MASTER EN SCIENCES INFORMATIQUES

Etude de l'utilisation d'un débogueur visuel pour le développement et la maintenance de logiciels

Dassonville, Jérémy

Award date:
2012

Awarding institution:
Universite de Namur

Étude de l'utilisation d'un débogueur visuel pour le développement et la maintenance de logiciels

Jérémy Dassonville

Mémoire réalisé
pour l'obtention du diplôme de Master en Sciences Informatiques

Année académique 2011-2012

Option : Ingénierie Logicielle
Promoteur : Naji Habra

Faculté d'Informatique
Facultés Universitaires Notre-Dame de la Paix • Rue Grandgagnage, 21 • B-5000 Namur

Résumé

Dans le domaine du génie logiciel, beaucoup de développeurs ignorent la plus-value que l'utilisation d'un débogueur peut apporter dans le cadre du développement et de la maintenance d'un nouveau projet informatique. À la place d'employer cet outil, nombreux sont ceux qui ont recours à la fonction `System.out.println` ou aux `loggers` pour afficher les états et la valeur de la variable sur la console de développement.

Il existe également de nombreux outils de visualisation qui ont pour but d'analyser divers programmes pour calculer des métriques. Suivant leurs résultats, les programmeurs peuvent corriger plus rapidement leur projet. Mais l'analyse de ces métriques peut prendre énormément de temps. C'est pourquoi ils souhaiteraient pouvoir examiner un programme lors de son exécution en temps réel et non plus sur des états dans le temps.

Le but de ce mémoire est d'apporter une solution qui consiste à ajouter une communication entre un outil de visualisation et le débogueur pour permettre aux développeurs d'utiliser ce service de manière intuitive, et aux analystes de pouvoir examiner un programme avec un état bien défini dans le temps.

Mots clés : Visualisation, Débogueur, *VERSO*, Maintenance, Développement, Logiciel, Eclipse, Plug-in

Abstract

In the software engineering field, lots of developers ignore the gain that the use of the debugger can bring for the maintenance and the development of a new computer project. Instead of employing this tool, lots of them use the `System.out.println` function or the `loggers` to display the state and the value of the variable in the development console.

There are also a lot of visualization tools which aim to analyze softwares to compute some metrics. According to the results, developers can correct quickly their project. But the analysis of these metrics can take a long time. It is the reason why they would like to examine a program during its own execution in real time rather than states over time.

The aim of this memory is to bring a solution which consists in adding a communication between a visualization tool and the debugger to allow developers to use it in an intuive manner, and the analysts to check a program with a well defined state in the time.

Keywords : Visualization, Debugger, *VERSO*, Maintenance, Development, Software, Eclipse, Plugin.

Table des matières

Table des matières **iii**

Table des figures **v**

Remerciements **vii**

1 Introduction **1**
1.1 Contexte 1
1.2 Problématique 2
1.3 Proposition 3
1.4 Structure du mémoire 4

2 État de l'art **5**
2.1 Quelques approches d'amélioration de l'interface d'Eclipse 5
2.1.1 Code Canvas 5
2.1.2 Code Bubbles 9
2.2 Quelques outils de visualisation 14
2.2.1 Visualisation de la qualité 14
2.2.2 Visualisation de traces d'exécution 16

3 Proposition d'un outil d'aide à la maintenance et au développement de logiciels **22**
3.1 L'outil de débogage Eclipse 22
3.1.1 Présentation de l'outil 23
3.1.2 Fonctionnement 26
3.2 L'outil de visualisation *VERSO* 29
3.2.1 Présentation de l'outil 29
3.2.2 Fonctionnement 31
3.3 Proposition de la solution 34

4 Développement de l'outil : *VersoDebug* 36
4.1 Un débogueur visuel 36
4.1.1 Ajout de la communication entre les deux outils 37
4.1.2 Écoute des notifications du débogueur 37
4.2 Représentation de la pile d'exécution 38
4.2.1 *FishEye* OpenGL 40
4.2.2 *FishEye* personnalisé 42
4.2.3 Aperçu de l'outil lors d'une exécution 44
4.3 Représentation du contexte 46
4.4 Le multi-threading 48

5 Évaluation de l'outil *VersoDebug* 50
5.1 Présentation de l'expérimentation 50
5.1.1 Les sujets 50
5.1.2 Le temps 51
5.1.3 L'environnement de travail 51
5.1.4 Les questions 52
5.2 Résultats et analyses 55
5.2.1 Questionnaire préliminaire 55
5.2.2 Correction du bogue 56
5.2.3 Questionnaire final 58
5.2.4 Commentaires de l'expérimentation 60

6 Conclusion 61
6.1 Réalisations de l'existant 61
6.2 Contributions du mémoire 62
6.3 Travaux futurs 63

Bibliographie 64

Annexe 67

A Code source de l'expérimentation 67

Table des figures

1.1 Tâches de maintenance avec leurs causes à effets 3

2.1 Environnement de ***Code Canvas*** dans l'éditeur Visual Studio 6
2.2 Exemple d'un zoom sémantique 7
2.3 Structure d'un système sous ***Code Canvas*** 8
2.4 Environnement de travail de ***Code Bubbles*** 10
2.5 Environnement de débogage de ***Code Bubbles*** 12
2.6 CodeCity qui prend les sources du projet Java 1.5 en paramètre 15
2.7 Représentation du projet Azureus par *VERSO* 16
2.8 Exemple de représentation d'un système dans Evospace 17
2.9 Vue jour et nuit dans Evospace 17
2.10 Exemple d'un arbre d'appels, d'un graphe et d'un arbre de contexte d'appels 18
2.11 Approche de transformation en arbre de contexte d'appels acycliques 19
2.12 Transformation d'un arbre d'appels en Sunburst 20
2.13 L'outil VASCO 20
2.14 Exemple de changement de racine 21

3.1 Représentation de la vue Debug dans Eclipse 23
3.2 Représentation de la vue *Variables* 25
3.3 Représentation d'une liste dans la vue *Variables* 25
3.4 Représentation de la vue *Breakpoint* 26
3.5 Représentation UML du modèle de débogage sous Eclipse 27
3.6 Illustration de *VERSO* dans l'environnement d'Eclipse 30
3.7 Matrice 3D des vues atteignables dans *VERSO* par rapport à la granularité et l'évolution du logiciel au cours du temps 30
3.8 Diagramme de classe représentant les composants importants de *VERSO* . . 33
3.9 Diagramme de classe représentant la fusion des composants importants du débogueur et de *VERSO* 35

4.1 Ajout des dépendances pour la communication entre *VersoDebug* et le débogueur d'Eclipse . . . 37
4.2 Champ de vision du poisson . . . 41
4.3 Champ de vision humain . . . 41
4.4 Effet FishEye . . . 42
4.5 Rose des vents . . . 43
4.6 Représentation du programme *JHotDraw* avant de commencer le débogage . 44
4.7 Début de débogage avec l'effet *FishEye* personnalisé autour de la classe contenant la méthode *main* . . . 45
4.8 Premier exemple de dégradé avec les effets *FishEye* . . . 45
4.9 Deuxième exemple de dégradé avec les effets *FishEye* . . . 46
4.10 Exemple de dégradé avec vue sur les méthodes . . . 46
4.11 Représentation du contexte d'une méthode . . . 47
4.12 Diagramme de séquence exprimant le comportement de *VersoDebug* lors du changement de sélection d'un *thread* . . . 49

5.1 Questions préliminaires . . . 52
5.2 Questionnaire concernant l'effort fourni pour accomplir une tâche . . . 53
5.3 Échelle de mesure pour le questionnaire de la Nasa-TLX . . . 54
5.4 Questionnaire après l'expérimentation pour les témoins . . . 54
5.5 Questionnaire après l'expérimentation pour ceux ayant utilisé *VersoDebug* . . 55

Remerciements

Je souhaite adresser mes remerciements les plus sincères aux personnes qui m'ont apporté leur aide et qui ont contribué à l'élaboration de ce mémoire, en particulier Krzysztof Magusiak et Nicolas Soyeur pour leurs conseils avisés.

Je tiens à remercier plus particulièrement Monsieur Naji Habra qui, en tant que promoteur de mémoire, s'est toujours montré à l'écoute et très disponible tout au long de sa réalisation. Aussi pour l'inspiration, l'aide et le temps qu'il a bien voulu me consacrer et sans qui ce mémoire n'aurait jamais vu le jour.

Je remercie également les Professeurs Houari Sahraoui et Bruno Dufour pour m'avoir accueilli au sein de leur laboratoire GEODES à Montréal (Canada) et pour leur disponibilité durant mes quatre mois de stage.

Je n'oublie pas ma famille et mes amis pour leur contribution, leur soutien et leur patience. Je tiens à exprimer ma reconnaissance à tous ceux qui ont eu la gentillesse de lire et corriger ce travail.

J'ai une pensée particulière pour Laura. Merci pour la patience dont elle a fait preuve durant mon absence et pour ses encouragements.

Merci à tous et à toutes.

Chapitre 1

Introduction

Le sujet de ce mémoire traite de l'élaboration d'un débogueur visuel dans le but d'améliorer les informations procurées par l'outil de débogage d'Eclipse. Ce débogueur s'intègre à un plug-in existant qui contient un contexte visuel. Ce contexte représente le projet en cours de développement en 3D dans son environnement de travail, et affiche la pile d'exécution d'un processus d'un programme en cours de débogage. Les informations récoltées par le plug-in, que sont les méthodes avec leurs variables globales et locales, sont fournies par le débogueur d'Eclipse et sont ensuite traduites par notre outil sous forme d'une représentation graphique. Cette représentation a pour but de faciliter la compréhension de l'exécution du programme en cours de débogage tout en montrant un maximum d'informations en utilisant peu de vue à l'intérieur de l'environnement de développement intégré d'Eclipse. Pour aider les développeurs à apercevoir les classes ou méthodes qui se trouvent dans la pile d'exécution du processus, un focus visuel va s'appliquer sur elles, dans le but d'attirer l'attention du programmeur vers ces classes ou méthodes. Les utilisateurs de cet outil peuvent donc voir très facilement quelles sont les classes qui sont le plus utilisées lors de l'exécution du logiciel et ainsi se concentrer sur elles.

1.1 Contexte

Dans le milieu professionnel, beaucoup de logiciels subissent plusieurs maintenances pour leur rajouter des fonctionnalités, corriger des bogues ou optimiser leur exécution. Cette opération est une tâche très coûteuse [1] pour les développeurs, car ces derniers doivent prendre en compte les parties de codes des autres programmeurs de leur équipe de travail. Par exemple, cela peut s'avérer très coûteux et inutile de trouver une fonction dans une partie de code qui fait exactement le même traitement qu'une autre fonction qui se trouve dans une autre partie du logiciel. C'est pourquoi il doit exister une certaine discipline et rigueur dans le

1. Nous entendons par tâche coûteuse, une tâche qui prend énormément de temps à se terminer. Le coût d'une tâche correspond donc au temps mis pour effectuer la tâche complètement.

développement et la maintenance d'un logiciel pour éviter d'effectuer du travail inutile. Cette discipline et cette rigueur doivent mener à une bonne documentation du logiciel en cours de développement, car c'est par l'absence de cette documentation que le coût de la compréhension d'un logiciel en cours de maintenance est plus important.

La visualisation du logiciel est un domaine d'application très jeune du génie logiciel. Elle a pour but d'afficher le contenu de l'information en utilisant les facultés du système visuel humain pour permettre aux utilisateurs une meilleure compréhension de celle-ci. L'information à visualiser peut représenter un processus, des données, une relation ou un concept. La représentation de cette information se fait via des entités graphiques tels que des points, des lignes, des formes géométriques,... Il arrive que ces éléments, ainsi que leurs attributs (taille, couleurs, position dans l'espace, forme,...), puissent être manipulés pour aider à la compréhension du système en cours de maintenance. La visualisation du logiciel transforme donc les données d'un programme dans une représentation graphique. Elle permet aux développeurs d'analyser les données du logiciel en cours de conception, car elle aide à mieux comprendre les relations et les interactions à l'intérieur du logiciel. Cela permet de résoudre des problèmes de communication que peuvent engendrer les changements d'équipes lors du développement et de la maintenance. Ces surcoûts dus à la compréhension du code peuvent ainsi être évités ou pour le moins diminués.

1.2 Problématique

Pour effectuer des tâches de maintenance et pour mieux connaître son fonctionnement, les développeurs ont recours à l'exécution du programme sur des données représentatives ou des «petites» données. Ce genre de pratique, qui s'avère nécessaire, peut prendre un certain coût dans la durée de la maintenance du logiciel. Beaucoup de développeurs ont alors recours à divers outils tels que les *loggers* ou encore l'affichage des valeurs des variables sur la console pour comprendre le comportement des objets instanciés. Ces développeurs perdent alors énormément de temps vu que plusieurs exécutions du logiciel sont nécessaires pour capter tous les événements du programme. Ceci est principalement dû à un manquement dans leur formation de programmeur où l'utilisation du débogueur ne leur a pas été expliquée.

Ensuite, même si cet outil aide à la correction de bogues en exécutant un programme de manière interactive permettant à l'utilisateur de comprendre le logiciel pas à pas, et de pouvoir consulter les valeurs des variables en temps réel, il ne résout pas les problèmes de la maintenance. Les principaux défauts sont qu'il est gourmand en ressources et qu'il n'est pas facile d'utilisation au premier abord. Ce sont ces points qui rebutent les jeunes développeurs à manier cet outil.

Comme cité dans le cours théorique du professeur Houari Sahraoui [Hou], la visualisation du logiciel est un outil qui est peu, voire jamais utilisé pour la maintenance ou le

développement d'un programme. En effet, les problèmes de maintenance sont des tâches très difficiles à automatiser à cause de leur complexité, du manque d'informations contextuelles, ainsi que par les décisions multiples que les développeurs peuvent être amenés à prendre. La figure 1.1 montre les différentes tâches de maintenance que les programmeurs peuvent rencontrer en fonction d'un problème posé.

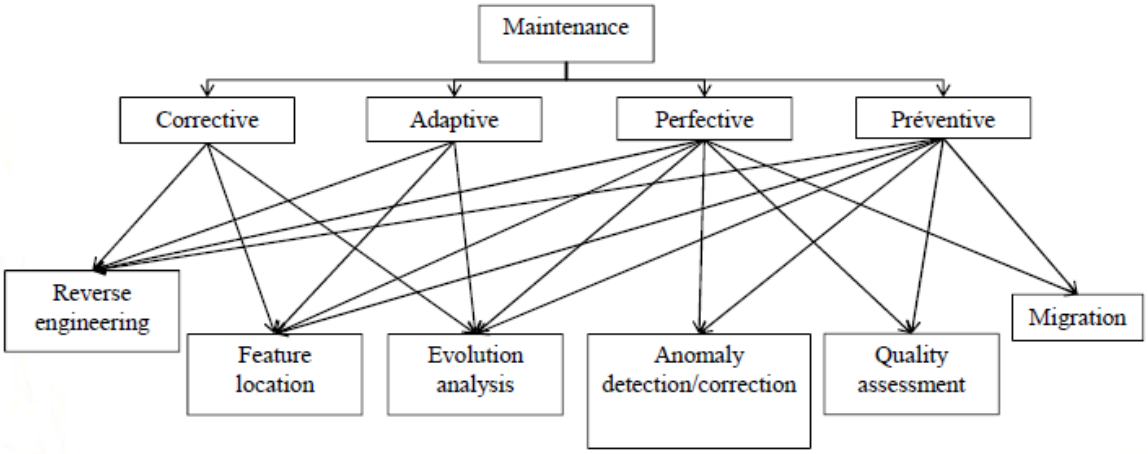

FIGURE 1.1 – Tâches de maintenance avec leurs causes à effets

Certaines tâches de maintenance sont difficiles à automatiser de manière certaine et déterministe. Alors, la solution courante est de faire des approximations pour pouvoir effectuer l'automatisation. Ces approximations peuvent produire des résultats incomplets, parfois inexacts. C'est pourquoi nous décomposons ces tâches en modules (ou sous-tâches). Certains de ces modules peuvent être implémentés de manière précise, donc nous pouvons les automatiser. Mais pour certains, le raisonnement et le calcul s'avèrent nécessaires, et la visualisation peut être vue comme l'interface entre ces modules et l'analyste humain. Mais la visualisation est peu utilisée, car elle n'est pas spécifique aux tâches, demande beaucoup d'efforts et son efficacité d'utilisation est très complexe suivant la tâche demandée. Actuellement, la modélisation de la maintenance par la visualisation consiste à traduire des données du programme (paquetages, classes, méthodes,...) en entités graphiques en vue d'améliorer le coût de la tâche. Les outils de visualisation font une étude du logiciel, c'est-à-dire une analyse du contenu du logiciel à travers le code source, et c'est à partir de cette étude que la modification peut s'effectuer. Donc elle fournit une analyse pour la maintenance, mais ne sert pas aux développeurs pour gérer les modifications à apporter au logiciel en tant que tel.

1.3 Proposition

Pour pallier le premier problème, à savoir le refus de beaucoup de concepteurs d'utiliser le débogueur lors du développement et la maintenance d'un logiciel, il convient de rendre cet

outil beaucoup plus attractif et facile d'utilisation. Pour ce faire, nous proposons d'utiliser des techniques de visualisation sur le débogueur. Cela permet de résoudre par la même occasion le problème qui concernait l'utilisation des techniques de visualisation pour aider les développeurs à effectuer la mise à jour de leurs programmes. Le développement d'un outil visuel communiquant avec un débogueur est donc nécessaire pour étudier la valeur ajoutée qu'il peut apporter dans le domaine du génie logiciel. Nous allons donc utiliser l'outil *VERSO*[2] pour le mettre en relation avec le débogueur de l'environnement de développement intégré Eclipse.

1.4 Structure du mémoire

Ce document est structuré en six chapitres bien distincts. Nous avons déjà introduit le thème de ce mémoire dans le premier chapitre. Nous parlerons au chapitre deux de l'état de l'art : nous nous consacrerons à l'étude de l'existant dans le domaine de la visualisation du logiciel ainsi que l'utilisation du débogueur. Au chapitre trois, nous expliquerons l'approche de notre solution en décrivant les différents outils utilisés et comment nous allons essayer de les faire interagir ensemble. Dans le quatrième chapitre, nous développerons notre solution en expliquant les divers choix d'implémentation de l'outil final. Dans le chapitre cinq, nous allons évaluer notre outil fraîchement développé sur une étude de cas en faisant une expérimentation concrète pour savoir si notre développement s'est avéré utile. Nous analyserons par la même occasion les résultats obtenus avec les commentaires des sujets qui ont participé à l'évaluation de l'outil. Enfin, au chapitre six, nous terminerons par une conclusion qui résumera la totalité des points que nous aurons abordés tout au long de ce document et en précisant quelques pistes futures pour le développement.

2. Un présentation de l'outil est décrite dans la section 3.2.1

Chapitre 2

État de l'art

Avant de commencer l'approche concernant la proposition de notre solution et de son développement, nous nous attarderons sur l'existant dans la domaine de la visualisation du logiciel ainsi que dans le changement de vue du débogage et plus généralement sur les vues dans un éditeur de développement intégré. C'est pourquoi nous allons étudier les différentes possibilités de représenter les vues dans un environnement de développement, ainsi que les outils de visualisation qui apportent une aide pour la maintenance et le développement du logiciel.

2.1 Quelques approches d'amélioration de l'interface d'Eclipse

De nombreux articles traitent du changement de l'environnement de travail pour essayer d'apporter une meilleure disposition de l'information à l'écran. C'est pourquoi nous avons parcouru deux articles qui peuvent apporter une meilleure lisibilité et agencement du code pour permettre une optimisation pour le développement et la maintenance d'un logiciel. La relation qui existe avec notre solution est que notre outil essaie de changer radicalement l'interface de développement via des figures graphiques. Via ces représentations, une multitude d'informations sont disponibles en un rien de temps. Le premier article que nous allons aborder va montrer les utilités d'un tel changement, tandis que le second va exprimer les avantages en modifiant l'espace de travail d'un développeur.

2.1.1 Code Canvas

Robert DeLine et Kael Rowann pensent aujourd'hui que les éditeurs de développement intégré sont comme des ≪bento box≫[DR10], c'est-à-dire que l'écran est partitionné en zones rectangulaires comprenant les éditeurs, les navigateurs, les outils de sorties,... Ces zones sont appelées `canvas`.

Cette ≪bento box≫ qui a prouvé son utilité depuis des années montre tout de même

quelques faiblesses qui sont :

1. les liens entre les vues dues au fait de devoir passer de l'une à l'autre pour avoir accès aux informations d'un projet,
2. les liens entre les perspectives où chacune est réservée pour un travail bien spécifique (exemple : débogage, versionning, développement web,...),
3. l'utilisation de plusieurs moniteurs rend difficile la navigation entre les fenêtres de l'environnement de développement intégré.

C'est pourquoi ils vont remplacer cette «bento box» par une simple surface zoomable qu'ils appelleront ***Code Canvas*** qui sera le centre de tous les documents, projets et codes source. L'outil aide les utilisateurs à s'orienter vu qu'il leur permet de former et d'exploiter l'espace de la mémoire pour trouver des objets. Il sert également de visualisation pour les différentes couches d'informations que sont les projets, les résultats de recherches,...
La figure 2.1 montre l'environnement de travail que propose Code Canvas dans Visual Studio.

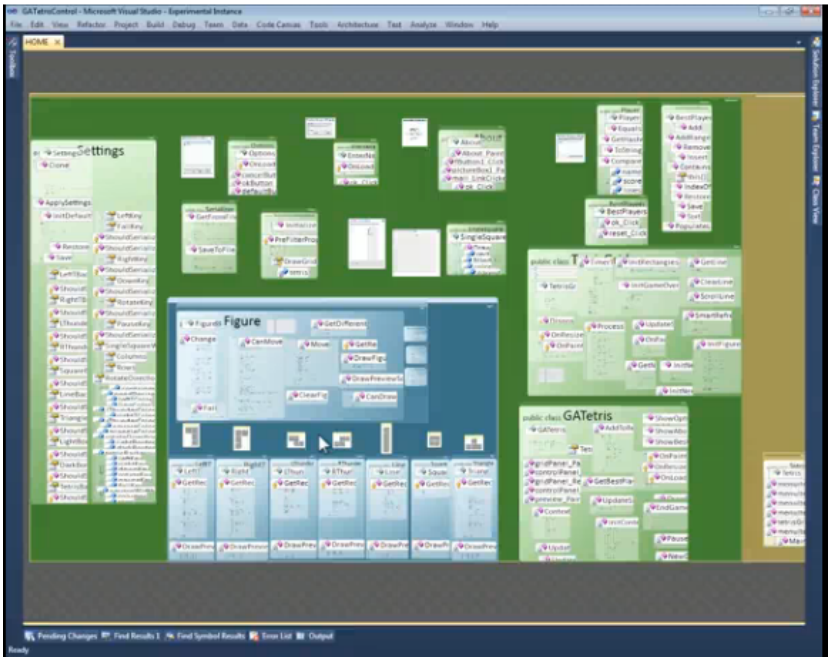

FIGURE 2.1 – Environnement de ***Code Canvas*** dans l'éditeur Visual Studio

Le zoom sémantique

Code Canvas utilise une technique de zoom sémantique qui permet d'afficher différents degrés de détails. Quand un utilisateur va zoomer sur une fonction, il examinera le code comme nous pouvons le voir dans un simple éditeur de texte. S'il dézoome, le code sera de moins en moins lisible et des étiquettes décrivant la fonction vont apparaître, comprenant le nom, les types et les membres de cette fonction. Ce texte sera toujours lisible quel que soit le niveau de granularité choisie. Il existe bien évidemment un niveau de priorité entre ces étiquettes. Les méthodes publiques sont prioritaires par rapport aux privées et il en va

de même pour les attributs. Si l'utilisateur effectue un dézoom jusqu'au maximum, ***Code Canvas*** montre la structure du projet sous forme d'un diagramme de classe UML.

