

5.2 MetaDONE

MetaDONE fait partie de la famille d'outils metaCASE. Il est développé dans le cadre des activités de l'Université de Namur, sous la responsabilité de Vincent Englebert.

Dans MetaDONE, le langage de méta-modélisation est $MetaL_2$, qui agit comme un méta-méta-modèle. En tant que langage de méta-modélisation, il doit pouvoir décrire de manière efficace et faciliter la description de langages de modélisation. Une des particularités de $MetaL_2$ est que tous les concepts qui y sont définis sont des objets. Ces concepts sont au nombre de quatre et sont au cœur de $MetaL_2$:

- meta-model : représente un agrégat de meta-objects, et par extension un méta-modèle;

- meta-object : cela représente tout objet qui a de l'intérêt dans un langage de modélisation. Il ne contient pas d'information particulière mais les meta-objects peuvent s'hériter entre eux ;
- meta-property : il s'agit d'une propriété qui contient une valeur d'un type donné et possède une cardinalité qui indique son occurrence. De plus il appartient à un meta-object qui est donc son propriétaire ;
- meta-role : dans ce contexte, un meta-role doit être vu comme une relation entre deux meta-objects, qui sont appelés *domain* et *range*. Un meta-role a aussi une cardinalité qui peut être *one-to-one*, *one-to-many*, *many-to-many* ou bien *manys-to-manys*.

Ces quatre concepts sont en fait tous des meta-objects : meta-model, meta-property et meta-role sont en fait des sous-types de meta-object. Il est à noter que grâce à cette structure d'héritage, un meta-model peut très bien inclure un autre meta-model par exemple. Cela offre beaucoup de possibilités ; avec ces quatre éléments de base, toute composition est permise. Il est en outre possible d'étendre ces objets pour en rajouter de nouveaux.

De manière incomplète mais en ayant les éléments essentiels, voici un diagramme de classes qui représente ce qui vient d'être vu. Il s'agit surtout d'offrir une vue schématique :

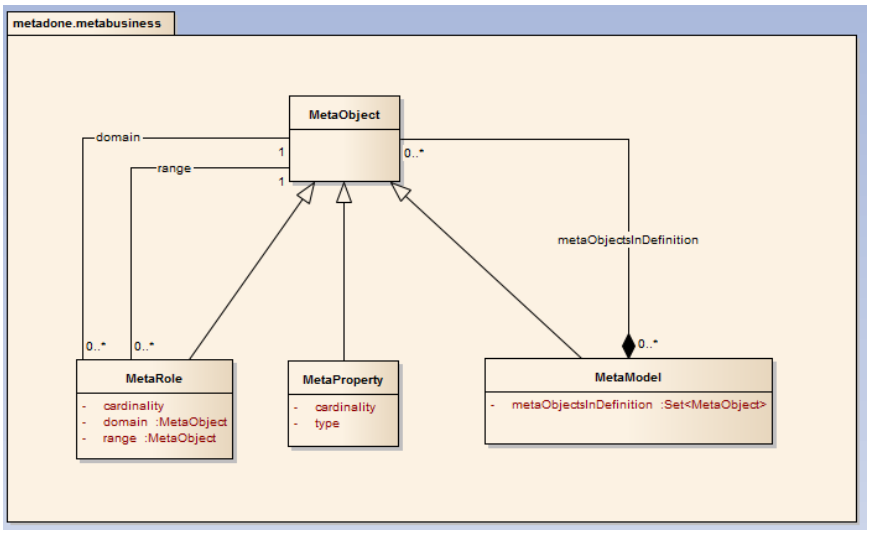


FIGURE 16 –DIAGRAMME DE CLASSES METAL$_2$ POUR LA PARTIE MÉTA.

Ces concepts, et plus si besoin, sont donc les éléments qu'il est possible d'utiliser pour définir un méta-modèle. Une fois le méta-modèle mis en place, l'étape suivante est son instanciation. Les concepts instanciés sont appelés éléments concrets : à nouveau, l'élément de base est le *concrete-object* (*ConcreteObject*) et correspond à au moins un meta-object. Tous les autres éléments concrets sont au moins concrete-object. Il y a un *concrete-role* et une *concrete-property* qui correspondent à leur équivalent méta. Le *concrete-model* est une composition de conrete-objects.

Une fois cela assimilé, il reste à prendre connaissance d'un composant central et puissant de MetaDONE : *Grasyla* (GRAphical SYmbolic LAnguage). Grasyla est un DSML permettant de décrire le rendu graphique d'un méta-modèle : la manière dont la visualisation d'un méta-modèle doit être construite. Grasyla est un DSML et possède un méta-modèle qui peut être directement implémenté dans MetaDONE à l'aide de MetaL$_2$. Le modèle concret de Grasyla est dès lors un script dans lequel figure une description de la visualisation désirée, description faite

par l'intermédiaire du langage Grasyla. Le script peut être interprété par MetaDONE qui va l'appliquer au modèle concret du méta-modèle pour lequel le script a été écrit.

Concrètement, un script Grasyla consiste en un ensemble d'équations qui mettent en relation des meta-objects (partie de gauche) avec des expressions (partie de droite). Ces dernières vont alors décrire le rendu graphique du meta-object cible. Ces expressions sont composées d'autres équations ou expressions ce qui permet d'enchaîner les déclarations et de construire la visualisation désirée. Grasyla associe donc des concepts abstraits à des notations concrètes pouvant prendre diverses formes : des images, du texte, des formes géométriques ou autres.

Des exemples plus pratiques seront présentés dans le prochain point et permettront certainement de concrétiser ce qui vient d'être vu. Pour plus de détails concernant le fonctionnement de MetaDONE et de Grasyla, il convient de se référer à la documentation officielle [Englebert et al., 2013]. [Magusiak, 2012] fournit de plus amples informations concernant Grasyla en lui-même et son intégration dans MetaDONE**.**

Il est à noter qu'afin de faciliter le travail et la mise en place de l'environnement de développement pour MetaDONE, un petit guide a été écrit et intégré à la documentation officielle de MetaDONE comme « technical report ».

5.3 Visualisation proposée

La visualisation proposée consiste en une sorte de diagramme de classe UML augmenté : il contient moins d'informations statiques (pas les attributs des classes par exemple) mais plus d'informations dynamiques (les appels de méthodes effectifs notamment). Les interactions, certains aspects dynamiques y sont donc intégrés.

Le but principal est d'identifier le rôle et l'importance de chaque classe qu'il a été décidé d'examiner, de même que pour les méthodes. Les interactions entre classes et méthodes (les appels donc) doivent aussi y figurer, avec en même temps la vision statique et la contrepartie dynamique.

Il serait bien entendu possible d'effectuer cette augmentation avec d'autres types de diagrammes et cela serait d'ailleurs un exercice particulièrement intéressant dans la mesure où ces différents diagrammes augmentés pourraient constituer les différentes vues dont on a besoin afin de compléter la compréhension.

5.3.1 Méta-modèle

Le méta-schéma définit pour notre modèle comprend six meta-objects reliés entre eux par neuf meta-roles, chacun de ces éléments ayant des meta-properties. Nous pouvons voir le schéma du méta-modèle sur la figure suivante :

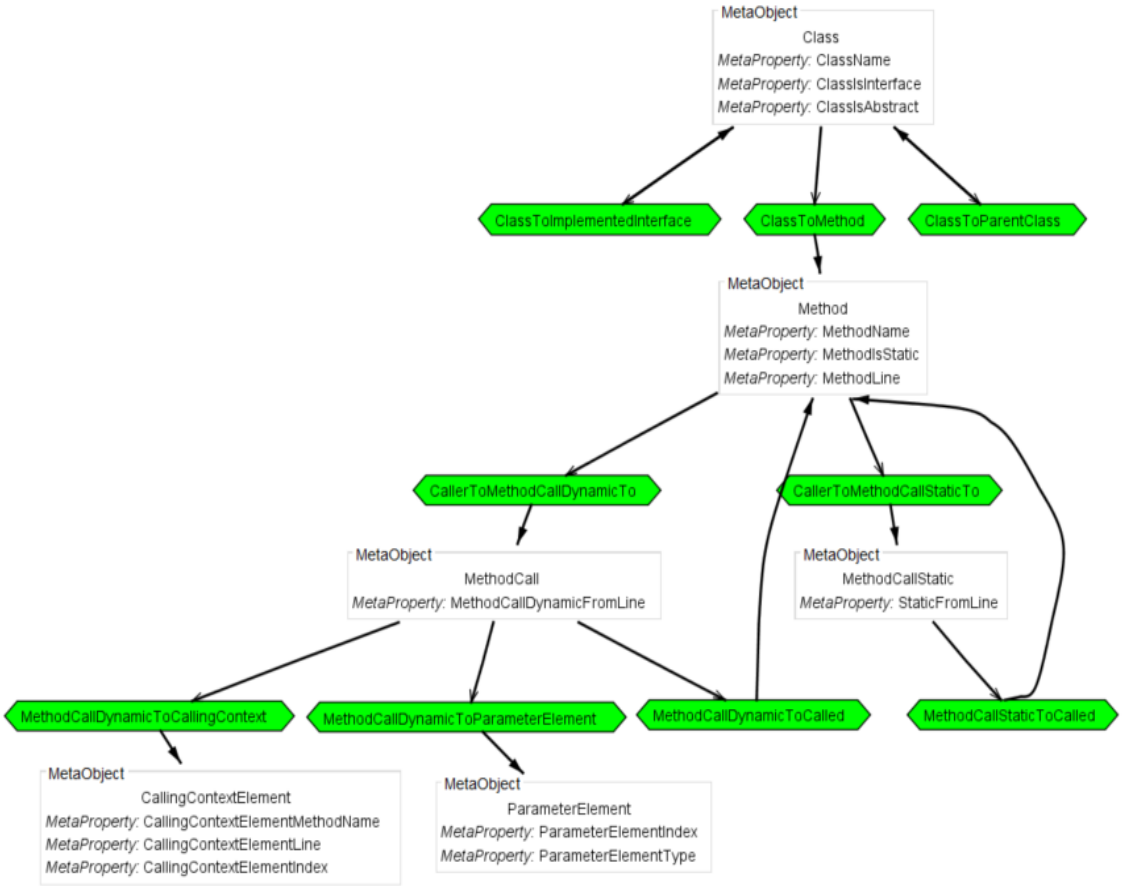


FIGURE 17 – RENDU OFFERT PAR METADONE DU MÉTA-MODÈLE POUR LA VISUALISATION DE L'EXÉCUTION D'UN LOGICIEL.

Description :

- *Class* : modélise simplement une classe au sens orienté objet du terme. Ses propriétés sont un nom complet (nom du package ainsi que le nom de la classe), une valeur booléenne qui indique si c'est une interface ou non, une valeur booléenne qui indique s'il s'agit d'une classe abstraite ou non. Le meta-object Class particpe à plusieurs relations (meta-roles) :
  - *ClassToImplementedInterface :* relie la classe aux interfaces éventuelles qu'elle implémente ;
  - *ClassToParentClass* : relie la classe à la classe dont elle hérite s'il y en a une. Petite exception, en Java toutes les classes héritent de *java.lang.Object*. Pour des raisons de lisibilité, si une classe hérite directement de *java.lang.Object*, ce n'est pas indiqué ;
  - *ClassToMethod* : lie une classe avec aux méthodes qu'elle déclare.
- *Method* : modélise une méthode. À nouveau plusieurs propriétés : le nom de la méthode, une valeur booléenne indiquant si la méthode est statique ou non, le numéro

de la première ligne de la déclaration de la méthode. Ce meta-object participe à deux relations :

  - *CallerToMethodCallStaticTo* : représente l'appelant d'une méthode ;
  - *CallerToMethodCallDynamicTo* : comme pour le meta-role précédent, mais cette fois-ci pour l'appel effectif, capturé à l'exécution.