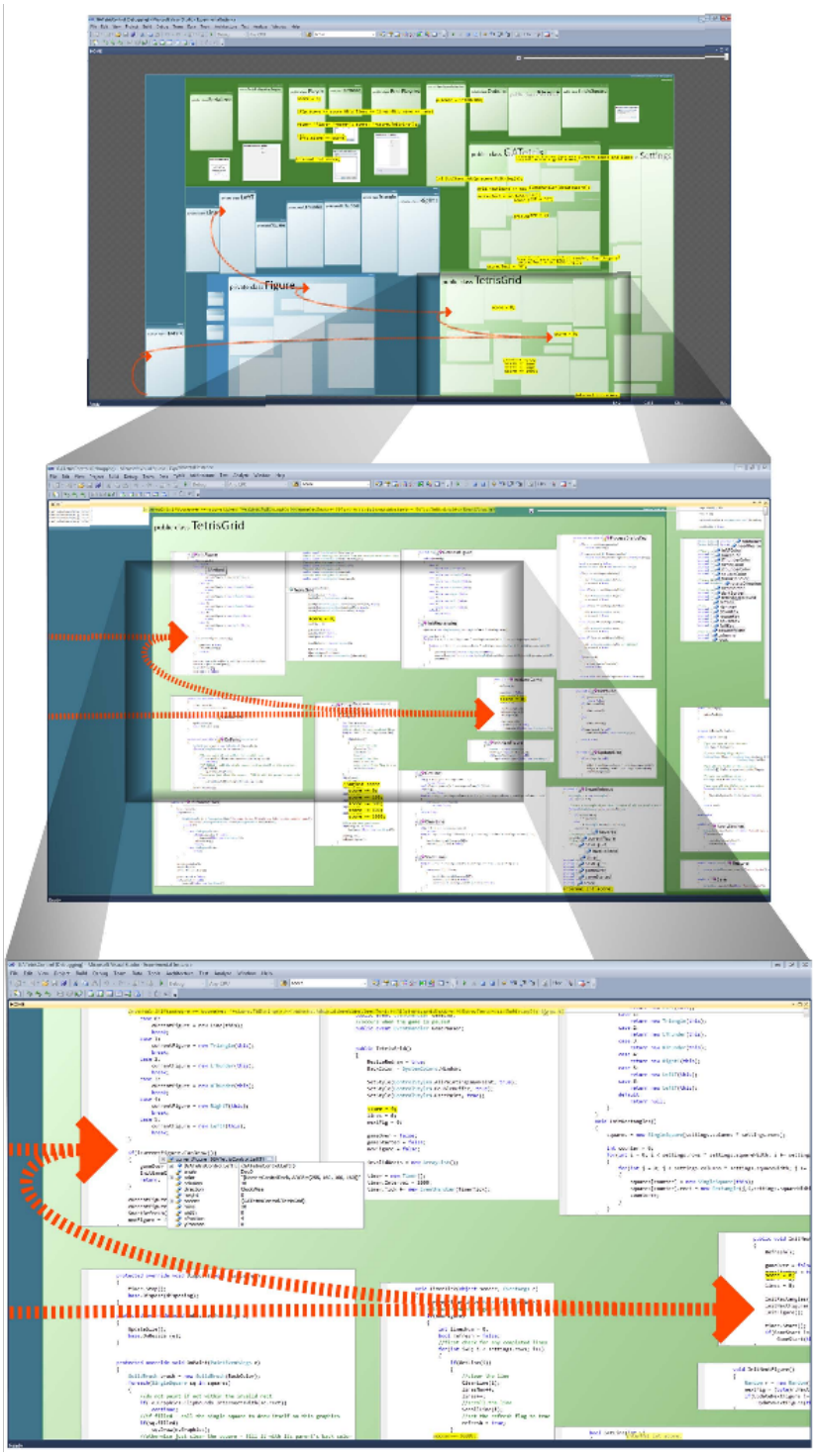

FIGURE 2.2 – Exemple d'un zoom sémantique

La figure 2.2 montre un exemple de zoom sémantique lors du débogage. En zoomant sur

l'étiquette qui se nomme `TetrisGrid`, nous arrivons dans la deuxième partie de la figure. Et en zoomant encore sur une partie du code, nous obtenons la dernière image de notre figure. La présence des flèches indique le chemin d'exécution d'un processus qui s'est immobilisé lors de la rencontre d'un point d'arrêt. Il s'agit de la pile d'exécution du processus qui est en cours de débogage par l'utilisateur. La figure 2.3 montre la structure du système qu'il est possible de générer en dézoomant au maximum. Elle est représentée sous forme d'un diagramme de classe UML.

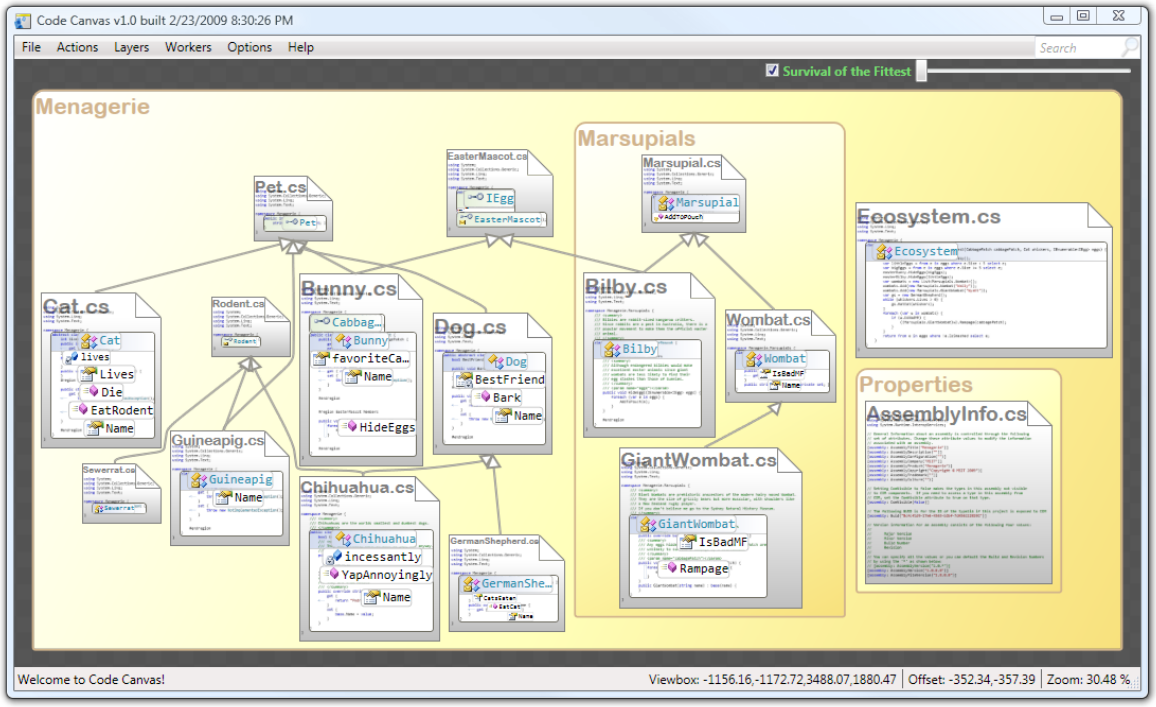

FIGURE 2.3 – Structure d'un système sous ***Code Canvas***

La présentation du code

L'utilisateur peut, à la demande, modifier la présentation du code de trois façons :

1. tous les objets (fichiers, dossiers, fonctions,...) peuvent être gérés par un mécanisme de déplacement où l'utilisateur peut leur donner une position. De plus, si le développeur ajoute des lignes de code dans une fonction, l'espace alloué à la fonction sera automatiquement agrandi en repoussant les fonctions voisines pour permettre au texte de s'organiser correctement,
2. en introduisant un nouveau conteneur pour représenter des concepts qui ne sont pas synthétiquement explicites dans le code. Par exemple : l'enregistrement de nouvelles informations dans une base de données et leur journalisation,
3. en «enlevant» une méthode de sa classe pour la diviser en plusieurs sous-parties.

L'utilisateur peut créer plusieurs `canvas` simultanément, ce qui apporte plusieurs avantages. Tout d'abord, chacun a sa propre vue, son propre niveau de zoom, son propre ensemble de présentation,... ce qui permet de distinguer tous les processus. De plus, l'utilisateur peut ainsi effectuer plusieurs tâches simultanées dans deux endroits distants du `canvas` global, sans devoir naviguer ou zoomer d'un `canvas` à l'autre. Ensuite la gestion du multi-tâches est rendue plus facile par la création d'un `canvas` par tâche. Cette création préserve non seulement la navigation entre ces `canvas`, mais également de la pollution massive d'informations affichées à l'écran. L'utilisateur peut aussi créer des copies de `canvas` pour sauvegarder les états d'un projet et ajouter des filtres pour en sortir des sous-ensembles de `canvas`.

Code Canvas apporte une nouvelle visualisation pour travailler avec le code et permet à l'utilisateur de s'orienter de manière hiérarchique à travers lui. Une démonstration concernant toutes les fonctions qui viennent d'être citées et avec quelques compléments se trouve sur YouTube [1].

2.1.2 Code Bubbles

Tout comme pour les auteurs de ***Code Canvas***, Andrew Bragdon et ses collègues proposent un nouvel environnement de travail qu'ils vont appeler ***Code Bubbles*** [BZR+10] [BRZ+10]. L'interface proposée est basée sur des collections de petits fragments éditables appelés `bubbles`. Les auteurs de cet outil décrivent la conception d'un prototype pour l'interface utilisateur d'un environnement de développement intégré basé sur des projets Java. Leur but est de supprimer la navigation fastidieuse d'une classe vers une autre pour pouvoir visionner le fil de l'exécution d'un programme à travers les méthodes qui sont appelées de manière hiérarchique. Ils soumettent une nouvelle approche où les vues d'un éditeur de codes, tel qu'Eclipse ou Netbeans, deviennent de multiples fragments que nous pouvons éditer simultanément.

La figure 2.4 montre un exemple de ce à quoi ressemble l'environnement de travail de ***Code Bubbles***. Nous allons décrire chaque fonction relative aux lettres qui se trouvent sur l'image.

- **A** : Barre d'espace de travail divisée en plusieurs compartiments.
- **B** : Espace de travail sélectionné.
- **C** : `Web bubble` qui consiste à fournir un accès à une base de connaissances distante. Dans l'exemple, il s'agit d'accéder à un bogue en vue de le corriger.
- **D** : Notes de travail.
- **E** : Formation d'un groupe représenté par la couleur qui entoure chaque `bubble`. Chaque groupe possédant plusieurs `bubbles` peut être assigné à un nom.

1. `http://www.youtube.com/watch?v=tsFfyli2Y9s`

- **F** : Résultat d'une recherche de toutes les références de la variable sélectionnée dans la `bubble` précédente.
- **G** : En cliquant sur une référence dans la `bubble`, le corps de la méthode référencée s'affiche.
- **H** : Navigateur de paquetages et de classes.
- **I** : `Bubble` contenant la *Javadoc* d'une méthode sélectionnée.
- **J** : `Flag bubble` qui consiste à indiquer à l'utilisateur ce qu'il doit faire. Dans l'exemple courant, il s'agit d'un rappel pour déboguer le groupe de `bubbles` sur lequel le `Flag bubble` a été posé.
- **K** : Méthode encapsulée dans une `bubble`.
- **L** : Affichage du corps de la méthode sélectionnée dans la `bubble` précédente.
- **M** : Représentation d'une relation d'appel entre deux `bubbles`.
- **N** : Fonction de recherche dans le code source avec auto-complétion dans le résultat.
- **O** : Résultat de la fonction de recherche.
- **P** : Affichage via une bulle d'information du détail du résultat à l'endroit où le curseur s'est arrêté.
- **Q** : Environnement où sont placées les `bubbles`.

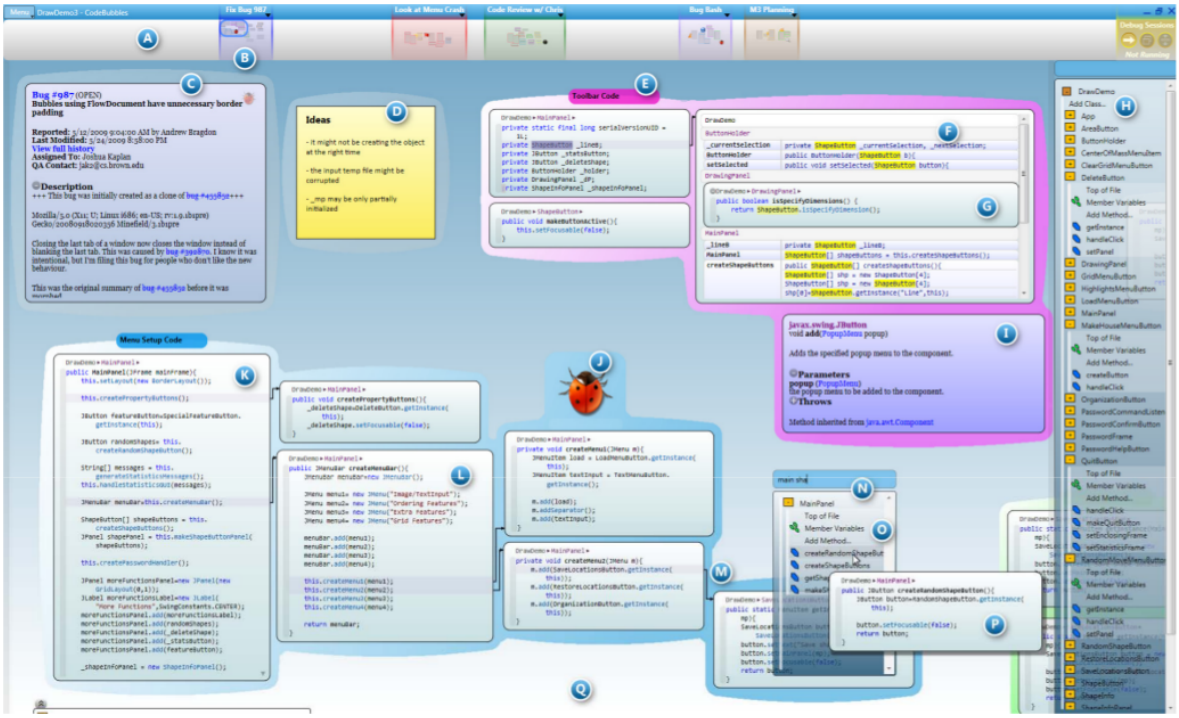

FIGURE 2.4 – Environnement de travail de ***Code Bubbles***

En créant l'IDE ***Code Bubbles***, les auteurs résolvent quatre problèmes critiques liés à l'affichage des fenêtres que posent les autres éditeurs tels qu'Eclipse ou Visual Studio. Ces problèmes sont :

- le code ne s'adapte pas automatiquement à la taille des fenêtres,
- les vues qui se chevauchent requièrent une interaction manuelle,
- la décoration des fenêtres est distrayante et perd de l'espace dans l'environnement de travail,
- éventuellement, l'utilisateur peut être amené à manquer d'espace selon la série de tâches qu'il accomplit.

Pour les résoudre, les développeurs de ***Code Bubbles*** ont décidé que la taille des `bubbles` se redimensionnait automatiquement en fonction du code. Ensuite, les `bubbles` ne se chevauchent pas l'une sur l'autre pour une meilleure disposition de l'information sur l'écran. De plus, elles n'ont pas d'ascenseurs ni de barres d'outils pour éviter une pollution d'affichage dans la `bubble`. Enfin, l'utilisateur peut ouvrir une même méthode dans plusieurs `bubbles` différentes et éditer une `bubble`. La modification va se répercuter dans toutes les `bubbles` contenant cette fonction.

Le débogage

Les traditionnels débogueurs fournissent les états d'un programme à un point donné. Bien que les programmeurs ont le besoin de connaître l'état d'un projet pour le comparer avec un autre, ils voudraient aussi annoter le programme pour partager les informations et ainsi effectuer un gain de temps.

Tout comme Eclipse ou NetBeans, ***Code Bubbles*** gère les points d'arrêt à la gauche du code ainsi que les fonctions de `stepping`[2]. Lorsque le logiciel rencontre un point d'arrêt, la vue de ***Code Bubbles*** change radicalement vers une vue de débogage tel qu'elle est représentée par la figure 2.5.

2. Ceci sera largement expliqué dans la section 3.1.1

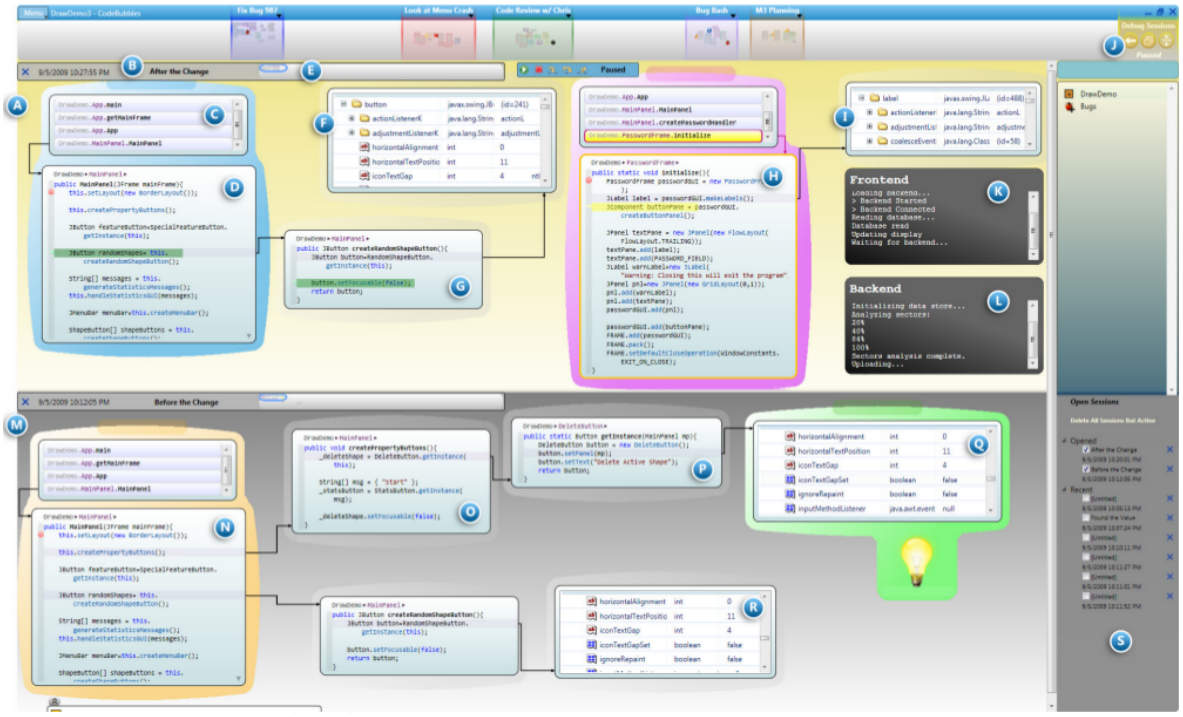

FIGURE 2.5 – Environnement de débogage de ***Code Bubbles***

Nous allons également expliquer les concepts qui se trouvent sur l'image présente.

- **A** : Instances du programme qui sont rassemblées en un seul espace de travail appelé *Channel*.
- **B** : Barre de titres qui incluent les dates de modifications et les titres optionnels.
- **C** : `Bubble` affichant la pile d'exécution du processus.
- **D** : `Bubble` contenant le code où le programme s'est interrompu au point d'arrêt.
- **E** : Miniature de chaque environnement de travail. Chaque groupe possédant plusieurs `bubbles` peut être assigné à un nom.
- **F** : `Bubble` contenant la structure d'une donnée ne faisant pas partie de l'exécution du processus.
- **G** : En entrant dans la méthode courante, une nouvelle `bubble` se crée avec le corps de la fonction.
- **H** : Un nouveau groupe est créé si le logiciel rencontre un nouveau point d'arrêt.
- **I** : En faisant un clic droit avec le curseur sur une variable (en **H**), une nouvelle `bubble` apparait et contient la structure de l'objet sélectionné.
- **J** : Étiquette qui indique dans quel environnement de travail l'utilisateur se situe. Dans l'exemple présent, il s'agit de l'environnement de débogage.
- **K - L** : Définition de console personnalisée encapsulée dans des `bubbles`.
- **M** : Premier espace de débogage ou ancienne vue de débogage en cas de plusieurs instances. Il s'agit d'un historique des exécutions précédentes pour permettre des com-

paraisons.
- **N** : `Bubble` où le débogage du programme débute.
- **O** : Une nouvelle `Bubble` est créée si nous entrons dans la méthode (similaire à **G**).
- **P** : Même fonction que **O**.
- **Q** : `Bubble` contenant la structure de la variable en cours de débogage.
- **R** : Même fonction que **Q**.
- **S** : Une session de débogage peut être sauvée et chargée à partir de cette interface.

Observations

En analysant le comportement de ***Code Bubbles*** sur de réelles applications en mettant en avant la lisibilité du code de manière simultanée, les auteurs ont pu démontrer que l'outil est capable d'afficher à l'écran une grande quantité d'informations de manière synchrone dans la plupart des cas. En d'autres termes, les développeurs sont capables de voir plus de méthodes en utilisant ***Code Bubbles*** par rapport à un autre éditeur. De plus, ils ont prouvé que le nombre d'interactions sur l'interface de l'IDE pour afficher des informations a grandement diminué[3]. Ceci apporte un gain de temps réel pour le développement et la maintenance de logiciels.

Ensuite, l'équipe de développement de l'IDE a proposé à des développeurs professionnels d'évaluer leur outil. Les auteurs leur ont demandé des critiques constructives sur son utilisation par rapport aux tâches qui leur ont été exigées. Ils ont dû comparer différentes parties de codes, d'en comprendre le fonctionnement, de corriger certains bogues avec l'outil,... Dans l'ensemble, les participants ont émis une note positive de l'outil en le trouvant très pratique. Ils ont repéré quelques faiblesses comme l'absence de support de fichier XML, ce qui est impensable aujourd'hui.

Les limitations

L'utilisation de cet IDE n'a pas que des avantages. Il demande une grosse configuration au niveau matériel. En effet, il requiert un processeur double-cœur, une carte graphique qui accepte l'accélération matérielle et un écran 24 pouces pour fonctionner correctement et dans un confort d'utilisation. Il est uniquement compatible avec le langage Java, XML et HTML et il n'est pas aussi sophistiqué que les autres éditeurs tel qu'Eclipse.

Un autre problème à souligner est que ***Code Bubbles*** utilise des signatures de noms de méthodes pour travailler. Dans un véritable environnement de développement, ceci ne peut fonctionner correctement étant donné que nous renommons, modifions et supprimons des fonctions au cours du temps.

Au niveau de l'interface de débogage, il est en cours d'optimisation, car il ne gère que les petits problèmes où les erreurs se produisent uniquement dans les mêmes branches d'un arbre

3. Le tableau des observations se trouve à la page 7 de l'article [BRZ+10]

d'appels.

Dans l'ensemble ***Code Bubbles*** est un outil qui a un gros potentiel et pourra être utilisé sur une plus grande échelle dans un avenir proche. Une démonstration est disponible sur leur site internet [4].

Nous allons maintenant examiner divers documents qui traitent de la qualité du logiciel en utilisant des techniques de visualisation.