- *MethodCallStatic* : représente un appel à une méthode qui a été détecté lors de la phase d'analyse statique. Sa seule propriété est le numéro de la ligne dans la méthode appelante d'où l'appel a été effectué. Il participe à deux relations :
  - *CallerToMethodCallStaticTo* : dans lequel il est le *range* ;
  - *MethodCallStaticToCalled* : dans lequel il est le *domain*. Cette relation se dirige vers un meta-object *Method* qui représente la méthode appelée.
- *MethodCall* : représente un appel effectif à une méthode, donc appel qui a été capturé à l'exécution, avec le code instrumenté. Nous disposons ici de plus d'informations car le contexte d'exécution est aussi capturé. Sa seule propriété directe est le numéro de la ligne dans la méthode appelante d'où l'appel a été effectué. *MethodCall* participe à quatre relations :
  - *CallerToMethodCallDynamicTo* : il agit ici en tant que *range* ;
  - *MethodCallDynamicToCalled* : il est ici le *domain* de la relation, relation qui se dirige vers un meta-object *Method* qui n'est autre que la méthode effectivement appelée ;
  - *MethodCallDynamicToParameterElement* : est le *range* dans la relation un-à-plusieurs. Tous les *ParameterElement* sont les paramètres effectifs envoyés à la méthode effectivement appelée ;
  - *MethodCallDynamicToCallingContext* : est le *range* dans la relation un-à-plusieurs. Tous les *CallingContextElement* auxquels il est relié forme le contexte d'appel effectif.
- *ParameterElement* : représente un paramètre d'une méthode. À ce titre, il contient deux propriétés qui sont la position du paramètre dans l'appel effectué, et le type effectif, déduit à l'exécution, du paramètre ;
- *CallingContextElement* : représente une entrée du contexte d'appel effectif. Il s'agit d'un des appels de méthode qui a conduit à l'appel auquel il est relié. Ce meta-object a pour propriétés le nom de la méthode appelante, le numéro de ligne à laquelle l'appel figure, et sa position dans le contexte d'appel, 1 étant l'appel précédent, 2 l'appelant de 1, etc.

5.3.2 Vue

Pour les besoins du travail et des expérimentations, un petit programme a été créé, dans lequel divers cas de figures ont été intégrés : de l'héritage et des implémentations, des appels statiques, des classes abstraites, des interfaces ainsi que du polymorphisme notamment. Une analyse complète a été faite sur ce petit programme, et la visualisation résultante est donnée par la figure suivante, qui est le diagramme de classe augmenté proposé :

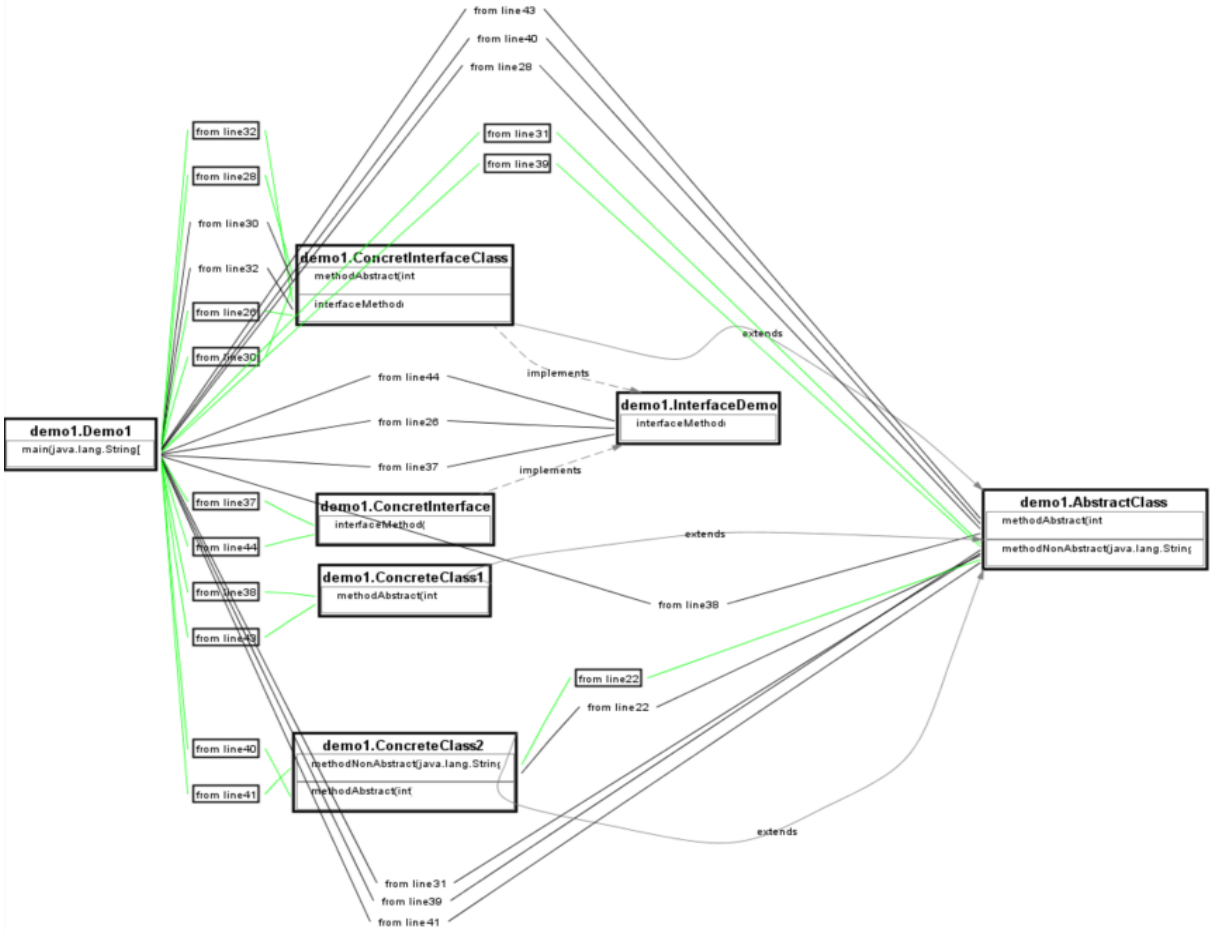


FIGURE 18 – VISUALISATION AVEC VIZUDONE RÉSULTANTE DE L'ANALYSE D'UN PROGRAMME DE DÉMONSTRATION.

Description :

- Chaque meta-object Class apparaît dans un cadre avec des bords noir épais et dont le nom est inscrit en gras (figure suivante) ;

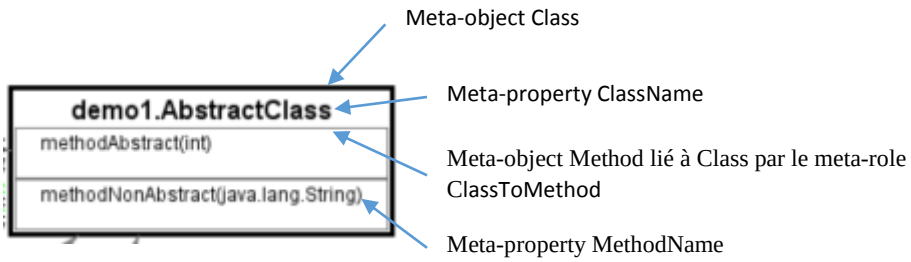


FIGURE 19 – DÉTAILS DU MÉTA-OBJECT CLASS ET METHOD.

- Chaque meta-object Method apparaît dans le cadre de la Class qui la déclare (figure précédente) ;
- Les appels entrants à une méthode sont des arcs qui viennent sur la gauche du cadre représentant la méthode (figure suivante à gauche). Les appels sortants d'une méthode, donc les appels émis depuis la méthode sont des arcs situés à droite du cadre représentant la méthode appelante (figure suivante à droite) ;

FIGURE 20 – REPRÉSENTATION DES APPELS SORTANTS ET ENTRANTS DANS LA VISUALISATION.

- Les appels détectés lors de la phase d'analyse sont indiqués en noir. Il n'y est affiché que le numéro de ligne depuis laquelle l'appel a été fait (figure suivante) :

FIGURE 21 – REPRÉSENTATION DU MÉTA-OBJECT METHODCALLSTATIC.

- Les appels effectifs, capturés à l'exécution sont indiqués en vert clair. Par défaut, en leur milieu un cadre contenant la ligne depuis laquelle l'appel a été fait est indiqué. Ce cadre s'ouvre quand on clique dessus pour laisser apparaître le détail du contexte d'appel ainsi que les paramètres effectifs (figure suivante) :

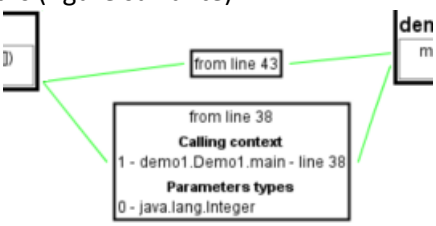


FIGURE 22 –REPRÉSENTATION DU MÉTA-OBJECT METHODCALL.

- Les héritages et les implémentations sont représentés par des arcs gris ayant une flèche du côté de la classe parent ou de l'interface implémentée (respectivement). Dans le cas d'une implémentation d'interface, l'arc est en pointillé (figure suivante) :

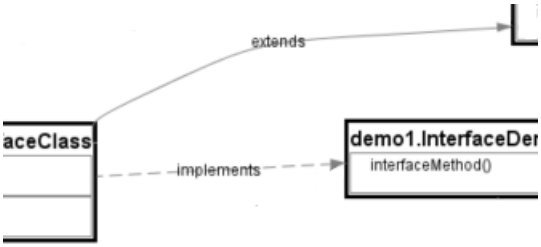


FIGURE 23 – REPRÉSENTATION DES LIENS D'HÉRITAGE ET D'IMPLÉMENTATION.

Comme on peut le voir sur la figure suivante, il avait été analysé qu'un appel depuis la ligne 22 de la méthode *methodNonAbstract(java.lang.String)* de la classe *demo1.ConcreteClass2* devait être fait vers la méthode *methodNonAbstract(java.lang.String)* de la classe *demo1.AbstractClass* et c'est bien ce qui s'est passé à l'exécution. En fait *demo1.ConcreteClass2* hérite de *demo1.AbstractClass* et il s'agit donc d'un appel depuis la méthode redéfinie *methodNonAbstract* dans le fils, vers la méthode du parent.

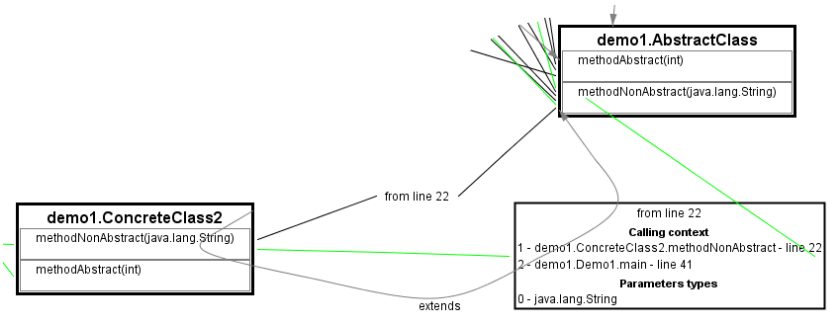


FIGURE 24 –EXEMPLE DE REPRÉSENTATION D'UN APPEL DE MÉTHODE.

Dans la figure suivante, un cas de polymorphisme est montré. Il avait été analysé qu'un appel depuis la ligne 40 de la méthode *main(java.lang.String[])* de la classe *demo1.Demo1* devait être fait vers la méthode *methodAbstract(int)* de la classe *demo1.AbstractClass*. À l'exécution ce n'est pas ce qui s'est passé : l'appel est en fait arrivé à la méthode *methodAbstract(int)* mais de la classe *demo1.ConcreteClass2* qui hérite de *demo1.AbstractClass*. Il s'agit donc d'un appel à une méthode redéfinie dans le fils.