2.2 Quelques outils de visualisation

Il existe de nombreux outils qui sont capables d'aider les développeurs à mettre à jour ou développer leur logiciel tout en améliorant et optimisant le code source. Parmi eux, il y a ceux qui utilisent des techniques de visualisation. Elle a pour but de traiter des métriques calculées au préalable et de les représenter sous forme graphique, ce qui permet une meilleure lisibilité des résultats et rend la comparaison avec d'autres métriques plus facile. C'est pourquoi les utilisateurs vont se servir de ce type d'outil pour mesurer la qualité de leur programme. Ceci leur permettra de corriger le code source de leur projet par rapport aux résultats obtenus. La version actuelle de notre outil fait partie de cette catégorie, mais nous comptons lui ajouter une composante pour améliorer le coût de la maintenance des systèmes informatiques.

Nous verrons plusieurs articles et un mémoire traitant de la qualité du logiciel et de l'aide que peut apporter ce type d'outil aux développeurs pour la maintenance et le développement de logiciels en perfectionnant le code source des projets.

2.2.1 Visualisation de la qualité

Il existe de nombreux outils de visualisation qui mesurent la qualité d'un logiciel. La plupart transforment les métriques calculées sous forme graphique, mais ces représentations varient du tout au tout. Nous vous en présentons quelque-uns.

CodeCity

Richard Wettel et al. [WL08] ont décidé d'utiliser la métaphore de ville pour décrire un projet. Les classes sont représentées par des buildings résidant dans des districts que sont les paquetages. Pour transformer un projet en une vue graphique, le code source doit être parsé puis modélisé suivant des contraintes et enfin l'outil va effectuer un rendu graphique grâce au calcul des métriques. Les auteurs ont relié chaque métrique du logiciel à un artefact graphique. Le nombre de méthodes est lié à la hauteur du bâtiment, le nombre d'attributs est associé à la taille de la base du bâtiment et la couleur indique le nombre de lignes de code de la classe

4. `http://www.andrewbragdon.com/codebubbles_site.asp`

partant du gris sombre (peu de lignes) vers un bleu intense (beaucoup de lignes). Le niveau d'imbrication des paquetages est également associé à la couleur où un dégradé du bleu clair vers le bleu foncé va être exécuté en partant de la plus haute imbrication vers la plus basse. L'utilisateur peut changer la représentation de chaque métrique par une autre correspondance graphique. La figure 2.6 montre un exemple d'un projet représenté sous CodeCity

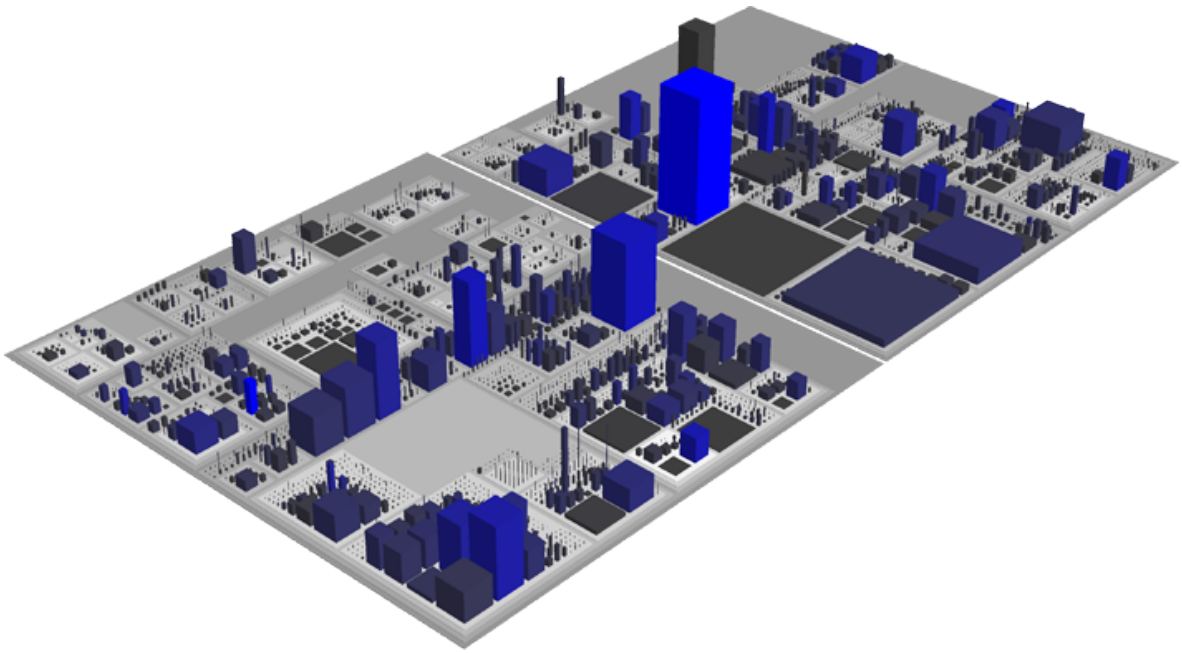

FIGURE 2.6 – CodeCity qui prend les sources du projet Java 1.5 en paramètre

Ensuite, ils ont évalué CodeCity pour essayer de démontrer son efficacité [WLR11] en posant diverses questions relatives à :

- la correction d'un programme,
- la réduction du temps pour comprendre un programme,
- les types de tâches effectuées avec CodeCity par rapport à un système non visuel,
- l'antécédent de l'utilisateur (académique et industriel),
- le niveau d'expérience de l'utilisateur se servant de l'outil.

Ils sont arrivés à la conclusion que leur outil permettait aux utilisateurs de gagner du temps dans la compréhension d'un programme (12% plus rapide) et de corriger plus efficacement (24%) certains problèmes dans les logiciels. Ils ont également remarqué que pour certaines tâches, CodeCity était utilisé de pair avec un tableur de type Excel. Tous les tests et résultats se trouvent dans leur article [WLR11].

VERSO

Dans la même lignée que Codecity, *VERSO* est un outil de visualisation conçu par Guillaume Langelier [LSP05]. Il est basé sur la métaphore du `Treemap` développé par Shnei-

derman [JS] mais qui a été modifié dans le cadre de ce projet. Cet outil a pour but de calculer des métriques et de les représenter sous forme graphique. La figure 2.7 montre l'environnement de travail de *VERSO* en prenant comme paramètre le projet «open source» Azureus [LD07].

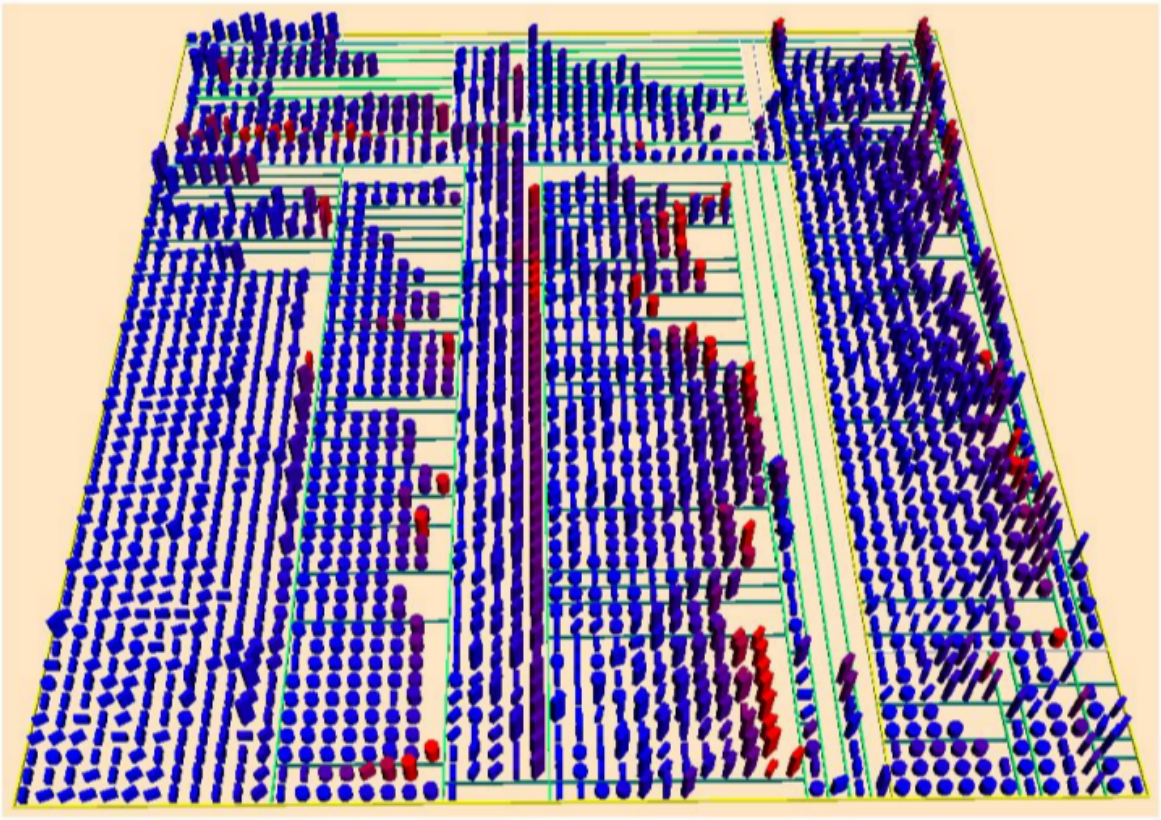

FIGURE 2.7 – Représentation du projet Azureus par *VERSO*

Une plus large description des fonctions de cet outil sera faite dans le chapitre suivant à la section 3.2.1.

2.2.2 Visualisation de traces d'exécution

Il existe également des outils qui utilisent les métaphores de `Treemap` [JS] mais qui, en plus de mettre l'accent sur la structuration d'un système, analysent également le comportement du logiciel grâce aux traces d'exécution [5]. Nous présentons ci-après divers outils qui utilisent ce système pour analyser des programmes.

5. Une trace d'exécution est un fichier qui enregistre les différentes étapes de l'exécution d'un scénario d'un programme.

Evospace

Comme pour *VERSO* Evospace utilise aussi la métaphore de «city» via des `Treemaps` pour représenter un projet, mais il se démarque par le fait que Philippe Dugerdil et Sazzadul Alam - les auteurs de l'outil - ont voulu mettre l'accent sur les liaisons qui peuvent exister entre les classes, comme les appels entre méthodes [DA08]. Ces liaisons sont représentées par des lignes partant d'un bâtiment et allant vers un autre (cfr. figure 2.8).

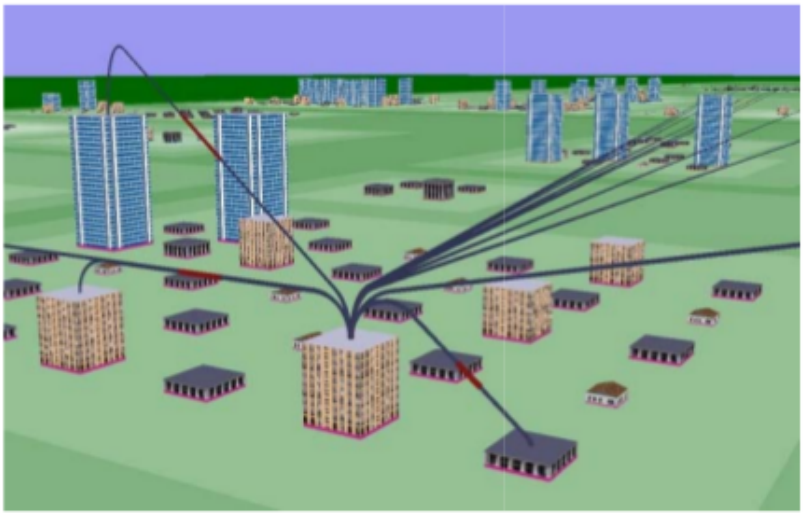

FIGURE 2.8 – Exemple de représentation d'un système dans Evospace

Étant donné qu'une trace d'exécution peut comprendre une grande quantité d'informations, les auteurs ont décidé de la segmenter en différentes parties. C'est pourquoi ils présentent les classes vues de jour lorsqu'il n'y a aucune analyse et de nuit lorsqu'il y en a une. Les classes colorées dans la vue de nuit sont celles qui se trouvent dans le scénario de la trace d'exécution (cfr. figure 2.9).

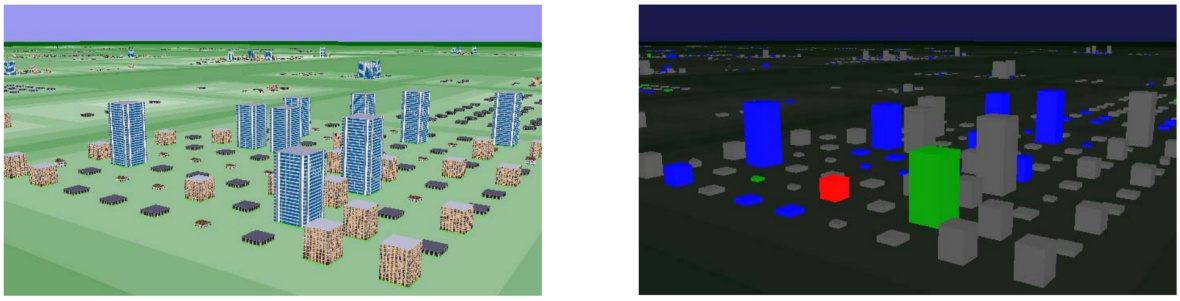

FIGURE 2.9 – Vue jour et nuit dans Evospace

Lorsqu'un utilisateur va commencer l'analyse d'une trace d'exécution, des liaisons vont apparaître sur les bâtiments pour montrer le chemin d'exécution du scénario enregistré. Ceci a un certain avantage de lisibilité au départ mais l'écran peut très vite être pollué par toutes ces lignes.

VASCO

Dans son mémoire, mademoiselle Fleur Duseau essaie d'identifier l'usage excessif d'objets temporaires pour l'exécution d'un programme via une technique de visualisation [Dus11]. En effet ces objets temporaires, communément appelés `object churn` nuisent aux performances d'une application. Leurs déclarations ne font qu'alourdir les ressources allouées au programme, ce qui donne plus de travail pour le ramasse-miettes [6], l'outil qui libère de l'espace en mémoire interne.

Récolte des données Pour étudier ce phénomène, Fleur Duseau a analysé différentes traces d'exécution d'un programme à l'aide d'un outil de visualisation qu'elle a elle-même implémenté.

En effet, une trace d'exécution peut contenir énormément d'informations sur le comportement d'un logiciel. Avec elle, nous pouvons très facilement calculer un arbre d'appels de méthodes. La figure 2.10 montre un arbre d'appels dynamique calculé avec la trace d'exécution. Nous avons la possibilité de fusionner un arbre pour en construire le graphe d'appels. Mais un graphe propose des chemins qui n'existent pas dans l'arbre. C'est pourquoi nous transformons ce graphe avec l'arbre d'appels, ce qui donne un arbre de contexte d'appels [7].

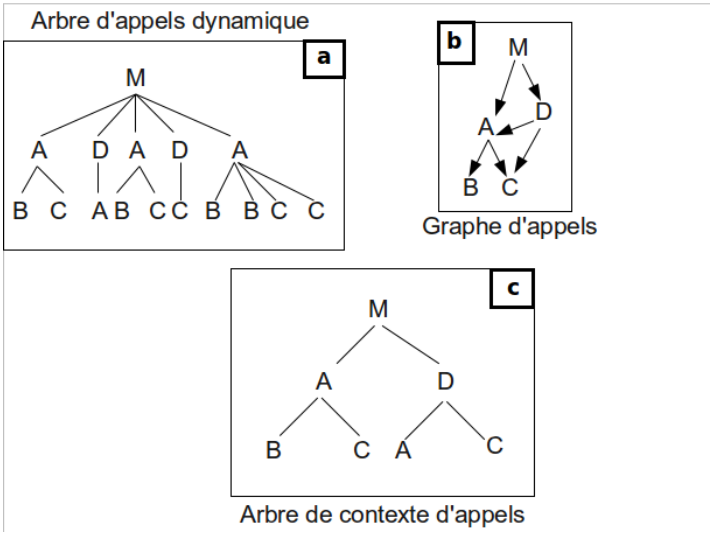

FIGURE 2.10 – Exemple d'un arbre d'appels, d'un graphe et d'un arbre de contexte d'appels

Cet arbre nous permet de tirer des premières analyses, à savoir quelles sont les méthodes les plus appelées dans les différents scénarios explorés. Dans chaque méthode, l'auteur va analyser les objets temporaires pour essayer de les remplacer ou tout simplement les supprimer grâce à l'utilisation de son outil. La figure 2.11 présente les différentes étapes pour récolter les données.

6. Ou `Garbage Collector` en anglais
7. Ou `Calling-Context Tree` (CTT) en anglais

Pour commencer, la trace dynamique de l'exécution est collectée par l'outil Jinsight [8], puis ce même outil construit un CCT à partir de la trace d'exécution. Elude [9] effectue l'analyse d'échappement sur l'arbre pour associer un graphe de connexions à chaque nœud du CCT. Enfin, l'outil Churni [10] fusionne le fichier en sortie d'Elude avec les informations d'allocations récupérées par Jinsight et produit un CCT avec des graphes de connexions réduits.

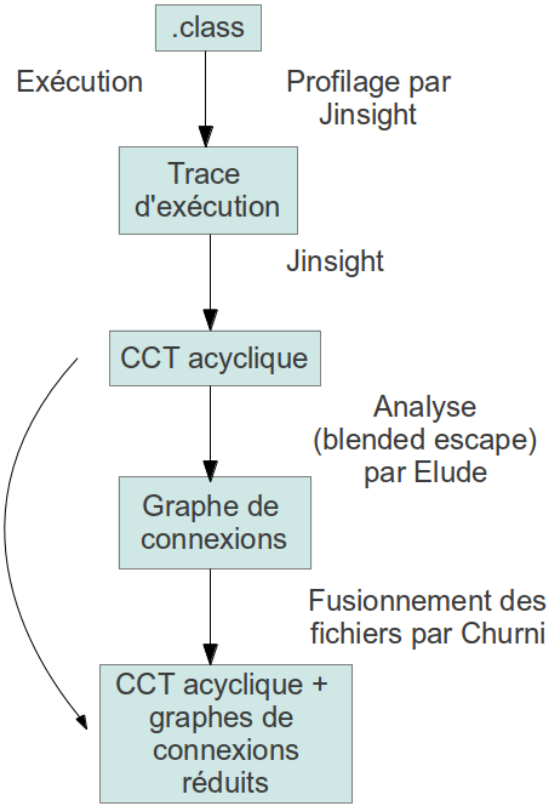

FIGURE 2.11 – Approche de transformation en arbre de contexte d'appels acycliques

Pour représenter ces données visuellement, Fleur Duseau a décidé d'utiliser une technique de Sunburst [11]. Cette technique a pour but de transformer un arbre d'appels sous une forme de cercles concentriques où la racine de l'arbre est placée au centre et les descendants sont disposés autour du centre. La figure 2.12 représente un arbre d'appels transformé en Sunburst.

8. Programme d'analyse d'exécution de programme. `http://www.research.ibm.com/jinsight/docs/index.htm`

9. Outil d'analyse, développé par Bruno Dufour, professeur à l'Université de Montréal, qui peut identifier des objets potentiellement temporaires dans une exécution de programme.

10. Outil non référencé par Fleur Duseau, qui a pour but de fusionner le CCT acyclique et le graphe de connexions.

11. Métaphore développée par J. Stako et E. Zhang dans leur article ≪*Information Visualization*≫ en 2000 à la conférence *InfoVis 2000*

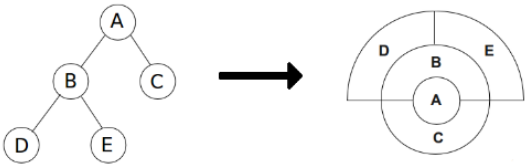

FIGURE 2.12 – Transformation d'un arbre d'appels en Sunburst

Analyse Fleur Duseau visualise des méthodes via le CCT fraichement généré à l'aide de son outil *VASCO*. Elle a défini le degré de l'angle d'une méthode dans le Sunburst par son importance par rapport à son parent. Plus l'angle est grand, plus la méthode est importante et inversement.

L'outil permet de visualiser deux métriques simultanément à l'aide de la couleur et de la longueur de l'arc. La couleur représente une valeur pour la méthode et l'angle décrit l'exécution d'une méthode. Il est possible de représenter d'autres métriques, ce qui changera la couleur et la longueur des arcs dans le Sunburst. La figure 2.13 constitue une illustration de l'utilisation de l'outil. Elle est représentée comme suit :

- **1** : l'espace réservé à la visualisation d'un projet,
- **2** : l'historique des visualisations effectuées,
- **3** : le panel indiquant la valeur des métriques,
- **4** : la barre d'outils qui permet de changer de métrique et de représentation.

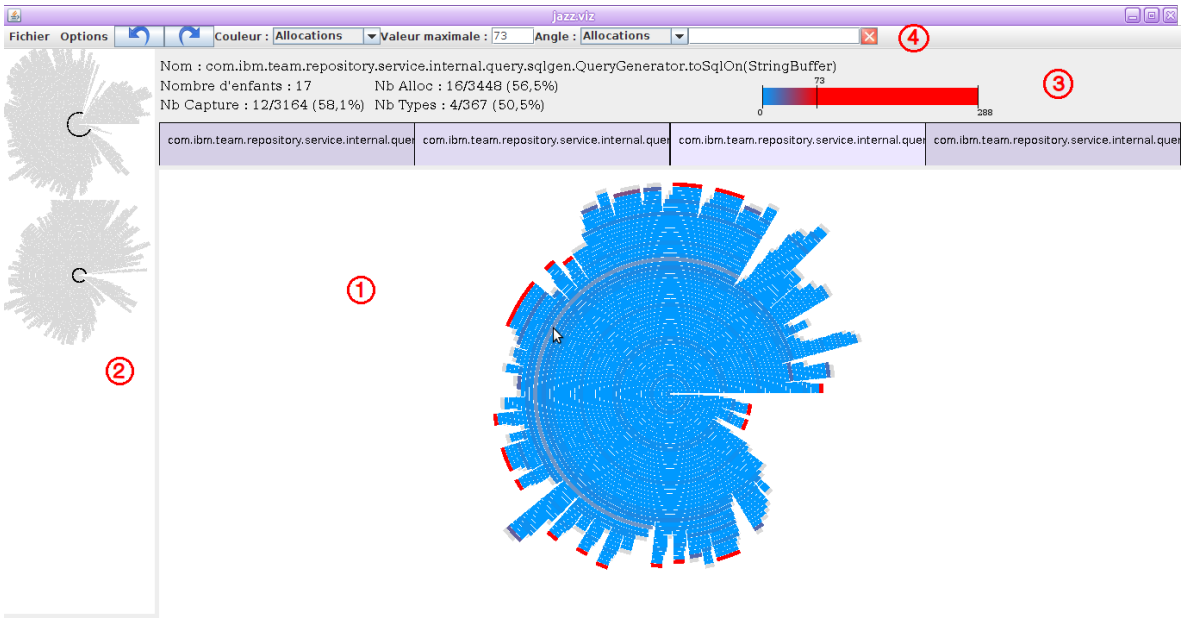

FIGURE 2.13 – L'outil VASCO

Pour détecter la présence d'objets temporaires, Fleur Duseau doit effectuer deux étapes de manière itérative. La première est d'étudier l'évaluation courante et la seconde est d'éliminer

les régions déjà étudiées. L'utilisateur peut dès lors filtrer des données pour avancer plus rapidement dans la tâche qu'il souhaite accomplir. Il peut par exemple changer la racine du Sunburst pour une meilleure visualisation des résultats calculés par VASCO. Ceci est représenté à la figure 2.14 où la méthode montrée par la flèche devient le centre du cercle. Il est possible d'effectuer d'autres opérations, mais celles-ci ne seront pas abordées dans ce document.