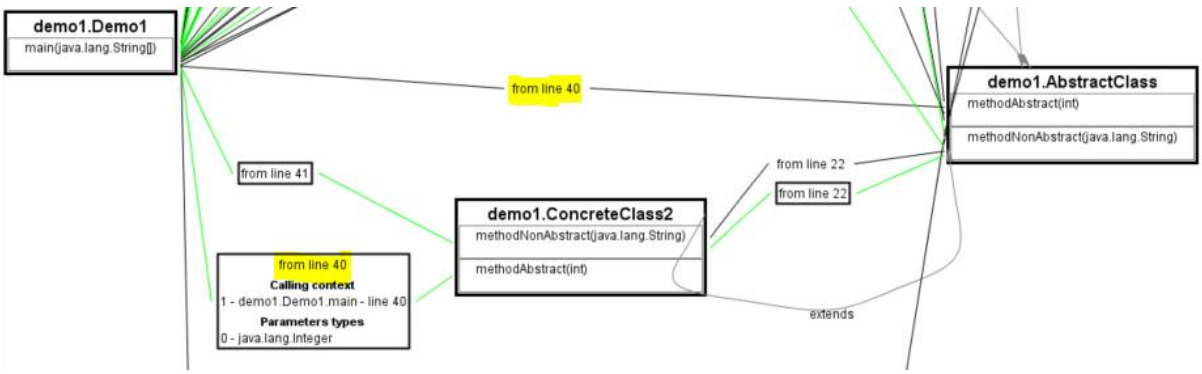
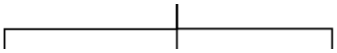


FIGURE 25 – EXEMPLE DE REPRÉSENTATION D'UN CAS DE POLYMORPHISME AVEC MÉTHODE REDÉFINIE.

Un reproche à faire à MetaDONE est la manière dont il gère le rendu des liens entre les éléments. Comme on peut le voir sur la figure précédente, même avec une vue localisée, sur un petit ensemble d'éléments, le rendu est assez anarchique, ce qui rend la visualisation assez lourde. Il serait notamment intéressant de regrouper automatiquement tous les arcs appartenant à un même point d'ancrage en un seul, comme montré ci-après:



De plus, il y a beaucoup de croisements et les arcs sont trop proches les uns des autres. Cela rendrait la visualisation encore plus fluide si ça n'était pas le cas. Il est tout à fait possible de déplacer les éléments à la main pour les ré-agencer, mais s'il faut le faire à chaque fois, cela devient vite lourd. Une solution assez rapide à mettre en place serait de gérer le contenu d'un arc (ce qui s'affiche en son milieu) de la même manière qu'est géré un meta-object.

FIGURE 26 – ILLUSTRATION DU PROBLÈME DE SUPERPOSITION DES ARCS DANS METADONE ET SA SOLUTION PARTIELLE.

En effet, il semblerait que le fait que ça soit un meta-role change la manière dont ses éléments sont gérés. Sur la figure précédente, à gauche on peut y avoir un arc classique qui représente un meta-role. Sur la droite, il s'agit d'un moyen détourné mais qui permet d'avoir des espacements corrects. Sa structure est : meta-role -> meta-object (contient le message « from line x) -> meta-role. C'est pourquoi il y a lieu de penser que leur traitement est différent. Il devrait donc être possible de traiter le contenu d'un meta-role comme un meta-object, ce qui éviterait d'avoir recours à une construction détournée.

La visualisation qui est présentée ici est le résultat de beaucoup d'essais et de changements. Au début, il était question d'avoir une vue plus globale, avec énormément d'informations mais la lourdeur de la visualisation qui en découlait nous a forcé à revoir cela. Il fallait faire un choix des données à afficher. Finalement, le fait que la vue soit localisée permet de résoudre partiellement ce problème.

5.4 Multi-vues

Comme cela a été vu dans la « *Partie I : État de l'art »,* il est difficilement possible d'afficher tous les différents aspects dynamiques en une fois sans rendre la représentation complètement illisible et incompréhensible (surcharge). À cela, la réponse est de spécialiser les vues. Il faut alors faire attention à ne pas les multiplier de manière trop intensive à défaut de quoi, il y aurait trop de vues et l'information serait dès lors trop diluée. Une plainte récurrente ([Wu et al., 2000]) des utilisateurs d'outils qui présentent plusieurs vues pour différents aspects, est qu'ils doivent établir de manière conceptuelle des relations entre les différentes vues. Cela peut provoquer une désorientation.

Dans [Caserta, 2012], il est conclu qu'une visualisation efficace, couvrant tous les aspects, permettant ainsi une compréhension complète de l'exécution n'existe pas. L'utilisateur désireux d'en arriver à ce point doit alors combiner les outils et les différentes représentations.

MetaDONE et les outils metaCASE semblent forts à propos pour ce genre d'exercice car ils peuvent facilement, à partir des mêmes données, construire un modèle unique et utiliser des vues, des interprétations différentes. MetaDONE le fait d'ailleurs très bien avec les scripts Grasyla qui offrent une très grande souplesse et permettent donc, pour le même méta-modèle d'avoir une multitude de scripts Grasyla, chacun représentant une vue différente.

Un autre atout est que le rendu est entièrement personnalisable. Ceci n'est peut-être pas vrai pour tous les outils metaCASE, mais pour MetaDONE avec Grasyla par exemple, il n'est pas obligatoire de se limiter à du rendu style graphes : les différentes visualisations disponibles utilisent principalement les graphes, en proposant des techniques de filtrages afin d'améliorer la lisibilité de ces graphes. Il est possible, dans les limites de l'outil bien entendu, de créer la représentation la plus adéquate possible. Notamment, le fait qu'une visualisation différente serait utilisée selon le niveau que l'utilisateur souhaite exploré : packages, classes, méthodes,…

Par contre, il faut que l'outil soit performant. Avec les diverses expérimentations qui ont été menées, pour un programme de bonne taille (+- 100 000 lignes de code) cela génère environ 9 millions de lignes dans le fichier résultat. Des mécanismes doivent être prévus afin de gérer ces volumes de données ; il ne s'agit pas de tout traiter en une fois, mais d'avoir un modèle qui se charge au fur et à mesure de l'exploration. Il serait possible de commencer avec une vue au niveau des packages, donc seules les données des packages et des classes qu'ils contiennent sont nécessaires à ce moment-là, puis quand l'utilisateur clique sur une classe, les détails de cette dernière sont chargés depuis le fichier résultat de l'analyse et affichés, ainsi de suite. C'est ce qu'on appelle du chargement à la demande (*lazy loading*).

CONCLUSION

La première partie du travail a dressé un état de l'art concernant trois questions fondamentales dans le processus de compréhension des programmes : quelles sont les informations disponibles et intéressantes dans le cadre de la compréhension d'un programme (les métriques), comment récolter ces informations (l'analyse) et la troisième, comment afficher ces informations (la visualisation). Tous les constituants de la démarche de compréhension ont été présentés afin d'avoir un ensemble de paramètres dont il a fallu tenir compte lors de la mise en place de notre solution.

Dans la deuxième partie, il a été présenté une solution complète visant à aider les développeurs dans leurs tâches de compréhension des programmes. Cette solution comprend principalement deux composants, à savoir AnaDONE et VizuDONE, qui seront résumés ci-après. La solution se concentre volontairement sur les aspects dynamiques car ce sont ceux qui représentent le fonctionnement réel d'un programme. De plus, les aspects statiques ont déjà été abondamment étudiés dans d'autres travaux, ce qui n'est pas le cas concernant les aspects dynamiques. Cela a aussi été l'occasion d'utiliser un outil metaCASE, MetaDONe, comme moyen de visualisation : un méta-modèle a été défini et bien entendu une visualisation possible lui a été associée.

Au niveau des apports :

- AnaDONE : il s'agit d'un programme Java, plus précisément d'un agent, qui va récolter les informations nécessaires à propos d'un programme cible, pendant le chargement de ce dernier et lors de son exécution, et ce grâce à une analyse du bytecode ainsi qu'à son instrumentation quand cela est nécessaire. AnaDONE est configurable en plusieurs points, comme le fait de pouvoir spécifier les parties du programme cible à prendre en compte ;
- La distance d'appel : cette métrique permet de détecter, à l'exécution, des design patterns ou des mécanismes de transformation du bytecode. Lorsque sa valeur est supérieure à 1, cela peut se traduire par le fait que ce qu'il se passe réellement dans le programme est en décalage par rapport au programme chargé initialement. Cela est rendu possible par la comparaison des données récoltées lors du chargement du programme cible par rapport aux données récoltées lors de l'exécution de celui-ci, données qui montrent ce qui se passe réellement ;
- Diagramme de classe augmenté : la visualisation proposée se base sur un diagramme de classe UML classique, diagramme auquel certaines données ont été retirées et des informations dynamiques ont été ajoutées. Il offre une vue localisée sur le fonctionnement réel d'un programme en se basant principalement sur les appels de méthode ;
- VizuDONE : c'est le plugin développé pour MetaDONE. Il contient l'implémentation de la visualisation, à savoir le diagramme de classe augmenté. Cette implémentation comprend la définition du méta-modèle, la logique de création du modèle concret à partir des données récoltées par AnaDONE, ainsi que la définition de l'affichage.

Perspectives

- Intégrer d'autres vues au moyen d'autres diagrammes UML dits « augmentés ». De cette manière, il y aurait plus d'informations dynamiques mais regroupées par thème (voir « *5.4 Multi-vues* ») ;
- Avoir une sorte de mode « suivi de flux » : lors de la compréhension d'un nouveau programme, il est généralement intéressant d'avoir une vue en flux. Il n'est pas toujours utile de n'avoir que les interactions directes, l'intérêt est aussi de savoir ce qui se passe ensuite. Cela devrait pouvoir se faire automatiquement. Avec ce qui est actuellement mis en place, ne sont pris en compte que les éléments qui sont explicitement demandés et donc leurs interactions directes. Le mode *suivi de flux* partirait toujours des éléments à prendre en compte mais prendrait aussi, de manière automatique, toutes les interactions partant de là jusqu'à ce que ça revienne.
  Comme d'habitude, il vaut mieux un schéma et un exemple pour rendre cela compréhensible : prenons le cas suivant où il y a trois classes (*ClassA*, *ClassB* et *ClassC*). ClassA a dans une de ses méthodes un appel vers une méthode *methodB* de ClassB. Cette méthode appelle une méthode *methodC* de ClassC, comme décrit sur la figure suivante :

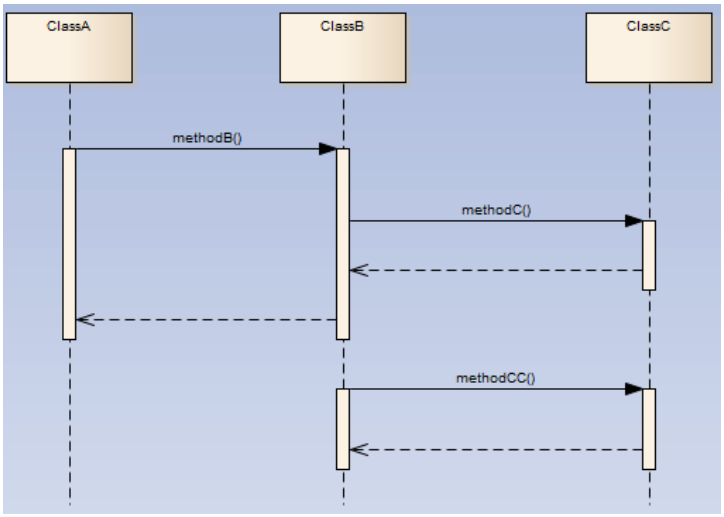


FIGURE 27 – DIAGRAMME DE SÉQUENCE ILLUSTRANT UN EXEMPLE POUR EXPLIQUER LE CONCEPT DE « SUIVI DE FLUX ».