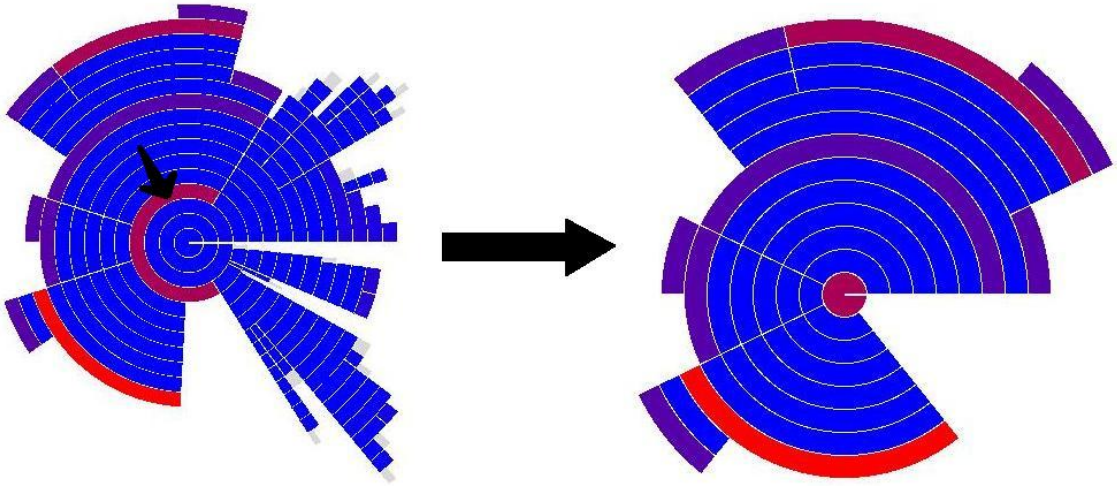

FIGURE 2.14 – Exemple de changement de racine

Pour terminer, Fleur Duseau a prouvé l'efficacité de son outil en permettant de représenter les données dans une vue unique. Les informations nécessaires à la recherche des objets temporaires à éliminer sont calculées à partir de ces données. Ceci garantit un moindre effort vu que la tâche à accomplir est rendue plus facile et les interactions avec l'outil sont sauvegardées au cours du temps, ce qui permet à l'utilisateur de revenir plusieurs étapes en arrière pour repartir vers une autre région.

Chapitre 3

Proposition d'un outil d'aide à la maintenance et au développement de logiciels

Depuis quelques années, les concepteurs ont acquis certains réflexes pour développer et effectuer la mise à jour de leurs logiciels. Certaines de ces habitudes peuvent être bonnes comme elles peuvent être mauvaises. Par exemple, le fait d'utiliser un débogueur pour trouver les anomalies d'un programme et ajouter de la documentation permet une plus grande facilité de compréhension d'un logiciel. Par contre, l'ajout de *logger* ou de *print* pour corriger des erreurs ne font qu'augmenter la taille d'un programme et son exécution sera plus lente.

Le fait de permettre à un outil de visualisation d'aider les programmeurs à la maintenance et au développement de logiciels serait un grand avantage. En effet, les développeurs ont tendance à utiliser deux outils lors de la conception d'un projet. Le premier est l'éditeur de code, généralement Eclipse ou NetBeans, le second étant un logiciel de calculs de métriques pour mesurer la qualité du logiciel en cours de développement. Le fait de rassembler un logiciel de visualisation et un éditeur de projets en un seul outil, permettrait de récolter un maximum d'informations en un seul endroit.

Mais avant de proposer une solution qui réglerait les deux problèmes cités dans l'introduction, intéressons-nous d'abord au fonctionnement du débogueur d'Eclipse et de l'outil de visualisation *VERSO* pour mieux les comprendre.

3.1 L'outil de débogage Eclipse

Lors du développement de tout logiciel, il existe un outil très utile pour aider à la maintenance et à la correction d'erreurs : le débogueur. Il en existe généralement un dans chaque éditeur de développement intégré tels qu'Eclipse ou NetBeans. Si ce n'est pas le cas, il est

toujours possible d'en installer un pour le langage de programmation que l'on utilise. Dans le cadre de notre mémoire, nous nous intéresserons uniquement à celui présent dans Eclipse, mais il faut savoir que la description de cet outil est valable pour tous les débogueurs quel que soit l'éditeur de développement utilisé pour écrire les lignes de code.

3.1.1 Présentation de l'outil

Un débogueur est un outil qui permet d'exécuter des programmes de manière interactive tout en parcourant le code source d'une classe à travers ses variables [DOU07]. Lorsqu'un utilisateur désire déboguer son programme, Eclipse lui propose de passer à la perspective *Debug*. Cette perspective contient ses propres vues qui sont :

- la vue *Debug*
- la vue *Variables*
- la vue *Breakpoints*
- la vue *Expressions*
- la vue *Display*

Dans le cadre du mémoire, nous ne décrirons que les trois premières (*Debug*, *Variables* et *Breakpoints*).

La vue *Debug*

C'est par cette vue que l'exécution d'un programme peut se faire de manière interactive. Elle est en quelque sorte le point central des notifications envoyées par l'utilisateur.

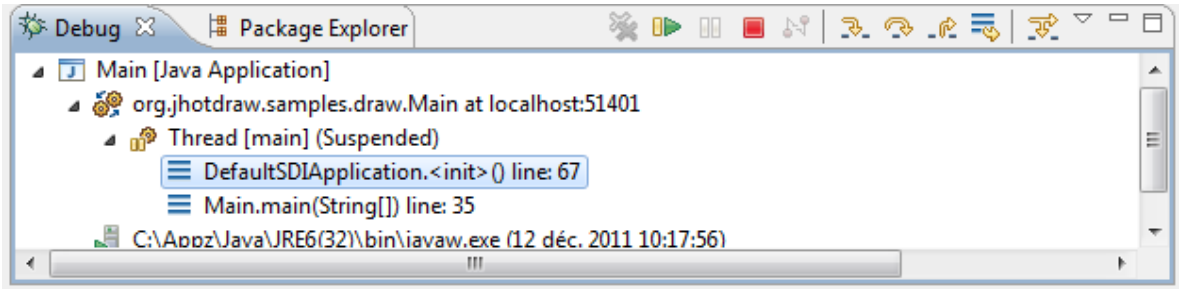

FIGURE 3.1 – Représentation de la vue Debug dans Eclipse

La figure 3.1 qui représente la vue *Debug*, affiche les différents processus d'un programme en cours d'exécution. Dans cet exemple, nous pouvons voir qu'il s'agit de la fonction Main qui est en cours de débogage. Les différentes icônes représentent un état ou un objet bien défini au débogage. Voici les plus importantes à retenir :

 Cette icône représente une tâche [1] en cours d'exécution.

 Cette icône représente un *thread* en suspens, car l'utilisateur est en train de le déboguer.

 Cette icône représente une `stackframe` [2] d'un processus.

C'est à partir de cette fenêtre que nous pouvons contrôler interactivement un logiciel grâce aux divers boutons qui sont proposés. En voici les plus importants :

 Le bouton RESUME permet de relancer le programme qui s'est interrompu à un point d'arrêt [3]. Le logiciel s'exécutera alors de façon normale jusqu'à ce que :
- soit il s'arrête normalement,
- soit il rencontre un autre point d'arrêt,
- soit une exception non capturée est levée jusqu'au somment de la pile d'exécution.

 Ce bouton arrête tout simplement l'exécution du programme qui est en cours d'exécution.

 Le bouton STEP INTO permet d'entrer dans le corps d'une méthode ou d'exécuter la ligne courante pour aller ensuite vers la suivante. C'est l'opération la plus couramment utilisée pour déboguer un programme.

 Le bouton STEP OVER exécute la ligne de code courante sans entrer dans le corps de la méthode pour directement passer à la ligne suivante.

 Le bouton STEP RETURN exécute le corps de la fonction courante et la quitte pour revenir au niveau supérieur.

 Ce bouton permet de suspendre l'exécution normale d'une tâche en cours d'exécution. Il ne peut être utilisé que si l'utilisateur a préalablement appuyé sur le bouton RESUME.

1. Ou `thread` en anglais. Il faut savoir que chaque `thread` contient une pile d'exécution ou `stack` en anglais
2. Une `stackframe` représente une partie de la pile d'exécution. Chaque `stackframe` compose la `stack` d'un `thread`
3. Un point d'arrêt ou `breakpoint` en anglais est un mécanisme qui permet d'arrêter un programme en cours d'exécution si ce dernier a été lancé en mode *debug*

La vue *Variables*

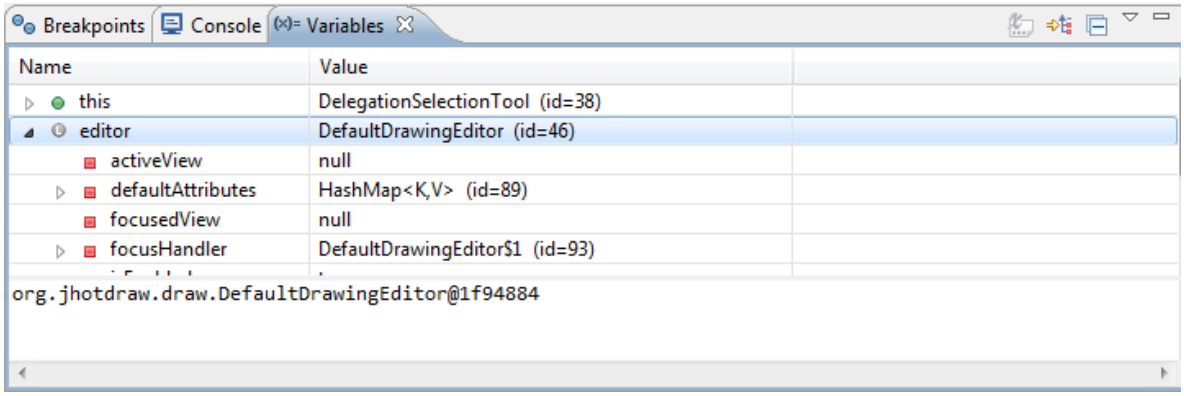

FIGURE 3.2 – Représentation de la vue *Variables*

La figure 3.2 qui montre la vue *Variables*, affiche les variables de la `stackframe` sélectionnée depuis la vue *Debug*. Elles sont les instances des objets qui sont accessibles depuis la méthode courante. Nous pouvons y voir leurs valeurs en temps réel qui sont représentées de façon linéaire si la variable est de type primitif, ou en arborescence contenant d'autres variables s'il s'agit d'une liste.

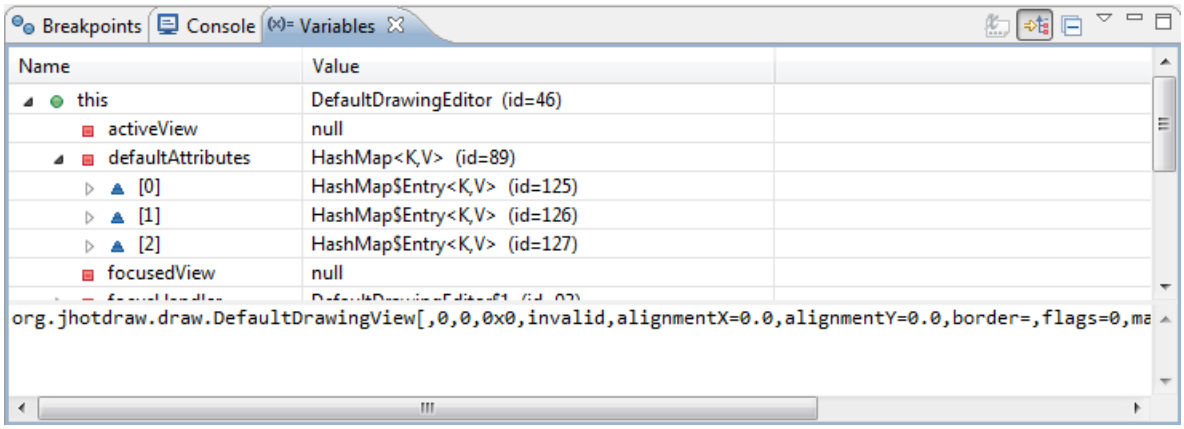

FIGURE 3.3 – Représentation d'une liste dans la vue *Variables*

Il faut noter qu'un utilisateur est capable de modifier la valeur d'une variable en temps réel, c'est-à-dire lors de l'exécution du programme en cours de débogage à travers cette vue. Ceci permet aux développeurs de vérifier le comportement d'un logiciel pour certaines valeurs et ainsi effectuer des corrections si cela s'avère nécessaire.

La vue *Breakpoints*

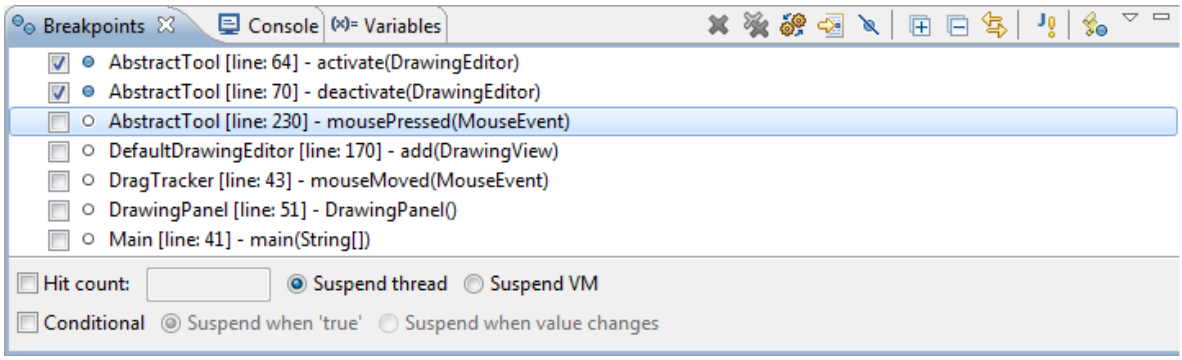

FIGURE 3.4 – Représentation de la vue *Breakpoint*

Cette vue, représentée par la figure 3.4, recense tous les points d'arrêt déposés dans l'environnement de travail. Nous pouvons les activer ou les désactiver à partir de cette fenêtre ou dans le code source même en double-cliquant sur une ligne de code.

Nous ne décrirons pas davantage ce mécanisme pour la simple raison qu'il n'est pas nécessaire d'en savoir plus pour la compréhension de la suite de ce document [4].

3.1.2 Fonctionnement

Présentation des éléments d'un débogueur

La section précédente expliquait comment utiliser le débogueur. Maintenant, nous allons décrire brièvement son fonctionnement. Pour ce faire, nous allons l'introduire via la figure 3.5 [WFB04], qui représente le cœur d'un débogueur sous Eclipse. Tous les éléments représentés sont des interfaces que l'utilisateur peut utiliser et ré-implémenter pour créer son propre débogueur.

4. Nous invitons les lecteurs intéressés à lire les différents mécanismes qu'il est possible d'effectuer dans le travail de Jean-Michel Doudoux [DOU07]

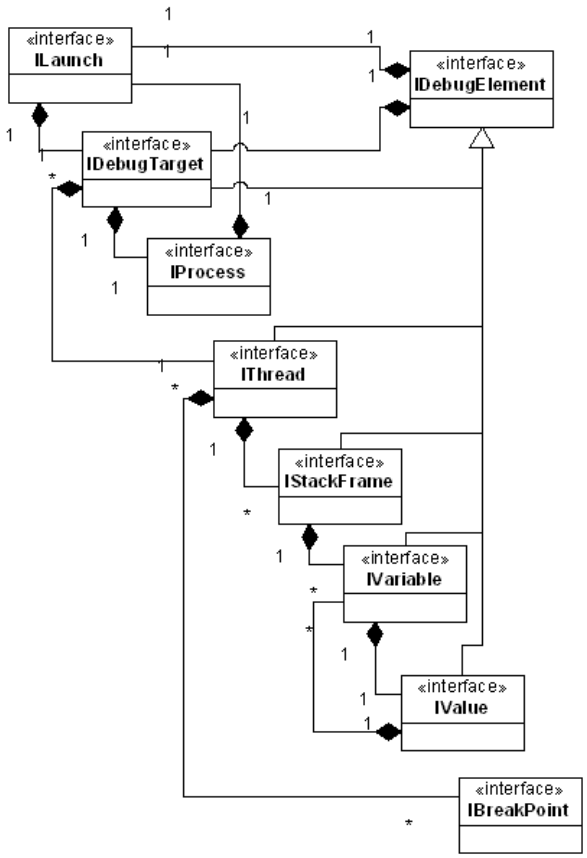

FIGURE 3.5 – Représentation UML du modèle de débogage sous Eclipse

Comme nous pouvons le voir, l'outil est composé de plusieurs artefacts qui ont chacun leur propre rôle bien défini. Nous nous focaliserons sur ceux qui nous concernent :

1. L'**`IDebugTarget`** est le contexte d'exécution de débogage, comme un processus ou une machine virtuelle. C'est la racine de chaque élément présent dans le débogueur et il supporte les différents états de l'outil qui sont :
   - `Terminate`
   - `Suspend` ou `Resume`
   - `Breakpoints`
   - `Disconnect`

   Il faut le voir comme une instance d'un débogueur.
2. L'**`IThread`** est un flux séquentiel de l'exécution d'informations du débogueur. Chaque `thread` contient une `stack`. Celle-ci n'est disponible que si le processus courant est dans l'état `Suspend`. Un thread peut être dans un état :
   - `Terminate`
   - `Suspend` ou `Resume`
   - `Stepping`
3. L'**`IStackframe`** représente le contexte d'exécution d'un processus suspendu par le

débogueur ou une méthode d'une classe. Cette interface contient les variables qui lui sont visibles, celles déclarées localement et ses arguments à l'exécution courante. Elle peut supporter différents états :
- `Terminate`
- `Suspend` ou `Resume`
- `Stepping`

4. L'**`IVariable`** représente une structure visible dans la pile ou une valeur. Chaque variable possède sa propre valeur qui peut contenir plusieurs sous-variables. Elle peut être modifiée en cours de débogage.
5. L'**`IValue`** montre la valeur d'une variable. Une valeur peut être une représentation d'une structure de données complexe contenant plusieurs variables.
6. L'**`IBreakpoint`** permet aux utilisateurs de suspendre l'exécution d'un programme à un certain point dans le code. C'est ce mécanisme qui suspend un `thread` et le débogueur notifiera la raison par l'envoi d'un signal `breakpoint`. Le processus ne sera remis en route que par une interaction de l'utilisateur.

L'interaction entre éléments

Après la présentation des composants importants du débogueur, expliquons maintenant son fonctionnement.

Tout d'abord, pour déboguer un programme, l'utilisateur doit poser un, voire plusieurs points d'arrêt dans le code source de son projet. S'il ne le fait pas, lancer le programme en mode *Debug* équivaudrait à le démarrer en mode *Run*.

Dès que les points d'arrêt ont été placés, le programme peut commencer à être débogué. Une instance de l'interface `IDebugTarget` sera alors créée et c'est elle qui va lancer les divers processus du programme en cours de débogage. Les divers `threads` vont s'exécuter normalement jusqu'à ce que l'un d'entre eux rencontre un `breakpoint`. À partir de cet instant, l'utilisateur a le contrôle total du programme et peut le manipuler de manière interactive. C'est en utilisant les fonctions de `stepping`[5] que le développeur empilera ou dépilera la `stack` du processus en suspens.

Nous pouvons ainsi voir la pile du `thread` qui augmente ou baisse selon la fonction `step` choisie. Les variables pourront alors être accessibles via la vue *Variables* pour voir leur valeur en temps réel ou encore pour pouvoir les modifier. Lors de la modification de la valeur d'une variable, le comportement du programme va prendre en compte ce changement sans devoir arrêter le débogage et le relancer. Cette opération s'effectue à la volée et le débogueur considère qu'il s'agit de la valeur initiale de la variable.

5. Voir les différentes fonctions `steps` qu'il est possible d'effectuer dans la description de la vue *Debug*

Il existe encore d'autres fonctionnalités que le débogueur permet d'effectuer, mais les principales sont celles qui viennent d'être expliquées et qui serviront pour l'implémentation de notre solution.

3.2 L'outil de visualisation *VERSO*

Dans le domaine de la qualité d'un logiciel, il existe diverses techniques pour pouvoir mesurer la qualité d'un processus et d'un produit. L'une d'entre elles est la visualisation d'un logiciel. Son but est de calculer des métriques d'un projet informatique et de les représenter sous forme graphique. Dans la lignée de ce type de logiciel, nous avons choisi l'outil *VERSO* qui permet ce genre d'opérations. De plus, étant donné qu'il a été développé comme un plug-in pour Eclipse, il nous permettra facilement d'ajouter une communication entre lui-même et le débogueur d'Eclipse. Nous allons donc présenter ci-après les fonctions importantes de *VERSO* qui nous seront nécessaires, et expliquer brièvement leur fonctionnement.

3.2.1 Présentation de l'outil

VERSO ou la **V**isualisation d'**É**valuation de **R**é-ingénierie des **S**ystèmes à **O**bjets est un outil de visualisation, développé par Guillaume Langelier [Lan10], qui calcule les métriques d'un projet et les affiche sous forme graphique. Deux versions de l'outil sont disponibles, l'une sous forme d'une application à part entière, l'autre sous la forme d'un plug-in à l'IDE Eclipse. C'est uniquement avec cette dernière que nous allons travailler. La figure 3.6 montre une illustration de l'outil visuel qui analyse un projet depuis l'environnement de travail de l'éditeur. Dans l'illustration, il s'agit du développement de *VERSO* qui est analysé par le plug-in. L'outil présente trois fenêtres principales :

- le contexte visuel où s'illustre le projet sous une forme graphique,
- la fenêtre supérieure droite qui recense les métriques sur l'objet sélectionné dans le contexte visuel,
- la fenêtre inférieure droite qui recense les bogues qui ont été repérés par l'utilisateur et qui les a énoncés.

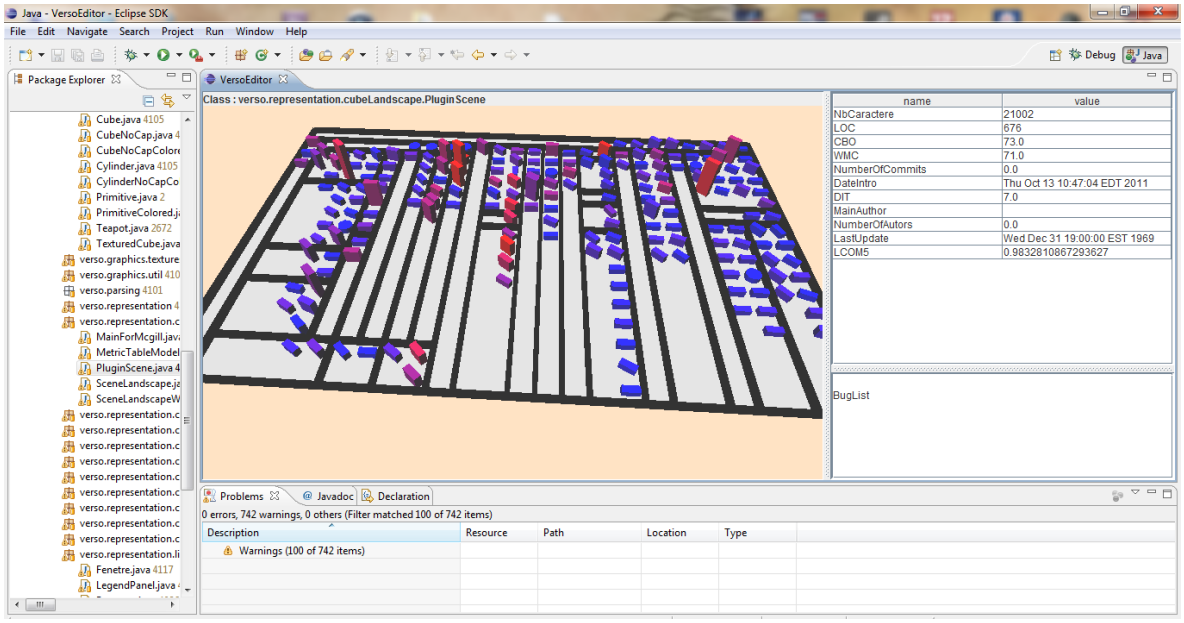

FIGURE 3.6 – Illustration de *VERSO* dans l'environnement d'Eclipse

Chaque élément d'un projet à sa propre représentation dans le contexte visuel. Les paquetages sont représentés par une technique de `Treemap` basée sur l'article de Brian Johnson et Ben Shneiderman [JS], modifiée par Guillaume Langelier lors du développement de *VERSO* [Lan10]. Les classes et les interfaces sont ensuite placées dans leur paquetage et sont respectivement symbolisées par un parallélépipède rectangle et un cylindre. Les méthodes ont la même représentation que les classes, à la différence près que leur taille et leur position dans la classe varient en fonction de leur nombre.

Pour représenter les métriques d'un logiciel dans le contexte visuel, *VERSO* utilise différents *Mappings* pour les classer tels que la qualité, le contrôle de version,... Il est donc possible de voyager d'un *Mapping* vers un autre sans trop de difficulté. La figure 3.7 propose un résumé des vues et des informations accessibles depuis *VERSO*.

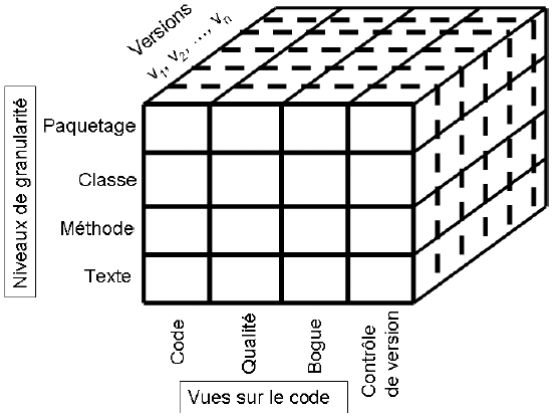

FIGURE 3.7 – Matrice 3D des vues atteignables dans *VERSO* par rapport à la granularité et l'évolution du logiciel au cours du temps

Le contexte visuel permet de représenter trois métriques maximum suivant le niveau de granularité, les autres étant exprimées dans la fenêtre supérieure droite de *VERSO*. Il faut savoir que la représentation de ces métriques a été fixée par une caractéristique visuelle et il n'est pas possible de la changer dans la version plug-in de l'outil. Le changement de représentation graphique ne se fait que dans la version applicative de *VERSO*. Par exemple, dans la vue *Qualité*, pour les métriques exprimées par la couleur, celle-ci va virer du bleu (si la métrique est faible) vers le rouge (si la métrique est forte). Le même raisonnement a été effectué pour la rotation et la hauteur des formes géométriques.

1. Au niveau des **paquetages**, deux métriques sont représentées graphiquement :
   - Le couplage entre les paquetages est représenté par la couleur.
   - La complexité d'un paquetage est représenté par sa hauteur.
2. Pour les **classes**, les trois représentations graphiques sont utilisées :
   - La couleur représente le couplage entre les objets d'une classe.
   - La hauteur affiche le poids des méthodes pour la classe.
   - La torsion indique le manque de cohésion dans les méthodes de cette classe ou l'héritage de la classe.
3. Enfin, les **méthodes** utilisent également les trois propriétés graphiques :
   - La couleur représente le couplage entre les méthodes.
   - La hauteur affiche la complexité de la méthode.
   - La torsion indique la cohésion entre les méthodes ou l'héritage.

Grâce à ce style de présentation des données, l'utilisateur emploie une et une seule vue pour avoir accès à ces informations et il n'est pas ainsi inondé par une multitude de fenêtres qui peuvent encombrer son environnement de travail.

3.2.2 Fonctionnement

La question que nous nous posons est de savoir comment *VERSO* peut calculer les métriques d'un projet. Pour cela, il utilise deux techniques :

- L'analyse statique
- L'analyse dynamique [6]

L'analyse statique

L'analyse statique d'un programme consiste à récupérer des informations sur le projet en cours de développement lors de sa compilation. En effet, le compilateur, et en particulier celui du langage Java, permet de repérer les erreurs de syntaxe et calculer certains problèmes sémantiques [7]. De cette manière, en se greffant au compilateur de la Java Virtual Machine

6. Développé par Wassim Yakoub [Yak11]
7. Une variable non initialisée ou une variable non utilisée

(JVM), *VERSO* peut calculer directement les différentes métriques du projet en cours de développement. De plus, lors de la modification du code source d'un programme, les métriques sont automatiquement recalculées et le contexte visuel est mis à jour instantanément. Par exemple, nous avons :

- le nombre de lignes de code d'un paquetage, d'une classe, ou d'une méthode (**LOC**),
- le manque de cohésion entre les méthodes (**LCOM**),
- la profondeur d'un arbre d'héritage (**DIT**),
- le poids d'une méthode dans une classe (**WMC**),
- le couplage entre les objets (**CBO**),
- etc.

Cette liste est non exhaustive.

L'analyse dynamique

L'analyse dynamique consiste à étudier le comportement d'un programme via une trace d'exécution. Grâce aux développements apportés par Wassim Yakoub [Yak11], *VERSO* peut calculer de nouvelles métriques et ainsi apporter plus d'informations sur la qualité d'un logiciel. Via ces nouvelles métriques, *VERSO* peut analyser le comportement d'un programme à travers le temps via la trace d'exécution. Étant donné qu'une trace d'exécution est en réalité statique, l'avantage est que nous pouvons faire des allers-retours dans le scénario pour mieux analyser les différents états d'une variable ou d'un objet.

Pour mieux distinguer les interactions entre les objets graphiques et les éléments du projet, la figure 3.8 montre les différents composants importants qui constituent une partie du cœur de *VERSO*.

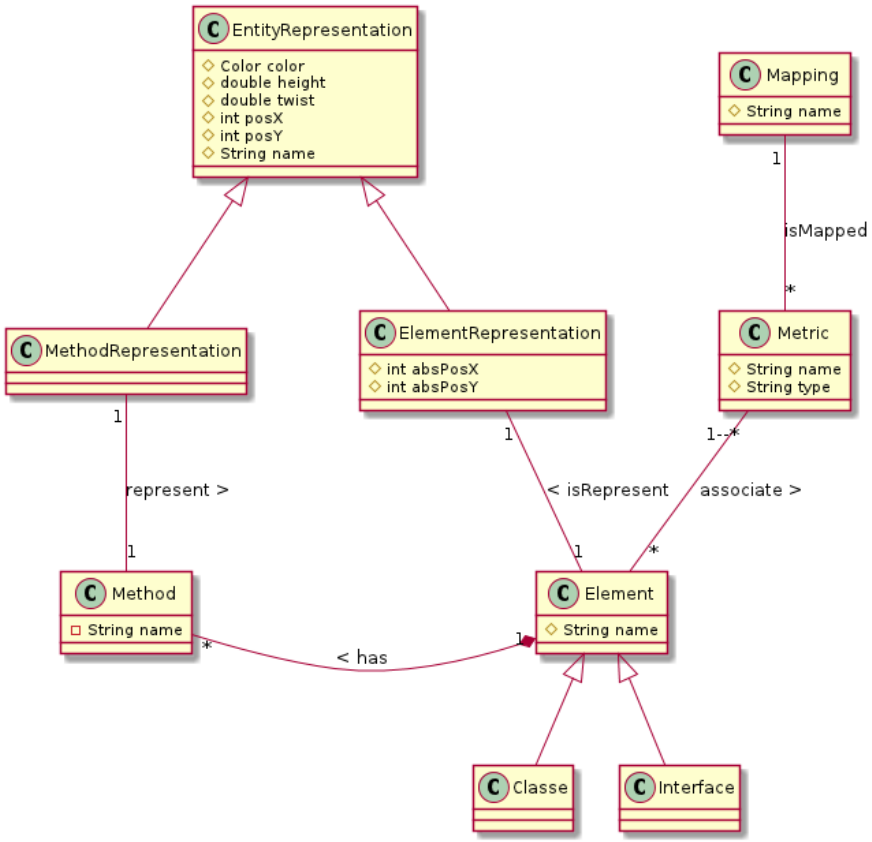

FIGURE 3.8 – Diagramme de classe représentant les composants importants de *VERSO*

Ces composants se décrivent comme suit :

- La classe **`EntityRepresentation`** est l'élément graphique qui crée la forme géométrique pour représenter la classe, l'interface, ou la méthode qu'elle désigne. Nous pouvons voir qu'elle possède une couleur, une hauteur et un angle de rotation. Nous définissons sa position dans le paquetage ou dans la classe dans le cas d'une méthode, grâce aux deux variables entières `posX` et `posY`.
- La classe **`MethodRepresentation`** est une spécification pour l'implémentation de la représentation des méthodes dans une classe.
- La classe **`ElementRepresentation`** est la spécification pour l'implémentation de la représentation d'une classe et d'une interface [8].
- La classe **`Mapping`** est la vue dans laquelle nous affichons certaines métriques relatives à la représentation choisie dans le contexte visuel.
- La classe **`Metric`** est la mesure d'un paramètre d'un élément d'un projet informatique. Elle a un nom et est d'un certain type : numérique, intervalle,...
- Les classes **`Method`**, **`Element`**, **`Classe`** et **`Interface`** représentent leur propre concept

8. Une classe sera représentée par un parallélépipède rectangle tandis que l'interface par un cylindre. Il va de soi que la rotation d'un cylindre n'a aucun sens, mais étant donné qu'une interface n'est jamais très complexe, cela ne pose aucun problème pour la métrique choisie.

dans un projet de développement Java.
Bien évidemment, *VERSO* possède bien plus de composants, mais nous nous focalisons sur ceux qui seront nécessaires à la création de notre débogueur visuel.

Maintenant que nous avons présenté et expliqué brièvement les deux outils, passons à la proposition d'une solution aux deux problèmes que nous avons énoncés dans l'introduction.

3.3 Proposition de la solution

Avant de soumettre une proposition aux deux problèmes que nous avons formulés, nous les rappelons brièvement. Le premier est de rendre le débogueur plus convivial à son utilisation, le deuxième est de permettre l'utilisation d'un outil de visualisation pour la maintenance et le développement de logiciels.

Après avoir présenté les deux outils, nous avons retenu une solution qui serait d'ajouter une communication entre ce plug-in et le débogueur d'Eclipse. L'avantage d'utiliser *VERSO* est qu'il a été développé comme un plug-in. De ce fait, l'intégration de nouveaux composants que l'éditeur peut fournir est beaucoup plus facile [9]. Pour réussir l'intégration du débogueur à l'intérieur de *VERSO*, les figures 3.5 et 3.8 peuvent nous aider. En ne retenant que les composants nécessaires du premier diagramme de classe et en les fusionnant avec le deuxième, nous obtenons un diagramme de classe préliminaire de ce que devra être notre outil (voir figure 3.9). Nous pouvons considérer ce schéma UML comme étant une partie de l'architecture de l'outil. Nous remarquons que les variables n'étant pas représentées dans *VERSO*, le seul moyen de créer la communication entre le débogueur et le contexte graphique se trouve au niveau des méthodes.

9. Voir les tutoriaux sur la conception d'un plug-in ou d'une application RCP à ces adresses [plua], [plub], [rcp]

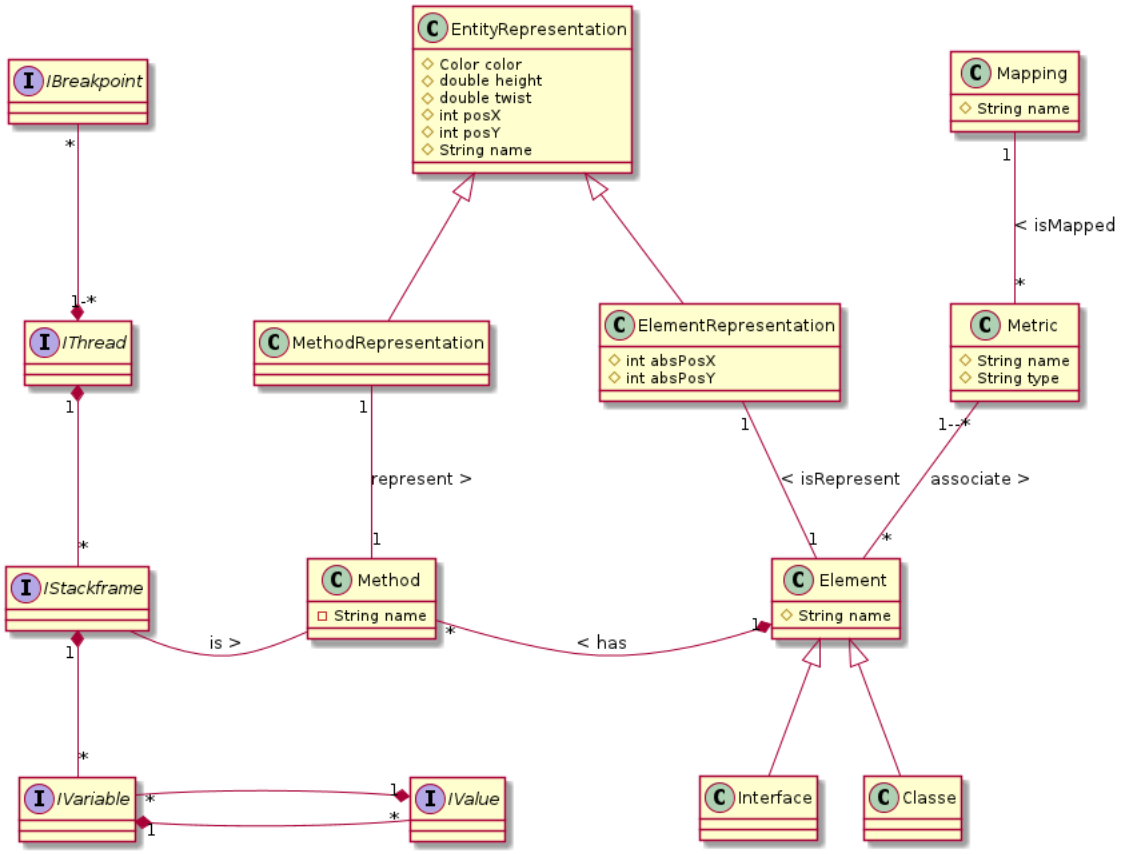

FIGURE 3.9 – Diagramme de classe représentant la fusion des composants importants du débogueur et de *VERSO*

À partir de ce point, l'outil de débogage sera plus ergonomique et permettra aux utilisateurs d'abandonner leurs mauvaises habitudes au profit de l'utilisation de cet outil. De plus, l'outil de visualisation qui incorporera une composante de débogage permettra d'être utilisé pendant le développement et la maintenance de logiciels.

Le chapitre suivant traitera des différentes étapes du développement, à savoir les questions posées pour réaliser l'implémentation de la communication, les objectifs fixés pour la réalisation de l'outil et les choix d'implémentation.

Chapitre 4

Développement de l'outil : *VersoDebug*

Pour atteindre l'objectif qui est de permettre d'utiliser des techniques de visualisation dans la maintenance et le développement de logiciels, nous allons ajouter une fonctionnalité à l'outil *VERSO* que nous appellerons *VersoDebug*, pour ne pas la confondre avec la version originale. Cette fonctionnalité lui permettra d'utiliser le débogueur d'Eclipse pour traduire les données fournies et les interpréter dans son contexte visuel. Le développement de l'outil va se baser sur le diagramme de classes présenté à la figure 3.9 qui est l'architecture de notre solution que nous allons implémenter.

4.1 Un débogueur visuel

Pour que *VersoDebug* utilise le débogueur d'Eclipse, nous allons ajouter une communication entre eux. Ensuite, à chaque notification que le débogueur enverra, notre solution devra interagir en conséquence. La première version consistera à afficher le chemin d'exécution dans l'environnement visuel du processus en cours de débogage ainsi que son contexte. Enfin, nous évaluerons l'outil pour en tirer les conclusions concernant l'ajout opéré.

Avant de commencer le développement du débogueur visuel, nous devions tout d'abord préparer l'environnement de débogage. En effet, *VersoDebug* utilise divers *Mappings* pour analyser la qualité, le *Versioning*, le contrôle,... en calculant des métriques. Nous avons donc décidé de créer notre propre *Mapping* que nous avons intitulé ***mvDebug***. Dès que le programme sera lancé en mode *Debug*, cela provoquera un changement automatique vers le nouveau *Mapping* pour permettre aux utilisateurs de voyager d'un *Mapping* à l'autre sans perte d'informations.

Nous avons décidé de créer une métrique qui calculera la position d'une méthode dans la pile d'exécution. Étant donné qu'il s'agit d'une métrique dynamique, nous ne pouvions utiliser

la technique de calcul des métriques à la compilation formulée dans la thèse de Guillaume Langelier[Lan10] et expliquée dans la section 3.2.2 dans le paragraphe sur l'analyse statique. C'est pourquoi le rapport de Wassim Yakoub [Yak11] fourni avec le code source et expliqué dans la section 3.2.2 dans le paragraphe concernant l'analyse dynamique, nous a aidés à créer la métrique dynamique. Nous déciderons plus tard à quelle technique visuelle nous l'associerons, à savoir la couleur, la hauteur ou l'orientation de la forme géométrique.

4.1.1 Ajout de la communication entre les deux outils

Comme expliqué dans les différents didacticiels concernant la conception d'un plug-in ou le développement d'une application RCP [plua], [plub] et [rcp], nous devons ajouter les dépendances nécessaires à la communication entre l'outil et le débogueur. La figure 4.1 montre les différents composants que nous avons insérés. Ces composants sont les suivants :

- *org.eclipse.debug.core*. Ce paquetage permet de communiquer avec le cœur du débogueur quel que soit le langage de programmation utilisé.
- *org.eclipse.debug.ui*. Ce paquetage permet d'utiliser les boutons de l'interface graphique relatifs à l'utilisation du débogueur.
- *org.eclipse.jdt.debug*. Ce paquetage permet d'utiliser le débogueur relatif au langage de programmation Java.
- *org.eclipse.jdt.debug.ui*. Ce paquetage permet l'utilisation des boutons de l'interface graphique relatifs au débogueur pour Java.

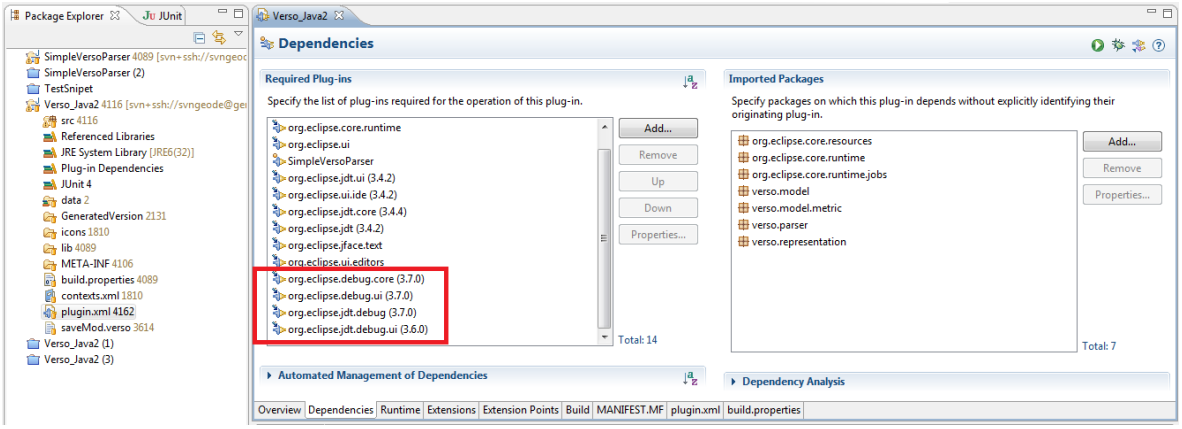

FIGURE 4.1 – Ajout des dépendances pour la communication entre *VersoDebug* et le débogueur d'Eclipse

4.1.2 Écoute des notifications du débogueur

Une fois la communication établie, il faut pouvoir réceptionner et réagir en fonction de la notification reçue. Pour ce faire, nous devons implémenter l'interface *IDebugEventSetListener* [api]. La classe *DebugEventListener.java* a été créée. En voici l'implémentation.

```
public class DebugEventListener implements IDebugEventSetListener {
    /**
    * Start the notification of the debugger
    */
    public void startDebugEventListener() {
        DebugPlugin.getDefault().addDebugEventListener(this);
    }

    /**
    * Stop the notification of the debugger
    */
    public void stopDebugEventListener() {
        DebugPlugin.getDefault().removeDebugEventListener(this);
    }

    @Override
    public void handleDebugEvents(DebugEvent[] events) {
        //TODO implement this
    }
}
```

Listing 4.1 – DebugEventListener

C'est dans la méthode *handleDebugEvents* que nous allons récupérer toutes les notifications du débogueur et cela nous permettra de faire interagir *VersoDebug* comme nous le souhaitons. Les méthodes *startDebugEventListener* et *stopDebugEventListener* ont pour but respectivement de démarrer ou d'arrêter l'écoute des notifications du débogueur.

Arrivés à ce stade, nous nous sommes posé deux questions :

1. Comment représenter la pile d'exécution ?
2. Comment représenter le contexte ?

4.2 Représentation de la pile d'exécution

Pour répondre à la première question, nous avons réfléchi à plusieurs solutions :

1. tracer des lignes au-dessus des blocs représentant les classes,
2. procéder à un dégradé de couleurs,

3. changer les dimensions des blocs représentant les classes qui sont dans la pile d'exécution.

Nous avons rejeté la première solution tout simplement parce que si la pile d'exécution du processus en cours de débogage est très grande, le nombre de lignes serait trop important et pourrait diminuer la visibilité du contexte visuel.

C'est pourquoi nous nous sommes orientés vers la deuxième proposition. En effet, le dégradé de couleurs ne diminuera pas la visibilité du contexte visuel ainsi que la lecture des informations affichées sur l'écran. Nous avons créé une métrique que nous avons nommée *StackLevel* et nous l'avons liée aux coloris des classes et méthodes du contexte visuel. Dans son article, en ce qui concerne les couleurs à prendre en compte, Maureen Stone [Sto06] décrit assez bien celles qu'il faut choisir et dans quelles circonstances. Elle précise que nous devons sélectionner notre palette entre deux ou trois couleurs maximum. De plus, le gris doit être utilisé pour montrer la neutralité. C'est pourquoi cette couleur sera choisie pour identifier les classes et les méthodes qui ne seront pas actives dans le processus en cours de débogage. Pour celles qui le seront, nous avons opté pour la couleur rouge, représentant la méthode qui se trouve au plus haut niveau de la pile d'exécution et le blanc pour celle qui est au plus bas. Pour calculer le dégradé, nous avons utilisé l'algorithme 1 qui prend le format de couleurs **RVB**, pour le rouge, le vert et le bleu des deux pigments choisis pour représenter les classes, l'une la plus ancienne et l'autre la plus récente.

Algorithme 1 Calcul du dégradé de couleurs

Entrées: $C1 \in RGB$, $C2 \in RGB$, $Cl \in ClassProject$, $N \in NbreElementInStack$, $List < ClassProject > \in ElementInStack$

pour $i = 0 \rightarrow N - 1$ **faire**
  $Class \leftarrow List.get(i)$
  si $i = 0$ && $N > 1$ **alors**
    $Class.color = C2$
  sinon
    $red = C2.red + \frac{C1.red - C2.red}{N} * (N - i)$
    $green = C2.green + \frac{C1.green - C2.green}{N} * (N - i)$
    $blue = C2.blue + \frac{C1.blue - C2.blue}{N} * (N - i)$
    $Class.color = RGB(red, green, blue)$
  finsi
fin pour
$Cl.color = C2$

Il faut noter que cet algorithme fonctionne de la même manière pour les méthodes. Si plusieurs méthodes d'une même classe se trouvent dans la pile d'exécution, la couleur que prendra la représentation de la classe sera celle de la méthode qui se trouvera au plus haut niveau de la pile d'exécution. De plus, ce seront uniquement les classes qui se trouveront dans

le projet qui changeront de couleur, les librairies [1] annexes importées dans un projet n'étant pas représentées dans *VersoDebug*.