Disons qu'il est indiqué à AnaDONE de ne prendre en compte que ClassA. Tel que c'est implémenté maintenant, la visualisation n'affichera qu'un appel de méthode *methodB* depuis une méthode de ClassA vers ClassB donc. Avec le mode *suivi de flux*, il y aurait en plus l'appel depuis *methodB* de ClassB vers *methodC* de ClassC. Cela permet de suivre le chemin complet. Par contre, ne serait pas présent, l'appel depuis une méthode de ClassB vers la méthode *methodCC* de ClassC car elle ne fait pas partie du flux qui a démarré depuis l'élément ClassA à prendre en compte dans l'analyse.
Nous pensons qu'il s'agirait là d'une fonctionnalité importante dans le processus de compréhension car, et cette image a déjà été utilisée auparavant dans le travail, une fois qu'on a le bout de ficelle, il suffit de dérouler et de suivre cette ficelle pour découvrir le reste. Ici cela se ferait automatiquement;

- Améliorer la gestion graphique des liens entre les éléments dans MetaDONE. Il faudrait : faire en sorte que ces liens de se coupent pas ; qu'une distance de séparation entre eux puisse être définie ou qu'ils ne se superposent pas par défaut ; pouvoir définir des points d'ancrage et que tous les arcs en partance/arrivant à un point d'ancrage soient regroupés afin de ne pas surcharger le rendu (voir pistes proposées dans « *5.3.2 Vue »*) ;
- Améliorer les performances de MetaDONE afin qu'il soit plus rapide pour la gestion de modèle de plus grosse taille, contenant plus d'objets. C'est une limitation assez contraignante pour le moment ; d'un autre côté cela force à optimiser ce qui est fait et à une réflexion quant au fait d'éliminer le superflu ;
- Trouver une méthode plus efficace ou optimiser l'existante pour calculer la métrique appelée *distance d'appel* (voir « *4.9 La distance d'appel »*). Il n'est pas viable de la calculer à l'exécution, cela prend trop de ressources et aurait donc un impact trop grand sur l'application en cours d'analyse, risquant de fausser les résultats. Une analyse après coup semble nécessaire ;
- Avoir une visualisation en temps réel. Cela est tout à fait possible en utilisant des sockets par exemple. Le traceur d'AnaDONE enverrait donc en direct les données au plugin VizuDONE, qui les interprèterait et pourrait compléter le modèle au fur et à mesure. En même temps, les vues liées à ce modèle seront mises à jour en temps plus ou moins réel : le rafraîchissement des vues liées ne devant pas se faire à chaque fois que des données arrivent, mais à intervalle régulier dans le cas où des données sont arrivées, sinon cela va impacter négativement les performances.

BIBLIOGRAPHIE

[Arisholm, 2004] E. Arisholm, L.C. Briand, and A. Foyen, « Dynamic coupling measurement for object-oriented software », Software Engineering, IEEE Transactions on, 30(8), pages 491–506, 2004.

[Ball, 1999] Thomas Ball, « The Concept of Dynamic Analysis », Software Engineering – ESEC/FSE '99, pages 216-234, 1999.

[Briand, 1999] Briand, Lionel C., John W. Daly and Jurgen K. Wust, « A unified framework for coupling measurement in object-oriented systems », Software Engineering, IEEE Transactions on 25.1, pages 91-121, 1999.

[Caserta, 2012] Pierre Caserta, « Analyse statique et dynamique de code et visualisation des logiciels via la métaphore de la ville : contribution à l'aide à la compréhension des programmes », Thèse, Université de Lorraine, 2012.

[Chawla et al., 2004] Anil Chawla and Alessandro Orso, "A generic instrumentation framework for collecting dynamic information", ACM SIGSOFT Software Engineering Notes 29.5, pages 1-4, 2004.

[Cornelissen et al., 2009] Cornelissen, Bas, et al., « A systematic survey of program comprehension through dynamic analysis. », Software Engineering, IEEE Transactions on 35.5, pages 684-702, 2009.

[Diehl, 2007] Stephan Diehl, *Software Visualization : Visualizing the Structure, Behaviour, and Evolution of Software*, Springer, Berlin, 2007.

[Dufour et al., 2003] Bruno Dufour, Karel Driesen, Laurie Hendren and Clark Verbrugge, « Dynamics Metrics for Java », Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, pages 149-168, 2003.

[Dunsmore, 1998] Alastair Dunsmore, « Comprehension and visualisation of object-oriented code for inspections. », Empirical Foundations of Computer Science (EFoCS), University of Strathclyde Livingstone Tower, 1998.

[Ebert et al., 1997] Jürgen Ebert, Roger Süttenbach and Ingar Uhe, « Meta-CASE in Practice: a Case for KOGGE. », Advanced Information Systems Engineering, Springer, Berlin, 1997.

[Englebert et al., 2013] Vincent Englebert and Krzysztof Magusiak, « The Grasyla 2.0 Language Edition 1.2 (Draft). », Technical report, University of Namur, 2013.

[Gamma et al., 1994] Gamma, Erich, Johnson and Vlissides, *Design patterns: elements of reusable object-oriented software*, Pearson Education, 1994.

[Gershon et al., 1997] Nahum Gershon, Stephen G. Eick, « Information visualization. », IEEE Computer Graphics and Applications 17.4, pages 29-31, 1997.

[Gračanin et al., 2005] Denis Gračanin, Krešimir Matković, and Mohamed Eltoweissy, « Software visualization. », Innovations in Systems and Software Engineering 1.2, pages 221-230, 2005.