Le problème est qu'avec un dégradé de couleurs, certains utilisateurs pourraient avoir du mal à distinguer deux nuances très proches l'une de l'autre. Ce sont principalement les personnes atteintes de troubles de la vue (daltonisme, mal-voyance,...) qui seraient le plus concernées. Malgré tout, ils doivent être capables d'utiliser l'outil. C'est pourquoi nous avons décidé de rajouter la troisième solution pour optimiser la représentation de la pile d'exécution, qui consiste à changer la dimension des classes et méthodes qui se trouvent dans la *stack* du processus en cours de débogage. Étant donné que nous ne pouvons *Mapper* une métrique à plusieurs facteurs visuels, nous avons créé une nouvelle métrique qui a la même fonction que la précédente. Pour les différencier, nous les avons respectivement appelées :

1. *StackLevelC* pour le lien avec la couleur.
2. *StackLevelH* pour le lien avec la hauteur.

Le changement de dimension se fait de telle sorte que la longueur et la largeur du cube augmentent d'une taille fixe, tandis que la hauteur augmente en fonction de la position de la méthode dans la pile d'exécution. Il se forme alors un dégradé de taille entre les différentes classes. De cette façon, nous pouvons nous poser la question de savoir si les dégradés de tailles et de couleurs suffisent à accentuer le focus sur les classes et méthodes qui se trouvent dans la pile d'exécution du processus en cours de débogage.

4.2.1 *FishEye* OpenGL

Étant donné que *VersoDebug* utilise une librairie OpenGL, nous nous sommes demandés si nous ne pouvions pas utiliser un effet graphique pour accentuer le focus sur les classes et méthodes qui figurent dans la pile d'exécution par rapport aux autres. C'est la raison pour laquelle nous avons voulu essayer d'effectuer un effet ≪*FishEye*≫ sur la représentation graphique de la classe.

Cet effet consiste à appliquer une lentille qui a pour effet de transformer le champ de vision humain en celui du poisson. Les figures 4.2 [chab] et 4.3 [chaa] montrent le champ de vision normal d'un poisson comparé à celui d'un humain. Nous remarquons que le champ de vision est beaucoup plus grand chez le poisson (180 °) que chez l'homme (120 °).

1. Nous entendons par librairie un répertoire de fonctions déjà existantes utilisées pour l'implémentation d'un projet.

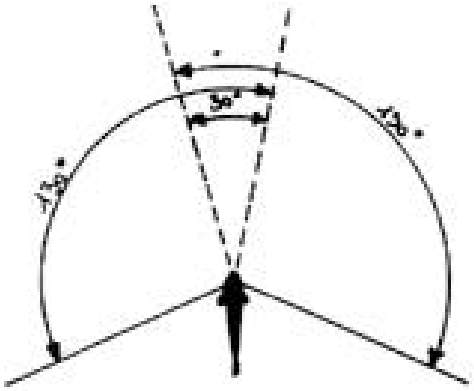

FIGURE 4.2 – Champ de vision du poisson

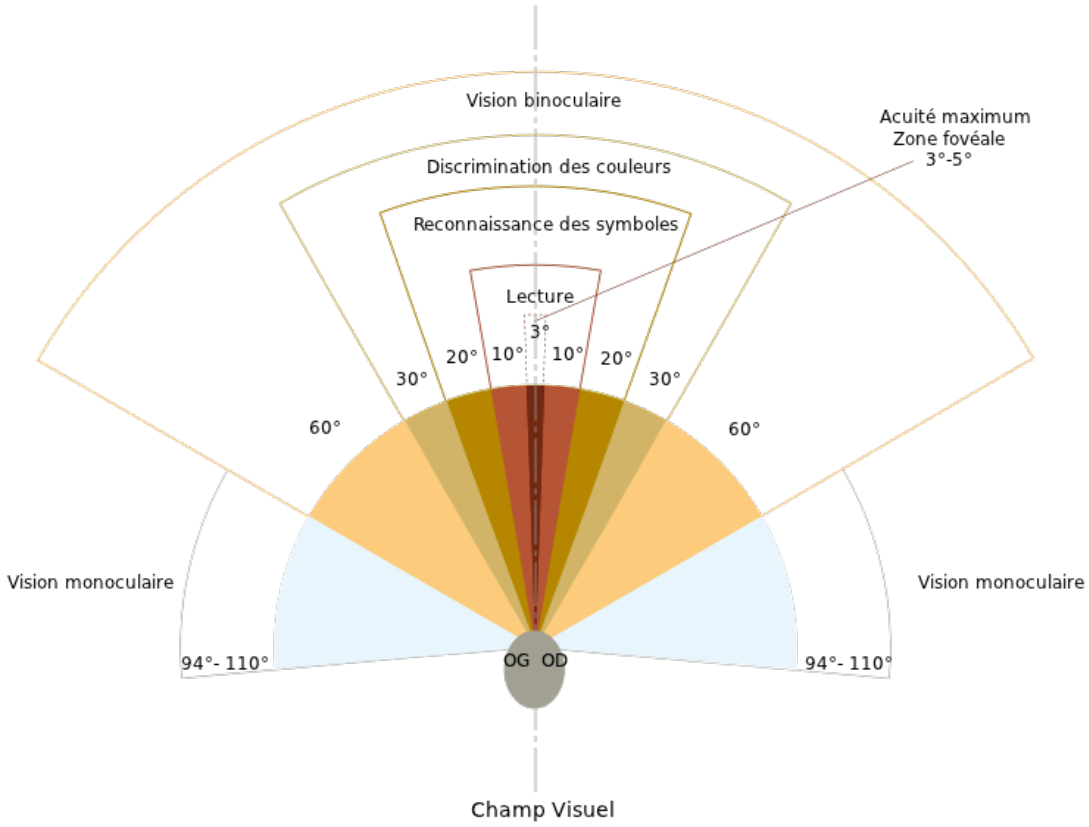

FIGURE 4.3 – Champ de vision humain

Le résultat d'un effet FishEye est une transformation du champ de vision humain vers celui du poisson qui a pour effet de rendre le centre de l'image nette et d'effectuer un rétrécissement vers l'extrémité de l'image. Nous pouvons voir le genre d'effet artistique obtenu à la figure 4.4.

FIGURE 4.4 – Effet FishEye

Le but est d'appliquer cet effet sur chaque représentation d'une classe qui se trouve dans la pile d'exécution en cours de débogage. Cette classe sera le centre de l'image et les classes en sa périphérie subiront une répulsion, ce qui accentuera le focus sur cette classe.

Mais le problème est que nous ne pouvons appliquer cet effet sur plusieurs points de la scène. En effet, le fait de poser une lentille qui provoque la transformation voulue ne peut se faire que sur toute l'image et pas sur plusieurs points de l'image. Nous avons donc abandonné cette solution pour aller vers la création personnalisée d'un effet *FishEye*.

4.2.2 *FishEye* personnalisé

Pour pallier le problème concernant l'utilisation du *FishEye* sur plusieurs points de la caméra, nous proposons de créer notre propre effet. Il consiste à créer une répulsion sur les objets se situant en périphérie de la classe ou méthode qui se trouve dans la pile d'exécution du processus en cours de débogage.

Pour illustrer nos propos, imaginons d'une part qu'une classe qui doit subir le focus, être le centre d'une rose des vents (figure 4.5) et d'autre part, les classes à sa périphérie qui subissent une répulsion par rapport au centre. Seules les classes éloignées de deux niveaux par rapport au centre seront affectées. Nous définissons une classe de niveau 1, une classe qui est une voisine directe avec la classe qui se trouve dans la pile d'exécution. Une classe de niveau 2 est une voisine directe avec une classe de niveau 1. Celles de niveau 1 verront leur dimension baisser très fortement, et celles de niveau 2 baisseront également, mais en moindre importance.

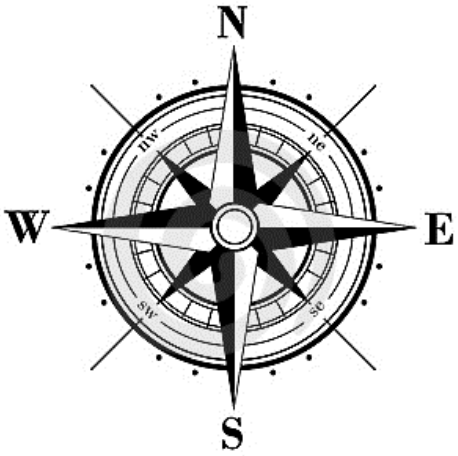

FIGURE 4.5 – Rose des vents

Le problème qui se pose alors est de savoir comment une classe va réagir si elle doit subir la répulsion de deux centres de gravité. Pour répondre à cette question, nous avons créé le tableau 4.1 qui synthétise le comportement qu'une classe va avoir en fonction de sa position d'origine et de la direction demandée.

	N	**S**	**W**	**E**	**NW**	**NE**	**SW**	**SE**
N	N	C	NW	NE	N	N	W	E
S		S	SW	SE	W	E	S	S
W			W	C	W	N	S	S
E				E	N	N	S	S
NW					NW	N	W	C
NE						NE	C	E
SW							SW	S
SE								SE

TABLE 4.1 – Comportement de la répulsion

Nous avons également décidé que si une classe de niveau 1 doit subir une répulsion de niveau 2, alors aucun changement ne s'opérera sur elle pour des raisons de priorité.

Ensuite, il nous a été permis de déterminer si nous voulions que le déplacement se fasse uniquement à l'intérieur du paquetage de la classe, ou que les classes qui se trouvent dans le paquetage voisin soient aussi soumises à la répulsion. Pour chaque représentation d'une classe, *VersoDebug* calcule la position logique à l'intérieur d'un paquetage ainsi que sa position réelle dans tout le projet, ce qui a rendu le travail très facile à implémenter.

En ce qui concerne les méthodes, nous voulions appliquer le même effet, mais leur agencement dans les classes ne se faisait pas de la même manière que celui dans les paquetages. En effet, la construction et le placement des méthodes [2] rendent le *FishEye* personnalisé inutili-

2. voir le paragraphe 4.1.2.2 de la thèse de Guillaume Langelier[Lan10]

sable. Les méthodes qui se trouvent dans la pile d'exécution grossissent et poussent les autres en dehors de la classe, ce qui rend l'affichage totalement illisible. Pour pallier ce problème, nous avons décidé d'afficher uniquement les méthodes qui se trouvent dans la pile d'exécution, les autres n'étant tout simplement pas représentées dans le contexte visuel.

4.2.3 Aperçu de l'outil lors d'une exécution

Après l'implémentation de la représentation du chemin d'exécution du processus en cours de débogage, nous avons testé notre outil sur le programme «open source» *JHotDraw*. Les figures 4.7, 4.8, 4.9 et 4.10 montrent l'interaction de notre outil avec ce logiciel lorsque nous le déboguons à partir de la méthode *main*.

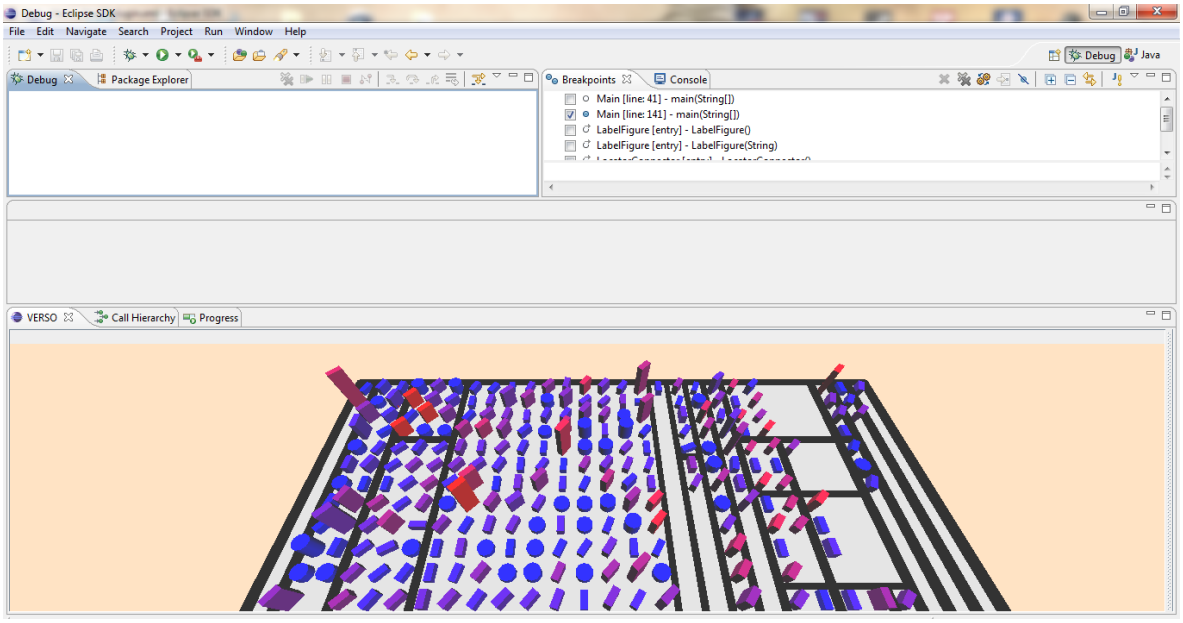

FIGURE 4.6 – Représentation du programme *JHotDraw* avant de commencer le débogage

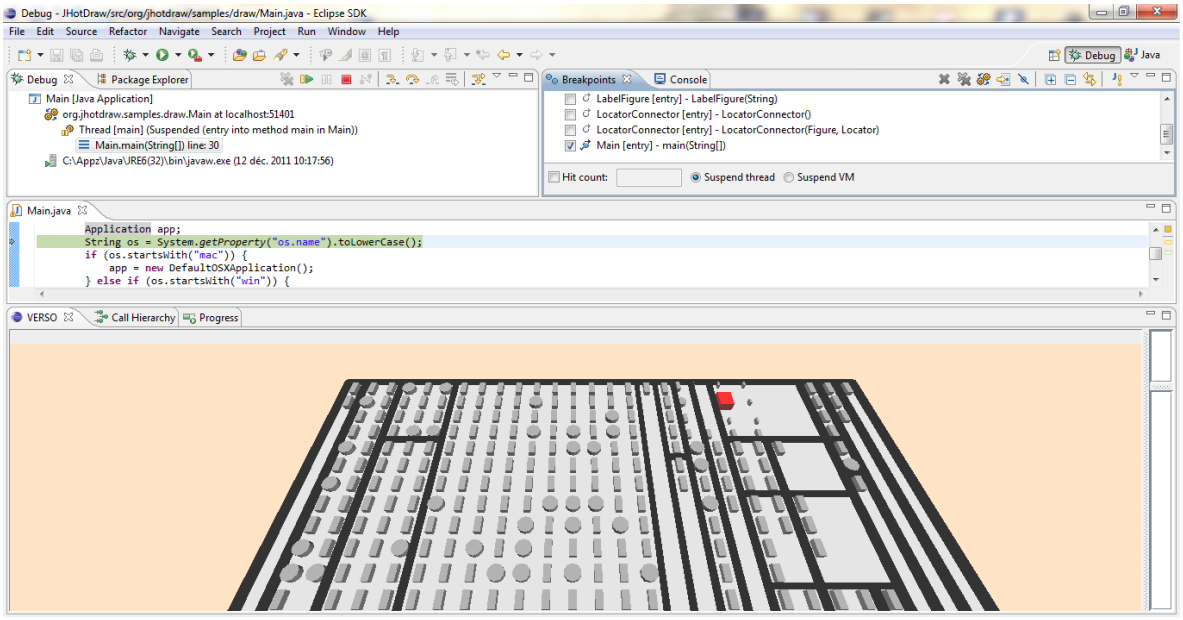

FIGURE 4.7 – Début de débogage avec l'effet *FishEye* personnalisé autour de la classe contenant la méthode *main*

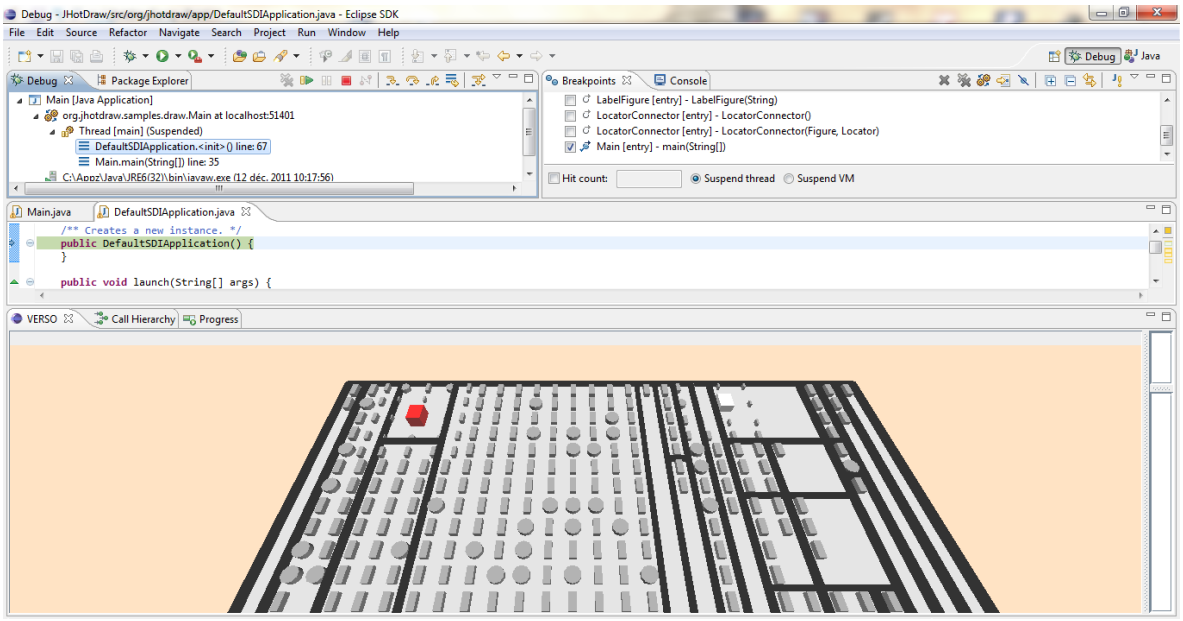

FIGURE 4.8 – Premier exemple de dégradé avec les effets *FishEye*

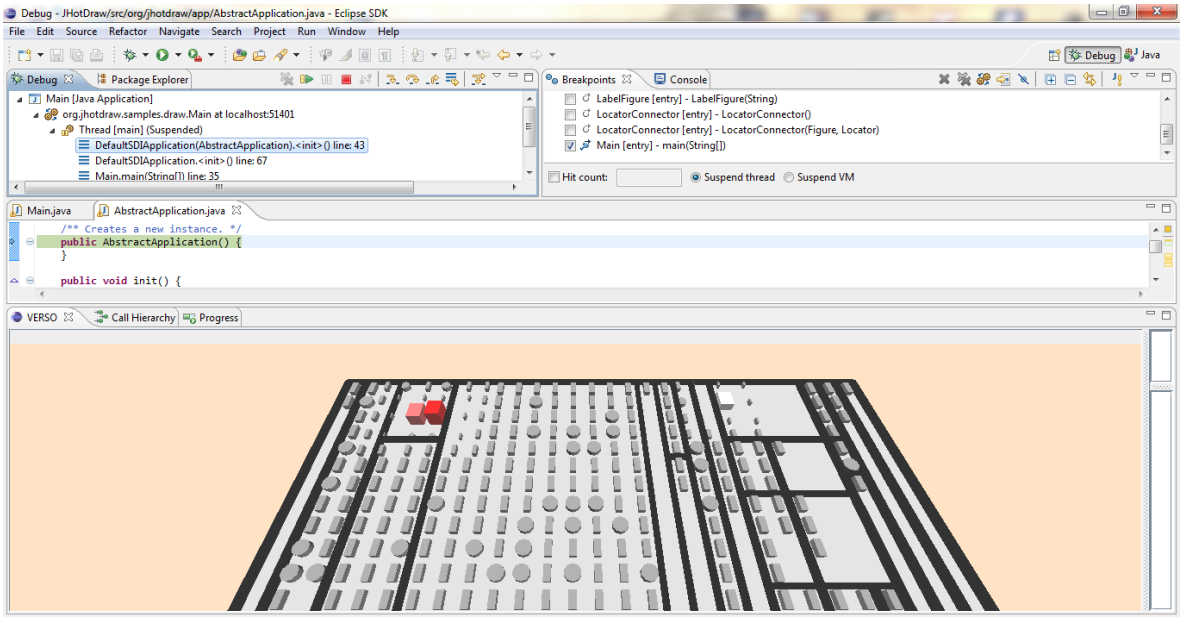

FIGURE 4.9 – Deuxième exemple de dégradé avec les effets *FishEye*

Nous remarquons qu'une des classes initialement petite dans la figure 4.8 a changé de dimension dans la figure 4.9 en raison de son implication dans la pile d'exécution

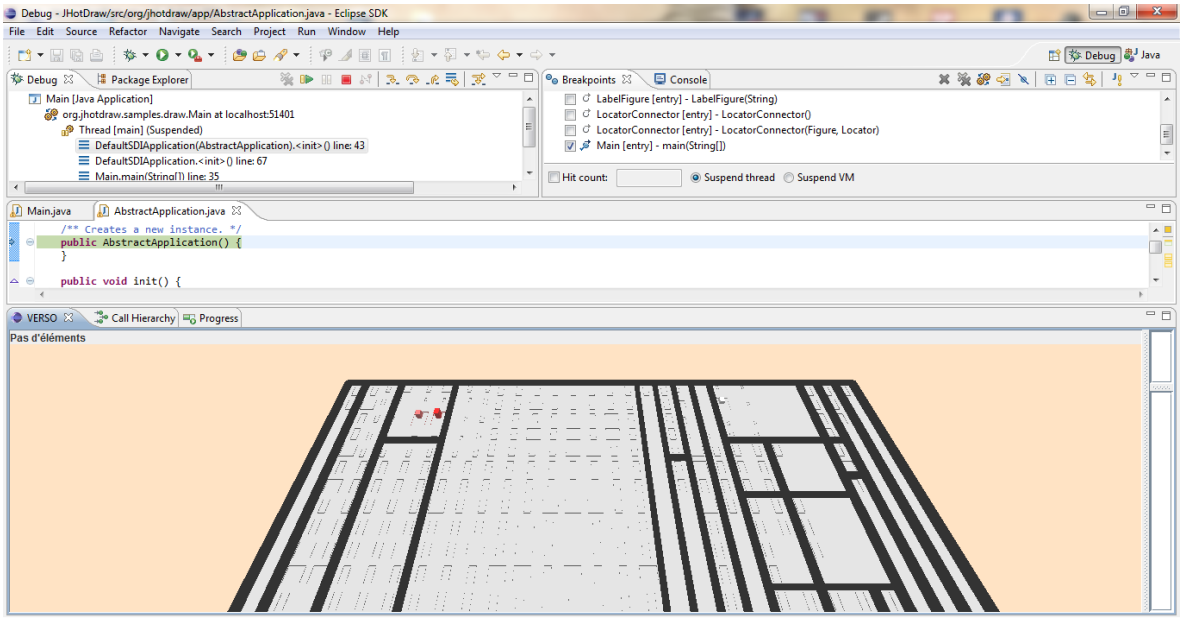

FIGURE 4.10 – Exemple de dégradé avec vue sur les méthodes

4.3 Représentation du contexte

La représentation de la pile d'exécution étant terminée, nous nous sommes occupés de la représentation du contexte des méthodes se trouvant dans la pile d'exécution. Pour rappel,

l'un des buts de *VersoDebug* est de contenir un maximum d'informations dans une seule vue pour éviter à l'utilisateur de naviguer d'une fenêtre à l'autre [3].

Nous avons décidé de ne plus utiliser la vue ***Variables*** d'Eclipse et de la remplacer par une technique de *Tooltip* [4] qui affiche les variables courantes lorsque nous passons le curseur sur la classe ou la méthode qui se trouve dans la pile d'exécution en cours de débogage. Cette technique est tirée des principes des *Tooltip* dans Eclipse. C'est-à-dire que lorsque nous passons le curseur sur des lignes de code, une fenêtre apparait en affichant la *javadoc* associée au code où la souris s'est arrêtée. Son corps est en fait un tableau qui comprend deux colonnes, à savoir : le nom de la variable et sa valeur. Comme pour la vue d'Eclipse, il est possible de descendre dans la hiérarchie d'une variable composée [5] pour voir les attributs et leur valeur en temps réel. La figure 4.11 représente le résultat obtenu en affichant les variables de la méthode *main*. Les variables sont *args* et *os* qui ont respectivement les valeurs *id=31* et *windows7* dans le cas présenté.

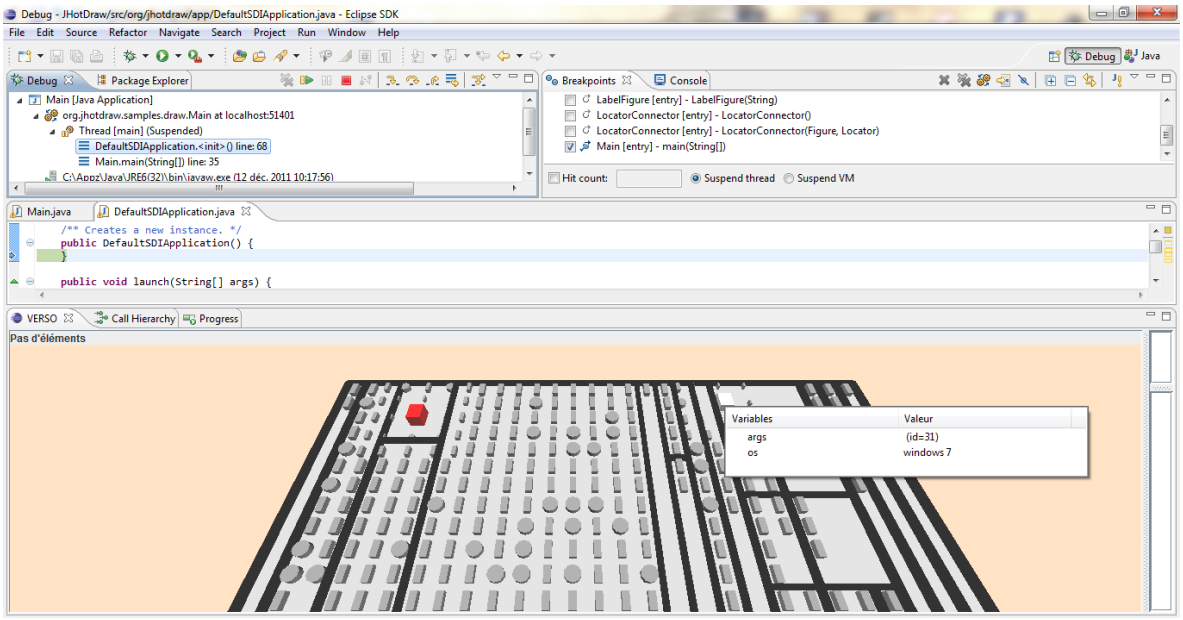

FIGURE 4.11 – Représentation du contexte d'une méthode

L'affichage des valeurs des variables apporte certains avantages : la suppression d'une vue dans Eclipse et un gain d'espace pour agrandir l'environnement visuel. Ceci permet une meilleure visibilité du contexte.

Deux questions principales sont apparues lors du développement de cette solution, à savoir :

3. Ceci est expliqué en détail dans la thèse de Guillaume Langelier [Lan10] dans les sections 3.1 à 3.3 inclus
4. Un `Tooltip` est une boite de dialogue qui apparaît lorsque nous passons le curseur sur un objet précis.
5. Variable composée de une à plusieurs variables.

– Quelle *stackframe* afficherons-nous si plusieurs méthodes d'une même classe participent au débogage lorsque la vue est axée sur les classes ?
– Quelle *stackframe* afficherons-nous si une méthode est récursive sur elle-même ?

En effet, ces problèmes sont apparus parce que nous ne pouvons afficher qu'un seul *tooltip* par méthode. C'est pourquoi nous avons décidé d'afficher la *stackframe* la plus récente dans le cas des deux questions posées.

4.4 Le multi-threading

Dans le cadre du développement ou de la maintenance de logiciels, il n'est pas rare que plusieurs processus tournent simultanément. Le cas le plus fréquent est lorsque deux processus veulent accéder à une même ressource de manière concurrente. C'est dans cette optique que se pose le problème du débogage de plusieurs processus à l'aide de notre outil.

La question que nous pouvons nous poser est comment pouvons-nous représenter le chemin d'exécution de deux, voire plusieurs processus différents qui sont en cours de débogage ? Il est rare que cela se produise, mais posons-nous tout de même la question. Un concepteur à la possibilité de l'appliquer dans Eclipse, donc nous devons être capables de gérer ce cas d'utilisation.

Une première proposition était de choisir deux palettes de couleurs pour les différencier. Ainsi nous aurions pu voir le tracé de toutes les tâches en cours dans le contexte visuel. Ceci aurait pu s'avérer intéressant, mais comment gérer une ressource partagée par plusieurs tâches ? En effet, une ressource peut être utilisée par divers processus de façon simultanée. De ce fait, la couleur ne peut être assignée qu'à une seule classe ou méthode. Cela provoquerait un «trou» dans le chemin d'exécution d'une tâche et il en va de même pour le dégradé sur la hauteur.

Mais est-il vraiment possible de déboguer deux processus en même temps ? La réponse à cette question est positive, mais l'utilisateur ne peut avancer que dans une seule pile d'exécution à la fois. C'est pourquoi la deuxième solution que nous avons proposée était tout simplement d'afficher uniquement la tâche courante choisie par le développeur. En suggérant cela, nous avons voulu rajouter un *Combobox* dans le contexte visuel qui contient tous les processus en cours de débogage et permet ainsi au concepteur de choisir la tâche à montrer. Mais en choisissant cette méthode, nous rajoutions des informations redondantes par rapport à la vue ***Debug***. En effet, elle liste tous les *threads* du programme et affiche leur pile pour ceux qui sont en cours de débogage. Nous avons donc décidé d'abandonner l'ajout de la liste des processus au profit d'un *Listener* qui écoute une tâche ou une pile d'une tâche en cours de débogage pour l'afficher dans le contexte visuel. La figure 4.12 exprime le comportement de *VersoDebug* lors du changement de *thread* en cours de débogage dans la vue ***Debug***.

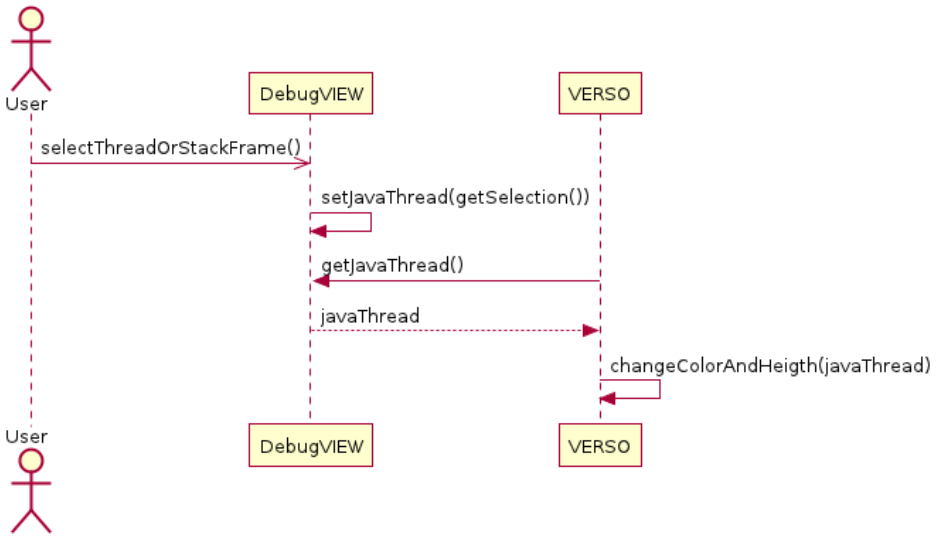

FIGURE 4.12 – Diagramme de séquence exprimant le comportement de *VersoDebug* lors du changement de sélection d'un *thread*

Dans ce chapitre, nous avons énoncé tous les choix de notre implémentation pour en arriver à une version de *VersoDebug* capable de communiquer et d'interagir avec le débogueur d'Eclipse. Notre outil affiche le chemin d'exécution d'un processus en cours de débogage et peut afficher le contexte d'une méthode lorsque nous passons le curseur sur sa représentation dans le contexte visuel. Le développement de la première version de notre outil étant terminé, il nous faut le tester sur un cas réel.

Le prochain chapitre expliquera le déroulement de l'évaluation, et traitera de l'analyse des résultats des sujets qui ont participé à l'expérimentation de l'outil.

Chapitre 5

Évaluation de l'outil *VersoDebug*

Après le développement de l'outil, l'étape suivante consiste à valider *VersoDebug*. Ce chapitre présente une évaluation quantitative de notre outil qui a été construite en se basant sur la méthodologie de C.Wohlin et al. [WRH+00]. L'expérimentation sera divisée en plusieurs points :

1. les sujets qui participent à l'expérience,
2. le temps imparti à la réalisation de la tâche globale à effectuer,
3. l'environnement de travail des utilisateurs,
4. les questionnaires et la tâche principale.

Enfin, nous analyserons les résultats de l'évaluation, ainsi que les commentaires des participants ayant utilisé notre outil pour finalement tirer des conclusions sur l'expérience qui s'est déroulée. La validation des résultats se fera par le biais de tests statistiques pour appuyer nos dires et nous terminerons par nos commentaires sur l'évaluation en tant que telle.

5.1 Présentation de l'expérimentation

Cette expérimentation vise à savoir si l'ajout d'une communication entre le débogueur d'Eclipse et le plug-in *VERSO* permet d'apporter une aide à la compréhension d'un logiciel, mais également à la correction des bogues plus rapidement. Les tâches qui ont été demandées ont pour but de simuler des cas réels d'utilisation. C'est pourquoi les sujets - qui ne connaissent pas l'outil - doivent avoir une connaissance approfondie des langages de programmation, en particulier Java, ainsi que de l'environnement de développement intégré Eclipse.

5.1.1 Les sujets

Les personnes ayant accepté de participer à cette expérimentation et qui sont au nombre de huit, sont des étudiants en dernière année de Master en Sciences Informatiques aux Facultés Universitaires Notre-Dame de la Paix de Namur. De ce fait, ils ont les compétences

nécessaires pour réaliser la tâche qui leur est demandée car ils ont normalement un bon niveau de codage, surtout dans le langage de programmation Java. Ils ont dû corriger un bogue dans un programme qu'ils ne connaissaient pas. Étant donné que nous faisons une évaluation quantitative, nous avons distingué deux groupes lors de ce test :

- les personnes utilisant l'outil (quatre personnes),
- ceux ne l'utilisant pas (quatre personnes).

Les huit sujets ont été divisés en deux groupes, l'un utilisant notre plug-in et l'autre pas, ce qui nous a permis d'évaluer notre outil. Bien que les participants étaient répertoriés dans un groupe, chacun devait accomplir cette tâche de manière individuelle. Il faut noter que les personnes ont collaboré de leur plein gré, et que la participation à cette expérience n'était pas obligatoire.

Il s'est avéré que certains sujets n'ont jamais utilisé de débogueur. Cela n'a pas posé de problème dans le sens où nous essayons de rendre son utilisation intuitive via *VersoDebug*. Si le sujet fait partie du groupe qui ne va pas utiliser l'outil, il est libre d'employer toutes les techniques de débogage qu'il connaît pour accomplir la tâche.

5.1.2 Le temps

L'expérience avait une durée totale d'une heure, avec une période d'adaptation à l'environnement de *VersoDebug* d'environ quinze minutes. Du temps supplémentaire était également accordé aux personnes désirant apprendre le fonctionnement d'un débogueur pour utiliser au mieux ses capacités. Cette durée, ainsi que le temps pris pour répondre aux diverses questions, ne sont pas pris en compte pour l'expérimentation. Par contre, l'accomplissement de la tâche principale qui devait durer jusqu'à trente minutes, était chronométrée dans le but d'établir des résultats de rapidité pour la correction du bogue dans le programme.

5.1.3 L'environnement de travail

Pour effectuer les tâches à accomplir, les participants disposaient de leur ordinateur personnel. Aucune connexion internet n'était exigée. Chaque machine devait être équipée d'un système d'exploitation Windows, d'une carte graphique dédiée gérant les librairies OpenGL (carte AMD ou nVidia), d'une machine virtuelle Java (JVM) utilisant un adressage en 32 bits et d'une version de l'environnement de développement intégré Eclipse. Étant donné que l'outil utilise des librairies OpenGL, il fallait que la JVM gère également cette spécification graphique en y insérant les fichiers `dll` dans le dossier lib de la `Java Runtime Environment`. L'installation de *VersoDebug* et son bon fonctionnement ne peut se faire qu'en respectant les contraintes qui viennent d'être citées. Les personnes témoins ont pu accomplir l'expérimentation sous un autre système d'exploitation comme Linux ou MacOS X avec une JVM adressée en 32 ou 64 bits.

5.1.4 Les questions

Chaque participant a dû compléter un questionnaire avant de commencer l'expérience. Les questions posées ont pour but de vérifier les capacités de l'utilisateur ainsi que ses antécédents, ceci pour ne pas prendre en compte les résultats d'une personne n'étant pas apte à accomplir la tâche correctement. Par exemple, quelqu'un n'étant pas capable de déboguer un programme par ses propres moyens ne pouvait être pris en considération lors de la mise en commun des résultats. Ensuite, deux autres questionnaires leur ont été remis, l'un dans le but de rassembler les efforts fournis par les utilisateurs pour résoudre la tâche, et le dernier pour récolter les commentaires des utilisateurs sur l'expérimentation en général. Ce dernier questionnaire variait selon le groupe dans lequel l'utilisateur se trouvait.

La figure 5.1 reprend les différentes questions auxquelles l'utilisateur a répondu avant de commencer l'expérience.

1. Avez-vous déjà utilisé un débogueur ? Si oui, le(s)quel(s) et dans quel environnement ?
2. Utilisez-vous le débogueur lorsque vous programmez et pourquoi ?
3. Si la réponse à la question 2 est positive, voudriez-vous ajouter une fonction au débogueur ? Si oui laquelle ? Sinon pourquoi ?
4. Si la réponse à la question 2 est négative, qu'est-ce que vous utilisez pour corriger vos programmes ?
5. Si la réponse à la question 2 est négative, qu'est-ce qui vous amènerait à l'utiliser volontairement ?
6. Avez-vous déjà utilisé un logiciel de visualisation ? Si oui lequel et qu'en pensez-vous ?

FIGURE 5.1 – Questions préliminaires

Chaque participant devait alors corriger un bogue sémantique dans un programme qu'il ne connaissait pas. Nous leur avons dit d'exécuter le logiciel et d'effectuer diverses fonctions pour qu'ils remarquent un mauvais comportement du système et ainsi commencer la correction de l'erreur. Lors de cette correction, ils pouvaient très facilement tester l'efficacité de leur modification dans le code source du projet. Le programme test était le logiciel `JHotDraw`, éditeur de dessin «open source» écrit dans le langage de programmation Java. L'anomalie se situait lors d'une sélection ou d'un mouvement d'une forme géométrique dans l'environnement de dessin. La partie de code où a été inséré le bogue avec le code original se trouve en annexe A.

Pour cette partie de l'expérimentation, nous avons posé deux hypothèses que nous allons vérifier suite aux résultats que nous avons obtenus :

- **H0** : Le temps mis pour trouver et corriger un bogue avec *VersoDebug* est égal ou inférieur à l'utilisation d'outils annexes.
- **H1** : *VersoDebug* réduit l'effort pour la découverte et correction de bogues.

Après l'expérimentation, chaque sujet a dû compléter un questionnaire basé sur la charge de travail qu'ils ont dû fournir pour accomplir la tâche qui leur a été demandée. Cette série de questions est basée sur celles de Nasa-TLX [CCPE09] et permet de calculer la charge mentale d'un travail à effectuer. Nous avons ajouté une troisième hypothèse qui est :

– **H2** : La charge mentale de la tâche à fournir est moins importante avec notre outil.

À chacune de ces questions, nous devions ajouter une pondération en fonction des hypothèses que nous cherchons à valider.

La liste de ces questions se trouve ci-après.

1. Quels niveaux de demande mentale et d'activité perceptive étaient nécessaires (par exemple : penser, décider, calcul, mémoire, regarder, chercher, etc) ?
   Les tâches étaient-elles faciles ou exigeantes, simples ou complexes ?
2. Quel niveau d'activité physique était nécessaire ? Par exemple : pousser, tirer, tourner, etc ?
   Le tâche physique était-elle facile ou exigeante, lente ou rapide, reposante ou laborieuse ?
3. Comment était le niveau de pression temporelle que vous avez ressentie en raison de la vitesse ou allure à laquelle les éléments des tâches se sont produits ?
4. Était-ce difficile (mentalement et physiquement) d'accomplir votre niveau de performance ?
5. Comment pensez-vous avoir réussi dans l'accomplissement des objectifs de la tâche fixée par l'analyste (ou vous-même) ?
   À quel niveau êtes-vous satisfait de votre performance dans l'accomplissement de ces objectifs ?
6. Comment était votre niveau d'insécurité, d'irritation, de découragement, de stress et de contrariété lors de la tâche ?

FIGURE 5.2 – Questionnaire concernant l'effort fourni pour accomplir une tâche

Pour notre expérimentation, nous avons choisi de pondérer chaque question dans l'ordre suivant :

– **Demande mentale :** 2 points
– **Demande physique :** 0 point
– **Demande temporelle :** 3 points
– **Effort fourni :** 2 points
– **Performance :** 5 points
– **Niveau de frustration :** 1 point

Dans le cadre d'une expérimentation en informatique, la demande physique d'un utilisateur ne peut être prise en compte car aucun effort musculaire n'est demandé pour réaliser la tâche. Chacune de ces questions est reportée sur une échelle (cfr. figure 5.3) et l'utilisateur devait choisir où il se situait. Chaque point équivaut à une valeur allant de l'ordre de 1 à 20.

Faible									Moyen									Haut	
O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O

FIGURE 5.3 – Échelle de mesure pour le questionnaire de la Nasa-TLX

Lorsqu'un participant a choisi toutes les valeurs, nous avons calculé la charge mentale qu'il a ressentie pendant la réalisation de la tâche.

Considérons N_i la question i posée dans le test Nasa-TLX et P_i sa pondération. Pour calculer la charge mentale pour un travail donné `MW`, il faut effectuer le calcul suivant : $\texttt{MW} = \sum_{i=1}^{6} N_i * P_i$. Plus le résultat est grand, plus il est mauvais et inversement.

Il faut souligner que pour la valeur de la performance, il faut inverser le résultat donné par l'utilisateur. Par exemple, s'il entre la valeur 16, nous devons prendre en compte 4. Ceci est dû au fait que s'il prétend avoir effectué de bonnes performances, la charge mentale doit diminuer et inversement.

Ensuite, chaque participant a dû compléter un questionnaire en fonction du groupe dans lequel il a été placé. Chaque questionnaire a pour but de récolter leurs commentaires relatifs à l'expérience qu'ils viennent d'effectuer. La figure 5.5 concerne les questions après l'expérimentation permettant de récolter les commentaires des utilisateurs quant à l'utilisation de *VersoDebug*. Pour les témoins, ils ont répondu aux questions présentées à la figure 5.4.

1. Est-ce que les outils que vous avez utilisés vous ont permis de trouver facilement le bogue et pourquoi ?
2. D'après vous, que faudrait-il ajouter à *Eclipse* pour qu'il aide les développeurs lors de la maintenance et le développement de logiciels ?
3. Suite à cette expérimentation, pensez-vous utiliser le débogueur lors du développement d'un autre projet ?
4. Avez-vous d'autres commentaires à faire ?

FIGURE 5.4 – Questionnaire après l'expérimentation pour les témoins

1. Est-ce que l'utilisation de l'outil était intuitif et pourquoi ?
2. Est-ce que le contexte visuel vous a permis de détecter le bogue rapidement ? Si oui, comment ?
3. D'après vous, que faudrait-il ajouter à *VersoDebug* pour qu'il aide les développeurs lors de la maintenance et le développement de logiciels ?
4. Suite à cette expérimentation, pensez-vous utiliser le débogueur lors du développement d'un autre projet ?
5. Avez-vous d'autres commentaires à faire ?

FIGURE 5.5 – Questionnaire après l'expérimentation pour ceux ayant utilisé *VersoDebug*

5.2 Résultats et analyses

Après l'expérimentation, nous avons récolté les résultats et les réponses aux questions que nous résumons et analysons dans cette section. Nous allons d'abord montrer les réponses concernant les questions préliminaires. Après, nous comparerons les résultats des deux groupes.

5.2.1 Questionnaire préliminaire

Avant de commencer l'expérimentation, tous les participants ont répondu aux questions préliminaires pour que nous puissions évaluer leurs antécédents. Parmi les volontaires, seulement deux ne connaissaient pas d'outil de débogage car ils n'avaient jamais pris le temps de l'apprendre. C'est pourquoi, pour corriger ou modifier des logiciels lors de la maintenance, ils utilisent des `System.out.println` ou des `loggers` pour afficher l'état de certaines variables lors de l'exécution de processus. Mais ils avouent qu'ils seraient prêts à l'utiliser si nous ajoutions une composante graphique intuitive au débogueur pour l'un, ou tout simplement nous améliorions le rendement du développement d'un projet pour l'autre.

Pour cette raison, nous avons décidé de ne pas mettre ces deux personnes dans le même groupe pour comparer leurs résultats respectifs et ainsi vérifier si notre outil s'avérait intuitif dès sa première utilisation.

Pour les autres, tous ont utilisé au moins une fois un débogueur, généralement celui fourni dans Eclipse pour le langage de programmation Java, sauf un sujet qui a utilisé ce type d'outil pour le langage Flex, ce qui ne pose aucun problème vu que le fonctionnement de ce type d'outil est le même quel que soit le langage employé.

Parmi ces personnes, quatre d'entre elles prétendent l'utiliser de façon générale ou occasionnelle et ce pour diverses raisons :

– détecter des erreurs de programmation lors de l'exécution du programme,
– corriger des fonctionnalités posant certains problèmes,

– analyser l'état de la mémoire allouée au processus.

Au niveau des fonctionnalités qu'ils se verraient ajouter à l'outil de débogage, un sujet se dit satisfait car il contient déjà toutes les informations nécessaires à la correction des erreurs dans un programme. Pour les trois autres, ils ont cité les ajouts suivants :

– la possibilité de voir (au moins) l'état d'un programme avant que l'erreur ne se produise,
– une fonction qui indique directement la ligne de code qui peut poser problème,
– une correction automatique du bogue découvert.