[Gupta et al., 2008] Gupta, Varun, and Jitender Kumar Chhabra, « Measurement of dynamic metrics using dynamic analysis of programs. », Proceedings of the Applied Computing Conference, 2008.

[Hassoun et al., 2004] Youssef Hassoun, Roger Johnson and Steve Counsell, « Empirical Validation of a Dynamic Coupling Metric », Birkbeck College London, Technical Report, 2004.

[Isazadeh, 1997] Hosein Isazadeh, « Architectural analysis of MetaCase. », Thesis, Queen's University, Canada, 1997.

[Koschke, 2003] Rainer Koschke, « Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. », Journal of Software Maintenance and Evolution: Research and Practice 15.2, pages 87-109, 2003.

[Kuleshov, 2007] E. Kuleshov, « Using ASM framework to implement common bytecode transformation patterns », AOSD.07, 2007.

[Magusiak, 2012] Krzysztof Magusiak, « Extensible DSL for specifying editors in a metaCASE tool. », Thesis, University of Namur, 2012.

[Maletic et al., 2002] Jonathan I. Maletic, Andrian Marcus, and Michael L. Collard, « A task oriented view of software visualization. », Visualizing Software for Understanding and Analysis, 2002. Proceedings. First International Workshop on, 2002.

[MetaCase, 2004] MetaCase, « ABC to metaCASE technology », White paper, http://www.metacase.com/papers , 2004.

[Petre et al., 1998] Marian Petre, A. F. Blackwell, T. R. G. Green, « Cognitive questions in software visualization. », Software visualization: Programming as a multimedia experience, pages 453-480, 1998.

[Reiss et al., 2003] Reiss, Steven P, « Visualizing Java in action. », Proceedings of the 2003 ACM symposium on Software visualization, 2003.

[Rohr et al., 2008] Matthias Rohr et al., « Kieker: Continuous monitoring and on demand visualization of Java software behavior. », Proceedings of the IASTED International Conference on Software Engineering, pages 80-85, 2008.

[Roman et al., 1992] Gruia Roman, Kenneth C. Cox, « Program visualization: The art of mapping programs to pictures. », Proceedings of the 14th international conference on Software engineering, 1992.

[Singh et al., 2013] Paramvir Singh, « Design and validation of dynamic metrics for object-oriented software systems. », Thesis, Guru Nanak Dev University, 2013.

[Storey et al., 1997] M-AD Storey, Kenny Wong, and Hausi A. Muller, « How do program understanding tools affect how programmers understand programs?. », Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on, 1997.

[Storey et al., 1997b] M-AD Storey et al, « On integrating visualization techniques for effective software exploration. », In Information Visualization, 1997. Proceedings., IEEE Symposium on, pages 38-45, 1997.

[Storey et al., 2005] Margaret-Anne Storey, Davor Čubranić, Daniel M. German, « On the use of visualization to support awareness of human activities in software development: a survey and a framework. », Proceedings of the 2005 ACM symposium on Software visualization, 2005.

[Wettel et al., 2007] Richard Wettel, and Michele Lanza, « Visualizing software systems as cities. », Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on, 2007.

[Wikipedia, 2013a] Wikipédia, l'encyclopédie libre, Couplage (informatique), http://fr.wikipedia.org/w/index.php?title=Couplage_(informatique)&oldid=92184803 (17 avril 2013, 15:28 UTC) (Page consultée le 18 juillet 2014).

[Wikipedia, 2013b] Wikipédia, l'encyclopédie libre, Probe effect, http://en.wikipedia.org/w/index.php?title=Probe_effect&oldid=566512685 (31 July 2013, 01:33 UTC) (Page consultée le 22 juillet 2014).

[Wikipedia, 2013c] Wikipédia, l'encyclopédie libre, Arbre syntaxique abstrait, http://fr.wikipedia.org/w/index.php?title=Arbre_syntaxique_abstrait&oldid=90164419 (14 mars 2013, 17:53 UTC) (Page consultée le 3 août 2014).

[Wikipedia, 2014a] Wikipédia, l'encyclopédie libre, Métrique (logiciel), http://fr.wikipedia.org/w/index.php?title=M%C3%A9trique_(logiciel)&oldid=103966580 (20 mai 2014, 14:21 UTC) (Page consultée le 19 juillet 2014).

[Wikipedia, 2014b] Wikipédia, l'encyclopédie libre, Polymorphisme (informatique), http://fr.wikipedia.org/w/index.php?title=Polymorphisme_(informatique)&oldid=104683357 (15 juin 2014, 14:53 UTC) (Page consultée le 3 août 2014).

[Wikipedia, 2014c] Wikipédia, l'encyclopédie libre, Fentes de Young, http://fr.wikipedia.org/w/index.php?title=Fentes_de_Young&oldid=105135054 (2 juillet 2014, 16:36 UTC) (Page consultée le 22 juillet 2014).

[Wikipedia, 2014d] Wikipédia, l'encyclopédie libre, Intelligence amplification, http://en.wikipedia.org/w/index.php?title=Intelligence_amplification&oldid=621851569 (19 August 2014, 01:02 UTC (Page consultée le 22 août 2014).

[Wikipedia, 2014e] Wikipédia, l'encyclopédie libre, Decorator pattern, http://en.wikipedia.org/w/index.php?title=Decorator_pattern&oldid=622607344 (24 August 2014, 14:44 UTC) (Page consultée le 27 août 2014).

[Wu et al., 2000] Jingwei Wu, and Margaret-Anne D. Storey, « A multi-perspective software visualization environment. », Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research, 2000.

[Yacoub et al., 1999] Sherif M. Yacoub, Hany H. Ammar and Tom Robinson, « Dynamic metrics for object oriented designs », Software Metrics Symposium, 1999. Proceedings. Sixth International, 1999.

[Young et al., 1998] Peter Young and Malcolm Munro, « Visualising software in virtual reality. », Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on, 1998.

[Zaidman, 2006] Andy Zaidman, « Scalability solutions for program comprehension through dynamic analysis. », Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on IEEE, 2006.