Étant donné que ces participants utilisent le débogueur lors du développement de leur projet, nous en avons placé deux dans le groupe témoin et les deux autres dans le groupe qui s'est servi de *VersoDebug*. Ces derniers n'emploient pas l'outil de débogage car ils prétendent perdre du temps à cause de sa lourdeur d'utilisation, ou bien parce qu'ils n'ont pas pris cette habitude lors de leur développement de logiciels.

Comme pour les deux sujets n'ayant jamais utilisé de débogueur, ils se servent des mêmes techniques pour corriger leurs erreurs de programmation, c'est-à-dire les affichages des valeurs des variables dans la console de développement à l'aide des `loggers` ou de la fonction `System.out.println`. En ce qui concerne l'utilisation volontaire de l'outil, l'un compte peut-être l'employer lorsqu'il faut évaluer les états de processus en cours d'exécution, tandis que l'autre voudrait une preuve de son efficacité.

Par rapport à la connaissance et l'utilisation d'un outil de visualisation, tous les sujets prétendent avoir déjà utilisé CodeCity, présenté dans la section 2.2.1, sauf deux personnes qui ne connaissent pas de logiciel de visualisation. Tous les autres ont un avis plus ou moins mitigé sur ses avantages ou ne comprennent pas son utilité dans le monde du génie logiciel. Certains trouvent qu'il est pratique pour représenter de gros projets et ainsi comprendre les dépendances qui peut exister entre les classes, tandis que d'autres cherchent encore son véritable intérêt ainsi que sa valeur ajoutée pour mesurer la qualité d'un logiciel.

À la vue de ces résultats préliminaires, nous avons pu déduire que tous les participants ont été aptes à effectuer notre tâche principale, même si certains ont eu besoin d'un temps d'adaptation. Nous avons donc pu prendre en compte tous les résultats des participants lors de l'expérimentation qui a suivi, ce qui nous a permis de valider l'expérience de chaque sujet.

L'analyse des réponses au questionnaire préliminaire étant terminée, nous allons étudier les résultats de la tâche principale qui consistait à corriger un bogue sémantique dans le programme «open source» `JHotDraw`.

5.2.2 Correction du bogue

Nous résumons dans cette section, les résultats de la tâche principale. Dans un premier temps, nous présentons les taux de réussite ou d'échec pour chaque sujet ayant participé à

l'expérimentation. Pour cela, nous allons retenir trois valeurs ordinales pour déterminer de la réussite ou de l'échec de l'exercice :

- *KO* pour les personnes n'ayant pas réussi à trouver le bogue,
- *OK* pour les personnes ayant corrigé le bogue,
- *Bien* pour les personnes ayant corrigé en partie le bogue ou celles qui étaient sur la bonne voie.

Ensuite, nous allons analyser les résultats de chacun en faisant des tests de statistiques pour pouvoir tirer les premières conclusions sur l'utilisation de notre outil. Le tableau 5.1 recense les temps affichés en minutes, la réussite de la tâche à accomplir lors de l'expérimentation, le montant de la charge mentale obtenue en fonction des résultats indiqués par chaque participant avec l'utilisation du plug-in *VersoDebug* ou non.

Sujets	1	2	3	4	5	6	7	8
Temps	>30	10	25	>30	>30	>30	>30	>30
Réussite	Bien	OK	OK	Bien	Bien	KO	Bien	KO
MW	45.5	24	45	56	94	124.5	86.5	103.5
Utilisation de *VersoDebug*	Non	Non	Non	Non	Oui	Oui	Oui	Oui

TABLE 5.1 – Tableau recensant le temps et la réussite de la tâche

En jetant un premier coup d'œil à ces résultats, nous pouvons dire qu'ils sont négatifs. En effet, nous remarquons que parmi tous les participants, seulement deux ont réussi l'expérimentation dans les temps et deux n'ont pas réussi à trouver et corriger le bogue. Pour les autres, nous avons dû arrêter l'expérimentation car ils ont mis plus de 30 minutes pour le trouver et le corriger. C'est pourquoi, en regardant de plus près les durées de chacun, nous nous apercevons que l'utilisation ou non de *VersoDebug* n'influence pas la rapidité du travail. De plus, les deux personnes qui n'ont pas pu réussir la tâche qui leur a été demandée ont utilisé notre plug-in. Pour valider ces propos, nous avons utilisé le test de *Mann-Withney*[1] pour valider les résultats concernant le temps et la charge mentale. Au niveau de la performance, nous avons employé le test *ANOVA*[2]. Chaque test a été effectué sur les deux groupes de personnes.

Test du temps Nous allons tester les deux groupes de personnes pour savoir si le temps moyen du groupe témoin est égal ou supérieur au temps moyen du groupe de personnes ayant utilisé *VersoDebug*. En calculant la moyenne des temps de chaque groupe, nous obtenons 29,5 minutes pour le groupe témoin, et 52,25 minutes pour le second groupe. En effectuant le test de *Mann-Withney* sur les données recueillies, le résultat de la P-VALUE est égal à 0.04331.

Étant donné que le résultat est inférieur à 0.05, nous pouvons conclure que *VersoDebug* n'améliore pas le temps pour trouver et corriger un bogue. L'hypothèse **H0** n'a donc pas pu

1. Test décrit dans [WRH+00] à la page 100
2. Test décrit dans [WRH+00] à la page 105

être démontrée.

Test de la charge mentale Comme pour le test sur le temps, nous avons calculé la moyenne des résultats obtenus des deux groupes. Pour rappel, plus les résultats sont hauts, plus un groupe a dû réfléchir plus longuement pour effectuer la tâche. Les personnes n'ayant pas utilisé notre outil ont une moyenne de 42.625 de charge mentale tandis que les autres ont 102.125. Nous constatons une grande différence entre les deux groupes, ce qui en résulte une P-VALUE égale à 0.02092 en ayant effectué le test de *Mann-Withney*.

Nous pouvons très facilement en déduire que notre outil n'est pas intuitif à la première utilisation et donc que l'hypothèse **H1** ne peut pas non plus être validée.

Test de la performance Pour le test sur la performance, nous avons attribué une valeur en fonction de la réussite de l'exercice ou non. Un point pour les personnes ayant trouvé et corrigé le bogue, un demi-point pour ceux étant sur la bonne voie, et zéro pour ceux qui ne l'ont pas trouvé. Avec ces points, les moyennes calculées suivant les groupes sont de 0.75 pour le groupe témoin et 0.25 pour le groupe ayant employé *VersoDebug*. Pour déduire les résultats, nous utilisons le test *ANOVA*, ce qui nous donne comme P-VALUE 0.0498.

L'écart étant moins important par rapport aux tests précédents, nous déduisons que la charge mentale avec ou sans l'outil est équivalente. L'hypothèse **H2** ne peut être démontrée.

Ceci nous permet de tirer une première conclusion : l'utilisation de *VersoDebug* n'aide pas les utilisateurs à trouver et corriger plus rapidement un bogue que le débogueur en lui-même. De plus, notre outil n'influence pas la charge mentale pour résoudre ce type de problème. Les trois hypothèses **H0**, **H1** et **H3** n'ont donc pas pu être prouvées.

5.2.3 Questionnaire final

En dernier lieu, nous avons récolté les commentaires des participants après l'expérimentation suivant le groupe auquel ils appartenaient.

Groupe ayant utilisé leurs propres outils

Les personnes témoins qui, pour rappel, ont employé leurs propres outils pour trouver et corriger le bogue, ont dû répondre aux questions présentées à la figure 5.4.

Comme mentionné dans le tableau 5.1 qui montre les résultats, seuls les sujets 2 et 3 ont réussi à trouver le bogue dans les temps. Ils ont tous les deux analysé la `stacktrace` pour parcourir le chemin d'exécution via la vue *Debug*, et de plus, grâce à divers outils proposés par Eclipse, il était également possible de remonter aux endroits où cela pouvait poser problème. Les deux autres personnes étaient, malgré le temps écoulé, presque arrivées à trouver le bogue.

Elles ont employé la même technique que les deux personnes qui ont trouvé et corrigé le bogue, mais d'une manière moins rapide.

Suite à l'expérimentation, certains sujets ont proposé de rajouter quelques fonctionnalités qui pourraient aider plus facilement les développeurs à trouver et corriger des erreurs de programmation. Parmi les ajouts présentés, nous avons :

- avoir le chemin d'exécution dans les méthodes avec les différents tests effectués (avec les tests de conditions `if-then-else` et les différents types de boucles `for-while`),
- pouvoir revenir à l'état stable du programme avant que l'erreur ne survienne.

Enfin, parmi les personnes témoins, toutes ont décidé d'utiliser le débogueur lors du développement d'un autre projet informatique car ils ont remarqué qu'il était d'une grande aide, malgré ses lacunes en ce qui concerne la quantité importante de mémoire qu'il doit utiliser pour fonctionner correctement. Sinon, ils préconisent tout de même d'effectuer une formation pour les développeurs débutants pour leur permettre de découvrir cet outil et de l'utiliser de manière efficace.

Groupe ayant utilisé *VersoDebug*

Les personnes ayant employé notre outil pour réaliser la tâche qui leur a été demandée ont répondu aux questions présentées à la figure 5.5.

À savoir si *VersoDebug* était intuitif d'utilisation, trois sujets sur quatre ont répondu de manière négative à la question car :

- la documentation qui a été fournie n'était pas assez explicite,
- l'interface manquait de fluidité,
- la spécification pour utiliser les points d'arrêt était absente.

Le dernier sujet trouve au contraire qu'il est très facile d'avoir des informations sur les classes et méthodes durant le débogage.

En ce qui concerne l'aide pour trouver le bogue, là encore les sujets sont unanimes. Après une petite utilisation du contexte visuel, ils passent directement aux anciennes méthodes qu'ils utilisent habituellement et laissent de coté *VersoDebug*. La principale raison est que le plug-in n'est pas assez présent pour le débogage, qu'il manque de fluidité lors de l'interaction avec les fonctions `stepping` ou tout simplement parce que la `stacktrace` est plus parlante pour certains. Le débogueur et *VersoDebug* utilisent tous les deux de grandes quantités de ressources système pour leur traitement. C'est une des raisons pour laquelle l'environnement de développement intégré Eclipse a arrêté de fonctionner deux fois de suite chez le sujet 8.

C'est pourquoi les utilisateurs proposent diverses améliorations pour une meilleure aide au débogage. Ces idées sont les suivantes :

- une plus grande présence dans le débogage d'un programme sous Eclipse,
- améliorer la fluidité de *VersoDebug* lorsque l'utilisateur cherche à corriger un problème dans un logiciel,

- une meilleure documentation de son utilisation pour qu'il soit plus intuitif,
- une utilisation indépendante de *VersoDebug* sans devoir utiliser la vue *Debug* pour travailler.

Suite à cette expérimentation, 50% des participants ayant employé notre outil sont prêts à commencer à utiliser le débogueur dans des projets futurs à condition d'avoir un certain confort d'utilisation lors du débogage de programmes et plus de fluidité. Mais les résultats et les commentaires relatifs à l'emploi de *VersoDebug* pour trouver et corriger un bogue sont moins probants.

5.2.4 Commentaires de l'expérimentation

Bien que nous ayons validé les résultats par des tests statistiques, nous devons les prendre avec quelques réserves. En effet, étant donné le nombre de sujets ayant participé à notre expérimentation, les analyses de ces résultats ne sont pas concluants. En fait, nous aurions dû prendre un plus grand échantillonnage pour les valider, mais la disponibilité des personnes ne le permettait pas. Enfin, nous aurions dû proposer une séance d'information aux participants pour apprendre à utiliser l'outil correctement, tout en leur accordant un peu plus de temps pour résoudre la tâche principale.

De ce fait, nous ne pouvons valider entièrement notre expérimentation à travers cette évaluation pour les raisons que nous venons de citer. Mais cette expérience a permis de mettre en avant certains défauts de notre outil qu'il faudra corriger à l'avenir.

Le prochain et dernier chapitre de ce mémoire traite des améliorations futures que nous pouvons apporter à *VersoDebug* et de la conclusion.

Chapitre 6

Conclusion

Dans ce mémoire, nous avons présenté un nouvel outil qui doit apporter une aide aux développeurs lors de la conception et la maintenance d'un logiciel. Dans ce chapitre, nous présentons la conclusion de ce document en énonçant ce qui a été fait avant la proposition de l'outil, la contribution de l'auteur par rapport à l'existant, et nous terminerons par les améliorations et travaux futurs.

6.1 Réalisations de l'existant

Le but est de permettre aux programmeurs d'utiliser le débogueur aux dépens des techniques trop souvent utilisées comme les `System.out.println`, ou les `loggers` pour afficher les valeurs des variables sur la console de développement. De plus, les personnes qui analysent les systèmes informatiques en utilisant des outils de visualisation, souhaitent pouvoir étudier l'état d'un programme à un instant bien précis quel que soit le scénario voulu. C'est pourquoi nous avons proposé un outil de visualisation qui communique avec le débogueur d'Eclipse pour répondre à la demande des analystes et rendre l'outil de débogage plus intuitif en lui fournissant une composante visuelle.

Avant d'aborder l'approche de développement de notre outil *VersoDebug*, nous avons étudié le domaine de l'existant en ce qui concerne la modification de la présentation du code source en premier lieu, et les outils de visualisation en second lieu. Dans le premier cas, nous nous sommes intéressés aux améliorations que pouvait apporter un tel changement pour les développeurs et ainsi en tirer un maximum d'avantages. Ceci a été longuement traité dans la toute première version de *VERSO* qui, pour rappel, a pour but de présenter le code source d'un projet de façon visuelle et d'en calculer diverses métriques pour aider les développeurs dans la maintenance et la conception de logiciels.

En second lieu, nous avons étudié divers outils de visualisation qui existaient dans le domaine de l'analyse dans le génie logiciel. Nous avons remarqué que le concept de ville a

été fortement utilisé par de nombreux outils comme CodeCity, *VERSO* ou encore Evospace. Ce dernier a d'ailleurs apporté une composante intéressante pour étudier le comportement de systèmes informatiques en analysant des traces d'exécution. En effet il permet l'affichage du chemin de l'exécution d'un processus en cours de débogage en introduisant le concept de jour/nuit dans le contexte visuel. Mais les outils de visualisation ne s'arrêtent pas là. *VASCO*, par exemple, utilise également des traces d'exécution, mais pour repérer l'utilisation excessive d'objets temporaires et pouvoir les supprimer. Nous pouvons ainsi optimiser le logiciel au niveau des ressources que le système peut lui allouer.

6.2 Contributions du mémoire

La différence que nous avons apportée par rapport aux différents outils qui analysent des traces d'exécution, est que nous avons utilisé le débogueur comme moyen de compréhension d'un système informatique. Cet outil permet d'exécuter un programme de manière interactive en progressant pas à pas suivant les lignes de code du projet. Ceci nous permet d'étudier un logiciel durant son exécution et non plus via des scénarios enregistrés qui limitent l'étude de son comportement. Pour rendre le débogueur plus attractif d'utilisation, nous avons proposé de lui ajouter une communication avec la version plug-in de *VERSO* qui s'intègre à Eclipse. Cela nous a permis de rendre le développement de notre outil beaucoup plus facile.

Au niveau de l'implémentation de *VersoDebug*, plusieurs choix ont été proposés. Après plusieurs analyses, nous avons gardé celui qui nous paraissait le plus adapté, c'est-à-dire la traduction de la `stackframe` du processus en cours de débogage dans le contexte visuel. Pour afficher le chemin d'exécution, nous avons opté pour un dégradé de couleurs allant du blanc (qui indique le bas de la pile) vers le rouge (le haut de la pile). Pour les gros projets avoisinant les 350 classes, nous avons voulu accentuer le focus sur celles qui participent au débogage. C'est pourquoi, en plus de la couleur, nous avons ajouté les dimensions comme autre caractéristique graphique et nous les avons augmentées. Pour celles ne faisant pas partie de la pile d'exécution et étant voisines d'une classe qui est dans la `stackframe`, nous avons diminué leur taille. Ensuite, de nombreux programmeurs affichent la valeur des variables sur la console de développement. Pour répondre à leurs besoins, nous avons décidé d'afficher ces informations dans un `tooltip` reprenant le nom et la valeur de la variable en temps réel.

En dernier lieu, nous avons évalué notre outil en faisant une expérimentation *in vitro*. Des étudiants de l'université de Namur y ont participé de leur plein gré pour vérifier les hypothèses que nous avons posées. Elles sont :

– **H0** : Le temps mis pour trouver et corriger un bogue avec *VersoDebug* est égal ou inférieur à l'utilisation d'outils annexes.

– **H1** : *VersoDebug* réduit l'effort pour la découverte et correction de bogues.

– **H2** : La charge mentale de la tâche à accomplir est moins importante avec notre outil.

L'évaluation consistait à corriger un bogue sémantique dans un programme inconnu des participants. Nous avons ensuite analysé les résultats retenus en effectuant les tests de *Mann-Withney* pour les hypothèses **H0** et **H2**, et celui d'*ANOVA* pour **H1**. En analysant les solutions, il s'est avéré que le contexte visuel du débogueur n'a pas significativement aidé à la découverte du bogue dans l'état actuel des choses. Certes, il affiche la pile d'exécution sous une forme graphique, mais il s'est avéré inutile dans la recherche du problème. Malheureusement, nous ne pouvons valider ces résultats pour la simple raison que l'échantillon que nous avons eu était très petit et que les participants ne connaissaient pas l'outil à utiliser. C'est pourquoi les trois hypothèses que nous avons posées n'ont pas pu être vérifiées.

6.3 Travaux futurs

Enfin, nous terminons par les critiques et les améliorations qui peuvent être apportées. Grâce aux commentaires récoltés pendant l'expérimentation, nous avons des pistes de développement futur.

Tout d'abord, il faudrait que notre outil soit plus fluide avec les interactions du débogueur. En effet, l'ajout d'un contexte visuel n'a pas du tout aidé les utilisateurs. La fluidité est apparue être une cause de la non utilisation de l'outil. C'est en réglant ce problème que nous pourrions avancer. Ensuite, il faudrait que *VersoDebug* ait son environnement propre. Le fait de devoir interagir avec la vue *Debug* et le contexte visuel n'a pas non plus aidé à la navigation durant l'expérimentation.

Au niveau des améliorations futures que nous pouvons apporter, nous proposons une interaction complète de l'outil avec l'utilisateur que ce soit lors de l'exécution du programme ou lors de son débogage. Imaginons qu'un logiciel tourne et qu'une exception intervient. Le contexte visuel met en évidence, via une couleur attirante, la classe d'où provient l'erreur. En fonction de l'exception qui est intervenue, nous pourrions remonter jusqu'à la source de l'erreur. Par exemple, si un `NullPointerException` intervient sur une variable, cela veut dire qu'elle n'a pas été initialisée. Dans ce but, le contexte visuel pourrait mettre en couleur toutes les classes ou méthodes où une initialisation de la variable est appelée dans la `stackframe` où l'erreur s'est produite.

En conclusion, l'outil que nous proposons, dans l'état actuel des choses et dans la version que nous avons testée, n'aide pas les développeurs à trouver et corriger des bogues dans des programmes qu'ils ne connaissent pas. Par contre, il peut servir de base de travail pour de nouvelles fonctionnalités qui seraient en mesure de fournir l'aide souhaitée.

Bibliographie

[api] IDebugEventSetListener API. http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fdebug%2Fcore%2FIDebugEventSetListener.html.

[BRZ+10] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. Code bubbles : Rethinking the user interface paradigmof integrated development environments. In *ICSE '10 : Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, May 2010. http://www.cs.brown.edu/people/acb/codebubbles_site.htm.

[BZR+10] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. Code bubbles : a working set-based interface for code understanding and maintenance. In *CHI '10 : Proceedings of the 28th international conference on Human factors in computing systems*. ACM, 2010.

[CCPE09] Alex Cao, Keshav K Chintamani, Abhilash K Pandya, and R Darin Ellis. Nasa tlx : software for assessing subjective mental workload. *Behavior Research Methods*, 41(1) :113–7, 2009. http://www.ncbi.nlm.nih.gov/pubmed/19182130.

[chaa] Champ visuel humain. http://fr.wikipedia.org/wiki/Champ_visuel.

[chab] Champ visuel poisson. http://www.pechemouche.be/fariofr.htm.

[DA08] Philippe Dugerdil and Sazzadul Alam. Execution trace visualization in a 3d space. In *Proceedings of the Fifth International Conference on Information Technology : New Generations*, ITNG '08, Washington, DC, USA, 2008. IEEE Computer Society.

[DOU07] Jean-Michel DOUDOUX. Déboguer du code java. In *Développons en Java avec Eclipse*. 2007. http://www.jmdoudoux.fr/java/dejae/chap008.htm#chap_8.

[DR10] Robert DeLine and Kael Rowan. Code canvas :zooming towards better development environments. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. ACM, 2010.

[Dus11] Fleur Duseau. Technique de visualisation pour l'identification de l'usage excessif d'objets temporaires dans les traces d'exécution. Master's thesis, Université de Montréal, December 2011.

[Hou] Sahraoui Houari. Principes de visualisation du logiciel.

[JS] Brian Johnson and Ben Shneiderman. Treempas :a space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the 2nd conference on Visualization '91*, VIS '91, pages 284–291. IEEE Computer Society Press.

[Lan10] Guillaume Langelier. *Intégration de la visualisation à multiples vues pour le développement du logiciel*. PhD thesis, December 2010.

[LD07] Guillaume Langelier and Karim Dhambri. Visualization-based analysis of quality for large-scale software systems. In *Visualizing Software for Understanding and Analysis*, VISSOFT '07. IEEE Computer Society, June 2007.

[LSP05] Guillaume Langelier, Houari Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05. ACM, 2005.

[plua] *Tutorial : développer des plugins Eclipse (1ère partie).* `http://www.eclipsetotale.com/articles/Developpement_de_plugins_Eclipse_partie1.html`.

[plub] *Tutorial : développer des plugins Eclipse (2ème partie).* `http://www.eclipsetotale.com/articles/Developpement_de_plugins_Eclipse_partie2.html`.

[rcp] *Tutorial : développement d'applications Eclipse RCP (1ère partie).* `http://www.eclipsetotale.com/articles/Developpement_applications_Eclipse_RCP_partie1.html`.

[Sto06] Maureen Stone. Choosing color for data vizualisation. *Business Intelligence Network*, January 2006. `www.perceptualedge.com`.

[WFB04] Darin Wright and Bjorn Freeman-Benson. *How to write an Eclipse debugger*, August 2004. `http://www.eclipse.org/articles/Article-Debugger/how-to.html`.

[WL08] Richard Wettel and Michele Lanza. Codecity. In *WASDeTT*, 2008.

[WLR11] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities : a controlled experiment. In *ICSE*, pages 551–560, 2011.

[WRH+00] C. Wohlin, P. Runeson, M. Host, C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering : an Introduction.* Kluver Academic Publishers, 2000.

[Yak11] Wassim Yakoub. Visualisation dynamique du logiciel. Technical report, Université de Montréal, Département d'Informatique et de Recherche Opérationnelle, August 2011.

Annexe A

Code source de l'expérimentation

```
public class SelectionTool extends AbstractTool
        implements ToolListener {

    protected void setTracker(Tool newTracker) {
        if (tracker != null) {
            tracker.deactivate(getEditor());
            tracker.removeToolListener(this);
        }
        tracker = newTracker;
        if (tracker == null) {
            tracker.activate(getEditor());
            tracker.addToolListener(this);
        }
    }
}
```

Listing A.1 – Partie du code à corriger

```
public class SelectionTool extends AbstractTool
        implements ToolListener {

    protected void setTracker(Tool newTracker) {
        if (tracker != null) {
            tracker.deactivate(getEditor());
            tracker.removeToolListener(this);
        }
        tracker = newTracker;
```

```
        if (tracker != null) {
            tracker.activate(getEditor());
            tracker.addToolListener(this);
        }
    }
}
```

Listing A.2 – Partie du code original